Canonical Computation without Canonical Representation

Alan Mishchenko Robert Brayton
Department of EECS
University of California, Berkeley
{alanmi, brayton}@berkeley.edu

Ana Petkovska Mathias Soeken
School of Comp. and Comm. Sciences
EPFL, Lausanne, Switzerland
{ana.petkovska, mathias.soeken}@epfl.ch

Luca Amarú Antun Domic
Synopsys Inc., Design Group
Sunnyvale, California, USA
{luca.amaru, antun.domic}@synopsys.com

Abstract

A representation *of a Boolean function is* canonical *if, given a variable order, only one instance of the representation is possible for the function. A* computation *is* canonical *if the result depends only on the Boolean function and a variable order, and does not depend on how the function is represented and how the computation is implemented.*

In the context of Boolean satisfiability (*SAT*)*, canonicity of the computation implies that the result* (*a satisfying assignment for satisfiable instances and an abstraction of the unsat core for unsatisfiable instances*) *does not depend on the functional representation and the SAT solver used.*

This paper shows that SAT-based computations can be made canonical, even though the SAT solver is not using a canonical data structure. This brings advantages in EDA applications, such as ISOP computation, counter-example minimization, etc, where the uniqueness of solutions and/or improved quality of results justify a runtime overhead.

1. Introduction

Canonical representations of Boolean functions, such as truth tables, binary decision diagrams (BDDs) [5], or zero-suppressed decision diagrams (ZDDs) [14], represent functions in a unique way, that is, for a given function with a given variable order, only one representation is possible. As a consequence, if a canonical representation is constructed for two functions, they are equivalent if and only if the representations are isomorphic.

Another consequence of using a canonical representation is that it allows for results of Boolean operations to be canonical, that is, unique for a given function and a given variable order. For example, when an irredundant sum of products (ISOP) is computed from the BDD by the Minato-Morreale algorithm [13][6], the resulting ISOP is unique and enjoys some other interesting properties [15].

The present paper focuses on SAT-based computations and shows that the artifacts produced by the solver can be made canonical for both satisfiable or unsatisfiable calls. This is achieved by introducing an additional computational layer between the SAT solver and the calling application.

It is somewhat non-intuitive that the canonicity of the SAT-based computations is achieved *without* building a canonical data structure, such as a BDD or a ZDD. Nevertheless, Boolean functions represented using Boolean circuits or conjunctive normal forms (CNFs), which are not canonical, can yield canonical SAT-based computations if a variable ordering is provided to the solver.

In practical applications of Boolean satisfiability (SAT), a SAT solver, in addition to the answer "yes" (satisfiable) or "no" (unsatisfiable), can be made to return other artifacts.

In the case of a satisfiable SAT instance, a witness represented as a variable assignment can be returned. This witness can be made unique (canonical) by selecting it to be the lexicographically smallest among all possible witnesses using a variable order. Such an algorithm, LEXSAT, was introduced by Knuth [8] and improved by Petkovska et al. [19]. In this work, we review LEXSAT and combine it with an associated novel algorithm called LEXUNSAT. This algorithm derives an assignment of variables that can be seen as a unique witness for unsatisfiable SAT calls.

To understand how unsatisfiable calls are made canonical, consider an incremental SAT run when a set of assumptions (temporary unit clauses) is passed to the solver. If the problem is unsatisfiable, the solver can return a sufficient subset of assumptions using procedure *analyze_final* [7]. This subset is a type of abstraction of the unsat core; the unsat core is expressed in terms of the original assumptions.

The main contribution of this paper is introducing efficient implementation of LEXUNSAT and demonstrating its usefulness in several practical applications. LEXUNSAT makes the computed subset of assumptions canonical similarly to how LEXSAT produces a canonical witness.

The LEXUNSAT algorithm was first mentioned in [21], without emphasizing its canonicity. Under a different name, LEXUNSAT was used in the context of a practical application [23] and led to superior results. The present paper focuses on different applications and on the importance of both LEXSAT and LEXUNSAT for making some SAT-based computations canonical.

We note that having a variable order plays a central role in making both BDD-based and SAT-based computations canonical. In the case of canonical reduced ordered BDDs, all variables have to be ordered; otherwise a BDD cannot be constructed and used. In the case of SAT, only a subset of all problem variables (such as inputs of a node) have to be ordered. This leads to canonical results without imposing substantial restrictions on the variable decision heuristics used in a SAT solver, which could slow it down.

