This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

A Simulation-Guided Paradigm for Logic Synthesis and Verification

Siang-Yun Lee, Heinz Riener, Alan Mishchenko, *Senior Member, IEEE,* Robert K. Brayton, *Fellow, IEEE,* and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—This paper proposes a new logic synthesis and verification paradigm based on circuit simulation. In this paradigm, high-quality, expressive simulation patterns are pre-generated to be reused in multiple runs of optimization and verification algorithms, resulting in reduced time-consuming Boolean computations such as SAT-solving. Methods to generate expressive simulation patterns are presented and compared, and a bit-packing technique to compress them is integrated into the implementation. The generated patterns are shown to be reusable across different algorithms and after network function modifications. A logic synthesis algorithm, Boolean resubstitution, and a verification algorithm, combinational equivalence checking, are two examples of using this paradigm. In simulation-guided Boolean resubstitution, simulation patterns are used for efficient filtering of optimization choices, leading to a lower cost in expanding the search space. By adopting the proposed paradigm, we achieve a 5.9% reduction in the number of AIG nodes, compared to 3.7% by a state-of-the-art resubstitution algorithm, within comparable runtime. In simulation-guided equivalence checking, the number of SAT solver calls is reduced by 9.5% with the use of the expressive simulation patterns accumulated in earlier logic synthesis stages.**

***Index Terms*—Logic synthesis, formal verification, circuit simulation, Boolean methods, simulation patterns.**

I. INTRODUCTION

LOGIC synthesis and verification play an important role in *electronic design automation* (EDA), and extensive research has been done on optimizing logic networks since the emergence of this field. The numerous logic optimization methods existing in the literature [1], [2] can be roughly classified into two classes, namely algebraic methods, which treat Boolean functions as polynomials and optimize the logic network locally, and Boolean methods, which exploit global Boolean logic and *don't-cares* to improve the optimization quality. As the size and complexity of digital circuits grow, there is often a trade-off between efficiency and quality. Algebraic methods, as well as other local-search methods such as structural analysis and window simulation, are efficient but often sacrifice optimality. In contrast, Boolean methods, such as Boolean decomposition [3], resynthesis [4] and rewriting [5], usually achieve better quality at the cost of solving NP-hard Boolean problems using a *binary decision diagram* (BDD) [6], [7] package in earlier research, or a *satisfiability* (SAT) problem [8], [9] solver in more recent literature.

To balance between the two extremes, circuit simulation is often used in Boolean methods as an efficient approximator of the Boolean functions embedded in logic networks. However, if the simulation is not exhaustive, formal verification, which is usually done with SAT-solving, is still required [10]. In this paper, we introduce a new paradigm, *simulation-guided logic synthesis and verification*, where efforts are made in pre-generating a set of high-quality, *expressive* simulation patterns to be reused many times. By increasing the expressive power of the simulation patterns, synthesis and verification algorithms become more efficient, and the extension of the search space in optimization algorithms becomes more affordable. The underlying hypothesis, which is confirmed by experimental results, is that expressive simulation patterns can be amassed for a logic network and used later as an efficient filter to avoid unnecessary SAT solver calls.

The proposed paradigm is useful for algorithms dominated by expensive Boolean computations. Two representative applications are presented in this paper: *Boolean resubstitution* [11] and *combinational equivalence checking* (CEC) [12].

The first representative application is to demonstrate a high-quality and efficient Boolean resubstitution framework based on the simulation-guided paradigm. The classic resubstitution algorithm iterates over the nodes in a logic network and attempts to re-express their functions using other nodes in the network. If updating a node's function makes other nodes in its fan-in cone dangling (i.e., having no fan-out), they can be deleted, resulting in the reduction of the network's size. For the special case of replacing a node directly with an existing node, it is equivalent to *functional reduction* (FRAIG) [13]. In the presented simulation-guided resubstitution framework, nodes fed into the resubstitution engine are represented by their simulation signatures, and a SAT solver is used to validate the computed resubstitution candidates. Using expressive simulation patterns, most illegal candidates can be quickly identified and ruled out within the engine by simply comparing simulation signatures and without the need for SAT-based validation. The experimental results show that simulation-guided resubstitution allows user-specified tuning of the efficiency-quality trade-off and improves optimization quality by considering a larger search space while maintaining reasonable efficiency. Comparing to a state-of-the-art *And-Inverter Graph* (AIG) resubstitution algorithm [11], average reduction in the number of AIG nodes improves from 3.65%

This research was supported in part by the EPFL Open Science Fund, by the SRC Contract 2867.001, and by the SNF grant "Supercool: Design methods and tools for superconducting electronics", 200021_1920981.

S.-Y. Lee, H. Riener and G. De Micheli are with the Integrated Systems Laboratory, Swiss Federal Institute of Technology Lausanne, 1015 Lausanne, Switzerland.

A. Mishchenko and R. K. Brayton are with the Department of EECS, UC Berkeley, Berkeley, California, USA.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

to 5.90%.

The second representative application shows that the simulation patterns can be used in CEC. Similarly, simulation-guided CEC leverages the expressive patterns generated in earlier synthesis stages to disprove more non-equivalent nodes than random simulation can do, thus reducing the effort needed in SAT-based formal verification. In our experiment, a 9.5% reduction in the number of SAT calls is achieved when expressive patterns are used in CEC.

This motivates us to study what makes simulation patterns expressive and profile different pattern generation strategies, including random simulation, stuck-at-value testing [14], observability checking [15], and combinations of these. In the process of resubstitution and CEC, pre-computed simulation patterns can be refined further with the counter-examples generated by SAT-solving. The generated patterns and the supplemented counter-examples can be reused in two schemes: across different algorithms, such as resubstitution followed by CEC, and across different versions of the same design. Reusability in the latter case is verified with experiments on *engineering change order* (ECO) [16] benchmarks, which are similar networks with functional modifications.

The contributions of the paper are: (1) a simulation-guided logic synthesis and verification paradigm, which pre-generates and reuses expressive simulation patterns to reduce the efforts needed in SAT-based verification; (2) methods to generate expressive simulation patterns, which are integrated with a bit-packing technique; (3) demonstrations of the benefits of the proposed paradigm with improved resubstitution quality and reduced SAT calls in CEC; (4) the reusability of the pre-generated patterns across different applications and with network modifications, shown with experimental results.

The rest of this paper is organized as follows: After preliminaries are given in Section II and related works introduced in Section III, we first describe the simulation-guided paradigm in Section IV. Then, pattern generation and compaction methods are explained in Section V. Two applications, Boolean resubstitution and CEC, are demonstrated in Sections VI and VII, respectively. Finally, experimental results are given in Section VIII, and conclusions in Section IX.

II. Preliminaries

A. Logic Networks

In this paper, we focus on technology-independent representations of digital circuits, referred to as *logic networks* (or simply *networks*). Logic networks are *directed acyclic graphs* (DAGs), where nodes represent logic gates and edges represent wires connecting them together. Incoming edges of a node are called *fan-ins*, whereas outgoing edges are called *fan-outs*. The *transitive fan-in* (TFI) or the *transitive fan-out* (TFO) of a node n is the set of nodes such that there is a path between n and these nodes in the direction of fan-in or fan-out, respectively. A logic gate computes a *Boolean function*, which is a function defined over the Boolean space $\mathbb{B} = \{0, 1\}$, of its fan-ins and passes the resulting output value to its fan-outs. Concatenating the computation of the logic gates according to the structure of a network, the global Boolean functions of each node can be derived, which take *primary inputs* (PIs) as inputs. Two nodes in a network are said to be *functionally equivalent* if their global functions are logically equivalent; otherwise, they are *functionally non-equivalent*. Overall, a logic network realizes Boolean functions of the *primary output* (PO) nodes. The size of a network is determined by its number of nodes.

In this paper, we work with AIGs [17], where every node is an AND gate and the inverters are represented by edges with a complement attribute and with no cost (that is, they do not add to the network size). Nevertheless, this paradigm can also be applied to other types of homogeneous logic networks, such as Majority-Inverter Graphs [18], Xor-And-Inverter Graphs [19], and Xor-Majority Graphs [20], as well as mapped networks such as k-LUT networks [21].

B. Don't-Cares

Boolean methods usually achieve better optimization quality than algebraic methods because they consider the flexibilities of the network, called *don't-cares*. The don't-care set in a logic network indicates where local functions can be modified without changing the global functions, which can be leveraged to optimize the network. There are two types of don't-cares:

1) For a set of internal nodes, there might be some value combinations that never appear at these nodes. For example, an AND gate g_1 and an OR gate g_2 sharing the same fan-ins can never have $g_1 = 1$ and $g_2 = 0$ at the same time. This combination is a *satisfiability don't-care* (SDC) of a common TFO node of g_1 and g_2.
2) A value assignment $\vec{x} \in \mathbb{B}^n$ to the PIs is said to be *unobservable* with respect to a node n if none of the POs changes its value when n is replaced by its negation $\overline{n}$. $\vec{x}$ is an *observability don't-care* (ODC) of n because the function of n under $\vec{x}$ does not matter.

C. Boolean Satisfiability Problem

Boolean optimization methods are often formulated as a Boolean satisfiability problem and solved with a SAT solver [9]. A SAT problem is asking whether a Boolean formula, usually presented in a *conjunctive normal form* (CNF) as a conjunction of *clauses*, is satisfiable. That is, whether there exists a value assignment making the formula evaluate to true. If so, the solver returns a *satisfiable* (SAT) result along with a satisfying value assignment; otherwise, it concludes that the problem is *unsatisfiable* (UNSAT). Logic networks can be translated into CNF formulae with the Tseytin transformation [22].

