Determining Cases of Scenarios to Improve Coverage in Simulation-based Verification

Shuo Yang[1] Robert Wille[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{shuo,rwille,drechsler}@uni-bremen.de

ABSTRACT

Functional verification of complex designs is still dominated by simulation-based approaches. In particular, *Coverage-driven Verification* (CDV) is well acknowledged and applied in industry. Here, verification gaps in terms of inadequately checked scenarios are addressed and closed by generating and applying dedicated stimuli. In order to ensure a good coverage and, by this, a high verification quality, each scenario is supposed to become *sufficiently* triggered. However, the considered scenario may be triggered in several fashions and information about that is hardly available in the existing CDV approaches. In this work, we propose an approach which automatically derives this information. Examples and experimental evaluations illustrate how this improves coverage in simulation-based verification.

1. INTRODUCTION

How to efficiently verify and validate the functional correctness of a *Design Under Verification* (DUV) remains an important research area in the design of highly integrated *Systems on Chips* (SoCs) or *Networks on Chips* (NoCs). Due to the tremendous computational cost of *formal verification*, e.g. property checking and completeness checking (see e.g. [1, 2]), *simulation-based approaches* (see e.g. [3]) still dominate functional verification. Dedicated stimuli are thereby generated and applied to the DUV. The responses of the design are then compared to the expected results.

However due to the exponential amount of possible stimuli to be generated and applied, it is infeasible to exhaustively cover the functionality of a DUV by this method. As a consequence, simulation-based approaches aim for explicitly focusing on certain (hard-to-reach) behaviors. For this purpose, a set of scenarios is defined which abstracts the hard-to-reach behavior and describes the respective functionality to be checked (see e.g. [4]). Afterwards, an adequate number of directed stimuli is generated which is supposed to sufficiently trigger the scenarios. By this, the hard-to-reach behaviors are assumed to properly be verified.

To ensure that the generated set of stimuli indeed sufficiently triggers the considered scenario, a noticeable number of advanced technologies has been presented in the past (see e.g. [5, 6, 7, 8, 9, 10, 11, 12, 13]). This led to *Coverage-driven Verification* (CDV) which is well acknowledged and applied in industry. Here, advanced coverage analysis methods (see e.g. [7, 8]) are applied. They analyse the coverage of the already applied stimuli and derive further information on which scenarios have not sufficiently been triggered yet. Based on these information, constraint-based random stimuli generation (see e.g. [9, 10, 11]) or *Coverage-driven Stimuli Generation* (CDG, see e.g. [5, 6]) is utilized to generate dedicated stimuli ensuring a sufficient coverage.

However, guaranteeing a sufficient coverage obviously relies on an appropriate definition of what is meant by *sufficiency.* In general, a scenario may be triggered in several fashions. Generating a set of stimuli, which triggers a considered scenario several times but always in the same fashion, does not significantly improve the coverage. Instead, all scenarios should be triggered in various fashions. Nevertheless thus far, information on how a scenario can be triggered is hardly available in the existing CDV approaches. This is discussed in more detail later in Section 2.

In this work, we aim for improving this. An approach is presented, which aids coverage analysis of simulation-based verification by providing explicit information about the possible fashions in which a scenario can be triggered. The proposed approach exploits thereby the implicative power of solvers for *Boolean satisfiability* (SAT). For a given DUV and a scenario to be considered, an encoding is introduced representing the question "Is there an assignment of c primary input and/or flip flop signals, which triggers the considered scenario?". By determining all such assignments, all cases triggering the scenario are derived. Since this may lead to a significant number of cases, a dedicated blocking scheme is introduced afterwards guiding the SAT solver so that not all but representative cases are derived.

Experimental evaluations confirm the efficiency and usefulness of the proposed approach. For different DUVs and the respectively given scenarios, representative cases to be considered by coverage analysis are determined in negligible run-time. This provides crucial information which can be utilized to improve coverage in simulation-based verification.

The remainder of the paper is structured as follows: The addressed problem is motivated and defined in the next section. Afterwards, the general ideas of the proposed solution are sketched in Section 3. Section 4 describes the precise implementation, which is followed by Section 5 on possible blocking schemes guiding the solver in order to determine representative cases only. Finally, a summary of the conducted experimental evaluations is provided in Section 6 and conclusions are given in Section 7.

2. BACKGROUND

In this work, we are aiming for improving coverage in simulation-based verification by proposing a method which determines cases of a scenario to be considered. This section motivates this goal and discusses the related work. Afterwards, a precise problem formulation is provided.

