Stimuli Generation through Invariant Mining for Black-Box Verification

Luca Piccolboni
Department of Computer Science
University of Verona, Italy
Email: luca.piccolboni@univr.it

Graziano Pravadelli
Department of Computer Science
University of Verona, Italy
Email: graziano.pravadelli@univr.it

***Abstract*—A pre-condition for any verification technique based on simulation is the generation of a high-quality set of stimuli that effectively and efficiently cover the whole state space of the Design Under Verification (DUV), including hard-to-reach corner cases. To cope with this necessity, several approaches for the automatic generation of stimuli have been proposed for both embedded software and high-level descriptions of hardware components. Most of these approaches use constraint solvers to generate the sequences of stimuli that trigger specific conditions, enabling the analysis of corner cases. However, the automatic identification of those conditions is still an open problem, especially for black-box designs. To fill in the gap, this paper proposes a stimuli generator, based on a dynamic invariant miner, that identifies and stresses DUV areas that are not deeply analysed by traditional pseudo-random high-level Automatic Test Pattern Generators (ATPGs), thus guaranteeing an higher coverage of corner cases during black-box verification.**

I. Introduction

The rising complexity of embedded hardware and software highlights the need of improving the effectiveness and the efficiency of verification in all the phases of the design cycle. In particular, due to the increasing number of functionalities required in modern embedded systems, the generation of a high-quality set of stimuli has become even more a critical step of the verification process. The generation of effective stimuli is essential for ensuring the correctness of the designs in all working conditions, and it is the first step of all simulation-based techniques. A low-quality set of stimuli fails to discover design problems or bugs in the corner cases of the designs. This causes a false sense of safety on the correctness, and reduces the accuracy of the simulation-based techniques. For these reasons, several approaches for the generation of stimuli, at different abstraction levels, have been proposed [1]. They have been classified in *concrete*, *symbolic* and *concolic* executions.

Concrete execution uses the stimuli generated by random, probabilistic or genetic algorithms [2]. It allows to efficiently explore several execution paths of the Design Under Verification (DUV) but it is not exhaustive. Conversely, symbolic execution explores the DUV through an interpreter [3]. The interpreter follows each program path assuming symbolic values as inputs. Then, it creates formulas that express the conditions found in the code, and then it uses a constraint solver to obtain the actual values. It has been shown that symbolic execution is effective but more research is needed to make it scalable. Such problems have been addressed by concolic execution [4]–[6] where concrete and symbolic executions have been combined. The key idea is to simplify the symbolic expressions by using concrete values, to reduce the complexity of solving constraints.

To complement the previous techniques, some approaches used the *invariants* (or more generally, *operational abstractions* [7]) to enhance the verification effectiveness. Invariants are logic formulas that hold in a specific instant during the execution, that is for a certain time window, or between a couple of program points of an implementation, such as the entry and the exit point of a function. Such types of logic formulas are also used by concolic engines, e.g. [4], [6] to represent program execution paths. Most of the approaches, that are based on invariants, infer these logic formulas with *Daikon* [8] and they are used to find bugs in software programs. For example, Harder et al. [7] presented the operational difference technique for generating, augmenting and minimising set of stimuli for software programs. The key idea is that the stimuli that change the operational abstractions are useful to improve the verification effectiveness. In fact, these stimuli can exercise the program in different scenarios with respect to the stimuli that are not able to change the operational abstractions. As a consequence, with these stimuli, it is more likely to discover bugs. Xie and Notkin [9] proposed the use of operational violations. The stimuli that falsify the invariants are selected among the others because the violations can represent features of the program that have not been checked. *Eclat* [10] is an automatic framework that generates unit tests for `Java` classes. Specifically, the framework helps to select a small portion of the input stimuli, from an input set of stimuli, that is likely to reveal bugs. The invariants are used as an oracle, and the inputs that are considered interesting are those that create improper program behaviours. Lastly, Zeng et al. [11] used the invariants as a metric to understand if software programs are sufficiently verified. The algorithm iteratively generates new stimuli. If they do not change any inferred invariant, they are discarded.

