Efficient Coverage-Driven Stimulus Generation Using Simultaneous SAT Solving, with Application to SystemVerilog

AN-CHE CHENG, National Chiao Tung University
CHIA-CHIH (JACK) YEN, Synopsys Taiwan Co., Ltd.
CELINA G. VAL, SAM BAYLESS, and ALAN J. HU, University of British Columbia
IRIS HUI-RU JIANG, National Chiao Tung University
JING-YANG JOU, National Central University and National Chiao Tung University

SystemVerilog provides powerful language constructs for verification, and one of them is the *covergroup* functional coverage model. This model is designed as a complement to assertion verification, that is, it has the advantage of defining cross-coverage over multiple coverage points. In this article, a coverage-driven verification (CDV) approach is formulated as a simultaneous Boolean satisfiability (SAT) problem that is based on covergroups. The coverage *bins* defined by the functional model are converted into Conjunction Normal Form (CNF) and then solved together by our proposed simultaneous SAT algorithm PLNSAT to generate stimuli for improving coverage. The basic PLNSAT algorithm is then extended in our second proposed algorithm GPLNSAT, which exploits additional information gleaned from the structure of SystemVerilog covergroups. Compared to generating stimuli separately, the simultaneous SAT approaches can share learned knowledge across each coverage target, thus reducing the overall solving time drastically. Experimental results on a UART circuit and the largest ITC benchmark circuits show that the proposed algorithms can achieve 10.8x speedup on average and outperform state-of-the-art techniques in most of the benchmarks.

Categories and Subject Descriptors: B.7.3 [**Integrated Circuits**]: Reliability and Testing—*Test generation*

General Terms: Algorithms, Verification

Additional Key Words and Phrases: Formal methods, covergroup, simultaneous satisfiability

ACM Reference Format:
An-Che Cheng, Chia-Chih (Jack) Yen, Celina G. Val, Sam Bayless, Alan J. Hu, Iris Hui-Ru Jiang, and Jing-Yang Jou. 2014. Efficient coverage-driven stimulus generation using simultaneous SAT solving, with application to systemVerilog. ACM Trans. Des. Autom. Electron. Syst. 20, 1, Article 7 (November 2014), 23 pages.
DOI: http://dx.doi.org/10.1145/2651400

1. INTRODUCTION

The rapidly increasing complexity of modern System-on-Chip (SoC) designs and urgent time-to-market requirements have made verification a major bottleneck in the design

A preliminary version of this article [Cheng et al. 2012] appeared in *Proceedings of the IEEE International High-Level Design Validation and Test Workshop (HLDVT12).*
This work is supported by the National Science Council under grant NSC101-2221-E-008-137-MY3 and grant NSC101-2628-E-009-012-MY2.
Authors' addresses: A.-C. Cheng (corresponding author), Department of Electronics Engineering and Institute of Electronics, National Chiao Tung University, Hsinchu 30010, Taiwan; email: guam75111@gmail.com; C.-C. (J.) Yen, Verification Group of Synopsys Taiwan Co., Ltd., Hsinchu 300, Taiwan; C. G. Val, S. Bayless, and A. J. Hu, Department of Computer Science, University of British Columbia, Vancouver, BC V6T 1Z4, Canada; I. H.-R. Jiang, Department of Electronics Engineering and Institute of Electronics, National Chiao Tung University, Hsinchu 30010, Taiwan; J.-Y. Jou, Department of Electrical Engineering, National Central University, Taoyuan 32001, and the Department of Electronics Engineering and Institute of Electronics, National Chiao Tung University, Hsinchu 30010, Taiwan.

DOI: http://dx.doi.org/10.1145/2651400

flow. Simulation-based verification methods are the most widely used techniques in industry. However, they usually cannot exhaust the whole functionality of the design under verification (DUV). Therefore, constrained random verification techniques [Yuan et al. 2006] have been proposed to make the input stimuli more effective. They either guide the simulation toward untriggered design behaviors [Yeh and Huang 2010] or make the input patterns more evenly distributed [Wu et al. 2011]. Nevertheless, hard-to-reach corner cases may stay untouched after random verification. In this context, coverage-driven verification (CDV) [Benjamin et al. 1999] can dynamically bias the constraints of stimuli generation based on the coverage status. Since deriving stimuli manually requires deep comprehension of the DUV, which is very error prone and laborious, automatic approaches for stimuli generation are desired.

In this article, we use Boolean satisfiability (SAT)-based techniques to quickly generate stimuli for the functional coverage model of SystemVerilog, which would be incorporated into a modern digital design flow. The functional coverage model of SystemVerilog specifies certain scenarios on a set of signals as coverage targets, such as specific value occurrences or value transitions, or even the cross-coverage of these value occurrences and/or transitions.

To use a SAT-based method, the functional coverage model needs to be translated into Conjunction Normal Form (CNF). Hence, the conversion techniques that we have proposed in Cheng et al. [2012] are also presented in this article for completeness. Briefly speaking, a coverage target is converted into a set of clauses, and there is a representative variable in the clauses that is assigned *true* if and only if the coverage target can be triggered by the generated stimulus.

A problem occurs when a functional coverage model contains a large set of coverage targets: using typical directed test generation methods, each stimulus will be derived separately, which is obviously not efficient. Recall that these coverage targets are all associated with design behaviors. Each time that the SAT solver gets a stimulus for a target, it learns additional knowledge about the DUV. Incremental SAT solvers are able to retain this knowledge over multiple queries, and this can lead to major speedups. However, the constraints must be formulated carefully to ensure the correctness of the learned information for the later queries. Accordingly, we propose two SAT-based algorithms, the Property List Narrowing SAT (PLNSAT) and the Grouped Property List Narrowing SAT (GPLNSAT) algorithms, that can simultaneously solve multiple targets (make multiple representative variables be *true*) in one SAT query and then, as targets are solved, incrementally refocus the solver toward the remaining targets efficiently and correctly. PLNSAT is the general algorithm and applicable to any coverage-driven stimulus generation, or other problem requiring solution of multiple, possibly conflicting properties; GPLNSAT further exploits the structure inherent in SystemVerilog covergroups.

The idea of solving multiple properties together was first proposed in Fraer et al. [2002] for Bounded Model Checking (BMC). The authors formed a conjunction of all the properties and checked the combined one. However, we cannot combine all the coverage bins into a single one for stimulus generation, since some of them may be contradictory. In the prior work [Cheng et al. 2012], we proposed the Minimum Rectangular Range Segmentation (MRRS) algorithm, that could check whether the specifications of the bins have intersections so these bins can be solved together. However, it may not gain benefit if not much overlap of specification can be found. Khasidashvili et al. [2005] proposed the Simultaneous SAT (SSAT) solver that maintains a watched list of solving objectives. It iteratively assumes an objective in the list can be solved and checks the validity of others in one solver run. This concept is inherited by the Multiple Similar Properties SAT (MSPSAT) algorithm [Franzén et al. 2010], which implements SSAT without any modifications to the SAT solver by using the solver's application programming interface (API) of assumptions. Our approaches, on the other hand, incrementally update solving constraints that facilitate the SAT solver to focus on the currently unsolved targets. Yang et al. [2013] also proposed a stimulus generation method that considers all properties in each SAT solve, but their aim is to generate a minimal set of stimuli, while we only care about the solving efficiency.

In the experimental results, we will demonstrate the improvements introduced by our approaches over the traditional single-target method. The effectiveness of the new approaches against the MRRS algorithm will also be presented. Furthermore, we will also show that our techniques can achieve better performance than MSPSAT in seven out of ten cases, while they also do not require any modifications of the SAT solver.

The rest of this article is organized as follows: The preliminaries and the problem formulation are introduced in Section 2. In Section 3, we will detail the implementation of translating SystemVerilog functional coverage constructs into CNF. Our simultaneous SAT solving algorithms are described in Section 4 and the proposed approaches are evaluated by experimental results in Section 5. In Section 6, we will give a comprehensive review of incremental SAT and simultaneous SAT techniques. Section 7 concludes this article.

2. PRELIMINARIES AND PROBLEM FORMULATION

In this article, the symbols "$\wedge$", "$\vee$", and "$\neg$" denote Boolean AND, OR, and COMPLEMENT operations, respectively.

This section first introduces an overview of the SystemVerilog functional coverage constructs. Then, a short introduction to modern SAT solvers is described. The problem formulation of our stimulus generation will be given in the end.

2.1. The SystemVerilog Functional Coverage Constructs

This section describes the basic syntax of the SystemVerilog functional coverage model, namely coverage groups, coverage points, and cross-coverage. In addition, we will also briefly describe the SystemVerilog Assertion (SVA) language to give a more complete view.

2.1.1. The Covergroup Construct. In a SystemVerilog-based testbench, monitoring a specific value occurrence or transition of a variable can be easily done by the SystemVerilog functional coverage model. The covergroup construct consists of functional coverage specifications. It can encapsulate multiple coverage points (sampled locations) that are monitored simultaneously during simulation. Each coverage point can further specify a set of bins associated with a range of values or a set of value transitions. A rough syntax of covergroup is as follows.

```
covergroup covergroup_id [coverage_event];
    <coverage_option>
    <cover_point>
    <cover_cross>
endgroup [: covergroup_id]
```

A *coverage_event* defines when a covergroup should be sampled; if it is not specified, users must trigger samplings via the built-in *sample*() method. Options control the behaviors of **covergroup**, **coverpoint**, or **cross**, such as the weight (used for computing coverage), the coverage goal (100% coverage by default), or the minimum number of times that it should be triggered (one, by default). Syntax of coverage points and cross-coverage is shown in the following sections.