2.1 Motivation

Given a DUV, a set of scenarios is usually provided in simulation-based verification. Each of them specifies certain behaviour, in particular a hard-to-reach behavior, to be verified. In the following, the term *scenario* is formally defined as:

DEFINITION 1. *A* scenario S_i *$(0 \leq i < n)$ is a Boolean function over variables from the set of DUV signals*[1]*. For the specification of a scenario, a constraint is formulated by using the typical HDL operators such as logic AND, logic OR, arithmetic operators, or relational operators. In the following, scenarios and constraints are used interchangeably. The set of scenarios is denoted by $S = \{S_0, \ldots, S_{n-1}\}$.*

Afterwards, scenarios are expected to be triggered by subsequently applying stimuli to the DUV. The responses of these stimuli are compared to the expected results. By this, the correctness of the DUV is validated. Each scenario is thereby expected to become *sufficiently* triggered in order to adequately verify the underlying behavior. The goal of CDV is to ensure that such sufficiency can be achieved as soon as possible, i.e. with as few stimuli as possible.

However, this goal obviously relies on an appropriate definition of what is meant by *sufficiency*. Thus far, scenarios are considered sufficiently covered, e.g. when they have been triggered a certain amount of times (see also the discussion of the related work in the next subsection). This is a weak criterion as scenarios may be triggered in various fashions. Generating a set of stimuli, which triggers a considered scenario several times but always in the same fashion, does not significantly improve the coverage. Hence, in this work we are aiming for determining all possible fashions in which a scenario can be triggered. For this purpose, we introduce the term *case* of a scenario.

DEFINITION 2. *A case $c_l^{S_i}$ $(0 \leq l < m)$ of the scenario S_i is a Boolean function over a (minimal) set of primary inputs and flip flops including their assignments which propagate through the DUV and trigger S_i. In the following, the set of cases of a scenario S_i is denoted by $C^{S_i} = \{c_0^{S_i}, \ldots, c_{m-1}^{S_i}\}$.*

The idea of cases is illustrated by the following example.

EXAMPLE 1. *Consider a simplified* Memory Management Unit *(MMU) with the primary inputs re_req (read request) and mem_ack (memory acknowledge) as well as the flip flop state. A scenario $S_i = re\_issue$ is formulated in order to monitor the issue of a memory read operation. According to the specification, a read operation is issued if*

- *the MMU is in state "idle" and a read request is pending,*
- *the MMU is in state "read" and a read request as well as a memory acknowledge are pending, or*
- *the MMU is in state "write" and a read request as well as a memory acknowledge are pending.*

Hence, there exist three cases of S_i:

1. $c_0^{S_i}$:$(state = idle) \wedge (re\_req = 1)$
2. $c_1^{S_i}$:$(state = read) \wedge (re\_req = 1) \wedge (mem\_ack = 1)$
3. $c_2^{S_i}$:$(state = write) \wedge (re\_req = 1) \wedge (mem\_ack = 1)$

When proving the scenarios, engineers or the applied CDV-environment are often not aware of all cases of a scenario. Instead, they only focus on the targeted behavior (the issue of a memory read operation in this case). Nevertheless in order to sufficiently cover this scenario, all cases should be considered.

However as discussed next, determining all cases of a scenario is not obvious and has hardly been considered in coverage analysis of simulation-based verification.

2.2 Related Work

The state-of-the-art CDV makes use of both, coverage analysis and stimuli generation, in order to sufficiently verify a DUV. Coverage analysis allows engineers to focus on a subset of scenarios with common properties (see e.g. [7]) and to discover shorter and more meaningful hints for the generation of better stimuli (see e.g. [8]). By this, the tedious analysis of a large set of scenarios is avoided and dedicated stimuli are derived earlier. However since these methods do not explicitly consider all cases of a scenario, the resulting hints neither pin-point to the missed cases nor provide information e.g. on how to trigger them in the subsequent simulations.

In constraint-based random stimuli generation (see e.g. [9, 10, 11]), constraints are formulated *manually* or *randomly*, which are expected to address the cases of a scenario. But since these cases are often not explicitly observable, the formulation can be tedious and the quality of the constraints strongly depends on the expertise of verification engineers.

In model-driven CDG (see e.g. [5]), methods are applied which aim for fairly exploiting the external model. By this, the cases of a scenario may be addressed by the resulting stimuli. However, the computational cost is a major problem in this field. Hence, researchers still focus on improving the efficiency of the respective approaches (see e.g. [14, 15]) rather than explicitly considering the cases of a scenario.

