Formal Methods for Ranking Counterexamples Through Assumption Mining

Srobona Mitra*, Ansuman Banerjee† and Pallab Dasgupta*
* Dept. of Computer Science & Engineering, Indian Institute of Technology Kharagpur, Kharagpur, India - 721302
† Advanced Computing & Microelectronics Unit, Indian Statistical Institute, Kolkata, India - 700108
Emails: {srobona,pallab}@cse.iitkgp.ernet.in, ansuman@isical.ac.in

***Abstract*—Bug-fixing in deeply embedded portions of the logic is typically accompanied by the post-facto addition to new assertions which cover the bug scenario. Formally verifying properties defined over such deeply embedded portions of the logic is challenging because formal methods do not scale to the size of the entire logic, and verifying the property on the embedded logic in isolation typically throws up a large number of counterexamples, many of which are spurious because the scenarios they depict are not possible in the entire logic. In this paper we introduce the notion of ranking the counterexamples so that only the most likely counterexamples are presented to the designer. Our ranking is based on *assume properties* mined from simulation traces of the entire logic. We define a metric to compute a *belief* for each assume property that is mined, and rank counterexamples based on their conflicts with the mined assume properties. Experimental results demonstrate an amazing correlation between the real counterexamples (if they exist) and the proposed ranking metric, thereby establishing the proposed method as a very promising verification approach.**

I. Introduction

Bugs often escape pre-silicon validation and are detected during post-silicon testing [1]. In large digital integrated circuits such bugs often appear in the glue logic which integrates the various functional units into the design. Today the quality of verification of the functional units is very high and formal methods are widely used in practice [2], [3], [4]. On the other hand, the glue logic, which accounts for a large fraction of the integrated circuit (40% or above) changes significantly from one design to another, is often not documented well enough and typically not formally verified in industrial practice. Designers / verification engineers add a lot of assertions with the glue logic based on their understanding on what the logic is supposed to do.

Bugs appearing in the glue logic are often due to incorrect interpretation of the architecture in specific corner case scenarios which were overlooked during simulation. When such a bug is detected, the appropriate place in the glue logic is corrected and an assertion is added which asserts the correct behavior in the bug scenario and related scenarios. The new assertion, $\mathcal{P}$, is typically a property which is local to the part of the glue logic which has been modified. Figure 1 illustrates the situation – $\mathcal{D}$ represents the entire glue logic, $\mathcal{B}$ represents a very small part of $\mathcal{D}$ which has been modified, and $\mathcal{P}$ is the new assertion which is defined over the signals in $\mathcal{B}$. Here $\mathcal{I}_B$ is the set of inputs feeding into the logic $\mathcal{B}$ from the enclosing modules in $\mathcal{D}$ and $\mathcal{O}_B$ is the set of outputs which are driven by $\mathcal{B}$ and influence the rest of the glue logic.

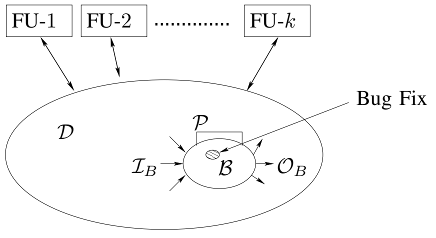

Fig. 1. Problem scenario

In order to make sure that the bug has been fixed properly, we must determine whether $\mathcal{P}$ formally holds on $\mathcal{D}$. Model checking [5], [6] $\mathcal{P}$ on $\mathcal{D}$ directly will not be a feasible option in industrial practice due to the large size of $\mathcal{D}$. Since $\mathcal{B}$ is small, it is feasible to formally verify $\mathcal{P}$ on $\mathcal{B}$ in isolation, that is, under the assumption that the inputs to $\mathcal{B}$ are unconstrained. If $\mathcal{P}$ holds on $\mathcal{B}$ in isolation, then $\mathcal{P}$ also holds on $\mathcal{D}$, however the reverse is not true. For example, a counterexample generated by model checking $\mathcal{P}$ on $\mathcal{B}$ in isolation may not exist in $\mathcal{D}$, that is, $\mathcal{D}$ may not be able to create the scenario depicted by the counterexample on the interface of $\mathcal{B}$. Typically the spurious counterexamples are numerous and often exceed the number of real counterexamples, if any.

When $\mathcal{P}$ does not hold on $\mathcal{B}$ in isolation, a model checking tool may present thousands of counterexamples to the verification engineer. The large set of counterexamples is often useless to the verification engineer, since she has no easy way of separating the real ones (if any) from the spurious ones. It is practically infeasible for the designer to manually examine each counterexample and determine whether it is real or spurious within the large glue logic $\mathcal{D}$. Formally verifying whether a counterexample over $\mathcal{B}$ is real in $\mathcal{D}$ is infeasible in practice due to the size of $\mathcal{D}$ and the fact that the counterexample is defined over signals which are not on the interface of $\mathcal{D}$.