By using SAT in logic optimization, we benefit from its global consideration of the Boolean functions and hence better optimization quality. However, SAT is an NP-complete problem [23]. Although many approaches have been proposed to solve SAT problems efficiently for EDA applications [9] and efficient SAT solvers have been developed, SAT-solving is still slower than algebraic and local-search methods in general. In practice, to avoid the program being stuck in a difficult SAT solve, a *timeout* can be set to limit the time spent in solving SAT; and/or a *conflict limit* can be set to restrict the effort made by the SAT solver.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

D. Windowing

A *window* is a sub-graph constructed from a *root* node r and a *cut* $C = (r, L)$, which is a pair of the root node and a set of *leaf* nodes L. The set of leaf nodes fulfills the requirement that any path from a PI to r passes through exactly one node in L. All the internal nodes on the paths from any node in L to r are included in the window. Additionally, nodes outside of the TFI cone of r but having all of their fan-ins in the window can also be added into the window. A window can be viewed as a smaller network with the leaf nodes as PIs and the root node as the PO.

E. Circuit Simulation

A *simulation pattern* (or abbreviated as a *pattern*) is a collection of Boolean values assigned to each primary input of a network. Circuit simulation is done by visiting nodes in a topological order and computing their output values with their input values. In practice, several simulation patterns can be bundled together by using machine words, instead of a single bit, to represent a sequence of Boolean values. This way, 32 or 64 patterns can be computed for a node within a single CPU instruction using bitwise logical operations supported by modern arithmetic logic units. The *simulation signature* of a node is an ordered set of values produced at the node under each simulation pattern.

A set of simulation patterns is *exhaustive* if it covers all possible combinations of value assignment, which requires 2^k patterns for k PIs. The simulation signatures produced by simulating an exhaustive pattern set are also called *truth tables* and they completely specify the Boolean functions of the nodes.

Simulation can be done globally in the entire network or locally in a small window. In the former case, the simulation pattern set is possibly non-exhaustive because 2^{16} patterns are already impractical to handle, but the number of PIs is usually larger than 16. To use an exhaustive set of patterns, simulation must be restricted to a window of less than 16 (typically 8 to 10) leaf nodes.

F. Resubstitution

Boolean resubstitution is one of the combinational optimization methods aiming at reducing sizes of logic networks. For each node in a network, called the *root*, the algorithm tries to find a smaller replacement for the sub-graph that only contributes to the root, called the *maximum fan-out free cone* (MFFC) [24]. A node n is said to be in the MFFC of the root node r if n is in the TFI of r and all paths from n to the primary outputs pass through r. The MFFC of a node can be efficiently computed by recursively referencing and dereferencing nodes in the network. If the root node is replaced and deleted, all nodes in its MFFC can also be deleted, reducing the size of the network.

The replacement for the root node, called the *dependency circuit*, is built upon a set of potentially useful nodes existing in the network, called *divisors*. A divisor should not be in the TFO cone of the root, otherwise the resulting network would be cyclic. It should also not be in the MFFC because nodes in the MFFC are to be removed after resubstitution. Nodes depending on primary inputs that are not in the TFI of the root node can also be filtered out from the set of divisors because their functions are unrelated to that of the root node. In practice, to keep the runtime reasonable, a priority is given to nodes in a window composed of the TFI cone of the root with maximum support size K and nodes outside of the TFI depending entirely on other divisors. [11]

A *resubstitution candidate* (also abbreviated as a *candidate*) is either a divisor itself or a single-output function, named the *dependency function*, built with several divisors. In the latter case, the candidate is represented by the top-most node of the dependency circuit. A *resubstitution*, or simply *substitution*, is a pair (r, c) of a root node r and a resubstitution candidate c, and it is said to be *legal* if replacing r with c does not change the functions of any PO. Otherwise, the resubstitution is said to be *illegal*.

III. Related Work

Random simulation is a core tool in logic synthesis and verification, which has been used successfully to reduce the runtime of various computations. In this section, we first review some works leveraging the power of random simulation. Then, with Boolean resubstitution being our main example algorithm adopting the simulation-guided paradigm, some existing resubstitution techniques are also described.

In functional reduction [13], random and guided simulation are used to identify equivalent nodes and merge them. In combinational equivalence checking [12], simulation is also used to find cut-points between two networks that serve as stepping stones for the proof of equivalence at the primary outputs. In [10], [25], a combination of random simulation and SAT solving was proposed to compute flexibilities (don't-cares) of Boolean networks within a window, and to compute the dependency function in resubstitution. Motivated by the efficacy of these techniques adopting *random* simulation, the simulation-guided paradigm in this paper focuses on identifying a set of *expressive* simulation patterns to further strengthen the power of simulation. Once identified, the patterns can be reused multiple times to speed up logic synthesis and verification for the same or a similar network in various applications.

Research in Boolean resubstitution techniques dates back to the 1990s [26], [27]. In the 2000s, efforts were made to improve the scalability of BDD-based computations [28] and to move away from BDDs to simulation and SAT solving [4], [10]. In [10], the dependency function is computed by enumerating its onset and offset cubes using SAT and interpolation [29], where random simulation is used for the initial filtering of potentially useful divisors. In [4], structural analysis (windowing) was introduced to speed up the algorithm further. Windowing is used to limit the search space and the SAT instance size, with the inner window as a working space, and the outer window as the scope for computing don't-cares.

An efficient Boolean resubstitution algorithm for AIGs using windowing was presented in [11], which is considered as the state-of-the-art to be compared to in this paper. It relies

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

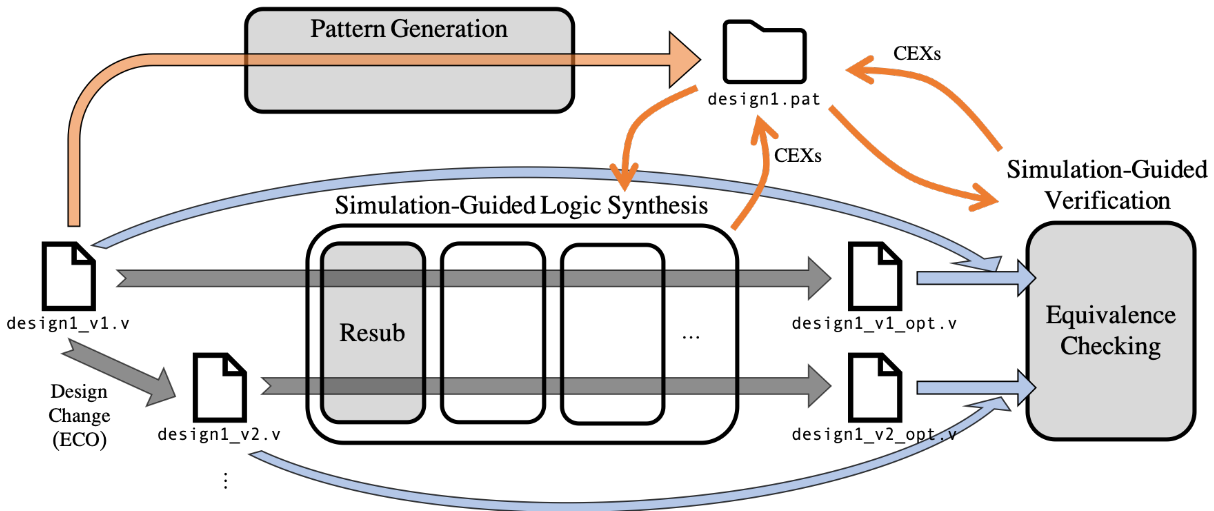

Fig. 1. The simulation-guided logic synthesis and verification paradigm. For each design (named `design1` in the figure), a set of expressive simulation patterns is generated once (`design1.pat`) and is used several times throughout logic synthesis and verification. The same pattern set is applicable for various versions of the design with functional modifications (`design1_v1`, `design1_v2`, etc.). When the pattern set is used in one of the simulation-guided algorithms, it is also supplemented and refined with the counter-examples (CEXs) generated as side-products during execution of the algorithm. The blocks shaded in grey are implemented and described in this paper: pattern generation in Section V, resubstitution in Section VI, and equivalence checking in Section VII. While other logic synthesis algorithms may also benefit from adopting the paradigm (the blank blocks in the figure), we present only resubstitution as an example in this paper.

entirely on truth table computation, without any use of BDDs or SAT. The search for divisors is limited to a window near the root node, which is constructed from a size-limited cut to allow exhaustive simulation. The node functions in the window are expressed in terms of the cut nodes. The dependency function is not computed as a separate step after minimizing its support, as in [4]. Instead, simple dependency circuits of up to three AND gates are explicitly tried for resubstitution using several heuristic filters. This windowing-based and truth-table-based resubstitution framework has been generalized for many different gate types including majority gates [30] and complex gates [31].

IV. The Simulation-Guided Paradigm

This paper introduces a new paradigm for logic synthesis and verification that exploits fast bit-parallel simulation to reduce the number of expensive NP-hard equivalence checks based on SAT. The rationale behind the idea is to pre-compute a set of simulation patterns for a given logic network, which can efficiently rule out most non-equivalences by simply comparing simulation signatures. Motivated by the fact that detecting and verifying functional equivalence are the major tasks in many logic optimization (especially Boolean methods) and verification algorithms, we define *expressive* simulation patterns as follows.

Definition: A non-exhaustive set of simulation patterns for a logic network is said to be *expressive* if the simulation signatures obtained by simulating the patterns can be used to pair-wisely distinguish functionally non-equivalent nodes that either already exist in the logic network or can be derived from some existing nodes.

The exhaustive set of simulation patterns satisfies the latter part of this definition, but this is typically too large for logic networks with 16 or more primary inputs. In practice, only expressive simulation patterns that can be efficiently stored and simulated using less than, say, a few hundred or thousand bits are of interest.

We assume that, for a given logic network of interest, a set of expressive simulation patterns with size proportional to the network size can be found. This means that, as depicted in Figure 1, the expressive simulation patterns can be pre-computed, stored, and reused by different logic synthesis or verification algorithms when applied to the same network, or by the same algorithm when invoked multiple times with slightly different networks. The assumption is verified with experimental results in Section VIII by showing pattern reusability after ECOs, which are typically small functional modifications to networks under design [16]. With this assumption, we claim that the time needed to generate the expressive patterns is not critical because they will be reused many times such that the benefits are more substantial.

