Automated Debugging of Missing Assumptions

Brian Keng[1], Evean Qin[3], Andreas Veneris[1,2], Bao Le[1]

***Abstract*—Formal verification has increased efficiency by detecting corner case design bugs but it has also 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. Previous work in debug automation has focused on the former two cases. This paper introduces a novel methodology to automatically debug missing assumptions. It begins by generating multiple formal counter-examples for the error. Next, 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. An extensive set of experimental results on OpenCores designs and assertions show that the number of generated assumptions can be reduced by an average of 38% using ten counter-examples, while an average of 28 assumptions is returned to the user.**

I. Introduction

Functional debugging today has become a bottleneck taking up 60% of the total verification time [1]. To cope with this burden, 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 exhaustively verifying an assertion encoding 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 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. Despite the need for designer intervention to determine the actual root-cause of the failure, previous work [4], [8] in debug automation has shown to be effective in aiding the designer in the former case. Today, diagnosing missing assumptions still poses a significant bottleneck in formal verification flow as they can cause up to 50% of the formal failures [7].

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. 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 multiple time-consuming calls to the formal tool. To alleviate this pain and make formal technology effective to its full potential, more debug automation is needed to help analyze the behavior of the counter-example and identify candidate missing assumptions.

Identifying missing assumptions has been researched before in the context of compositional verification, software model checking and reactive system synthesis [9]–[11]. More recently, a technique to tackle this problem in a hardware formal verification context has shown promising results [12]. Given a counter-example due to a missing assumption, this technique generates a list of fixed cycle assumptions that prevent the assertion failure under conditions similar to the original counter-example. In essence, the *goal* for these added assumptions is to be used as hints by the engineer to identify 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. Results from that paper [12] on a handful of instances are encouraging despite their dependence on a single counter-example, a fact which may severely limit the quality of these generated assumptions.

In this work, we present an automated assumption debugging methodology. The proposed work is based upon the framework developed in [12] but overcomes its limitations as it provides higher quality assumptions that can be used during debugging. It also complements other debugging techniques as it provides additional information through new counter-examples. In detail, the contributions of the work are twofold. At first, a novel algorithm is presented 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. As an added benefit, 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]. Next, a method of using multiple distinct counter-examples to improve the quality of results of the assumption debugging methodology is also presented.

An extensive set of experiments is performed over a wide variety of `OpenCores` [13] designs with SystemVerilog assertions written from their specification documents. Multiple counter-examples are shown to reduce the number of generated assumptions by 38% on average while an average of 28 assumptions are returned to the user. This confirms the benefit of the proposed approach.

The remaining paper is as follows. Section II presents background material. Section III describes an overview of the assumption debugging methodology, while Section IV present the details of the proposed work. Section V presents the experiments and Section VI contains the conclusion.

II. Preliminaries

A. Minimal Correction Sets and Unsatisfiable Cores

For a given unsatisfiable (UNSAT) Boolean formula ϕ in conjunctive normal form (CNF), an *UNSAT core* is a subset of clauses of ϕ 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

[1]University of Toronto, ECE Department, Toronto, ON M5S 3G4 ({briank, veneris, lebao}@eecg.toronto.edu)

[2]University of Toronto, CS Department, Toronto, ON M5S 3G4

[3]Vennsa Technologies, Inc., Toronto, ON M5V 3B1 (evean@vennsa.com)

minimal set of clauses of ϕ such that removing them will result in ϕ being SAT. There exists a duality relationship between MUSs and MCSs such that, if one has all MUSs, then all MCSs can be computed and vice versa [14].

MCSs of ϕ 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 ϕ 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 [15], [16] to compute MCSs as well as design debugging [3], [4] applications.

B. Minimally Unsatisfiable Input Sets and Debugging Missing Assumptions

Let I represent the initial state, X the counter-example input vector, T the unrolled circuit transition relation, and P the property to be checked. The unrolled counter-example in CNF, denoted by ϕ, is given by:

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

Equation 1 is UNSAT by construction because the counter-example exposes an assertion failure.

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 ϕ 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 $I \cdot T \cdot P \cdot U^k$ 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.

