Contradiction Analysis for Constraint-based Random Simulation

Daniel Große\textsuperscript{1} Robert Wille\textsuperscript{1} Robert Siegmund\textsuperscript{2} Rolf Drechsler\textsuperscript{1}

\textsuperscript{1}Institute of Computer Science, University of Bremen, 28359 Bremen, Germany \{grosse, wille, drechsle\}@informatik.uni-bremen.de

\textsuperscript{2}AMD Saxony LLC & Co. KG, Dresden Design Center, 01330 Dresden, Germany robert.siegmund@amd.com

Abstract

Constraint-based random simulation is state-of-the-art in verification of multi-million gate industrial designs. This method is based on stimulus generation by constraint solving. The resulting stimuli will particularly cover corner case test scenarios which are usually hard to identify manually by the verification engineer. Consequently, constraint-based random simulation will catch corner case bugs that would remain undetected otherwise. Therefore, the quality of design verification is increased significantly. However, in the process of constraint specification for a specific test scenario, the verification engineer is faced with the problem of over-constraining, i.e. the overall constraint specified for a test scenario has no solution. In this case the root cause of the contradiction has to be identified and resolved. Given the complexity of constraints used to describe test scenarios, this can be a very time-consuming process.

In this paper we propose a fully automated contradiction analysis method. Our method determines all “non relevant” constraints and computes all reasons that lead to the over-constraining. Thus, we pinpoint the verification engineer to exactly the sets of constraints that have to be considered to resolve the over-constraining. Experiments have been conducted in a real-life SystemC-based verification environment at AMD Dresden Design Center. They demonstrate a significant reduction of the constraint contradiction debug time.

1. Introduction

The continued advance of circuit fabrication technology that persisted over the last 30 years now allows the integration of more than 1 billion transistors in System-on-Chip (SoC) designs. The development of SoCs of such complexity leads to enormous challenges in Computer-Aided Design (CAD), especially in the area of design verification, which needs to ensure the functional correctness of a design. Because the capacity of formal verification is limited, simulation is still the most frequently used verification technique [22].

In directed simulation explicitly specified stimulus patterns (e.g. written by verification engineers) are applied to the design. Each of those patterns stimulates a very specific design functionality (called a verification scenario) and the response of the design is compared thereafter with the expected result. Due to project time constraints, it is inherent for directed simulation that only a limited number of such scenarios will be verified.

With random simulation these limitations are compensated. Random stimuli are generated as inputs for the design. For example, to verify the communication over a bus, random addresses, and random data are computed.

A substantial time reduction for the creation of simulation scenarios is achieved by constraint-based random simulation (see e.g. [2, 22]). Here, the stimuli are generated directly from specified constraints by means of a constraint solver, i.e. stimulus patterns are selected by the solver which satisfy the constraints. The resulting stimuli will also cover test scenarios for corner cases that may be difficult to generate manually. As a consequence, design bugs will be found that might otherwise remain undetected, and the quality of design verification increases substantially.

For constraint-based random simulation several approaches have been proposed (see e.g. [23, 4, 11, 21, 12]). However, a major problem that arises when stimuli are specified in form of constraints is over-constraining, i.e. the constraint solver is not able to find a valid solution for the given set of constraints. Whenever such a contradiction occurs in a constraint-based random simulation run, this run has to be terminated as no valid stimulus patterns can be applied. Note that over-constraining may not necessarily happen at the very beginning of the simulation run, as modern test-bench languages such as SystemVerilog [9] allow the addition of constraints dynamically during simulation. In any case of over-constraining the verification engineer has to identify the root cause of the constraint contradiction. As this is usually done manually by either code inspection or trial-and-error debug, it is a tedious and time-consuming process.

To the best of our knowledge in this work we propose the first non-trivial algorithm for contradiction analysis for constraint-based random simulation. In the area of constraint satisfaction problems methods for diagnosing over-constrained problems have been introduced (see e.g. [1, 16]). These methods aim to find a solution for the over-constrained problem by relaxing constraints according to a given weight for each constraint. In the considered problem no weights are available. Also, the approaches do not determine all minimal reasons that cause the overall contradiction. In contrast, Yuan et al. proposed an approach to locate the source of a conflict using a kind of exhaustive enumeration [22]. But since a very large runtime of this method is supposed – neither an implementation nor experiments are provided – they recommend to build an approximation. In the domain of Boolean Satisfiability (SAT) a somewhat similar problem can be found: computing an unsat core of an unsatisfiable formula, i.e. to identify an unsatisfiable sub-formula of the overall formula [5, 24]. However, to obtain a minimal reason the much more complex problem of a minimal unsat core has to be considered [15, 7, 14]. Furthermore, all minimal unsat cores are required to determine all contradictions. In general this is very time consuming (see e.g. [13]).

