SimGen: Simulation Pattern Generation for Efficient Equivalence Checking

Carmine Rizzi
ETH Zurich, Switzerland
crizzi@ethz.ch

Sarah Brunner
ETH Zurich, Switzerland
sarbrunn@ethz.ch

Alan Mishchenko
UC Berkeley, USA
alanmi@berkeley.edu

Lana Josipović
ETH Zurich, Switzerland
ljosipovic@ethz.ch

ABSTRACT

Combinational equivalence checking for hardware designs 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 a lot of time disproving nodes that appear equivalent under random simulation. This work focuses on the generation of expressive simulation patterns that help disprove the majority of non-equivalent nodes without the SAT solver, thus substantially speeding up equivalence checking. We propose and evaluate several novel heuristics and strategies inspired by Automatic-Test Pattern Generation (ATPG). Experiments demonstrate the effectiveness of using the resulting simulation patterns, compared to those generated by state-of-the-art random and guided simulation.

1 INTRODUCTION

Combinational equivalence checking (CEC) determines whether two circuit networks or subnetworks have an equivalent logic function. A common approach to discovering equivalent points in two circuit networks is *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 [13]. However, a naive checking of all point pairs is time-consuming due to the requirement to disprove equivalence among numerous non-matching points [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 time-consuming equivalence checks.

The key to the success of simulation is to employ simulation vectors that will effectively partition the equivalence classes. Several approaches use fully random simulation patterns [12, 18]. However, random simulations cannot guarantee the separation of a specific class and, consequently, often remain stuck at a local minimum. For this reason, Zhang et al. [26] propose *reverse simulation* to efficiently separate equivalence classes and reduce the number of expensive equivalence proofs (e.g., using SAT calls). Reverse simulation generates 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.

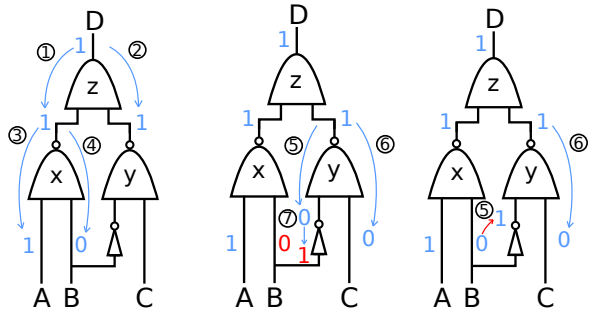


Figure 1: The limitations of reverse simulation. Figures a and b show the steps of a classic reverse simulation. Assuming that D needs to evaluate to '1', reverse simulation propagates values backward to find a suitable input assignment. In Figure 1a, a series of propagations (represented by arrows and enumerated by their order) assign '1' and '0' to A and B. Yet, the propagation of the remaining values, shown in Figure 1b, causes a conflict when it tries to set the input B to '1' at step 7 (in red). Applying an implication would mitigate this issue, as shown in Figure 1c (the red arrow at step 5): the 0 value of B implies the value of 1 at the left input of gate y. This, in turn, sets the value of input C to '0' (step 6 in the figure). This strategy successfully identifies a correct input vector without a collision. The goal of SimGen is to exploit this and other strategies to reduce the number of conflicts and improve the success rate of reverse simulation.

If reverse simulation successfully reaches the network's inputs, the values assigned to the inputs serve as a simulation vector that aims to disprove the equivalence of the target nodes.

1.1 The Limitations of Reverse Simulation

Figure 1 illustrates the concept of reverse simulation. Assume that the node z belongs to an equivalence class; we select this node as the target node (step 1 of the procedure above). For simplicity, we do not show the other nodes of the class and their networks. We aim to demonstrate that node z is non-equivalent to other nodes of the same class. To this end, we need to identify a simulation pattern that, when applied to the inputs of these networks, produces a value at D opposite to the other nodes' outputs. Thus, the second step of the procedure above assigns D with a value (e.g., '1'). In the third step, we assign values to the inputs of gate z that ensure the desired value of D (e.g., by assigning '1' to the output of gate x and to the one of y). This procedure repeats for the preceding nodes (see step 4). For gate x, it assigns to the network's inputs, A and B, the values shown in Figure 1.a1 ('1' and '0' respectively).

Similarly, we have to decide which values the inputs of gate y should have to obtain '1' at its outputs. Since gate y represents a

nand gate, there are three possible input assignments (e.g, '0' to one input and '1' to the other or '0' to both inputs); we randomly choose to assign '0' to both inputs. In this way, input C and the output of the inverter have the value '0'. The last step assigns '1' to the input of the inverter since we allocated '0' to its output. Yet, this generates a *collision* since the algorithm already assigned '0' to input B, as depicted in Figure 1.a2. At this point, the reverse simulation fails and terminates without identifying an appropriate input vector. Thus, the procedure repeats with the next pair of target nodes.

This example 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.

1.2 SimGen: A Guided and Controlled Reverse Simulation Strategy

Figure 1.b shows an alternative solution to the strategy above. After setting the value of input B to '0', we realize that an inverter uses the value of B as input; due to the inverter's logic function, we know that its output is '1' and make this assignment before making any other decision. This information helps us to assign the other input of y: now, the only possibility that respects the previous assignments is to set input C to '0'. At this point, all internal values are assigned with no conflict and we successfully reached the inputs of the network with a valid simulation vector that guarantees the desired value at output D. Clearly, exploiting functional and structural information can help with avoiding conflicts and obtaining a simulation vector.