In this paper we present a method for ranking the counterexamples obtained from model checking $\mathcal{P}$ on $\mathcal{B}$, with the expectation that higher ranked counterexamples are more likely to be real than the lower ranked ones. The proposed approach for ranking counterexamples consists of the following steps:

1) *Assumption mining.* We use assertion mining tools on simulation traces of $\mathcal{D}$ to create a large set of possible *assume properties* over the interface of $\mathcal{B}$. The assumptions are mined with the goal of capturing the effect of $\mathcal{D}$ on the interface of $\mathcal{B}$. The assertion miner is biased

accordingly.

2) *Assumption weighting.* We assign a *belief* value to each mined assume property based on the evidences received in support of the assume property from the simulation traces. These weights are real values between zero and one, and thereafter treated as confidence measures for the assume properties.
3) *Counterexample ranking.* We group counterexamples based on the sets of assume properties they contradict. We aggregate the belief values for the assume properties in the conflict set to compute a confidence metric for the counterexample. This confidence metric is used to rank the counterexamples. Intuitively, counterexamples which contradict assume properties of high support are ranked lower than the others. Counterexamples which do not have any conflict with the assume properties are ranked the highest. Finally counterexamples are presented to the verification engineer in descending order of rank.

The key requirement in the third step is to be able to rank the counterexamples *without actually generating all of them through model checking*. This is achieved by grouping the counterexamples based on the sets of assume properties they contradict and *by ensuring that a counterexample from the same family is not generated again while searching for counterexamples having higher confidence*.

It is important to realize that model checking $\mathcal{P}$ on $\mathcal{B}$ under the mined assume properties is not a good option because all the mined properties are not truly valid. A real counterexample, which conflicts with a fictitious assume property may not be generated if we adopt this approach. Ranking counterexamples is a better option because it does not suppress real counterexamples.

The proposed approach was applied on several benchmark circuits, and the results show an amazing correlation between the nature of the counterexamples (that is, real or spurious) and the ranks assigned to them by our method. We developed an experimental set up in which the ranking tool was made to work without any additional knowledge about the nature of the circuit, and yet it was able to consistently rank the real counterexamples higher than most of the spurious ones. In the cases where no real counterexamples existed, the confidences of most counterexamples were found to be low.

II. Running Example

In this section, we introduce a running example gleaned out of one of the benchmark circuits. Figure 2 shows the components of our verification problem, namely:

1) The embedded logic, $\mathcal{B}$, which has been magnified for the purpose of showing the details. $\mathcal{B}$ is actually much smaller than $\mathcal{D}$. Further:
   - $\mathcal{I}_B = \{A3, B2, B3, CT0, n99, \ldots, n137\}$
   - $\mathcal{O}_B = \{CT2, CT1, ACVQN3, \ldots, n218, n223\}$
2) The entire logic, $\mathcal{D}$, having inputs, $\{A3, B2, B3, \ldots, START\}$ and outputs, $\{P0, P1, \ldots, READY\}$. Most of the signals in $\mathcal{I}_B$ (except A3, B2 and B3) are internal signals of $\mathcal{D}$.

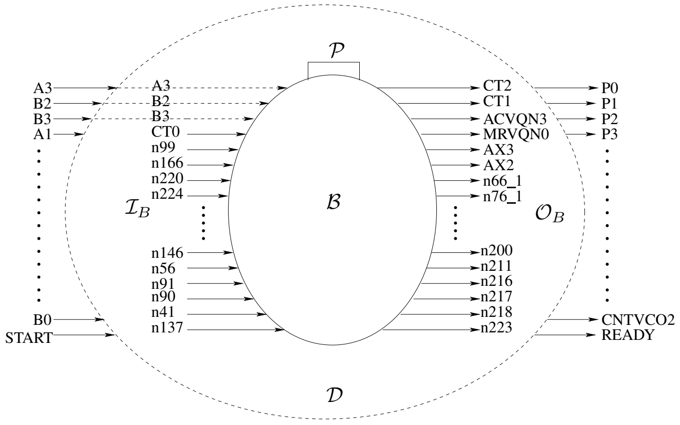

Fig. 2. A snapshot of a running example from ISCAS-89 benchmarks

3) Property $\mathcal{P}$ over $\mathcal{I}_B$ and $\mathcal{O}_B$ that covers the bug scenario:

```
property P;
   @(posedge clk) ((n146==0 ##1 n146==0 ##5 n220==1
   && n99==0 && n137==0 && n146==0 && n56==1 && n91==0
   ##1 A3==0 && n224==1 && n99==0 && n137==0 && n166==0
   && n91==1 ##1 n90==0) |=> (AX3==0));
endproperty
```

The example presented here may appear to be non-intuitive and dry to the reader, but in reality, the verification engineer is faced with very limited documented information on the glue logic. The example presented here is representative of the information (or lack of it) available to the verification engineer.