In this paper we propose a fully automatic technique for analyzing contradictions in constraint-based random simulation. The basic idea is as follows: The overall constraint is reformulated such that (contradicting) constraints can be disabled by introducing new free variables. Next, an ab-

\footnote{This research work was supported in part by the German Federal Ministry of Education and Research (BMBF) in the project URANOS under the contract number 01M3075.}
2. SystemC Verification Library

This section briefly reviews the SystemC Verification (SCV) library that is used for constraint-based random simulation in this work. The SCV library was introduced in 2002 as an open source C++ class library [20, 17, 10] on top of SystemC [19, 8]. In the following we focus only on the basic features of the SCV library for constraint-based random simulation.

Using the SCV library, constraints are modeled in terms of C++ classes. That way constraints can be hierarchically layered using C++ class inheritance. In detail a constraint is derived from the scv_constraint_base class. The data to be randomized is specified as scv_smart_ptr variables.

An example of an SCV constraint is shown in Figure 1. The name of the constraint is cstr. Here, the three 64 bit unsigned integer variables a, b, and addr are randomized. The conditions on the variables a, b, and addr are defined by expressions in the respective SCV_CONSTRAINT() macro.

Internally, a constraint in the SCV library is represented by the corresponding characteristic function, i.e., the function is true for all solutions of the constraint. This characteristic function of a constraint is represented as a Binary Decision Diagram (BDD), a canonical and compact data structure for Boolean functions [3]. For stimuli generation a weighting algorithm is applied for the constraint BDD to guarantee a uniform distribution of all constraint solutions and hence maximizing the chance for entering unexplored regions of the design state space. As BDD package CUDD [18] is used in the SCV library.

3. Contradiction Analysis

In this section first the considered problem, that is the contradiction of constraints, is formalized. Then, we present concepts for the contradiction analysis approach.

3.1. Problem Formulation

Before the problem is formulated we define the type of constraints that are considered in this paper.

**Definition 1.** A constraint is a Boolean function over variables from the set of variables V. For the specification of a constraint, the typical HDL operators such as e.g. logic AND, logic OR, arithmetic operators, and relational operators can be used.

Usually a constraint consists of a conjunction of other constraints. We formalize the resulting overall constraint in the following definition.

**Definition 2.** An overall constraint is defined as

\[ C = \bigwedge_{i=0}^{n-1} C_i \]

where \(C_i\) are constraints according to Definition 1.

In practice, the conjunction is built by the explicit use of several SCV_CONSTRAINT() macros or by applying inheritance, i.e., parts of the constraints are defined in a base class and inherited in the actual constraint. Note that this is not specific to constraint-based random simulation using the SCV library. In fact, the same principles are found, for example, in the random constraints of SystemVerilog [9].

During the specification of complex non-trivial constraints, the problem of over-constraining arises:

**Definition 3.** An overall constraint \(C\) is over-constrained or contradictory iff \(C\) is not satisfiable, i.e. \(C\) evaluates to 0 for all assignments to the constraint variables.

Typically, if \(C\) is over-constrained the verification engineer has to manually identify the reason for the over-constraining. This process can be very time-consuming because several cases are possible. For example, one of the constraints \(C_i\) may have no solution. Another reason for a contradiction may be that the conjunction of some of the constraints \(C_i\) leads to 0. In the following the term reason as used in the rest of this paper is defined.

**Definition 4.** A reason for a contradictory overall constraint \(C\) is the set \(R = \{C_1, C_2, \ldots, C_k\} \subseteq \{C_0, C_1, \ldots, C_{n-1}\}\) with the two properties:

1. The constraints in \(R\) form a contradiction, i.e. the conjunction always evaluates to 0.

The overall constraint \(C\) is contradictory.

