# Debugging Missing Assumptions in a Formal Verification Environment

Brian Keng[1], Evean Qin[2], Andreas Veneris[1,3], Djordje Maksimovic[1]

***Abstract*—In the past decade, formal tools have increased functional verification efficiency by searching for corner-case bugs using mathematical reasoning. At the same time, this practice has introduced new challenges when failures are detected. Once a counter-example is returned by a formal tool, the user typically does not know if the failure is caused by a design bug, an incorrectly written assertion, or a missing assumption. This paper introduces a novel methodology to automatically debug missing assumptions. We first present an algorithm to automatically generate missing input constraints given a failing counter-example. The algorithm is then extended to provide higher quality assumptions using multiple failing counter-examples. A function is extracted from these counter-examples that encodes the input combinations that cause the assertion to fail. This function is later used to generate a list of fixed cycle assumptions that prevent failures similar to the generated counter-examples. These filtered assumptions can then be used as hints for the actual missing assumption. Further, if a missing assumption is not the cause of the failure, the method offers the additional benefit that the counter-examples it generates can be utilized to debug the RTL and/or the assertion. Preliminary experimental results show that the generated properties provide a strong intuition as to what input constraints may be missing.**

## I. Introduction

Functional debugging is one of the largest bottlenecks in the design cycle taking up 60% of the total verification time [1]. To cope with this bottleneck, many debugging techniques [2]–[4] have been introduced to automatically localize design errors and improve debugging efficiency. At the same time, techniques such as formal property checking and assertion-based verification [5] have grown in popularity, leading to new challenges that extend beyond traditional design error debugging.

Formal property checkers [6] aim to increase verification efficiency by globally verifying an assertion using mathematical models which encode the design intent. In the ideal case, if an assertion is violated, the formal tool returns a single counter-example allowing detection and debugging of corner case design bugs. However, as extensively documented in industry reports [7], debugging formal counter-examples can be challenging, as the engineer does not have confidence whether the observed failure is due to a design bug, an incorrectly written assertion, or a missing assumption.

Assumptions are necessary in formal verification as they model the design's intended environment and ensure that Register Transfer Level (RTL) bugs can be detected. Nevertheless, debugging missing assumptions can be a challenging task because – unlike assertions – they are rarely explicitly documented. Instead, they are expressed implicitly by either the design specification or the functionality of adjacent design blocks. For the engineer, this can lead to a tedious "guess-and-check" iterative debugging process, introducing many time-consuming calls to the formal tool. To alleviate this pain and make formal technology effective to its full potential, the engineer today needs more debug automation to help analyze the behavior of the counter-example and identify candidate missing assumptions.

In this work, we present an algorithm that takes the first steps towards automated debugging of missing input constraints in a formal Register Transfer Level (RTL) verification flow. This algorithm automatically generates fixed cycle input constraints in the form of SystemVerilog properties from a failing formal counter-example. In essence, the goal for these added assumptions is to provide suggestions or hints to the engineer in identifying the missing assumption, or to provide confidence that the problem is most likely into the RTL and/or the assertion and not into the assumption(s) itself.

This algorithm is then extended to overcome limitations in the quality of the assumptions it generates. This is accomplished by generating extra counter-examples from an existing designer specified set, and then using these new counter-examples to generate useful assumptions. The algorithm complements other debugging techniques as it provides additional information through new counter-examples. In detail, the contributions of the work are twofold:

- A novel algorithm to generate multiple distinct counter-examples from a single assertion failure by iteratively extracting constraints from previous counter-examples and re-running the formal tool. When a missing assumption is not the root-cause of the observed failure, the generated counter-examples can be used by the engineer to improve the resolution of existing automated tools when debugging incorrect assertions [8] and/or RTL design errors [3].
- A method of using multiple distinct counter-examples to improve the quality of results of the assumption debugging methodology.

An extensive set of experimental results have been performed over a wide variety of `OpenCores` [9] designs with SystemVerilog assertions written from their specification documents. Results confirm the efficiency in generating the new properties as well as their ability to provide effective guidance as to what input constraints may be missing. Multiple counter-examples are shown to reduce the number of generated assumptions by 38% on average for ten counter-examples, and an average of 28 assumptions returned to the user.

The remaining sections of this paper proceed as follows. Section II presents background material. Section III presents the generation of assumptions using a single counter-example. Section IV describes an overview of the extended assumption debugging methodology, while Section V present the details of the proposed work. Section VI presents experimental results and Section VII concludes this work.

