Solving Satisfiability in Combinational Circuits

João Marques-Silva and Luís Guerra e Silva
Technical University of Lisbon

Editor's note:
As EDA evolves, researchers continue to find modeling tools to solve problems of test generation, design verification, logic, and physical synthesis, among others. One such modeling tool is Boolean satisfiability (SAT), which has very broad applicability in EDA. The authors review modern SAT algorithms, show how these algorithms can account for structural information in combinational circuits, and explain what recursive learning can add to SAT.
—Marcelo Lubaszewski, Federal University of Rio Grande do Sul

Boolean satisfiability is a widely used modeling tool in EDA. Well-known applications of SAT include test-pattern generation for stuck-at faults, delay and bridging faults, equivalence checking, redundancy removal, and logic synthesis.[1] More recent applications include FPGA routing, bounded model checking, and crosstalk noise analysis. In addition to finding new applications of SAT to EDA, researchers have strived to propose algorithms for solving instances of the SAT problem that state-of-the-art EDA tools will have to solve.

Given a Boolean function ψ with n variables, the SAT problem consists of assigning values to the variables such that ψ assumes value 1, or prove that no such assignment exists and the function is equal to 0. A Boolean product of sums, or *conjunctive normal form* formula, is the most often used representation of the Boolean function.

A CNF formula is a conjunction of clauses, each of which is a disjunction of literals. A literal is either a variable or its negation. For example, CNF formula $\psi=(a+\overline{b})(b+\overline{c}+d)$ contains two clauses, $(a+\overline{b})$ and $(b+\overline{c}+d)$, and five literals. A clause is *satisfied* if at least one of its literals assumes value 1, *unsatisfied* if all its literals assume value 0, *unit* if all but one literal assume value 0, and *unresolved* otherwise. Literals with no assigned Boolean value are *free literals*. A formula is satisfied if all its clauses are satisfied, and unsatisfied if at least one clause is unsatisfied. It is often simpler to refer to clauses as sets of literals, and to the CNF formula as a set of clauses.

Here, we review current algorithms for solving instances of SAT, emphasizing the algorithms that are particularly significant for SAT in combinational circuits, and the attempts to develop unified algorithmic frameworks. Consequently, we study the techniques Marques-Silva et al. proposed,[2–4] but also review techniques by Bayardo and Schrag[5] and Tafertshofer, Ganz, and Henftling.[6]

Current SAT solvers

Of the many approaches proposed for solving the SAT problem, the two most widely used are *local search* and *backtrack search*. Local search is seldom used in EDA, because local search algorithms cannot prove unsatisfiability. Consequently, backtrack search algorithms are the most promising for solving SAT in EDA.

Most, if not all, backtrack search SAT algorithms apply the *unit clause rule*[7]—that is, if a clause is unit, the formula can be satisfiable only if the sole free literal has value 1. The iterated application of the unit clause rule is often referred to as Boolean constraint propagation (BCP).

Implementing some of the techniques shared by some backtrack search algorithms requires explaining the Boolean variable assignments implied by the CNF formula clauses. For example, let $(w+z+\overline{u})$ be a CNF formula, and assume variable assignments $\{w=0, u=1\}$. To satisfy the clause according to the unit clause rule, the value of z must be 1. The implied assignment, $z=1$, has explanation $\{w=0, u=1\}$. A more formal description of explanations for implied variable assignments in the SAT context, as well as a description of identification mechanisms, is available elsewhere.[4]

Figure 1 illustrates a generic backtrack search SAT algorithm that captures the organization of several of the

0740-7475/03/$17.00 © 2003 IEEE Copublished by the IEEE CS and the IEEE CASS

most competitive algorithms. The algorithm searches the space of possible assignments for the problem instance variables. At each stage of the search, the *Decide( )* function selects an assignment. Selected assignments have no explanation because these assignments are not implied by a unit clause. The algorithm also associates a decision level, λ, with each selected assignment.

The *Deduce( )* function, which usually corresponds to the BCP procedure, identifies implied necessary assignments. Whenever a clause becomes unsatisfied, the *Deduce( )* function returns a conflict indication, which the *Diagnose( )* function analyzes. The conflict's diagnosis returns a backtracking decision level, β (the level to which the search process can provably retrace its steps). The *Backtrack( )* function clears variable assignments from current decision level λ up to decision level β.