2. Removing an arbitrary constraint from \(R\) resolves the contradiction, i.e. minimality of \(R\) is required.

Often the root of over-constraining results from more than one contradiction, i.e. there is more than one reason. If in this case only one reason is identified by the verification engineer, the constraint solver has to solve the fixed constraint again, but still there is no solution.

Based on these observations, the following problem is considered in this paper:

How can we efficiently compute all minimal reasons for an over-constraining and thereby support the verification engineer in constraint debugging?

Analyzing the contradictions in the overall constraint \(C\) and presenting all reasons is facilitated by our approach. In particular excluding all constraints which are not part of a contradiction reduces the debugging time significantly.

3.2. Concepts for Contradiction Analysis

The general idea of the contradiction analysis approach is as follows: The overall constraint \(C\) is reformulated such that the conflicting constraints can be disabled by the constraint solver and \(C\) becomes satisfiable. By analyzing the logical dependencies of the disabled constraints, we can identify all reasons for the over-constraining.
C_0 \iff b() < 3 \& \& \& b() == 7 \\
C_1 \iff a() + b() == c() \\
C_2 \iff a() < 6 \\
C_3 \iff a() == 5 \\
C_4 \iff a() == 10 \\
C_5 \iff d() == 8 \\
C_6 \iff d() > 10

Table 2. Contradictory constraint

Definition 5. Let \( C \) be over-constrained. Then the reformulated constraint \( C' \) is built by introducing a new free variable \( s_i \) for each constraint \( C_i \) and substituting each constraint \( C_i \) with an implication from \( s_i \) to \( C_i \). That is,

\[
C' = \bigwedge_{i=0}^{n-1} (s_i \rightarrow C_i).
\]

For the reformulated constraint \( C' \) the following holds:

1. If \( s_i \) is set to 1, then the constraint \( C_i \) is enabled.
2. If \( s_i \) is set to 0, then the constraint \( C_i \) is disabled because \( C_i \) can evaluate to 0 or 1.

Note that the usage of an implication is crucial. If an equivalence is instead used instead of an implication, \( s_i = 0 \) would imply the negation of \( C_i \).

Example 1. Figure 2(a) shows a constraint \( C \) which is over-constrained. Reformulating \( C \) to \( C' \) avoids the over-constraining because a constraint \( C_i \) may be disabled by assigning \( s_i \) to 0. The table in Figure 2(b) gives all assignments to \( s_i \) such that the reformulated overall constraint \( C' \) evaluates to 1.\(^1\) That is, the table shows which constraints have to be disabled to get a valid solution. For example, from the first row it can be seen that disabling \( C_0, C_2, C_3, \) and \( C_5 \) avoids the contradiction.

Based on the reformulated constraint the verification engineer is able to avoid the over-constraining. But to understand what causes the over-constraining, i.e., to identify the reason of each contradiction, a more detailed analysis is required. Here, two properties of the assignment table obtained from the reformulated overall constraint can be exploited.

Note that for simplicity we always refer to the assignment table in the presentation. As shown later in the implementation the assignment table needs not to be build explicitly.

Property 1. The value of variable \( s_i \) is 0 for all solutions (i.e., in each row of the table) iff the respective constraint \( C_i \) is self-contradictory (that is \( C_i \) has no solution).

Proof. \( \Rightarrow \): We show this by contraposition: If \( C_i \) has at least one solution, then there is a row where \( s_i \) is 1. Obviously this solution (row) can be constructed by assigning 1 to \( s_i \) and 0 to \( s_j \) for \( j \neq i \), because \((s_i \rightarrow C_i) = s_i \lor C_i = 0 \lor C_i = C_i = 1 \) and \((s_j \rightarrow C_j) = s_j \lor C_j = 1 \lor C_j = 1 \) for \( j \neq i \).
\n\( \Leftarrow \): To satisfy \( C' \) each element of the conjunction must evaluate to 1, so \((s_i \rightarrow C_i) = s_i \lor C_i \). Since \( C_i \) has no solution \( (C_i \) is always 0) \( s_i \) must be 0.

Thus, each constraint \( C_i \) whose \( s_i \) variable is always assigned to 0, is a reason for the contradictory overall constraint \( C \).

\(^1\)Here ‘\( \rightarrow \)’ denotes a don’t care, i.e., the value of \( s_i \) can be either 0 or 1. The table is derived from a symbolic BDD representation of all solutions for the \( s_i \) variables after abstraction of all other variables.