## II. Preliminaries

### A. Minimal Correction Sets and Unsatisfiable Cores

For a given unsatisfiable (UNSAT) Boolean formula $\phi$ in conjunctive normal form (CNF), an *UNSAT core* is a subset of clauses of $\phi$ that are unsatisfiable. A *Minimal Unsatisfiable Subset* (MUS) is an UNSAT core where every proper subset is satisfiable (SAT). A *Minimal Correction Set* (MCS) is a minimal set of clauses of $\phi$ such that removing them will result in $\phi$ being SAT. There exists a duality relationship between

[1]University of Toronto, ECE Department, Toronto, ON M5S 3G4 ({briank, veneris, djordje}@eecg.toronto.edu)
[2]Vennsa Technologies Inc., Toronto, ON M5R 3L8 (evean@vennsa.com)
[3]University of Toronto, CS Department, Toronto, ON M5S 3G4

MUSs and MCSs such that, if one has all MUSs, then all MCSs can be computed and vice versa [10].

MCSs of $\phi$ can be computed by introducing a fresh *relaxation variable* to each clause. If the variable is active, then the clause is effectively removed from the problem. By additionally introducing cardinality constraints on these relaxation variables, one can find all minimal sets of relaxation variables which will result in $\phi$ being SAT. Each one of these solutions represents an MCS corresponding to the associated relaxation variables. This idea has been used extensively in modern Max-SAT solvers [11], [12] to compute MCSs as well as design debugging [3], [4] applications.

## III. Debugging Missing Input Constraints with a Single Counter-Example

In this section, we develop a methodology to quickly determine whether a candidate input constraint will prevent a failure from occurring. A naive way to detect this is to simply re-run the formal tool with the added candidate constraint. This can be very computationally intensive especially if multiple input constraint candidates need to be tested. Instead, we will generate an approximate solution to this process by generating a function that represents all MUSs with respect to the input unit clauses of the unrolled counter-example. Using this function, potential input constraints can be efficiently checked to ensure that they do not cause a failure in a similar manner to the given counter-example.

Consider the CNF formula $\phi$ of the time-frame expanded circuit and the corresponding counter-example:

$$\phi = S \cdot X \cdot T \cdot P \tag{1}$$

where $S$ represents the initial state, $X$ the counter-example input vector, $T$ the unrolled circuit transition relation, and $P$ the property to be checked. Since $\phi$ models the counter-example of the unrolled circuit, it is guaranteed to be UNSAT.

Instead of computing all MUSs for $\phi$ to generate our desired function, a less expensive computation can be performed by examining only the inputs clauses from $X$. The intuition here is that we are only concerned with missing input constraints, so it is unnecessary to perform extra computation for finding all MUSs not relating to inputs.

More precisely, we wish to extract all minimal* subsets of input unit clauses from $X$ (denoted by $U^k$ for the $k^{th}$ such set) such that $S \cdot T \cdot P \cdot U^k$ is UNSAT. This will allow us to build a function, $F$, that represents the disjunction of all MUSs with respect to the inputs, shown in the next equation:

$$F = U^0 + ... + U^k \tag{2}$$

Given a candidate input constraint, $A$, if $F \cdot A$ is SAT, then $A$ does not prevent the failure given in the counter-example since at least one of $U^k$ is SAT. Inversely, if $F \cdot A$ is UNSAT, then $A$ will ensure that future failures will not occur in the same way as the given counter-example. However in the latter case, $A$ may not constrain the input space enough to prevent all failures, but it at least prevents failures similar to those seen in the counter-example.

For the $i^{th}$ literal in $U^k$, denoted by $u_i^k$, Equation 2 can be expanded to give:

$$\begin{aligned} F =& u_0^0 u_1^0 ... u_{|U^0|}^0 + ... + u_0^k u_1^k ... u_{|U^k|}^k \\ =& \overline{(\overline{u}_0^0 + \overline{u}_1^0 + ... + \overline{u}_{|U^0|}^0) ... (\overline{u}_0^k + \overline{u}_1^k + ... + \overline{u}_{|U^k|}^k)} \end{aligned} \tag{3}$$

* Minimal in the sense that removing any clause from $U^k$ will make $S \cdot T \cdot P \cdot U^k$ become SAT.

Notice that when $F$ evaluates to false, at least one literal in each $U^k$ term is false. In other words, all $U^k$ MUSs can be broken by negating at least one literal from each term in $F$. Correspondingly, $\phi$ can be made SAT if at least one literal from each term in $F$ is negated for the respective unit clauses in $\phi$. Further, removing a minimal set of the corresponding unit clauses from the original problem will give an equivalent effect. Define this minimal set to be $V^k \subseteq X$ for the $k^{th}$ set.

The set $V^k$ can be thought of as the $k^{th}$ MCS with respect to the input literals. In fact, the relationship between the minimal subsets of inputs to make $\phi$ UNSAT ($U^k$), and the minimal subsets of inputs that need to be removed to make $\phi$ SAT ($V^k$), is analogous to the relationship between MUSs and MCSs.