Finally, data-driven CDG (see e.g. [6]) generates stimuli by traversing the transition relationship. To date, such relationship is constructed (or trained) using machine learning (see e.g. [12]). The effectiveness of them relies on the quality of the training samples. Since such samples are made of data from the previous simulations, the quality of the relationship is hard to improve. Moreover, since the relationship is gradually derived during CDV, it can not be guaranteed that indeed all cases of a scenario are triggered.

[1]For sequential behavior, state signals may additionally be considered which are processed by un-rolling the DUV the respectively required number of times.

2.3 Problem Formulation

The discussions from above motivate the explicit consideration on cases of scenarios in order to improve coverage in simulation-based verification. Overall, this

- enables coverage analysis tools to conduct a more precise evaluation on the sufficiency,
- facilitates stimuli generators to determine more dedicated stimuli which aim for triggering a scenario through specific, e.g. not covered, cases, and
- provides means to evaluate stimuli generation methods as well as evidences to convince engineers on a simulation result.

Nevertheless, how to determine all cases of a given scenario remains an unsolved problem. This problem is addressed in this work. In the next section, the general ideas of the proposed solution are presented.

3. GENERAL IDEAS

Cases represent possible fashions to trigger a scenario. They are usually implicitly described in abstractions of the DUV provided e.g. for *Transaction Level Modeling* (TLM) or at the *Register Transfer Level* (RTL). In order to determine these cases, formal methods particularly suited for these abstractions can be utilized, e.g. *Binary Decision Diagrams* (BDDs) or solvers for *Boolean satisfiability* (SAT solvers). In this work we consider the DUV being represented at the RTL and exploit the implicative power of SAT solvers. Nevertheless, the proposed approach can also be adapted to the other abstraction levels and/or solving schemes.

Given the RTL description of a DUV and a scenario S_i, the general idea of our approach is to determine all cases of S_i by solving a sequence of decision problems. Each decision problem asks whether there is an assignment of c signals (composed of primary inputs and/or flip flops) which triggers S_i. If such an assignment can be determined, it represents the first case $c_0^{S_i}$ according to Definition 2. In order to determine the remaining ones, $c_0^{S_i}$ is blocked from further consideration. If it can be proven that no (further) assignment over c signals exists, the search continues with a c increased by 1. By this, all cases of S_i are determined. The respective decision problems are solved using Boolean satisfiability, i.e. the decision problem is encoded as an propositional formula, which afterwards is passed to a SAT solver.

In order to explicitly consider only c primary inputs/flip flops, a *three-valued* encoding with values from $\{0, 1, X\}$ is applied, i.e. not only the Boolean values 0 and 1, but also X which specifies an *observability don't care* are allowed. Applying an X-value to a primary input/flip flop possibly causes succeeding signals to assume the value X, too. As obviously X-values cannot trigger the scenario, this can be used in order to check whether a number of c primary inputs/flip flops with Boolean values already trigger the scenario. Hence, an instance is created in which c primary inputs/flip flops are allowed to assume Boolean values, while all remaining ones have to be set to X.

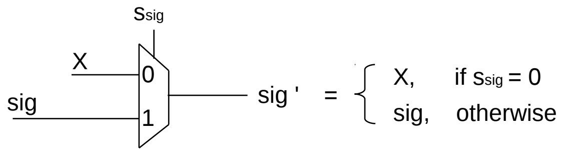

Figure 1: Applied multiplexers

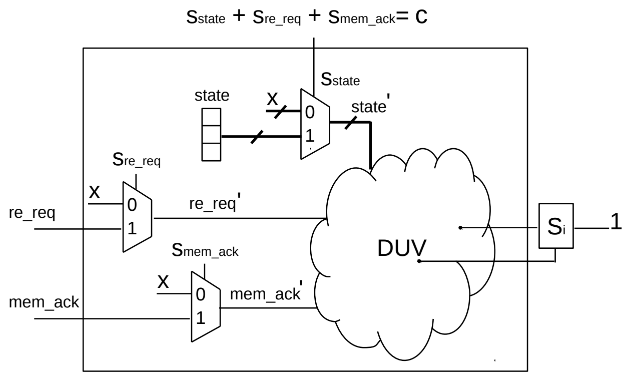

Figure 2: Problem Structure of the MMU

Which primary inputs/flip flops are to assign a Boolean value and which ones are to assign a X-value is to be determined by SAT solvers. In order to encode this, a multiplexer structure as shown in Fig. 1 is applied for each primary input/flip flop[2]. More precisely, each primary input/flip flop (here, denoted by *sig*) is used as an input of the added multiplexer. The other multiplexer-input is set to the fixed value X. Then, the value of the select-signal s_{sig} decides whether an arbitrary (Boolean) value or the observability don't care is assumed for this primary input/flip flop. By additionally limiting the number of select-signals with value 1 to c, only assignments with exactly c Boolean primary inputs/flip flops are considered.