In this paper, we propose SimGen, an open-source simulation vector generator for effective and controlled equivalence class partitioning. The main idea of our strategy is that, after determining a desired simulation value of targetted network points (e.g., those of a critical equivalence class), we traverse the graph to identify a simulation vector compatible with the desired targets. Yet, in contrast to standard reverse simulation, we incorporate techniques from the Automatic-Test Pattern Generation (ATPG) domain 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 these 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.

The rest of the paper is organized as follows. Section 2 describes previous works related to SimGen and the relevant background. Section 3 outlines the structure of SimGen. Section 4 and Section 5 explain two fundamental ATPG concepts incorporated in SimGen. Section 6 details our experimental setup and results.

2 BACKGROUND AND RELATED WORK

In this section, we present previous work related to SimGen and the background necessary to understand its mechanisms.

2.1 Boolean Network

A Boolean network is a Directed Acyclic Graph (DAG). Each node of this graph represents a logic function corresponding to a logic gate or any logic expression. In this paper, without the loss of generality, we assume that each node produces 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 set 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 [19].

2.2 CEC and SAT-Sweeping

Combinational equivalence checking (CEC) evaluates the equivalence of the outputs of two circuits. Different verification tools can prove this equivalence; initially, they were based on BDDs [13, 14] but, due to their large memory consumption, researchers' interest has shifted towards ATPG and SAT-based CEC methods [4, 15, 20]. A common technique across these methods is *sweeping*; it proves the equivalence of the internal nodes to simplify the equivalence proof of the circuit's outputs. Apart from CEC, sweeping is a key constituent of various applications: extracting structural choices for technology mapping [6], pipelining an untimed circuit (e.g., to track the circuit's topological modifications and accurately identify its critical path [23, 25]), logic optimizations [16], and Engineering Change Order (ECO) synthesis [11]. Yet, to avoid excessive calls to the verification tool, the formal proof is typically preceded by circuit simulation, which can prove the inequality of two candidate nodes with lower memory and runtime consumption [11, 20].

2.3 Circuit Simulation

Circuit simulation determines the value of each wire based on the values assigned to the circuit's PIs (also called input vectors) and the functions within the circuits. Different applications employ simulation to quickly assess circuit properties [16, 21].

The workflow of a typical sweeping tool is depicted in Figure 2, inside the blue box. This tool receives as an input two networks. The first step of this flow is circuit simulation. It generates a set of random input vectors and simulates them through both networks. Then, it categorizes nodes into multiple equivalence classes. If two nodes have the same values at their outputs across all the simulations, they are in the same equivalence class. Once all nodes are considered, the simulator can send these classes to a verification tool like BDD or SAT to test the nodes' equivalence. The verification tool will select a pair of nodes from the equivalence classes and it will attempt to find an input combination for which they are not equivalent. If it succeeds, it will obtain a counter-example that the random generator could not create; it will thus send this input vector to the circuit simulator to separate additional classes. Hence, the main sources of the input vectors in SAT-sweeping are

the counter-examples from the verification tools and the randomly generated input vectors. However, both these techniques come with restrictions: the former requires the use of the verification tool and can generate only one input vector per call; the latter cannot generate input vectors to disprove the equivalence of specific nodes and it might become trapped in a local minimum.

For this reason, researchers proposed alternative simulation techniques. Mischenko et al. [20] introduce a *1-distance* simulation vector which selectively flips one bit of the input vector obtained from a counter-example. Yet, the effectiveness of the flipping is difficult to control and predict. Lee at al. [16] and Amarù et al. [3] employ a SAT solver to generate respectively "expressive" and "high-toggle rate" input vectors. Yet, the newly proposed input vector still depends on SAT calls. On the other hand, Zhang et al. [26] propose "reverse simulation" that does not depend on SAT or other verification tools. Despite its effectiveness with respect to random simulation, it is prone to failure, as we have shown in Section 1.

2.4 ATPG

Automatic-Test Pattern Generation (ATPG) represents a fundamental step in circuit fabrication [10, 15]. It generates input vectors to expose the presence of a fault inside a fabricated circuit. The generation of test patterns is divided into two steps: activation and propagation. In the first step, the tool identifies the values of the PIs needed to activate the desired value to test the fault; in the second, it selects the input values to propagate the fault value to the network's output and make it visible outside the circuit.

Definition 2.1. 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. Analogously to ATPG, SimGen propagates the value of a certain node towards the network's PIs to retrieve a meaningful input vector.

The D-algorithm [24] is one of the first ATPG methods that employs information related to the circuit's structure for efficient test generation; PODEM [9] and FAN [8] build upon this algorithm. They all rely on backtracking: once a collision occurs during the signal propagation, they backtrack to their last decision and change their choice. Yet, backtracking is time-consuming; thus, it is advantageous to avoid conflicts whenever possible. To this end, PODEM employs forward implication to assign values to internal nodes, whereas FAN uses MFFCs to identify independent circuit portions and to postpone the propagation of specific values. To reduce execution time, SimGen omits backtracking and exploits different variations of implication and MFFCs, as we will later discuss.

Two fundamental concepts that we inherit from ATPG are implication and decision.

Definition 2.2. 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.

We note that 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 discussed in the example of Figure 1. Some implementations [5] 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.