2.1.2. Coverage Points. A coverage point specifies an integral expression (represented by one or more variables) to be covered and contains a set of bins associated with some sampling values or value transitions of the expression. The syntax is given as follows.

```
[cover_point_id:] coverpoint expression {
    wildcard bins            bin_id = {range_list} | translist;
    wildcard ignore_bins     bin_id = {range_list} | translist;
    wildcard illegal_bins    bin_id = {range_list} | translist;
}
```

A bin declared **wildcard** is specified in bit-level in which all **X**, **Z**, and **?** are treated as 0 or 1. For example, a bin that specifies its sampling value is less than four is shown next.

wildcard bins *less_than_4* = {4'b00XX};

The bin *less_than_4* is considered as 0000, 0001, 0010, or 0011, that is, from zero to three. An **ignored** bin specifies sampling values or value transitions that should be excluded from coverage computation, even if they are specified in other bins. On the other hand, an **illegal** bin is similar to an ignored bin, but a runtime error will be issued if the specified values or transitions occur.

Three kinds of repetition transition could be specified. First, the **consecutive repetition** is specified using: *trans_item* [* *repeat_range*]. The *trans_item* is repeated for *repeat_range* times continuously. For example, 2 [* 2:3] represents (2 =>2) or (2=>2 =>2). Second, the **goto repetition** can be specified using: *trans_item* [−> *repeat_range*]. The *trans_item* repeats *repeat_range* times not necessarily on continuous clock cycles, and the transition following the **goto repetition** must immediately follow the last repeated *trans_item*. For example, 1 => 2[−>3] => 4 means 1... =>2... =>2... =>2=>4, where the dots (...) denote any sequence that does not contain the value 2. Last, the **nonconsecutive repetition** is specified using: *trans_item* [= *repeat_range*]. It is very similar to the **goto repetition**, except that the transition following the nonconsecutive repetition could occur after any number of cycles, provided that the *trans_item* does not occur again. For example, 1 => 2[=3] => 4 is the same as 1... => 2... => 2... => 2... => 4.

2.1.3. Cross-Coverage. A cross-coverage between two or more coverage points could be specified in a covergroup by the cross-construct. These crossed coverage bins provide higher levels of abstraction of design behavior. The syntax for specifying cross-coverage is given as follows.

```
[cross_ id :] cross list_of_coverpoints {
    bins            bin_id = select_expression;
    ignore_bins     bin_id = select_expression;
    illegal_bins    bin_id = select_expression;
}
```

The *select_expression* specifies which bins of each coverage point should be crossed. For example, a bin that is the cross of bin a_1 of coverage point a and bin b_2 of coverage point b could be specified as follows.

bins c_0 = **binsof**($a.a_1$) **&&** **binsof**($b.b_2$)

2.1.4. The SVA Property Construct. Assertion is widely used for design validation. SystemVerilog also provides its own assertion syntax, namely the SVA language. In SVA, the basic usage of the property construct is as follows.

```
property property_id;
  <assertion_variable_declaration>
  [clocking_event] property_expr;
endproperty [: property_id]
```

The property can not only be used as ordinary assertion, that is, **assert property**, but also as **cover property** to get property coverage, just like a coverage bin.

Covergroup and cover property have their individual advantages: covergroup provides additional coverage options and can define cross-coverage, whereas cover property can precisely define complex sequential design behaviors such as a protocol, thus transition bins are rarely used. Practically, they are used complementarily in a testbench. In this article, we only consider covergroup. Readers interested in using SVA for BMC may refer to the work of Wille et al. [2008], or to the articles of Das et al. [2006], Long and Seawright [2007], Boulé and Zilic [2008], Kastelan and Krajacevic [2009], or Mammo et al. [2012], in which the authors synthesize SVA into hardware modules for verification.

Complete semantics of covergroup, coverage points, transition bins, cross-coverage, and SVA can be found in the IEEE standard for SystemVerilog [IEEE 2013].

2.2. Background of Modern SAT Solvers

Although SAT was the original NP-complete problem, modern SAT solvers can routinely solve very large instances in practice. The Davis-Putnam-Logemann-Loveland (DPLL) algorithm [Davis and Putnam 1960; Davis et al. 1962] is the basic search algorithm used by most SAT solvers. Modern SAT solvers enhance the original Boolean constraint propagation (BCP) [Zabih and Mcallester 1988] method and also use several new techniques such as conflict-driven learning [Marques-Silva and Sakallah 1999] and dynamic variable ordering [Moskewicz et al. 2001]. The SAT solvers accept a formula in CNF; such a formula is also called a SAT (or CNF) instance. The CNF is a conjunction of clauses; a clause is a disjunction of literals, where a literal is a Boolean variable or its negation. A solver tries to find a *satisfiable* assignment to all the clauses, namely a *model*; if there is no satisfiable solution, the SAT instance is said *unsatisfiable*.

A key characteristic of modern SAT solvers that we exploit in this article is that they expose an *incremental* API. Specifically, the practical efficiency of modern SAT solvers arises from the clauses and variable weights heuristically learned during the solving process. If we next need to solve a closely related SAT query, we would like to reuse as much of this learned information as possible, which will greatly improve the efficiency of solving the new problem, but we must not compromise logical correctness. A typical modern SAT solver's incremental interface allows adding clauses after solving a query while still retaining all learned information. Deletion of clauses is problematic, however, because the solver may have learned a great deal of facts relying on the deleted clauses. Nevertheless, it is possible to efficiently delete *unit clauses*, that is, clauses that contain only a single literal. By constructing our algorithms specifically to conform to this incremental API, the successive SAT queries in our algorithm can typically be solved much faster than they otherwise could have been with a conventional, nonincremental algorithm. A detailed discussion of these issues and related research is deferred to Section 6.

2.3. Problem Formulation

This section briefly formalizes the stimulus generation problem of this article.

Definition 1. A coverage bin b_i $(0 \leq i < n)$ defines a set of value occurrences, or value transitions, or cross of value occurrences and/or value transitions over the set of

DUV signals. The set of bins is denoted by

$$B = \{b_0, \ldots, b_{n-1} | n \geq 1\}.$$

Definition 2. A constraint c_j $(0 \leq j < m)$ is a random simulation constraint, or a functional constraint derived from the design specifications, over the set of DUV signals. For the specification of a constraint, the general hardware description language operators such as arithmetic operators, relation operators, and logical operators are used. The set of constraints is denoted by

$$C = \{c_0, \ldots, c_{m-1} | m \geq 1\}.$$

During the procedure of stimulus generation, the DUV will be unrolled incrementally. Because every coverage bin has its individual smallest bounds for reach, once a bin is *true* in a smaller bound, it could be ignored from later time frames to improve the solving efficiency. Hence, we address the following problem in this article.

Given a set of bins B, a set of constraints C, a DUV model D, and the known maximum bound K of the bins, efficiently generate a set of stimuli that satisfies all the bins $b_i \in B$ with arbitrary smallest bound k for each bin.

Essentially, a generated stimulus should satisfies the following Boolean formula.

$$I(s_0) \wedge \bigwedge_{t=0}^{K-1} T(s_t, s_{t+1}) \wedge \bigvee_{i=0}^{n-1} \bigvee_{t=0}^{K} b_i(s_t) \wedge C \tag{1}$$

This formula contains four parts: (1) $I(s_0)$ is the initial state; (2) $T(s_t, s_{t+1})$ represents the state transition from state s_t to state s_{t+1}; (3) $b_i(s_t)$ test whether any coverage bin could be *true* on state s_t; and (4) C is the set of given constraints. In Section 3, we will detail how to convert part 3 into CNF. Also recall that formula (1) is only a sufficient condition for the satisfaction of a stimulus, while our approaches would add extra constraints to boost the stimulus generation process.

2.3.1. A UART Register Access Coverage Example. Figure 1 shows a covergroup example [Mentor Graphics 2013]. Its purpose is to check that all valid register accesses of a Universal Asynchronous Receiver/Transmitter (UART) circuit have occurred. It has named bins for each of the valid register addresses and for each state of the read/write bit. These are crossed with an **ignore_bins** to ensure that the read-only LSR and MSR registers are not counted for write bit coverage. A constraint could be derived from the **ignore_bins** to prune the searching space.

$$!((addr == 8'\text{h}14 \,||\, addr == 8'\text{h}18) \,\&\&\, (we == 1))$$

3. IMPLEMENTATION DETAILS

This section details the implementations of covergroup-to-CNF conversion.

3.1. Mapping Bins to Circuit Signals

In the example of Figure 1, there are two variables, namely *we* and *addr*. We always add the definition of variables before covergroups, and insert instrument comments after the definition to indicate the corresponding circuit signals. Taking *addr* as example, the definition and comment are as follows.

$$bit[7:0]addr; \quad //PI : PADDR$$

This means the 8-bit variable *addr* will be mapped to the primary input *PADDR*. In our implementations, we only care about those bins associated with primary inputs/

```
covergroup reg_access_cg();
    RW: coverpoint we {
        bins read = {0};
        bins write = {1};
    }

    ADDR: coverpoint addr {
        bins data = {0};
        bins ier = {8'h4};
        bins iir_fcr = {8'h8};
        bins lcr = {8'hC};
        bins mcr = {8'h10};
        bins lsr = {8'h14};
        bins msr = {8'h18};
        bins div1 = {8'h1c};
        bins div2 = {8'h20};
    }

    REG_ACCESS: cross RW, ADDR {
        ignore_bins read_only =
            binsof(ADDR) intersect {8'h14, 8'h18} && binsof(RW) intersect {1};
    }
endgroup: reg_access_cg
```

Fig. 1. A covergroup example for UART register access coverage. © Mentor Graphics, the Verification Academy. The code is under the Apache 2.0 license.