Property 2. The value of variable \( s_i \) is don’t care for all solutions (i.e., for all rows of the table) iff the constraint \( C_i \) is never part of a contradiction of \( C \).

Proof. \( \Rightarrow \): This property is shown by contradiction. Assume that \( s_i \) is don’t care for all solutions and \( C_i \) is part of a contradiction. Then, without loss of generality there has to be another satisfable constraint \( C_j \) such that \( C_i \land C_j = 0 \).\(^2\) If \( s_i \) is set to 1 and all other constraints \( C_k \) with \( k \neq j \) are disabled by \( s_k = 0 \), then \( C' \) is 1. However, switching \( s_i \) to 1 is not possible due to the conflict of \( C_i \) and \( C_j \). But this contradicts the assumption that the value of \( s_i \) is don’t care for all solutions.
\n\( \Leftarrow \): Because the constraint \( C_i \) is never part of a contradiction, \( C_i \) can be enabled or can be disabled. In other words, \( s_i \) can be set to 0 and also to 1 for each solution of the overall constraint, which is equivalent to \( s_i \) is don’t care.

Thus, each constraint \( C_i \) whose \( s_i \) variable is always don’t care, is not part of a reason for the contradictory overall constraint. Therefore these constraints are not presented to the verification engineer and can be left out in the next steps.

Example 2. Consider again Example 1. Because the value of \( s_0 \) is 0 for all solutions, \( C_0 \) is self-contradictory. Thus, \( R_0 = \{C_0\} \) is a reason for \( C \). Since the value of \( s_1 \) is always don’t care, \( C_1 \) is never part of a contradiction. As a result the first two constraints can be ignored in the further analysis.

Note that the overall constraint of the example in Figure 2(a) has been specified to demonstrate the two properties. In practice, the number of constraints that are never part of a contradiction is considerably larger. Thus, applying Property 2 reduces the debugging effort significantly because each “non relevant” constraint does not have to be considered anymore by the verification engineer.

In fact, all remaining constraints (if there are any) are part of at least one contradiction. Furthermore, since self-contradictory constraints have been filtered out by Property 1 only a conjunction of two or more constraints causes a contradiction. Now the question is, how can we identify the minimal contradicting conjunctions of the remaining constraints, i.e., the reasons?

Example 3. Again Example 1 is considered. The constraints \( C_0 \) and \( C_1 \) have been handled already according to Property 1 and Property 2. Now, the conjunction of two or more of the remaining constraints, \( C_2, C_3, C_4, C_5, \) and \( C_6 \), causes a contradiction. Only identifying the product of all these constraints certainly does not help to resolve the conflict easily. In contrast, the over-constraining can only be fixed if the different contradictions are understood. But this requires the computation of all minimal reasons according to Definition 4. In the example, three reasons can be found in total: \( R_1 = \{C_2, C_4\} \) and \( R_2 = \{C_3, C_4\} \) which overlap as well as \( R_3 = \{C_5, C_6\} \) which is independent of the two before.

To find the minimal reason for each contradiction, all constraint combinations are tested for a contradiction starting with the smallest conjunction. For each tested combination the respective \( s_i \) variables are set to 1. Thus, if the conjunction \( C_1 \land \ldots \land C_6 \) leads to a contradiction...
4. Implementation

As already mentioned earlier, the SCV library uses BDDs for the representation of constraints. More precisely the characteristic function of the overall constraint is represented as a BDD. This characteristic function is true for all solutions of the constraint, false otherwise. We implemented the contradiction analysis approach using the SCV library. Therefore our implementation is “BDD driven”.

The pseudo-code of the contradiction analysis approach is shown in Figure 3. As input the approach starts with the BDD representation of the reformulated constraint \( C' \) and the set of all constraint variables \( V \). At first, all constraint variables are existentially quantified from the reformulated constraint (line 3). Thus, the resulting function \( C'' \) only depends on the \( s_i \) variables. In other words, this function is the symbolic representation of the assignment table described in the previous section. In general the quantified BDD is much more compact than the BDD for the reformulated constraint. Thus, the following BDD operations can be executed very fast.

