Symbolic Counter-Example Generation for Model Checking

Mahmoud Askari, Hamid Shojaei, Fataneh Faghani
Computer Department, Islamic Azad University,

Abstract- Because of the complexity of digital system validation, formal verification has become an important member of the EDA tools. In formal verification, counterexamples help identify design problems and debugging. Almost all existing verification tools generate one counterexample when a property fails. This paper investigates, how debugging can benefit from using more than one counter-example generated by the verification tool. A new method for making counterexample is proposed that is based on PIRE (Property Intermediate Representation with Extensibility) structure. In this method, all possible counterexamples are generated.

I. Introduction

Today up to 80% of the cost of integrated circuit design is for design validation. This, in turn, causes validation to be an important part of the design process. Formal verification [1] [2] [3] is the process of checking whether a design satisfies certain requirements (properties).

Model checking is a common formal verification technique for property validation. A model checker verifies whether the model satisfies all the desired properties under all possible input sequences.

Properties are usually specified in a property specification language such as Computation Tree Logic (CTL) [8]. The output of the model checker tool is either a confirmation that properties are maintained or a non-confirmation in which case a counterexample [5] is provided. Most existing verification tools generate only one counterexample when a property fails that it is a fairly inarticulate counterexample.

Model checker tools guarantee completeness if properties for a design are proven, otherwise one counterexample is generated to refute it and designer debugs the design with this counterexample. Most counterexamples are not articulate and debugging the design is difficult.

This paper investigates the generation of a good counterexample. In fact we generate all possible counterexamples and then select a good counterexample from them. We use a universal subset of CTL formulae called ACTL.

This paper is organized as follows: ACTL property language is explained in Section 2. The PIRE (Property Intermediate Representation with Extensibility) structure is described in Section 3. Our counterexample generation algorithms are based on this intermediate format. Section 4 is on counterexamples and formal verification. An algorithm for computing the counterexample and its correctness along with an outline of the proof are presented in Section 5. Section 6 is on selecting the good counterexample and advantages of our method. Experimental results are presented in Section 7. We conclude the paper in Section 8.

II. ACTL Property Language

ACTL is a temporal language obtained from the CTL branching temporal language by removing its existential operators. An ACTL formula is satisfied in a state if all outgoing paths from that state satisfy the formula. So if we can find only one path that does not satisfy the formula, the formula fails and that path is a counterexample. However, we will find all erroneous paths and show them as design errors.

ECTL is another subset of CTL. An ECTL formula is satisfied in a state if one outgoing path from that state satisfies the formula. When an ECTL formula fails, it means that there is no path for satisfying the formula. Therefore to generate a counterexample we must show all of paths in a design. Obviously, this is a lengthy process. However, when “!(ECTL)” fails, it means that there exists a path that satisfies the ECTL formula; therefore we can generate counterexamples for it. Generally “A” equals to “!E” and in this paper we generate counterexamples only for ACTL formulas. The following shows equivalency of ACTL and ECTL formulas.

\[
\begin{align*}
AXf & \iff !(EX(!f)) \\
AGf & \iff [(E\ true \ U \ !f)] \\
AFf & \iff ![EG(!f)] \\
A[fUg] & \iff ![E[!g \ U \ (!f*!g)] \ + \ (EG!g)]
\end{align*}
\]

The algorithms, presented for generating counterexamples for a subset of CTL, have a better order of complexity than model checking algorithms. Thus, counterexamples for a certain property can be generated if the property fails.

III. PIRE Data Structure

The PIRE intermediate format [4] has a class corresponding to each ECTL construct. Each class has special elements that are needed for model checking, counterexample construction and coverage computation. These classes have some common features like memory management capabilities. On the other hand, there exist some features that only apply to some classes. For example, some classes have two operands and others have one.

At the top of the hierarchy is the CECTL class, from which all PIRE classes are derived. All common
features and methods are implemented in this class so that all classes can share these features.