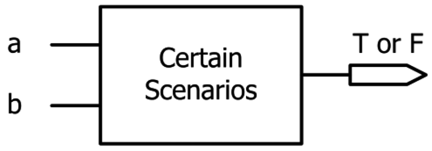


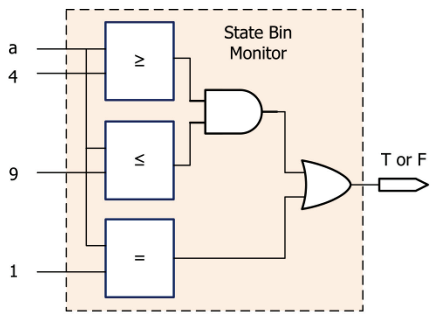
Fig. 2. The general monitor block.

outputs and flip-flops. It is possible to calculate coverage for testbench scope variables, but generating stimuli for them is trivial, that is, they can be directly assigned values in a testcase.

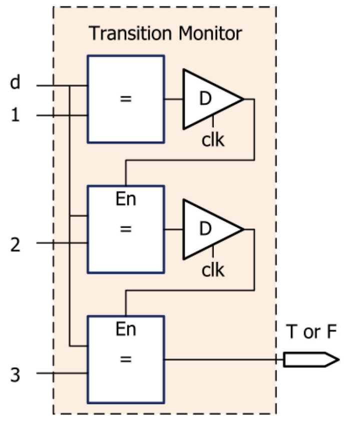
3.2. CNF Conversion

3.2.1. CNF Conversion of Circuit and Constraints. The Tseitin translation method [Tseitin 1970] can convert a propositional formula into CNF in polynomial time by introducing additional variables.

3.2.2. Translating Covergroup Constructs into Monitors. Because each bin associates with a set of sampled values or value transitions of one or more variables, it is intuitive to model a bin as a *monitor*. A monitor is a kind of circuit that checks certain scenarios and outputs *true* if the scenarios are matched, otherwise it outputs *false*. Figure 2 illustrates the block diagram of a monitor. In this section, we discuss how to convert the covergroup constructs into monitors, so they can be translated into CNF by existing methods [Tseitin 1970]. Some further restrictions are applied for practical considerations.



(a) *bins a* = {1, [4:9]}



(b) *bins d* = (1=>2=>3)

Fig. 3. The monitor for bins.

State (Value) Bins. A coverage bin that specifies a range of values could be converted into a set of comparators. The output signal of the monitor is the logical OR of all the comparators' outputs. Figure 3(a) shows an example of a state bin $a = \{1, [4{:}9]\}$.

Transition Bins. To check a sequence of transitions, the time frame in which the initial transition value occurs should be recognized first. Then, the following transition values should be checked consecutively.

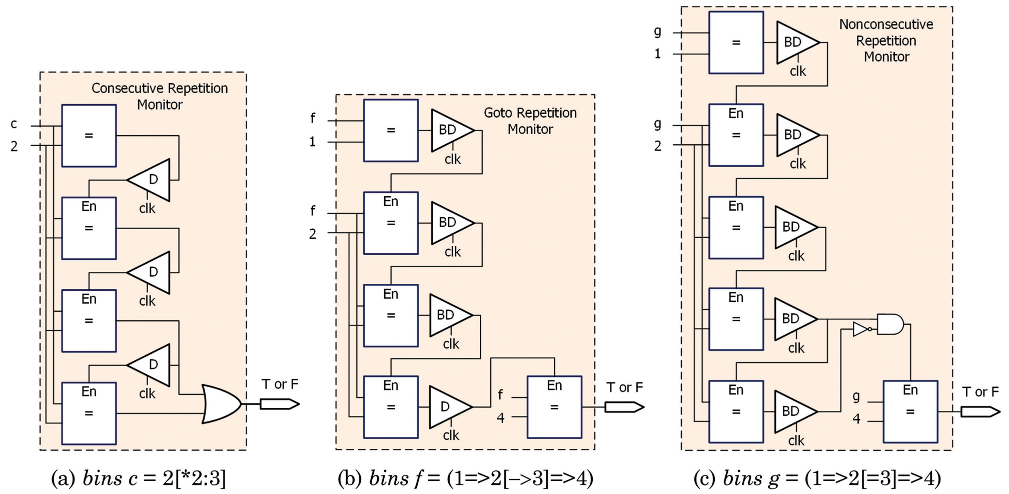
Figure 3(b) gives an example of a transition bin $d = (1 => 2 => 3)$. The triangular blocks labeled "D" are delay blocks that consist of a 1-bit register. If an equality comparator before a delay block outputs *true* in time frame t, then the delay block would output *true* in time frame $t + 1$, otherwise it would outputs *false*. An equality comparator with an enable input only checks the equality when the enable signal is *true*.

Conversion of a consecutive repetition transition is very similar to that of a transition sequence with different values. It could be more compact when a range of repetition is specified, as shown in Figure 4(a). The bin c specifies that value two consecutively repeats two or three times. Instead of constructing two modules for (2 => 2) and (2 => 2 => 2) respectively, a module of (2 => 2 => 2) is constructed and the module output is the logic OR of the second and the third equality comparators.

Before we describe how to convert the other two repetition transitions, we first introduce a small circuit called the blocking delay (BD) block. Different from a delay block, once its input signal is *true*, the blocking delay block's output signal blocks to *true* forever. It can be built by concatenating an OR gate and a 1-bit register. The output of the OR gate is the input of the register, and the output of register branches into the OR gate as one of the inputs.

To construct a **goto repetition** monitor, we first build a consecutive repetition monitor and then replace all but the last delay block to BD blocks. Figure 4(b) demonstrates a monitor for *bins* $f = (1 => 2[->3] => 4)$. When a BD block outputs *true*, it means that a sequence of transitions currently meet the specification, either consecutively or intermittently. The last transition must follow the last repetition value immediately, thus a delay block is used.

Finally, as discussed in Section 2.1.2, a nonconsecutive repetition differs from a **goto repetition** in that the transition follows the final repetition value. This transition


Consecutive Repetition Monitor
c
2
En
=
D
clk
T or F
Goto Repetition Monitor
f
1
2
BD
f
4
Nonconsecutive Repetition Monitor
g
1
2
g
4

(a) *bins c* = 2[*2:3] (b) *bins f* = (1=>2[->3]=>4) (c) *bins g* = (1=>2[=3]=>4)

Fig. 4. The monitors for repetition transition bins.

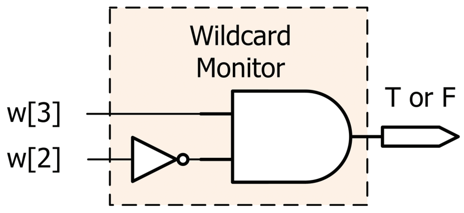


Fig. 5. The monitor for *bins* w = {4'b10XX}.

could occur after any other transitions, provided that the repetition value does not occur again. Such a difference is shown in Figure 4(c). An additional equality comparator is used to avoid an illegal repetition before the last transition.

Cross-Coverage Bins. A cross-bin includes a subset of bins from the coverage points specified in the *list_of_covepoint* syntax. Therefore, to build a cross-bin monitor, we just build all the monitors of its including bins, AND the output signals of crossed-bin monitors, and then OR all the output signals of the AND gates. This could be a very complicated circuit.

Ignored and Illegal Bins. The bins declared ignored or illegal should be excluded from coverage computation. We do not convert these bins but conduct a preprocessing to remove ignored/illegal values and sequences of transitions from any associated coverage bins.

Wildcard Bins. Wildcard bins are specified in bit-level. To convert it as a monitor, we just do logical AND for all its deterministic bits. For example, the monitor that checks a bin: w = {4'b10XX} is shown in Figure 5.

Clocking Events.. A clocking event that defines when a covergroup should be sampled may be a complex description. Generating test stimuli under such an event requires a lot of analytic effort. On the other hand, if a clocking is not even specified, samplings must be procedurally triggered by users. Recognizing these sampling points properly is also a hard task, therefore, a single clock signal *clk* for a DUV is assumed in this article, and all the covergroups are triggerd at the rising edge of this clock signal. As a result, we do not need to check the triggering points, and the solving procedure of stimuli is consistent with an unrolled DUV model.

Coverage Options. Coverage options could be defined at the **covergroup** or **coverpoint** (or **cross-**) syntactic level. They specify weights, target coverage goals, the minimum number of hits for each bin, etc. In this work, our target is to validate uncovered parts for one time at least, so the weights at different syntactic levels could be neglected. The minimum number of hits for each bin could simply be seen as a solving constraint of the iterative solving procedure, that is, the generated test patterns should guarantee to hit target bins for some threshold times. For simplicity, we currently do not incorporate options into the solving procedure.

4. STIMULUS GENERATION VIA SIMULTANEOUS SAT

This section details our SAT-based stimulus generation approach. It is an incremental solving process: a set of target bins are considered simultaneously by a SAT solver and, if the result is satisfiable, one or more solved bins are removed from the target list and then a single new clause is added, permitting the solver to continue incrementally from where it left off and thus benefitting from the learning already performed; if the result is unsatisfiable, the bound will be extended by one, the clauses will be modified slightly, and again the solver resumes solving incrementally. This procedure continues until all targets are solved or the given maximum bound is reached.