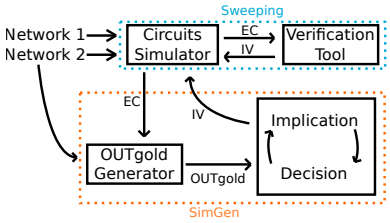


Figure 2: Workflow of a sweeping tool with our SimGen plugin. The flow receives as input two networks for sweeping. The circuits pass through a simulator that generates random input vectors (IV). The extracted equivalence classes (EC) are then sent to the verification tool or to SimGen. SimGen generates OUTgold values that aim to split the equivalence classes for one of the two networks. Then, it generates the IV that propagates these values towards the inputs by interleaving two propagation techniques, *decision* and *implication*.

Definition 2.3. A *decision* is a propagation that occurs when multiple available assignments respect all previously assigned values; it chooses one of the possible assignments according to a decision policy. Then, it sets the values of the selected assignment.

As shown in Figure 1, to set the output of gate z to '1', there is only one possible combination for the inputs (e.g., both should be equal to '1'). Hence, this propagation is an implication. On the other hand, the propagation of the output of gate x is a decision since we could select among three possible assignments (e.g., one of the two inputs is set to '0' or both are set to '0'). We will see in the following sections how SimGen uses these techniques to efficiently compute input vectors to separate equivalent classes.

3 SIMGEN FLOW OVERVIEW

So far, we highlighted the constraints and limits of the state-of-the-art and hinted at the potential application of ideas from the ATPG domain. We next show how we leverage these concepts within the SimGen framework to create impactful simulation input vectors.

Figure 2 depicts the interaction of SimGen with a typical sweeping tool. SimGen receives as inputs the equivalence classes and a network. There are two main steps in SimGen:

(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 aims to maximize the number of target nodes whose value is equal to its desired OUTgold value. For simplicity, we assign alternating values of zeros and ones as OUTgold values according to the node IDs to split them into different classes. Other 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.* Input vectors are the vectors applied to the PIs of the network during simulation. As shown in Figure 2, the main steps for the generation of the input vector are *implication*

Algorithm 1: Input Vector Generator

```
   getInputVectors(OUTgold[N_targets], targetNodes[N_targets],
     network, implicationStrategy, decisionStrategy){
1  nodeVals = []
2  orderedTargetNodes = orderTargetNodes ( targetNodes ) ;
3  for targetNode in orderedTargetNodes do
4      initVals = nodeVals;
5      nodeVals[targetNode] = OUTgold[targetNode];
6      listDfs = dfs ( targetNode );
7      candidateNode = targetNode;
8      while ! PIsSet ( listDfs , nodeVals ) do
9          tmpVals = implication ( nodeVals, candidateNode ,
             implicationStrategy );
10         conflict= compareVals ( tmpVals , nodeVals );
11         if conflict== 1 then
12             nodeVals= initVals;
13             break;
14         nodeVals= tmpVals ;
15         candidateNode = latestUpdated ( listDfs , nodeVals );
16         decision ( nodeVals, candidateNode , decisionStrategy );
17 return nodeVals; }
```

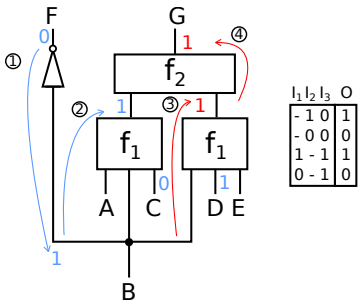


Figure 3: Implication example. The truth table on the right describes the nodes f_1; node f_2 is an *and* gate. Initially, the output F and the inputs C and D have the values '0', '0', and '1'. In step 1, an implication from F assigns '1' to A. This enables the assignment of '1' to the output of the left node f_1. Yet, this implication strategy does not apply further. However, by examining the truth table, *advanced implication* can determine the output of this node can be only '1'; it enables this value's propagation and avoids a decision. Consequently, there is a new implication opportunity for node f_2 that assigns its output to '1'. Implication opportunities enabled by advanced implication are highlighted in red in the figure.

and *decision*. The rest of the paper will focus on how to efficiently employ and interleave these techniques.

If SimGen cannot generate an input vector to respect at least a pair of target nodes of opposite OUTgold values, the simulation is skipped and SimGen receives a new equivalence class to split.

3.1 SimGen's Input Vector Generation Algorithm

Algorithm 1 describes our input vector generation strategy. The algorithm's core is a nested loop. In the outer loop (Line 3), we iterate through the target nodes and assign them the desired OUTgold values. In the inner loop (Line 8), we search for a PI assignment that satisfies the target node value. We execute a series of implications; if we identify a conflict (Line 11), we terminate the inner loop and select a new target node. If no conflict is found and there is still a candidate node, we make a decision. We repeat the inner loop until all PIs in the fanin cone of the target node are set or we identify a conflict. The rest of this section details the algorithm.

The inputs to our algorithm are the desired *OUTgold* values, the *network* with the target nodes, the list of *targetNodes*, and the implication and decision strategies (respectively, *implicationStrategy* and *decisionStrategy*). $N_{targets}$ is the number of target nodes. The variable *nodeVals* is the main result of this function. It represents a mapping between nodes' IDs and the node's output values for all network nodes, including the PIs and POs. Initially, the *nodeVals* does not contain any value.