Nevertheless, almost all of the previous approaches extract the invariants only for verifying if they are changed during the program execution, e.g. [7], [11]. Specifically, the stimuli that are able to change the invariants are considered more effective than the others. In this way, invariants are used only for *evaluating* the effectiveness of the generated stimuli. However, invariants are intended to represent behaviours and, therefore, they can represent particular verification scenarios. As a consequence, invariants can be used to *drive* the stimuli generation because, by falsifying or by satisfying certain invariants, the execution can be guided towards different execution paths, with the aim of discovering bugs in specific corner cases. As in the case of concolic engines with the formulas that represent execution paths, e.g. [4], [6]. Furthermore, all the previous approaches focus only on trace invariants, i.e. logic formulas that are satisfied in the whole execution of a program. But, *time window invariants* [12], i.e. logic formulas satisfied only in specific instants during the execution, can be used for the stimuli generation, because they represent the temporary conditions that are satisfied along specific execution paths. By

978-1-5090-3561-8/16/$31.00 ©2016 IEEE

stressing the conditions that have not been exercised enough during the simulation, a stimuli generator can identify new corner cases. Finally, some approaches, for example [10], have been used in the context of *white-box* verification, thus requiring the analysis of the DUV source code or becoming dependent on the considered programming language or abstraction level.

A. Contributions of the Paper

This work is intended to present a way to complement the existing stimuli generation techniques by using an invariant miner in the context of *black-box* verification. The goal of this work is proposing a *stimuli generator* that exploits the invariants extracted from an initial set of stimuli to effectively and efficiently guide the generation of new stimuli that increase the coverage of the state space of the DUV. Specifically, the approach extracts invariants during the DUV simulation. The types of invariants to extract is selected by the designer in according with the DUV specifications. Then, the invariants are automatically converted in constraints to represent specific execution paths. Finally, the proposed generator exercises the DUV with the values that satisfy those constraints, thus ensuring a more uniform exploration of the DUV state space.

B. Organisation of the Paper

The rest of the paper is organised as follows. Section II provides the fundamental concepts about invariant mining. Section III gives an overview of the stimuli generator which is detailed in Sections IV and V. Section VI deals with experimental results. Lastly, Section VII is devoted to conclusions.

II. Background

The invariant mining approaches can be orthogonally classified with respect to how invariants are inferred and which types of invariants can be extracted. First, the approaches can be either static or dynamic. The former explore design models or implementations to discover the invariants [13]. The latter infer the invariants from program execution traces [8], [12]. Second, some approaches can extract temporal propositions [14]. Others can infer only boolean or arithmetic propositions [8], [12]. In our approach, a dynamic invariant miner that extracts boolean and arithmetic propositions has been integrated [12]. In the following, the most related definitions are provided.

In the rest of the paper, given a model M, the set of *variables* V represents the set of inputs of M. The internal variables are not considered as part of V, because the approach targets black-box verification.

Definition 1. Given a model M, a set of variables V of M and a finite sequence of simulation instants $\langle t_1, \ldots, t_n \rangle$, an *execution trace* E of M is a finite sequence of pairs $\langle (V_1, t_1), \ldots, (V_n, t_n) \rangle$ where V_i, also called the i-th *sample*, is the evaluation of variables in V at simulation instant t_i.

Definition 2. Given a set of variables V and an execution trace $E = \langle (V_1, t_1), \ldots, (V_n, t_n) \rangle$, a *time window* for E is a finite subsequence of contiguous pairs $W_{i,j} = \langle (V_i, t_i), \ldots, (V_j, t_j) \rangle$ of E, where $1 \leq i \leq j \leq n$ and $i \neq 1 \vee j \neq n$ ($W_{i,j} \subset E$).

Definition 3. Given an execution trace E and a set of variables V a *trace invariant* (*t-invariant*) is a logical formula over V that is always satisfied in the execution trace (in each sample).

Definition 4. Given an execution trace E, a set of variables V and a time window $W_{i,j} \subset E$, a *time window invariant* (*tw-invariant*) is a logical formula over V that is always satisfied in the time window $W_{i,j}$ (in each simulation instant in $W_{i,j}$).

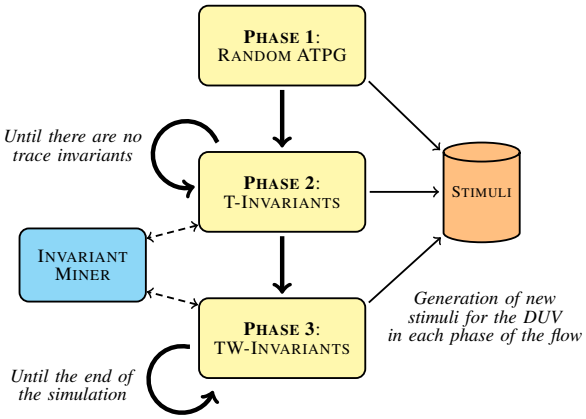

Fig. 1. Overview of the execution flow of the proposed stimuli generator.

