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. 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 the 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 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 interactive theorem prover. The assumptions are outlined in Section II and the abstract algorithm is presented in Section III. 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 IV. 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. Related work appears in Section V and conclusions in Section VI.

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
and returns a single formula that is equivalent to \( c \) under assumptions.

We assume the existence of an Interactive Theorem Prover (ITP) that 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 well-formed 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 if the original goal is valid. Modern interactive theorem provers use various techniques for this, including decision procedures for Boolean logic, case analysis, evaluation, linear arithmetic, congruence closure, and rewriting.

SIMPLIFY takes as input an \( L \)-formula, \( c \), and a list of assumptions, \( H \). SIMPLIFY simplifies \( c \) assuming \( H \) is true, and returns a single formula that is equivalent to \( c \) under assumptions \( H \).

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, \( hyp_1 \land \cdots \land hyp_n \), 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 analyzes \( P \) by recursively decomposing \( P \) into simpler properties and searching for counterexamples to them.

Analyze first (lines 2-4) tries to repeatedly search for counterexamples until either a user-defined stopping condition is satisfied or limit on the number of search attempts is reached. The limit is a user-defined parameter stored in \( \text{SLIMIT} \). The procedure \( \text{Search} \) (described next) uses a DPLL-like algorithm to incrementally search for falsifying assignments to \( P \). Assignments obtained are checked (if they are indeed complete falsifying assignments) and recorded as counterexamples.

Useful information is tracked in a global data structure \( \text{summary} \). It is used to record counterexamples, successful proofs (if \( P \) was proved by the theorem prover), subgoals that failed to provide either proofs or counterexamples (these subgoals, which correspond to a particular case of the original property can be examined more closely by the user) and other statistics like, the number of unsuccessful search attempts, the number of counterexamples and witnesses found,\(^1\) and the number of subgoals analyzed.

The user-specified stopping condition is a predicate on \( \text{summary} \), for example a typical stopping condition would be: number of counterexamples found should be greater than 3; a more intricate stopping condition would involve some notion of coverage. If the user-specified stopping condition is satisfied a summary of the analysis is printed and the procedure exits. Otherwise the property is semantically decomposed (line 7) using the SMASH procedure of the theorem prover into simpler properties. Each such simpler property is recursively analyzed (lines 9-10). In case the theorem prover is unable to simplify the input property, or it successfully proves the validity of the input property, the appropriate information is recorded and the procedure simply returns, unless the input property is the top-level goal, in which case (lines 11-12), we print the summary and exit.

**Algorithm 1: Analyze**

**Input:** Property \( P \)

1. \( n := 0 \)
2. while \( \neg SCond \land n \leq \text{SLIMIT} \) do
   3. \( A, n := \text{Search}(P), n + 1 \)
   4. update \( \text{summary} \) (record counterexample)
5. if \( SCond \) then
   6. print \( \text{summary} \) and exit
7. \( S := \text{SMASH}(P) \)
8. if \( S \neq \{P\} \land S \neq \{\} \) then
   9. for all \( p \in S \) do
      10. \( \text{Analyze}(p) \)
11. if \( P \) is “goal” then
   12. print \( \text{summary} \) and exit
13. return

---

\(^1\)To simplify the exposition we only show how counterexamples are found, but witnesses can also be found in a similar manner.

FMCAD 2011, Page 47
of assignment to the variable, if it was decided by Assign, string “decision” is stored, else string “implied” is stored.

Algorithm 2 Search

Input: Property $P$

1: local Assignment $A$
2: local Stack $B$ (of (var, # assigns, type of assign))
3: local Stack $S$ (of Property)
4: $A, S :=[ ]$, push($P, S$)
5: $x_0 := \text{Select}(P)$
6: push($(x_0, 0,”na”), B)$
7: while $A$ is not complete do
8: $P := \text{head}(S)$
9: $(x, i, _) := \text{head}(B)$
10: if $P$ has a constraint of form $x = c$ then
11: $v := [[c]]; t := “implied”$
12: else
13: $v, t := \text{Assign}(x, P)$
14: /*Update # and type of assign of $x*/
15: pop($B$);
16: if $x = x_0$ and $i > \text{BLIMIT then}$
17: return fail
18: $P' := \text{Propagate}(x, v, P)$
19: $\text{hyps} := \text{hyps}(P'); \text{concl} := \text{concl}(P')$
20: if false $\in \text{hyps} \land \text{concl} \neq true$ then
21: $A, S := \text{push}(x, v, A), \text{push}(P', S)$
22: $B := \text{push}(\text{Select}(P'), 0,”na”), B)$
23: else /*Incomplete assignment */
24: repeat
25: $S, B, A := \text{pop}(S), \text{pop}(B), \text{pop}(A)$
26: $(x', i', t') := \text{head}(B)$
27: until $(t’ = “decision” \land i’ < \text{BLIMIT}) \lor$
28: size($B$) $\leq 1$
29: if $x’ = x_0$ \&\& $(t’ = “implied” \lor i’ = \text{BLIMIT})$
30: return $A$