4. IMPLEMENTATION

Taking the general ideas sketched above, an approach based on Boolean satisfiability results, which determines all cases of a given scenario. In this section, the resulting SAT encoding is described in detail. Afterwards, the overall scheme of the proposed approach is presented.

4.1 Proposed SAT Encoding

The proposed approach relies on the encoding of a SAT instance Φ representing the question "*Is there an assignment of c primary input and/or flip flop signals which triggers the considered scenario S_i?*". For this purpose, the respective "ingredients" discussed above need to be encoded. Fig. 2 exemplarily illustrates them for the DUV from Example 1 realizing the MMU.

First, the DUV itself as well as the considered scenario S_i needs to be encoded. It is well known that, this can be done in time and space linear in the size of the DUV [18]. This leads to the sub-instances Φ_{DUV} and Φ_{S_i}, whereby Φ_{DUV}

[2]This structure is inspired from works such as [16, 17], where a similar multiplexer-encoding is applied for the purpose of debugging.

represents the DUV and Φ_{S_i} enforces that only assignments are obtained which indeed trigger S_i.

Then, the multiplexers as shown in Fig. 1 are added for each primary input/flip flop. In the considered example, this concerns *re_req* and *mem_ack* as well as *state*. Corresponding encodings for the multiplexers are also well known [18]. This leads to the sub-instance Φ_{MUX}.

Finally, the restriction on the number c of the considered primary inputs/flip flops has to be incorporated. This depends on the values of the respective select-signals s_{sig} of all the multiplexers. In fact, we need to enforce that exactly c of those select-signals are set to 1 (ensuring a primary input/flip flop set to a Boolean value), while all remaining ones are set to 0 (leading to a primary input/flip flop with an observability don't care). That is, for all k primary input/flip flop signals $sig_0, sig_1, \dots, sig_k$, the constraint $s_{sig_0} + s_{sig_1} + \dots + s_{sig_k} = c$ is added and has to be satisfied. This is done using existing encodings for *cardinality constraints* [19].

The conjunction of all constraints eventually results in the desired SAT instance, i.e.

$$\Phi = \Phi_{DUV} \wedge \Phi_{S_i} \wedge \Phi_{MUX} \wedge (\sum_{j=0}^{k} s_{sig_j} = c).$$

EXAMPLE 2. *Consider again the MMU with its scenario S_i from Example 1 and the respective illustration in Fig. 2. Initially, the value of c is set to* 0*, i.e.* all *primary inputs/flip flops are enforced to assume an observability don't care. Obviously, this does not trigger S_i and, hence, each SAT solver proves this instance to be unsatisfiable.*

The same happens when c is set to 1*, i.e. even if one primary input/flip flop is allowed to contain a Boolean value. This concurs with the disucssion from Example 1: The scenario S_i is only triggered if the state* and *at least the re_req signals have a certain (specified) value.*

In fact, the first assignment satisfying all constraints for this DUV results for $c = 2$. This leads to state = idle and re_req = 1 which is the first case $c_0^{S_i}$ triggering S_i.

The resulting case is stored in a set C^{S_i}, i.e. after this first iteration $C^{S_i} = \{c_0^{S_i}\}$. However as we are interested in *all* cases of the given scenario, the SAT solver is asked for further assignments. For this purpose, we have to ensure that the already obtained case is not determined again. This is accomplished by excluding $c_0^{S_i}$ through a *blocking constraint*, i.e. the SAT instance Φ is extended to

$$\Phi = \Phi_{DUV} \wedge \Phi_{S_i} \wedge \Phi_{MUX} \wedge (\sum_{j=0}^{k} s_{sig_j} = c) \wedge \overline{c}_0^{S_i}.$$

The extended SAT instance is then again passed to a SAT solver.