Using this relationship and the fact that these sets only contain unit clauses, $F$ can be simplified further. Let the $i^{th}$ literal in $V^k \subseteq X$ be denoted by $v_i^k$. Equation 3 can be simplified, by distributing the conjunctions and removing redundant terms/literals, to:

$$F = \overline{\overline{v}_0^0 \overline{v}_1^0 ... \overline{v}_{|V^0|}^0 + ... + \overline{v}_0^k \overline{v}_1^k ... \overline{v}_{|V^k|}^k} \tag{4}$$

Now each term of Equation 4 contains the conjunction of the negated literals of each $V^k$. Thus to build the function $F$, one only needs to find all $V^k$.

This can be accomplished in a similar manner to computing all MCSs. Begin by adding a fresh relaxation variable to each clause in $X$. Using cardinality constraints, find all minimal SAT solutions with respect to these relaxation variables similar to the process used by modern Max-SAT solvers [11], [12]. Each such solution will correspond to a $V^k$. After all such solutions are found, construct a SAT instance of the form $F \cdot A$, where $A$ is the given input constraint to be checked. This instance checks whether $A$ can restrict the input space to prevent a failure similar to the one seen in the counter-example.

Although computing MCSs can be computationally intensive in general, the proposed method only calculates them with respect to the input unit clauses.

## IV. Assumption Debugging Flow

This section extends our methodology for debugging missing assumptions in a formal property checking environment. Although the overall flow is presented in the context of debugging missing input assumptions, as noted earlier, the work here can still be valuable in debugging other types of formal failures such as RTL design errors or incorrectly written assertions.

Let $C^k$ denote the $k^{th}$ *Minimal Correction Input Set* (MCIS), defined as a minimal set of input unit clauses of $X$ that when removed, will result in $\phi$ being SAT *i.e.,* $C^k$ is the minimal set of input clauses to remove to correct the failure. This is analogous to the idea of a MCSs except with respect to only input unit clauses.

Let $U^k$ denote the $k^{th}$ *Minimal Unsatisfiable Input Subset* (MUIS), defined as a minimal unsatisfiable set of input unit clauses such that $S \cdot T \cdot P \cdot U^k$, in Section III, is still UNSAT *i.e.,* $U^k$ is the minimal set of input clauses needed to expose the failure. Similarly, $U^k$ is analogous to MUSs except with respect to input unit clauses.

The overall methodology is shown in Figure 1 and consists of two major phases. Given an assertion failure and its associated counter-example, the *first phase* attempts to iteratively generate multiple formal counter-examples. For each counter-example, MCISs are extracted and used to generate constraints which are then passed back into the formal tool. These constraints ensure that another distinct counter-example is found. This process is repeated until the formal tool cannot

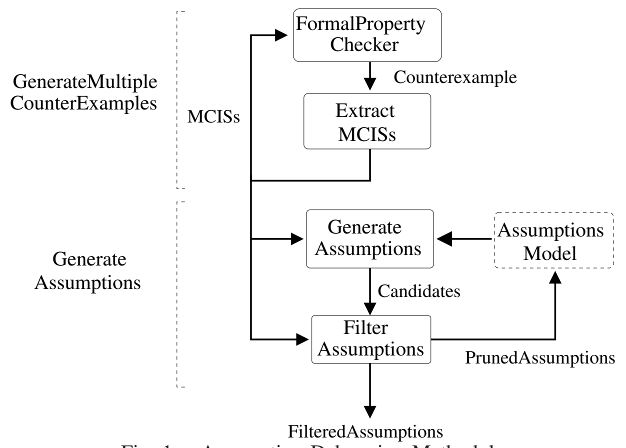

Fig. 1. Assumption Debugging Methodology

return any more counter-examples (*i.e.* the number of counter-examples produced lead to assumptions that fully constrain the undesired behaviour of the design), or a desired number of vectors has been reached.

The *second phase* of the methodology iteratively generates input assumptions that can prevent failures similar to those seen in the formal counter-examples. This is accomplished by using a model of simple assumption structures and filtering them based on the MCIS constraints extracted from the counter-examples. The resulting filtered assumptions are returned to the user and can either be used as suggestions for the missing assumptions, or as hints to which signals and expressions might be needed.

This flow improves debugging efficiency in two ways. First, multiple counter-examples can greatly improve debugging of formal failures regardless of their type. This is because they provide a more general representation of the assertion failure, which benefits both manual and automated debugging techniques [3], [4], [8]. In particular, they can greatly reduce the number of assumptions returned to the user, dramatically improving the quality of results. Second, the assumptions returned by the methodology improve upon previous work by generating easy-to-understand properties based upon common assumption structures. The next section describes the phases of our methodology in greater detail.

## V. Generating Multiple Counter-examples

Multiple counter-examples are beneficial for debugging because they allow a broader view of the root-cause of failure and may improve the resolution of automated debugging techniques [3], [8]. Despite the benefits of multiple counter-examples, to the best of the author's knowledge, existing formal property checkers do not support this feature.