Examples of invariants that are considered in this paper are: $\text{var}_1 < \text{var}_2$, $\text{var}_3 \neq 0$, $\text{var}_4 = true$, $\text{var}_5 = \text{var}_6 \ \& \ \text{var}_7$ etc.

III. Overview

The execution flow of the proposed stimuli generator is showed in Fig. 1. The objective of the framework is exploring in a uniform way the state space of the DUV to cover more easily the corner cases. The execution flow of the stimuli generator is divided into three *incremental* phases. The idea is that each phase contributes to enhance the set of stimuli for the DUV with complementary strategies of generation:

(i) PHASE 1 uses a pseudo-random Automatic Test Pattern Generator (ATPG) to initially explore the behaviour of the DUV. This step is important because with few random stimuli many or almost all functionalities of the DUV can be efficiently verified [15]. However, this phase can fail to check specific corner cases, for example those that rely on conditions that are hard to satisfy with random values. Thus, other more-focused strategies are necessary;

(ii) PHASE 2 creates stimuli that falsify all the trace invariants that are satisfied by the previous set of stimuli. The key idea is that the trace invariants can represent execution paths that have already been explored. By falsifying those invariants, other execution paths can be exercised with the aim of discovering the corner cases that have not been checked. Details on this phase are reported in Section IV;

(iii) PHASE 3 uses the time window invariants as a metric to understand which execution paths have been exercised less. The key idea is to exercise each path in a more uniform way independently from the probability of satisfying the conditions on which it relies. This augments the chance of checking all the corner cases in the different paths. Details on this phase are reported in Section V.

PHASES 2 and 3 of the proposed stimuli generator are based on an *invariant miner*. The invariant miner infers the trace and the time window invariants respectively. In the proposed framework, the invariant miner described in [12] has been adopted. It has also been extended to infer many other types of logic and arithmetic relations among variables and constants, and to count the total number of time windows that satisfy the time window invariants that have been extracted.

```
1  if (inputVar1 == 0) then
2  |  // A. Difficult to be executed
3  |  return function1(inputVar1, inputVar2);
4  else if (inputVar1 == inputVar2) then
5  |  // B. Difficult to be executed
6  |  return function2(inputVar1, inputVar2);
7  else
8  |  // C. Very easy to be executed
9  |  return function3(inputVar1, inputVar2);
10 end
```

Fig. 2. Example of hard-to-verify corner cases.

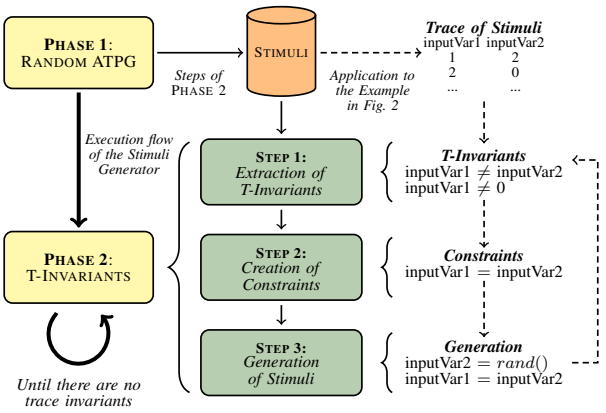

Fig. 3. Application of PHASE 2 to the Example in Fig. 2.

IV. T-INVARIANT-BASED GENERATION

In PHASE 2, the stimuli generator uses the invariant miner to extract the trace invariants from the set of stimuli[1] generated in PHASE 1. The idea is that *the trace invariants can represent the execution paths of the DUV that the random simulation has exercised*. According to Definition 3, the trace invariants are logic formulas that are satisfied in the whole simulation of the DUV. Thus, the trace invariants represent particular scenarios that have been verified. By falsifying the trace invariants, new execution paths can be discovered. In this way, the corner cases in those execution paths can be verified (as pointed out in [9]).

A. Motivating Example

Consider, for example, the code reported in Fig. 2. Suppose that the values of the variables are represented with 32 bits. In this case, it is very improbable that the random simulation is able to generate the value zero for the first input variable (inputVar1) or two identical values for both the input variables to test the function calls at line 3 or 6 respectively. However, by using the trace invariants, these function calls can be checked (Fig. 3). The trace invariants are inferred from the set of random stimuli, that has been previously used to exercise the code. Then, a subset of the trace invariants is selected, negated and converted into constraints (it is necessary to select only a subset of the invariants to avoid cyclic dependencies on the constraints as clarified later). Lastly, the constraints are used to generate the values that falsify the original trace invariants. Note that, in this case, only one invariant has been used (inputVar1 $\neq$ inputVar2). But, by repeating the process, also the other invariant can be selected. In this way, both the function calls can be verified.