Each U^k represents a minimal set of input combinations from the counter-example that can directly excite the assertion failure. The disjunction of all U^k represents all combinations of inputs from the counter-example that can lead to the assertion failure, given by:

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

As shown in [12], this function can be computed by first calculating all C^k using relaxation variables, and then building F using a duality relationship analogous to the one between MUSs and MCSs. Thus, F can be re-written in terms of C^k, where c_i^k represents the i^{th} input unit clause of C^k:

$$F = \overline{c_0^0} \cdot ... \cdot \overline{c_{|C^0|}^0} + ... + \overline{c_0^k} \cdot ... \cdot \overline{c_{|C^k|}^k} \tag{3}$$

Given an assertion failure due to missing assumptions and its associated counter-example, the technique in [12] uses the function from Equation 3 to aid in debugging missing assumptions. This is accomplished in several steps. First, F from Equation 3 is extracted from the given counter-example. Next, a list of candidate fixed-cycle assumptions on primary inputs are generated from a dictionary model. Each candidate input assumption, A, can be filtered out by creating a SAT instance, ψ, as such:

$$\psi = F \cdot A \tag{4}$$

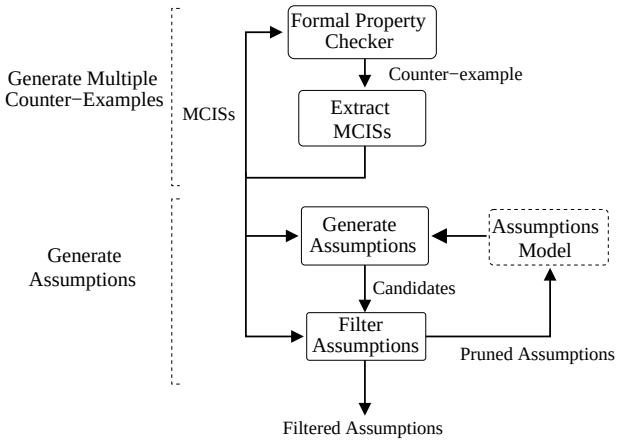

Fig. 1. Assumption Debugging Methodology

If ψ is SAT, then the assumption cannot prevent the assertion failure in the given counter-example (implicitly encoded in F) and should be pruned. Otherwise, A prevents a failure similar to the given counter-example and can be returned to the user. The end result is a list of assumptions that can prevent failures similar to those by the given counter-example, allowing the engineer to either potentially use the assumption directly, or build a strong intuition for the actual missing assumption.

III. Assumption Debugging Flow

This section presents an overview of 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. This is because a key by-product of our flow (*i.e.,* generating multiple formal counter-examples) can be utilized by "traditional" automated techniques [3], [4], [8] to aid debugging for all types of failures.

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 return any more counter-examples, or a desired number of vectors has been reached.

The *second phase* iteratively generates input assumptions that can prevent failures similar to those seen in the existing 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 because they provide a more general representation of the assertion failure, benefiting both manual and automated debugging [3], [4], [8]. 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 detail.

IV. Generating Multiple Counter-examples

In a typical formal flow when an assertion fails, the formal tool generates a single counter-example to be used later in debugging. 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, existing formal property checkers do not support this feature. In the past, there has been some work to generate diverse SAT solutions [17], although there is guarantee as to how different the counter-example will be.

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. 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 result.

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, ϕ_R and ϕ_S, let M_R and M_S represent the set of all MUSs from ϕ_R and ϕ_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 ϕ from Equation 1 and MCIS $C^k = \{c_0, ..., c_{|C^k|}\}$, removing C^k will break all MUISs (and thus all MUSs) in ϕ 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 ϕ, 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 ϕ_R be the unrolled counter-example in CNF. If B^k is the k^{th} blocking constraint of ϕ_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 ϕ_R will result in the instance being SAT, effectively breaking all the MUSs from ϕ_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 ϕ_R, and therefore it 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 (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 11-7), 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 3. 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.

Algorithm 1 Generating Multiple Counter-Examples

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

C. Applications for Debugging Missing Input Assumptions

As mentioned in Section II-B, a single counter-example can be used to filter a candidate assumption A using Equation 4. 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. Similar to Equation 4, 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).

D. Assumption Model

Table I shows a summary of the model used to generate candidate missing input assumptions that is similar to [12]. 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.

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

V. 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 [18] is used 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` [19] as the SAT engine. Nine designs are selected for evaluation from OpenCores [13] with assertions written based upon their specification documents.

For each of the gathered designs and assertions, the formal property checker is run 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 information for each of the designs used in the experimental results. The columns of the table list the design name, number of gates (including state elements), number of state elements, and number of primary input pins.

A. Generating Multiple Counter-Examples

This subsection presents experimental results for the proposed approach to generate multiple counter-examples from Section IV. Experiments in this subsection are conducted for each instance by 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

TABLE III
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

between instances of the same design. Table III shows the results of these experiments.

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 IV-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 do generalize the assertion failure, which shows that our definition of distinct is indeed a valid one.