Procedure Search first initializes $A$ to be empty and pushes $P$ onto stack $S$. It calls the procedure Select (described next) to choose the first variable $x_0$ to be assigned. $x_0$ is pushed onto stack $B$, its assign counter (number of times the variable is assigned) is initialized to 0 and the string specifying the type of assignment is set to “na” (denoting not assigned).

The main search loop (lines 7-29) implements the iterative construction of $A$. The selected variable $x$ and property $P$ in the current iteration of the search loop are obtained by reading the top of the stacks $B$ and $S$. If $x$ is constrained by an equality $x = c$ where $c$ is a constant expression), then we simply assign $x$ the value $v$ (obtained by evaluating $c$), otherwise, the instantiation is performed by the procedure Assign which returns the value $v$ to be assigned to $x$ and also the type of assignment $t$. The assign counter for $x$ is incremented and the type of assignment is recorded in $B$. We will defer discussing the details of Assign to the next section, for now think of it as an oracle that finds a value $v$ that satisfies simple local constraints involving only $x$.

The procedure Propagate (described later) is used to simplify $P$ using the theorem prover in light of the new assignment to $x$, deducing as much new information as possible, resulting in either a partially concretized property ($P'$) or an inconsistency. Inconsistency is (syntactically) recognized if either false is found in the hypotheses (of $P'$) or the conclusion (of $P'$) is equal to true.

If no inconsistency was found (checked in line 20), the assignment $A$ is extended, the partially concretized property $P'$ is pushed onto $S$ and a free variable from $P''$ is selected and pushed onto $B$ to be instantiated in the next iteration of the main search loop (lines 21-22).

If an inconsistency is found, we backtrack to the last decision (by popping the stacks and undoing the assigns in $A$) that has not exhausted its limit, SLIMIT (lines 24-27). While backtracking to the last decision, we never pop the first variable selection stored in $B$ at the start of the search loop. If assigns to $x_0$ are exhausted, then Search fails (lines 16-17, 28-29), moreover if we backtracked to $x_0$ and it’s type of assign is “implied” then too we return fail. The search is repeated until all free variables have been assigned values (line 7) returning a complete assignment (line 30).

Algorithm 3 Select

Input: $P$ is a property

1: Do congruence closure on $P$
2: $G := \text{buildVariableDependencyGraph}(P)$
3: $\text{dagG} := \text{ComputeSCCs}(G)$
4: $\text{sortedList}_{\text{dagG}} := \text{TopologicalSort}(\text{dagG})$
5: $X := \text{pickLast}(\text{sortedList}_{\text{dagG}})$
6: if $X$ is set (of mutually-dependent variables) then
7: return some vertex in $X$
8: else
9: return $X$

Selecting variables to assign

Select (Algorithm 3) procedure describes the mechanism to choose a variable in a property. It takes as input a property $P$, performs static analysis to determine a certain type of dependency relationship among the variables (described below) of $P$, and selects the variable with the least dependency. We will motivate this notion of dependency in the context of the Search algorithm with an example. In the following $x, y, z, w$ are constrained to be integers and hash is a standard hash function.

$$P : \quad z = y^2 \land y = \text{hash}(x) \land w = \text{hash}(y) \Rightarrow z > w^2$$

Since we are interested in finding counterexamples, we have four constraints to satisfy, the first three are the hypotheses, and the final constraint is the negated conclusion. Lets assume

\footnote{In the concrete algorithm (next section) we randomly sample the variable’s type domain, but in general one could use more heavyweight methods such as constraint-solving.}
there is some procedure available for assigning a value to a variable without falsifying any constraint. Which variable should we (select and) assign first? 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 \( \text{hash}(x) \) to obtain the value of \( y \). Selecting \( y \) before \( x \), causes difficulty in satisfying the constraint \( y = \text{hash}(x) \), since computing the inverse hash function might be non-trivial. Moreover, any constraint solver used in \text{Assign} might not be powerful enough to handle non-linear arithmetic of \text{hash}.

