SimGen: Simulation Pattern Generation for Efficient Equivalence Checking

Carmine Rizzi
D-ITET, ETH Zurich
Zurich, Switzerland
crizzi@ethz.ch

Sarah Brunner
D-ITET, ETH Zurich
Zurich, Switzerland
sarbrunn@student.ethz.ch

Alan Mishchenko
Department of EECS, UC Berkeley
Berkeley, USA
alanmi@berkeley.edu

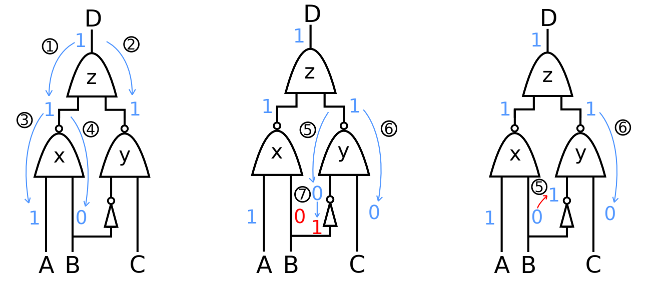
Lana Josipović
D-ITET, ETH Zurich
Zurich, Switzerland
ljosipovic@ethz.ch

***Abstract*—Combinational equivalence checking for hardware design tends to be slow due to the number and complexity of intermediate node equivalences considered by the SAT solver. This is because the solver often spends extensive time disproving nodes that appear equivalent under random simulation. We propose SimGen, an open-source and expressive simulation pattern generator inspired by Automatic Test Pattern Generation (ATPG); it exploits the circuit's structure to disprove the equivalence of circuit nodes and avoid excessive SAT calls. We demonstrate the effectiveness of SimGen's simulation patterns over those generated by state-of-the-art random and guided simulation.**

I. Introduction

Combinational equivalence checking (CEC) determines whether two circuit networks have an equivalent logic function. It is commonly performed using *boolean satisfiability* (SAT) or *binary decision diagram* (BDD) sweeping, where a SAT or BDD solver proves or disproves the equivalence of a pair of candidate points [14]. Yet, checking all point pairs is time-consuming [3]; thus, sweeping is usually performed after iterative circuit simulation that partitions the *equivalence classes* (i.e., sets of circuit points that may be equivalent and must be checked) of the considered networks and, consequently, reduces the number of checks.

The key to successful simulation lies in employing simulation vectors that efficiently partition the equivalence classes. Fully random simulation patterns [13], [19] cannot guarantee the separation of specific classes and often remain stuck at a local minimum. *Reverse simulation* [26] promises to efficiently separate equivalence classes by generating targeted simulation vectors as follows: (1) Randomly select a pair of nodes from the same class, also called *target* nodes. (2) Assign complementary values to the target node outputs. (3) For each target node, determine a set of inputs for which the node's logic function produces the desired value; assign these values to the target node inputs (i.e., the outputs of the predecessor nodes). If multiple assignments are possible, pick one randomly. (4) Traverse the networks backward while assigning a value to each node following the same strategy (i.e., to honor the logic function of the previously visited nodes). (5) Terminate if the inputs of the networks are reached or if a conflicting assignment occurs at any internal node. If reverse simulation successfully reaches the network's inputs, the values assigned to the inputs serve as a high-quality simulation vector that can disprove the target nodes' equivalence; otherwise, the procedure repeats with the next pair of target nodes.



(a) Reverse simulation: steps 1-4 (b) Reverse simulation: steps 5-7 (c) SimGen (our work): steps 5-7

Fig. 1: Reverse simulation vs. SimGen. Reverse simulation in Figures a and b results in a *conflict* in step 7 (red) due to a local and random decision of the input values of gate y. In contrast, SimGen *implies* the input values of y based on the already assigned value of '0' at input B, thus identifying a correct input vector without a conflict. SimGen aims to exploit this and other strategies to mitigate conflicts and improve the success rate of reverse simulation.

The Limitations of Reverse Simulation. Consider the example in Figure 1. Assume that node z belongs to an equivalence class (other class nodes and their networks are omitted for simplicity); we want to separate it from its class by following the reverse simulation procedure outlined above. To this end, assume that D needs to evaluate to '1'; reverse simulation propagates values backward from D to find a suitable input assignment. In Figure 1a, a series of propagations (represented by enumerated arrows) assigns '1' and '0' to inputs A and B; then, the output of y is assigned a '1'.

In the following propagation steps (Figure 1b), three possible input assignments to gate y result in a '1' at its output: either one input is '0' and the other is '1', or both inputs are '0'. Reverse simulation chooses one input assignment at random—in this example, it assigns '0' to both inputs. Yet, further propagation causes a *conflict* at step 7 (shown in red): the inverter function dictates that B must be '1', but it was previously assigned a '0'. At this point, reverse simulation fails and terminates without identifying an appropriate input vector. This illustrates the limitations of reverse simulation: due to the inability to exploit information regarding previous assignments or the circuit's structure, in many situations, it terminates unsuccessfully; the number of subsequent SAT calls remains excessive and impractical for sweeping realistic circuits.