We propose two simultaneous SAT approaches, namely Property List Narrowing SAT (PLNSAT) and Grouped Property List Narrowing SAT (GPLNSAT). The term "property" is adopted instead of "bin" to make the concept more general, since each bin can be deemed as a property of a DUV.

Section 4.1 and Section 4.2 explain the single-pass PLNSAT and GPLNSAT algorithms, respectively. In Section 4.3, the complete PLNSAT and GPLNSAT algorithms are described.

4.1. Property List Narrowing SAT (PLNSAT)

To solve multiple bins at the same time, a constraint is introduced to the SAT instance, namely that at least one of the bins should be solved [Cheng et al. 2012; Yang et al. 2012, 2013]. Recall that all the bins are translated into CNF clauses as monitors. Since a bin monitor outputs *true* when its specifications are satisfied, conversely, we can restrict the output signal to be *true* and then try to find a stimulus that satisfies this constraint. Considering a set of bin monitors, we can OR all their outputs and then restrict the output of the OR gate to be *true*. If a stimulus can be derived from such a constraint, it means one or more bin monitors output *true*. This section will describe how the PLNSAT algorithm translates this constraint into CNF and then, after finding a solution, how the PLNSAT algorithm can refocus attention on the remaining properties in an efficient, incremental manner.

4.1.1. Solving Constraints Formulation. The constraint clauses can be generated by a conventional OR gate translation method, that will generate $n+2$ clauses for n bins ($n+1$ clauses for an n-input OR gate and a unit clause to restrict the output of the OR gate to be *true*). However, PLNSAT uses a succinct encoding by adding a single clause that contains all bins, that is, the *property list clause*. For example, if there are two bins b_1 and b_2 and the output of the OR gate is represented by the variable$y$, a general translation is $(\neg b_1 \vee y) \wedge (\neg b_2 \vee y) \wedge (b_1 \vee b_2 \vee \neg y) \wedge (y)$, while the property list clause is $(b_1 \vee b_2)$. Note that the variables b_i in the clauses represent the output of the i-th bin monitor. For simplicity, we just use the term "bin" instead of "the representative variable of the output of a bin monitor" in this article.

The efficient encoding via a property list clause itself is already known. But such an encoding has an additional benefit: by such a translation method, an efficient simultaneous SAT solving procedure can be derived. Consider a SAT instance that contains the following clause: $(b_1 \vee b_2 \vee b_3 \vee b_4 \vee b_5)$. Assume b_1 and b_4 are *true* in a satisfied solution, so the solving constraint would be changed to focus on b_2, b_3, and b_5 only. An intuitive but wrong way to change the constraint is to add two unit clauses $(\neg b_1)$ and $(\neg b_4)$ since, because there may be dependencies among the bins, blocking a bin could make other satisfiable bins to be unsatisfiable. For example, assuming b_2 is a cross-bin specified as b_1 && b_3, preventing b_1 will make b_2 unsatisfiable. The PLNSAT algorithm adds a *narrowed* property list clause, namely $(b_2 \vee b_3 \vee b_5)$, to formulate the new constraint instead. There are two effects by adding this clause: (1) if the SAT instance is satisfiable, at least one of b_2, b_3, and b_5 is *true*; (2) the solved bins b_1 and b_4 are not prohibited from the searching space, thus all the conflict clauses can be reused. And since the algorithm is only adding a clause and not removing any clauses, the SAT solver can continue incrementally, reusing all facts that it had learned already while solving for previous bins.

If the SAT instance is unsatisfiable under the new constraint, that is, $(b_1 \vee b_2 \vee b_3 \vee b_4 \vee b_5) \wedge (b_2 \vee b_3 \vee b_5)$, the bound will be extended and a new constraint clause for the next bound is added: $(b_2' \vee b_3' \vee b_5')$. However, the previous unsatisfiable constraints must be removed before new solving starts. Because we want to reuse all the learned conflict clauses, we do not actually remove the old constraint clauses, but just *disable* these clauses. Disabling a clause can be achieved by adding an auxiliary literal to the clause. Considering the clauses $(b_1 \vee b_2 \vee b_3 \vee b_4 \vee b_5 \vee a) \wedge (b_2 \vee b_3 \vee b_5 \vee a)$, if the variable a is *false*, the behavior of the clauses is just as the clauses $(b_1 \vee b_2 \vee b_3 \vee b_4 \vee b_5) \wedge (b_2 \vee b_3 \vee b_5)$; on the other hand, if the variable a is *true*, all the bins become *don't-care* for the clauses, which also means the constraints imposed on the bins are released. As in the satisfiable case, the SAT solver can continue incrementally.

4.1.2. The Single-Pass PLNSAT Algorithm. Algorithm 1 describes the generic method for single-pass PLNSAT and GPLNSAT. The inputs are a set of currently unsolved bins, a SAT instance (constructed from formula (1)), and an auxiliary variable. The stimulus is initialized in line 1. In line 2, the SAT instance is augmented by property list clauses, while the clauses are generated by the currently unsolved bins and an auxiliary variable. The two methods GenPLClause and GenGPLClause are described in Algorithms 2 and 3, respectively. The single-pass PLNSAT algorithm uses the GenPLClause method, which puts all unsolved bins and an auxiliary variable in a single clause to form the property list clause (line 1 of Algorithm 2). An assumption that $a = \mathit{false}$ is parsed to the SAT solver in line 3 and if there is an assignment that satisfies the SAT instance and the assumption in line 4, a stimulus s and the unsolved bins are analyzed in line 5 and returned in line 7. The extension of a SAT solver to accept a set of unit literals as assumptions can easily be added to any modern SAT solver [Eén et al. 2010]—the first n decisions should be made on the assumption literals.

4.2. Grouped Property List Narrowing SAT (GPLNSAT)

Recall that the PLNSAT algorithm embeds the constraint that at least one of the bins should be solved to the SAT instance. This simple constraint can be further refined to improve the stimulus generation flow [Cheng et al. 2012; Yang et al. 2012]. In Yang et al. [2012], design knowledge is introduced to the constraint: a set of unsolved properties that can be triggered simultaneously are formed into a *conjunction* R', such that if R' is satisfiable, then more than two properties are triggered by one stimulus. However, this method is not fully automatic, that is, the properties in the conjunctions are selected manually. In Cheng et al. [2012], the MRRS algorithm could

ALGORITHM 1: Generic Method for Single-pass PLNSAT and GPLNSAT

```
Input: Unsolved bins, $U$; A SAT instance, $CNF$; An auxiliary variable, $a$.
Output: A stimulus $s$; Unsolved bins, $U$.
1: $s = \emptyset$;
2: $CNF = CNF \cup$ Gen(G)PLClause $(a, U)$;
3: $assumps = \neg a$;
4: if SAT $(assumps, CNF) == TRUE$ then
5:     $(s, U) =$ Analyze();
6: end if
7: return $(s, U)$;
```

ALGORITHM 2: GenPLClause

```
Input: Unsolved bins, $U$; An auxiliary variable, $a$.
Output: The property list clause, $PLClause$.
// assume $b_i \in U = \{b_0, \dots, b_{n-1} | n \geq 1\}$
1: $PLClause = (\bigvee_{i=0}^{n-1} b_i \vee a)$
2: return $PLClause$;
```

find the intersections of specifications of the bins for efficient CNF conversion. However, empirically there are not many intersections among the specifications of bins in typical test plans. Because the MRRS algorithm needs to compute the intersections first, if there are few of them, the overall solving time may not be decreased. Another drawback of the MRRS algorithm is that it can only handle two-dimensional cross-coverage (cross-coverage of two coverpoints), while there may be three- or higher-dimensional cross-coverage specifications. Comparisons between MRRS and (G)PLNSAT will be demonstrated in Section 5.

In this section, we describe the GPLNSAT algorithm, that extends the PLNSAT algorithm with new constraints introduced to refine the solving procedure. These constraints are derived automatically from the structure of SystemVerilog covergroups.

4.2.1. Solving Constraints Formulation. In the GPLNSAT algorithm, the bins are divided into groups (put into different clauses) in a natural way: group the bins that come from the same SystemVerilog coverpoint. Recall that the bins coming from the same coverpoint have the same source, namely the expression of the coverpoint (see Section 2.1.2). Thus, in typical cases, if a bin b_0 is determined to be *true*, the other bins of the same coverpoint are also determined at the same time (wildcard bins may not have this characteristic, but we can expand them into non-wildcard bins for GPLNSAT). Some of the bins will be implied *true,* indicating that their specifications have intersections with b_0, and the solved value for b_0 is in the intersections. Other bins will be implied *false,* indicating that their specifications have no intersections with b_0, or the solved value of b_0 is not in the intersections. In either case, once a bin is determined to be *true*, the whole group (a clause) is fully assigned and then the SAT solver can focus on other groups. Furthermore, this approach is fully automatic and does not require additional computation to determine the relationships between bins.