Treating \text{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. We shall call such a relation a \textit{v-dependency} which we define more precisely below. The idea behind the algorithm is to select the variable with the \textit{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 and backtracking. We construct a directed graph with variables as nodes and the directed edges in the graph denote the \textit{depends on} binary relation. The edges are also annotated with the logical relation that caused the edge to be drawn in the first place. We call an edge annotated with relation \( R \) an \( R \)-edge.

The \textit{variable dependency graph} for \( P \) initially consists of only nodes, one for each variable and no edges. The graph is constructed by iterating over the constraints of \( P \) using the following rules, which form the core of the procedure \text{buildVariableDependencyGraph}. We assume \( x \) and \( y \) are (distinct) free variables of \( P \) and \textit{term} is inductively defined to be either a variable, a constant expression, or a function application with arguments that are \textit{terms}.

1) If \( P \) has a constraint of the form \( x = c \), where \( c \) is a constant expression, we force \( x \) to be a leaf node (no outgoing edges). Once a node is marked leaf, it overrides the other rules.

2) If \( P \) has a constraint of the form \( x = \text{term} \) such that \( y \in \text{freeVars}() \text{term} \) and \( x \notin \text{freeVars}() \text{term} \), we add an \( =\)-edge from node \( x \) to node \( y \). \textit{term} is a function application as defined above.

3) If \( P \) has a constraint of the form \( x \gg \text{term} \) such that
\[
\gg \text{is a binary relation, } y \in \text{freeVars}() \text{term} \text{ and } x \notin \text{freeVars}() \text{term}.
\]
we add an \( \gg\)-edge from node \( x \) to node \( y \).

4) If \( P \) has a constraint of the form \( x \bowtie y \) where \( \bowtie \in \{<,\leq,\geq\} \) we don’t draw an edge.

5) If \( P \) has a constraint of the form \( R()\text{term}_1, \text{term}_2, \ldots, \text{term}_n \), such that \( x \in \text{freeVars}()\text{term}_i \), \( y \in \text{freeVars}()\text{term}_j \), \( i \neq j \), \( n \geq 2 \) and \( R \) is an arbitrary \( n \)-ary relation, then we perform the following. Let \( n_{=} \) and \( n_{>\bowtie} \) be the number of incoming edges to a node labeled with \( = \) and \( \bowtie \) respectively. If \( x \) and \( y \) have no incoming or outgoing edges, we add a bidirectional \( R \)-edge between \( x \) and \( y \), else we add a \( R \)-edge between \( x \) and \( y \) pointing to the node (variable) that has a greater \( (n_{=}, n_{>\bowtie}) \) value lexicographically, else we don’t add an edge.

Using the above definition of \textit{v-dependency}, procedure \text{Select} constructs the \textit{variable dependency graph} for \( P \) after applying congruence closure (replace equivalent variables by their representative chosen lexicographically) to \( P \) (lines 1-2). Congruence closure helps simplify the graph since constraints such as \( x = y \) are quite common. After the graph is constructed, using the aforementioned rules, its strongly-connected components are computed (lines 2-3). The resultant directed acyclic graph (dag) obtained is topologically sorted. The algorithm then picks the component (a set of variables) which has no outgoing edges (i.e., has no \textit{dependency} on other components). If the component has just one variable, then usually it is the node which is a leaf (i.e., no dependency), 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 the lexicographically smallest name (lines 4-6). Note that \text{Select} tries to ensure the following rule of thumb: select a variable only when every variable it \textit{depends on} has already been assigned a value; this is not always the case.