Expressive simulation patterns cannot be derived directly from the Boolean functions of the primary outputs, but must account for some structural information of the network. An intuitive explanation of this observation is that a PO function can be implemented by a large number of structurally different logic networks. Despite this, the idea of reusing simulation patterns in multiple optimization or verification runs is still valid because the initial structure of the network often is determined by high-level synthesis and later carefully fine-tuned by logic optimization. Consequently, only a small fraction of closely-related structures is encountered during logic optimization and the final verification of the network. Several pattern generation strategies are discussed in Section V.

The proposed simulation-guided paradigm can be adopted by algorithms dealing with the Boolean relation among nodes in logic networks. For example, in Section VI, the paradigm

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

is demonstrated with Boolean resubstitution, where simulation signatures are used as an approximation of node functions when finding resubstitution candidates. This way, restriction to local windows is avoided and global information is utilized with low cost. In Section VII, benefits of the expressive patterns in CEC are demonstrated as another application of this paradigm. As simulation patterns are already generated for the optimization algorithms prior to verification, reusing them in CEC comes at no extra cost. With their stronger ability to distinguish non-equivalent nodes without SAT solving, the overall number of SAT calls in CEC can be reduced. The paradigm is potentially suitable for other algorithms, such as computation of structural choices [32], to improve the quality of mapping and gate matching between several versions of the same logic network. Furthermore, the resulting patterns can also be used in *automatic test pattern generation* (ATPG) [33] and in circuit reliability analysis [34].

To conclude, simulation signatures are used as efficient approximations of node functions to reduce NP-hard equivalence checks. As they may not cover all circuit states under all possible input assignments, formal verification (in this paper, by SAT-solving) is inevitable in simulation-guided algorithms, which generates counter-examples in terms of PI value assignments, i.e., new simulation patterns. To reduce unnecessary SAT-solving, we seek to increase the accuracy of this approximation. On one hand, we propose to pre-generate an expressive pattern set to be reused across multiple optimization runs and across different algorithms, and we study methods to ensure good quality of these patterns in the first place. On the other hand, motivated by the success of various counter-example-guided logic synthesis and verification works [10], [13], [35], [36], we propose to collect and keep the counter-examples generated by different algorithms and use them to enhance the initial pattern set.

V. Simulation Pattern Generation

Following the previous section, several strategies to generate expressive simulation patterns are formulated in this section. Two types of patterns are used as the basis: *random patterns* which are random values generated with equal probability of 0 or 1 for each primary input, and *stuck-at patterns* which are generated by trying to distinguish each node from constant functions 0 and 1. Generating random patterns is straightforward. The procedure to generate stuck-at patterns is described in Section V-A. Then, in Section V-B, an observability-based method to strengthen stuck-at patterns is elaborated. Finally, a bit-packing method to compress the pattern set is explained in Section V-C.

A. Stuck-at Values

In random simulation, the possibility of a certain bit value (0 or 1) appearing in the simulation signature of some nodes in the network may be relatively low. For example, a 2-input AND gate only produces 1 when both of its fan-ins are 1, which is of 25% possibility if the fan-in values are randomly assigned. However, a value of 1 at this node may be necessary for disproving some non-equivalence. Thus we refine the set

StuckAtCheck
input: a logic network N
output: a set S of expressive simulation patterns
01 S := a small set of random patterns
02 N*.simulate*(S)
03 **initialize** *Solver*
04 *Solver.generate_CNF*(N)
05 **foreach** node n in N **do**
06   **if** n.signature $= \vec{0}$ **or** n.signature $= \vec{1}$ **do**
07   **if** n.signature $= \vec{0}$ **do**
08   *Solver.add_assumption*(n)
09   **else do**
10   *Solver.add_assumption*($\neg n$)
11   *result* := *Solver.solve*()
12   **if** *result* = SAT **do**
13             $S := S \cup \{Solver.pi_values\}$
14   **else if** *result* = UNSAT **do**
15   Replace n with constant node.
16 **return** S

Fig. 2. Algorithm *StuckAtCheck*: Generation of expressive simulation pattern by asserting stuck-at values.

of simulation patterns by checking that every node has both values appearing in its simulation signature. If only one value occurs, a new simulation pattern is created by solving a SAT problem, which forces the node to have the other value.

The algorithm, named *StuckAtCheck*, is illustrated in Figure 2. In lines `01-02`, we start with a small set of random simulation patterns and simulate the network to get the initial simulation signatures of each node. A SAT solver is also initialized and loaded with the CNF clauses translated from the network in lines `03-04`. Then, in line `05`, for each node in the network, if 0 or 1 does not appear, we try to generate a pattern by assuming the missing value and solving the SAT instance (lines `06-11`). If the solver finds a satisfying assignment, the desired pattern is generated (lines `12-13`). In an un-optimized network, there may be nodes which never take one of the values and the solver will conclude that the problem is unsatisfiable (line `14`). These nodes can be replaced by a constant node in line `15`. If the solver times-out or a given conflict limit is exceeded, we simply skip the node and continue the process with the next node.

The pattern set can be further strengthened by assuring both values appear multiple times (for example, at least 10 times) in the signature of every node. This can be done by running the SAT solver multiple times while making sure it takes different computation paths.

An example is shown in Figure 3. Suppose there are two random patterns in the initial set $S = \{\texttt{000}, \texttt{110}\}$. After simulation, the simulation signature obtained for node n is `00` where 1 does not appear. Hence, by assuming $n = 1$ and solving SAT a new pattern `011` is generated and added to the end of S. Now the simulation signature of n is `001`.

B. Observability

As described in Section II-B, there may be some simulation patterns that are not observable with respect to an internal

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

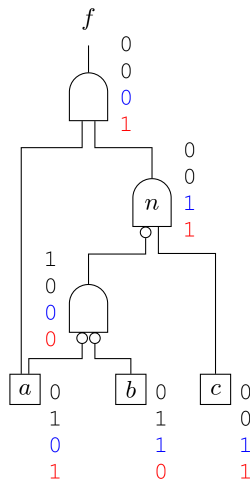

Fig. 3. Example network for pattern generation methods. A simulation pattern is a value assignment to $\vec{x} = (a, b, c)$. Suppose initially, two patterns 000 and 110 are generated randomly. Node n is stuck at 0 because its output value is 0 under both patterns. *StuckAtCheck* thus generates a pattern 011 to produce a 1 at n. However, this pattern is not observable because flipping the value of n from 1 to 0 does not affect the PO f, as $a = 0$ keeps the PO value at 0. Hence, another pattern 101 is generated by *ObservablePatternGeneration* to ensure that the pattern is observable and makes $n = 1$.

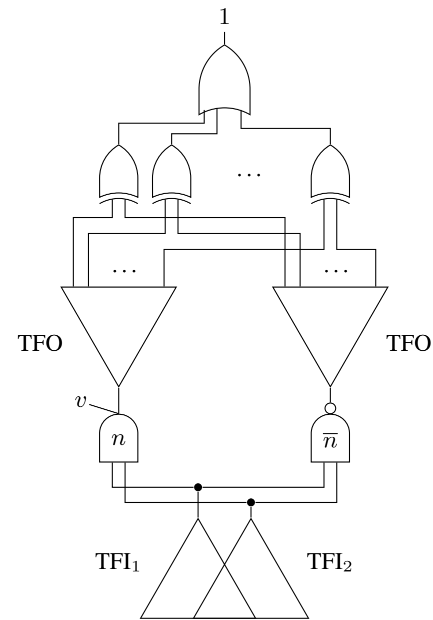

Fig. 4. Corresponding network of the CNF instance built in procedure *ObservablePatternGeneration*. The lower two triangles TFI$_1$ and TFI$_2$ are the TFI cones of the two fan-ins of node n. $\overline{n}$ is created and connected to the same TFI cones as n. The TFO cone of n is duplicated (the upper two triangles) and the counterpart is connected to $\overline{n}$. Primary outputs in the two TFO cones are matched and connected to XOR gates, and the XOR gates are fed to an OR gate, whose output is asserted to be 1, forming a miter sub-network. The output value of node n is asserted to be v.

node; these patterns are possibly less useful in disproving non-equivalence. Here, two cases are identified where a generation or re-generation of an observable pattern may be done:

- Case 1: In *StuckAtCheck* when a node is stuck at a value, and a new pattern is generated to express the other value, but this pattern is not observable.
- Case 2: A node assumes both values, but for all the patterns under which the node assumes one of the values, it is not observable.

The first case is identified during *StuckAtCheck*. Whenever a new pattern is generated (line 13), its observability with respect to the node n is checked according to the definition in Section II-B using the following steps: (1) Simulate the network to obtain the PO values under this pattern. (2) Flip the simulation value at the output of n and simulate its TFO cone again. (3) Check if all of the PO values remain the same. If so, the pattern is un-observable. (4) Restore the value of n and simulate again.

The second case is checked after procedure *StuckAtCheck* is completed. We iterate over all the nodes in the network again and check if for each node, there is at least two patterns which are observable with respect to the node and the node assumes 0 and 1 respectively under the two patterns. The procedure to check whether each pattern is observable is the same as described above.

To *resolve* un-observable patterns, a procedure *ObservablePatternGeneration* is devised, which generates an observable simulation pattern $\vec{x}$ with respect to a given node n and makes sure that n expresses a specified value v under $\vec{x}$. This procedure builds a CNF instance, whose corresponding network is shown in Figure 4, and solves it using the SAT solver. If the instance is SAT, an observable pattern is generated (Claim 1), and we say that the originally un-observable pattern is *resolved*. Otherwise, if the solver returns UNSAT, n is found to be un-observable with value v and can be replaced by the constant node in the respective polarity (Claim 2).

Claim 1: A satisfying input assignment $\vec{x}$ in the network of Figure 4 is an observable pattern with respect to node n.

Proof. By the definition in Section II-B, $\vec{x}$ is observable with respect to n if the value of at least one of the primary outputs of the network under $\vec{x}$ is different when n is replaced by $\overline{n}$. This condition is ensured by the miter of the TFO cones of n and $\overline{n}$ in Figure 4. ∎

Claim 2: If a node n is never observable with value v ($v \in \{0, 1\}$), then it can be replaced by constant $\neg v$ ($\neg 0 = 1, \neg 1 = 0$) without changing the network function(s). That is, there does not exist a primary input assignment $\vec{x}$, such that one of the primary outputs has different values in the original network and in the modified network.