The difficulty in this process is not simply generating a second counter-example, but rather generating a *useful* second counter-example that causes the assertion to fail in a different manner; *i.e.* a failure that propagates through a different path, and/or it isn't simply a time shifted instance of a preceding counter-example. The following sub-sections describe a method to generate multiple formal counter-examples that are quantitatively different from each other. It also outlines how to apply them to filter candidate input assumptions and improve quality of the final results.

### A. Minimal Correction Input Sets as Blocking Constraints

In the context of debugging, a failure can be viewed as a counter-example exciting an error, propagating its effect through design components, and causing an assertion to fail. This corresponds to the unrolled (in time) CNF of the counter-example from Equation 1. The initial states and input vector propagate through the clauses that model the design, and cause a conflict with the modeled property. The corresponding clauses can be abstractly viewed as a set of MUSs. As such, a natural way to quantify two counter-examples as being different is when the observed failures occur with no identical MUSs. This leads to the following definition of distinct counter-examples:

**Definition 1** *Given two counter-examples $R$ and $S$, and their respective unrolled CNF instances from Equation 1, $\phi_R$ and $\phi_S$, let $M_R$ and $M_S$ represent the set of all MUSs from $\phi_R$ and $\phi_S$, respectively. Counter-examples $R$ and $S$ are said to be* distinct *iff* $M_R \cap M_S = \emptyset$.

Using this definition, we can generate multiple distinct counter-examples by preventing previously seen MUSs from occurring again. To prevent a MUS, we need to ensure at least one of its clauses is not present. Since the circuit behavior should not change, only the clauses corresponding to the primary input vector should be blocked to prevent previously found MUSs. This corresponds directly to generating a blocking constraint on the inputs to prevent previously found MUISs. Using the duality between MUISs and MCISs, this constraint can be computed from a single MCIS.

In more detail, for the unrolled counter-example $\phi$ from Equation 1 and MCIS $C^k = \{c_0, ..., c_{|C^k|}\}$, removing $C^k$ will break all MUISs (and thus all MUSs) in $\phi$ since their removal will make the instance SAT. Since $C^k$ is minimal, this is equivalent to reversing the polarity of the corresponding input unit clauses in $\phi$, and it can be expressed as the following blocking constraint $B^k$ for the $k^{th}$ MCIS:

$$B^k = \overline{c_0^k} \cdot \overline{c_1^k} \cdot ... \cdot \overline{c_{|C^k|}^k} \tag{5}$$

This blocking constraint can be used in conjunction with the design and assertion to generate another distinct counter-example using an additional call to the formal tool. The following lemma describes this idea:

**Lemma 1** *For counter-example $R$, let $\phi_R$ be the unrolled counter-example in CNF. If $B^k$ is the $k^{th}$ blocking constraint of $\phi_R$, then any counter-example that satisfies $B_k$ is distinct from $R$.*

*Proof:* From Equation 5, $B^k$ is the conjunction of all the negations of the $k^{th}$ MCIS, $C^k$. By definition, removing the literals of $C^k$ from the $\phi_R$ will result in the instance being SAT, effectively breaking all the MUSs from $\phi_R$. Since $C^k$ is minimal and no proper subset has the property of being a correction set, any SAT assignment will contain the negation of all the literals of $C^k$, precisely the expression $B^k$. It follows then that any counter-example that contains the assignment from $B^k$ will necessarily not contain any MUSs from $\phi_R$, and therefore is distinct. ■

As such, to generate a new distinct counter-example we can use Lemma 1 and pass a blocking constraint in the form of Equation 5 to the formal tool. If the formal tool returns a counter-example, it implicitly guarantees that $B^k$ is satisfied, resulting in a distinct counter-example.

### B. A Practical Algorithm

Algorithm 1 shows the pseudo-code for generating multiple counter-examples. The algorithm begins by generating the first counter-example from the formal property checker and extracting all MCISs from it (lines 2-5). The loop from line 6-14 generates multiple counter-examples. For a given MCIS, it will attempt to find a new counter-example. If successful

**Algorithm 1** Generating Multiple Counter-Examples

```
 1: procedure MULTIPLECOUNTEREXAMPLES($max$)
 2:     $blocking = \emptyset$
 3:     c-ex = RUNFORMAL($blocking$)
 4:     CEX = {c-ex}
 5:     MCIS = EXTRACTALLMCIS(c-ex)
 6:     while MCIS $\neq \emptyset$ and —CEX— $< max$ do
 7:         $C$ =EXTRACTBLOCKING(MCIS)
 8:         c-ex = RUNFORMAL($blocking \cup C$)
 9:         if c-ex $\neq \emptyset$ then
10:             $blocking = blocking \cup C$
11:             CEX = CEX $\cup$ c-ex
12:             MCIS = EXTRACTALLMCIS(c-ex)
13:         end if
14:     end while
15:     return CEX
16: end procedure
```