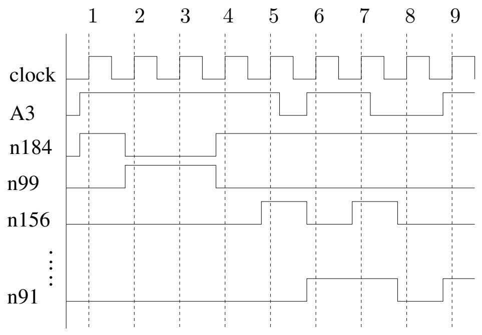

Fig. 3. Example counterexample obtained by model checking $\mathcal{P}$ on $\mathcal{B}$

Model checking $\mathcal{P}$ on $\mathcal{B}$ in isolation returns the counterexample shown in Figure 3. Though the model checker initially comes up with one counterexample as a testimony of failure, a large number of counterexamples may exist, corresponding to different possible valuations of the signals of $\mathcal{I}_B$. Not all of these counterexamples are real in $\mathcal{D}$. Our goal is to associate a rank with the counterexamples (without generating all of them of course), and report those counterexamples that are most likely to be real. This paper presents a formal basis for ranking the counterexamples.

We used Goldmine [7] for mining assumptions for this example. Goldmine produced a set, $\mathcal{A}$, of 93 assume properties as likely invariants. A few of them are listed below. It may be noted that the signals, $n146$, $n99$, $n91$ and $n137$ are inputs of $\mathcal{B}$ as shown in Figure 2.

```
property A1; @(posedge clk) ((ACVQN3==1) |=> (n146==0));
endproperty
property A4; @(posedge clk) ((n184==1) |=> (n99==0));
endproperty
property A8; @(posedge clk) ((n184==0) |=> (n99==1));
endproperty
property A9; @(posedge clk) ((n156==1) |=> (n91==0));
```

```
endproperty
property A12; @(posedge clk) ((n123==1 ##1 n180==1 && n217==0)
                          |=> (n137==0));
endproperty
............
```

III. Counterexample Ranking Metric

This section presents the proposed formal approach behind counterexample ranking. Figure 1 illustrates the scenario captured by the following problem statement.

Given:

1) The logic implementation of $\mathcal{B}$.
2) A property $\mathcal{P}$ on $\mathcal{B}$. If $\mathcal{B} \models \mathcal{P}$, then there are no counterexamples, hence we consider those cases where $\mathcal{B} \not\models \mathcal{P}$.
3) A set of assume properties $\mathcal{A} = \{A_i\}$ defined on the signals in $\mathcal{D}$. Since these are mined from simulation traces, not all of these assume properties are necessarily valid in $\mathcal{D}$. Recall that $\mathcal{D}$ is too large for model checking, hence validity of the mined assume properties cannot be formally verified.
4) Belief values, $Bel(A_i)$, associated with each assume property $A_i \in \mathcal{A}$, where $0 \leq Bel(A_i) \leq 1$. We shall explain later how these values are computed.

Goal:

- To group counterexamples based on the subsets of $\mathcal{A}$ they refute.
- To associate a confidence metric with each family of counterexamples, and rank these families based on the confidence metric.
- To demonstrate that typically real counterexamples belong to high-ranked families.

Since the assume properties are mined from the simulation traces, and the simulation traces are not exhaustive (that is, they do not cover all behaviors), we associate a *belief value* with each assumption. The belief values are computed based on the available evidence in favor of the assume property and are directly related with simulation coverage. Intuitively the belief value of an assume property is the probability that the assume property is true, given the available evidence from the simulation traces. Computation of belief values is explained in Section IV.

A. Confidence computation

Given a counterexample, $\mathcal{C}$, returned by model checking $\mathcal{P}$ on $\mathcal{B}$, we compute our *confidence* on the counterexample as follows.

