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 irredundant sum of product (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 using a variable order. Such an algorithm, LEXSAT, was introduced by Knuth [8] and Nadel and Ryvchin [18], 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.

In this paper, 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 the presentation below, 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.

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 unsatisfiable 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. 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.

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 \text{LEXUNSAT}( \text{cnf } F, \text{array } A ) \{
\text{if} ( \text{| } A \text{| } = \text{1} ) \ { // \text{there is only one assumption in } A
\text{if} ( \text{F} \text{ is UNSAT with } A \text{ as a negative literal })
\text{return } A \text{ as negative literal}; // \text{this assumption is needed}
\text{else}
\text{return } A \text{ as positive literal; // this assumption is needed}
}
\text{// divide assumptions into two parts}
A_{\text{msb}} \text{ } \text{= entries in } A \text{ with index in range [0; } \text{|} A \text{|}/2\text{]};
A_{\text{lsb}} \text{ } \text{= entries in } A \text{ with index in range [}\text{|} A \text{|}/2+1; \text{|} A \text{|}/2\text{]};
\text{// solve the problem for the MSB part}
\text{assume } A_{\text{lsb}} \text{ as positive literals;}
B \text{ } \text{= LEXUNSAT( } F, \text{A}_{\text{msb}} \text{ as positive literals );}
\text{unassume } A_{\text{lsb}} \text{ as positive literals;}
\text{// solve the problem for the LSB part}
\text{assume literals in } B;
C \text{ } \text{= LEXUNSAT( } F, \text{A}_{\text{lsb}} \text{ as positive literals );}
\text{unassume literals in } B;
\text{return literals in } B \text{ followed by literals in } C;
\}
\text{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 $O(\max(\log(2(N), M)))$ 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 independently proposed in the context of SMT solving [18]. An overview of other applications of lexicographical algorithms can be found in [12].

### 5. EXPERIMENTS

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 an 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 with a random variable order.

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 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 unsatisfiable 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.

ACKNOWLEDGMENTS

This work was supported in part by SRC contract 2710.001 “SAT-based methods for scalable synthesis and verification”, NSF/NSA grant “Enhanced equivalence checking in cryptoanalytic applications”, and by the Swiss National Science Foundation (200021-169084 MAJesty).

REFERENCES

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

<table>
<thead>
<tr>
<th>Testcase</th>
<th>Problem statistics</th>
<th>BDDovo</th>
<th>BDDdvr</th>
<th>SATovo</th>
<th>SATrvo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PIs</td>
<td>POs</td>
<td>ANDs</td>
<td>Levels</td>
<td>Cubes</td>
</tr>
<tr>
<td>test01</td>
<td>160</td>
<td>1</td>
<td>405</td>
<td>21</td>
<td>105</td>
</tr>
<tr>
<td>test02</td>
<td>104</td>
<td>1</td>
<td>321</td>
<td>12</td>
<td>97</td>
</tr>
<tr>
<td>test03</td>
<td>70</td>
<td>1</td>
<td>215</td>
<td>12</td>
<td>64</td>
</tr>
<tr>
<td>test04</td>
<td>135</td>
<td>1</td>
<td>381</td>
<td>14</td>
<td>128</td>
</tr>
<tr>
<td>test05</td>
<td>148</td>
<td>1</td>
<td>348</td>
<td>29</td>
<td>232</td>
</tr>
<tr>
<td>test06</td>
<td>85</td>
<td>1</td>
<td>228</td>
<td>26</td>
<td>60</td>
</tr>
<tr>
<td>test07</td>
<td>50</td>
<td>1</td>
<td>147</td>
<td>16</td>
<td>95</td>
</tr>
<tr>
<td>test08</td>
<td>60</td>
<td>1</td>
<td>187</td>
<td>11</td>
<td>56</td>
</tr>
<tr>
<td>test09</td>
<td>94</td>
<td>1</td>
<td>256</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>test10</td>
<td>105</td>
<td>1</td>
<td>136</td>
<td>14</td>
<td>67</td>
</tr>
<tr>
<td>test11</td>
<td>107</td>
<td>1</td>
<td>282</td>
<td>22</td>
<td>110</td>
</tr>
<tr>
<td>test12</td>
<td>31</td>
<td>1</td>
<td>70</td>
<td>13</td>
<td>270</td>
</tr>
<tr>
<td>test13</td>
<td>119</td>
<td>1</td>
<td>292</td>
<td>41</td>
<td>56</td>
</tr>
<tr>
<td>test14</td>
<td>6246</td>
<td>1</td>
<td>173954</td>
<td>387</td>
<td>-</td>
</tr>
<tr>
<td>test15</td>
<td>10</td>
<td>1</td>
<td>15</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>test16</td>
<td>10</td>
<td>1</td>
<td>20</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>test17</td>
<td>8</td>
<td>1</td>
<td>18</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>test18</td>
<td>4110</td>
<td>1</td>
<td>12288</td>
<td>26</td>
<td>4097</td>
</tr>
</tbody>
</table>

Geomean

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

<table>
<thead>
<tr>
<th>Testcase</th>
<th>Original property cone</th>
<th>Time frames</th>
<th>The number of care-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pls</td>
<td>FFs</td>
<td>ANDs</td>
</tr>
<tr>
<td>test01</td>
<td>160</td>
<td>1</td>
<td>321</td>
</tr>
<tr>
<td>test02</td>
<td>104</td>
<td>1</td>
<td>215</td>
</tr>
<tr>
<td>test03</td>
<td>70</td>
<td>1</td>
<td>215</td>
</tr>
<tr>
<td>test04</td>
<td>135</td>
<td>1</td>
<td>381</td>
</tr>
<tr>
<td>test05</td>
<td>148</td>
<td>1</td>
<td>348</td>
</tr>
<tr>
<td>test06</td>
<td>85</td>
<td>1</td>
<td>228</td>
</tr>
<tr>
<td>test07</td>
<td>50</td>
<td>1</td>
<td>147</td>
</tr>
<tr>
<td>test08</td>
<td>60</td>
<td>1</td>
<td>187</td>
</tr>
<tr>
<td>test09</td>
<td>94</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>test10</td>
<td>105</td>
<td>1</td>
<td>136</td>
</tr>
<tr>
<td>test11</td>
<td>107</td>
<td>1</td>
<td>282</td>
</tr>
<tr>
<td>test12</td>
<td>31</td>
<td>1</td>
<td>70</td>
</tr>
<tr>
<td>test13</td>
<td>119</td>
<td>1</td>
<td>292</td>
</tr>
<tr>
<td>test14</td>
<td>6246</td>
<td>1</td>
<td>173954</td>
</tr>
<tr>
<td>test15</td>
<td>10</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>test16</td>
<td>10</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>test17</td>
<td>8</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>test18</td>
<td>4110</td>
<td>1</td>
<td>12288</td>
</tr>
</tbody>
</table>

Geomean