The rest of the paper is organized as follows. Section 2 contains necessary background. Section 3 gives an overview of the algorithm. Section 4 discusses practical applications. Section 5 presents experimental results. Section 6 concludes the paper.

2. Background

2.1 Boolean function

In this paper, *function* refers to a completely specified Boolean function $f(X)$: $B^n \rightarrow B$, $B = \{0,1\}$. The *support* of function f is the set of variables X, which influence the output value of f. The support size is denoted by $|X|$. A *minterm* of the function is an assignment of its inputs, for which the function evaluates to 1.

2.2 Boolean network

A *Boolean network* (or *circuit*) is a directed acyclic graph (DAG) with nodes corresponding to Boolean functions and edges corresponding to connections between the nodes.

A node n may have zero or more *fanins*, i.e. nodes driving n, and zero or more *fanouts*, i.e. nodes driven by n. The primary inputs (PIs) are nodes without fanins. The primary outputs (POs) are a subset of nodes of the network, connecting it to the environment. A *transitive fanin (fanout) cone* (TFI/TFO) of a node is a subset of nodes of the network, which are reachable through the fanin (fanout) edges of the node. The *TFO support* of a node is the set of primary outputs reachable through the fanouts of the node.

2.3 Boolean satisfiability

A *satisfiability problem* (SAT) is a decision problem that takes a propositional formula of a Boolean function and answers the question whether the formula is satisfiable, that is, whether the Boolean function is not a constant 0. The formula is *satisfiable* (SAT) if there is an assignment of variables such that the formula evaluates to 1. Otherwise, the formula is *unsatisfiable* (UNSAT). A software program that solves SAT problems is called a *SAT solver*.

2.4 Conjunctive Normal Form

To use the SAT solver, important aspects of the problem are encoded using Boolean *variables*. Presence or absence of a given aspect of the problem is represented by a positive or negative *literal* of the variable. A disjunction of literals is called a *clause*. A conjunction of clauses is called a conjunctive normal form (CNF), or a *SAT instance*. The CNF is loaded into the SAT solver and manipulated by it to determine whether the CNF is *satisfiable* or *unsatisfiable*.

If the instance is unsatisfiable, it is often possible to remove some clauses from it without making it satisfiable. A subset of the original clauses of the CNF, for which the instance is still unsatisfiable, is called an *unsat core*.

In logic synthesis and formal verification, which deal with hardware designs, a large portion of the CNF is derived from the circuit representation of the design, while the remaining portions may be derived by considering other aspects of the problem, such as timing constraints, user specified cost-functions, etc.

2.5 Incremental SAT

Modern SAT solvers, such as MiniSAT [7] and Glucose [1], offer a mode of solving that allows executing multiple SAT calls without restarting the SAT solver. In this mode, the CNF first loaded into the SAT solver is reused with the possibility of adding new clauses or deleting old ones between the calls. To do this efficiently, the SAT solver accepts assumptions, which are single-literal clauses holding for one call to the SAT solver. The process of determining the satisfiability of a SAT instance under assumptions is called *incremental SAT solving*.

2.6 Witnesses and sufficient assumption subsets

When a SAT problem is satisfiable, the solver returns a counter-example, or a *witness*, which is useful in practice to reproduce a particular situation and/or understand the reason of a malfunction in the design. For example, a witness for the CNF representing a Boolean function shows one assignment of values of the input variables for which the function takes value 1.

When an incremental SAT problem is unsatisfiable, a modern SAT solver, such as MiniSAT [7] or Glucose [1], can use a procedure called *analyze_final* to derive a subset of assumptions that are sufficient to make the instance unsatisfiable. This *sufficient assumption subset* can be seen as an abstraction of the unsat core, that is, the unsat core expressed in terms of the assumptions given to the solver.

For example, if a satisfying assignment of a CNF representing a Boolean function is given as a set of assumptions to a SAT solver whose CNF represents the complement of the function, the incremental run is unsatifiable because a minterm of the function cannot be a minterm of its complement. Moreover, the resulting sufficient assumption subset returned by the SAT solver in this case represents an implicant of the function computed by expanding the given minterm.