Proof. Assume the opposite: there exists a primary input assignment $\vec{x}$, such that at least one of the primary outputs has a different value after replacing n with $\neg v$. If the value of n is $\neg v$ under $\vec{x}$, all node values in the network, including primary outputs, remain unchanged if n is replaced by $\neg v$. If the value of n is v under $\vec{x}$, because n is not observable with v, all primary outputs remain at the same value when the node value of n changes to $\overline{n} = \neg v$, which contradicts the assumption. ∎

In order to limit the computation in large networks, the TFO in Figure 4 can be restricted to nodes within a certain distance from n, called the *depth* of the TFO cone, instead of extending all the way to primary outputs. In this case, all the leaves of the cone should be XOR-ed with their counterparts to build the miter. Note that restricting the TFO depth weakens

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

the definition of observability, but is essential for scalability. Empirically, using a depth of 5 logic levels is a good tradeoff between quality and runtime.

After an observable pattern $\vec{x}$ is generated, in Case 1, we can replace the pattern generated by *StuckAtCheck* with $\vec{x}$. In Case 2, we simply add $\vec{x}$ to the set of patterns.

We continue with the example in Figure 3 with three patterns in the set $S = \{\texttt{000}, \texttt{110}, \texttt{011}\}$. By checking the observability of each pattern, it is found that only `110` is observable and the value of n under this pattern is 0. Hence, procedure *ObservablePatternGeneration* generates another pattern `101` making $n = 1$. This pattern is indeed observable because flipping the value of n from 1 to 0 also makes the PO value f change from 1 to 0.

C. Bit-Packing

For some large benchmarks with many primary inputs, the size of the generated pattern set can be large, slowing down simulation. In the field of ATPG, test patterns are often compressed by first identifying *care* and *don't-care* bits in them [37]. The set of care bits in a test pattern is the set of PI values that contribute to detecting a certain fault, while the don't-care bits are the PIs that can be assigned to any value. We integrated a similar technique in our simulation pattern generation.

Similar to test pattern compression, the care bits in a simulation pattern are the PI values that contribute to proving that the node is not stuck-at and in fact observable at one of the outputs. During simulation pattern generation with the previously described methods, care bits are identified by a simple structural support analysis, which highlights control paths from the inputs to the target node, and from the target node to at least one output where it is observed.

After generating several patterns, the pattern set is compressed by trying to pack each new pattern into one of the preceding patterns. Two patterns can be packed together if their care bits do not overlap. To pack a pattern p_1 into another pattern p_2, the care bits of p_1 are written into don't-care bits of p_2, and these bits are marked as cares in p_2.

D. Discussion

In this section, we illustrate methods to derive an initial set of expressive patterns serving as the basis of the simulation-guided paradigm. Starting from a mixture of random patterns and stuck-at patterns as the basis and depending on the computation effort taken by the pattern generation phase, observability checks can be applied to strengthen or append the pattern set. It may seem, from the algorithms, that each pattern is generated for a specific node in the network, which may be removed later during logic optimization and the pattern becomes useless. However, we argue that this is not a problem because even random patterns play an important role in this paradigm, as shown in our experimental results. Moreover, it is practically inefficient to keep track of which pattern is generated for which node and which patterns are still useful, especially after bit-packing. As another evidence, our experimental results on ECO benchmarks show that the generated patterns are as useful for a functionally modified network even if they are generated with the original version of the design.

VI. Simulation-Guided Resubstitution

In this section, the simulation-guided paradigm is demonstrated with Boolean resubstitution as an example application in logic synthesis. The main difference of our algorithm, compared to a state-of-the-art resubstitution algorithm [11], is in the representation of the divisors. Instead of using the complete truth table of the *local* function of the node, we use the simulation signature approximating the *global* function of the node. The algorithm consists of the following steps:

1) Generation of a set of expressive simulation patterns, as described in Section V.
2) Simulation of the network with these patterns to obtain simulation signatures for each node.
3) Iterating over all nodes in the network and calling the currently chosen node the root node. Estimating the gain by computing the root node's MFFC and collecting the divisors. Skipping the node if the gain is too small or if there are no divisors. This step is the same as in [11], so we omit the details here.
4) Searching for resubstitution candidates in terms of dependency functions using simulation signatures.
5) Validating the resubstitution with SAT solving by assuming non-equivalence. An UNSAT result validates the resubstitution, while a SAT result provides an input assignment under which the optimized network is not equivalent to the original network. In the latter case, the counter-example is added to the set of simulation patterns.
6) Iterating starting from Step 3, until all nodes in the network have been processed.

Simulation of the entire network in Step 2 enables better incorporation of global satisfiability don't cares without extra cost, which allows more optimization potential comparing to the windowing-based approach as in [11]. Collection of counter-examples in Step 5 expands the simulation pattern set, which further improves the efficiency of later optimization runs. In the remainder of this section, we focus on Steps 4 and 5, shown in Figure 5, which differ the most.

A SAT solver is initialized and the CNF clauses encoding gate logic are generated and added to the solver in lines `01-02`. In line `04`, a simulation-signature-based dependency function computation algorithm is used to find a dependency circuit of up to N^* nodes, where N^* is the smaller value among a user-specified parameter N and the size of the MFFC. Procedure *compute_function* heuristically searches for a minimum-node AIG implementation F of the target function f_t using a set of divisors D as PIs. Both the target function and the divisors are represented by their simulation signatures. The PO of F has the same signature as the given target f_t. The divisors are classified as either *unate* or *binate* by the implication relationship of their signatures f_d with the target function f_t. If either $f_d \to f_t$ or $f_t \to f_d$ holds, the divisor d is said to be unate. Since inverters are for free in AIGs,

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

SimResub
input: a root node n in a simulated network N,
its MFFC *MFFC*, and a set D of divisors
output: a legal (verified) candidate to substitute n, if exists
01 **initialize** *Solver*
02 *Solver.generate_CNF*(N)
03 **while** TRUE
04 F := *compute_function*(n, D, $min\{|MFFC|, N\}$)
05 **if** $F \neq$ **NULL do**
06 *result* := *Solver.verify*(n, F)
07 **if** *result* $=$ TRUE **do**
08 **return** F
09 **else if** *result* $=$ FALSE **do**
10 N.*re_simulate*()
11 **else break**
12 **else break**
13 **return NULL**

Solver.verify
input: a root node n in a simulated network N, and
a dependency circuit F with some nodes in N as PIs
and F_{out} as PO
output: whether it is legal to substitute n with F
14 *Solver.generate_CNF*(F)
15 *Solver.add_assumption*(*literal*(n) $\oplus$ *literal*(F_{out}))
16 *result* := *Solver.solve*()
17 **if** *result* $=$ UNSAT **do**
18 **return** TRUE
19 **else if** *result* $=$ SAT **do**
20 N.*add_pattern*(*Solver.pi_values*)
21 **return** FALSE
22 **return** UNKNOWN

Fig. 5. Algorithm *SimResub*: One iteration of Steps 4 and 5 in simulation-guided Boolean resubstitution.

the complement $\overline{d}$ of a divisor d is also considered, separately from d. If neither d nor $\overline{d}$ is unate, d is said to be binate. First, it is checked if the function can be implemented by a constant node or if one divisor can implement it in the direct or complemented polarity, both meaning that no gate insertion is needed to express the function. Next, it is checked if the function or its complement can be implemented with an AND gate, leading to a single-node dependency circuit. Then, if there are some unate divisors, the function or its complement is implemented using an AND gate whose one input is a unate divisor d and the other input is an incompletely-specified remainder function f_r satisfying $f_r \wedge f_d = f_t$. The unate divisor covering the most of the onset (or offset) minterms of f_t is selected first, and the implementation of f_r is computed by calling *compute_function* recursively.

Since the simulation signatures are an approximation of the node's function, the resubstitution candidate needs to be formally verified. Procedure *verify* in line `06` uses the SAT solver to try to find a pattern, under which nodes n and F_{out} have different values. The resubstitution is legal if the solver returns UNSAT (lines `17-18`); otherwise, a new pattern is added to the set and the network is re-simulated if the solver returns SAT (lines `19-21` and `09-10`). Note that if the simulation signatures are stored as sequences of multiple machine words, a new pattern is appended to the end of the last word and only this word needs to be re-computed because the other words remain the same. With the appended signatures, *compute_function* gives a different result in the next invocation. The process continues until one resubstitution is validated (lines `07-08`), or the SAT solver times-out (lines `22` and `11`), or until the engine cannot find another candidate dependency function (line `12`).

VII. SIMULATION-GUIDED EQUIVALENCE CHECKING

CEC after logic synthesis can benefit from the simulation information collected and used for logic optimization. This is because, in the process of CEC [12], one of the major tasks is disproving candidate equivalences, which relies on SAT-solving when counter-examples cannot be easily found with random simulation. The pre-computed expressive simulation patterns provided to the CEC engine can be used to disprove many of the non-equivalent nodes directly without any SAT-solving.

The command `&cec` in ABC[1] [38], which is an improved version of `cec` [12], compares AIGs derived from two versions of the design presented for CEC. Internally, it generates random simulation patterns iteratively to detect candidate equivalent pairs and to filter out non-equivalent nodes. Random simulation is repeated until no more refinement can be made, i.e., no more non-equivalent nodes being distinguished. Then, a SAT solver is called to formally prove the equivalence pairs by assuming non-equivalence, similarly to the verification procedure in the resubstitution algorithm presented in the previous section. If the solver returns UNSAT, the equivalence pair is formally proved; otherwise, if the solver returns SAT, a counter-example is generated. The counter-example disproves the given candidate equivalence and potentially other unproved ones.