SimGen: A Guided and Controlled Reverse Simulation Strategy. Figure 1c shows an alternative strategy. It recognizes that the ‘0’ value of input B *implies* that the inverter’s output must be ‘1’ and immediately makes this assignment (the red arrow at step 5). There is now only one possible assignment for the right input of gate y: input C must be set to ‘0’. At this point, all internal and input values are assigned without a conflict, achieving a valid simulation vector that guarantees the desired value at output D. This highlights the importance of leveraging functional and structural information to prevent conflicts and achieve the desired simulation vector.

In this paper, we propose SimGen, an open-source simulation vector generator for effective and controlled equivalence class partitioning. In contrast to standard reverse simulation, we incorporate techniques from Automatic Test Pattern Generation (ATPG) to decrease the chances of failure. We exploit structural and logic information of the circuit to postpone random decisions as much as possible, like in the example above: we establish signal correlations between a node and its neighboring nodes, postpone critical decisions and, when they are inevitable, leverage structural information to make educated decisions. We identify different ways of interleaving ATPG techniques to minimize the number of required SAT calls and, consequently, SAT runtime. On a set of standard logic synthesis benchmarks, we show that our strategy is superior to recent reverse and random simulation approaches.

In the rest of this paper, Section II describes the background and previous works. Section III outlines SimGen’s structure. Section IV and Section V explain two key ATPG concepts incorporated in SimGen. Section VI details our results.

II. Background and Related Work

Boolean Network. A Boolean network is a Directed Acyclic Graph (DAG). Each graph’s node represents a logic function; we assume that each node has a single output bit.

The fanins of a node *n* are the input nodes of *n*. The fanouts of a node *n* are the output nodes of *n*. A Primary Input (PI) and Primary Output (PO) are, respectively, the nodes with no fanins and with no fanouts. The level of a node corresponds to the length of the longest path from any PI. The fanin/fanout cones of a node *n* are the sets of nodes that can reach one of the fanins/fanouts. A Fanout-Free Cone (FFC) is a subset of the fanin cone of a node *n* where all the paths from each node inside the cone towards the POs of the network have to pass through *n*. The leaves of a cone are the first nodes of the cone encountered on any path from any PI to any node of the cone. A Maximum Fanout-Free Cone (MFFC) is the largest among the possible FFCs of a certain node [20].

Sweeping and Simulation. CEC relies on BDD [15], [14] or SAT [16], [21], [5] *sweeping* in conjunction with circuit simulation [21], [12], [17], [22], [24], as illustrated by the blue box in Figure 2. First, both input networks are simulated using random input vectors to separate nodes into independent equivalence classes. The classes are then sent to the verification tool to prove the non-equivalence of its nodes; the resulting input vector can be employed by the simulator for further class separation. Due to the ineffectiveness of random simulation vectors, Mischenko et al. [21] introduce a *1-distance* vector which selectively flips one bit of the input vector obtained from

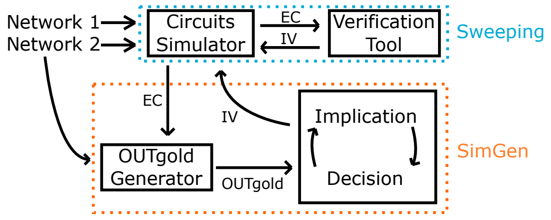


Fig. 2: A sweeping tool with SimGen. SimGen generates OUTgold values that aim to split the equivalence classes (EC) for one of the two networks. Then, it generates the input vector (IV) that propagates these values toward the inputs via decisions and implications.

a counter-example. Yet, the flipping effectiveness is difficult to control and predict. Others [17], [3] employ a SAT solver to generate “expressive” and “high-toggle rate” vectors, but the newly proposed input vector still depends on SAT calls. Reverse simulation [26] is more effective, but still prone to conflicts, as shown in Section I. To the best of our knowledge, industrial tools like Conformal Cadence [7] rely on analogous strategies for CEC and, thus, face the same limitations; the strategies we propose would also benefit them.

Automatic Test Pattern Generation (ATPG). ATPG generates input vectors to expose circuit faults [16], [11]. It identifies the PI values that activate the desired fault and propagates them to the network’s outputs to make them externally visible.

Definition 1: A *propagation* is the assignment of input/output node values such that all previously assigned values remain unchanged and the node’s function is respected.

In this work, we assume that propagation assigns values 0 and 1; a don’t-care is treated as an unassigned value.

ATPG algorithms [25], [10], [9] rely on the circuit’s structure for efficient test generation. If a conflict occurs during the signal propagation, they *backtrack* to their last decision and change their choice; yet, backtracking is time-consuming. Thus, to avoid conflicts, ATPG employs forward implication [10] to assign values to internal nodes and a concept similar to MFFCs [9] to identify independent circuit portions and postpone the propagation of values. To reduce execution time, SimGen omits backtracking and exploits different types of implication and MFFCs, as we will later discuss.

We inherit implication and decision concepts from ATPG.

Definition 2: An *implication* is a propagation that occurs when there is only one available input/output assignment that respects all previously assigned values; it sets all input/output values to respect the values of this single assignment.

In Figure 1, the propagation of D to the inputs of gate z is an implication, as the only possible assignment is (‘1’, ‘1’).