It is important to note that the algorithm effectiveness in exploring the DUV state space depends on the invariants that the miner is able to extract. Designers have to choose a proper set of invariant templates (i.e. which types of relations the variables satisfy) that can be useful to discover all the DUV corner cases. It is sufficient, in most cases, to consider which are the exceptional situations that the DUV has to handle and derive the types of templates accordingly. In Fig. 2 two templates are considered: $v \neq u$ and $w \neq 0$ with $u, v, w \in V$.

[1]Note that in our approach the set of stimuli of PHASE 1 has been generated by using a pseudo-random ATPG. Nevertheless, any other type of ATPG (e.g. more sophisticated ATPGs) could be used to generate the initial set of stimuli.

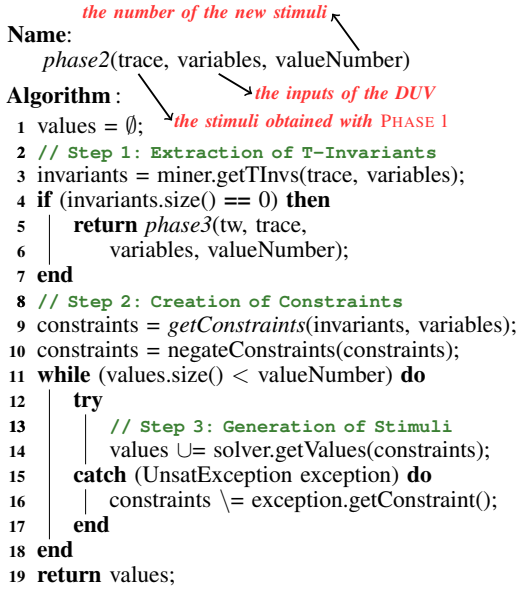

Fig. 4. The algorithm used in PHASE 2 of the flow in Fig. 1.

B. Stimuli Generation

The algorithm that automatises the flow in Fig. 3 is realised by the function *phase2*, depicted in Fig. 4. The main steps are:

1) Step 1: Extraction of T-Invariants: the trace invariants are inferred with the invariant miner. If all the trace invariants have already been falsified, the algorithm passes to PHASE 3 (line 5 of Fig. 4). Otherwise, the following steps are performed.

2) Step 2: Creation of Constraints: after the mining of the trace invariants, a set of constraints is created. The algorithm is depicted in Fig. 5. The function $getConstraints$ returns the largest set of constraints that can be obtained from the given set of invariants by using each variable at most one time in all the constraints. Each variable is used at most once because in this way the algorithm avoids to create cyclic dependencies on different constraints and some unsatisfiable situations. For example, the set of constraints $\{v_1 = v_2, v_2 = v_3, v_3 \neq v_1\}$ defines an unsatisfiable dependency on the generation of values (this comes from the invariants $\{v_1 \neq v_2, v_2 \neq v_3, v_3 = v_1\}$). Moreover, using distinct variables permits to generate stimuli that are uniformly distributed. For instance, consider again the flow reported in Fig. 3. Two trace invariants have been extracted, but only the first has been used at the first iteration of the algorithm. If both of them would have been selected at the same iteration, the only case, that satisfies both the constraints, consists of generating zero for both variables. But, it is clear that in the code of Fig. 2 this would have not been let to test both the function calls. In general, it is better to limit the dependencies among the selected constraints to have the possibility of generating uniformly-distributed sets of stimuli.

```
Name:
    getConstraints(invariants, variables)        ← the inputs of the DUV
Algorithm:                                       ← the set of inferred invariants
1  usedVars = ∅;
2  constraints = ∅;
3  foreach inv in invariants do
4      if (inv.getVariables() ∩ usedVars == ∅) then
5          constraints ∪= convertConstraint(inv);
6          usedVars ∪= inv.getVariables();
7      end
8      if (usedVars.size() == variables.size()) then
9          return constraints;
10     end
11 end
12 return constraints;
```

Fig. 5. The algorithm used in Step 2 of Fig. 4.