We implemented simulation-guided CEC by modifying command `&cec` to use pre-generated patterns instead of generating random patterns. This can be useful when the design is optimized with the proposed paradigm, for example, the simulation-guided resubstitution developed in this paper, so that an expressive set of patterns pre-generated, and maybe even supplemented with the counter-examples generated during optimization, is already in hand. Without any extra cost, the patterns can be reused in CEC to reduce SAT calls disproving equivalence.

VIII. EXPERIMENTAL RESULTS

The pattern generation algorithms and the simulation-guided resubstitution framework are implemented in C++-17 as part of the EPFL logic synthesis library *mockturtle*[2] [39]. In Sections VIII-A and VIII-B, we first investigate the expressiveness of simulation patterns generated using different methods by comparing the number of counter-examples encountered in resubstitution. After finding a good strategy, we use it to generate a pattern set to be used for other experiments and report its size before and after bit-packing in Section VIII-C. Then, Section VIII-D demonstrates how an expressive pattern

[1]Available: github.com/berkeley-abc/abc
[2]Available: github.com/lsils/mockturtle

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

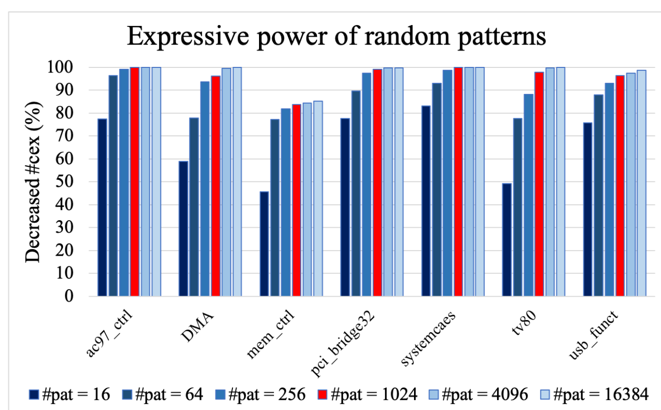

Fig. 6. Decreased percentages of counter-examples when provided with different number ($\#pat$) of random simulation patterns, compared to the baseline $\#pat = 4$.

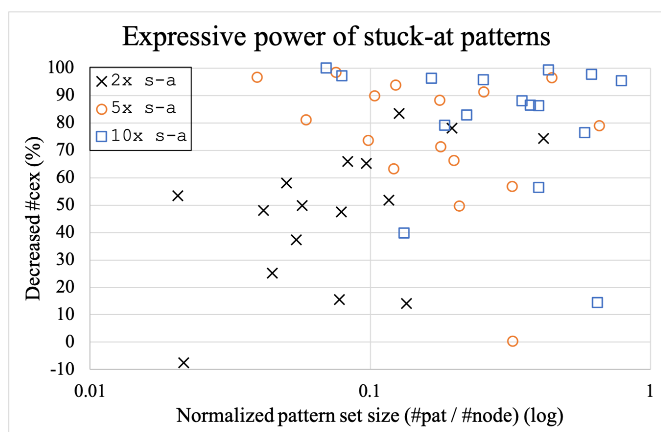

Fig. 7. Decreased percentages of counter-examples when using different sets of stuck-at simulation patterns, compared to the baseline set "`1x s-a`".

set makes a shift in runtime from optimization to pattern generation, and Section VIII-E confirms the reusability of patterns for functionally-modified networks with a set of ECO benchmarks. Finally, the advantages of simulation-guided resubstitution and simulation-guided equivalence checking are shown in Sections VIII-F and VIII-G, respectively.

The experiments are performed on a Linux machine with Xeon 2.5 GHz CPU and 256 GB RAM. The OpenCore designs from IWLS'05 benchmark[3] are used in all experiments, except for those in Section VIII-E. When generating the patterns and testing the quality of resubstitution and equivalence checking in Sections VIII-C, VIII-D, VIII-F and VIII-G, the benchmarks are preprocessed with redundancy removal by iterating command `ifraig` in ABC until no reduction in size. The results for the preprocessed benchmarks are reported in Table I. The preprocessed benchmarks and the simulation patterns used can be found online[4].

A. Size of Simulation Pattern Set

Intuitively, the more simulation patterns used, the higher is the chance that the paradigm saves time by not attempting to prove non-equivalences, i.e., a larger set of simulation patterns is expected to be more expressive. Following the definition of expressive patterns in Section IV, we measure the *expressive power* of a pattern set using the percentage decrease, as compared to a baseline set, in the number of counter-examples encountered in resubstitution, which is calculated separately for each benchmark. Different from the resubstitution framework described in Section VI, the counter-examples are not added to the simulation set, to isolate the impact of the provided patterns.

We start by investigating the expressive power of random patterns based on their count. In Figure 6, each bar represents how expressive is a pattern set of the respective size, compared to the baseline of using only four simulation patterns. The smaller sets are subsets of the larger sets to avoid the biasing effect of randomness. Since the trend is similar for each benchmark, only some medium-sized benchmarks (with around 10 to 20 thousand nodes) are shown here. As the size grows by the factor of four (leading to 4, 16, 64, etc. patterns), the expressive power increases very fast at first, as expected, but saturates at a few hundreds to a few thousands of patterns. Fortunately, a thousand patterns is still a practical size, for which bit-parallel simulation runs fast.

A similar phenomenon is observed when patterns are generated by *StuckAtCheck*. As discussed in Section V-A, additional patterns can be used to ensure that every node has at least b bits of 0 and b bits of 1 in its signature. In the following experiments, stuck-at patterns are abbreviated as "`s-a`", with a prefix "bx" listing parameter b. In Figure 7, since the stuck-at pattern counts are different for each benchmark, the pattern set size is normalized to the network size and plotted in the logarithmic scale. Only benchmarks that are smaller than 25k nodes are included. The baseline pattern set is "`1x s-a`". It is observed that larger sets of patterns are usually more expressive. Note that randomness plays a role in this case, since the default variable polarities, which determine initial variable values in the SAT solver, are randomly reset before each run.

B. Pattern Generation Strategies

In this section, the expressive power of simulation patterns generated by *StuckAtCheck* is compared with the case when observability is used (suffix "`-obs`") and/or when an initial random pattern set of size 256 is used (prefix "`rand 256`").

The observability check and observable pattern generation are done with a fan-out depth of 5 levels. A conflict limit of 1000 is set for the SAT solver, and there is no time-out limit set. A set of 256 random patterns is used as the baseline in Figure 8. Four small benchmarks, for which the random pattern sets are more expressive than "`1x s-a`" and/or "`1x s-a-obs`", are not shown in the figure. Larger benchmarks with more than 25k nodes are also excluded. The geometric means of the sizes of the pattern sets are 143 for "`1x s-a`", 244 for "`1x s-a-obs`", 354 for "`rand 256 + 1x s-a`" and 462 for "`rand 256 + 1x s-a-obs`". On the other hand, the geometric means of the decreased percentages of the counter-examples are 91.3%, 96.5%, 97.1% and 99.5%, respectively.

[3]Available: iwls.org/iwls2005/benchmarks.html
[4]Available: github.com/lsils/sim-LSV_exp

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

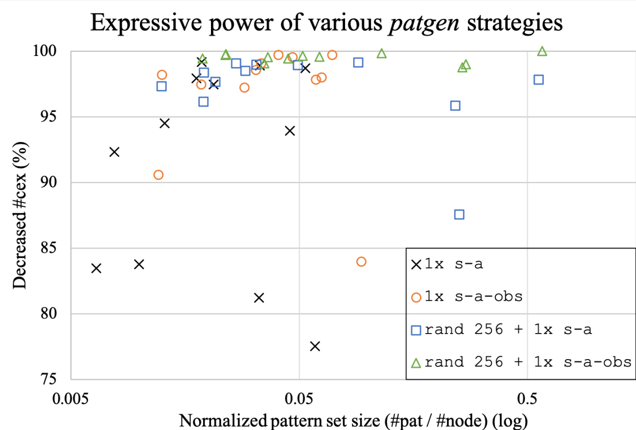

Fig. 8. Decreased percentages of counter-examples when using pattern sets generated with different strategies, compared to the baseline set "`rand 256`".

It is observed that patterns generated by *StuckAtCheck* are usually more expressive than random patterns, except for a few, typically small, benchmarks. Also, using observability increases the expressive power of the generated patterns. Finally, seeding the pattern generation engine with an initial set of random patterns not only speeds up the generation process, but also makes the resulting patterns more expressive.

As the patterns generated with "`rand 256 + 1x s-a-obs`" are shown to be the most expressive, these pattern sets are used in the following experiments in Sections VIII-C, VIII-D, VIII-F and VIII-G. Table I lists some information of the benchmarks and their pattern sets. On average, about 80% of the runtime (about 50% for the largest five benchmarks) in pattern generation were spent in the observability-based methods, including time for checking if a pattern is observable, SAT-solving with the TFO cone, and re-simulation after a new pattern is generated. As seen in Figure 8, using observability increases the expressive power of the generated patterns, but not much. Thus, in practice, one may consider disabling observability awareness for larger benchmarks. There is no constant node detected because the benchmarks are preprocessed with redundancy removal, and there is about 0.1% unobservable nodes found, on average.

C. Pattern Compression with Bit-packing

As discussed in Section V-C, the generated patterns can be packed together to reduce the pattern set size and speed up the simulation. This technique becomes more important in larger benchmarks with huge amounts of primary inputs. The middle part of Table I shows the total number of generated patterns (column *gen.*), the final number of patterns after bit-packing (column *packed*), and the ratio of the two sizes (column *(%)*). The 256 random patterns are not bit-packed, neither included in this table. On average, the sizes of the packed pattern sets are about 70% of the original sets.

D. Effect of Expressive Patterns in Resubstitution

As stated in the introduction, an expressive set of simulation patterns is used to shift the computation effort from the optimization algorithms to pattern pre-computation. Table II shows how the quality of the patterns affects the runtime of pattern generation (*patgen*) and resubstitution (*resub*). For simplicity, only some of the larger benchmarks with more obvious effect are shown in this table. A better set of patterns (Table II, "`rand 256 + 1x s-a-obs`") efficiently filters out many illegal resubstitutions without calling the SAT solver, resulting in the reduced counter-example counts (#cex) and faster runtimes. Note that there is no difference in optimization quality (i.e., circuit size reduction) caused by using different patterns because if an illegal resubstitution is not filtered out by simulation signatures, it is still disproved by SAT solving.

