Automated Debugging of Missing Input Constraints in a Formal Verification Environment

Brian Keng
Dept. of Elec. and Comp. Eng.
University of Toronto
Toronto, Canada

Andreas Veneris
Dept. of Comp. Sci & Elec. and Comp. Eng.
University of Toronto
Toronto, Canada

***Abstract*—In the past decade, formal tools have increased functional verification efficiency by exhaustively searching for hard to find bugs. Often the counter-examples returned are not due to design bugs but due to missing constraints that are needed to model the surrounding environment. These types of false positives have become a great concern in the industry today. To address this issue, input constraints are typically added by the engineer to restrict the input space a formal tool is allowed to explore. These constraints are difficult to generate as they are usually implicit in the documentation or implementation of adjacent design blocks. As a consequence, this process reduces the efficiency of formal methodologies because missing input constraints must be determined before deep design bugs can actually be detected.**

In this work, we present an algorithm to automatically generate missing input constraints given a failing counter-example. The process begins by building a filtering function that models the failing behaviors from the counter-example. Next, using this function a list of fixed cycle properties are generated and filtered to return a set of candidate input constraints for use in debugging. Preliminary experimental results show that the generated properties provide a strong intuition as to what input constraints may be missing.

I. Introduction

Functional verification is one of the most time consuming steps in the VLSI design flow taking up to 46% of the total design time [1]. To ease this growing burden, new tools and technologies have been developed such as assertion-based verification (ABV). ABV has shown to improve observability and increase overall verification efficiency. Along with traditional simulation-based techniques, modern ABV flows make wide use of formal technologies.

Formal methods allow a user to exhaustively explore the state space of a design in an attempt to find corner case counter-examples that elude traditional simulation-based verification. In formal property checking, a design block is verified against a precisely defined formal property written in an assertion language such as SystemVerilog Assertions (SVA) or Property Specification Language (PSL). As such, when a formal verifier returns a counter-example, the expectation is that a design bug has been detected. Although ideal, reports from the industry indicate that many failures are due to missing constraints from the surrounding environment and not because of design errors [2]. In the context of this work, we refer to such a situation as a *false positive*. These false positive are typically caused by missing constraints that are built into the environment but not explicitly documented. This results in formal tools reporting a failure when, in fact, the design may work as intended for the given environment.

To solve this issue, constraints in the form of formal properties are added by the engineer to restrict the space in which the formal tool can explore. The purpose of these constraints is to precisely model the restricted input space allowing the formal tool to find "real" design bugs. However, this presents a large debugging challenge to the engineer who is asked to play a guessing game as to which constraints need to be added. Adding to this overhead, often these constraints are implicitly specified in the documentation or implementation of adjacent design blocks. In many cases, the time-consuming manual process needed to identify these missing input constraints dominates the formal verification process leading to reduced efficiency.

This situation of generating constraints has also appeared in other contexts. During constrained random simulation, the work in [3] automatically generates constraint properties to bias the stimulus generator towards missing coverage holes. In compositional verification [4], a key step is generating assumption properties in order to verify the correctness of components separately. Previous work [5]–[7] aims to automatically generate an assumption on the interface between two components with the goal of proving the target property. Additionally, generating environmental constraints for software model checking [8], [9] and reactive system synthesis [10], [11] have also been studied. In these situations, the techniques effectively generate constraints to accomplish their respective goals. However, none of them addresses the wide-spread pain of debugging missing input constraints in a formal hardware verification flow.

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. The benefit of these generated constraints is twofold. First, the constraints are generated efficiently from the counter-example without the need to re-run the entire formal flow thus providing feedback quickly in the verification cycle. Further, the constraints are in the form of simple properties that can aid debugging by either being

978-0-9835678-2-0/12/$31.00 ©2012 FMCAD Inc.

directly used for the actual missing constraint(s), or indirectly used to give intuition about the failure. The key insight is that the engineer cannot be taken out of the debugging loop entirely. Instead, the algorithm aims to efficiently return easy to understand feedback to speed up the debugging of missing constraints.