In Line 2, we order the target nodes by decreasing network depth and save them into the variable *orderedTargetNodes* ; the first nodes to be processed will be the ones furthest from the PIs. Then, we save the initial value assignment in the variable *initVals* and assign a value to the first target node. We generate the list of nodes in the fanin cone of the target node and save the result of the depth-first search in the variable *listDfs* . The first candidate node is the target node. In Line 8, in a while loop, we search for a PI assignment that results in the desired OUTgold value at this node. The condition of the while loop checks if the PIs present in the variable *listDfs* (i.e., those that belong to the input cone of the target node) have already been set in the *nodeVals* variable.

The next step is the *implication* function. It starts from the *candidateNode* and uses the values in *nodeVals* to identify implication opportunities. It executes a certain number of implications according to the strategy specified with the variable *implicationStrategy* and returns the new values after the implications. We then validate if the implication assignments conflict with any previous assignments using the function *compareVals* . If there is a conflict, the values are reset to the initial values before the OUTgold assignment and the search for this target node terminates; we repeat the procedure for a new target node from the list.

On the other hand, if there is no conflict, we update the node values with the implied ones. Then, we set the latest updated value in *nodeVals* (i.e., the input cone of the targeted node) as the new *candidateNode* . Since there are no implication opportunities at this point, we decide its inputs using the *decision* function. Depending on the variable *decisionStrategy*, this function will select a different assignment and modify the *nodeVals*.

In the following sections, we will explain the strategies used for implication and decision and the insights behind them.

4 HOW MUCH TO IMPLY?

Implication can drastically affect the quality of results and the presence of conflicts. In this section, we investigate the effectiveness of standard implication and propose a more powerful alternative.

Our procedure to apply implication from Definition 2.2 considers the truth table of the node's function and its already assigned input/output values. We iterate through the rows of the truth table and identify those that match the already set values. If only one row matches the already assigned values, we apply implication by

assigning the row's values to the previously unassigned inputs and output. The newly set values may create new implication opportunities that we can exploit next, as per Algorithm 1.

A complete 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 respectively values '0' and '1'. We start the propagation from output F. Given the value of F, the only possible inverter input value is '1'; hence, we imply this value for input B. This new assignment generates a sequence of implication opportunities: the values now assigned to B and C fully respect only the first row of the truth table of f_1. Thus, we imply the values of O as 1, as specified by the row. Note that 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 applying the same strategy to the rest of the network; however, the implication strategy that we have applied so far will not be able to proceed. We now know that inputs B and D both evaluate to '1'. In this case, two table rows match these input values: the first row and the third one. Hence, we cannot apply an implication as we cannot identify a single suitable row. A typical way to proceed would be to now 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 4.1. An *advanced implication* is a propagation that occurs when multiple available assignments respect all previously assigned values and evaluate the output to the same value. It sets all previously unassigned inputs/outputs whose values are equal in all matching assignments and leaves the different values unassigned.

In the example above, advanced implication sets O to 1 as both the first and the third row 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. We will quantitatively evaluate this benefit in Section 6.

5 WHICH ROW IS THE BEST?

When no further implications can be done, we have to make a decision for a specific candidate node (i.e., we have to pick one out of multiple possible assignments from the candidate node's truth table). Of course, it is not always possible to predict how an assignment in one portion of the circuit could influence another. In this section, we propose a set of heuristics to decide on a truth table row depending on the probability of avoiding a future conflict.

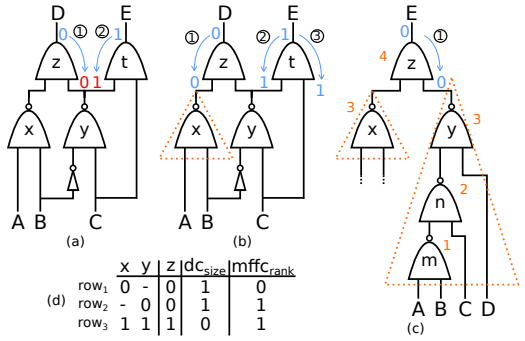


Figure 4: Example of value assignment without and with the MFFC heuristic. In graph a, we randomly assign a DC value to x, which causes a conflict at y. In graph b, we identify the MFFC (the orange-dotted triangle) and assign the DC to the output of y since it does not belong to any MFFC. In graph c, the right MFFC has a higher depth than the left one. We assign '0' to the right one and a don't-care to the left one reducing the possible number of collisions. Hence, we can exploit MFFCs to avoid conflicts.

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 of its inputs 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.

We incorporate this strategy in SimGen; for every decision, we rank candidate truth table rows based on the number of DCs that each row contains, dc_{size}:

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

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

Max fanout free cones. Our strategy above shows the importance of considering the DCs during row selection. However, the truth table of a function could contain multiple rows with an identical number of DCs. For this reason, we propose an additional metric for the decision of a row based on Max Fanout-Free Cones (MFFC).

Figure 4.a shows a circuit with initial assignments '0' and '1' to D and E, respectively. 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 gate x and DC to the output of gate 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 gate x and '0' to the other input, as shown in the figure. 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 and the propagation fails. The reason

is the assignment of '0' to the output of y, which is the input of two gates connected to two different POs. In other words, y 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; this scenario is shown in Figure 4.b.

The key to avoiding conflicts as in this example is to identify gates that are shared across different fanin cones—these gates will be reached during propagations from different POs and, thus, conflicts may occur. Hence, it is favorable to assign them a DC when making a decision. To this end, we rely on the concept of MFFC, defined in Section 2.1. 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 gate x, which is the output of the MFFC of z, shown in dashed in Figure 4.b . 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 gate y. Our strategy favors the assignment of a DC to the latter; more generally, it assigns DCs to smaller MFFCs.