Currently, the most efficient SAT algorithms implement several key techniques for solving difficult instances of SAT:

- The algorithms can use conflict analysis to implement nonchronological backtracking strategies. Hence, they can skip assignment selections deemed irrelevant during the search.[4,5]
- They can also use conflict analysis to identify and record new clauses that denote implicates of the Boolean function associated with the CNF formula (recall that an implicate, α, of Boolean function f is such that $\alpha = 0 \Rightarrow f = 0$). Clause recording plays a key role in new SAT algorithms, even though most large recorded clauses will eventually be deleted.[4,5]
- Relevance-based learning extends the life spans of large recorded clauses that will eventually be deleted.[5]
- Conflict-induced necessary assignments[4] denote variable assignments that will prevent a given conflict from recurring during the search.

Before executing the actual search, the SAT algorithm can apply different forms of preprocessing to the CNF formula.[4] In general, a *Preprocess( )* function denotes the desired preprocessing techniques.

The search-pruning techniques described above rely extensively on the search algorithm's ability to explain the conflicts' causes. Most approaches to identifying conflict causes construct a new clause that can subsequently prevent the same conflict from recurring.

The CNF formula in Figure 2 illustrates clause recording. Assume a sequence of decision assignments, $\{v = 1, y = 0, x = 1\}$. The last decision assignment causes clause $(y + w + z)$ to become unsatisfied. Because this conflict is due to assignments $\{v = 1, y = 0, x = 1\}$, we can say

$$(v = 1) \wedge (y = 0) \wedge (x = 1) \rightarrow \psi = 0$$

However, we want the formula to be satisfied ($\psi = 1$), so we must have $v = 0 \vee y = 1 \vee x = 0$ or, in clausal form $(\bar{v} + y + \bar{x})$. Thus, by identifying the conflict's causes, we can create an implicate of the CNF formula, which we can add to the formula as a new clause. The most competitive SAT algorithms implement techniques for recording new clauses from conflict causes.[4,5]

```
int SAT(ψ)
{
        λ = 0;
        Preprocess(ψ);
        while ( Decide(ψ, λ) == DECISION ) {
                if ( Deduce(ψ, λ) == CONFLICT ) {
                        β = Diagnose(ψ, λ) ;
                        if ( β == −1 )
                                return UNSATISFIABLE;
                        else {
                                Backtrack(ψ, λ , β);
                                λ = β;
                        }
                }
                λ = λ + 1;
        }
        return SATISFIABLE;
}
```

Figure 1. Generic satisfiability algorithm.

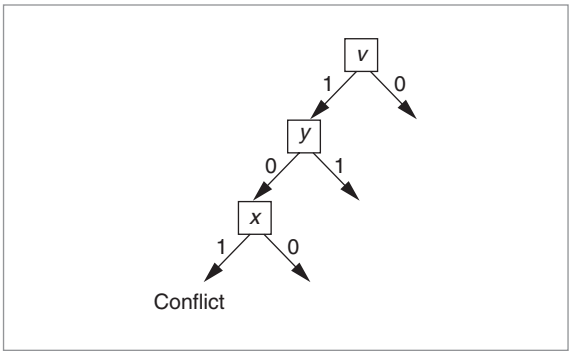

Figure 2. Diagram of the clause recording example $\psi = (y + w + z)(\bar{z} + \bar{x})(\bar{w} + \bar{v} + \bar{x})$. The assignments v = 1, y = 0, and x = 1 imply a conflict, creating the new clause $(\bar{v} + y + \bar{x})$.

SAT in combinational circuits

The algorithms described in the previous section have proven to effectively solve real-world instances of SAT. In particular, using CNF models and SAT algorithms has important advantages:

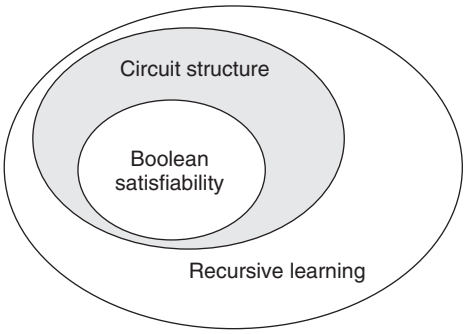

Figure 3. Proposed layered approach for solving SAT.

Table 1. Threshold values on justification counters.