Fig. 1 shows the hierarchy of PIRE [4] which will be described here. As shown, in this figure, five classes are derived from $CECTL$. The $CSCTL$ class corresponds to the standard CTL constructs. $CSCTL$ is further divided into two parts, $C1OPSCTL$ and $C2OPSCTL$. The $C1OPSCTL$ class is for the standard CTL constructs with one operand. This class has a member variable named $m_pFirstOperand$ to store its operand. The $C2OPSCTL$ class is for the standard CTL constructs with two operands. The $C2OPSCTL$ class is derived from $C1OPSCTL$, so that it can use the $m_pFirstOperand$ member variable to store its first operand. For storing the second operand, this class has a member variable named $m_pSecondOperand$. Depending on their number of operands, classes corresponding to the standard CTL formulas are derived from one of these two classes.

The $CTempOp$ class, shown in Fig. 1 corresponds to temporal operators in ECTL. These constructs have one operand and use the $m_pECTLFormula$ member variable of this class to store it. All temporal operator classes (e.g., $CAlways$, $CNever$) are derived from this class.

$CCERE$ corresponds to $CERE$ (CTL Extended Regular Expression) constructs of the ECTL. The $CERE$ constructs are divided into two parts, $CSingleCERE$ and $CMultipleCERE$. The $CSingleCERE$ class is for $CERE$ classes with one operand. This class has a member variable named $m_pECTLFormula$ to store its operand. The $CMultipleCERE$ class is for the $CERE$ classes with multiple operands.

$CIdentifier$ class shown in Fig. 1 is designed for storing identifiers and numbers. The $CIdentifier$ Hierarchy is shown in Fig. 2. For example the $CName$ class is designed to store alphabetical identifier names. There are classes like $CBit$, $CBitVector$, $CInteger$ and $CReal$ to store all the information about different types of numbers. And finally, in the hierarchy of PIRE intermediate format (Fig. 1), $CECTLList$ is designed to store an array of ECTL formulas.

Depending on the class type, a special algorithm is applied inside the $ComputSatisfyingStates$ function. This function computes the states that satisfy a formula and returns them in form of BDD (Binary Decision Diagram). A system satisfies a formula if all its initial states are in the satisfying set of the formula. The output of this function is a Boolean value resulted from this condition.

The $ComputeCoveredStates$ class is a function that uses the Hoskote [14] coverage metric to estimate the completeness of a set of properties verified by model checking and computes the set of states covered by the formula of each class.

The $GenerateFailedPath$ function uses approach of [11] and generates a Counter Example when a property fails.

**IV. A Tour of Our Tool**

Attempting to verify a specification that does not hold on the transition system made from a design, a model checker will indicate that the specification is "false". In addition, it will produce a counterexample. A counterexample is a path through the transition system that fails to satisfy the specification. Transition system is a model of a system that contains all relations of the design (Kripke structure).

We have used the symbolic model checking technique for verification. The overall structure of our tool is shown in Fig. 3. As shown, two sets of inputs are required: a VHDL design and ECTL properties.

To extract an FSM description, the design, modeled in VHDL, must be synthesized to hardware elements. First, the input VHDL code is analyzed into $CHIRE$ [9] [10]. $CHIRE$ is an intermediate object oriented $C++$ format, which has a class for every $VHDL$ construct. In the next step of the synthesis process, $CHIRE$ classes are converted to $DFG$ (Data Flow Graph) [11] representations. A $DFG$ is a tree like structure that represents the interconnection of hardware components. This format is considered a generic synthesizable format. $DFG$ consists of discrete functions and memory elements. The discrete functions can be conveniently represented by $BDD$s [6] [7].

Fig. 1. PIRE Intermediate Format Hierarchy

Fig. 2. Identifier Hierarchy in PIRE Intermediate Format

A class of PIRE has three key functions, $ComputSatisfyingStates$, $ComputeCoveredStates$ and $GenerateFailedPath$ which compute satisfying states, covered states and counter example of formulas corresponding to this class respectively.

Properties shown in Fig. 3 are in $ECTL$ which is an extension of $CTL$. We use PIRE that is an object-oriented format for representing ECTL constructs. For a universal CTL formula, all states in a design that are reachable from the initial states should be checked.
However, for an existential CTL formula only one case from the initial states should be found that satisfies the formula. It is clear that algorithms of existential CTL formula can be implemented easier than the universal CTL formula, so universal formulas are converted to existential formulas (as shown in Fig. 3, A to E Conversion). That is, all universal path quantifiers are replaced with the appropriate combination of existential quantifiers and Boolean negations. The "original Formula" field of each new sub-formula is set to point to the original formula.