EXAMPLE 3. *Consider again the MMU-example from above. In order to determine further assignments and, by this, further cases, the blocking constraint $\overline{c}_0^{S_i} = \overline{state = idle \wedge re\_req = 1}$ is added to the SAT instance. Solving this new instance yields no further satisfying assignments for $c = 2$. Hence, the value of c is increased to* 3*. Here, two more satisfying assignments can be obtained from which the cases $c_1^{S_i}$ and $c_2^{S_i}$ are derived. Both are subsubsequently added to C^{S_i}. As no further assignments are available and the total number of primary inputs/flip flops has been reached by c, the whole process terminates. Eventually, all cases of the scenario S_i have been obtained.*

Algorithm 1: Scenario Case Generation

```
Input: DUV, $S_i$
Output: $C^{S_i}$
1  $c = 0$ ;
2  $C^{S_i} = \emptyset$ ;
3  $\Phi = \Phi_{DUV} \wedge \Phi_{S_i} \wedge \Phi_{MUX}$
       $\wedge (\sum_{j=0}^{k} s_{sig_j} = c) \bigwedge_{c^{S_i} \in C^{S_i}} \overline{c}^{S_i}$ ;
4  if solve ($\Phi$) then
5      $c^{S_i}$ = derive_from_sat_assmnt($\Phi$) ;
6      $C^{S_i} = C^{S_i} \cup \{c^{S_i}\}$ ;
7      go to Line 3 ;
8  else
9      if $c < max$ then
10         $c$++ ;
11         go to Line 3 ;
12     else
13         return $C^{S_i}$ ;
```

Taking all issues described above into account, the complete SAT encoding for each iteration is composed of the encoding of the DUV, the encoding of the scenario, the encoding of the multiplexers, the cardinality constraint restricting the number c of the considered primary inputs/flip flops, and the blocking constraints, i.e.

$$\Phi = \Phi_{DUV} \wedge \Phi_{S_i} \wedge \Phi_{MUX} \wedge (\sum_{j=0}^{k} s_{sig_j} = c) \wedge \bigwedge_{c^{S_i} \in C^{S_i}} \overline{c}^{S_i}.$$

4.2 Overall Flow

The proposed SAT encoding is iteratively applied in order to derive all cases of a given scenario. The precise flow is thereby provided in Algorithm 1. Given the DUV and the scenario S_i to be considered, first the variable c and the set C^{S_i} are initialized with 0 and $\emptyset$, respectively (Lines 1-2). Afterwards, the SAT instance Φ is created and solved for the first time (Lines 3-4). If the instance is satisfiable, a case c^{S_i} is derived from the satisfying assignment and added to the set C^{S_i} (Lines 4-6). Afterwards, the process continues at Line 4, i.e. a further satisfying assignment is sought. This process continues until no further assignments and, by this, cases are determined, i.e. until Φ becomes unsatisfiable (Line 8). Then, it is checked whether c can further be increased, i.e. if $c < max$ (Line 9). The value of max is thereby either the total number of primary inputs/flip flops or a threshold provided by the designer[3]. If this is the case, the algorithm continues with a c increased by one (Lines 10-11). Otherwise, the algorithm terminates and returns the obtained set C^{S_i} of cases of S_i (Line 13).

[3]This is discussed in more detail in the next section.

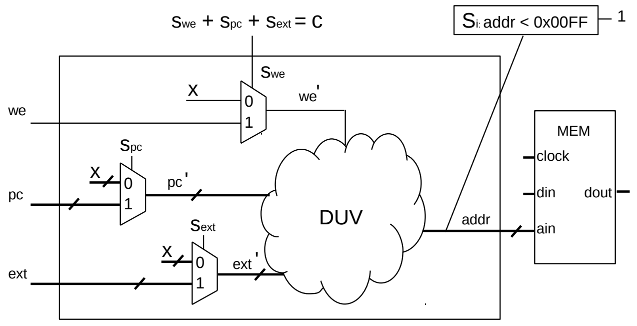

Figure 3: Problem Structure of the ROM

5. REPRESENTATIVE CASES

The approach presented in the previous section leads to the determination of *all* cases of a given scenario. However, as illustrated by the next example often not all, but just a set of *representative* cases are desired in practice.

Example 4. *Consider a simple program memory unit (i.e. a ROM) with the primary inputs we, pc, and ext. Typically, certain regions of a ROM are reserved for special operations like interrupts. Hence, these regions are expected to be checked by simulation-based verification which is why a scenario* $S_i = addr < 0x00FF$ *is considered. An illustration of the corresponding SAT encoding is provided in Fig. 3.*

Let's assume that the region specified by the scenario is accessed if

- A ROM-read operation is conducted (i.e. $we = 0$*)* *Then, addr is assigned with the address of the pc, i.e.* $addr = pc$. *This leads to numerous cases of the form* $(we = 0) \wedge (pc = 0x0000)$, *…*, $(we = 0) \wedge (pc = 0x00FE)$.
- A ROM-write operation is conducted (i.e. $we = 1$*)* *Then, addr is assigned by the external address ext. This leads to numerous cases of the form* $(we = 1) \wedge (ext = 0x0000)$, *…*, $(we = 1) \wedge (ext = 0x00FE)$.