We now describe the main steps of our algorithm and illustrate it on Figure 4.c. Just 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 input of the node under decision.* In Figure 4.c, the MFFCs of gate z are shown as dotted orange triangles.

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

$$depth = \sum_{leaf=0}^{N_{leaves}} \frac{(level(output) - level(leaf))}{N_{leaves}}, \tag{2}$$

where N_{leaves} is the number of leaves of the MFFC, *output* is the highest-level node of the MFFC, and $level(i)$ is a function that returns the level of node i. Figure 4.c 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:

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

where $dc(input)$ and $depth(input)$ are the values defined in Equations 1 and 2. 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. Yet, 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). Hence, we combine the MFFC ranking with the previously described DC metric.

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

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

where dc_{size} and $mffc_{rank}$ are the two functions of the two heuristics and α and β are two coefficients. We choose $\alpha >> \beta$ to prioritize DC count over the MFFC metric. Without the loss of generality, we incorporate our priorities into a standard roulette wheel selection algorithm [17], where the priorities serve as probabilities of selecting a truth table row; the algorithm thus preferentially chooses rows with the fewest input value assignments and targets inputs with the lowest chance of conflict.

6 EVALUATION

In this section, we evaluate SimGen and compare it with reverse [26] and random simulation. In Section 6.2, we analyze the effectiveness of our different implication and decision strategies and show their superiority over reverse simulation at a negligible runtime cost. In Section 6.3, we evaluate the impact of SimGen on SAT calls and runtime. Finally, we demonstrate the effectiveness of combining random simulation and SimGen in Section 6.5.

6.1 Methodology and Benchmarks

The workflow of SimGen is shown in Figure 2. Our open-source tool [1] is integrated with ABC [5] which performs SAT sweeping. We evaluate SimGen on a total of 42 benchmarks from the VTR [22], EPFL [2], and ITC'99 benchmark suites [7]. We omit benchmarks whose SAT sweeping runtime is below 1 ms.

We use the command "*if -K 6*" from ABC to apply technology mapping to each benchmark. Then, the sweeping tool receives as input the LUT-mapped version of the benchmark. Firstly, ABC executes a series of random simulations; in our experiments of Section 6.2, we employ a single round of random simulation, whereas we will tune this number in Section 6.5. Once the random simulation terminates, SimGen receives the equivalence classes, the LUT-mapped circuit, and the OUTgold values, and runs for 20 iterations. Our goal is to separate LUTs from the same equivalence class; thus, we use OUTgold with an equal number of zeros and ones for the nodes of each class. We use the following formulation to evaluate the cost of the classes:

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

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 a better separation of the classes.

6.2 Cost and Simulation Runtime Analysis

We use reverse simulation (*RevS*) as a baseline to evaluate different combinations of our implication and decision strategies. In particular, we evaluated the simple and advanced implication methods (Section 4) with random decisions (respectively, *SI+RD* and *AI+RD*). Then, we combined advanced implication with the don't-care heuristic and with the fanout-free cone heuristic (Section 5) as decision techniques (i.e., *AI+DC* and *AI+DC+MFFC*).

Table 1 shows the average cost value (Equation (5)) achieved by the different techniques and the average execution time across

	RevS	SI+RD	AI+RD	AI+DC	AI+DC+MFFC
Cost	1.000	0.814	0.812	0.810	0.807 **(-19.3%)**
Simulation Runtime	1.000	1.204	1.263	1.262	1.130 **(+13.0%)**

Table 1: Average normalized *Cost* and *Simulation Runtime* of our methods with respect to reverse simulation (*RevS*) for a total of 42 benchmarks from VTR [22], EPFL [27], and ITC'99 [7] benchmark suites. We evaluate simple implication with random decision (*SI+RD*), advanced implication with random decision (*AI+RD*), advanced implication and don't-care heuristic (*AI+DC*), and the MFFC heuristic on top of the latter (*AI+DC+MFFC*). The addition of each method contributes to the cost improvement. The minor runtime increase indicates the practicality of our methods.

all the benchmarks after one round of random simulation. We normalized both values with respect to RevS. If the average value of a method is lower than 1, this method has a smaller cost or runtime. All the proposed methods from SimGen outperform on average the reverse simulation approach in terms of cost at the price of a slight runtime increase. Especially, the last method achieves the best average cost, indicating the usefulness of all implication and decision methods we presented in the previous sections.

Figure 5 shows the per-benchmark results of *AI+DC+MFFC*, where the first and the second bar indicate the normalized difference in cost and simulation runtime of SimGen with respect to RevS. SimGen either improves both metrics or trades off runtime for improved cost, which aligns with our optimization goals. Also, it achieves significant SAT call reductions, as we will discuss next. In the rest of the paper, we refer to *AI+DC+MFFC* as SimGen.

6.3 SAT Calls and Runtime

To assess the benefits of SimGen in reducing SAT sweeping time, we evaluated the number of SAT calls and runtime of the SAT sweeping tool inside ABC of RevS and SimGen. Table 2 shows the number of SAT calls and SAT time of these two techniques, with SimGen's relative improvements in SAT time. 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 arise from variations in execution time for each call, stemming from differences in circuit complexities and target nodes. The fact that SimGen improves execution time over the majority of benchmarks indicates its effectiveness.