The algorithm begins by using the time-unrolled counter-example and extracting all minimal correction sets with respect to the inputs of the design. This information is used to build a filtering function that encodes the incorrect input combinations that led to the failure in the counter-example. Next, a dictionary of fixed cycle properties is used to generate a list of candidate input constraints based on relevant signals from the counter-example. Each property on the list is then used in conjunction with the filtering function to generate a small SAT instance to determine if the property is a candidate for a missing constraint. The result is a set of input constraints that each can restrict the bad input behavior seen in the counter-example. Preliminary experimental 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.

The remaining sections of this paper proceed as follows. Section II and Section III present background material and the proposed approach, respectively. Section IV presents experimental results and Section V concludes this work.

II. Preliminaries

A. Minimal Correction Sets and Unsatisfiable Cores

Given an unsatisfiable (UNSAT) Boolean formula ϕ in conjunctive normal form (CNF), an *UNSAT core* is a subset of clauses 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 subset of clauses of ϕ such that removing the subset will result in ϕ being satisfiable. There exists a duality relationship between MUSs and MCSs as it is possible to compute the set of one from the other [12]. Using this relationship, one can calculate all MUSs from all MCSs.

Given an UNSAT CNF formula ϕ, MCSs can be computed by introducing a fresh variable to each clause called a *relaxation variable*. If the variable is active, then the clause is effectively removed from the problem. Using this idea, cardinality constraints [13] can be used to find all minimal sets of relaxation variables that make ϕ SAT. For each solution, the set of active relaxation variables correspond to an MCS. This idea has been used extensively in modern Max-SAT solvers [14], [15] to compute MCSs.

With respect to debugging, a MUS intuitively represents one way in which a counter-example can excite an error, traverse its effects through the design components and cause a failure at the observation points. In this view, clauses correspond to the counter-example, components of the design and target property. Alternatively, an MCS represents a minimal set of clauses related to components that are potentially erroneous. In other words, removing the components related to the MCS clauses is a potential way to "correct" the design. UNSAT cores and MCSs have been widely used in various debugging applications such as [16].

III. Debugging Missing Input Constraints

A. Extracting Failing Behaviors from a Counter-Example

In this subsection, 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 intuitively represents the disallowed input behaviors from the unrolled counter-example. More precisely, this function will represent 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 ϕ 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 ϕ models the counter-example of the unrolled circuit, it is guaranteed to be UNSAT.

Instead of computing all MUSs for ϕ 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, ϕ can be made SAT if at least one literal from each term in F is negated for the respective unit clauses in ϕ. 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} such 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 ϕ UNSAT (U^k), and the minimal subsets of inputs that need to be removed to make ϕ 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:

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

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 [14], [15]. 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. This allows the method to be more efficient as shown in the experimental results.

Example 1 *Consider the implementation of the simple state machine shown in Figure 1 that implements a modulo-2 counter that counts up when $a = 1$ and resets if $b = 1$. The property to be verified is:*

```
P: s == 2'b01 && a |=> s == 2'b10
```

Informally, if the counter is at `01` *and a is high, then in the next cycle it should be at* `10`*. If sent to a formal property checker, the property will fail because the property was written under the assumption that the reset signal b does not go high. A two cycle counter-example to this property is $X = < (a^0, \overline{b^0}), (a^1, b^1) >$, where the superscripts indicate the clock-cycle. Solving for all V^k, we find: $V^0 = \{a^0\}, V^1 = \{\overline{b^0}\}, V^2 = \{a^1\}, V^3 = \{b^1\}$. Which can directly be used to build $F = \overline{a^0} + b^0 + \overline{a^1} + \overline{b^1}$.*

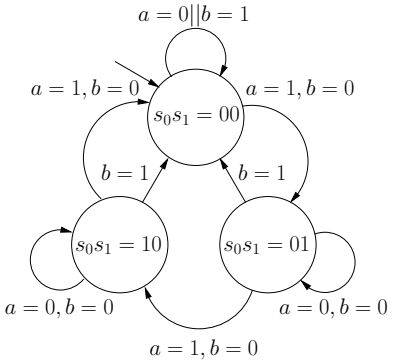

Fig. 1. Example 1: A Simple Modulo-2 Counter

B. Generating Fixed Cycle Properties

Missing constraints can be arbitrarily complex properties ranging from constant values to complex bus protocols that depend on the specifications. In general, there is no automated method to precisely generate these missing constraints that model the external environment. Even in cases where it may be possible, it is usually not practical. This is because algorithmically computed properties will likely be in some complex form that is unintelligible to user. This limits the benefit of any such technique to the user.