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 `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. One needs to note that in this situation, it is often the case that there may only be one way to cause the assertion fail.

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.

B. Assumption Debugging Methodology

This section presents experimental results for the overall assumption debugging methodology from Section III. For each instance, counter-examples are generated within a time limit of 1800 seconds up to a maximum of 10, which was chosen qualitatively to be a good balance between filtering and run-time. Additionally, the full assumption model from Section IV-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 IV shows the quantitative results of these experiments.

Table IV 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 overwhelming 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 as described in Section V-C, 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 more prudent to only generate simpler properties or limit the number of pins used to generate assumptions. Both these solutions are easily implementable within the proposed flow.

C. Qualitative Analysis of Generated Assumptions

The following is a detailed discussion on the qualitative aspects of how the proposed flow can be used to aid debugging of missing assumptions. In the proposed flow, the engineer will analyze the assumptions and decide which assumption is appropriate. We selected two instances from Table IV to illustrate the benefits.

a) mips_1: The assertion for this instance checks to see that a finite state machine transition occurs with the correct conditions:

```
P: (CurrState == 'IDLE) && (irq && ~iack)
   |=> (CurrState == 'IRQ)
```

The approach generated 22 assumptions, a sample of which is listed here:

```
A1: !pause
A2: pause
A3: $onehot(zz_ins_i[31:0])
A4: $onehot0(zz_ins_i[31:0])
```

In this case, both `pause` and its negation are suggested by the technique. This is because holding pause either high or low for

TABLE IV
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

the entire trace will prevent the assertion from failing. This is a common occurrence in many of the generated assumptions, however, usually only one of the two stuck-at assumptions will be relevant.

By tracing the relationship between `pause` and the state machine, it is clear that when `pause` is asserted, the state transition will be stopped. In this case, `A1` is precisely the needed constraint. Interestingly, the specification does not explicitly mention that the state transition will be stopped by the `pause` signal, a common omission that causes counter-examples due to missing assumptions.

Finally, the last two assumptions are on the primary input bus corresponding to the CPU's instruction. They are obviously not very meaningful and correspond to a vacuous fix (*i.e.,* a case where the antecedent is always false).

b) usbf_1: The assertion for this instance checks the property where a buffer overflow occurs when a packet has been received that does not fit into the buffer. The packet will be discarded and a `NACK` will be sent to the host.

```
P: buffer_overflow ##0 send_token[->1]
   |-> (token_pid_sel == NACK)
```

The approach generated 58 assumptions, several of which are listed here:

```
A1: !wb_stb_i
A2: !wb_cyc_i
A3: !wb_addr_i[17] & wb_cyc_i & wb_stb_i
A4: $stable(DataIn_pad_i[7:0])
```

The first two assumptions that pull down `wb_stb_i` and `wb_cyc_i` are vacuous fixes. But the assumptions with both of these signals high (`A3`) are more interesting. In this assumption, the 17^{th} bit in `wb_addr_i` controls the source of the data to be sent. This assumption tells the user that during the assertion, the data should be selected from the register file instead of memory, avoiding the cause for the failure.

The next assumption `$stable(DataIn_pad_i[7:0])` provides a useful hint. This signal controls which data will be selected from the endpoint. It tells the user that during the assertion, the same endpoint should be selected, providing another way to avoid the failure.

VI. 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 on OpenCores designs and assertions show the efficacy and usability of the approach in an industrial formal verification environment.

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-challen%ges-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] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu, "Learning assumptions for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2003, pp. 331–346.

[10] S. Joshi, S. K. Lahiri, and A. Lal, "Underspecified harnesses and interleaved bugs," in *Principles of Programming Languages*, 2012, pp. 19–30.

[11] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *Int'l Conf. on Formal Methods and Models for Codesign*, 2011.

[12] B. Keng and A. Veneris, "Automated debugging of missing input constraints in a formal verification environment," in *Formal Methods in CAD*, 2012.

[13] OpenCores.org, 2007. [Online]. Available: http://www.opencores.org

[14] 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.

[15] J. Marques-Silva and J. Planes, "Algorithms for maximum satisfiability using unsatisfiable cores," in *Design, Automation and Test in Europe*, 2008, pp. 408–413.

[16] 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.

[17] A. Nadel, "Generating Diverse Solutions in SAT," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2011, pp. 287–301.

[18] Cadence Design Systems, "Incisive Formal Verifier," 2012. [Online]. Available: http://www.cadence.com/products/ld/formal_verifier/pages/default.aspx

[19] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.