3) Step 3: Generation of Stimuli: to generate the stimuli a constraint solver is used. The constraints obtained from the trace invariants are converted in the format of SMT-LIB [16] (at line 5 of Fig. 5) and a model (i.e. a set of assignments for all the variables), in which all these constraints are satisfied, is determined with the SMT solver *MathSAT* [17]. However, some constraint solvers, in the state of the art, do not always allow to obtain uniformly-distributed models that satisfy a given constraint. They often return the same assignments for a set of constraints unless the constraints are not modified. Other constraint solvers can return the different models in which the constraints are satisfied but without a uniform distribution. Nevertheless, this would be a valuable feature for the stimuli generation because with different values (that satisfy the same expressions) it is more likely that specific parts of the code can be exercised better. For instance, consider the case of fault simulation where a 32-bit variable is injected with stuck-at faults. To activate a certain fault (i.e. by assigning a value to the injected variable so that the value of the bit injected differs from the original one), it is essential to generate highly-distributed values. In this way, the faults that are both in the most and in the least significant bits can be activated and found.

Thus, to generate highly-distributed values (independently from the solver, that could be changed for performance reasons), given an invariant, a variable is selected. Then, this variable is constrained so that it has to falsify that logic formula. For all the other variables, the algorithm generates random values instead. In this way, several assignments can be always obtained. Consider the flow in Fig. 3 and the constraint inputVar1 = inputVar2 (that derives from the corresponding negated invariant). The first variable is chosen so that it has to satisfy the constraint. For the other variable a random value is generated. In this way, by generating different values, many uniformly-distributed assignments can be obtained. However, it can happen that for certain values the constraint cannot be satisfied. If this happens, the constraint is removed (line 16 in Fig. 4). It will be considered later if the corresponding invariant has not been already falsified. The constraints that are always unsatisfiable (e.g. two variables with disjoint ranges that must be equal) are statically identified and discarded (for performance reasons).

V. TW-Invariant-Based Generation

In this phase, the stimuli generator uses the invariant miner to extract the time window invariants from the previous stimuli (of Phases 1-2). For each time window invariant the number of time windows (also called number of occurrences), that satisfy it, is also calculated. The key insight is that *the number of time windows that satisfy a time window invariant is a quantitative measure of how much the execution paths, where that invariant is verified, have been stressed*. According to Definition 4, a time window invariant is satisfied only in a specific time window of a given length. If the number of occurrences for a given invariant is too much low compared to the others, probably this invariant has not been stressed enough (e.g. if it is hard to satisfy). On the other hand, if the number of occurrences is high, it is likely that the corresponding execution paths have been sufficiently exercised. Therefore, in this phase the algorithm tries to stress the time window invariants with a low number of occurrences with the aim of discovering the corner cases that reside in the corresponding execution paths.

```
1  if (inputVar1 == inputVar2) then
2      if (inputVar3 == 8) then
3          // A. Very difficult to be stressed
4          return function1(inputVar1, inputVar2);
5      else
6          // B. Difficult to be stressed
7          return function2(inputVar1, inputVar2);
8      end
9  else
10     // C. Very easy to be stressed
11     return function3(inputVar1, inputVar2);
12 end
```

Fig. 6. Example of paths with different probabilities to be exercised.

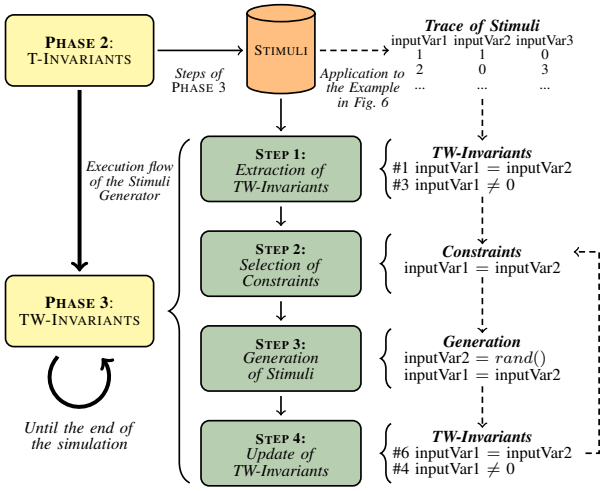

Fig. 7. Application of PHASE 3 to the Example in Fig. 6.