Either way, this leads to a significant amount of cases – way to much for an efficient coverage analysis. Instead, one is interested in representative *cases only, e.g. cases which cover the fact that this scenario can be triggered either by setting* $we = 0$ *and* $pc < 0x00FF$ *or* $we = 1$ *and* $ext < 0x00FF$.

A possible scheme to restrict the number of determined cases is to simply lower the threshold *max* in Algorithm 1. Then, depending on the heuristics of the SAT solver a fair distribution of cases is determined or not.

Example 5. *Consider again the DUV from Example 4. Following Algorithm 1, the first case* $c_0^{S_i} = (we = 0) \wedge (pc = 0x0000)$ *might result. In the next iteration, this case is blocked by adding the constraint* $\overline{c}_0^{S_i}$. *Then, depending on the SAT solvers' heuristics, e.g. either* $(we = 0) \wedge (pc = 0x0001)$ *or* $(we = 1) \wedge (ext = 0x0000)$ *might result as the second case* $c_1^{S_i}$. *While the former one is just a slightly changed variant of* $c_0^{S_i}$, *the latter one shows an entirely new fashion to trigger* S_i. *By setting the value of max, the designer can define the chances of determining representative cases. If max is set to the total maximum, all cases and, by this, the best possible distribution is obtained. However, particularly for DUVs like the considered memory unit, this might lead to an infeasible large number of cases. Hence, a quality vs. efficiency trade-off has to be conducted.*

As an alternative, more elaborated blocking schemes can be applied in order to explicitly guide the SAT solver in determining representative cases. For this purpose, strategies proposed in [20] for constraint-based stimuli generation can be applied. One possible scheme is exemplarily illustrated next.

Example 6. *Consider again the DUV from Example 4. After a particular number of cases has been determined using the simple blocking scheme from Algorithm 1, a brief analysis on them is conducted. This may unveil that all cases derived thus far are composed of the primary inputs we and pc only. Hence, in order to increase the diversity of the determined solutions, an additional constraint* $s_{we} = 0 \vee s_{pc} = 0$ *might be added to the SAT instance. This forces the SAT solver to determine an assignment where either of the corresponding primary inputs assumes an observability don't care and, hence, is not part of a case. This would directly lead to the derivation of a case involving ext.*

Using more elaborated blocking schemes lead to a good trade-off between the diversity of the obtained cases and the effort required to determine them. Although those cases provide no guarantee that the behavior is comprehensively been triggered, they still lead to representative cases that may improve the overall coverage of stimuli-based verification. The trade-off between quality and efficiency has been shown in our experimental evaluation which is summarized next.

6. EXPERIMENTAL EVALUATION

The approach presented in the previous sections has been implemented in C++. As SAT solver, we utilized MiniSAT [21]. In order to evaluate the performance with respect to both, efficiency and quality, we conducted several experiments. In the following, we present the results obtained by considering

- a *Memory Management Unit* (MMU), i.e. an interface between a CPU and an external memory which manages the respective data transactions, and
- an *Arithmetic Logic Unit* (ALU), i.e. a 128-bit module which supports the standard logic and arithmetic operations.

Both designs have previously been applied for the evaluation of verification approaches (see e.g. [13]). For each benchmark, a suitable set of scenarios is considered, e.g. READ_access and WRITE_access targeting on the read and write operations of the MMU and ADD_overflow and ADD_out targeting on certain behaviors of the addition operation in the ALU (i.e. an overflow and the derivation of a

Table 1: Cases of Scenarios of the MMU

	Simple		Advanced	
Scenarios	$\lvert C^{S_i}\rvert$	Time (s)	$\lvert C^{S_i}\rvert$	Time (s)
MMU_idle	3	0.04	1	0.01
READ_issue	3	0.03	1	0.01
WRITE_issue	3	0.04	1	0.03
NO_access	4	0.06	2	0.04
READ_access	9	0.11	2	0.06
WRITE_access	5	0.09	2	0.03
MEM_data_0	5	0.09	2	0.06
MEM_data_1	5	0.08	2	0.05
MEM_data_2	13	0.16	2	0.04
TO_idle	4	0.06	2	0.03
TO_read	4	0.06	2	0.04
TO_write	5	0.07	2	0.03
TO_rw_pend	5	0.06	2	0.04
RACK	3	0.01	2	0.01
WBUF_full_0	5	0.09	2	0.06
WBUF_full_1	5	0.09	2	0.06
WBUF_full_2	7	0.08	2	0.04
WBUF_full_3	5	0.08	2	0.06

$|C^{S_i}|$: Number of cases determined by the proposed approach

specific value). Both benchmarks allow for a representative case study including designs with scenarios inheriting a relatively small set of cases (MMU) and a significantly large set of cases (ALU). All experiments have been conducted on a 64-bit AMD Athlon Dual Core machine with 4 GB of memory running Linux.