Figure 5 links the cost and runtime of SimGen with the SAT sweeping calls and runtime: the first and second columns for each benchmark indicate the normalized difference of cost and execution runtime of SimGen with respect to reverse simulation, the second and third refer to SAT sweeping performance. Generally, the cost, the SAT calls, and SAT runtime follow a similar trend: a decrease in the cost matches a decrease in SAT calls and runtime, which is exactly what we aim to achieve. Occasional differences occur due to effective SAT counter-examples that reduce the number of SAT calls in RevS more than in SimGen; this effect is accidental and orthogonal to the particular simulation strategies. The main takeaway is that SimGen either Pareto-dominates RevS by reducing cost, simulation runtime, SAT calls and SAT runtime simultaneously, or achieves Pareto-optimality by achieving lower cost at a runtime

Bmk	SAT calls		SAT time (ms)		Bmk	SAT calls		SAT time (ms)	
	RevS	SGen	RevS	SGen		RevS	SGen	RevS	SGen
alu4	138	136	2.3	2.4	table5	134	97	2.0	1.4
apex1	137	120	2.0	1.7	sin	124	134	4.5e3	2.5e3
apex2	226	204	4.7	4.3	square	54	27	3.0	1.6
apex3	184	173	2.5	2.3	arbiter	44	49	2.2	2.4
apex4	519	511	6.4	6.5	dec	194	200	1.8	1.8
apex5	78	67	1.3	1.0	m_ctrl	3739	3756	191.6	200.2
cordic	105	72	2.3	1.8	priority	83	79	3.1	3.3
cps	105	83	1.8	1.4	voter	1550	1522	2.1e3	2.7e3
dalu	45	42	1.3	1.1	log2	734	690	1.4e6	1.0e6
des	151	143	2.1	2.2	b14_C	159	102	11.2	6.7
e64	61	56	1.4	1.2	b14_C_2	156	101	10.5	8.4
ex1010	1157	1145	16.1	15.6	b15_C	1495	1395	107.2	93.7
ex5p	104	115	4.0	3.5	b15_C_2	1379	1370	118.3	98.2
i10	149	124	4.5	3.7	b17_C	4837	4741	258.2	215.0
k2	91	73	1.3	1.1	b17_C_2	4744	4469	270.2	260.3
misex3	210	203	3.9	3.9	b20_C	258	167	22.7	19.2
misex3c	50	35	1.0	0.8	b20_C_2	1227	305	72.5	36.0
pdc	989	976	20.9	18.8	b21_C	1369	271	66.6	22.9
seq	331	232	5.9	4.5	b21_C_2	1280	275	71.4	35.3
spla	807	776	13.5	13.5	b22_C	1762	1778	81.0	88.0
table3	148	108	2.2	1.5	b22_C_2	1808	1802	82.4	92.1

Bmk	SAT calls		SAT time (s)		Bmk	SAT calls		SAT time (s)	
	RevS	SGen	RevS	SGen		RevS	SGen	RevS	SGen
alu4 (15)	4872	4239	79.8	72.0	b17_C_2 (5)	10K	8748	27.9	24.9
square (7)	339	326	5.3	4.2	b20_C_2 (8)	3220	2828	26.2	15.9
arbiter (15)	13.9K	13.4K	93.2	67.7	b21_C_2 (8)	3544	3169	20.7	17.7
b15_C_2 (8)	4691	3989	20.7	18.7	b22_C (6)	2948	2494	55.9	43.9
b17_C (5)	9650	8298	48.3	29.5					

Table 2: *SAT calls* and *SAT time* (execution runtime in *ms*) of the SAT sweeping tool for the same 42 benchmarks as in Table 1. *RevS* and *SGen* refer to reverse simulation and SimGen. Generally, the improvement in the number of SAT calls translates into decreased SAT time; the magnitude of this runtime reduction depends on the verification tool's efficiency and the benchmark complexity. The lower table shows similar results when we increase the complexity of some of the benchmarks with ABC's "*&putontop*" command. The number in parenthesis represents the quantity of repeated graphs.

expense. Only rarely SimGen designs is Pareto-dominated by RevS, demonstrating SimGen's broad usefulness and effectiveness.

The experiments of this and the previous section demonstrate the superiority of SimGen over reverse simulation: it successfully improves cost and SAT sweeping runtime—sometimes, at a simulation runtime increase. In Section 6.5, we propose a practical way to combine SimGen with random simulation to exploit SimGen's benefits while also improving the simulation time.

6.4 SimGen's Scalability

The SAT runtime of many standard benchmarks we evaluated so far—and, consequently, the gains of SimGen—are small in absolute terms. We now explore whether SimGen's advantages hold for prolonged SAT times. To this end, we increase the complexity of our benchmarks as follows: for each benchmark, we put several copies of its network on top of each other by connecting the outputs of a bottom network to the inputs of the one on top of it. If there are more outputs than inputs, we create new POs; if there are more inputs than outputs, we create new PIs. We execute this operation by applying the command "*&putontop*" in ABC.