The constraint is translated to a set of *grouped property list* clauses. For example, if there are five bins in which b_1, b_2, and b_3 come from the same coverpoint and b_4 and b_5 come from another coverpoint, the constraint clauses will be $(b_1 \vee b_2 \vee b_3 \vee \neg q_1)(b_4 \vee b_5 \vee \neg q_2)(q_1 \vee q_2)$. The variables q_1 and q_2 in the clauses are auxiliary variables: if the SAT instance is satisfiable and $q_1 = true$, at least one of the bins in the group $\{b_1, b_2, b_3\}$ is solved and, if $q_2 = true$, at least one of the bins in the group $\{b_4, b_5\}$ is solved. By such encoding, the clause $(q_1 \vee q_2)$ guarantees at least one of the bins is solved. Note that, if $q_1 = \mathit{false}$, bins of $\{b_1, b_2, b_3\}$ could be either *true* or *false*. This gives the SAT solver more flexibility in that it can leave a group unsatisfied temporarily, and try to solve another group; on the other hand, if q_1 is decided *true* prior to the assignments of $\{b_1, b_2, b_3\}$ during search, at least one bin will be *true* if there is a satisfiable solution. Just as in the PLNSAT algorithm, if the SAT instance is satisfied, new narrowed grouped property list clauses will be added. For example, if b_1 and b_4 are solved, two clauses will be added, namely $(b_2 \vee b_3 \vee \neg q_1)$ and $(b_5 \vee \neg q_2)$. A difference between PLNSAT and GPLNSAT is that GPLNSAT adds more clauses while PLNSAT adds fewer, but longer, clauses. Empirically, more clauses and longer clauses both decrease solving efficiency, so it is hard to predict which algorithm will have better performance by the structure of clauses only, but GPLNSAT is more likely to get a solution quickly, as we have discussed. We will support this empirically in Section 5.

4.2.2. The Single-Pass GPLNSAT Algorithm. The single-pass GPLNSAT algorithm can be achieved by Algorithm 1 and the GenGPLClause method, which is shown in Algorithm 3. In Algorithm 3, the unsolved bins have been divided into groups according to their belonging coverpoints. In line 4, each group is parsed to the GenPLClause method with a negated auxiliary literal. The grouped property list clauses are augmented by the clauses returned by the GenPLClause method, as well as a clause that contains all the auxiliary literals. Algorithm 1 then merges these clauses and the original SAT instance to generate a stimulus.

ALGORITHM 3: GenGPLClause

```
Input: Unsolved bins, $U$; An auxiliary variable, $a$.
Output: The grouped property list clauses, $GPLClauses$.
// assume $B_i \subseteq U = \{B_0, \ldots, B_{m-1} | m \geq 1\}$
// $B_i$ is a set of bins come from a same coverpoint
1: $GPLClauses = \emptyset$;
2: for each $B_i$ do
3:     $q_i$ = newVar();
4:     $GPLClauses = GPLClauses \cup$ GenPLClause$(\neg q_i, B_i)$;
5: end for
6: $GPLClauses = GPLClauses \cup (\bigvee_{i=0}^{m-1} q_i \vee a)$
7: return $GPLClauses$;
```

4.3. Stimulus Generation by PLNSAT or GPLNSAT

Algorithm 4 presents the PLNSAT/GPLNSAT algorithm. The inputs of the algorithm are a DUV model, a set of input constraints, a set of coverage bins, and the maximum bound. In line 1, the stimuli S, bound k, the unsolved bin list U, and the SAT instance CNF are initialized. The outer while-loop in line 2 terminates when all bins are solved or the maximum bound is reached. Line 3 augments the SAT instance by one frame, while new auxiliary variables are generated for every bound in line 4. The U_{old} represents the unsolved bins before simultaneous SAT solving; it is initialized to empty in line 5. The inner while-loop in line 6 terminates when the number of unsolved bins cannot be further decreased in the current bound. After U_{old} is assigned as$U$ in line 7, line 8 uses the single-pass PLNSAT/GPLNSAT algorithm to generate a stimulus and update currently unsolved bins. The new stimulus is recorded in line 9. In line 11, a unit clause is added to the SAT instance to disable the constraints of the current bound, as discussed in Section 4.1.1. The bound is increased by one in line 12. Finally, the stimuli are returned in line 14.

ALGORITHM 4: (G)PLNSAT

```
Input: DUV, $D$; Constraints, $C$; Bins $B$; Maximum bound $K$.
Output: Stimuli $S$
 1: $S = \emptyset, k = 0, U = B, CNF = \emptyset$;
 2: while $U \neq \emptyset$ and $k \leq K$ do
 3:     $CNF = CNF \cup GenCNF(D, C, U, k)$;
 4:     $a_k$ = newVar();
 5:     $U_{old} = \emptyset$;
 6:     while $U_{old} \neq U$ do // new stimulus can be found
 7:         $U_{old} = U$;
 8:         $(s, U)$ = Algorithm1($U, CNF, a_k$);
 9:         $S = S \cup s$;
10:     end while
11:     $CNF = CNF \cup (a_k)$; // disable previous constraints
12:     $k++$;
13: end while
14: return $S$;
```

5. EXPERIMENTAL RESULTS

We have implemented the proposed algorithms in C++. The adopted SAT solver is MiniSAT [Eén and Sörensson 2003a] because it provides a practical API for adding clauses incrementally and taking literals as assumptions. All the experiments are conducted on a Linux PC using 2.53 GHz Intel Xeon CPU E5649 with 48GB RAM. The experiments are designed to demonstrate the following:

—the efficiency of the simultaneous SAT approaches, in contrast with generating a stimulus individually;
—the difference in solving potential between MSPSAT and our approaches;
—the ability for GPLNSAT to achieve better performance than PLNSAT; and
—the merits of (G)PLNSAT over MRRS.

A UART circuit and large circuits from the ITC99 benchmark are chosen. The circuit and the specification of functional coverage model of UART are taken from Mentor Graphics [2013]. For ITC99 circuits, we predetermine some reachable states and write coverage models for them. The detailed steps are: (1) run SAT solving on interested PIs/POs/FFs for all concerned values; for example, a 3-bit counter with value 0 to 7; (2) from step 1, we can get reachable values in a given runtime or time frames, and these values are chosen as the candidate bins; and (3) we then write cross-coverage on these bins and use methods similar to steps 1 and 2 to select cross-bins for experiments.

5.1. Performance Evaluation of Simultaneous SAT Approaches

Table I shows the information of test circuits. The first column shows the name of the circuits. The second and third columns show the number of coverage bins and the maximum bound of the bins of the circuits. The fourth and fifth columns indicate the size of initial SAT instance, including the number of variables and clauses.

Table II gives the stimulus generation results. The second column shows the stimulus generation time of the single-objective-based method. Each of the bins is solved individually and the learned clauses are reused across bounds (incremental SAT). The third column presents the stimulus generation time of MSPSAT and the fourth column gives the speedup compared with MiniSAT. The same timing information of PLNSAT and GPLNSAT is shown in the columns 5 and 6, and in columns 7 and 8, respectively.

Table I. Circuit Statistics

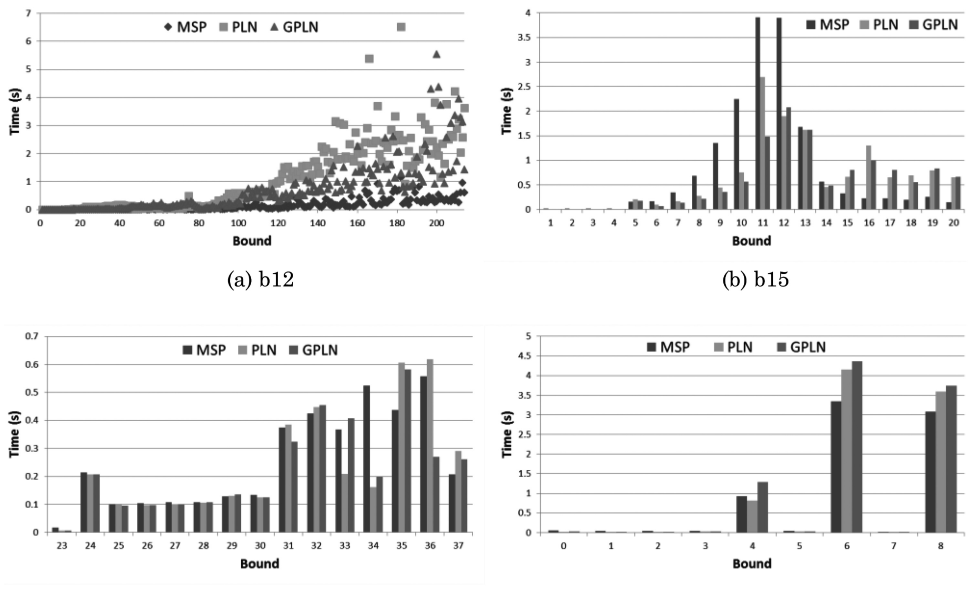
			CNF Size	
Circuit	# Bin	Max Bound	# Variable	# Clause
UART	391	1702	3904	7677
b12	38	216	1134	2782
b14	134	4	8006	24664
b15	421	20	6108	15253
b17	269	30	18679	54354
b18	29	34	58571	180528
b19	182	14	111840	341808
b20	512	8	17462	52255
b21	512	8	17304	51996
b22	640	8	25849	78506

Table II. Stimulus Generation Results

	MiniSAT	MSPSAT		PLNSAT		GPLNSAT	
Circuit	Time (s)	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
UART	15762.86	7094.89	2.22	6663.36	2.37	6355.55	2.48
b12	241.32	36.79	6.56	194.07	1.24	147.94	1.63
b14	4.68	0.24	19.50*	0.25	18.72	0.24	19.50*
b15	243.30	17.03	14.29	13.95	17.44	11.47	21.21
b17	105.13	44.58	2.36	37.95	2.77	36.50	2.88
b18	328.90	184.35	1.78	153.22	2.15	128.81	2.55
b19	62.89	3.94	15.96	3.41	18.44	3.20	19.65
b20	76.69	6.58	11.66	6.40	11.98	6.39	12.00
b21	68.85	4.87	14.14	5.01	13.74	4.83	14.25
b22	110.39	7.33	15.06	8.52	12.96	9.28	11.90

We have implemented the MSPSAT algorithm as the following procedures: (1) a list of unsolved bins is maintained and, for each bound, the SAT solver takes the bins in the list one-by-one as the assumption; (2) if an assumption is satisfied, the status of the undetermined bins will be checked to see whether there are also other bins being solved simultaneously, then all the solved bins will be removed from the list and the remaining bins will be kept to the next bound. Also, during the whole process of stimulus generation, all the learned clauses are passed across bounds. Thus, the main difference between the MSPSAT algorithm and our approaches is that it only considers one bin at a time (and learns knowledge from previously checked bins), while our approaches consider all bins simultaneously.