6.1 Efficiency of Case Determination

In a first series of experiments, we evaluated the efficiency of the proposed approach. For the evaluations, the threshold *max* has been set to the total amount of primary inputs and flip flops. Furthermore, the algorithm was terminated after a set of 100 cases has been determined.

The results are summarized in Table 1 and Table 2 for the MMU and the ALU, respectively. The first column (denoted by *Scenarios*) gives the identifier of the respectively considered scenario. The remaining columns provide the results obtained by the proposed approach, i.e. the number of cases which actually have been determined by the proposed approach (denoted by $|C^{S_i}|$) as well as the run-time required for that in CPU seconds (denoted by *Time*). We distinguish thereby between the results obtained by applying the simple blocking scheme as discussed in Example 5 and the ones derived by using the advanced blocking scheme as discussed in Example 6.

The results clearly show the efficiency of the proposed approach. For both DUVs, the desired cases can be determined in negligible runtime. The simple scheme works thereby as a pure workhorse: All cases of the scenarios of the MMU as well as the desired 100 cases of the scenarios of the ALU are determined. However, as discussed in Section 5 some of these cases only differ e.g. in their respective data inputs

Table 2: Cases of Scenarios of the ALU

	Simple		Advanced	
Scenarios	$\lvert C^{S_i}\rvert$	Time (s)	$\lvert C^{S_i}\rvert$	Time (s)
ADD_overflow	100	3.566	1	0.088
ADD_out	100	5.514	1	0.072
ADC_overflow	100	2.864	1	0.063
ADC_out	100	4.919	1	0.087
SUB_overflow	100	2.500	1	0.115
SUB_out	100	7.726	1	0.136
SBC_overflow	100	3.260	1	0.081
SBC_out	100	4.041	1	0.093
SHR_out	100	2.233	1	0.138
SHL_out	100	2.160	1	0.088
ROR_out	100	2.266	1	0.080
ROL_out	100	2.234	1	0.104

$|C^{S_i}|$: Number of cases determined by the proposed approach

and, hence, are not that helpful for coverage analysis. In contrast, the advanced scheme is capable of detecting these similarities and, hence, eventually returns a smaller set of representative cases only. Either way, the results of both schemes help to improve coverage in simulation-based verification by providing explicit information about the possible fashions in which a scenario can be triggered.

6.2 Benefit of Case Determination

In order to illustrate the benefits of the information obtained by the proposed approach, a second series of experiments has been conducted. Here, we considered stimuli generated for the MMU by the approach presented in [13]. This approach follows a coverage-driven stimuli generation scheme as reviewed in Section 2.1, i.e. scenarios are considered to be sufficiently covered when they have been triggered a certain amount of times (20 in this case). For the MMU, this is accomplish after a total number of 120 stimuli have been generated.

Having this set of stimuli, we evaluated how well they trigger the respective cases determined by the approach proposed in this work. Table 3 presents the respective numbers for this purpose. The first two columns denote the scenarios and the respective number of cases as determined in the first series of experiments. The next two columns present the number of stimuli which trigger the respective scenario and how many cases are covered by those. As can clearly be seen, the determined stimuli hardly provide a complete coverage of the scenarios. Although all scenarios have been triggered the given number of times, none of them got triggered by all possible cases. Moreover, for scenarios like *MMU_idle*, WRITE_issue, etc. always the same case got triggered.

Conclusions like those can only be drawn, if precise information on the cases is available. The approach proposed in this work is able of determining these information and, by this, helps to improve coverage-driven stimuli generation.