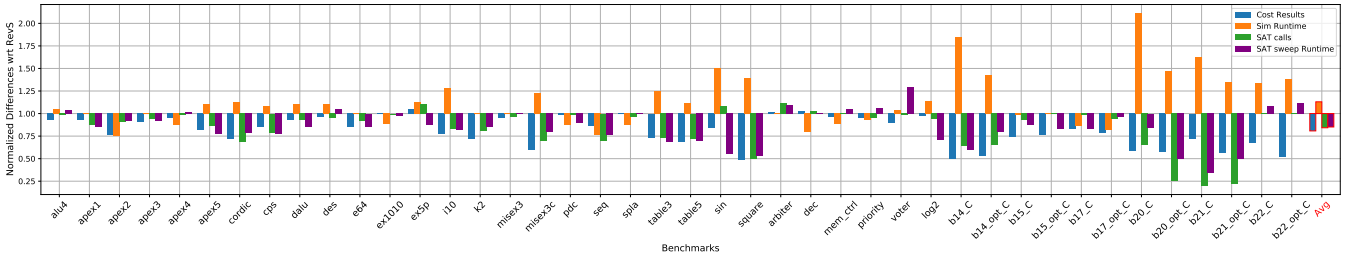


Figure 5: Normalized difference of cost, simulation runtime, SAT calls, and SAT runtime of SimGen with respect to reverse simulation. SimGen achieves significant improvements in cost, SAT calls, and SAT runtime, at an occasional simulation time penalty.

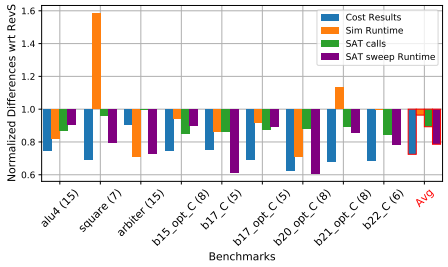
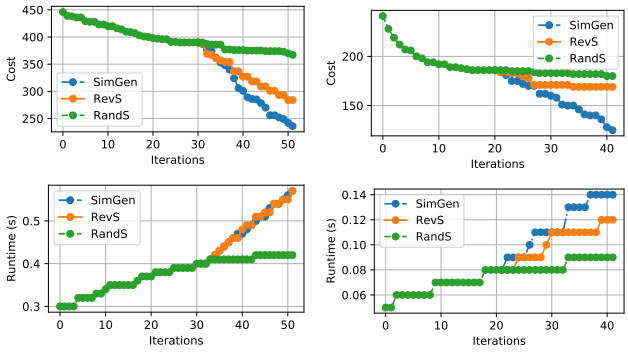


Figure 6: The same metrics as in Figure 5, plotted for more complex versions of our benchmarks. The results follow the same trends as before, indicating the effectiveness and scalability of SimGen.



(a) Benchmark *apex2* **(b) Benchmark *cps***

Figure 7: Runtime and cost at different iterations by random simulation (*RandS*) and a combination of random simulation with SimGen or RevS in *apex2* and *cps*. After a few iterations, RandS gets stuck in a local minimum and cannot improve the cost. Replacing it SimGen at that point achieves a lower cost at an increased runtime. This points to the usefulness of enhancing random simulation with SimGen.

In Table 2 and Figure 6, we evaluate and plot the same metrics as in Sections 6.2 and 6.3 for the modified benchmarks whose runtimes are greater than 1 s; each benchmark is annotated with the number of graph copies we use. The results follow the same trends as before: SimGen reduces the number of SAT calls and, consequently, the SAT runtime, at only an occasional simulation time penalty. This shows that SimGen is not only effective but also scalable.

6.5 Random Simulation and SimGen

As mentioned in Section 2, the first step of SAT sweeping is random simulation. Despite the speed of this technique [26], its lack of information about the circuit typically hinders efficient class separation. On the other hand, guided simulation strategies (such as reverse simulation and SimGen) can split equivalence classes by generating efficient input vectors but may suffer from prolonged runtimes, as Table 1 suggests. In this section, we contrast these techniques and show that their synergy—especially that of random simulation and SimGen—can successfully exploit their complementary benefits.

Figure 7 compares the cost and runtime of three types of simulation runs for two benchmarks, *apex2* and *cps*: (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. While SimGen Pareto-dominates RevS in apex2 (i.e., it achieves better cost at a lower execution time), in cps, SimGen compromises runtime for cost; this is in line with our findings in Table 1, indicating the continuous cost superiority of SimGen at an occasional runtime tradeoff. Overall, these results demonstrate that a synergy between random simulation and SimGen is desirable: the former achieves fast class division, and the latter is most effective in splitting classes when the former is stuck. This points to the relevance and need to incorporate SimGen into sweeping tool simulators.

7 CONCLUSION

Simulation is a fundamental step to accelerating CEC and SAT sweeping; yet, is it only effective if provided with appropriate input vectors to simulate. In this work, we propose SimGen, a framework for effective simulation input vector generation. SimGen borrows concepts from the ATPG domain 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 simluation): 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] . 2024. *"SimGen: Simulation Pattern Generation for Efficient Equivalence Checking" source code.* https://doi.org/10.5281/zenodo.11119715

[2] Luca Gaetano Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. The EPFL Combinational Benchmark Suite. In *Proceedings of the 24th International Workshop on Logic & Synthesis*. Mountain View, CA.

[3] L. Amarú, F. Marranghello, E. Testa, C. Casares, V. Possani, J. Luo, P. Vuillod, A. Mishchenko, and G. De Micheli. 2020. SAT-Sweeping Enhanced for Logic Synthesis. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, Virtual Event, USA, 1–6. https://doi.org/10.1109/DAC18072.2020.9218691

[4] D. Brand. 1993. Verification of Large Synthesized Designs. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design*. IEEE, Washington, DC, USA, 534–537.