A. Motivating Example

Consider the code reported in Fig. 6. Suppose that the values of the variables are represented with 32 bits. In this case, it is very unlikely that all the execution paths are stressed in a uniform way. In fact, as in the example of Fig. 2, the first branch (lines 2-8) has a lower probability of being exercised. However, by using the time window invariants, the visit of this code can be better uniformed. In fact, it is often not sufficient to visit a path only one time. After a new execution path has been discovered (e.g. with PHASE 2) it is necessary to visit it multiple times to discover all the corner cases that can be nested inside, as in the case of the first branch. Moreover, in the case of fault simulation this is even more important because, in this way, different values that satisfy the same invariants can be brought in specific parts of the code. For these reasons, the time window invariants are inferred from the set of stimuli, that has been used to exercise the code (Fig. 7). Then, a subset of the time window invariants, that have the lowest number of

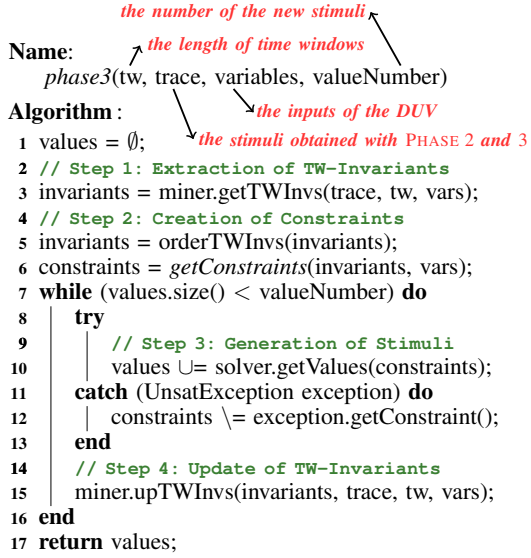

Fig. 8. The algorithm used in PHASE 3 of Fig. 1.

occurrences, is selected and converted into constraints. Lastly, the constraints are used to generate the values that satisfy the original time window invariants. Note that, in this case, only one time window invariant has been selected. But, by repeating this process, the number of occurrences of the time window invariants (that is updated as soon as new stimuli are generated) can be made very similar, allowing to exercise all the different execution paths and the corner cases in a more uniform way.

B. Stimuli Generation

The algorithm that automatises the flow in Fig. 7 is realised by the function *phase3*, depicted in Fig. 8. The main steps are:

1) Step 1: Extraction of TW-Invariants: instead of mining the trace invariants, in this phase, the algorithm infers the time window invariants. In particular, all the time window invariants for a specific time window length are extracted. The length of the time windows allows to change the order with which the invariants are selected at every iteration of the algorithm. In our approach, it is sufficient to consider small time windows, up to five samples. With longer time windows, the number of occurrences, for all the time window invariants, goes to zero [12], making the history of the previous stimuli completely meaningless. On the contrary, with smaller time windows, it is possible to verify which invariants have been satisfied less by the set of stimuli from which they have been extracted. As a result, the less-exercised executions paths can be explored.

2) Step 2: Creation of Constraints: the algorithm used for selecting the constraints is the same used in the previous phase and reported in Fig. 5. The only difference is that the invariants are ordered with respect to the number of occurrences before they are selected by the algorithm (line 5 in Fig. 8). In this way, at each iteration of the loop in Fig. 5, the time window invariants with the lowest number of occurrences are chosen with respect to the variables that are still free (i.e. not used in any other constraint). This allows to stress the corner cases that have been exercised less by the previous set of stimuli.

3) Step 3: Generation of Stimuli: the algorithm used for generating the stimuli is similar to that used in PHASE 2 (see Section IV-B3), but the inferred invariants are not negated.

BENCH. NAME	PRIMARY INPUTS	PRIMARY OUTPUTS	COMP. NLOC	COMP. CCN	SIMULATION TIME (2M)
CRC	55	34	∼800	22	250.65 *sec*
MUL	65	32	∼800	17	261.89 *sec*
DIV	99	65	∼200	11	197.99 *sec*
GCD	65	32	∼100	8	2196.14 *sec*
MUX	214	97	∼200	9	213.28 *sec*

TABLE I
CHARACTERISTICS OF BENCHMARKS

4) Step 4: Update of TW-Invariants: the number of occurrences of each time window invariant is updated in order to reflect the current set of stimuli. Specifically, to update the number of occurrences it is sufficient to consider all the new stimuli that have been generated and some of the stimuli that has been previously used depending on the length of the time windows. On this set of stimuli, the time window invariants are checked and the occurrences are updated accordingly.

VI. EXPERIMENTAL RESULTS

The effectiveness of the proposed stimuli generator has been evaluated on a set of `SystemC` RTL HW components. The effectiveness of the approach has been evaluated by measuring the achieved fault coverage. The injected faults realise the bit coverage model [18], that is an RTL abstraction of the well-known stuck-at model generally used at gate level.