(line 9), this MCIS is saved in $blocking$ to ensure that future counter-examples remain distinct. This process greedily selects new MCISs to add to $blocking$ only when it can find a new counter-example. Once the new counter-example is saved, a new set of MCISs are extracted (lines 11-12), and the process repeats using this new set of MCISs. The loop stops when either none of MCISs combined with the existing constraints in $blocking$ can generate another counter-example, or when the maximum number of user-specified counter-examples has been reached. The following theorem confirms the benefits of these counter-examples:

**Theorem 1** *All counter-examples returned by Algorithm 1 are mutually distinct.*

*Proof:* Each counter-example generated from the run of the formal tool on line 8 will run under the set of blocking constraints, $blocking \cup C$. By Lemma 1, any counter-examples that derived any of the constraints in $blocking \cup C$ will be distinct from the newly generated one. Since blocking constraints are added only when a new counter-example is found, $blocking \cup C$ maintains a set of MCISs from each of the previously seen counter-examples. Therefore, each newly generated counter-example will respect these blocking constraints and it will be mutually distinct. ■

One important aspect of Algorithm 1 is that it iteratively adds a blocking constraint in the form of Equation 5. This could have alternatively been implemented using the disjunction of all blocking clauses from a single counter-example *i.e.,* the negation of Equation 4. However, our experience with an industrial formal property checker shows that this latter approach significantly slows down the tool causing time-outs or bounded proofs, an observation that can be explained as follows. Our blocking constraints are just unit clauses, which are easily modeled within many different model checking algorithms. Whereas, the disjunction of multiple MCISs can be significantly more complicated to model (or at least require specialized optimizations). This allows for a more generic method without any need to use a specialized property checker.

### C. Applications for Debugging Missing Input Assumptions

As mentioned in Section III, a single counter-example can be used to filter a candidate assumption $A$. The filtering function $F$ used to rule out candidate assumptions is derived directly from a set of MCISs. If Algorithm 1 is used to derive multiple counter-examples, all MCISs from each counter-example are indirectly generated as a by-product. These can be used to generate a set of filtering functions $F_1, ..., F_N$ for $N$ counter-examples, respectively, which can naturally be combined to extend the filtering function and generate the following instance:

$$\psi = (F_1 + ... + F_N) \cdot A \tag{6}$$

The disjunction of all the $F_i$ in Equation 6 correspond to all the MUISs for each of the counter-examples. This implicitly encodes all the input behaviors that led to the observed assertion failures in the given counter-examples. If the instance is SAT, then the assumption is not strong enough to prevent at least one of the observed failures. Otherwise, the assumption is generalized enough to prevent all the observed failures and should be returned to the user.

It should be noted that although we present this filtering function in the context of pruning generated assumptions, it is equally valid to say that this function can be used to test manually generated assumptions by the engineer. Thus, the filtering function can provide quick feedback to determine if a given assumption can prevent the failure(s) present in the current counter-example(s).

TABLE I
ASSUMPTION MODEL

| Category | Model |
|---|---|
| Unit Booleans (`unit`) | `input, !input` |
| Combined Booleans | `<unit> & <unit>,` `<unit> & <unit> & <unit>,` `<unit> \| <unit>,` `<unit> \| <unit> \| <unit>` |
| One-hot | `$onehot(bus), $onehot0(bus),` `$onehot({<unit>, <unit>}),` `$onehot({<unit>, <unit>, <unit>})` |
| Stability | `$stable(bus), bus == 0` `input \|=> !input, !input \|=> input` |

TABLE II
DESIGN INFORMATION

| Design Name | # Gates (k) | # Flops | # Inputs |
|---|---|---|---|
| cpu | 50.9 | 1270 | 51 |
| ddr2 | 55.5 | 2475 | 431 |
| hpdmc | 9.8 | 431 | 210 |
| mips | 51.1 | 2250 | 82 |
| mrisc | 9.9 | 1372 | 69 |
| pci | 60.3 | 3886 | 162 |
| spi | 1.7 | 133 | 16 |
| usbf | 33.2 | 1954 | 128 |
| wb | 4.0 | 98 | 143 |

### D. Assumption Model

Table I shows a summary of the model used to generate candidate missing input assumptions as in Section III. Each row corresponds to one of four categories of properties presented in SystemVerilog. These categories correspond to simple unit Booleans, combined Boolean operators, one-hot operators, and stability expressions. An assumption is generated by taking the property and using the same clock and reset as the target failing assertion. Each assumption is then checked against the filtering function to determine if it should be returned to the user. In the table, `input` refers to a single bit primary input pin, while `bus` refers to a semantic grouping of primary input pins.