For all the cases, the three simultaneous SAT approaches all outperform MiniSAT. The GPLNSAT method has better performance in seven out of ten cases. On the other hand, MSPSAT has better performance in two out of ten cases, and the result on b14 equals that of GPLNSAT. To figure out the different situations between the circuits, we measured the solving time of each bound. The results are illustrated in Figure 6. Figure 6(a) shows the solving time of the three simultaneous SAT approaches in each bound of b12. In this benchmark, MSPSAT almost outperforms the other two methods in every bound. This indicates that the overheads introduced by the additional constraint clauses might degrade the performance of SAT solving for small-size circuits. Figure 6(b) presents the solving situation of b15. The solving time of MSPSAT during bounds 6 to 12 increases drastically and then suddenly drops. On the other hand, PLNSAT and GPLNSAT have the same solving potential and are more stable



(a) b12 (b) b15

(c) b19 (d) b22

Fig. 6. The solving time of individual bounds by MSPSAT, PLNSAT, and GPLNSAT.

and faster than MSPSAT. Another case where GPLNSAT has better performance is b19; the result is illustrated in Figure 6(c). In this case, the solving times of the three approaches are approximately the same in bounds 0 to 8, but are diverse in the remaining bounds. Figure 6(d) demonstrates the result of b22. From the graph, we can see that bounds 4, 6, and 8 occupy most of the solving time because most of the target bins are solved in these bounds. The GPLNSAT method has the worst performance in this case, and this is the only case where PLNSAT outperforms GPLNSAT. Actually, the 640 target bins come from only five coverpoints, each having 128 bins. The result of b22 may indicate that PLNSAT and GPLNSAT are not good at dealing with intensive targets from the same source with the same bound.

5.2. Performance Evaluation of Different Group Sizes

To show the effectiveness of the GPLNSAT approach, we randomly divided the bins into groups of different sizes. The results are shown in Table III. The first column indicates the name of each circuit. The rest of the columns present the solving time of different group sizes. Size 1 means PLNSAT, and the letter “G” means GPLNSAT; sizes 2 to 10 mean the number of literals in each constraint clause. In each case the solving time of different group sizes is normalized by dividing with the solving time of PLNSAT. Hence, a solving time lower than 1 means better performance than PLNSAT. Moreover, the lowest (best) time in each row is underlined. The total scores of each group size are also presented in the last row. If a group size has the lowest time in a circuit case, it gets 10 scores; otherwise, its score shrinks according to the ranking, and the lowest score is zero. The results show that grouping bins randomly does not guarantee better performance than PLNSAT. Besides, the solving times of random grouping methods are average values, that is, the actual time depends on the grouping conditions and may differ drastically. Of course, one can group the bins according to their affinities, but this requires additional calculating effort. In contrast, GPLNSAT has a more stable solving time and does not require any preprocessing effort. Furthermore, GPLNSAT

Table III. Solving Time of Different Group Sizes

	Group Size										
Circuit	1	2	3	4	5	6	7	8	9	10	G
UART	1	0.92	0.98	1.05	1.01	1.04	1.44	1.03	1.16	0.96	0.95
b22	1	1.00	1.03	0.96	0.94	0.99	0.99	0.97	0.95	0.97	1.09
b21	1	0.89	0.93	0.91	0.91	0.97	0.94	0.93	0.94	0.94	0.96
b20	1	1.03	1.01	1.01	1.03	1.01	1.00	1.02	0.97	0.96	0.99
b19	1	0.96	0.94	0.93	0.93	1.04	0.89	0.91	0.96	0.96	0.94
b18	1	0.96	1.01	0.93	0.96	0.91	0.92	0.91	1.61	1.62	0.89
b17	1	1.11	0.98	1.06	1.01	0.93	1.01	1.04	0.94	0.98	0.96
b15	1	0.99	1.24	1.11	1.13	0.96	0.94	1.07	1.12	1.24	0.82
b14	1	0.98	0.99	0.98	0.98	0.97	0.98	0.98	0.98	0.97	0.96
b12	1	0.94	0.79	0.80	0.70	0.87	0.81	0.81	0.78	0.81	0.76
Score	29	46	42	54	54	48	52	51	54	50	70

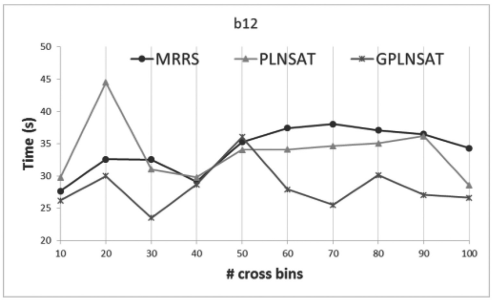
outperforms PLNSAT in 9 out of 10 cases and also gets the highest total score among all group sizes.

5.3. Comparison with the MRRS Algorithm

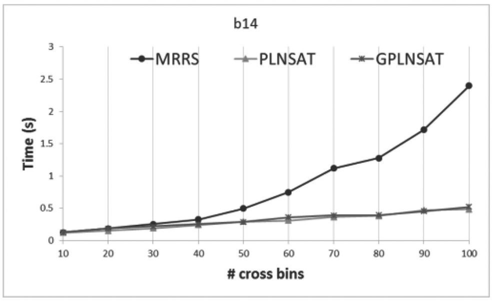
To compare with the MRRS algorithm, we design the following experimental settings: in each case, two coverpoints for cross-coverage are selected and the size of cross-coverage is gradually increased by 10 to test the scalabilities of different approaches. The bins not belonging to the two coverpoints are filtered so their reachable bounds are not too far from those of the cross-coverage bins. This can make the reported timing more related to the cross-coverage bins. Furthermore, the MRRS algorithm proposed in Cheng et al. [2012] was not incremental, so we also add auxiliary variables into the OR constraint of monitors to make it incremental for fair comparison.

Figure 7(a) to Figure 7(f) demonstrate results of cases b12, b14, b15, b17, b19, and b22, respectively. In Figure 7, the y-axes indicate the solving time, while the x-axes represent the size of cross-coverage, ranging from 10 to 100. In general, GPLNSAT outperforms PLNSAT, as we can expect from the experimental results of Table I. On the other hand, PLNSAT is more efficient than MRRS in most cases. The only exception is b17. In this case, none of the three approaches appears definitely better than the others, but comparing the summing time of the 10 conditions, the MRRS algorithm still takes the longest time.

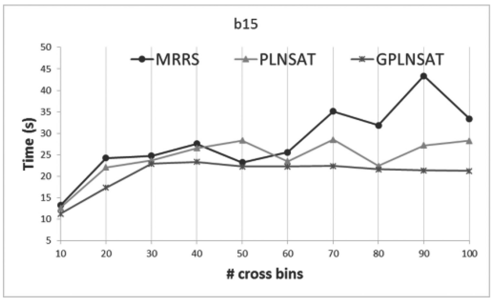
Figure 8 shows the final number of clauses (the model is unrolled into the maximum bound, together with all constraint clauses) in each condition of the three approaches in each case. In each circuit case, the number of clauses is normalized by dividing with the number of clauses of MRRS with cross-size 10. The y-axis indicates the accumulated clause number of different cross-sizes, and the x-axis presents the approaches used in each case. The results show that the clause number of MRRS increases rapidly whenever the size of cross is raised. This result is predictable because: (1) as we have mentioned, the specifications of the cross-bins have few in common, so the MRRS approach cannot gain benefit; and (2) each time a cross-bin is solved, the MRRS approach will reconstruct a new set of monitors for all cross-bins, which are then converted into additional clauses. In the cases of small circuits, the number of monitor clauses produced by MRRS could be even larger than the number of model clauses, like b14. On the other hand, PLNSAT and GPLNSAT do not reconstruct bins, so the number of clauses increases linearly and slowly. In the case of b12, PLNSAT and GPLNSAT with larger cross-sizes definitely do not have a larger number of clauses. For example, GPLNSAT with cross-size 70 has a smaller number of clauses than that of the case



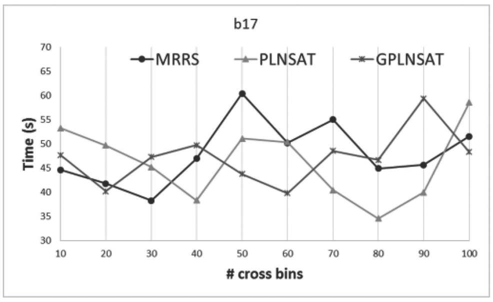
(a) b12



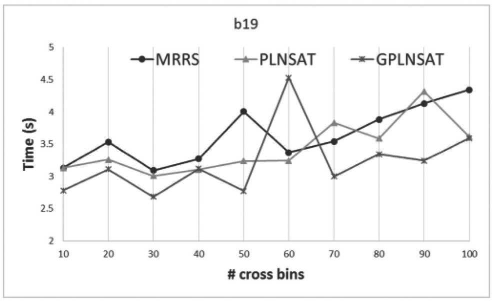
(b) b14



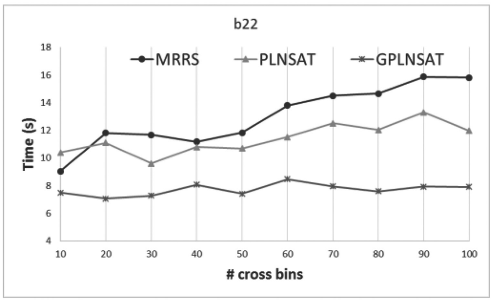
(c) b15



(d) b17



(e) b19



(f) b22

Fig. 7. Solving time versus cross-size of the three different approaches.

with cross-size 60. This is possible since the circuit size is small; the learned conflict clauses may occupy a greater percentage than property list clauses.