Instead, we take a different approach where simple fixed cycle properties are generated to give guidance to the user. In this way, the feedback can be used in conjunction with the user's knowledge to determine the missing input constraint, which frequently requires higher level design semantics. These properties may not be able to model all the complexities of the surrounding environment in all cases. However, the benefit of the proposed approach lies in the fact that it points the user to what types of constraints may be needed. Note that a more comprehensive set of properties can be used to expand upon the simple models presented in this preliminary study to gain greater benefit.

The process begins by selecting which input signals are involved in the counter-example failure. Any signal whose bit is used in F is considered to be a candidate for use in a generated property. Here, signals are categorized either as single-bit or multi-bit based upon the definition in the RTL.

For single bit signals and bits composing multi-bit signals, denoted by `a`, the following family of properties are generated:

- Stuck-at properties: `!a` and `a`.
- Hold: `$past(a) == a, $rose(a) |=> a, $rose(a) |=> !a, $fell(a) |=> a, $fell(a) |=> !a`

This family comprises of simple stuck-at properties and hold properties. These types of properties can be useful for detecting many different types of issues such as setting incorrect modes, or writing incorrect data.

Next, these multi-bit properties provide detection for common bus constraints such as one-hot, or incorrect addresses. `b1` and `b2` represent multi-bit signals, while `<val>` represents an assignment to the respective signal seen when simulating

the counter-example. The following is a family of multi-bit properties:

- One-hot properties: `$onehot(b)` and `$onehot0(b)`.
- Equality operators: `b1 <op> <val>` and `b1 <op> b2`. Where `<op>` is one of {`<`, `<=`, `==`, `>=`, `>`}, and where the size of `b1` and `b2` match.

These are slightly higher-level properties that may give intuition about certain missing constraints.

Once a list of properties are generated, each one can be efficiently filtered, as described in Section III-A, by creating a small SAT instance $F \cdot A$. Each instance is significantly smaller than the original unrolled circuit, allowing an efficient means of filtering these potential constraints without having to do an entire formal check.

Example 2 *Consider the filtering function F generated from Example 1 and the four stuck-at fault properties that would be generated: a, $\overline{a}$, b, and $\overline{b}$. Of these, only the first one would be filtered out since it would return SAT when run with F, while the others all return UNSAT. Of the remaining, it is easy to see how they translate to high-level behavior of the design: $\overline{a}$ prevents the counter from incrementing (a vacuous condition), b continually resets the machine (also vacuous), and $\overline{b}$ turns off reset (desired result).*

IV. Experimental Results

This section presents preliminary experimental results for the proposed approach. All experiments are run on a single core of a Intel Core i5 3.1 GHz quad-core workstation with 8 GB of RAM. Three designs are selected for our evaluation. The first two designs are from OpenCores [17] (`hpdmc`, `spi`), while the last one is a DDR2 controller from the OpenSparc project (`ddr`) [18]. For the OpenCores designs, SVA assertions are written based upon the accompanying design documentation. For the DDR2 controller, assertions from [19] are used which are based on the DDR2 specifications. These assertions are formally verified against the design using a commercial formal property checker [20], and any failures are considered instances of missing constraints. Each failing assertion is considered separately and is labeled by adding a number to the suffix of the circuit name.

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` [21] 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 I.

The first four columns of Table I show the instance name, number of gates, number of state elements, and counter-example length. The next three columns list the overall run-time in seconds of the proposed approach, which includes creating the filtering function as well as filtering, along with the original number of generated properties candidates from Section III-B, followed by 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 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.

This is not a big surprise considering the simplicity of the generated constraints. However, a main point of this work is to aid debugging of missing constraints, not necessarily generate the exact constraint for the user. The simplicity of the generated constraints in this case is beneficial since it gives a intuitive method for the user as to which constraint is potentially missing. To further illustrate this point, we describe in detail the results of several cases from Table I.