Gate	$\upsilon_0(x)$	$\upsilon_1(x)$
AND/NAND	1	$\|FI(x)\|$
OR/NOR	$\|FI(x)\|$	1
BUFF/NOT	$\|FI(x)\|$	$\|FI(x)\|$
XOR/XNOR	$\|FI(x)\|$	$\|FI(x)\|$

- Existing and extensively validated SAT algorithms can replace dedicated algorithms.
- Improvements and new SAT algorithms can be easily applied to target applications.

However, their use in combinational circuits also has several drawbacks:

- As Tafertshofer, Ganz, and Henftling observe, the circuit's structural information, often crucial, is lost.[6]
- In many EDA problems, designers must solve numerous instances of SAT for each circuit. Hence, mapping a given problem description into SAT can take a significant percentage of the overall running time.
- Computed input patterns are generally overspecified, a serious drawback in applications such as circuit testing and binate constraint solving.
- Powerful circuit-based reasoning techniques, such as recursive learning,[8] cannot easily be applied.

Our approach uses structural information in SAT algorithms. We add a layer that maintains circuit-related information (fan in and fan out, for example) and value justification relations to a generic SAT algorithm. (We can use any SAT algorithm that will accept this layer.)

This approach eliminates some of the drawbacks of using CNF models and SAT algorithms in combinational circuits—the inaccessibility to structural information and overspecification of input patterns, for example. Unlike Tafertshofer, Ganz, and Henftling's approach,[6] ours does not require modifying the data structures used for SAT, so we can easily augment existing algorithmic SAT solutions with the layer we propose for handling structural information. Moreover, our approach requires only minor modifications to SAT algorithms, making it significantly simpler.[6]

Figure 3 illustrates the proposed unified approach for solving SAT in combinational circuits. Basically, adding new layers to an existing SAT algorithm lets us exploit circuit structure and incorporate search techniques targeted specifically to combinational circuits.

We start by formalizing the Boolean satisfiability problem in combinational circuits. Assume we want to satisfy property C_p of combinational circuit C to objective value o. We denote this satisfiability problem by (C_p, o) and map it into SAT instance ψ. To solve instances of SAT in combinational circuits, and with the goal of using structural information in the SAT algorithm, we associate the following information with each variable x of ψ, which also represents circuit node x of C:

- $FI(x)$ denotes the set of fan-in nodes of x.
- $FO(x)$ denotes the set of fan-out nodes of x.
- $\upsilon_v(x)$ denotes the threshold value on the number of suitable assigned inputs of x needed to justify value v on node x.
- $\iota_v(x)$ denotes the actual number of assigned inputs of x that are involved in justifying value v on node x.

According to these definitions, each circuit node x with assigned value v becomes justified whenever $\iota_v(x) \geq \upsilon_v(x)$.

Table 1 lists example threshold values on the number of assigned inputs required for justifying a given node. For example, an AND gate needs at least one input assigned value 0 to justify assigning 0 to x, whereas to assign value 1 to x, all inputs must be assigned 1. Hence, $\upsilon_0(x) = 1$ and $\upsilon_1(x) = |FI(x)|$. As another example, an XOR gate justification of any assigned value requires assignments to all gate inputs; hence, $\upsilon_0(x) = \upsilon_1(x) = |FI(x)|$. This information is easy to derive for other simple gates as well, and in all cases we have $\upsilon_0(x), \upsilon_0(x) \in \{1, FI(x)\}$.

For any simple gate with output x, we can associate each fan-in node w with the counters to be updated as a result of assigning value v to w. For an AND gate, for example, assigning 0 to fan-in node w increments $\iota_0(x)$

by 1, and assigning 1 to node w increments $\iota_1(x)$ by 1. For an XOR gate, assigning an input node updates both counters.

Standard search algorithms in combinational circuits[9] maintain a *justification frontier*, denoting the sets of variables and nodes that require justification. The condition indicating the need for node justification is $(v(x) = v) \wedge (\iota_v(x) < \upsilon_v(x))$, where $v \in \{0, 1\}$.

Given these definitions, we can adapt a SAT algorithm to allow proper maintenance of justification information. Moreover, we can use the fan-in information to implement structure-based heuristic decision-making procedures, such as simple or multiple backtracing.[9]

In Figure 1, functions *Deduce()* and *Diagnose()* must invoke dedicated procedures for updating node justification information. Additionally, *Decide()* now tests for satisfiability by checking for an empty justification frontier instead of checking whether all clauses are satisfied. These are the only required modifications to the general SAT algorithm. Observe that the *Decide()* function can be adapted to perform backtracing, given the fan-in information associated with each variable.