Reverse simulation applies a subset of implication, sometimes referred to as *backward implication*: it sets all inputs according to the output value when only one input assignment is possible, as shown in Figure 1. Some implementations [6] extend this strategy to check the subsequent (i.e., lower) level for conflicts before making an assignment. We will consider the more general definition above and imply values both backward (from output to inputs) and forward (from inputs to output), independently from the levels of a node.

Definition 3: A *decision* is a propagation that occurs when multiple available assignments respect all previously assigned values; it chooses one assignment according to a decision policy. Then, it sets the values of the selected assignment.

In Figure 1, the propagation of the output of gate x is a decision since there are three possible input assignments.

ATPG has been explored for sweeping in CEC, focusing on addressing false negatives during ATPG proofs [16]. Similarly, circuit-based SAT solvers include circuit information during SAT proofs to order decision variables and for non-chronological backtracking [19], [27]. SimGen is complementary to these *sweeping* improvements and could be used alongside them, as Figure 2 suggests: we achieve better input vectors by avoiding conflicts using ATPG techniques.

III. SimGen Flow Overview

Figure 2 shows SimGen added to a sweeping tool; it inputs the network's equivalence classes and performs the following:

(1) *OUTgold value generation.* OUTgold values are the desired output values for target nodes belonging to the same equivalence classes. SimGen will compute an input vector that maximizes the number of target nodes whose value is equal to its desired OUTgold value. We elaborate on our simple OUTgold value assignment in Section VI; more complex strategies could be explored for OUTgold selection (e.g., circuit topology-aware methods or runtime-adaptive OUTgold generation) and effortlessly integrated into SimGen.

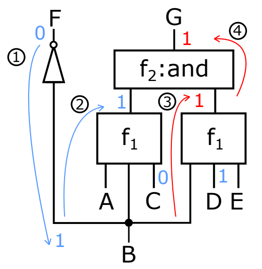
(2) *Input vector generation.* SimGen iterates through the target nodes of an equivalence class in decreasing network depth order. For each target node, it performs the following steps: (a) Assign the desired OUTgold value to the target node. (b) Propagate the value using implications. If a conflict occurs at any implication, terminate the process and restart with a new target node. Otherwise, continue implying until no further implication is possible. (c) Once all implication opportunities are exhausted, make a decision to enable further implications. (d) Repeat steps (b) and (c) until all PIs in the target node's fanin cone are set or a conflict occurs.

After completing the process for all target nodes of an equivalence class, we evaluate the resulting input vector (i.e., obtained PI values). If the vector respects at least a pair of target nodes of the desired OUTgold values, we use it to simulate the circuit and partition the equivalence class. Otherwise, we skip the simulation and repeat the process with another equivalence class. This procedure is conceptually similar to classic reverse simulation and we extend it to multi-bit OUTgold values accordingly [26]; the main novelties of SimGen are the implication and decision strategies used in steps (b) and (c) above. We describe these strategies and their effectiveness in the remainder of the paper.

IV. How Much to Imply?

In this section, we investigate the effectiveness of standard implication and propose a more powerful alternative.

To apply implication from Definition 2, we consider the truth table of the node's function and its already assigned input/output values. We iterate through the truth table rows and identify those that match the set values. If only one row is a match, we assign the row's values to the previously unassigned inputs/output. This may create new implication opportunities.



$I_1I_2I_3$	f_1
- 1 0	1
- 0 0	0
1 - 1	1
0 - 1	0

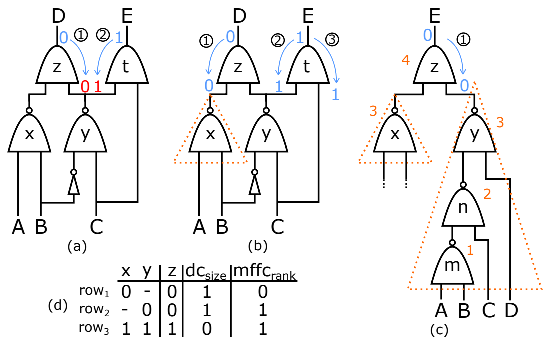
Fig. 3: Implication example. The truth table on the right describes f_1; f_2 is an *and* gate. F, C, and D have initial values '0', '0', and '1'. Standard implication can only assign '1' to the left node f_1's output. *Advanced implication* uses the truth table to determine that f_1's output can be only '1'. This enables new implications (red).

An implication example is shown in Figure 3. On the left, there is a portion of a circuit with different propagation steps. On the right, the table shows the truth table of node f_1. The symbol '-' represents a don't-care. We assume that node f_2 implements a logical *and* function. We also assume an initial value assignment in the graph: the output F has value '0' and the inputs C and D have values '0' and '1'. We start the propagation from output F; given its value, the only possible inverter input value is '1'; hence, we imply this value for input B at step 1. This assignment generates a sequence of implication opportunities. If we consider the left f_1 node, its inputs fully respect only the first row of the truth table of f_1 since B (I_2) and C (I_3) have values '1' and '0' respectively; thus, we imply the value of node's output as 1 accordingly at step 2. As the row has a don't-care for A, in line with our propagation definition, we keep A's value unassigned.