Consider the first failing property for `hpdmc1` that specifies that after a read, an acknowledge signal should be asserted several cycles later based on the `tim_cas` register.

```
P: $rose(read) |-> (!tim_cas ##5 $rose(ack))
                   or (tim_cas ##6 $rose(ack))
```

The proposed approach generates 29 constraints, which deal primarily with bus and address line input pins. In particular, these generated constraints seemed relevant:

```
A1: wbc_adr_i[3:2] == 2'b00
A2: !wbc_we_i
A3: wbc_dat_i[6]
A4: !wbc_dat_i[6]
```

The first two constraints force `tim_cas` not to be over-written during programming of the control registers, while the last two[†] ensure that regardless of what is programmed, `tim_cas` should be held stable. These constraints give intuition that the `tim_cas` register should be held constant when checking this property.

For `spi1`, the assertion is a simple property to detect that the internal FIFO raises the `empty` flag correctly:

```
P: (re && (rp+2'h1)==wp) |=> empty;
```

The proposed approach generated 10 constraints, where the following were of particular interest:

[†] The reason that both `wbc_dat_i[6]` and its complement are suggested is that it ensures that the signal is held constant throughout the trace so that it does not toggle.

TABLE I
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

```
A1: adr_i[1:0] != 2'b10
A2: !we_i
```

These constraints attempt to disable writing to the FIFO. In this case, the assertion was written under the assumption that the writes cannot happen if the current operation has not been acknowledged yet, as given by this property: `!ack_o |=> !we_i`. In this case, the missing constraint involves a more complex protocol that is dependent on an output pin, `ack_o`. Despite this, the returned constraints can remind the user that this protocol should be followed.

In the case of `ddr1`, the assertion involves a more complex setup described in [19] to reach the target property of: "No more than 4 *activate* commands may be issued to the DDR2 SDRAM within a window of t_FAW clock cycles." All the returned constraints deal with the `other_que_pos` signal, which controls the `we` signal, which in turn causes an *activate* command. The work in [19] suggests that the issue is either a design error, or a constraint is missing to model an adjacent block that enforces this behavior. In the latter case, `other_que_pos` constraints point to this conclusion.

V. CONCLUSION

In this work, an algorithm is proposed that automatically generates missing input constraints from a failing counter-example. It begins by building a filtering function that models the failing behaviors from the counter-example. Next, a list of fixed cycle properties are generated and filtered to return a set of constraints that restrict the failing behavior in the counter-example. Preliminary experimental results show that the constraints can be efficiently generated and they provide effective guidance to improve the formal verification flow.

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] 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

[3] H.-H. Yeh and C.-Y. R. Huang, "Automatic constraint generation for guided random simulation," in *ASP Design Automation Conf.*, 2010, pp. 613–618.

[4] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

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

[6] A. Gupta, K. L. Mcmillan, and Z. Fu, "Automated assumption generation for compositional verification," *Formal Methods in System Design: An International Journal*, vol. 32, no. 3, pp. 285–301, June 2008.

[7] Y.-F. Chen, E. M. Clarke, A. Farzan, M.-H. Tsai, Y.-K. Tsay, and B.-Y. Wang, "Automated assume-guarantee reasoning through implicit learning," in *Computer Aided Verification*, 2010, pp. 511–526.

[8] F. Ivancic, G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, H. Tokuoka, T. Imoto, and Y. Miyazaki, "DC2: A framework for scalable, scope-bounded software verification," in *Automated Software Engineering*, 2011, pp. 133 –142.

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

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

[11] K. Chatterjee, T. A. Henzinger, and B. Jobstmann, "Environment Assumptions for Synthesis," in *Int'l Conf. on Concurrency Theory*, 2008.

[12] 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.

[13] N. Eén and N. Sörensson, "Translating pseudo-boolean constraints into SAT," in *JSAT*, vol. 2, 2006, pp. 1–26.

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

[15] 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.

[16] S.Safarpour, M.Liffton, H.Mangassarian, A.Veneris, and K.A.Sakallah, "Improved Design Debugging Using Maximum Satisfiability," in *Formal Methods in CAD*, 2007.

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

[18] OpenSparc, 2012. [Online]. Available: http://www.opensparc.net

[19] A. Datta and V. Singhal, "Formal Verification of a Public-Domain DDR2 Controller Design," in *VLSI Design*, 2008, pp. 475–480.

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

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