**Algorithm 4 Propagate**

<table>
<thead>
<tr>
<th>Input:</th>
<th>Var ( x ), Value ( v ), Property ( P )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>( \text{hyps} := \text{hyps}(P) )</td>
</tr>
<tr>
<td>2.</td>
<td>( \text{hyps}.add(x = v) )</td>
</tr>
<tr>
<td>3.</td>
<td>( \text{shyps} := \text{simplifyAssumingRest}()\text{hyps} )</td>
</tr>
<tr>
<td>4.</td>
<td>( \text{concl} := \text{conclusion}(P) )</td>
</tr>
<tr>
<td>5.</td>
<td>( \text{sconcl} := \text{SIMPLIFY}()\text{concl}, \text{shyps} )</td>
</tr>
<tr>
<td>6.</td>
<td>( P' := \bigwedge \text{shyps} \Rightarrow \text{sconcl}; \text{return } P' )</td>
</tr>
</tbody>
</table>

}\text{FMCA 2011, Page 49}

\text{Propagating new assignments}
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 $x, y, z, w$:

$$x = \text{hash}(y) \land y = \text{hash}(z) \land z > 0 \land w < \min(x, y) \Rightarrow w < z$$

Before the main search loop begins, a variable is selected to be instantiated. The variable dependency graph for $P$ (constructed following the forementioned rules) is shown in Figure 1.

![Variable dependency graph for P](image1.png)

Fig. 1. Variable dependency graph for $P$

The graph has two strongly-connected components, one containing just $z$, the other containing $x, y, w$. The topological sort returns the vertices in decreasing finish times of the depth-first search on the dag. We pick the last component (i.e., the one with the least dependency). Since this component $z$ is not a set 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. Let's say the oracle procedure Assign picked 1. Then we propagate this assignment by adding the constraint $z = 1$ in $P$ and using the ITP to simplify the hypotheses and conclusion in light of this new information. Propagate returns the following simplified property:

$$P': \quad x = \text{hash}(y) \land y = 5184444 \land w < \min(x, y) \Rightarrow w < 1$$

whose dependency graph is shown in Figure 2.

![Dependency graph for P'](image2.png)

Fig. 2. Dependency graph for $P'$

Since false does not appear in the hypotheses (and neither does true in the conclusion), $P'$ is not inconsistent and we add $z = 1$ to the partial assignment $A$ and the search for the rest of the assignment is continued. Note that Propagate helps eliminate some edges in the variable dependency graph of $P$, breaking cycles (mutual dependency) in the connected component, invariably helping the Select algorithm in the next iteration of the main search loop.

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 $y$ being selected (because it is a leaf node) and being directly assigned a value by virtue of the equality constraint $y = 5184444$. Notice that this is an assignment of type “implied” and was propagated due to the decision assignment ($z = 1$) by the oracle procedure Assign in the previous iteration. This information is again propagated resulting in the further grounded property:

$$P'': \quad x = 5562452 \land w < \min(x, 5184444) \Rightarrow w < 1$$

whose dependency graph is shown in Figure 3.

![Dependency graph for P''](image3.png)

Fig. 3. Dependency graph for $P''$

Notice that since $x$ is constrained to be equal to the constant expression 5562452, it is a leaf node and this eliminates the edge that existed in Fig 2 from $x$ to $w$. The last node in the topological sort of the dag of Fig 3, $x$, is returned by Select, thereby forcing a value satisfying the equality constraint $x = 5562452$. This assignment is further propagated, resulting in the almost grounded property having just one free variable:

$$P''' : \quad w < 5184444 \Rightarrow w < 1$$

Assigning $w$ (using implicit constraint that $w$ is an integer and the local constraints $w < 5184444$ and $w \geq 1$) a value 0 or value 5184445, will lead to inconsistency (after the propagation), in which case we need to throw away the current assign. 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 assignment for $x$ and $y$, in $A$, popping $P'''$ and $P'$ from $S$ and continuing (the main search loop). If an assign to $w$, say $w := 2$, 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 $P$.

We have implemented the proposed method in ACL2 Sedan [10]. We employ the ACL2 interactive theorem proving system [15] to provide the interface methods SIMPLIFY and SMASH. The engineering of the interface with the ACL2 theorem prover and the extension to ACL2, in support of this interface, is described in [5]. We will briefly describe how we implemented the Assign method that was 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 [6]. 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 decided 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.
domain. In our implementation we enable automatic testdata
generation by supporting a notion of an enumerable type in the
otherwise untyped language of ACL2. Separation of concerns
between enumerators and random number generators also
gives us the flexibility to choose between pseudo-geometric,
pseudo-uniform random testing and bounded exhaustive test-
ing. 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.