It is desirable to continue implying across the rest of the network; however, the current implication strategy will not be able to proceed. We cannot imply the value of the output of node f_2 without first knowing the output value of the right f_1 node. The inputs B (I_1) and D (I_2) of the right f_1 node are evaluated to '1'; both the first and the third table rows match these input values. Hence, we cannot apply an implication as we cannot identify a single suitable row. A typical way to proceed would be to employ a decision; yet, as discussed before, decisions may set an unsuitable value—we should employ them as late and as rarely as possible.

There is an alternative, though: if we analyze the truth table, we notice that the already existing input assignments match only rows where f_1 evaluates to '1'. Meanwhile, the second and fourth rows (in which f_1 evaluates to '0') cannot be fulfilled. Therefore, the only possible output value for the right f_1 node is '1' and we can safely assign the node's output value, even if we cannot set all the input values. We define this type of implication as *advanced implication*:

Definition 4: An *advanced implication* is a propagation that occurs when multiple assignments respect all previously set values and have the same output value. It sets all previously unassigned inputs/outputs whose values are equal in all match-



(d)

	x	y	z	dc_{size}	$mffc_{rank}$
row_1	0	-	0	1	0
row_2	-	0	0	1	1
row_3	1	1	1	0	1

Fig. 4: MFFC heuristic. In Figure a, a decision without the MFFC heuristic causes a conflict at y. In Figure b, we identify the MFFC (the orange-dotted triangle) and assign the DC to y's output as it does not belong to any MFFC. In Figure c, assigning '0' to the deeper right MFFC and a DC to the left minimizes conflicts.

ing assignments and leaves the different values unassigned.

In the example above, advanced implication sets the output of the right f_1 node to 1 at step 3 as both the first and the third rows contain this value. E remains unassigned as its values in the two rows are different. The benefit of advanced implication is, intuitively, clear: assigning more values will enable other implications and postpone the usage of undesired decision-making. For instance, in the example above, the advanced implication now enables the implication of G to 1 at step 4. We will quantitatively evaluate this benefit in Section VI.

V. Which Row is the Best?

When no further implications can be done, we have to make a decision by picking one out of multiple possible assignments from the candidate node's truth table. Yet, it is not always possible to predict the influence of a decision on the rest of the circuit. In this section, we propose a set of decision heuristics to increase the probability of avoiding a future conflict.

Don't-cares. Truth table rows can have *don't cares* (DCs). Choosing a row with DCs provides more flexibility for future propagations: leaving a particular input unassigned reduces the chance of conflict with other assignments at this network point. Consider an *and* gate whose output is set to '0', and the input values need to be decided on. One input must certainly be set to '0'. If we were to decide on a '0' or '1' value for the other input, this value could potentially conflict with some future propagation (e.g., if the same input is connected to another node that requires the opposite value); leaving it unassigned prevents this issue, while still enforcing the desired *and* gate function. In other words, selecting a truth table row with the largest number of DCs reduces the number of decision-assigned values and, consequently, the chance of conflict.

For every decision, SimGen ranks candidate truth table rows based on the number of DCs that each row contains, dc_{size}:

$$dc_{size}(row_i) = \sum_{j=0}^{N_{inputs}} dc(row_i(j)), \tag{1}$$

where $dc(row_i(j))$ is 1 if the value in row_i of j-th input is a DC. SimGen then prioritizes the row with the largest dc_{size}.

Max fanout free cones. A function's truth table may have rows with equal DCs amount. We propose using Max Fanout-Free Cones (MFFC) as an additional metric.

Figure 4a shows a circuit with initial assignments '0' and '1' to D and E. Assume that we first propagate the value of D. There are two possible assignments for the inputs of gate z, as the truth table below suggests: '0' to the output of x and DC to the output of y, and vice versa. Since they both have one DC, we randomly select one of the two—for instance, a DC to the output of x and '0' to y. The next step is the propagation of E. It is an implication since t is an *and* gate and both inputs must be '1'. Yet, this causes a conflict with the previous assignment to the output of y, which belongs to the fanin cones of both z and t. Assigning a DC to the output of y during the propagation of D would have been a wiser choice, as it would allow the subsequent assignment to set this value without a conflict, as shown in Figure 4b.

The key to avoiding conflicts is to identify gates that are shared across different fanin cones—they will be reached during propagations from different POs and, thus, conflicts may occur. Hence, it is favorable to assign them a DC when deciding. To this end, we rely on the concept of MFFC from Section II. Gates that are in the MFFC of the gate under decision lead exclusively to this gate and, thus, conflicts cannot occur—this is the case for x, which is the output of the MFFC of z (dashed in Figure 4b). Gates that are not in any MFFC lead to other gates as well and, thus, may be points of conflict—this is the case for y. Our strategy favors the assignment of a DC to the latter and, more generally, to smaller MFFCs.

We now describe our algorithm and illustrate it on Figure 4c. Like in the previous example, the initial assignment of output E is '0'; we have to propagate it down the *and* gate and there are two possible input combinations.

(1) *Calculate the MFFC for each node input under decision.* Figure 4c shows the gate z's MFFCs as dotted orange triangles.