After quantification the two sets \( R \) and \( S \) are initialized to the empty set. \( R \) stores all reasons that are found. Note that for simplicity \( R \) contains the sets of the corresponding \( s_i \) variables of a reason, not the constraints itself. The set \( S \) is used to save all \( s_i \) variables that are passed to the detailed analysis later. So this set corresponds to the remaining constraints. Then, for each constraint \( C_i \), it is checked if \( C_i \) is either self-contradictory (line 9) or never part of a contradiction (line 12) according to Property 1 and Property 2. In the former case the respective \( s_i \) variable is added to the set of reasons \( R \) (line 11). Both checks are conducted on the quantified representation \( C'' \) of the reformulated constraint, that is:

- To check if \( s_i \) is 0 for all solutions (see Property 1) the conjunction \( C'' \land s_i = 1 \) is carried out. If the result is the constant zero-function, \( s_i \) is never 1 in any solution, i.e. \( s_i \) is always zero. Thus, \( C_i \) becomes a reason.
- The check if \( s_i \) is don’t care in all solutions (see Property 2) is carried out by \( (C''\land s_i = 0) \equiv (C''\land s_i = 1) \). If the respective BDDs are equal, it has been shown that \( s_i \) is don’t care, since regardless of the value of \( s_i \), the solutions are identical. Therefore, the constraint \( C_i \) is not relevant for a contradiction and thus neither added to the set \( R \) nor to the set \( S \).

If both properties cannot be applied (line 14), then the respective constraint \( C_i \) is part of a contradiction caused by the conjunction of \( C_i \) with one or more other constraints. Thus, \( C_i \) is passed to the detailed analysis by inserting the respective \( s_i \) into \( S \) (line 16).

Finally, the detailed analysis for all elements in \( S \) – the remaining constraints – is performed (line 18 to 25). First, the power set \( \mathcal{P}(S) \) of \( S \) is created resulting in all subsets (i.e. combinations) of constraints considered for detailed analysis. Note that we exclude the empty set as well as all sets which only contain one element (this is already covered by Property 1) from the power set. Furthermore, during the construction the elements of the power set are ordered according to their cardinality. Then, for each subset \( X \) (i.e. for each combination) the conjunction of the respective constraints is tested for a contradiction. Therefore, the conjunction of the current combination \( X' \) – represented as a cube of all variables \( s_i \in X' \) and \( C'' \) is created, i.e. all respective constraints \( C_i \) are enabled (line 23). If the conjunction leads to a contradiction, then \( X \) is a reason and thus, \( X \) is added to \( R \) (line 25). To ensure minimality each contradiction test of a subset \( X \) is only carried out if no reason \( X' \in \mathcal{P}(X) \) exists such that \( X' \subseteq X \) (line 20-22), i.e. no subset of \( X \) has already been identified as reason for a contradiction (see also Definition 4).

In summary, the presented contradiction analysis procedure computes all minimal reasons \( R \) of a contradictory overall constraint \( C \). First, the proposed reformulation of the overall constraint allows a representation where all contradictory constraints can be disabled. From this representation a much more compact one is computed by quantification. All following operations have to be carried out on this representation only. Then, the two properties are applied which significantly reduces the problem size since only \( 2^n - |Z| - |D_C| \) instead of all \( 2^n \) subsets have to be considered in the detailed analysis (\( Z \) denotes the set of self-contradictory constraints, and \( D_C \) denotes the set of constraints, which are not part of a contradiction). In practice, especially the number of “non relevant” constraints that belong to the set \( D_C \) is very large, so the input for the detailed analysis shrinks considerably.

5. Experimental Evaluation

This section provides experimental results for the contradiction analysis. We show the efficiency of our approach by
several testcases. Finally, the application of our approach in an industrial setting is presented.

In all examples the partitioning of the constraints is given according to the specification in the constraint classes, i.e., each $C_i$ in the following corresponds to a separate $SCV\_CONSTRAINT(i)$ macro (see also Section 3.1). The contradiction analysis is started by an additional command-line switch and runs fully automatic in the SCV library environment.

5.1. Effect of Property 1 and Property 2

Applying the two properties introduced in Section 3.2 significantly reduces the complexity of the contradiction analysis since each matched constraint can be excluded from further considerations. To show the increasing efficiency we tested our approach for several examples which contain some typical overconstraining errors (e.g. typos, contradicting implications, hierarchical contradictions, etc.).