The implementation of *analyze_final* is efficient because it involves only one additional call to the conflict analysis procedure of the SAT solver to determine the result of the root-level (final) conflict in terms of the root-level (assumption-based) decisions. However, the resulting subset is not guaranteed to be minimal, that is, it may be possible to remove some assumptions from the subset while preserving the unsatisfiability of the incremental SAT call. Thus, in the example above the implicant may not be prime.

2.7 Activation literals

In many practical applications, assumptions given to the incremental SAT solver are *activation literals*, that is, literals of Boolean variables activating some constraints in the problem formulation. Activation literals are assumed to be active-high, that is, when an activation literal is positive, the constraint is activated. For example, if a certain node appears in the support of another node, the corresponding assumption is a positive activation literal.

If assumptions given to the SAT solver are *not* activation literals, a set of new Boolean variables can be introduced to create activation literals. A new two-literal clause is added for each activation literal. The clause is such that, when a new variable is 1, the corresponding assumption is implied; otherwise, when it is 0, there is no implication.

3. Proposed algorithms

This section introduces two algorithms, LEXSAT and LEXUNSAT. They are introduced first using a simple pseudo-code, followed by a more efficient formulation.

3.1 Simple LEXSAT

The pseudo-code of LEXSAT is given in Algorithm 1. It takes a CNF of a function F loaded into the SAT solver, and an array of literals A. Initially A contains all negative literals in the given variable order. Upon termination, A contains the LEXSAT assignment. This assignment is the smallest satisfying assignment of F in the following sense: if we look at each literal as a binary digit in the order of their appearance in A and consider an integer number formed by these digits, the resulting number is the smallest among all numbers created for any satisfying assignment.

```
array LEXSAT( cnf F, array A )  {
    for ( i = 0; i < |A|; i++ ) { // consider literals in the given order
        if ( F is UNSAT under assumptions A[0] through A[i])
            invert the polarity of literal A[i] to be positive;
    }
    return A;
}
```

Algorithm 1: A naïve implementation of LEXSAT.

Algorithm 1 considers literals in the given order. It checks if a satisfying assignment with the *negative* literal exists. If so, the literal remains negative (stands for 0 in the satisfying assignment). Only if a satisfying assignment with the negative literal does not exist, the literal is changed to be positive (stands for 1 in the satisfying assignment). The resulting assignment is the lexicographically smallest, where the MSB (LSB) is the first (last) literal in the array A.

It should be noted that Algorithm 1 assumes that F is satisfiable. If F is unsatisfiable, the algorithm returns an invalid satisfying assignment.

3.2 Simple LEXUNSAT

As shown in Section 2.7, it can be assumed that the assumptions given to the SAT solver are active-high activation literals, that is, a constraint is activated when a corresponding assumption is a positive literal.

The pseudo-code of LEXUNSAT, given in Algorithm 2, takes a CNF of F loaded into the SAT solver, and an array of assumptions A in the given order. Initially, the array contains all *positive* literals. Upon termination, the array contains the lexicographically smallest sufficient assumption subset. This subset is the smallest for function F in the following sense: if we look at each activation literal as a binary digit in the order of their appearance in A and consider an integer number formed by these digits, this number is the smallest among all similar numbers created by considering other unsatisfiable subsets of assumptions.

```
array LEXUNSAT( cnf F, array A )  {
    for ( i = 0; i < |A|; i++ ) { // consider literals in the given order
        if ( F is UNSAT under assumptions in A excluding A[i] )
            invert the polarity of A[i] to be negative;
    }
    return A;
}
```

Algorithm 2: A naïve implementation of LEXUNSAT.

It is easy to see that the resulting subset is minimal. The fact that the subset is lexicographically smallest implies that we cannot replace a positive literal by a negative one without making the problem satisfiable.

Algorithm 2 considers assumptions one at a time in the order given. It checks if F is unsatisfiable with one assumption skipped (the corresponding activation literal complemented). If so, the assumption literal is made negative and excluded from future checks. If F is satisfiable with assumption i excluded, the $A[i]$ is kept positive. Thus, the resulting polarity of assumption literals is the lexically smallest, where the MSB is the first literal in A.

Algorithm 2 assumes that F is unsatisfiable. If F is satisfiable, the algorithm returns an invalid subset.