The data structures described above operate in much the same way as justification works in combinational circuits.[9] The main difference is that in our approach, justification and value consistency are formally dissociated: The SAT algorithm handles value consistency, and the added layer handles justification.

Consider the example circuit in Figure 4, assuming the first assignment is $y = 0$. The justification frontier becomes $\{y\}$. We assume that the next assignment is $e = 0$. Because this assignment justifies $y = 0$, the justification frontier becomes $\{e\}$. After assigning $e = 0$, we have $\upsilon_0(y) = \iota_0(y) = 1$, $\upsilon_0(e) = 1$, and $\iota_0(e) = 0$, causing y to be justified and e to enter the justification frontier. Finally, we consider primary input assignment $a = 1$, which causes e to be justified and leaves the justification frontier empty.

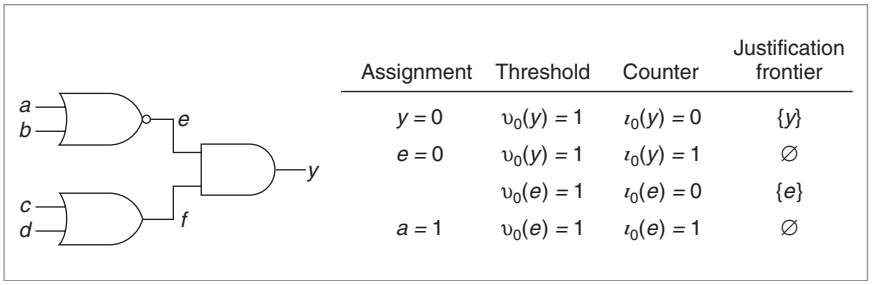

Figure 4. Justification example.

Recursive learning

Recursive learning has been used extensively to solve Boolean satisfiability in combinational circuits.[8] Recursive-learning applications include test pattern generation, combinational equivalence checking, and logic synthesis. We can also apply recursive learning to CNF formulas.

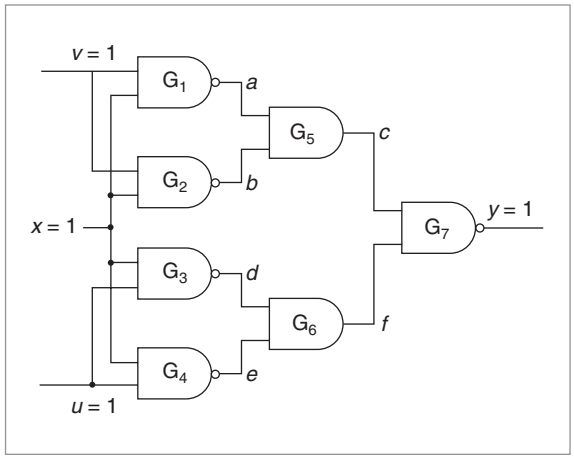

Figure 5. Recursive learning on combinational circuits. $x = 1$ is necessary to justify $y = 1$.

Combinational circuits

Consider the example in Figure 5, in which output value $y = 1$ of gate G_7 is not justified. Recursive learning analyzes the justifications for each unjustified gate output value, trying to identify common necessary assignments. The gate justification analysis process continues recursively until a predefined depth limit is reached.

Assume that for our example the depth limit is 2. The possible justifications for assignment $y = 1$ are $c = 0$ and $f = 0$. Hence, we start with depth 1 of the recursive-learning procedure by considering $c = 0$, which does not imply additional assignments. Next, we go to depth 2 and consider the possible justifications for gate G_5 with output $c = 0$. In this case, the possible justifications are $a = 0$ and $b = 0$. The first justification, $a = 0$, implies assignment $x = 1$ (due to $v = 1$). Because the second justification, $b = 0$, also implies assignment $x = 1$, we can conclude that assignment $c = 0$ implies $x = 1$.

The other justification at depth 1 of the recursive-learning procedure is $f = 0$. Using the same reasoning we used previously, we can readily conclude that this assignment also implies $x = 1$. It is straightforward to

Assignments $\{z = 1, u = 0\}$

$$\omega_1 = (u + x + \bar{w})$$
$$\omega_2 = (x + \bar{y})$$
$$\omega_3 = (w + y + \bar{z})$$

Figure 6. Recursive learning on a CNF formula. An analysis of the assignments shows us that we need assignment $x = 1$ to satisfy the formula.