1) *Identify the assume properties which are in conflict with $\mathcal{C}$:* If the counterexample is true, then the conflicting assume properties must all be untrue. Let $\mathcal{A}_{\neg\mathcal{C}}$ be the set of assumptions in $\mathcal{A}$ which are in conflict with $\mathcal{C}$.
2) *Calculate the confidence on $\mathcal{C}$:* If $\mathcal{A}_{\neg\mathcal{C}}$ is empty, then we associate a confidence of 1.0 with the counterexample, $\mathcal{C}$. Otherwise, the confidence on $\mathcal{C}$ is computed as:

$$\pi(\mathcal{C}) = \Pi_{A_j \in \mathcal{A}_{\neg\mathcal{C}}}(1 - Bel(A_j))$$

The intuitive meaning of this metric is as follows. If any of the assume properties in $\mathcal{A}_{\neg\mathcal{C}}$ is valid, then $\mathcal{C}$, which contradicts it, cannot be a real counterexample. Therefore, based on available evidence, $\mathcal{C}$ can be a real counterexample, if each assume property in $\mathcal{A}_{\neg\mathcal{C}}$ is not valid. $Bel(A_j)$ models the probability that A_j is valid based on available evidence, and hence $\pi(\mathcal{C})$ computes the joint probability that all assume properties in $\mathcal{A}_{\neg\mathcal{C}}$ are invalid.

In the approach proposed in this paper, we assume that the assume properties are independent of each other, and hence the joint probability distribution is as given above. In future, we intend to make this analysis more accurate by considering the dependence between assume properties.

Example 3.1: Consider the counterexample $\mathcal{C}$ shown in Figure 3. In Step-1, we find that A_4, A_8 and A_9 are the assumptions which fail on this counterexample trace. A_4 fails at cycle 2, A_8 fails at cycle 4 and A_9 fails at cycle 6. Therefore, $\mathcal{A}_{\neg\mathcal{C}} = \{A_4, A_8, A_9\}$. $\mathcal{C}$ can be a true counterexample in $\mathcal{D}$ if each of A_4, A_8 and A_9 are invalid in $\mathcal{D}$. Since A_4, A_8, A_9 are assumed to be independent, the confidence on $\mathcal{C}$ is computed as $(1 - Bel(A_4)) * (1 - Bel(A_8)) * (1 - Bel(A_9))$. □

B. Counterexample Families

All counterexamples that refute the same set of assume properties will have the same confidence value. This allows us to group counterexamples into families as formally defined below.

Definition 3.1: [**Counterexample Family**] Counterexamples which refute the same set of assume properties constitute a counterexample family. □

It is important to note the following about counterexample families. Suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ are two sets of assume properties. If $\mathcal{A}_1 \subset \mathcal{A}_2$, then the counterexamples which refute all members of $\mathcal{A}_2$ also refute all members of $\mathcal{A}_1$. These counterexamples do not belong to the same family as those which refute only the members of $\mathcal{A}_1$. In fact counterexamples in the family of $\mathcal{A}_2$ have strictly lower confidence than counterexamples in the family of $\mathcal{A}_1$ because they refute more assume properties. Therefore, once a counterexample family is found around a set of assume properties, the search for higher ranked counterexample families need not examine families involving supersets of this set of assume properties. This observation leads to significant pruning of the search space.

After finding a family of counterexamples, we search for counterexamples from families having higher confidence. A model checker, unless biased, is not guaranteed to generate a new counterexample when asked to check $\mathcal{P}$ on $\mathcal{B}$. Let $\mathcal{A}_\mathcal{C}$ denote the disjunction of assume properties in $\mathcal{A}_{\neg\mathcal{C}}$. After finding the first counterexample, $\mathcal{C}$ and computing its confidence, we model check the property $\mathcal{P}$ on $\mathcal{B}$ in presence of the assume property, $\mathcal{A}_\mathcal{C}$. This has the following effects:

1) It prevents counterexamples from the same family as $\mathcal{C}$.
2) It prevents counterexamples from families represented by supersets of $\mathcal{A}_{\neg\mathcal{C}}$, which have lower confidence than the ones in the family of $\mathcal{C}$.

After each counterexample is found, we prevent counterexamples from its family to show up in future by adding a new assume property for model checking $\mathcal{P}$ on $\mathcal{B}$. This enables us to explore families of counterexamples without actually generating all counterexamples in each family.