3.3 Comparing LEXSAT and LEXUNSAT

Comparing the pseudo-code of Algorithms 1 and 2 shows that they are very similar and can be derived from each other by the following modifications:

- initialize array A with all negative (or positive) literals;
- check satisfiability of the formula with literals up to the given one (or with all literals);
- if the formula is UNSAT, complement the literal, making it positive (or negative).

LEXSAT and LEXUNSAT work in different directions. LEXSAT starts with the *smallest* (all-0) assignment and works its way up to the smallest valid satisfying assignment. LEXUNSAT starts with the *largest* (all-1) subset of assumptions and works its way down to find the smallest valid subset that makes the problem unsatisfiable. Both algorithms use the variable order by setting the MSB to 0 whenever possible before moving to less significant digits.

3.4 Efficient implementation

Since the algorithms are similar, only LEXUNSAT is considered in this section.

The procedure takes a CNF representation of F and the ordered array of assumptions A as a set of positive activation literals. It returns a minimized version of A.

```
array LEXUNSAT( cnf F, array A )
{
    if ( |A| == 1 ) {  // there is only one assumption in A
        if ( F is UNSAT with A as a negative literal )
            return A as negative literal; // this assumption is not needed
        else
            return A as positive literal; // this assumption is needed
    }
    // divide assumptions into two parts
    A_msb = entries in A with index in range [0; |A|/2]
    A_lsb = entries in A with index in range [|A|/2+1; |A|-1]

    // solve the problem for the MSB part
    assume A_lsb as positive literals;
    B = LEXUNSAT( F, A_msb as positive literals );
    unassume A_lsb as positive literals;

    // solve the problem for the LSB part
    assume literals in B;
    C = LEXUNSAT( F, A_lsb as positive literals );
    unassume literals in B;

    return literals in B followed by literals in C;
}
```

Algorithm 3: An efficient implementation of LEXUNSAT.

The implementation uses a divide-and-conquer strategy for minimizing the subset of assumptions. It divides the array A into two parts, the MSB part and the LSB part.

First, the algorithm assumes the LSB assumptions in the solver and calls itself recursively for the MSB part. Then, it reverses the order of parts, and calls itself recursively for the resulting LSB part. Finally, the subsets computed by the two recursive calls are combined.

3.5 Complexity considerations

In this section, we compare the complexities of the following three procedures:

- the non-canonical implementation,
- the naïve canonical implementation,
- the efficient canonical implementation.

The complexity of a procedure is expressed as the number of incremental SAT calls needed to implement it. The following observations can be made.

3.5.1 Non-canonical implementation

The non-canonical implementation has complexity O(1) because it takes only one SAT call. For example, when we run the solver, we either get a satisfying assignment or prove it unsatisfiable and use *analyze_final* to compute a sufficient assumption subset.

3.5.2 Simple canonical implementation

The naïve implementation in Sections 3.2 and 3.3 has complexity O(N), where N is the number of variables in the ordered subset of variables (for LEXSAT) or the number of assumptions given to the SAT solver (for LEXUNSAT). The complexity is O(N) because we need to have a separate call to the SAT solver for each variable (assumption).

3.5.3 Efficient canonical implementation

The efficient implementation of the algorithms in Section 3.4 has complexity O(max(log2(N), M)) where N is the number of variables (assumptions) and M is the number of positive literals in the resulting assignment (subset).

If the resulting satisfiable assignment (unsatisfiable subset) is sparse, that is, contains only a few positive literals, the complexity is logarithmic in the number of variables, because the efficient implementation uses binary search as a divide-and-conquer strategy.

If the resulting assignment (or subset) is dense, that is, contains many positive literals, the complexity deteriorates to linear in the number of these literals, because for each of them, we need a SAT call to confirm that it is needed.

3.6 Practical considerations

In practical applications, the simple way of using the SAT solver is the fastest one but it is not canonical and the result returned is not guaranteed to be minimal. For example, if we use *analyze_final* to expend minterm into a cube, the resulting cube is not guaranteed to be prime.

The naïve canonical implementation discussed above guarantees uniqueness and minimality but it is often much slower than the non-canonical implementation.

The efficient canonical implementation is the best one for using in practical applications. It guarantees uniqueness and minimality while keeping the runtime reasonable.

4. Applications

Listed below are practical applications of LEXSAT and LEXUNSAT in the synthesis and verification areas. In each case, their use is to make the solution unique (independent of the CNF computation and the SAT solver) and/or to improve the quality of results (by guaranteeing the minimality of the solution).