[5] Robert Brayton, Alan Mishchenko, Leonardo De Moura, and Alexandre Piette. 2011. *ABC: A System for Sequential Synthesis and Verification.* EECS Dept. UC Berkeley. http://www.eecs.berkeley.edu/~alanmi/abc/

[6] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam. 2005. Reducing structural bias in technology mapping. In *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.* IEEE, USA, 519–526. https://doi.org/10.1109/ICCAD.2005.1560122

[7] F. Corno, M.S. Reorda, and G. Squillero. 2000. RT-level ITC'99 benchmarks and first ATPG results. *IEEE Design & Test of Computers* 17, 3 (2000), 44–53. https://doi.org/10.1109/54.867894

[8] H. Fujiwara. 1985. FAN: A Fanout-Oriented Test Pattern Generation Algorithm. In *Proceedings of the 1985 IEEE International Symposium on Circuit and Systems (ISCAS)*. 671–674.

[9] P. Goel and B. C. Rosales. 1981. PODEM-X: An Automatic Test Generation System for VLSI Logic Structures. In *18th Design Automation Conference*. Association for Computing Machinery, Nashville, TN, USA, 260–268.

[10] R. Hulle, P. Fiser, J. Schmidt, and J. Borecky. 2016. SAT-ATPG for application-oriented FPGA testing. In *2016 15th Biennal Baltic Electronics Conference (BEC)*. IEEE, Tallinn, Estonia, 83–86.

[11] S. Krishnaswamy, H. Ren, N. Modi, and R. Puri. 2009. DeltaSyn: An efficient logic difference optimizer for ECO synthesis. In *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*. IEEE, San Jose, CA, USA, 789–796. https://doi.org/10.1145/1687399.1687546

[12] A. Kuehlmann. 2004. Dynamic transition relation simplification for bounded property checking. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*. IEEE, San Jose, CA, 50–7.

[13] A. Kuehlmann and F. Krohm. 1997. Equivalence Checking Using Cuts And Heaps. In *Proceedings of the 34th Design Automation Conference*. IEEE, New York, NY, USA, 263–268. https://doi.org/10.1109/DAC.1997.597155

[14] A. Kuehlmann, V. Paruthi, F. Krohm, and M.K. Ganai. 2002. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21, 12 (2002), 1377–1394. https://doi.org/10.1109/TCAD.2002.804386

[15] R. P. Lajaunie and M. S. Hsiao. 2005. An Effective and Efficient ATPG-Based Combinational Equivalence Checker. In *Proceedings of the 15th ACM Great Lakes Symposium on VLSI*. Association for Computing Machinery, New York, NY, USA, 248–253.

[16] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli. 2022. A Simulation-Guided Paradigm for Logic Synthesis and Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 8 (2022), 2573–2586. https://doi.org/10.1109/TCAD.2021.3108704

[17] A. Lipowski and D. Lipowska. 2012. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications* 391, 6 (2012), 2193–2196. https://doi.org/10.1016/j.physa.2011.12.004

[18] Feng Lu, L.-C. Wang, Kwang-Ting Cheng, and R.C.-Y. Huang. 2003. A circuit SAT solver with signal correlation guided learning. In *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, USA, 892–897. https://doi.org/10.1109/DATE.2003.1253719

[19] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. 2006. Improvements to technology mapping for LUT-based FPGAs. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '06)*. Association for Computing Machinery, New York, NY, USA, 41–49. https://doi.org/10.1145/1117201.1117208

[20] A. Mishchenko, S. Chatterjee, R. K. Brayton, and N. Een. 2006. Improvements to Combinational Equivalence Checking. In *2006 IEEE/ACM International Conference on Computer Aided Design*. Association for Computing Machinery, New York, NY, USA, 836–843. https://doi.org/10.1109/ICCAD.2006.320087

[21] A. Mishchenko, J.S. Zhang, S. Sinha, J.R. Burch, R. Brayton, and M. Chrzanowska-Jeske. 2006. Using simulation and satisfiability to compute flexibilities in Boolean networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 5 (2006), 743–755. https://doi.org/10.1109/TCAD.2005.860955

[22] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jai Min Wang, Mohamed ElDafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. 2020. VTR 8: High Performance CAD and Customizable FPGA Architecture Modelling. *ACM Transactions on Reconfigurable Technology and Systems* 13, 2, Article 9 (Jun. 2020), 55 pages.

[23] Carmine Rizzi, Andrea Guerrieri, and Lana Josipović. 2023. An Iterative Method for Mapping-Aware Frequency Regulation in Dataflow Circuits. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, San Francisco, CA, 1–6. https://doi.org/10.1109/DAC56929.2023.10247686

[24] J. Roth. 1966. Diagnosis of Automata Failures: A Calculus and a Method. In *IBM Journal of Research and Development*. IBM Corp., USA, 278–291.

[25] M. Tan, S. Dai, U. Gupta, and Z. Zhang. 2015. Mapping-aware constrained scheduling for LUT-based FPGAs. In *Proceedings of the 23rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Association for Computing Machinery, Monterey, CA, 190–9.

[26] H.-T. Zhang, J.-H. R. Jiang, L. Amarú, A. Mishchenko, and R. Brayton. 2021. Deep Integration of Circuit Simulator and SAT Solver. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, San Francisco, CA, USA, 877–882. https://doi.org/10.1109/DAC18074.2021.9586331

[27] École polytechnique fédérale de Lausanne (EPFL). 2023. *EPFL Combinational Benchmark Suite.* École polytechnique fédérale de Lausanne (EPFL). https://www.epfl.ch/labs/lsi/page-102566-en-html/benchmarks/