(2) *Compute the depth of each MFFC.* The depth represents the average distance between each MFFC leaf and its output:

$$depth(m_i) = \frac{\sum_{j=0}^{N_{leaves}} (lev(out(m_i)) - lev(leaf(m_i, j)))}{N_{leaves}}, \tag{2}$$

where m_i represents the MFFC of the i-th input of the node under decision, N_{leaves} is the number of leaves of the MFFC, $out(m_i)$ is the highest-level node of the MFFC m_i, $leaf(m_i, j)$ is the j-th leaf node of the MFFC m_i, and $lev(j)$ is a function that returns the level of node j. Figure 4c shows the level next to each gate. The left MFFC has only one leaf, gate x, with level 3; its depth is 0. The right MFFC has three leaves, m, n, and y, with levels 1, 2, and 3; its depth is $((3-1)+(3-2)+(3-3))/3 = 1$. The higher the MFFC depth, the more conflicts are potentially avoided by a non-DC assignment at its output; thus, we prefer to assign DC values to outputs of MFFCs with lower depths.

(3) *Compute the rank of each truth table's row based on MFFC depths.* We compute the ranks of the rows as follows:

$$rank(row_i) = \sum_{j=0}^{N_{inputs}} (1 - dc(row_i(j))) \times depth(m_j) \tag{3}$$

where dc and $depth$ are the functions defined in Equations 1 and 2, and m_j is the MFFC of the j-th input. The truth table of z in Figure 4 has two rows for which z is '0', row_1 and row_2; their ranks are 0 and 1, respectively. We prioritize rows with a higher rank—in this case, row_2. As this metric on its own does not differentiate rows with different DC counts, (e.g., although not acceptable for an assignment of '0' to z, row_3 ranks the same as row_2, even though it has no DCs), we combine it with the previously described DC metric.

(4) *Calculate row priority based on DC and MFFC ranking.* Our final row priority metric is:

$$priority(row_i) = \alpha \times dc_{size}(row_i) + \beta \times rank(row_i) \quad (4)$$

where α and β are coefficients. We choose $\alpha >> \beta$ to prioritize the DC over the MFFC metric. We incorporate our priorities into a roulette wheel selection algorithm [18], where the priorities serve as probabilities of selecting a truth table row; the algorithm prefers rows with the fewest input value assignments and targets inputs with the lowest conflict chance.

VI. Evaluation

In this section, we evaluate SimGen. In Section VI-B, we analyze the effectiveness of our implication and decision strategies. In Section VI-C, we evaluate the impact of SimGen on SAT calls and runtime. We demonstrate the benefits of combining random simulation and SimGen in Section VI-D.

A. Methodology and Benchmarks

SimGen is open-sourced [1] and integrated into ABC [6] via the command "*&adv_sim_gen*". We evaluate SimGen on 100 benchmarks from VTR [23], EPFL [2], and ITC'99 benchmark suites [8] and from a previous work [26]. We omit benchmarks whose SAT sweeping runtime is under 1 ms. We use SAT sweeping with "*&fraig -x*" in ABC via the Glucose SAT solver [4] and we disable *Cec4_ManSimulate* in *Cec4_ManSweepNode* to avoid SAT counter-example simulations. For the OUTgold values, we use 3 vectors: all zeroes, all ones, and alternating zeroes and ones.

We use the ABC command "*if -K 6*" for technology mapping. Then, the sweeping tool receives as input the LUT-mapped version of the benchmark. Firstly, ABC executes a series of random simulations; in Section VI-B, we use a single round of random simulation, whereas we tune this number in Section VI-D. Once random simulation terminates, SimGen receives the equivalence classes, the LUT-mapped circuit, and the OUTgold values, and runs for 20 iterations. We evaluate the cost of the classes after separation as:

$$cost = \sum_{i=0}^{N} (size(i) - 1), \quad (5)$$

where N is the number of equivalence classes and $size(i)$ is the number of LUTs in class *i*. The function computes the number of SAT calls to execute in the worst-case scenario if the SAT tool does not generate useful counter-examples (e.g. if the nodes are all equivalent). Lower cost corresponds to a lower number of SAT calls and better class separation, indicating a better quality of input vectors. Due to the intrinsic runtime variability of the Glucose SAT solver, we repeat all benchmark runs 5 times and report the average values.

	RevS	SI+RD	AI+RD	AI+DC	AI+DC+MFFC
Cost	1.000	0.991	0.888	0.888	0.864 **(-14.6%)**
Simulation Runtime	1.000	1.204	1.263	1.262	1.130 **(+13.0%)**

TABLE I: Average normalized *Cost* and *Simulation Runtime* of our methods with respect to reverse simulation show cost improvement with minimal runtime increase, proving practicality.

B. Cost and Simulation Runtime Analysis

We use reverse simulation (*RevS*) as a baseline to evaluate different combinations of our implication and decision strategies. We consider simple and advanced implication (Section IV) with random decisions (respectively, *SI+RD* and *AI+RD*); we then combine advanced implication with the DC heuristic (*AI+DC*) and with both DC and MFFC heuristics (*AI+DC+MFFC*) for decision-making.