- SAT-based computation of irredundant sum-of-products (ISOP) [20] (see also Section 5.1). In [20], efficient LEXSAT and naïve LEXUNSAT are used, without identifying that the algorithm is LEXUNSAT.
- LEXUNSAT can be used to efficiently compute minimal subsets of Boolean divisors in various SAT-based formulations of re-substitution [9], functional decomposition [10][11], and ECO [22].
- LEXSAT can be used to enumerate different satisfying assignments of a Boolean formula; e.g. for finding diverse witnesses falsifying safety properties in Bounded Model Checking (BMC) [4], which is currently done using other methods [17].
- LEXUNSAT can be used to minimize the number of care bits in the counter-examples, which helps debugging hardware designs (see Section 5.2).
- LEXSAT has been used in SMT solving [18].

An overview of other applications of lexicographical algorithms can be found in [12].

5. Experimental results

The proposed algorithms were implemented in C and evaluated in two practical applications.

5.1 Computing canonical ISOP

In this experiment, we compute ISOP for single-output functions extracted from design blocks considered for collapsing (deriving the SOP representation from the original circuit structure, followed by deriving a new circuit structure by factoring the resulting SOP).

The following ISOP computations have been compared:

- **BDDovo**: BDD-based ISOP [13][6] for the original variable order *without* dynamic variable reordering,
- **BDDdvr**: BDD-based ISOP [13][6] for the original variable order *with* dynamic variable reordering,
- **SATovo**: SAT-based ISOP with LEXSAT and LEXUNSAT for the *original* variable order,
- **SATrvo**: SAT-based ISOP with LEXSAT and LEXUNSAT for a *random* variable order.

Table 1 shows the problem statistics (the number of inputs, outputs, AIG nodes, logic levels, and the number of cubes in the ISOP) and compares the runtimes, in milliseconds, averaged over 10 runs for each algorithm measured on a Intel Xeon E5-2698 v4 CPU @ 2.20 GHz. A dash in the algorithm means that the algorithm timed out after 5 minutes. The shaded cells represent the best runtimes for each example.

Although all ISOPs are canonical, they are not the same due to using different variable orders. However, for these benchmarks, all methods produced ISOPs with the same cube count listed in column “Cubes”, except one case (*test11*), for which SAT-based computation with a random order resulted in the ISOP with four additional cubes.

The following observations can be made from Table 1:

- The quality of results (the cube count) and average runtime of the BDD-based ISOP and the SAT-based ISOP are close, with the SAT-based ISOP being on average about 70% slower, compared to BDDs, for those test-cases where both computations finished.
- SAT-based computation is more robust and finishes on all examples where BDD-based fails. For case *test14*, all algorithms timed out.
- SAT-based computation is less variable-order dependent, except the first case (*test1*) whose runtime increases substantially when a random variable order is used. We are investigating the reason for this increase.

It should be noted that the results listed in Table 1 differ from those appearing in [20] in that they use the efficient implementation of LEXUNSAT to achieve canonicity. For a fair comparison between the algorithms, only one single-output function (the function with the largest support set and AIG node count) is considered for each benchmark.

5.2 Minimizing counter-examples

Many formal verification algorithms return a counter-example to demonstrate a failure of a safety property or a difference in the behavior of two designs that are expected to be equivalent. Because the counter-example is used by the designer to find the reason of the failure, it is often desirable to minimize the number of care-bits in it; the remaining don't-care bits can be set to any value without affecting the failure.

The counter-example minimization problem can be solved by a call to LEXUNSAT. The SAT instance is created by unfolding of the design to the failure depth, asserting that the property *holds*, and using assumptions to represent the values of primary inputs in each timeframe, according to the counter-example. The resulting problem is UNSAT because the counter-example should *fail* the property. A sufficient assumption subset produced by LEXUNSAT yields a minimal set of care-bits of the counter-example.

The first section of Table 2 lists statistics (the number of primary inputs, flip-flops, and AIG nodes) for several single-output safely model checking benchmarks from the recent Hardware Model Checking Competitions [3]. The second section of Table 2 lists the statistics of counter-examples produced by BMC [4] applied to the AIG representations of the benchmarks preprocessed by a synthesis script. The statistics include the zero-based number of the timeframe, in which the property failed, and the AIG node count in the unfolding up to this timeframe.