Assignments: $\{y = 1, u = 1, v = 0\}$

$$\omega_1 = (\bar{x} + a)$$
$$\omega_2 = (v + \bar{x} + b)$$
$$\omega_3 = (\bar{a} + \bar{b} + c)$$
$$\omega_4 = (\bar{x} + d)$$
$$\omega_5 = (\bar{u} + \bar{x} + e)$$
$$\omega_6 = (\bar{d} + \bar{e} + f)$$
$$\omega_7 = (\bar{c} + \bar{f} + y)$$
$$\omega_8 = (x + w + \bar{z})$$

Figure 7. Depth-2 recursive learning on a CNF formula. x = 0 is a necessary assignment.

conclude that any justification for assignment $y = 1$ implies assignment $x = 1$. Hence, $x = 1$ is a necessary assignment, given assignment $y = 1$.

CNF formulas

We can apply similar reasoning to CNF formulas. We begin by observing that to justify gates (that is, to identify consistent assignments), we check the possible justifications for common necessary assignments. For clause ω with one or more literals assigned value 0, at least one unassigned literal must eventually be assigned value 1 if the formula is to be satisfiable. Hence, we assign value 1 to each unassigned literal in ω and analyze the results. Assignments that are implied by every such assignment are deemed necessary for the formula to be satisfiable.

For example, consider the CNF formula and assignments in Figure 6. Our objective is to analyze how we might satisfy ω_3. For ω_3 to be satisfied, and because $z = 1$, we must either have $w = 1$ or $y = 1$. We consider each assignment separately.

- Assignment $w = 1$ implies assignment $x = 1$, due to assignment $u = 0$ and due to ω_1.
- Similarly, assignment $y = 1$ implies assignment $x = 1$ due to ω_2.

We can thus conclude that assignment $x = 1$ is necessary to satisfy the CNF formula. Moreover, we can conclude by inspection that this is true only because $z = 1$ and $u = 0$. Thus, $(z = 1) \wedge (u = 0) \rightarrow (x = 1)$; or, in clausal form, $(\bar{z} + u + x)$.

The most significant conclusion is that the recursive-learning procedure not only identifies necessary assignments but also can help identify new implicates of the CNF formula that we can add to the original formula as new clauses. Because this recursive-learning procedure identifies new clauses, it does not repeat the same reasoning process at a later stage in the search to rederive the same necessary assignment. Moreover, it can facilitate identification of other necessary assignments that would otherwise not be identifiable.

Figure 7 gives a more complete example of the depth-2 recursive-learning procedure on CNF formulas.

We can initiate the recursive-learning process on ω_7 because $y = 1$. Assignments $c = 0$ and $f = 0$ satisfy clause ω_7. The recursive-learning process for depth 1 considers each of these assignments separately. Assignment $c = 0$ implies no assignments. However, clause ω_3 now exhibits a new literal with value 0. Hence, at depth 2 of the recursive-learning process, we analyze clause ω_3. Assignments $a = 0$ or $b = 0$ can satisfy ω_3. Considering each assignment individually yields the implied assignment $x = 0$. For $a = 0$, we get $x = 0$ from clause ω_1; for $b = 0$, we get the implied assignment from ω_2 because $v = 0$.

We perform a similar analysis for $f = 0$. Because $f = 0$, clause ω_6 exhibits a literal with value 0. Hence, we analyze ω_6 at depth 2 and find that assignments $d = 0$ and $e = 0$ can satisfy it. Both $d = 0$ and $e = 0$ imply assignment $x = 0$. For $d = 0$, we get $x = 0$ from clause ω_4; for $e = 0$, we get $x = 0$ from ω_5 because $u = 1$. Hence, $(y = 1) \wedge (v = 0) \wedge (u = 1) \rightarrow (x = 0)$; or in clausal form, $(y + \bar{v} + u + \bar{x})$. As in the previous example, the recursive-learning process identifies a new clause that implies assignment $x = 0$, and, during the subsequent search, it can imply the assignment again.