6. RELATED WORKS ON INCREMENTAL SAT AND SIMULTANEOUS SAT

This section reviews related work on incremental SAT and simultaneous SAT. Generally speaking, incremental SAT exploits the reusability of conflict clauses, while simultaneous SAT tries to solve multiple objectives in one model. Incremental SAT is naturally incorporated with simultaneous SAT because conflict clauses are shared by all objectives.

6.1. Incremental SAT

As described in Section 2.2, SAT solvers use conflict learning to facilitate the searching process. The basic idea of incremental SAT solving is to reuse the learned clauses during solving closely related SAT instances [Whittemore et al. 2001; Strichman 2004; Jin

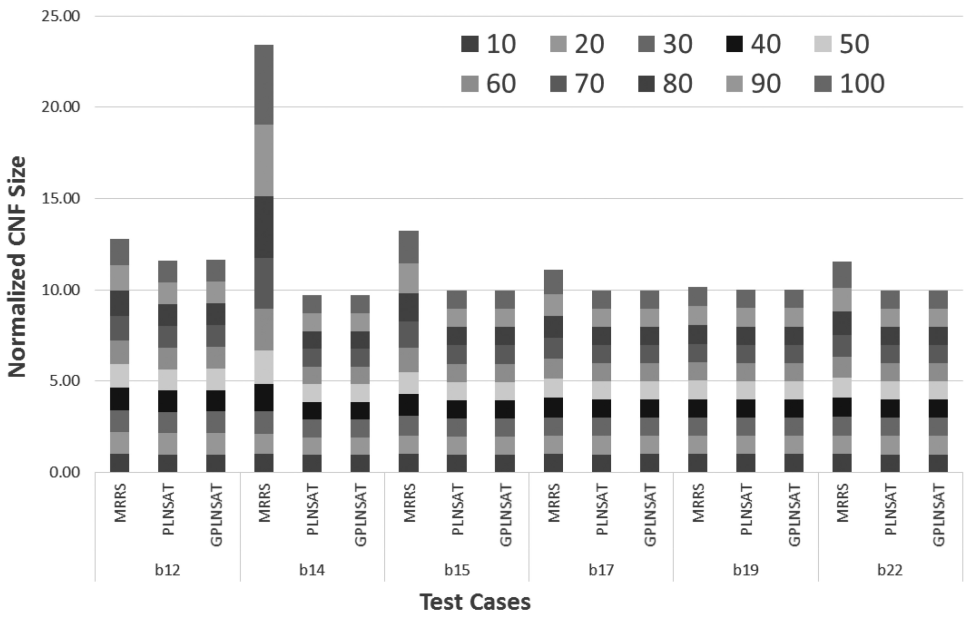


Fig. 8. CNF size comparison of different cross-size. The numbers 10 to 100 are the number of cross-bins.

and Somenzi 2004; Novikov and Goldberg 2001; Eén and Sörensson 2003b; Hooker 1993]. However, due to the differences between each SAT instance, not all of the learned clauses can (normally) be reused. Several approaches were proposed in this context.

Whittemore et al. [2001] recorded each conflict clause and removed those learned from clauses no longer existing in a newer SAT instance. However, tracking the dependencies between the learned and the removed clauses can be costly. Strichman [2004] was the first to observe that, in the case of BMC, those clauses learned solely from the structure of the model are valid to all the remaining instances. These clauses learned from the common part of all the instances are called *pervasive conflict clauses* [Silva and Sakallah 1997]. By marking the common clauses, pervasive conflict clauses can be easily identified: if all clauses leading to the conflict are marked, the derived conflict clause is pervasive. This approach avoids the overheads of tracking the whole relationships between clauses. Nevertheless, many profitable learned clauses are discarded. Jin and Somenzi [2004] extend the approach of Strichman [2004] by reusing *objective-independent* clauses. A *primary objective* is a proof objective literal (PO for short) that tends to be *true* or *false* in a satisfied solution. If one of the ancestors of the conflict node in the implication graph is a PO, the conflict clause is said to be *objective dependent*, thus not forwarded. In contrast, objective-independent clauses are kept. However, the authors proposed another technique to *distill* the objective-dependent clauses to be forwarded. These two techniques drastically increase the number of forwarded clauses compared with Strichman [2004], and also improve solving performance.

Novikov and Goldberg [2001] proposed a method that allows all conflict clauses to be reused when checking whether multiple *cubes* (the conjunction of a set of literals) are implications of a formula in Disjunctive Normal Form (DNF). The literals of a cube are deemed as internal assumptions to the formula. Because the conflict clauses are guaranteed independent of the assumptions (if the algorithm backtracks to one of the initial assignments, the solving stops), all of them can be forwarded. Eén and Sörensson

[2003b] mentioned that if only unit clauses are removed, all the learned clauses can be reused. In their implementation, the POs of BMC are treated as assumptions during individual solving. This idea is basically the same as Novikov and Goldberg [2001] in the special case that cubes only contain one literal.

If an SAT instance is augmented by some clauses to form a new one, all the conflict clauses can be reused because the new instance implies the old one [Hooker 1993]. But this kind of incremental method is too limited. The method of Eén and Sörensson [2003b] treats unit clauses as assumptions, thus enabling the removal of unit clauses. Moreover, this method can be extended to add-and-remove clauses with any length, just as what we do to disable the constraint clauses.

6.2. Double Incremental SAT

Basically, the aim of Whittemore et al. [2001], Strichman [2004], Jin and Somenzi [2004], Eén and Sörensson [2003b], and Hooker [1993] is to effectively or maximally reuse the conflict clauses of a single PO through bounds. On the other hand, the goal of Novikov and Goldberg [2001] is to share learned clauses through multiple cubes. The SSAT algorithm [Khasidashvili et al. 2005] is a *double incremental* approach in that the learned clauses are not only reused across bounds but also across properties at each bound. Another important feature of SSAT is that it tries to resolve multiple POs related to the same SAT instance by one model. This could possibly further reduce the whole solving time. In this section, we will introduce double incremental approaches that consider learning across bounds as well as across properties. And we will leave those approaches that can resolve multiple POs by one model to the next section.

In directed test generation, Chen and Mishra [2010] proposed a method to share conflict clauses between a group of properties with given bounds. First, properties are divided into several groups according to their similarity and then, for each group, a *base property* is solved first and afterwards the conflict clauses learned from the base property are *filtered* by the other properties in the same group to get individual reusable clauses. However, determining base properties can be a bottleneck. Chen and Mishra [2011] have proposed another learning technique: again, properties are divided into groups, but the learning across properties is derived from the decision orders of the variables. Since conflict clauses are not reused across properties, there is no need to compute the intersections between properties to determine the base property. This technique shows 2× speedup compared to Chen and Mishra [2010]. The idea of Qin et al. [2010] is to *synchronize* the solving process of multiple properties for different bounds. This method is an intuitive extension of Strichman [2004]. In each bound, properties are solved individually and the pervasive clauses are forwarded across properties. If a property is proved, it will be removed, otherwise it will be kept to next bound. Also, the pervasive clauses are forwarded to the next bound as in Strichman [2004]. The experimental results demonstrated that this approach is superior to the combination of Strichman [2004] and Chen and Mishra [2010]. This approach is further applied to multicore architectures in Qin and Mishra [2012].

6.3. Simultaneous SAT

As mentioned in the previous section, the method of Khasidashvili et al. [2005] is both double incremental and simultaneously solves multiple properties. It is worth noting that these two techniques are *orthogonal*, which means that they can be adopted separately or combined. The benefit of solving multiple properties simultaneously is the opportunity to share subsolutions and all the learned clauses, so there is no need to determine pervasive clauses. Besides, in the view of verification, the properties stand for behaviors of the design model, so it is preferable to consider them together.