The third section of Table 2 contains the number of bits in the original counter-example (column "Init") and the number of care-bits produced by the following algorithms:

- **Struct**: Structural priority-based counter-example minimization without SAT [16].
- **SATaf**: SAT-based minimization, which proves the instance UNSAT and uses fast heuristic procedure *analyze_final*, resulting in a non-canonical subset of care-bits that is not guaranteed to be minimal.
- **SATlu**: SAT-based minimization, which uses one call to LEXUNSAT to derive a canonical minimal subset of care-bits for the given variable order.

The variable order given to LEXUNSAT is the original order of inputs in each time frame, considered from the initial one to the one where the failure happens. Reversing this order tends to produce different subsets, which can be up to 10% larger or smaller than the one reported Table 2.

The last section of Table 2 compares the runtimes of the two SAT-based algorithms on an Intel i7-4600U CPU @ 2.1GHz. The comparison shows that LEXUNSAT is more time-consuming but leads to 30% fewer care-bits than the procedure *analyze_final*. We believe that the improvement in designer productivity afforded by smaller counter-examples justifies the increase in runtime.

6. Conclusions

The two popular computation engines, BDDs and SAT, offer complementary ways of solving Boolean problems, which can be seen as trading space for time. The canonicity of BDDs makes them easy to use but difficult to construct without exceeding memory limits, in particular, for some practical functions, such as multipliers. The non-canonicity of SAT makes SAT instances easy to construct but difficult to solve, due to exceeding time limits for hard problems.

When BDDs are used, canonical representation results in canonical computation. This paper shows, for the first time, that without building a canonical representation, SAT-based computations can be made canonical for both satisfiable and unsatisfiable instances, at the cost of increased runtime.

The canonicity of computation in the case of both BDDs and SAT is achieved by fixing a variable order. For BDDs, the ordering of *all* variables is necessary. In the case of SAT, the order has to be fixed only for variables used to express the results of the computation, which also helps the SAT solver perform well on difficult problem instances.

We have shown that, unlike BDDs, SAT solvers are practically insensitive to variations in the variable order. This allows for exploiting variable orders in LEXUNSAT to achieve optimization objectives. For example, ordering less important variables first decreases the likelihood of their appearance in the solution. However, the exploration of the impact of a variable order on the quality of solutions is deferred to future work.

Also, unlike BDDs, SAT-based computations can relax canonicity by using regular SAT calls, instead of LEXSAT and LEXUNSAT, in order to improve the runtime when canonicity is not required.

In summary, the canonicity of SAT-based computations has several practical advantages, such as uniqueness of solutions, reproducibility of runs for different SAT solvers, good performance for various variable orders, and improved quality of results.

7. REFERENCES

[1] G. Audemard and L. Simon, SAT solver Glucose 3.0 (2013), http://www.labri.fr/perso/lsimon/glucose/

[2] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[3] Hardware Model Checking Competitions. fmv.jku.at/hwmcc

[4] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs", *Proc. DAC'99*, pp. 317-320.

[5] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation”. *IEEE Trans. Computers*, Vol. C-35 (8), August 1986, pp. 677-691.

[6] O. Coudert, J. C. Madre, H. Fraisse, and H. Touati, ”Implicit prime cover computation: An overview”, *Proc. SASIMI '93*.

[7] N. Een and N. Sörensson, “An extensible SAT-solver”, *Proc. SAT’03*, LNCS 2919, pp. 502-518.

[8] D. E. Knuth. *Volume 4, Fascicle 6: Satisfiability*, *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., 2015.

[9] C.-C. Lee, J.-H. R. Jiang, C.-Y. (R.) Huang, and A. Mishchenko, “Scalable exploration of functional dependency by interpolation and incremental SAT solving”, *Proc. ICCAD'07*.

[10] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung, “Bi-decomposing large Boolean functions via interpolation and satisfiability solving”, *Proc. DAC'08*, pp. 636-641.

[11] H.-P. Lin, J.-H. R. Jiang, and R.-R. Lee, “To SAT or not to SAT: Ashenhurst decomposition in a large scale”, *Proc. ICCAD'08*, pp. 32-37.

