# SAT-Based Algorithms for Logic Minimization[*]

Samir Sapra Michael Theobald Edmund Clarke

Carnegie Mellon University
Pittsburgh, PA

## Abstract

*This paper introduces a new method for two-level logic minimization. Unlike previous approaches, the new method uses a SAT solver as an underlying engine. While the overall minimization strategy of the new method is based on the operators as defined in* ESPRESSO-II*, our SAT-based implementation is significantly different. The new minimizer* SAT-ESPRESSO *was found to perform 5–20 times faster than* ESPRESSO-II *and 3–5 times faster than* BOOM *on a set of large examples.*

## 1. Introduction

Two-level logic minimization is an important problem of computer-aided digital design in several respects. While its original motivation has been to provide efficient circuit implementations of any logic function using just two levels of logic gates, the problem now also plays a central role in multi-level logic synthesis, state encoding, test generation, and power estimation [5, 9]. In addition, the significance of two-level logic minimization is not restricted to digital design; it has important applications, e.g. in reliability analysis and artificial intelligence [4].

This paper presents a new algorithm for two-level logic minimization that employs a satisfiability checker as an underlying engine. A Boolean satisfiability (SAT) checker is a program that checks whether a given Boolean formula in CNF (conjunctive normal form) is satisfiable or not. SAT is a well known NP-complete problem. However, in practice, SAT checkers perform very well and are able to solve 'real-world' formulae containing hundreds of thousands variables. The recent advances in satisfiability checkers [10, 13, 16] have had a major and positive impact on areas such as equivalence checking, processor verification, and model checking [8, 3, 2].

To the best knowledge of the authors, this paper is the first to present an approach to logic minimization that employs a SAT engine. We explore a SAT-based approach for *heuristic* minimization. State-of-the-art heuristic minimizers like ESPRESSO-II [12, 1] are used world-wide. The minimization strategies used by ESPRESSO-II almost always lead to near-minimum solutions in practice. However, for large problems (functions with over 100 input variables) ESPRESSO-II takes a long time to execute. SAT checkers, on the other hand, have recently become capable of handling comparatively huge numbers of variables. We therefore try to combine the strengths of ESPRESSO-II (quality of approximation) and SAT (speed on large problems) by adopting the same basic strategies as ESPRESSO-II but performing them efficiently by developing algorithms that use – appropriately adapted – SAT checkers.

Our new minimizer SAT-ESPRESSO was found to perform 5–20 times faster than ESPRESSO-II and 3–5 times faster than BOOM [6] on a set of large examples. BOOM is a recently developed heuristic minimizer that specializes in large examples.

The remainder of the paper is organized as follows. Section 2 introduces background material. Section 3 gives an overview of the new SAT-based method. Sections 4 through 6 introduce our new SAT-based algorithms for the ESPRESSO-II operators. Section 7 gives experimental results, and Section 8 gives conclusions.

## 2. Background

This section first reviews a number of definitions in logic synthesis. Then, two-level logic minimization algorithms are surveyed.

### 2.1. Basic Logic Synthesis Definitions

The following definitions are taken from Rudell [11], and standard textbooks [5, 9] with small modifications [14].

Let $\mathbb{B} := \{0, 1\}$ be the set of binary values. $\mathbb{B}^n$ can be modeled as a binary $n$-cube, and each element $e =$

[*]This research was supported by the National Science Foundation (NSF) under grants no. CCR-0121547 and CCR-0098072, by the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, by the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARO, ONR, NRL, NSF, SRC, the U.S. Government or any other entity.

$(e_1, \ldots, e_n) \in \mathbb{B}^n$ is called a **minterm**. Note that the well-known binary Boolean algebra is given by the the set $\mathbb{B}$ together with the operations $+$ (also called disjunction, sum, OR) and $\cdot$ (conjunction, product, AND).