A. Description of Benchmarks

The most important characteristics of the considered benchmarks are reported in TABLE I. The first column reports the names of the benchmarks while the second and third show the sizes of input and output ports in bits. The first benchmark performs the *cyclic redundancy check*, a common error detecting technique used in digital networks or store devices. The second design is a floating point unit that calculates the *multiplication* of two given floating point numbers specified with the IEEE 754 standard. The next two realise calculations using unsigned integers. Specifically, they calculate respectively the *integer division* and the *greatest common divisor* of two given inputs. Finally, the last one is a *bus multiplexer* that can be found in common CPUs [19]. The fourth column reports the number of code lines. The fifth column shows the maximum *cyclomatic complexity* [20] among all the processes of each benchmark, obtained with *Lizard* [21]. According to [20], a value greater than 10 indicates a code hard to check. Finally, the last column, as a way to understand the cost for the pure simulation of the considered benchmarks, reports the time required for a fault-free simulation consisting of two millions of input stimuli. Note that GCD can be particularly slow in case of large unsigned integers.

B. Stimuli Generation

The result of the stimuli generation is reported in TABLE II[2]. The first column shows the names of the benchmarks, while the second reports the total number of bit coverage faults that have been injected in the source code. The untestable faults have not been considered. The proposed stimuli generator has been compared with a pseudo-random ATPG that uses an incremental approach: it generates $2 * n$ new input stimuli if the last n stimuli have detected at least one fault, with

[2]The experimental results have been carried out on an Intel Xeon E5649 @2.53Ghz equipped with 8 GB of RAM and running Ubuntu 12.04. The length of the time windows that has been used for the experiments of PHASE 3 is *2*. Finally, all the reported experimental results are the average of 10 tests.

Bench. Name	Stuck-at Faults	Pseudo-Random ATPG I			Pseudo-Random ATPG II			Proposed Approach			
		#Stimuli	%Cov.	Time	#Stimuli	%Cov.	Time	#Stimuli	%Cov.	#Invs	Time
CRC	616	100000	57.9%	1650.66 *sec*	200000	72.0%	3188.97 *sec*	9932	73.4%	56	218.68 *sec*
MUL	918	1581	98.7%	94.67 *sec*	100000	98.7%	5561.46 *sec*	990	99.8%	21	37.03 *sec*
DIV	782	32212	64.9%	85.71 *sec*	100000	65.5%	251.36 *sec*	4670	98.6%	22	81.14 *sec*
GCD	393	3629	78.9%	734.50 *sec*	100000	79.8%	10215.86 *sec*	800	100.0%	4	66.34 *sec*
MUX	844	87	99.6%	0.44 *sec*	100000	99.6%	211.19 *sec*	91	100.0%	210	1.20 *sec*

TABLE II
Experimental Results for the Benchmarks in Table I

at least 50% of fault coverage, but up to a maximum of 100000 stimuli. The number of generated stimuli, the achieved fault coverage and the execution time are reported between the third and the fifth columns (Pseudo-Random ATPG I). Then, each benchmark has been further stressed with the same pseudo-random ATPG, but asking it to generate a larger set of stimuli. The results are reported in the following three columns (Pseudo-Random ATPG II). Specifically, for the last four benchmarks a generation of 100000 stimuli has been carried out, since in the previous generation (Pseudo-Random ATPG I) this limit has not been reached, while 200000 stimuli were generated for the CRC. The additional stimuli generated by Pseudo-Random ATPG II were not useful to sensibly increase the fault coverage, for all benchmarks, but CRC. In fact, in these cases, the probability of discovering the remaining faults is very low because their activation and propagation require to exercise hard-to-traverse execution paths. However, the generation of those additional stimuli has an important impact on the performance, since the execution times significantly grow. Finally, the final part of TABLE II (Proposed Approach) reports experimental results related to the proposed approach, i.e., respectively, the number of generated stimuli, the achieved fault coverage, the number of invariants and the total execution time of the proposed ATPG. For the last four benchmarks, the proposed approach is able to find a higher number of faults with a lower number of stimuli with respect to the pseudo-random ATPG, reaching 100% of coverage for the last two benchmarks. Thanks, to the trace invariants and the time window invariants, the proposed approach is able to exercise the execution paths that have not been exercised enough in the previous simulations, guaranteeing a more uniform exploration of the state space of the DUV. For example, in the case of MUL, DIV and GCD exceptional behaviours in the arithmetic calculations have been covered (like division by zero or multiplication of two infinite numbers). In the case of MUX, the randomly generated stimuli fail to test the situation in which the two input registers are the same. On the other hand, with the proposed approach this situation has been verified, by specifying the corresponding template $u = v$ with $u, v \in V$. Finally, in the case of CRC, approximately the same fault coverage of the pseudo-random ATPG is achieved, but a reduction of one order of magnitude in the number of generated stimuli is observed. Thanks to a lower number of stimuli the execution times for the proposed approach remain quite low in all the cases, compared to the random generation.