Furthermore, in practice, when the same design is repeatedly synthesized during development or when simulation patterns are reused by different optimization engines, counter-examples from the previous runs can be saved for later use. In this case, the additional counter-example count during later runs can go down to nearly zero, and the runtime is only spent on logic synthesis or verification tasks, such as proving equivalences among the nodes or computing dependency functions and validating them. The latter scheme will be verified in the next section and be used from then on.

TABLE I
NUMBER OF GENERATED PATTERNS BEFORE AND AFTER BIT-PACKING.

benchmark name	size	#PIs	#patterns gen.	packed	(%)	runtime (s)
leon2	787972	298888	23526	14858	63.2	17080.75
leon3_opt	972952	370159	24820	16448	66.3	24566.67
leon3	1085718	370159	24739	16161	65.3	23471.45
leon3mp	650722	217858	13799	9483	68.7	5045.94
netcard	802846	195730	28206	13944	49.4	8896.10
ac97_ctrl	14199	4482	88	27	30.7	0.38
aes_core	21441	1319	163	18	11.0	0.74
des_area	4827	496	18	18	100.0	0.19
des_perf	81998	17850	54	54	100.0	3.95
DMA	21992	5070	886	384	43.3	2.11
DSP	44132	7835	1374	736	53.6	6.87
ethernet	86293	21216	2787	1340	48.1	27.59
i2c	1120	275	65	57	87.7	0.02
mem_ctrl	7870	2281	601	393	65.4	0.70
pci_bridge32	22521	6880	714	207	29.0	1.82
RISC	73789	15678	3139	1012	32.2	17.30
sasc	770	250	1	1	100.0	0.00
simple_spi	1034	280	32	25	78.1	0.01
spi	3762	505	184	184	100.0	0.18
ss_pcm	405	193	2	2	100.0	0.00
systemcaes	12108	1600	39	38	97.4	0.23
systemcdes	2857	512	3	3	100.0	0.07
tv80	9091	732	408	404	99.0	0.55
usb_funct	15245	3620	643	238	37.0	0.92
usb_phy	440	211	9	8	88.9	0.00
vga_lcd	126427	34247	5142	2957	57.5	120.34
wb_conmax	47449	2670	206	170	82.5	1.60

TABLE II
RESUBSTITUTION RUNTIME AS A FUNCTION OF THE NUMBER OF COUNTER-EXAMPLES PRODUCED.

benchmark	rand 256 #cex	runtime (s) *patgen*	*resub*	rand 256 + 1x s-a-obs #cex	runtime (s) *patgen*	*resub*
aes_core	69	0.01	0.72	7	0.74	0.34
des_perf	11	0.01	3.23	2	3.95	3.50
DMA	4923	0.01	2.15	440	2.11	0.41
DSP	8436	0.01	5.71	510	6.87	1.71
ethernet	50334	0.01	67.27	5329	27.59	10.63
pci_bridge32	3303	0.01	2.61	484	1.82	0.96
RISC	15052	0.01	16.02	589	17.30	2.81
vga_lcd	88008	0.01	182.36	3749	120.34	13.16
wb_conmax	920	0.01	0.66	146	1.60	0.64

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE III
RESUBSTITUTION EFFICIENCY AFTER ECO WITH OR WITHOUT COUNTER-EXAMPLE LEARNING.

		A		B		C		D		B vs. D		(A+B) vs. (A+D)	
benchmark version pattern set		old generated		new generated		old with CEX from A		new with CEX from A		reduction rate		reduction rate	
benchmark	size	#cex	time (s)	#cex	time (s)	#cex	time (s)	#cex	time (s)	#cex (%)	time (%)	#cex (%)	time (%)
design1	218679	2711	30.92	2869	31.75	0	12.39	441	17.63	84.63	44.47	43.51	22.53
design2	344	16	0.01	11	< 0.01	0	0.00	11	< 0.01	0.00	N/A*	0.00	0.00
design3	453920	3089	48.52	3006	46.76	0	19.56	219	23.37	92.71	50.02	45.73	24.55
design4	30819	579	0.81	594	0.82	0	0.36	150	0.55	74.75	32.93	37.85	16.56
design5	3582	76	0.04	63	0.04	0	0.03	6	0.03	90.48	25.00	41.01	12.50
design6	77555	1161	4.40	1180	4.51	0	2.24	126	2.78	89.32	38.36	45.02	19.42
design7	62336	844	2.10	907	2.18	1	1.13	123	1.41	86.44	35.32	44.77	17.99
design8	20517	540	0.59	575	0.65	0	0.31	130	0.46	77.39	29.23	39.91	15.32
design9	4650	69	0.05	83	0.05	0	0.03	26	0.04	68.67	20.00	37.50	10.00
design10	15995	86	0.23	138	0.21	0	0.18	71	0.20	48.55	4.76	29.91	2.27
design11	48817	949	2.17	931	2.10	0	1.06	94	1.18	89.90	43.81	44.52	21.55
average		920.00	8.17	941.55	8.10	0.09	3.39	127.00	4.33	72.99	32.39*	37.25	14.79

*The runtime is too fast to compute the reduction rate, hence this benchmark is excluded from the average.

E. Reusability of Simulation Patterns

In support of our assumption, the reusability of the generated patterns and the counter-examples are verified with a set of ECO benchmarks [40]. For each design, there is an old version and a new version which are functionally different. The results of two runs of resubstitution with the two versions of benchmarks are reported and compared in Table III. First, a set of patterns is generated for the old version with "`rand 256 + 1x s-a-obs`" where only the first case of observability check is performed. Columns A and B show the number of counter-examples (#cex) and the runtime of resubstitution on the two versions of benchmarks using this generated pattern set. Comparing them, it is observed that the patterns are as effective on the new benchmarks, even though they are generated with the old ones. In columns C and D, resubstitution is performed again, but using the generated patterns appended with the counter-examples collected in A. There are almost no new counter-examples in column C when the same optimization algorithm is applied on exactly the same benchmarks, as expected. Moreover, when applying on slightly different networks in column D, the number of counter-examples is reduced by 73% comparing to the first run (B). The runtime in D is only slightly higher than C, showing that most of the runtime is spent on computing dependency functions and validating the legal resubstitutions, which are inevitable. The last column compares a flow optimizing first the old networks and then the new ones without learning of counter-examples (A+B) against one that learns the counter-examples from previous runs (A+D).

F. Quality of Simulation-Guided Resubstitution

This section shows the improvements in terms of resubstitution quality. Table IV compares the proposed framework with command `resub` [11] in ABC [38], which performs truth-table-based resubstitution. Because computing simulation patterns in our framework results in detecting combinational equivalences [13], for a fair comparison, the benchmarks are preprocessed by repeating the command `ifraig` in ABC until no more size reduction is observed. The quality of results, presented in the *gain* columns, is measured with the reduction percentage in network size after optimization, i.e., the difference in the number of nodes before and after resubstitution, divided by the original network size. Simulation patterns used in our framework are initially generated with "`rand 256 + 1x s-a-obs`", bit-packed (as described in Section V-C), and then incrementally supplemented with the counter-examples generated from the previous runs of the same resubstitution settings in each column. After the resubstitution run in the last column, the sizes of pattern sets increase by 30% on average.

Two parameters can be set in both flows: the maximum cut size K used to collect divisors in the TFI of the root node and the maximum number N of nodes in the dependency function. Since [11] relies on computing truth tables in the window, $K \leq 10$ is typically used as a reasonable trade-off between efficiency and quality. In contrast, windowing in our framework is applied only to avoid potential runtime blow-up for large benchmarks, and K can be set to arbitrarily large values when longer runtime is acceptable.

When the algorithms are limited to at most one node insertion ($N = 1$), the middle part of Table IV shows that our framework achieves 2.18% network size reduction on average using the same, small window size ($K = 10$), comparing to 1.58% by the state-of-the-art. This improvement is due to better consideration of global satisfiability don't-cares. Moreover, we are able to arbitrarily extend the window size and achieve up to 2.78% gain when longer runtime is acceptable.

In the last columns of Table IV, parameters in `resub` are set to their extreme values ($K = 16, N = 3$), and parameters in our framework are set to large values semantically close to infinity. It is observed that our framework can achieve up to 5.90% reduction while 3.65% is the best `resub` can do, and the improvement comes even with faster runtime in most of the benchmarks. The reason why our framework is especially slow in the largest five benchmarks is because they also have large numbers of primary inputs and large sizes of pattern sets (shown in Table I), which slow down simulation as well as the computation of dependency functions. This can be ameliorated, however, by fine-tuning the trade-off between quality and runtime according to the user's needs.

Furthermore, the proposed framework is also shown to be applicable on 2-LUT networks, or essentially, *Xor-And*

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE IV
RESUBSTITUTION QUALITY ON AIGS COMPARING AGAINST ABC'S `RESUB` COMMAND.