Table I shows the average cost value (Equation (5)) achieved by the different techniques and the average simulation time across all benchmarks after one round of random simulation. We normalized both values with respect to RevS. All SimGen's methods outperform RevS in terms of cost at the price of a slight simulation runtime increase. This outcome is expected since SimGen performs additional graph and logic function analyses to imply more aggressively and make informed decisions; this overhead will be compensated by SAT time reductions, as we will see in the next section. The last method achieves the best average cost, indicating the usefulness of all the methods we introduced. In the rest of the paper, we further evaluate *AI+DC+MFFC* and refer to it as SimGen.

C. SAT Calls and Runtime

In Table II and Figure 5, we compare the SAT calls and sweeping time of the SAT tool (Glucose) employed by RevS and SimGen for 47 benchmarks with the largest SAT runtime.

The SAT time and SAT calls follow similar trends: SimGen's decrease in SAT calls, generally, results in a decrease in SAT time with respect to RevS. The occasional discrepancies between the two metrics arise from variations in execution time for each call, stemming from differences in circuit complexities and target nodes, as well as Glucose's runtime variability. In only three cases (i.e., *6s392r*, *6s203b41* and *vga_lcd*), effective SAT counter-examples reduce the number of SAT calls in RevS more than in SimGen; this accidental effect in Glucose is orthogonal to the particular simulation strategy. The fact that SimGen improves SAT sweeping time in all but three benchmarks indicates its effectiveness.

In Figure 5, the columns for each benchmark indicate the normalized difference of cost (Equation (5)), number of SAT calls, SAT runtime, and total runtime (i.e., the sum of simulation and SAT runtime) of SimGen with respect to RevS. Generally, all metrics improve and follow the same decreasing trend as the cost, which is exactly what we aim to achieve. The reduction of total runtime in almost all benchmarks indicates that the occasional simulation time increases of SimGen (implied by Table I) are negligible compared to

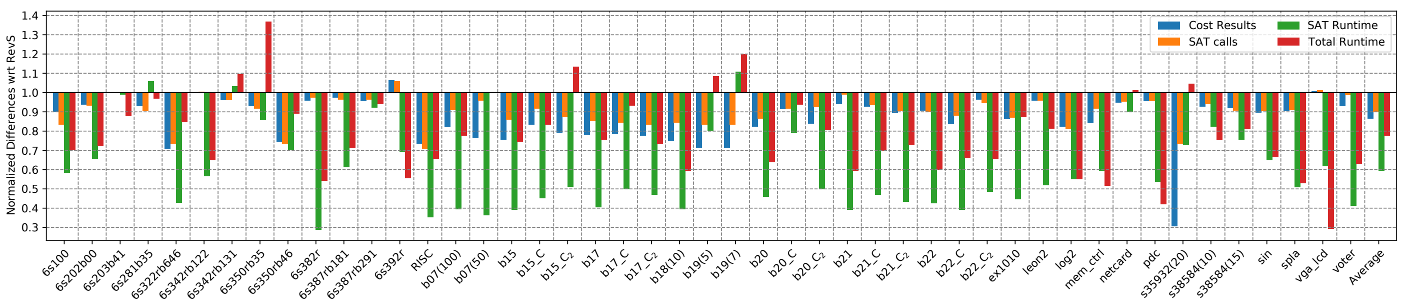


Fig. 5: Normalized difference of cost, SAT calls, SAT runtime, and total runtime of SimGen with respect to reverse simulation. SimGen achieves significant improvements in cost, SAT calls, and SAT runtime, at an occasional total time penalty due to prolonged simulations.

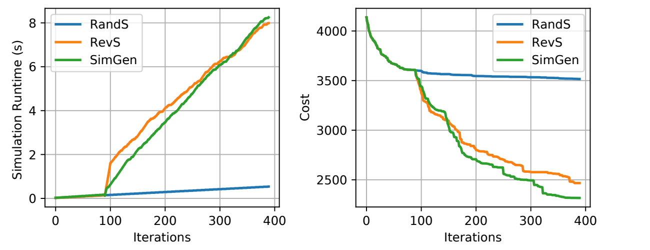


Fig. 6: Simulation runtime and cost across iterations for random simulation (*RandS*) and its combination with SimGen or RevS on benchmark *mem_ctrl*. RandS stalls in a local minimum after a few iterations. Switching to SimGen lowers cost despite higher runtime, highlighting SimGen's value.

SimGen's significant SAT runtime savings. The main takeaway is that, in most benchmarks, SimGen Pareto dominates RevS by reducing cost, SAT calls, SAT runtime, and total runtime. Occasionally, it achieves Pareto-optimality by achieving lower cost at a minor runtime expense; notably, it is never Pareto-dominated by RevS. This shows SimGen's broad usefulness and effectiveness. In Section VI-D, we propose a practical way to combine SimGen with random simulation to exploit SimGen's benefits while also improving simulation time.

D. Random Simulation and SimGen

Random simulation (see Section II) is fast but lacks circuit insight and hinders class separation, while guided simulation (reverse simulation and SimGen) improves class splitting with efficient inputs but can be slower (see Table I). We show that combining these techniques, especially random simulation and SimGen, effectively leverages their complementary benefits.