## VI. EXPERIMENTAL RESULTS

This section presents experimental results for the proposed methodology. All experiments are performed on a single core of an Intel Core i5 3.1 GHz quad-core workstation with 16 GB of RAM. A commercial property checker [13] is used

TABLE III
AUTOMATED GENERATION OF MISSING CONSTRAINTS EXPERIMENTAL RESULTS

| **instance info** | | | | **algorithm** | | | **check** | | |
|---|---|---|---|---|---|---|---|---|---|
| instance name | # gates | # states | c-ex len | time (s) | cand | filter | time (s) | passing | vacuous |
| hpdmc1 | 9794 | 430 | 13 | 25 | 211 | 29 | 716 | 11 | 2 |
| hpdmc2 | 9794 | 430 | 12 | 58 | 325 | 45 | 984 | 1 | 5 |
| hpdmc3 | 9794 | 430 | 2 | 1 | 14 | 5 | 46 | 3 | 1 |
| spi1 | 1724 | 132 | 4 | 1 | 40 | 10 | 80 | 1 | 8 |
| spi2 | 1724 | 132 | 21 | 4 | 82 | 40 | 169 | 0 | 10 |
| ddr1 | 55069 | 2474 | 9 | 248 | 310 | 20 | 3477 | 0 | 0 |
| ddr2 | 55069 | 2474 | 6 | 42 | 180 | 20 | 1869 | 0 | 0 |

with default settings to perform all formal checks, while the extraction of MCISs as well as generation and filtering of candidate assumptions are all implemented in C++, using Minisat [14] as the SAT engine. Nine designs are selected for evaluation from OpenCores [9] with assertions written based upon their specification documents.

The formal property checker is run on each design and any failure is considered to be an instance of a missing assumption. The instances listed in the following tables correspond to a single failing assertion and are labeled by appending a number to the design name. Table II presents relevant statistics and a description for each of the designs used in the experimentals.

### *A. Debugging Missing Input Constraints with a Single Counter-Example*

Using these instances, our experimental methodology proceeds as follows. First, for each failing assertion, a counter-example is generated using a formal property checker. Next, the proposed approach from Section III uses the counter-example to generate a filtered list of missing constraints. Minisat [14] is used to solve all SAT instances, including generating the filtering function $F$. Finally, to check if any of the generated properties can be used as actual missing constraints, each property is re-run in a separate formal check with the original failing assertion. The comprehensive results for each instance are shown in Table III.

The first four columns of Table III show the instance name, number of gates, number of state elements, and counter-example length. Column 5 lists the overall run-time of the proposed approach, including creating the function $F$ as well as filtering. Column 6 lists the amount of assumptions generated, and Column 7 lists the number remaining after filtering with function $F$. From the filtered list, the last three columns show the total run-time, number of non-vacuous passing instances and vacuous passing instances when re-running all generated constraints separately with the formal tool.

Overall, the results in Table III show that the filtering function can significantly reduce the number of candidates constraints from an average of 166 properties in column 6, down to an average of 24 in column 7 after filtering. Moreover, this is done with relatively little run-time making it ideal for fast analysis for use when debugging missing constraints. Compared to running each generated constraint in a separate formal check (column 8), the proposed method shows a 33.4x speedup on average. The last two columns show that in certain cases (e.g. hpdmc and spi), the simple properties can generate an exact constraint to prevent the failing assertion. Although in the case of ddr, none of the generated properties are able to prevent the failing assertion.

### *B. Generating Multiple Counter-Examples*

This subsection presents experimental results for the proposed approach to generate multiple counter-examples from Section V. Experiments in this subsection are conducted for each instance by, first, running Algorithm 1 to generate as many counter-examples as possible within 1800 seconds to a maximum of 15. Next, using either 1, 5, 10, or 15 counter-examples, candidate assumptions are generated and filtered to examine if multiple counter-examples are useful in reducing the number of generated assumptions. To simplify the experiments, combined and one-hot type properties from Table I are omitted when generating assumptions. Additionally, each pin of an input bus is also used in unit Boolean properties so that we have a sufficient set of properties across all input pins. Note that a cone of influence [6] optimization is run on the failing assertion of each instance, resulting in a potentially different number of total generated assumptions between instances of the same design. Table IV shows the results of these experiments.