Fraer et al. [2002] formed a conjunction of properties and checked them simultaneously in BMC. If the conjunction can be proved, all the properties are resolved, otherwise, the provable subset is found after several solving iterations. In each iteration, some of the properties that cannot be proved are eliminated. The implementation of this approach in Fraer et al. [2002] was not incremental, but can be easily made so by the methods mentioned in Section 6.1. In SSAT [Khasidashvili et al. 2005], a list of POs is maintained and each time a PO is selected (called the currently watched PO, or CWPO) to be resolved. The CWPO is treated as an assumption in that if it can be proved, the other properties are checked and possibly some of them are also resolved (also proved or determined as global conflicts), otherwise the CWPO cannot be proved in the current bound, so it will be kept to the next bound. The solved properties are removed from the list and then a next CWPO is picked and a new search starts. Franzén et al. [2010] also proposed an algorithm called MSPSAT for simultaneously solving multiple targets. Actually, their goal is to solve multiple formulae together, so they create predicate literals p_i for each formula φ_i, and add the formulae $p_i \equiv \varphi_i$ to the solver. The basic idea of MSPSAT is the same as SSAT, but the authors use the assumption API of the MathSAT [2013] solver for implementation, so it does not require any modification to the solver. Cabodi and Nocco [2011] also grouped properties according to their mutual affinities as in Chen and Mishra [2010, 2011], then they used SSAT to solve each group of properties. The solving orders of groups are predetermined by their expected verification effort. The reason that they group the properties is because properties are often associated with parts of local behaviors of the design, so the effort to verify a subset of properties is much lower than that of verifying all properties together. However, they did not compare this approach with pure SSAT in the experimental results. Khasidashvili and Nadel [2011] modified SSAT [Khasidashvili et al. 2005] into an *implicative* version. In some applications, the POs are of the form PO $\equiv o_s \leftrightarrow o_i$, hence translating many of them into CNF can be a bottleneck for the SAT solver. The new SSAT, called implicative SSAT, takes pairs of (o_s, o_i) as inputs, and can check their equivalence without translating them into CNF. This approach shows great improvements in solving time of in-depth BMC and invariant strengthening.

Our stimulus generation methods use double incremental approaches so every conflict clause can be reused and they also simultaneously solve all the objectives. The experimental results demonstrate the advantage of our simultaneous SAT approaches.

7. CONCLUSIONS

Coverage-driven verification is effective for functional verification because the coverage targets for stimulus generation are dynamically adjusted. In this article, we utilize the functional coverage model of SystemVerilog so that it may be integrated into modern digital design flows. The language constructs of the functional coverage model are studied and converted into CNF for SAT-based stimulus generation. Since there could be a lot of coverage holes, we want to get a set of stimuli to trigger them efficiently. In this context, the PLNSAT and GPLNSAT algorithms are proposed to simultaneously solve all the targets. The experimental results on a UART circuit and the largest ITC99 benchmark circuits show that both of the approaches can increase performance drastically, with speedup 1.6x to 21.2x, compared to a single-target generation method. They also are more efficient than the MRRS algorithm that we have proposed in Cheng et al. [2012]. Furthermore, GPLNSAT can achieve better performance than the prior best simultaneous SAT method, MSPSAT, in seven out of ten cases.

REFERENCES

Mike Benjamin, Daniel Geist, Alan Hartman, Yaron Wolfsthal, Gerard Mas, and Ralph Smeets. 1999. A study in coveragedriven test generation. In *Proceedings of the Design Automation Conference (DAC'99)*. 970–975.

Marc Boulé and Zeljko Zilic. 2008. *Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Springer.

Gianpiero Cabodi and Sergio Nocco. 2011. Optimized model checking of multiple properties. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'11)*. 1–4.

Mingsong Chen and Prabhat Mishra. 2010. Functional test generation using efficient property clustering and learning techniques. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 29, 3, 396–404.

Mingsong Chen and Prabhat Mishra. 2011. Property learning techniques for efficient generation of directed tests. *IEEE Trans. Comput.* 60, 6, 852–864.

An-Che Cheng, Chia-Chih Yen, and Jing-Yang Jou. 2012. A formal method to improve system verilog functional coverage. In *Proceedings of the International High Level Design Validation, and Test Workshop (HLDVT'12)*. 56–63.

Sayantan Das, Rizi Mohanty, Pallab Dasgupta, and Partha P. Chakrabarti. 2006. Synthesis of system verilog assertions. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'06)*. 1–6.

Niklas Eén, Alan Mishchenko, and Nina Amla. 2010. A single-instance incremental sat formulation of proof- and counterexample-based abstraction. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'10)*. 181–188.

Niklas Eén and Niklas Sörensson. 2003a. An extensible sat-solver. In *Proceedings of the 6th International Conference* on *Theory and Applications of Satisfiability Testing (SAT'03)*. 502–518.

Niklas Eén and Niklas Sörensson. 2003b. Temporal induction by incremental sat solving. *Electron. Not. Theor. Comput. Sci.* 89, 4, 543–560.

Ranan Fraer, Shahid Ikram, Gila Kamhi, Tim Leonard, and Abdel Mokkedem. 2002. Accelerated verification of rtl assertions based on satisfiability solvers. In *Proceedings of the International High Level Design Validation and Test Workshop (HLDVT'02)*. 107–110.

Anders Franzén, Alessandro Cimatti, Alexander Nadel, Roberto Sebastiani, and Jonathan Shalev. 2010. Applying smt in symbolic execution of microcode. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'10)*. 121–128.

John N. Hooker. 1993. Solving the incremental satisfiability problem. *J. Logic Program* 15, 1–2, 177–186.

IEEE Std. 2013. IEEE standard for systemverilog–Unified hardware design, specification, and verification language. IEEE Std 1800-2012 (revision of ieee std 1800-2009). 1–1315. http://dx.doi.org/10.1109/IEEESTD.2013.6469140.

ITC'99 Benchmark. 1999. ITC'99 benchmark homepage. http://www.cerc.utexas.edu/itc99-benchmarks/bench.html.

Hoonsang Jin and Fabio Somenzi. 2004. An incremental algorithm to check satisfiability for bounded model checking. *Electr. Not. Theor. Comput. Sci.* 119, 2, 51–65.

Ivan Kastelan and Zoran Krajacevic. 2009. Synthesizable system verilog assertions as a methodology for soc. In *Proceedings of the 1st IEEE Eastern European Conference on the Engineering of Computer Based Systems (ECBS-EERC'09)*. 120–127.

Zurab Khasidashvili and Alexander Nadel. 2011. Implicative simultaneous satisfiability and applications. In *Proceedings of the 7th International Haifa Verification Conference (HVC'11)*. 66–79.

Zurab Khasidashvili, Alexander Nadel, Amit Palti, and Ziyad Hanna. 2005. Simultaneous sat-based model checking of safety properties. In *Proceedings of the 1st International Haifa Verification Conference (HVC'05)*. 56–75.

Jiang Long and Andrew Seawright. 2007. Synthesizing sva local variables for formal verification. In *Proceedings of the Design Automation Conference (DAC'07)*. 75–80.

Biruk Mammo, Debapriya Chatterjee, Dmitry Pidan, Amir Nahir, Avi Ziv, Ronny Morad, and Valeria Bertacco. 2012. Approximating checkers for simulation acceleration. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'12)*. 153–158.

Joao P. Marques-Silva and Karem A. Sakallah. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.* 48, 5, 506–521.

Joao P. Marques-Silva and Karem A. Sakallah. 1997. Robust search algorithms for test pattem generation. In *Proceedings of the 27thAnnual International Symposium on Fault-Tolerant Computing (FTCS'97)*. 152–161.

Davis Martin, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Comm. ACM* 5, 7, 394–397.

Davis Martin and Hilary Putnam. 1960. A computing procedure for quantification theory. *J. ACM* 7, 3, 201–215.

MathSAT. 2013. The mathsat 5 smt solver. http://mathsat.fbk.eu/.

Mentor Graphics. 2013. Coverage cookbook. https://verificationacademy.com/cookbook/coverage.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an efficient sat solver. In *Proceedings of the Design Automation Conference (DAC'01).* 530–535.

Yakov Novikov and Evgueni Goldberg. 2001. An efficient learning procedure for multiple implication checks. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'01).* 127–133.

Xiaoke Qin, Mingsong Chen, and Prabhat Mishra. 2010. Synchronized generation of directed tests using satisfiability solving. In *Proceedings of the 23rd International Conference on VLSI Design (VLSID'10)*. 351–356.

Xiaoke Qin and Prabhat Mishra. 2012. Directed test generation for validation of multicore architectures. *ACM Trans. Des. Autom. Electron. Syst.* 17, 3, 1–21.

Ofer Strichman. 2004. Accelerating bounded model checking of safety properties. *J. Formal Methods Syst. Des.* 24, 1, 5–24.

Grigorii S. Tseitin. 1970. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part II,* Consultants Bureau, 115–125.

Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. 2001. SATIRE: A new incremental satisfiability engine. In *Proceedings of the Design Automation Conference (DAC'01).* 542–545.

Robert Wille, Görschwin Fey, Marc Messing, Gerhard Angst, Lothar Linhard, and Rolf Drechsler. 2008. Identifying a subset of system verilog assertions for efficient bounded model checking. In *Proceedings of the 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools, (DSD'08).* 542–549.

Bo-Han Wu, Chun-Ju Yang, Chia-Cheng Tso, and Chung-Yang (RIC) Huang. 2011. Toward an extremely-high-throughput and even-distribution pattern generator for the constrained random simulation techniques. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'11)* 602–607.

Shuo Yang, Robert Wille, Daniel Große, and Rolf Drechsler. 2012. Coverage-driven stimuli generation. In *Proceedings of the 15th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools (DSD'12).* 525–528.

Shuo Yang, Robert Wille, Daniel Große, and Rolf Drechsler. 2013. Minimal stimuli generation in simulation-based verification. In *Proceedings of the 16th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools (DSD'13).* 439–444.

Hu-Hsi Yeh and Chung-Yang Huang. 2010. Automatic constraint generation for guided random simulation. In *Proceedings of the 15th Asia and South Pacific Design Automation Conference (ASPDAC'10)*. 613–618.

Jun Yuan, Carl Pixley, and Adnan Aziz. 2006. *Constraint-Based Verification*. Springer.

Ramin Zabih and David Mcallester. 1988. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*. 155–160.

Received January 2014; revised May 2014; accepted July 2014