Example 3.2: We continue with our running example. In order to generate a counterexample from a different family than the one shown in Figure 3, we add the disjunction of the assume properties $\{A_4, A_8, A_9\}$*, as an assume property while model checking* $\mathcal{P}$ *on* $\mathcal{B}$ *again. This combination of assume properties ensures that the model checker will not generate a counterexample from the same family, nor from a family which refutes a superset of* $\{A_4, A_8, A_9\}$. $\square$

Finally, counterexamples are ranked based on their confidence values and the top ranked counterexamples are presented to the designer for diagnosis. The designer will use her experience to attempt to construct a scenario over $\mathcal{D}$, which reproduces the counterexample over $\mathcal{B}$. If a counterexample is found to be real, then it proves that the assume properties it refutes are all invalid. On the other hand, if a counterexample is found to be spurious, then the designer will typically be able to confirm the validity of one or more of the assume properties which were contradicted by the counterexample. This knowledge will eliminate all those counterexample families which contradicted one or more of these assume properties, and then the proposed approach can be used to present the most likely counterexamples in the light of the new knowledge.

IV. Assumption Mining and Confidence Metric

In the previous section we skipped the method for computing the *belief* values for the mined assume properties. This computation is the focus of this section, but we first describe the source of our assume properties.

There has been a large volume of literature on the automatic generation of invariants in both software [8], [9] and hardware domains [10], [11], [12], [7]. Invariant generators typically work either on the program source (*static analysis*) [13] or on program simulation traces (*dynamic analysis*) [8], [7] or on a combination of both [7] to identify recurrent temporal patterns as likely invariants. Source-code based invariants are sound (that is, the invariants reported are always valid) but not scalable. Invariant generators which work on simulation traces are scalable but not sound, since invariants are identified from available simulation data, which is usually less likely to be specified for all possible input patterns.

A number of research articles on dynamic invariant generators has been reported in recent literature in the hardware verification community. In [10], [14], dynamic invariant generators have been proposed (tools called PropGen and IODINE) for extracting simple patterns having pre-specified limited temporal depth. Bertacco et al. have proposed abstracting out the control behavior and transactions of a hardware design from traces [11] (tool - Inferno). In [15], Fey et al. propose combining the simple properties generated as in [10], [14] temporally to form more complex temporal properties (tool - Dianosis). In [12], Seshia et al. have proposed mining temporal patterns from a trace by finite pattern automaton-based mining (tool - Scalable Assertion Miner or SAM).

In [7], [16], Vasudevan et al. have proposed a tool called Goldmine, which combines data mining and static analysis techniques for inferring dynamic invariants from simulation traces. GoldMine uses decision tree based supervised learning algorithms to analyze simulation traces using domain-specific information incorporated from the lightweight static analyzer in its learning algorithms. We use Goldmine in our experimental set up as discussed in Section V.

In this work, we use Goldmine [7], [16] for mining likely candidate invariants from simulation traces of $\mathcal{D}$. The mined properties are represented as invariants over bounded temporal expressions in SystemVerilog Assertions (SVA) over the signals in $\mathcal{D}$. Let $\mathcal{S}_\mathcal{D}$ denote the total set of signals in $\mathcal{D}$.

Definition 4.1: **[Mined Invariant:]** A mined invariant is a SVA property of the form:
`always @(posedge clk)` ψ `|=>` φ
where ψ is a bounded SVA expression over $\mathcal{S}_D$, and φ is a literal from $\mathcal{S}_\mathcal{D}$. $\square$

There are two important considerations for biasing the invariant miner to address our needs. These are as follows:

1) Our objective is to use the invariant miner to produce *assume properties* at the interface of $\mathcal{B}$, that is, the desired invariants should capture the effect of $\mathcal{D}$ on the inputs of $\mathcal{B}$. Therefore we bias the invariant miner to look for assertions of the above form where φ *is a literal from the input signals of* $\mathcal{B}$.
2) The invariant miner allows us to choose an upper bound on the sequential depth of ψ. Choosing this bound judiciously is an important consideration. Invariants with smaller sequential depth are fewer in number as compared to invariants with larger sequential depth but are typically more reliable than the rest. In other words, since simulation coverage of longer sequential patterns are less than shorter ones, spurious invariants are more likely to be of larger sequential depth. We do not yet have a mathematical theory for choosing this bound, but in our experiments choosing a bound of about 5 seemed to work well.

On our running example, we used GoldMine for mining likely assumptions. The sequential depth of ψ was set up to 5, indicating that ψ can have a sequential depth between 1 and 5. We used a single simulation trace with random inputs for 10000 simulation cycles. Some of the assumptions produced by GoldMine are shown in Section II.

We now focus our attention on associating a *belief* value with each mined assume property. Since the mined assume properties are reported as possible *invariants*, there is no evidence to the contrary, that is, there is not even a single instance in the simulation traces where a reported invariant is refuted. In other words, the invariant miner never reports a property if any counterexample exists in the simulation traces.