[12] J. Marques-Silva, J. Argelich, A. Graca, and I. Lynce, “Boolean lexicographic optimization: algorithms and applications”, *Annals of Mathematics and Artificial Intelligence*, Vol. 62(3), pp. 317-343, May 2011.

[13] S. Minato, “Fast generation of irredundant sum-of-products forms from binary decision diagrams”, *Proc. SASIMI'92*, pp. 64-73.

[14] S. Minato, “Zero-suppressed BDDs for set manipulation in combinatorial problems”. *Proc. DAC ‘93*, pp. 272-277.

[15] S. Minato and G. DeMicheli. “Finding all simple disjunctive decompositions using irredundant sum-of-products forms”, *Proc. ICCAD'98*, pp. 111-117.

[16] A. Mishchenko, N. Een, and R. Brayton, "A toolbox for counter-example analysis and optimization", *Proc. IWLS'13*.

[17] A. Nadel, “Generating diverse solutions in SAT”, *Proc. SAT’11*, pp. 287-301.

[18] A. Nadel and V. Ryvchin, “Bit-vector optimization”, *Proc. TACAS’16*, pp. 851-67.

[19] A. Petkovska, A. Mishchenko, M. Soeken, G. De Micheli, R. Brayton, and P. Ienne, "Fast generation of lexicographic satisfiable assignments: Enabling canonicity in SAT-based applications", *Proc. ICCAD'16*.

[20] A. Petkovska, A. Mishchenko, D. Novo, M. Owaida, and P. Ienne. “Progressive generation of canonical irredundant sums of products using a SAT solver”. In: *Advanced Logic Synthesis*. Springer, 2018, pp. 169-188.

[21] The reference is removed for blind review.

[22] B.-H. Wu, C.-J. Yang, C.-Y. (R.) Huang, and J.-H. R. Jiang. ”A robust functional ECO engine by SAT proof minimization and interpolation techniques”. *Proc. ICCAD'10*, pp. 729-734.

[23] The paper is currently under review.

Table 1: Comparing BDD-based and SAT-based ISOP computation algorithms.

Testcase	Problem statistics					BDDovo	BDDdvr	SATovo	SATrvo
	PIs	POs	ANDs	Levels	Cubes	T,msec	T,msec	T,msec	T,msec
test01	160	1	405	21	105	11.5	20.1	89.5	22141.8
test02	104	1	321	12	97	8.7	-	28.9	29.9
test03	70	1	215	12	64	5.7	10.8	14.3	13.9
test04	135	1	381	14	128	-	92.0	65.3	64.8
test05	148	1	348	29	232	13.6	22.8	240.3	298.6
test06	85	1	228	26	60	10.0	15.4	32.2	36.3
test07	50	1	147	16	95	-	175.7	22.8	27.2
test08	60	1	187	11	56	6.5	11.6	11.3	12.2
test09	94	1	256	20	16	6.0	9.7	12.2	12.4
test10	105	1	136	14	67	-	-	18.5	17.4
test11	107	1	282	22	110	75.0	60.9	64.4	73.3
test12	31	1	70	13	270	8.2	9.6	32.3	37.0
test13	119	1	292	41	56	-	74.2	19.7	21.4
test14	6246	1	173954	387	-	-	-	-	-
test15	10	1	15	5	16	6.3	7.0	2.9	2.9
test16	10	1	20	9	6	5.5	7.6	2.1	2.2
test17	8	1	18	9	5	5.4	7.6	1.9	1.8
test18	4110	1	12288	26	4097	1716.9	7462.7	53335.5	67215.2
Geomean						1.000	1.400	1.593	1.702

Table 2: Comparing structural and SAT-based counter-example minimization algorithms.

Testcase	Original property cone			Time frames		The number of care-bits				Runtime, sec	
	PIs	FFs	ANDs	Depth	ANDs	Init	Struct	SATaf	SATlu	Taf	Tlu
6s41	19	959	3274	73	112493	1406	173	197	90	0.01	5.52
6s134	36	571	2095	168	22511	6084	944	295	240	0.02	1.08
6s162	73	156	1244	73	72424	5402	602	565	401	0.01	8.83
6s199	144	1660	13666	49	178258	7200	459	229	184	0.01	9.15
bob12s03	617	5174	32335	12	41643	8021	618	97	57	0.01	0.59
bobtuttt	2807	111	9482	27	249240	78596	308	216	207	0.02	22.80
Geomean								1.00	0.70		