Figure 6 compares the cost and simulation runtime of three types of simulation runs for the benchmark *mem_ctrl*: (1) Only random simulation (RandS), (2) RandS followed by reverse simulation (RevS), and (3) RandS followed by SimGen. In the second and third scenarios, we switch to RevS and SimGen, respectively, after random simulation achieves the same cost in three consecutive iterations. As the figure shows, RandS quickly reduces the number of SAT calls in the first iterations, but it soon reaches a local minimum and all subsequent simulations improve the cost only marginally or not at all. In contrast, SimGen and RevS continue splitting equivalence classes and reducing the overall cost, but at a runtime increase; the cost achieved by SimGen is superior to that of RevS.

Bmk	SAT calls		SAT time (s)	
	RevS	SGen	RevS	SGen
6s100	8.04e3	**6.71e3**	1.08	**0.63**
6s202b00	1.29e3	**1.21e3**	0.04	**0.02**
6s203b41	**817**	818	0.04	0.04
6s281b35	5.19e3	**4.69e3**	**1.08**	1.15
6s322rb646	28.09e3	**20.67e3**	0.65	**0.28**
6s342rb122	1.58e3	1.58e3	0.09	**0.05**
6s342rb131	1.64e3	**1.57e3**	**0.09**	0.10
6s350rb35	42.38e3	**38.86e3**	0.98	**0.84**
6s350rb46	56.69e3	**41.52e3**	1.21	**0.85**
6s382r	9.96e3	**9.70e3**	1.71e3	**489.95**
6s387rb181	974	**939**	0.04	**0.03**
6s387rb291	978	**942**	0.04	0.04
6s392r	**2.38e3**	2.51e3	0.28	**0.19**
RISC	714	**504**	0.05	**0.02**
b07(100)	1.69e3	**1.54e3**	15.32	**6.05**
b07(50)	836	**801**	1.79	**0.65**
b15	414	**356**	0.08	**0.03**
b15_C	438	**402**	0.06	**0.03**
b15_C_2	460	**402**	0.08	**0.04**
b17	1.05e3	**898**	0.18	**0.07**
b17_C	1.15e3	**967**	0.17	**0.09**
b17_C_2	1.47e3	**1.23e3**	0.24	**0.11**
b18(10)	35.39e3	**29.87e3**	685.06	**270.19**
b19(5)	32.75e3	**27.31e3**	87.96	**70.58**
b19(7)	48.22e3	**40.15e3**	**187.63**	207.96
b20	162	**140**	0.02	**0.01**
b20_C	170	**156**	0.01	0.01
b20_C_2	250	**231**	0.04	**0.02**
b21	218	**216**	0.03	**0.01**
b21_C	252	**236**	0.02	**0.01**
b21_C_2	221	**200**	0.03	**0.01**
b22	310	**279**	0.05	**0.02**
b22_C	220	**194**	0.03	**0.01**
b22_C_2	277	**262**	0.03	**0.02**
ex1010	1.91e3	**1.66e3**	0.03	**0.02**
leon2	70.31e3	**67.34e3**	2.74	**1.43**
log2	287	**233**	826.50	**454.71**
mem_ctrl	886	**813**	0.14	**0.08**
netcard	32.92e3	**31.37e3**	0.78	**0.70**
pdc	1.35e3	**1.29e3**	0.04	**0.02**
s35932(20)	1.23e3	**902**	2.62	**1.91**
s38584(10)	2.53e3	**2.38e3**	12.94	**10.66**
s38584(15)	4.29e3	**3.89e3**	78.05	**59.02**
sin	105	**95**	2.33	**1.52**
spla	1.12e3	**1.02e3**	0.03	**0.01**
vga_lcd	**4.52e3**	4.57e3	0.09	**0.06**
voter	524	**518**	1.25	**0.52**

TABLE II: *SAT calls* and *SAT time* of the SAT sweeping tool for 47 benchmarks of Table I with highest SAT times. *RevS* and *SGen* are reverse simulation and SimGen. Generally, the reduction in the number of SAT calls translates into decreased SAT time.

These results show the synergy between random simulation and SimGen: RandS enables fast class division, while SimGen excels when RandS is stuck, highlighting the need to integrate SimGen into sweeping simulators.

VII. Conclusion

Simulation can accelerate CEC and SAT sweeping; yet, it is only effective if provided with appropriate input vectors. We propose SimGen, a framework for effective simulation input vector generation. SimGen borrows ATPG concepts to leverage structural and logic information of the network under simulation, thus generating input vectors that are customized to the network at hand and suitable for separating its equivalent classes. We explore several ATPG concepts and their interactions, and demonstrate their success over prior strategies (i.e., random and reverse simulation): at a modest runtime increase, SimGen leverages simulation more effectively and reduces the number and runtime of SAT calls. Our open-source framework serves as a foundation for implementing and exploring further simulation vector generation strategies.

References

[1] "SimGen: Simulation Pattern Generation for Efficient Equivalence Checking" source code, https://doi.org/10.5281/zenodo.12735762, July 2024.

[2] L. Amarú, P.-E. Gaillardon, and G. De Micheli. The EPFL combinational benchmark suite. In *24th International Workshop on Logic & Synthesis*, pages 1–5, Mountain View, CA, June 2015.