Therefore our belief on an assume property proposed by an invariant must be based on the coverage of scenarios that are relevant for the property. Intuitively, if simulation has covered most of the scenarios that are relevant for a given assume property, then we strongly believe that the assume property is valid. Alternatively, if simulation has covered very few scenarios that are relevant for a given assume property, then our belief that the assume property is true is low. *How can we quantify this notion of belief?*

To explain our notion of belief computation, we consider a toy example with four signals, $y1, y2, y3, y4$, and the following invariant:

```
property Q; @(posedge clk) y1 ##1 y2 |=> y4;
endproperty
```

The signals, $y1, y2, y3, y4$, can have 2^4 distinct valuations in each cycle, and therefore can have 2^8 valuations over two cycles. Out of these valuations, the antecedent, $\psi = (y1\#\#1y2)$, is satisfied in $2^6 = 64$ valuations. This figure is obtained by counting all possible valuations of $y2, y3, y4$ in the first cycle, times all possible valuations of $y1, y3, y4$ in the second cycle. Now suppose simulation has seen 48 out of these 64 valuations and has seen that $y4$ is true in all of them. Then our belief that the above property is valid is $48/64$, which is 0.75.

Now suppose a cone-of-influence analysis reveals that $y3$ cannot affect $y4$. With this information, we can remove $y3$ from our analysis, and then the number of ways in which ψ can be satisfied reduces to $2^4 = 16$. We also need to merge those matches of ψ in the simulation traces which differ only in $y3$. Suppose the simulation has seen 14 distinct valuations after merging. Then our revised belief that the property is valid is $14/16$, which is 0.875. It is easy to see that cone-of-influence reductions can only increase the belief metric. In other words, belief computation becomes more informed through cone-of-influence analysis, which works to our advantage.

Definition 4.2: **[Belief Computation]** Formally, our belief, $Bel(\mathcal{P})$, in a mined property, $\mathcal{P}$ of the form:
`always @(posedge clk)` ψ `|=>` φ
is defined as the fraction of ψ-satisfying valuations of the signals affecting φ that has been seen in the simulation traces. We assume that the miner never reports a property if any counterexample exists in the simulation traces. □

How is our belief affected if more simulation is performed? If simulation covers new ψ-satisfying scenarios, then our belief improves. If simulation hits any scenario which refutes the mined property, then the belief computation becomes irrelevant since we know that the property is invalid.

Example 4.1: We return to our running example. A cone-of-influence analysis reveals that a set of 12 *variables affect the signal,* $n99$*. Now consider the following two properties returned by the invariant miner:*

```
property A4; @(posedge clk) ((n184==1) |=> (n99==0));
endproperty
property A8; @(posedge clk) ((n184==0) |=> (n99==1));
endproperty
```

The antecedents of both these properties can be matched in $2^{11} = 2048$ *ways. The number of distinct matches found in the simulation traces for the antecedents of* A_4 *and* A_8 *were respectively* 43 *and* 1320*. Therefore* $Bel(A_4) = 43/2048$ *and* $Bel(A_8) = 1320/2048$*.* □

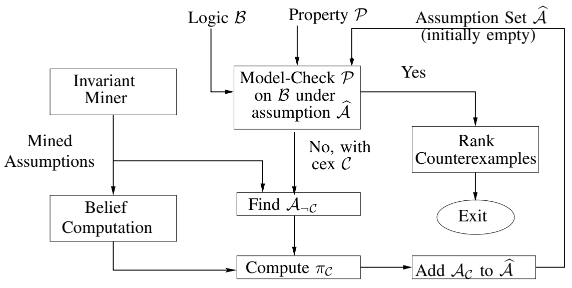
Logic $\mathcal{B}$
Property $\mathcal{P}$
Assumption Set $\widehat{\mathcal{A}}$ (initially empty)
Invariant Miner
Model-Check $\mathcal{P}$ on $\mathcal{B}$ under assumption $\widehat{\mathcal{A}}$
Yes
Mined Assumptions
No, with cex $\mathcal{C}$
Rank Counterexamples
Belief Computation
Find $\mathcal{A}_{\neg\mathcal{C}}$
Exit
Compute $\pi_\mathcal{C}$
Add $\mathcal{A}_\mathcal{C}$ to $\widehat{\mathcal{A}}$

Fig. 4. Counterexample ranking tool flow

V. Tool Flow and Experimental Results

Figure 4 shows our overall flow for ranking of counterexamples. As described in Section III, $\mathcal{A}_{\neg\mathcal{C}}$ represents the set of mined assume properties having conflict with the counterexample $\mathcal{C}$. $\pi(\mathcal{C})$ represents the confidence on $\mathcal{C}$ and $\mathcal{A}_\mathcal{C}$ represents the disjunction of the assume properties in $\mathcal{A}_{\neg\mathcal{C}}$. In each iteration, a new counterexample family is found and a new assume property is added into the set $\widehat{\mathcal{A}}$. At the beginning of each iteration, $\mathcal{P}$ is model checked on $\mathcal{B}$ under the conjunction of the assume properties in $\widehat{\mathcal{A}}$.