TABLE IV
MULTIPLE COUNTER-EXAMPLE EXPERIMENTS

| Instance Name | # CE | MCIS Time (s) | Form Time (s) | Tot Can | Filt Using $n$ CE | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | 1 | 5 | 10 | 15 |
| cpu_1 | 15 | 653 | 356 | 154 | 2 | 2 | 2 | 2 |
| cpu_2 | 10 | 778 | 616 | 154 | 3 | 3 | 3 | - |
| ddr2_1 | 3 | 625 | 86 | 226 | 68 | - | - | - |
| ddr2_2 | 9 | 383 | 1395 | 257 | 15 | 2 | - | - |
| hpdmc_1 | 15 | 112 | 148 | 97 | 17 | 16 | 11 | 11 |
| hpdmc_2 | 15 | 123 | 200 | 97 | 25 | 16 | 13 | 11 |
| mips_1 | 4 | 278 | 93 | 163 | 36 | - | - | - |
| mips_2 | 12 | 813 | 959 | 163 | 13 | 13 | 13 | - |
| mrisc_1 | 8 | 88 | 1126 | 92 | 11 | 5 | - | - |
| mrisc_2 | 7 | 190 | 1339 | 92 | 8 | 8 | - | - |
| pci_1 | 8 | 611 | 761 | 267 | 9 | 9 | - | - |
| pci_2 | 8 | 648 | 723 | 267 | 11 | 11 | - | - |
| spi_1 | 15 | 8 | 177 | 22 | 6 | 6 | 6 | 6 |
| spi_2 | 15 | 47 | 214 | 22 | 19 | 10 | 7 | 7 |
| usbf_1 | 12 | 901 | 858 | 131 | 61 | 28 | 26 | - |
| usbf_2 | 10 | 186 | 1152 | 131 | 22 | 0 | 0 | - |
| wb_1 | 15 | 6 | 846 | 13 | 9 | 7 | 6 | 6 |
| wb_2 | 15 | 8 | 344 | 41 | 10 | 2 | 2 | 2 |

The first five columns list the instance name, number of counter-examples generated, run-time to extract MCISs from the counter-examples, run-time of the formal tool to generate that many counter-examples, and the total number of candidate assumptions for that instance. The last four columns show how many of the candidate assumptions remain after filtering using the technique from Section V-C with 1, 5, 10, and 15 counter-examples, respectively.

Overall the last four columns show that using more counter-examples can effectively reduce the number of filtered assumptions. On average, for instances that are able to generate either 5, 10, or 15 counter-examples, the number of filtered assumptions are reduced by 30.4%, 37.9% and 38.3%, respectively, compared to a single counter-example. This confirms that the additional counter-examples generated result in assertion failures which can be blocked using a smaller assumption set.

The ability of the proposed technique to filter candidate assumptions works well in most of the instances (such as ddr2_2 and usbf_1), but not all (such as cpu_1 and

TABLE V
ASSUMPTION DEBUGGING METHODOLOGY EXPERIMENTS

| Instance Name | # CE | MCIS Time (s) | Form Time (s) | Gen Time (s) | Tot Can | Filt Can | Instance Name | # CE | MCIS Time (s) | Form Time (s) | Gen Time (s) | Tot Can | Filt Can |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cpu_1 | 10 | 255 | 100 | 5 | 31 | 3 | mrisc_4 | 9 | 116 | 898 | 5 | 39 | 14 |
| cpu_2 | 10 | 778 | 616 | 7 | 28 | 5 | pci_1 | 8 | 611 | 761 | 7 | 25 | 10 |
| ddr2_1 | 3 | 625 | 86 | TO | 857 | 21 | pci_2 | 8 | 648 | 723 | 7 | 22 | 11 |
| ddr2_2 | 9 | 383 | 1395 | 1504 | 4094 | 333 | pci_3 | 8 | 564 | 518 | 8 | 25 | 10 |
| hpdmc_1 | 10 | 70 | 60 | 4 | 90 | 33 | pci_4 | 2 | 466 | 60 | 27 | 261 | 82 |
| hpdmc_2 | 10 | 77 | 65 | 8 | 65 | 18 | spi_1 | 10 | 4 | 48 | 1 | 20 | 9 |
| hpdmc_3 | 10 | 6 | 77 | 1 | 8 | 3 | spi_2 | 10 | 28 | 60 | 15 | 74 | 29 |
| mips_1 | 4 | 278 | 93 | 9 | 59 | 22 | usbf_1 | 10 | 737 | 334 | 14 | 148 | 58 |
| mips_2 | 10 | 455 | 276 | 8 | 39 | 10 | usbf_2 | 10 | 186 | 1152 | 132 | 1135 | 44 |
| mips_3 | 5 | 134 | 458 | 7 | 39 | 6 | usbf_3 | 10 | 18 | 244 | 2 | 16 | 7 |
| mips_4 | 10 | 589 | 631 | 10 | 59 | 7 | wb_1 | 10 | 3 | 123 | 1 | 16 | 5 |
| mrisc_1 | 8 | 88 | 1126 | 5 | 39 | 10 | wb_2 | 10 | 4 | 111 | 1 | 81 | 2 |
| mrisc_2 | 7 | 190 | 1339 | 6 | 34 | 9 | wb_3 | 10 | 5 | 79 | 1 | 19 | 2 |
| mrisc_3 | 5 | 79 | 169 | 4 | 20 | 9 | wb_4 | 10 | 4 | 98 | 1 | 81 | 2 |

mips_2). This can be explained as follows. The former case is the ideal behavior where the second counter-example does indeed find a different way to excite the design and cause the assertion to fail. While the latter case finds a counter-example similar to the original one but shifted in time.