C. Limitations

The effectiveness of the proposed approach has been shown on control-dominated designs or designs that heavy rely on arithmetic computations. These designs more easily highlight the effectiveness of the proposed approach since it is clear how to define the templates that permit to drive the simulation towards specific paths. For other kinds of designs, e.g. data-dominated designs, it could be more difficult to find such invariant templates. The applicability of the proposed technique to different kinds of designs will be addressed in future work.

VII. Concluding Remarks

This paper presented an invariant-based stimuli generator for a more uniform exploration of the DUV state space, thus guaranteeing an higher coverage of corner cases. The framework (i) uses an invariant miner to dynamically extract invariants from an initial set of stimuli, (ii) converts the invariants into constraints, and (iii) exploits a constraint solver to automatically generate the values that satisfy these constraints to generate further stimuli that cover DUV areas not uniformly explored. Experimental results showed the effectiveness of the approach compared with a pseudo-random ATPG both in terms of achieved fault coverage, number of generated stimuli and execution times.

References

[1] S. J. Galler and B. K. Aichernig, "Survey on test data generation tools," *International Journal on Software Tools for Technology Transfer*, 2014.
[2] W. Jiawen, L. Zhigui, W. Suliang, L. Yang, L. Yufei, and Y. Hao, "Coverage-directed stimulus generation using a genetic algorithm," in *Proc. of IEEE ISOCC*, 2013, pp. 298–301.
[3] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Communications of the ACM*, pp. 82–90, 2013.
[4] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proc. of OSDI*, 2008, pp. 209–224.
[5] G. D. Guglielmo, M. Fujita, F. Fummi, G. Pravadelli, and S. Soffia, "EFSM-based model-driven approach to concolic testing of system-level design," in *Proc. of ACM/IEEE MEMOCODE*, 2011, pp. 201–209.
[6] B. Lin, Z. Yang, K. Cong, and F. Xie, "Generating high coverage tests for SystemC designs using symbolic execution," in *Proc. of ACM/IEEE ASP-DAC*, 2016, pp. 166–171.
[7] M. Harder, J. Mellen, and M. D. Ernst, "Improving Test Suites via Operational Abstraction," in *Proc. of ACM/IEEE ICSE*, 2003, pp. 60–71.
[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," in *Proc. of ACM/IEEE ICSE*, 1999, pp. 213–224.
[9] T. Xie and D. Notkin, "Tool-assisted unit test selection based on operational violations," in *Proc. of ACM/IEEE ASE*, 2003, pp. 40–48.
[10] C. Pacheco and M. D. Ernst, "Eclat: Automatic Generation and Classification of Test Inputs," in *Proc. of ACM ECOOP*, 2005, pp. 504–527.
[11] F. Zeng, C. Deng, and Y. Yuan, "Assertion-Directed Test Case Generation," in *Proc. of WCSE*, 2012, pp. 41–45.
[12] A. Danese, L. Piccolboni, and G. Pravadelli, "A parallelizable approach for mining likely invariants," in *Proc. of ACM/IEEE CODES+ISSS*, 2015, pp. 193–201.
[13] C. Flanagan, R. Joshi, and K. R. M. Leino, "Annotation Inference for Modular Checkers," *Inf. Process. Lett.*, pp. 97–108, 2001.
[14] J. Yang and D. Evans, "Dynamically Inferring Temporal Properties," in *Proc. of ACM SIGPLAN-SIGSOFT PASTE*, 2004, pp. 23–28.
[15] J. W. Duran and S. C. Ntafos, "An Evaluation of Random Testing," *IEEE Transactions on Software Engineering*, pp. 438–444, 1984.
[16] SMT-LIB. [Online]. Available: http://smtlib.cs.uiowa.edu/
[17] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *Proc. of TACAS*, Berlin, Heidelberg, 2013, pp. 93–107.
[18] F. Fummi, C. Marconcini, and G. Pravadelli, "Functional fault coverage: The chamber of secrets or an accurate estimation of gate-level coverage?" *Proc. of IEEE ETS*, pp. 154–159, 2004.
[19] OpenCores. [Online]. Available: http://opencores.org/
[20] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, pp. 308–320, 1976.
[21] Lizard. [Online]. Available: https://github.com/terryyin/lizard