		Baseline: at most one node insertion						Best achievable quality			
		`abc> resub -K 10 -N 1`		Ours, $K = 10, N = 1$		Ours, $K = 100, N = 1$		`abc> resub -K 16 -N 3`		Ours, $K = 100, N = 20$	
benchmark	size	gain (%)	time (s)	gain (%)	time (s)	gain (%)	time (s)	gain (%)	time (s)	gain (%)	time (s)
leon2	787972	0.11	69.48	0.13	65.52	0.32	1639.16	0.35	1811.35	0.65	5984.96
leon3_opt	972952	0.18	55.40	0.23	82.55	0.28	1113.55	0.73	1273.16	1.02	5462.90
leon3	1085718	0.10	55.11	0.11	90.25	0.19	1347.85	0.28	1824.90	0.63	5239.15
leon3mp	650722	0.08	30.16	0.10	41.04	0.19	406.59	0.80	875.65	0.57	1342.39
netcard	802846	0.08	52.79	0.09	60.21	0.13	1062.90	0.28	1562.19	0.56	5425.19
ac97_ctrl	14199	1.25	0.15	1.25	0.08	1.27	0.10	2.24	4.81	6.87	0.93
aes_core	21441	1.50	0.42	1.60	0.48	2.32	2.59	3.02	19.53	6.29	8.62
des_area	4827	1.82	0.08	2.15	0.07	2.15	0.50	3.09	3.50	5.72	1.08
des_perf	81998	6.07	1.37	7.01	2.91	7.17	3.61	8.70	74.10	15.78	7.32
DMA	21992	0.89	0.27	1.04	0.20	1.29	1.12	1.93	8.49	2.78	3.36
DSP	44132	2.13	0.54	2.71	0.64	3.32	4.08	4.14	48.02	5.74	13.92
ethernet	86293	0.31	2.03	0.34	1.95	0.49	15.76	0.95	106.04	2.72	74.15
i2c	1120	4.29	0.01	5.09	0.01	7.68	0.02	8.48	0.56	11.88	0.13
mem_ctrl	7870	1.91	0.08	3.44	0.07	5.17	0.89	4.08	3.67	8.93	2.64
pci_bridge32	22521	0.78	0.40	0.86	0.26	1.19	0.76	2.33	17.52	2.78	3.27
RISC	73789	1.83	0.71	2.18	0.91	4.21	3.94	3.47	56.22	7.56	17.04
sasc	770	0.65	< 0.01	0.65	< 0.01	0.65	< 0.01	1.56	0.13	1.82	0.02
simple_spi	1034	1.74	0.01	1.64	0.01	2.22	0.01	4.64	0.35	5.32	0.06
spi	3762	2.15	0.07	2.23	0.04	2.37	0.36	3.19	2.16	5.24	0.74
ss_pcm	405	0.25	< 0.01	0.25	< 0.01	0.25	< 0.01	0.99	0.03	1.23	< 0.01
systemcaes	12108	0.30	0.11	0.40	0.10	0.45	0.48	0.64	11.04	1.68	2.16
systemcdes	2857	4.83	0.04	5.50	0.06	5.67	0.23	7.46	1.87	11.41	0.28
tv80	9091	2.41	0.15	2.85	0.13	4.93	2.67	5.26	8.62	11.75	7.34
usb_funct	15245	2.93	0.16	3.67	0.14	7.65	0.35	7.04	7.56	11.82	1.96
usb_phy	440	2.73	< 0.01	3.64	< 0.01	3.64	< 0.01	7.73	0.07	10.91	0.01
vga_lcd	126427	0.09	5.07	0.12	4.59	0.14	51.31	0.26	207.27	0.48	153.19
wb_conmax	47449	1.19	0.78	9.59	0.67	9.59	1.99	14.95	48.41	17.15	6.54
average		1.58	10.20	2.18	13.07	2.78	209.66	3.65	295.45	5.90	879.98
geomean		0.81	0.38*	1.02	0.39*	1.35	1.81*	2.13	14.72	3.55	6.15*

*The values smaller than 0.01 are replaced with 0.005 when calculating geomean.

TABLE V
RESUBSTITUTION QUALITY ON XAGS COMPARING AGAINST ABC'S `&MFS` COMMAND.

		`abc> &mfs -a`		Ours, $K = 10, N = 1$	
benchmark	size	gain (%)	time (s)	gain (%)	time (s)
leon2	785623	0.12	612.10	0.11	103.89
leon3_opt	970570	0.13	697.90	0.21	139.80
leon3	1082547	0.10	705.60	0.09	139.79
leon3mp	649333	0.13	317.60	0.09	59.96
netcard	800880	0.07	676.90	0.09	91.23
ac97_ctrl	13945	0.47	0.50	1.23	0.09
aes_core	18951	0.82	5.54	1.89	0.48
des_area	4673	1.16	2.20	2.23	0.08
des_perf	76458	3.23	11.96	7.53	2.87
DMA	21435	0.55	3.37	1.03	0.25
DSP	41795	1.06	15.97	1.90	0.55
ethernet	85355	0.17	19.00	0.30	2.06
i2c	1101	3.72	0.09	5.09	0.01
mem_ctrl	7408	4.94	1.96	3.62	0.07
pci_bridge32	21759	0.38	1.79	0.86	0.25
RISC	69514	1.72	12.79	1.46	0.90
sasc	733	0.82	0.02	0.68	0.01
simple_spi	1003	1.60	0.05	1.69	0.01
spi	3697	0.70	1.29	1.87	0.06
ss_pcm	398	0.00	0.01	0.25	0.01
systemcaes	10652	0.70	1.55	0.58	0.09
systemcdes	2744	3.72	0.62	5.69	0.07
tv80	8751	2.79	9.26	2.43	0.13
usb_funct	14201	1.88	1.00	3.15	0.13
usb_phy	408	3.19	0.01	3.43	0.01
vga_lcd	126093	0.06	56.83	0.11	5.25
wb_conmax	47449	14.38	8.75	9.59	0.63
average		1.80	117.21	2.12	20.32
geomean		N/A	3.93	0.97	0.46

Inverter Graphs (XAGs). Table V compares the proposed framework with command `&mfs` [4] in ABC.[5] The `ifraig`-preprocessed benchmarks are mapped into 2-LUT networks by the command `&if -K 2` in ABC and read in as XAGs in *mockturtle*. The simulation pattern set generated in Section VIII-C with the AIG benchmarks and used in the experiments in Table IV is reused for the XAG experiment. In Table V, the numbers of 2-LUTs (or XAG nodes) are reported in column *size*, and the percentage reduction and runtime of the two algorithms are reported in columns *gain* and *time*, respectively. Using only an unaggressive parameter setting ($K = 10, N = 1$), our framework outperforms command `&mfs` in both optimization quality and efficiency.

G. Reduction on SAT Calls in CEC with Expressive Patterns

Finally, to show the effectiveness of the proposed paradigm on other logic synthesis and verification algorithms, we take CEC as another example. The `&cec` command in ABC [12] is considered the state of the art. It iteratively generates random patterns for simulation to find equivalent pair candidates. This command is modified to take pre-generated patterns and use them for simulation. The number of SAT results (disproving equivalence; #SAT) and UNSAT results (proving equivalence; #UNSAT) in `&cec` with and without using pre-generated expressive patterns are reported in Table VI. For simulation efficiency, an upper limit of 3200 on the number of patterns is set. It can be observed from the table that the average number

[5] While the paper was published in 2011, the technical implementation has been continuously improved over time and there are several versions of the same concept in ABC, such as commands `mfs` and `mfs2`. Among them, `&mfs` is believed to be the newest and the best version.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE VI
EFFICIENCY OF CEC WITH OR WITHOUT USING EXPRESSIVE PATTERNS.

		abc> &cec			&cec with expressive patterns			
benchmark	size	#SAT	#UNSAT	time (s)	#patterns	#SAT	#UNSAT	time (s)
leon2	787972	8579	19738	32.32	3200	7150	19465	41.53
leon3_opt	972952	19529	50162	42.15	3200	14751	50020	49.10
leon3	1085718	113427	127162	88.64	3200	80242	127163	82.64
leon3mp	650722	65439	90482	43.78	3200	37522	84326	35.52
netcard	802846	21691	107513	31.14	3200	19269	107523	28.93
ac97_ctrl	14199	0	2215	0.19	384	41	2215	0.17
aes_core	21441	0	3177	0.71	320	2	3177	0.65
des_area	4827	0	393	0.08	320	0	393	0.07
des_perf	81998	0	5423	1.22	320	0	5423	0.99
DMA	21992	337	2981	0.45	832	298	2981	0.34
DSP	44132	911	6232	1.60	1600	249	6230	1.23
ethernet	86293	596	10505	1.19	1408	9817	10486	2.25
i2c	1120	65	165	0.03	320	33	163	0.03
mem_ctrl	7870	651	927	0.24	832	166	929	0.18
pci_bridge32	22521	612	3132	4.44	576	511	3132	4.40
RISC	73789	3638	9084	2.37	1472	500	9083	1.37
sasc	770	0	116	0.03	320	0	116	0.02
simple_spi	1034	14	157	0.03	320	24	157	0.03
spi	3762	109	469	0.12	448	160	469	0.12
ss_pcm	405	0	62	0.02	320	0	62	0.02
systemcaes	12108	0	1384	0.24	384	6	1384	0.23
systemcdes	2857	0	329	0.06	320	1	329	0.05
tv80	9091	279	1160	0.33	704	225	1160	0.27
usb_funct	15245	809	2003	0.37	512	275	2003	0.25
usb_phy	440	0	57	0.02	320	0	57	0.02
vga_lcd	126427	13852	13682	4.28	3584	1055	13670	2.28
wb_conmax	47449	2	3793	0.61	448	3	3793	0.51
average		994.32	3065.73	0.85	730.18	607.55	3064.18	0.70

of SAT results is reduced by about 40%; when combined with the UNSAT results, which are unchanged, the total number of SAT solver calls is reduced by about 9.5%. In most cases, the runtime does not decrease because it is dominated by the UNSAT calls, and that too many patterns slow down simulation. Nevertheless, the runtime overhead in simulation can be mitigated if the patterns can be better compacted, or if the simulation can be speeded up (e.g., by using *Haswell New Instructions* (AVX2) which provides single-cycle bitwise operations on longer machine words) in a future implementation of simulation-guided CEC. More importantly, by showing a decrease in unnecessary SAT solver calls, the idea of guiding CEC with expressive simulation patterns is shown to be useful in verification as well.

IX. CONCLUSIONS AND FUTURE WORK

In this paper, we (1) present a simulation-guided logic synthesis and verification paradigm, which leverages pre-generated expressive simulation patterns to approximate the global Boolean functions with reduced need for SAT-based verification; (2) propose several strategies to generate expressive simulation patterns, including seeding with random patterns, stuck-at value checking, and resolving un-observability; (3) demonstrate the benefits of the proposed paradigm with improved resubstitution quality and reduced SAT solver calls in CEC; (4) show the reusability of the expressive patterns and counter-examples across different algorithms and with ECO modifications.