Table 3: Distribution of Stimuli with respect to Cases

Scenarios	$\lvert C^{S_i} \rvert$	#	$\lvert C_T^{S_i} \rvert$
MMU_idle	3	20	1
READ_issue	3	29	2
WRITE_issue	3	20	1
NO_access	4	20	1
READ_access	9	60	6
WRITE_access	5	40	4
MEM_data_0	5	20	3
MEM_data_1	5	20	1
MEM_data_2	13	80	7
TO_idle	4	20	1
TO_read	4	20	2
TO_write	5	40	4
TO_rw_pend	5	40	4
RACK	3	78	2
WBUF_full_0	5	20	4
WBUF_full_1	5	20	3
WBUF_full_2	7	40	3
WBUF_full_3	5	40	4

$\lvert C^{S_i} \rvert$: Number of cases of S_i
#: Number of stimuli triggering the scenario
$\lvert C_T^{S_i} \rvert$: Number of triggered cases

7. CONCLUSION

In this work, we proposed an automatic approach for determining all cases in which a scenario can be triggered. For this purpose, the implicative power of solvers for Boolean satisfiability has been utilized. Examples illustrated the advantages of the proposed method, while experimental evaluations confirmed its efficiency and benefits. The information derived by the proposed approach is crucial in order to improve coverage in simulation-based verification. In this sense, this work represents a first step towards case determination. Future work will focus on improving the efficiency and, hence, the applicabaility of the approach for more complex circuitry.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SPECifIC under grant no. 01IW13001 as well as the German Research Foundation (DFG) within a Reinhart Koselleck project under grant no. DR 287/23-1.

8. REFERENCES

[1] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 118–149, 2003.

[2] D. Große, U. Kühne, and R. Drechsler, "Analyzing functional coverage in bounded model checking," *IEEE Trans. on CAD*, vol. 27, no. 7, pp. 1305–1314, 2008.

[3] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models.* Kluwer Academic Publishers, 2003.

[4] A. Piziali, *Functional Verification Coverage Measurement and Analysis.* Springer, 2004.

[5] S. Ur and Y. Yadin, "Micro architecture coverage directed generation of test programs," in *Design Automation Conf.*, 1999, pp. 175–180.

[6] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Design Automation Conf.*, 2003, pp. 286–291.

[7] S. Asaf, E. Marcus, and A. Ziv, "Defining coverage views to improve functional coverage analysis," in *Design Automation Conf.*, 2004, pp. 41–44.

[8] H. Azatchi, L. Fournier, E. Marcus, S. Ur, A. Ziv, and K. Zohar, "Advanced analysis techniques for cross-product coverage," *IEEE Trans. on Comp.*, vol. 55, no. 11, pp. 1367–1379, 2006.

[9] J. Yuan, C. Pixley, and A. Aziz, *Constraint-based Verification.* Springer, 2006.

[10] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *AI Magazine*, vol. 28, no. 3, pp. 13–30, 2007.

[11] Y. Katz, M. Rimon, A. Ziv, and G. Shaked, "Learning microarchitectural behavious to improve stimuli generation quality," in *Design Automation Conf.*, 2011, pp. 848–853.

[12] C. Ioannides and K. Eder, "Coverage directed test generation automated by machine learning - a review," *ACM Trans. on Design Automation of Electronic Systems*, vol. 17, no. 1, pp. 7:1–7:21, 2012.

[13] S. Yang, R. Wille, D. Große, and R. Drechsler, "Coverage-driven stimuli generation," in *EUROMICRO Symp. on Digital System Design*, 2012, pp. 525–528.

[14] M. Chen, X. Qin, and P. Mishra, "Efficient decision ordering techniques for sat-based test generation," in *Design, Automation and Test in Europe*, 2010, pp. 490–495.

[15] M. Chen and P. Mishra, "Decision ordering based property decomposition for functional test generation," in *Design, Automation and Test in Europe*, 2011, pp. 1–6.

[16] A. Sülflow, G. Fey, C. Braunstein, U. Kühne, and R. Drechsler, "Increasing the accuracy of sat-based debugging," in *Design, Automation and Test in Europe*, 2009, pp. 1326–1331.

[17] G. Fey and R. Drechsler, "Efficient hierarchical system debugging for property checking," in *Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2005, pp. 41–46.

[18] G. Tseitin, "On the complexity of derivation in propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, 1968, pp. 115–125, (Reprinted in: J. Siekmann, G. Wrightson (Ed.), Automation of Reasoning, Vol. 2, Springer, Berlin, 1983, pp. 466-483.).

[19] N. Eén and N. Sörensson, "Translating pseudo-Boolean constraints into SAT," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 1–26, 2006.

[20] R. Wille, D. Große, F. Haedicke, and R. Drechsler, "SMT-based stimuli generation in the SystemC verification library," in *Forum on Specification and Design Languages*, 2009, pp. 1–6.

[21] N. Eén and N. Sörensson, "An extensible SAT solver," in *Conference on Theory and Applications of Satisfiability Testing*, ser. LNCS, vol. 2919, 2004, pp. 502–518.