Extended recursive learning on combinational circuits

When solving SAT on a CNF formula derived from a combinational circuit for which we have structural information (using the additional layer described earlier), we can improve the recursive-learning procedure's perfor-

mance by restricting it to unjustified node assignments. In addition, for each unjustified node assignment, $y = v_y$, we can restrict the reasoning procedure to unresolved clauses that relate y to immediate fan-in nodes. This approach yields a significantly faster recursive-learning procedure, similar to the original procedure for circuits, and can also identify implicates of the CNF formula.[2]

SAT SOLVERS have dramatically improved during the past few years. The most recent generation of SAT solvers uses well-established techniques—clause recording and nonchronological backtracking—but it is also based on search restart strategies,[10] new lazy-data structures,[11] new variable-branching heuristics,[11] and new recorded-clause-deletion policies.[12] Because these improvements have been applied solely to SAT solvers operating on CNF formulas, they do not exploit the structural information of combinational circuits. In the near term, these improvements should be integrated in SAT solvers that exploit the structural information of combinational circuits. In the far term, we can expect to see adaptive algorithms that can be dynamically reconfigured depending on the search algorithm's progression. ■

■ References

1. A. Saldanha and V. Singhal, "Solving Satisfiability in CAD Problems," *Proc. Cadence Technical Conf.*, 1997, pp. 191-195.
2. J.P. Marques-Silva and L. Guerra e Silva, "Solving Satisfiability in Combinational Circuits with Backtrack Search and Recursive Learning," *Proc. XII Symp. Integrated Circuits and Systems Design,* IEEE Press, 2000, pp. 192-195.
3. J.P. Marques-Silva and T. Glass, "Combinational Equivalence Checking Using Satisfiability and Recursive Learning," *Proc. Design, Automation and Test in Europe* (DATE 99), IEEE CS Press, 1999, pp. 145-149.
4. J.P. Marques-Silva and K.A. Sakallah, "Grasp: A New Search Algorithm for Satisfiability," *Proc. Int'l Conf. Computer-Aided Design* (ICCAD 96), IEEE CS Press, 1996, pp. 220-227.
5. R. Bayardo Jr. and R. Schrag, "Using CSP Look-Back Techniques to Solve Real-World SAT Instances," *Proc. Nat'l Conf. Artificial Intelligence*, AAAI Press, 1997, pp. 203-208.
6. P. Tafertshofer, A. Ganz, and M. Henftling, "A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking, and Optimization of Netlists," *Proc. Int'l Conf. Computer-Aided Design* (ICCAD 97), IEEE CS Press, pp. 648-657.
7. M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *J. ACM*, vol. 7, July 1960, pp. 201-215.
8. W. Kunz and D. Stoffel, *Reasoning in Boolean Networks,* Kluwer Academic, 1997.
9. M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design,* Computer Science Press, 1990.
10. L. Baptista and J.P. Marques-Silva, "Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability," *Proc. Int'l Conf. Principles and Practice of Constraint Programming*, *Lecture Notes in Computer Science*, vol. 1894, Springer, 2000, pp. 489-494.
11. M. Moskewicz et al., "Engineering an Efficient SAT Solver," *Proc. Design Automation Conf.* (DAC 01), ACM Press, 2001, pp. 530-535.
12. E. Goldberg and Y. Novikov, "BerkMin: A Fast and Robust SAT-Solver," *Proc. Design, Automation and Test in Europe* (DATE 02), IEEE CS Press, 2002, pp. 142-149.

João Marques-Silva is a professor in the Information Systems and Computer Science Department of the Technical University of Lisbon. His research interests include algorithms for Boolean satisfiability and optimization and their applications to design automation problems. Marques-Silva has a BS and MS in electrical and computer engineering from the Technical University of Lisbon, and a PhD in electrical engineering and computer science from the University of Michigan, Ann Arbor. He is a senior member of the IEEE and a member of the ACM.

Luís Guerra e Silva is a PhD candidate at the Technical University of Lisbon, where he is also an assistant lecturer. His research interests focus on efficient data structures and algorithms for solving complex EDA problems. Guerra e Silva has a BSc in electrical and computer engineering and an MSc in electrical and computer engineering from the Technical University of Lisbon. He is a member of the IEEE.

■ Direct questions and comments about this article to João Marques-Silva, Dept. of Informatics, Technical Univ. of Lisbon, IST/INESC-ID/CEL, Lisbon, Portugal; jpms@inesc-id.pt.