For the considered constraints we give some statistics in Table 1. In the first column a number to identify the testcase is given. Then, in the next columns information on the constraint variables and their respective sizes are provided. Finally, the total number of constraints is given. The results after application of our contradiction analysis are shown in Table 2. The first four columns give some information about the testcase, i.e. the number of constraints in total ($n$), the number of contradictions/reasons ($|R|$), and the runtime in CPU seconds needed to construct the BDD in the SCV library (BDD TIME). The next columns provide the results for the trivial analysis approach without (W/O PROPERTIES) and with the application of the properties (WITH PROPERTIES), respectively. Here the number of checks in the worst case ($2^n$ or $2\cdot n$ respectively), the number of checks actually executed by the approach ($#_v$), and the runtime for the detailed analysis (TIME) are given. Additionally the number of “non relevant” constraints ($|DC|$) and self-contradictory constraints ($|Z|$) obtained by the two properties are provided.

The results clearly show, that identifying all reasons without applying the properties leads to a large number of checks in the worst case (e.g. $2^{40} > 9.0 \times 10^{15}$ in example #5). In contrast, when the properties are applied most of the constraints can be excluded for the analysis since they are “non relevant”. This significantly reduces the number of checks to be performed at detailed analysis. Instead of all $2^n$ only $2^n - |Z| - |DC|$ checks are needed in the worst case (only 64 in example #5). As a result the runtime of the detailed analysis is magnitudes faster when the properties are applied. Moreover, for the last three testcases the reasons can be determined within the timeout of 7200 CPU seconds only when the properties are applied.

### Table 1. Constraint characteristics

| $n$ | $|R|$ | $2^n$ | $#_v$ | TIME |
|-----|------|-------|-------|-------|
| 1   | 1    | 2     | 2     | 3.38  |
| 2   | 3    | 6     | 6     | 22.30 |
| 3   | 10   | 15    | 15    | 64.88 |
| 4   | 30   | 45    | 45    | 2.38  |
| 5   | 50   | 51    | 51    | 1.925 |

5.2. Real-life Example

The constraint contradiction analysis algorithm has been evaluated using a real-life design example. The Design Under Verification (DUV) is a PCIe root complex design with an AMD-proprietary host bus interface which is employed in a SoC recently developed by AMD. The root complex supports a number of PCIe links. The verification tasks are to show (1) that transactions are routed correctly from the host bus to one of the PCIe links and vice versa, (2) that the PCIe protocol is not violated and (3) that no deadlocks occur when multiple PCIe links communicate to the host bus at the same time.

Host bus and PCIe links are driven by Bus Functional Models (BFMs) which convert abstract bus transactions into the detailed signal wigglings on those buses. The abstract bus transactions are generated by means of random generators which are in turn controlled by constraints. Bus monitors observe the transactions sent into or from either interface and send them to checkers which perform the end-to-end transaction checking of the DUV. The verification environment is implemented in SystemC 2.1, the SCV library, and SystemVerilog, with a special co-simulation interface synchronizing the SystemVerilog and SystemC simulation kernels. The constraint-random verification methodology was chosen in order to both reduce effort in stimulus pattern development and to get high coverage of stimulation corner cases. The PCIe and host bus protocol rules were captured in SCV constraint descriptions and are used to generate the contents of the abstract bus transactions driving the BFMs.

The PCIe constraints used to control stimulus generation within the PCIe transaction generator is a layered constraint. The lower level layer describes generic PCIe protocol rules and is comprised of a number of 16 constraint terms. They are shown in Figure 4(a) (denoted from $C_0$ to $C_{15}$). The meaning of the constraint variables is given in the table (Figure 4(b)). The upper level layer imposes user-specific constraints on the generic PCIe constraints (denoted by $C_U$) in order to generate specific stimulus scenarios. Generic PCIe constraints and user-defined constraints are usually developed by different verification engineers; the former by the designer of the test environment and the latter by the engineer who implements and runs the tests. The engineer who writes the tests and hence the user-specific constraints which are layered on top of the generic PCIe constraints is faced with the problem to resolve contradictions which are generated by imposing the user-defined constraints on the PCIe generic constraints. Given the complexity of the constraints, this is usually a non-trivial task. Two real-life examples of contradictions that are not easy to resolve by manual constraint inspection are depicted in Figure 4(c).