Parameters influencing the expressiveness of the simulation patterns are studied. In particular, stuck-at patterns generated with observability awareness and seeded with a small set of random patterns are found to be the most expressive. The expressive patterns are shown to be able to move runtime from optimization and verification to their pre-generation, which is advantageous because they are also shown to be reusable in resubstitution after ECO and in a different algorithm such as CEC. The experimental results show that the simulation-guided resubstitution framework allows low-cost consideration of global satisfiability don't-cares and unlimited extension of the window sizes used, which improves the average network size reduction from 1.58% to 2.77%, compared to a state-of-the-art windowing-based resubstitution algorithm. When comparing the best achievable quality of the two frameworks, a larger improvement from 3.65% to 5.83% is shown. Effectiveness of the proposed paradigm in CEC is also supported by experimental results with a 9.5% reduction in the number of SAT solver calls.

While resubstitution guided by simulation signatures automatically accounts for satisfiability don't-cares, observability don't-cares can also be considered in resubstitution, resulting in better quality. Our preliminary result on utilizing ODCs in simulation-guided resubstitution shows about 1% further circuit size reduction at the cost of 5x more runtime. It remains our future work to enhance the efficiency of resubstitution with ODCs. On the other hand, as shown in Section VIII-D, using expressive patterns reduces the chance of encountering counter-examples, making it possible to further reduce the use of SAT solving by validating several candidates at the same time if the majority of them are legal.

Other future works include developing strategies to refine and enhance the generated simulation patterns further and metrics to evaluate and sort the patterns. To maximize the benefit of the generated patterns, other algorithms adopting this paradigm can also be developed so that the patterns can be reused more often in a logic synthesis flow.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

References

[1] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, 1990.

[2] G. De Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[3] H.-P. Lin, J.-H. R. Jiang, and R.-R. Lee, "To SAT or not to SAT: Ashenhurst decomposition in a large scale," in *Proc. of ICCAD*. IEEE, 2008, pp. 32–37.

[4] A. Mishchenko, R. K. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Tran. on Reconfigurable Technology and Systems*, vol. 4, no. 4, pp. 1–23, 2011.

[5] H. Riener, W. Haaswijk, A. Mishchenko, G. De Micheli, and M. Soeken, "On-the-fly and DAG-aware: Rewriting Boolean networks with exact synthesis," in *Proc. of DATE*. IEEE, 2019, pp. 1649–1654.

[6] S. B. Akers, "Binary decision diagrams," *IEEE Tran. on Computers*, no. 6, pp. 509–516, 1978.

[7] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Tran. on Computers*, vol. 100, no. 8, pp. 677–691, 1986.

[8] C. A. Tovey, "A simplified NP-complete satisfiability problem," *Discrete Applied Mathematics*, vol. 8, no. 1, pp. 85–89, 1984.

[9] J. P. Marques-Silva and K. A. Sakallah, "Boolean satisfiability in electronic design automation," in *Proc. of DAC*, 2000, pp. 675–680.

[10] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. K. Brayton, and M. Chrzanowska-Jeske, "Using simulation and satisfiability to compute flexibilities in Boolean networks," *IEEE Tran. on CAD*, vol. 25, no. 5, pp. 743–755, 2006.

[11] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Proc. of IWLS*, 2006.

[12] A. Mishchenko, S. Chatterjee, R. K. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *Proc. of ICCAD*. IEEE, 2006, pp. 836–843.

[13] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," ERL Technical Report, Tech. Rep., 2005.

[14] H. Cox and J. Rajski, "Stuck-open and transition fault testing in CMOS complex gates," in *Proc. of ITC*. IEEE, 1988, pp. 688–694.

[15] M. Damiani and G. De Micheli, "Observability don't care sets and Boolean relations." in *Proc. of ICCAD*, 1990, pp. 502–505.

[16] T. Jarratt, C. M. Eckert, N. H. Caldwell, and P. J. Clarkson, "Engineering change: an overview and perspective on the literature," *Research in Engineering Design*, vol. 22, no. 2, pp. 103–124, 2011.

[17] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Tran. on CAD*, vol. 21, no. 12, pp. 1377–1394, 2002.

[18] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Proc. of DAC*. IEEE, 2014, pp. 1–6.

[19] I. Háleček, P. Fišer, and J. Schmidt, "Are XORs in logic synthesis really necessary?" in *Proc. of DDECS*. IEEE, 2017, pp. 134–139.

[20] W. Haaswijk, M. Soeken, L. Amarú, P.-E. Gaillardon, and G. De Micheli, "A novel basis for logic rewriting," in *Proc. of ASPDAC*. Ieee, 2017, pp. 151–156.

[21] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Tran. on CAD*, vol. 26, no. 2, pp. 240–253, 2007.

[22] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of reasoning*. Springer, 1983, pp. 466–483.

[23] T. J. Schaefer, "The complexity of satisfiability problems," in *Proc. of ACM Symposium on Theory of Computing*, 1978, pp. 216–226.

[24] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," *IEEE Tran. on VLSI*, vol. 2, no. 2, pp. 137–148, 1994.

[25] A. Mishchenko and R. K. Brayton, "SAT-based complete don't-care computation for network optimization," in *Proc. of DATE*. IEEE, 2005, pp. 412–417.

[26] H. Sato, Y. Yasue, Y. Matsunaga, and M. Fujita, "Boolean resubstitution with permissible functions and binary decision diagrams," in *Proc. of DAC*, 1991, pp. 284–289.

[27] V. N. Kravets and K. A. Sakallah, "M32: A constructive multilevel logic synthesis system," in *Proc. of DAC*, 1998, pp. 336–341.

[28] V. N. Kravets and P. Kudva, "Implicit enumeration of structural changes in circuit optimization," in *Proc. of DAC*, 2004, pp. 438–441.

[29] W. Craig, "Linear reasoning: A new form of the Herbrand-Gentzen theorem," *The Journal of Symbolic Logic*, vol. 22, no. 3, pp. 250–268, 1957.

[30] H. Riener, E. Testa, L. Amaru, M. Soeken, and G. De Micheli, "Size optimization of MIGs with an application to QCA and STMG technologies," in *Proc. of International Symposium on Nanoscale Architectures*, 2018, pp. 157–162.

[31] L. Amarú, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, J. Olson, R. K. Brayton, and G. De Micheli, "Improvements to Boolean resynthesis," in *Proc. of DATE*. IEEE, 2018, pp. 755–760.

[32] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," *IEEE Tran. on CAD*, vol. 25, no. 12, pp. 2894–2903, 2006.

[33] J. P. Roth, "Diagnosis of automata failures: A calculus and a method," *IBM Journal of Research and Development*, vol. 10, no. 4, pp. 278–291, 1966.

[34] J. Cong and K. Minkovich, "LUT-based FPGA technology mapping for reliability," in *Proc. of DAC*, 2010, pp. 517–522.

[35] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer Aided Verification*. Springer, 2000, pp. 154–169.

[36] B. Alizadeh and Y. Abadi, "Incremental SAT-based correction of gate level circuits by reusing partially corrected circuits," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 12, pp. 3063–3067, 2020.

[37] S. Mitra and K. S. Kim, "XPAND: an efficient test stimulus compression technique," *IEEE Tran. on Computers*, vol. 55, no. 2, pp. 163–173, 2006.

[38] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. of CAV*. Springer, 2010, pp. 24–40.

[39] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli, "The EPFL logic synthesis libraries," *arXiv preprint arXiv:1805.05121v2*, 2019.

[40] V. N. Kravets, J.-H. R. Jiang, and H. Riener, "Learning to automate the design updates from observed engineering changes in the chip development cycle," in *Proc. of DATE*. IEEE, 2020, pp. 738–743.

Siang-Yun Lee is a Ph.D. candidate at the Integrated Systems Laboratory at EPFL, Switzerland led by Prof. Giovanni De Micheli. She graduated from the Department of Electrical Engineering of National Taiwan University, Taipei, Taiwan, in 2019. In NTU, she worked with Prof. Jie-Hong Roland Jiang on threshold logic synthesis. Her research interests include logic synthesis and design automation for emerging technologies. She is currently a maintainer of the EPFL logic synthesis library mockturtle.

Heinz Riener is a researcher at EPFL, Lausanne, Switzerland. He holds a Ph.D. degree in Computer Science from University of Bremen, Germany. He received his B.Sc. and M.Sc. degree from the Technical University Graz, Austria. From 2015 to 2017, he worked at the German Aerospace Center, Bremen, Germany, in the group of Avionics Systems. His research interests are logic synthesis, formal methods, and computer-aided verification of hardware and software systems.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3108704, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Alan Mishchenko received the M.S. degree from the Moscow Institute of Physics and Technology, Moscow, Russia, in 1993 and the Ph.D. degree from the Glushkov Institute of Cybernetics, Kiev, Ukraine, in 1997. In 2002, he joined the EECS Department, University of California at Berkeley, Berkeley, CA, USA, where he is currently a Full Researcher. His current research interests include computationally efficient logic synthesis and formal verification.

Robert K. Brayton received his Ph.D. degree in mathematics from MIT in 1961. He was a member of the Mathematical Sciences Department of the IBM T. J. Watson Research Center until he joined the EECS Department at Berkeley in 1987. He is a Fellow of the IEEE and a member of the National Academy of Engineering. Prof. Brayton held the Buttner Chair and the Cadence Distinguished Professorship of Electrical Engineering and is currently a Professor in the Graduate School at Berkeley.

Giovanni De Micheli is Professor and Director of the Integrated Systems Laboratory at EPFL, Lausanne, Switzerland. Previously, he was Professor of Electrical Engineering at Stanford University. Prof. De Micheli is a Fellow of ACM and IEEE, a member of the Academia Europaea and an International Honorary member of the American Academy of Arts and Sciences. His current research interests include several aspects of design technologies for integrated circuits and systems, such as synthesis for emerging technologies.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/