We use the tool Goldmine as the invariant miner, the Magellan [17] tool from Synopsys [18] as the model checker, and the VCS [19] simulator from Synopsys for finding the set of assume properties that conflict with a given counterexample. We evaluated our approach on test-cases of various representative sizes from the ISCAS-89 [20] benchmark circuits. To be able to verify the counterexamples (by model checking the property $\mathcal{P}$ on $\mathcal{D}$) and classify them as real or spurious, we limited ourselves to only circuits of modest sizes which the model checker is able to handle. We experiment with both kinds of properties, ones which actually fail on $\mathcal{D}$, i.e., they have real counterexamples in $\mathcal{D}$ and ones which actually pass on $\mathcal{D}$ and have no real counterexamples in $\mathcal{D}$, and only spurious counterexamples in $\mathcal{B}$.

In Table I, Columns 2 and 3 present the dimensions of the circuit. Columns 4-6 present the results of assumption mining and belief computation. Specifically, Column 4 gives the number of assume properties returned by GoldMine for the circuit and Column 5 presents the run time of Goldmine. Column 6 presents the run time for the belief computation step. Columns 7-8 present the results of counterexample ranking. Specifically, Column 7 gives the number of different counterexample families seen in our ranking methodology and Column 8 presents the total run time of our approach (excluding the initial assumption mining time).

In our experiments we used a greedy approximation of the method described in Section III-B. Instead of adding the disjunction of the set, $\mathcal{A}_{\neg\mathcal{C}}$ of assumptions which refute a given counterexample, $\mathcal{C}$, into the set of assume properties, $\widehat{\mathcal{A}}$, we add the assumption having the highest belief from $\mathcal{A}_{\neg\mathcal{C}}$. This heuristic confines the search to families having higher confidence and produces excellent results, as demonstrated by our experiments.

Figure 5 summarizes the results of our ranking methodology obtained graphically. For each circuit, we have plotted a

Ckt	Flops #	Gates #	Inv #	GM (mins)	Belief (mins)	Cex families#	Rank (mins)
s298	14	119	200	0.402	30.09	15	8.28
s344	15	160	100	1.72	17.95	11	5.25
s349	15	161	125	1.65	20.87	20	5.43
s1196	18	529	300	0.43	41.135	34	18.35
s713	19	393	150	1.74	22.12	19	8.12
s400	21	164	170	2.11	25.31	21	21.34
s526	21	193	235	2.96	35.5	15	32.2
s1423	74	657	400	5.23	37.2	40	25.10
s5378	179	2779	550	12.42	51.7	50	55.15
s9234.1	211	5597	130	20.14	65.2	50	68.00
s1238	18	508	180	1.5	25.05	20	6.13
s838.1	32	446	150	3.54	15.23	38	8.13
s641	19	379	80	0.53	12.0	14	7.26
s444	21	181	450	5.76	45.0	26	25.94
s420.1	16	218	58	0.43	10.5	30	22.75

TABLE I
RESULTS: ASSUM. MINING AND BEL. COMPUTATION

line showing the distribution of counterexample confidence of top 10 counterexamples in descending order of confidence. We have shown which of the counterexamples we saw are actually real and which are spurious by different signs. A dotted vertical line separates the counterexamples assigned high ranks above 80% confidence from the lower ranked ones. The distribution shows an interesting correlation between the ranks assigned and the nature of the counterexamples (real / spurious). For the failing property (which fails on $\mathcal{D}$), most of the counterexamples which are real in $\mathcal{D}$ belong to the right side of the dotted line, indicating that they have been assigned high ranks by our metric. On the other hand, spurious ones are mostly distributed to the left of the 80% confidence line and hence are of lower rank. For the cases where the property is actually true on $\mathcal{D}$ (hence all counterexamples are spurious), most of the counterexamples are indeed assigned low confidences (to the left of the 80% confidence line). Overall, the results demonstrate a high degree of correlation between the real counterexamples and our ranking metric, thereby establishing our method as a promising approach.