In the first example the user sets the maximum transaction length to a value greater than 128 bytes ($C_{U_1}$), thereby causing a contradiction to constraint $C_{13}$, which states that the total transaction length must not exceed 128 bytes. In the second example, the user independently constrains the address space to address space 0000 (in $C_{U_2}$) and the transaction length to 100 bytes (in $C_{U_3}$). While both values, viewed independently, are each perfectly legal (the address space should be in 32 bit range and the transaction length is less than 128), an over-constraining occurs. The reason identified by our approach is $R_1 = \{C_{12}, C_{U_2}, C_{U_3}\}$. By manual constraint inspection it is not immediately obvious that a PCIe protocol rule is violated when combining constraints $C_{U_2}$ and $C_{U_3}$. However, reason $R_1$ found for the contra-

### Table 2. Effect of using properties

| $n$ | $|R|$ | $2^n$ | $#_v$ | TIME |
|-----|------|-------|-------|-------|
| 2   | 1    | 2     | 2     | 3.38  |
| 3   | 3    | 6     | 6     | 22.30 |
| 4   | 10   | 15    | 15    | 64.88 |
| 5   | 30   | 45    | 45    | 2.38  |

### Footnote

1 Bit operators are used as introduced in [6].
C_0 ⇔ (addr,space := memory || ((mem_addr_base0 <= addr) && ((addr+length) < mem_addr_base0 + mem_size)))

|| ((mem_addr_base1 <= addr) && ((addr+length) < mem_addr_base1 + mem_size))) // address boundaries for memory
C_1 ⇔ (addr,space := io || ((iio_addr_base <= addr)&&(addr + length) <= io_addr_base + io_size)) // address boundaries for io
C_2 ⇔ (addr,space := config || ((cfg_base_addr <= addr)&&(addr+length) <= config_base_addr + config_size)) // address boundaries for config
C_3 ⇔ (be[] <= 0xf // valid byte enables are in 0x0-0f
C_4 ⇔ (be[len] == length || generate as many byte enables as we have dword data
C_5 ⇔ (data[len] == length || set data length
C_6 ⇔ cmd != read || posted == false // read transactions are always non-posted
C_7 ⇔ gen_host_trans.addr == memory || ((addr+length) <= 4 || only generate transactions in memory/IO/config space
C_8 ⇔ length > 0 // requests must have length > 0
C_9 ⇔ addr == sr::mem // addr <= 0xFFFFFFFF
// IO and config space are restricted to 32 bits
C_10 ⇔ (addr & 4095) + length <= 4096 // transactions must not cross 4k page boundary
C_11 ⇔ ((mem_addr_base0 <= addr) && (addr+length) <= 4000) // keep transaction length to max. 128 bytes
C_12 ⇔ (length == 100 || generate requests only (not responses)
C_13 ⇔ (mem_addr_base0 || length <= addr) && ((addr+length) <= mem_addr_base0 + mem_size)) // keep transaction to max. 128 bytes

(b)

Example 1: C_{u1} ⇔ length > 128
Example 2: C_{u2} ⇔ addr == 4000 / C_{u3} ⇔ length == 100

Figure 4. PCIe transaction generator constraint with examples

diction by our algorithm shows that when combining constraints C_{u2} and C_{u3}, then PCIe protocol rule C_{12} is violated: “A transaction must not cross a 4k page boundary”. Our user constraints of transaction start address set to 4000 and transaction length of 100 bytes would result in addresses that cross a 4k page and therefore violate this constraint.

The algorithm described in this paper is able to identify exactly the violating constraint expressions for both examples in about 30 seconds. The PCIe constraint to be analyzed contained a total of 21 random variables to be solved which are constrained by 17 and 18 constraint expressions for the respective examples. The total bit count for the random variables amounted to 781 bits. Without such an analysis capability, we would have had to spend several hours on manual constraint inspection in order to identify the root cause for the constraint contradiction. Thus, a significant speed up of the contradiction debug cycle was achieved.

6. Conclusions

In this paper we have presented a fully automatic approach to analyze contradictory constraints that occur in constraint-based random simulation. After reformulating the overall constraint and building an abstraction, the self-contradictory constraints and all “non relevant” constraints are determined in an initial step. Then for the small set of remaining constraints, all minimal reasons for a contradiction are computed efficiently and presented to the verification engineer. The minimality and completeness of the reasons allows to fully understand the over-contraining. Thus, the verification engineer is able to resolve the conflict in one single step. In total, as shown by industrial experiments, the debugging time is reduced significantly.

References