When analyzing the run-time, there are two main contributors. The first is the extraction of MCISs, which depends on the size of the design, number of input pins, and the length of the counter-example. For many cases, such as mrisc_1 and wb_1, it is relatively fast. In the case of ddr2_1, however, the excessive number of inputs (431) cause the run-time of extracting the MCISs to be large. The other contributor to overall run-time is multiple iterations of the loop in Algorithm 1, which may require many calls to the formal tool before a counter-example is found. However within the 1800 second timeout, 7 out of the 18 instances were able to generate 15 counter-examples, and 16 out of the 18 were able to generate at least 5. This shows that this technique is effective in generating multiple counter-examples within a short amount of time.

## C. Assumption Debugging Methodology

This section presents experimental results for the overall assumption debugging methodology from Section IV. For each instance, counter-examples are generated within a time limit of 1800 seconds up to a maximum of 10. Additionally, the full assumption model from Section V-D is used to generate and filter assumptions. Note that this may result in a different number of candidate assumptions compared to the previous subsection. Similarly, a cone of influence [6] optimization is run on the failing assertion for each instance. Table V shows the quantitative results of these experiments.

Table V is divided into two parallel sections. The columns in each section list the instance name, number of counter-examples generated, time to extract MCISs from the counter-examples, time of the formal tool to generate that many counter-examples, time to generate and filter candidate assumptions, total number of candidate assumptions, and the number of assumptions after filtering.

From columns 7 and 14, the absolute number of filtered assumptions returned to the user is relatively small with an average of 28. It is important that this number is not too large, or else the list of assumptions may become unwieldy for a user to analyze. Although most of the instances fall close to this average, there is one outlier ddr2_2 with 333 returned assumptions. This is due to the large number of input pins which generates a significant number of candidate assumptions (4094). However, the different categories of properties allow one to narrow down the analysis. In this case, only analyzing the unit Booleans assumptions proved most useful.

When analyzing run-time of generating and filtering candidates in columns 5 and 12, in most instances the time is relatively small and both tasks can be completed within 60 seconds. However, ddr_1 and ddr_2 are again outliers, where the former hit a time limit of 1800 seconds. Here, the excessive number of input pins cause an exponential number of generated assumptions in the more complex properties. In these cases, it may be prudent to generate simpler properties or limit the number of pins used to generate assumptions.

## VII. CONCLUSION

In this work, a novel debug automation methodology for missing input assumptions is presented. It begins by generating multiple formal counter-examples for the failure along with a function that encodes the input combinations that caused the assertion to fail. This function is later used to generate a list of fixed cycle assumptions that prevent the failures seen in the counter-examples, which can then be used as hints for the actual missing assumption. An extensive set of experimental results show the efficiency of this work.

## REFERENCES

[1] H. Foster, "Applied assertion-based verification: An industry perspective," *Foundations and Trends in Electronic Design Automation*, vol. 3, no. 1, pp. 1–95, 2009.
[2] S. Huang and K. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publisher, 1998.
[3] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.
[4] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Trans. on CAD*, vol. 27, no. 6, pp. 1138–1149, 2008.
[5] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*. Kluwer Academic Publishers, 2003.
[6] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.
[7] A. Matsuda. (2011, May.) Overcoming the challenges of formal verification and debug. [Online]. Available: http://www.eetimes.com/design/eda-design/4216119/Overcoming-the-challenges-of-formal-verification-and-debug
[8] B. Keng, S. Safarpour, and A. Veneris, "Automated debugging of SystemVerilog assertions," in *Design, Automation and Test in Europe*, 2011, pp. 323–328.
[9] OpenCores.org, 2007. [Online]. Available: http://www.opencores.org
[10] M. H. Liffiton and K. A. Sakallah, "On Finding All Minimally Unsatisfiable Subformulas," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2005, pp. 173–186.
[11] J. Marques-Silva and J. Planes, "Algorithms for maximum satisfiability using unsatisfiable cores," in *Design, Automation and Test in Europe*, 2008, pp. 408–413.
[12] M. H. Liffiton and K. A. Sakallah, "Generalizing Core-Guided Max-SAT," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2009, pp. 481–494.
[13] Cadence Design Systems, "Incisive Formal Verifier," 2012. [Online]. Available: http://www.cadence.com/products/ld/formal_verifier/pages/default.aspx
[14] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.