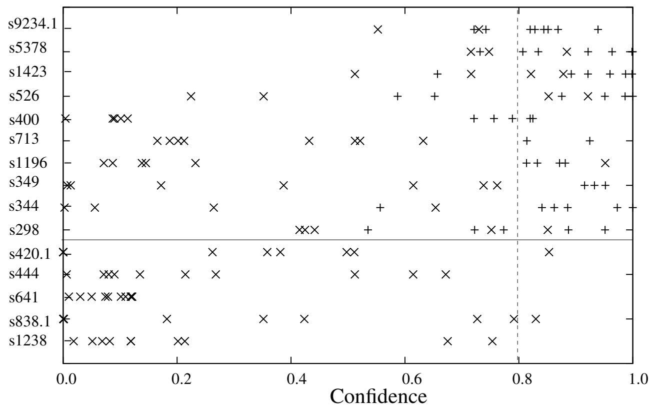

Fig. 5. Confidence of top 10 counterexamples for each circuit: Real ones are shown with + and spurious ones are shown with ×. The ones above the horizontal line have real counterexamples on $\mathcal{D}$, while the ones below the line have all spurious counterexamples ($\mathcal{P}$ holds actually on $\mathcal{D}$)

VI. CONCLUSION

In this work, we present a methodology for ranking counterexamples obtained by model checking a property on a deeply embedded portion of the glue logic of a large design. Our experiments on benchmark circuits demonstrate that for circuits where real counterexamples exist, our methodology assigns higher ranks to the real counterexamples than most of the spurious ones and most of the highest ranked counterexamples in these cases are real. For circuits on which there are no real counterexamples, the confidences computed by our method for most of the obtained counterexamples are found to be low. This demonstrates the correlation between our ranking metric and the nature of the counterexamples.

VII. ACKNOWLEDGEMENT

The authors thank Intel Technology India Pvt. Ltd. for supporting this research and Harish Kumar, MGD-BDC Xeon Architecture Manager, Intel Technology India Pvt. Ltd., for his help and insight in identifying this problem. The authors also thank Rakshith M, M. Tech student, Department of Computer Science and Engineering, IIT Kharagpur, for his help in implementing a part of the tool flow.

REFERENCES

[1] K.-H. Chang, I. L. Markov, and V. Bertacco, “Automating post-silicon debugging and repair,” in *ICCAD*, 2007, pp. 91–98.

[2] T. Schubert, “High level formal verification of next-generation microprocessors,” in *DAC*, 2003, pp. 1–6.

[3] B. Bentley, “Validating the intel pentium 4 microprocessor,” in *DAC*, 2001, pp. 244–248.

[4] R. Kaivola, “Formal verification of pentium 4 components with symbolic simulation and inductive invariants,” in *CAV*, 2005, pp. 170–184.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla, “Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach,” in *POPL*, 1983, pp. 117–126.

[6] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra, “Automatic verification of sequential circuits using temporal logic,” *IEEE Transactions on Computers*, vol. 35, no. 12, pp. 1035–1044, 1986.

[7] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, “Goldmine: Automatic assertion generation using data mining and static analysis,” in *DATE*, 2010, pp. 626–629.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *IEEE Trans. on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[9] S. Hangal and M. S. Lam, “Tracking down software bugs using automatic anomaly detection,” in *ICSE*, 2002, pp. 291–301.

[10] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty, “Iodine: a tool to automatically infer dynamic invariants for hardware designs,” in *DAC*, 2005, pp. 775–778.

[11] A. DeOrio, A. Bauserman, V. Bertacco, and B. Isaksen, “Inferno: Streamlining verification with inferred semantics,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 728–741, 2009.

[12] W. Li, A. Forin, and S. Seshia, “Scalable specification mining for verification and diagnosis,” in *DAC*, 2010, pp. 755 –760.

[13] B. H. Cheng and G. C. Gannod, “Abstraction of formal specifications from program code,” in *International Conference on Tools for Artificial Intelligence*, 1990, pp. 125–128.

[14] G. Fey and R. Drechsler, “Improving simulation-based verification by means of formal methods,” in *ASP-DAC*, 2004, pp. 640–643.

[15] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rülke, “Automatic generation of complex properties for hardware designs,” in *DATE*, 2008, pp. 545–548.

[16] L. Liu, D. Sheridan, B. Tuohy, , and S. Vasudevan, “Towards coverage closure: Using goldmine assertions for generating design validation stimulus,” in *DATE*, 2011, pp. 1–6.

[17] “Magellan: Hybrid rtl formal verification,” http://www.synopsys.com/tools/verification/functionalverification/pages/magellan.aspx.

[18] “Synopsys,” http://www.synopsys.com/.

[19] “Vcs,” http://www.synopsys.com/tools/verification/functionalverification/pages/vcs.aspx.

[20] “Iscas’89 benchmarks,” http://www.cs.ubc.ca/spider/ajh/courses/cpsc538d/ISCAS89/.