![Fig. 3 High level block diagram of model checker structure](image-url)

Fig. 3 High level block diagram of model checker structure

Fig.4 Block Diagram of verification engine

**Problem Statement**: Given an FSM $M$ with a set of initial states $S_i$ and an ACTL formula $F$ such that $M, S_i \not\models F$, the task of the counterexample generator is to generate counterexamples and locate the design error. For generating a counterexample we used two important functions, $\text{Satisfies}(f)$ and $\text{Image}(f)$. $\text{Satisfies}(f)$ uses a copy of the set of all states for which formula $f$ is true. $\text{Image}(f)$ is for image computation. Image computation is the process of finding all the successors of a given set of states $S$ according to a set of transitions $T$.

Counterexample for each formula is created with respect to a set of starting states. The generated counterexample for formula $F$ with respect to the start state set $S_i$ is denoted by $C(S_i, F)$. As we traverse down the parse tree, the set of starting states, used to create a counterexample for a particular sub-formula changes. Counterexample for the top-level formula $F$ is created with respect to the set of initial states $S_i$ of $M$.

Often for generating a counterexample, we start from the initial states (i.e., root of the parse tree) and move on the formula to the leaves of the parse tree. It means that for $\forall x. (a = 2 \Rightarrow (AX(a = 3)))$, we first find the bad_state (the state that does not satisfy the formula) for “$\Rightarrow$” then we find the bad_state for “$\Rightarrow$” and … and finally, find the bad_state for “AG”.

In some approaches the counterexample generation is done differently. They start from leaves of the parse tree and move on to the root. In the above example, they start from $a = 2$ and then $a = 3 …$ and finally AG. In our approach the number of transition relation states that are manipulated is less. This is due to the construct of failed formula which is used first in the counterexample generation process (i.e., AG against $(a = 2)$). In our approach the failed property construct is usually complex which makes the traversed states very few. Our method speeds up the counterexample generation because many unused states in the top-down pass are not processed.

V. Counterexample Generation

In this section, we present a new algorithm to compute the set of states in the state space of an FSM for a given ACTL formula for which the formula is not correct. As we mentioned before, model checking is applied only on existential formula and all universal formulas are converted to existential form. However, for counterexample generation the algorithm operates on the original formula. For starting the counterexample generation process, we need those initial states that do not satisfy the formula. These states are computed by model checking engine that is applied on the converted formula.

A. $AG$ Formula Algorithm

Suppose we want to generate all counterexamples for the $AG(f)$ formula. Inputs are $ps$ and the formula itself, $ps$ is an initial state that does not satisfy $AG(f)$. The pseudo code of the algorithm for handling this formula is shown in fig. 5.
We first use $S_1$, $S_2$ and $NS$ computed in the model checking process with $Satisfies(...)$ and $Image(...)$ functions. $S_1$ is the set of all states that satisfy $f$, $S_2$ is the set of all states that satisfy $AG(f)$ and $NS$ is the set of the next states of $ps$ computed by the $Image(...)$ function.

When the $AG(f)$ formula fails, it means that at least one state in the transition relation does not satisfy the $f$ formula. Therefore, we first check whether $ps$ is in $S_1$ or not. If it is not, it means that $ps$ does not satisfy $f$ (i.e., we can conclude that there is something wrong in $f$). So we continue our algorithm with $ps$ as an initial state and $f$ as the formula. On the other hand, if $ps$ is in $S_1$, it means that an error exists in the next states (i.e., we conclude that there should be something wrong in the next states of $ps$). We select all next states ($NS$) one by one. For all states that are not in $S_1$ but are in $S_1$ (i.e., they do not satisfy the $AG(f)$ formula), we continue our algorithm with these bad states as initial states and $f$ as the formula. For all states which are not in $S_2$ but are in $S_1$ (i.e., they do not satisfy the $AG(f)$ formula), we continue our algorithm with these bad-states as its initial states and the $AG(f)$ as its formula. This method continues until we reach the leaf in the parse tree of the failed property. In summary, we act on a failed property based on its definition.

Suppose we want to generate all counterexamples for $AX (f)$ formula with one input named $ps$. See Fig. 6 for the pseudo code of this formula.