[3] L. Amarú, F. Marranghello, E. Testa, C. Casares, V. Possani, J. Luo, P. Vuillod, A. Mishchenko, and G. De Micheli. SAT-sweeping enhanced for logic synthesis. In *Proceedings of the 57th ACM/IEEE Design Automation Conference*, pages 1–6, Virtual Event, July 2020.

[4] G. Audemard and L. Simon. On the glucose SAT solver. *International Journal on Artificial Intelligence Tools*, 27:1–25, Feb. 2018.

[5] D. Brand. Verification of large synthesized designs. In *Proceedings of the 12th International Conference on Computer-Aided Design*, pages 534–537, Santa Clara, CA, Nov. 1993.

[6] R. Brayton and A. Mishchenko. ABC: an academic industrial-strength verification tool. In *Proceedings of the 22nd International Conference on Computer-Aided Verification*, page 24–40, Berlin, July 2010.

[7] Cadence Design Systems. *Conformal Equivalence Checker Datasheet, https://www.cadence.com/en_US/home/resources/datasheets/conformal-equivalence-checker-ds.html*, 2024.

[8] F. Corno, M. Reorda, and G. Squillero. RT-level ITC'99 benchmarks and first ATPG results. *IEEE Design & Test of Computers*, 17(3):44–53, Aug. 2000.

[9] H. Fujiwara. FAN: A fanout-oriented test pattern generation algorithm. In *Proceedings of the 18th IEEE International Symposium on Circuit and Systems*, pages 671–674, Kyoto, Japan, June 1985.

[10] P. Goel and B. C. Rosales. PODEM-X: An automatic test generation system for VLSI logic structures. In *Proceedings of 18th Design Automation Conference*, pages 260–268, Nashville, TN, June 1981.

[11] R. Hulle, P. Fiser, J. Schmidt, and J. Borecky. SAT-ATPG for application-oriented FPGA testing. In *Proceedings of 15th Biennal Baltic Electronics Conference*, pages 83–86, Tallinn, Nov. 2016.

[12] S. Krishnaswamy, H. Ren, N. Modi, and R. Puri. DeltaSyn: An efficient logic difference optimizer for ECO synthesis. In *Proceedings of the 28th International Conference on Computer-Aided Design*, pages 789–796, San Jose, CA, Nov. 2009.

[13] A. Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *Proceedings of the 23rd International Conference on Computer-Aided Design*, pages 50–57, San Jose, CA, Nov. 2004.

[14] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th Design Automation Conference*, pages 263–268, New York, NY, June 1997.

[15] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, Dec. 2002.

[16] R. P. Lajaunie and M. S. Hsiao. An effective and efficient ATPG-based combinational equivalence checker. In *Proceedings of the 15th ACM Great Lakes Symposium on VLSI*, page 248–253, Chicago, Illinois, Apr. 2005.

[17] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli. A simulation-guided paradigm for logic synthesis and verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(8):2573–2586, Aug. 2022.

[18] A. Lipowski and D. Lipowska. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications*, 2012.

[19] F. Lu, L.-C. Wang, K.-T. Cheng, and R.-Y. Huang. A circuit SAT solver with signal correlation guided learning. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 892–897, Munich, Mar. 2003.

[20] A. Mishchenko, S. Chatterjee, and R. Brayton. Improvements to technology mapping for LUT-based FPGAs. In *Proceedings of the ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, pages 41—-49, Monterey, CA, Feb. 2006.

[21] A. Mishchenko, S. Chatterjee, R. K. Brayton, and N. Een. Improvements to combinational equivalence checking. In *Proceedings of the 25th International Conference on Computer-Aided Design*, pages 836—-843, San Jose, CA, Nov. 2006.

[22] A. Mishchenko, J. Zhang, S. Sinha, J. Burch, R. Brayton, and M. Chrzanowska-Jeske. Using simulation and satisfiability to compute flexibilities in Boolean networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(5):743–755, May 2006.

[23] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. ElDafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. P. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz. VTR 8: High performance CAD and customizable FPGA architecture modelling. *ACM Transactions on Reconfigurable Technology and Systems*, 13(2):1–55, Jun. 2020.

[24] H. Pan, R. Zhang, Y. Xia, L. Wang, F. Yang, X. Zeng, and Z. Chu. A semi-tensor product based circuit simulation for SAT-sweeping. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6, Valencia, Mar. 2024.

[25] J. Roth. Diagnosis of automata failures: A calculus and a method. In *IBM Journal of Research and Development*, pages 278–291, USA, July 1966. IBM Corp.

[26] H.-T. Zhang, J.-H. R. Jiang, L. Amarú, A. Mishchenko, and R. Brayton. Deep integration of circuit simulator and SAT solver. In *Proceedings of 58th ACM/IEEE Design Automation Conference*, pages 877–882, San Francisco, CA, Dec. 2021.

[27] H.-T. Zhang, J.-H. R. Jiang, and A. Mishchenko. A circuit-based SAT solver for logic synthesis. In *Proceedings of the 40th International Conference On Computer-Aided Design*, pages 1–6, Munich, Dec. 2021.