IV. EXPERIMENTAL EVALUATION AND DISCUSSION

We present two experiments\(^3\) to evaluate our method. In
Section IV-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 IV-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 coun-
terexamples). 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
architecture (ISA) model is a natural functional specification
for any pipelined design. The correctness of the implemen-
tation 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 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 avoid-
ing 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 BEZJUMP 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 (explain a little more).

The objective of the experiment was to evaluate the ef-
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 bisim-
ulation (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 [17]. 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) i.e., \((\text{obs}= S M) \Rightarrow (\text{obs}= S M') \lor (\text{obs}= S' M')\)

Analyzing this high-level property, our method is able to
uncover both the design errors in our MA machine which
manifested as hazards. 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 (micropro-
grams) were provided as inputs. No test driver needed to be
given. The only effort on part of the designer was in writing

\(^3\)We recommend the reader download the experiments from
http://ccs.neu.edu/home/harshrc/fmcad11

FMCAD 2011, Page 51
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 [13] is a declarative modeling language based on sets and relations, primarily used for describing high-level specifications and designs. Alloy Analyzer [14] 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 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 modelling/design examples: recursive definitions and arithmetic. The ACL2 language, on the other hand, has excellent support for recursive definitions (in fact, most interesting properties are expressed using recursive definitions) and arithmetic [19]. 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 multiplications), and restrict ourselves to examples (from the Alloy distribution) that we think Alloy performs well on.

<table>
<thead>
<tr>
<th>Property</th>
<th>Alloy Analyzer</th>
<th>Our method</th>
</tr>
</thead>
<tbody>
<tr>
<td>markSweepComplete</td>
<td>7 46.51 –</td>
<td>0.34 QED</td>
</tr>
<tr>
<td>pasteAffectsHidden</td>
<td>29 138.34 –</td>
<td>0.42 QED</td>
</tr>
<tr>
<td>hidePreservesInv</td>
<td>87 86.03 –</td>
<td>0.26 QED</td>
</tr>
<tr>
<td>cutPaste</td>
<td>3 0.19 CE</td>
<td>0.49 CE</td>
</tr>
<tr>
<td>writeIdempotent</td>
<td>29 98.03 –</td>
<td>0.01 QED</td>
</tr>
<tr>
<td>writeRead</td>
<td>3 0.05 CE</td>
<td>0.83 CE</td>
</tr>
<tr>
<td>lookupYields</td>
<td>3 0.05 CE</td>
<td>0.83 CE</td>
</tr>
<tr>
<td>addLocal</td>
<td>3 0.05 CE</td>
<td>0.83 CE</td>
</tr>
<tr>
<td>addIdempotent</td>
<td>31 112.66 –</td>
<td>0.19 QED</td>
</tr>
<tr>
<td>cutUndoesAdd</td>
<td>31 80.91 –</td>
<td>0.07 QED</td>
</tr>
</tbody>
</table>

TABLE I
Comparison with Alloy Analyzer (AA)

We analyzed 11 properties from 4 Alloy problems (specifications), except the markSweep problem, all the others are from the Alloy book [13] and can alternatively be downloaded from the Alloy distribution.4 Table I 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 SAT solver. The time taken by our method is what the ACL2 macro times 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 reaching the timeout fixed at 180 seconds. 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 3 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 swept into a freelist. The mark-and-sweep Alloy model was taken from an experiment in [12] where Alloy specifications are automatically translated to SMT2 language supported by the Z3 SMT solver [9].

We took the above examples and modelled 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 [10]. These respectively make use of the ordered sets library [8] and the records library [16]) 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 a change in the ACL2 settings when random testing (default) was not good enough to catch the counterexample, we had to revert to bounded exhaustive testing, which is also as automatic as Alloy, but not as efficient, as we observe in Table 1 in the entry of addLocal.

In experiments shown in [12], 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 modelled the problem in ACL2, used sets and maps as mentioned before, the mark procedure (involving transitive closure) is modelled 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.

V. 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 [6] have implemented random testing libraries e.g., Isabelle/HOL [1], Agda [11] and PVS [18]. 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 [20], SAT Checking [21], Refute [22] 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 combining formal specifications and testing date back to at least 1981 [4]. One of the first examples of combining testing and interactive theorem proving was carried using Agda [11]. 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.

A. 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 checking [7].

VI. 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.

VII. ACKNOWLEDGMENTS

We would like to thank Peter Dillinger, Mitesh Jain, and Matt Kaufmann.

REFERENCES