When the $AX (f)$ formula fails, this means that at least one state in the next states of $ps$ does not satisfy $f$ formula. Therefore, we consider $S_1$, $NS$, in which $S_1$ is the set of all states that satisfy $f$ and $NS$ is the set of next states of $ps$. However, for all next states ($NS$) we select states not in $S_1$ (i.e., they do not satisfy the $f$ formula) and we continue our algorithm with these bad-states as initial states and $f$ as the formula.

C. AF Formula Algorithm

Referring to Fig. 7, when $AF (f)$ fails, it means that all reachable states in at least a path from $ps$ do not satisfy $f$. Therefore, starting with $ps$, we move step by step to the next states and in every step we check whether the state considered satisfies $AF (f)$. If it does, then that state is a bad-state and we continue with this state as the initial state and $AF (f)$ as the formula. At any time, if a state we are considering satisfies $f$, then we will not continue.

D. AU Formula Algorithm

When the $A (f U q)$ formula fails, this means that a reachable state in at least a path from $ps$ does not satisfy $q$ while all its previous states satisfy $f$ (see Fig. 8).

For $A (f U q)$, we do the same as $AF (f)$, except that in this formula, there are two parameters. Therefore, in each step, we decide based on the two parameters of $AU$.

VI. Selecting a good counterexample

Debugging all counterexamples is a very hard, therefore we need select a good counterexample from all counterexamples, but this is a problem that what counterexample is proper to debugging. Our metric to select counterexample is based on short path. We select shortest counterexample because it is potentially simpler for debugging.
Other benefits of our method are:

1. We implemented our method on PIRE that is an object oriented structure. PIRE is fast as far as the CPU time is concerned, and has an efficient memory usage. Moreover, implementation on PIRE is simple.

2. We are implementing a new metric (Genetic Algorithm) for selecting good counterexample and improve the verification process.

3. We can select a good counterexample among all possible counterexamples. Considered as a good counterexample is the counterexample that is common among all counterexamples for all properties.

4. Counterexamples can be useful for test generation purposes.

VII. Experimental Results

In order to demonstrate the effectiveness of our proposed method, we have conducted some experiments on several of the Texas-97 Verification Benchmarks [15] circuits. Our counterexamples results are useful for several reasons. First, all design transition relations are already generated in the program data structure; therefore we do not use new memory allocations for generating counterexamples. Table 1 shows the results of PIRE structure and Table 2 shows the results of generating counterexamples. Results in Table 1 are based on Win XP and P4. This table compares memory usage in VIS and PIRE structure. Table 1 shows that PIRE is better in terms of memory usage. In Table 2 we compare memory and CPU times of 3 cases. The first case is when all counterexamples are generated, the second case is when one counterexample is generated, and the last case is when all counterexamples are generated and one is selected. This table shows that CPU time and memory usage for generating all counterexamples and selecting a good counterexample is not much more than generating a counterexample randomly.

VIII. Conclusions

In this paper we used a new Object Oriented intermediate structure useful for verification tools. We generated all counterexamples based on this structure and then selected a good counterexample from them. The selected counterexample helps hardware designers in debugging their designs.

Our Experimental results on properties generated for various Texas-97 Verification Benchmark circuits show the effectiveness of our structures for the verification process in terms of memory usage and effectiveness to generate counterexamples. Having these counterexamples available, enables us to select good counterexamples, and helps us in the test generation process.

<table>
<thead>
<tr>
<th>Example</th>
<th>VIS Tool</th>
<th>PIRE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MEM (k)</td>
<td>MEM (k)</td>
</tr>
<tr>
<td>4-arbit</td>
<td>136</td>
<td>24</td>
</tr>
<tr>
<td>8-arbit</td>
<td>128</td>
<td>20</td>
</tr>
<tr>
<td>abp</td>
<td>132</td>
<td>40</td>
</tr>
<tr>
<td>arbiter</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>bakery</td>
<td>132</td>
<td>32</td>
</tr>
<tr>
<td>coherence</td>
<td>144</td>
<td>48</td>
</tr>
<tr>
<td>counter</td>
<td>120</td>
<td>32</td>
</tr>
<tr>
<td>crd</td>
<td>112</td>
<td>32</td>
</tr>
<tr>
<td>Cltp3</td>
<td>144</td>
<td>32</td>
</tr>
<tr>
<td>dcnew</td>
<td>72</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 1. Comparison with VIS
References