A **Boolean function** $f$ of $n$ variables, $x_1, \ldots, x_n$, is a mapping $f : \mathbb{B}^n \to \{0, 1, *\}$. Here, the symbol $*$ denotes a *don't care* condition, i.e. the value of the function does not matter. Note that a minterm $(e_1, \ldots, e_n)$ indicates which values are assigned to the variables of a function, i.e. $x_1 = e_1, x_2 = e_2$, and so on.

The **ON-set** of a Boolean function $f$ is defined as the set of minterms for which the function has value 1. Similarly, the **OFF-set** and **DON'T-CARE-set** are defined as the sets of minterms for which the function has value 0 and *, respectively.

Boolean functions as defined above are often referred to as *single-output* Boolean functions. A *multi-output* Boolean function is a mapping $f : \mathbb{B}^n \to \{0, 1, *\}^m$. Note that each of the output functions $f_1, \ldots, f_m$ has its own ON-set, OFF-set, and DON'T-CARE-set associated with it. For the sake of simplicity of presentation, only single-output functions are considered in the remainder of this section. The presented algorithms in this paper can handle multi-output functions.

Each variable $x_i$ has two **literals** associated with it: an *uncomplemented* (or *positive*) literal $x_i$, and a *complemented* (or *negative*) literal $\overline{x}_i$ or $x'_i$. The literal $x_i$ ($\overline{x}_i$) represents a Boolean function which evaluates to 1 (0) for minterms with $e_i = 1$, and to 0 (1) for minterms with $e_i = 0$.

A **product** term is a Boolean product (AND) of literals. That is, a product evaluates to 1 for a minterm $e$, if each literal *included* in the product evaluates to 1 for the minterm $e$. Otherwise, the product evaluates to 0. In the former case, the product is said to *contain* minterm $e$. Note that each minterm corresponds to a product that only contains the given minterm. More specifically, the minterm $e = (e_1, \ldots, e_n)$ corresponds to the product $x_1^{e_1} \cdots x_n^{e_n}$, where $x_i^{e_i}$ denotes the positive (negative) literal of $x_i$ if $e_i = 1(0)$. For example, the minterm $e = (1, 0, 1)$ corresponds to the product $x_1\overline{x_2}x_3$, which is often used as a convenient abbreviation. Since a product corresponds to a set of adjacent minterms in the binary $n$-cube, a product is also referred to as a **cube**.

A cube $\alpha$ is *contained in* a cube $\beta$ ($\alpha \subseteq \beta$) if each minterm contained in $\alpha$ is also contained in $\beta$. The *intersection* of cubes $\alpha$ and $\beta$ ($\alpha \cap \beta$) is the uniquely defined cube which contains those minterms contained in both cubes. The *supercube* of cubes $\alpha$ and $\beta$, denoted *supercube($\alpha, \beta$)*, is the uniquely defined smallest cube that contains both cubes. For example, if $\alpha = x_1\overline{x}_2$, and $\beta = \overline{x}_1\overline{x}_2x_3$, then *supercube($\alpha, \beta$)* = $\overline{x_2}$. In general, to compute the supercube each literal must be considered. A literal is included in the supercube of two cubes if and only if it is included in both cubes. The supercube of a set of cubes is defined similarly.

A **sum-of-products** is a Boolean sum (OR) of products. That is, a sum-of-products evaluates to 1 for a given minterm if some product contains the minterm.

An **implicant** of a Boolean function is a cube which contains no minterm in the OFF-set. A **prime implicant** is an implicant contained in no other implicant of the function. An **essential prime implicant** is a prime implicant containing at least one ON-set minterm which is not contained in any other prime implicant.

A **cover** of a Boolean function is a set of implicants interpreted as a sum-of-products, which evaluates to 1 for all the minterms of the ON-set, and none of the OFF-set. We use the term *prime cover* to refer to a cover containing only prime implicants.

The **complement** of a Boolean function $f$ is denoted by $\overline{f}$, or $f'$, and evaluates to 1 (0) if $f$ evaluates to 0 (1).

## 2.2. Two-Level Logic Minimization

The *two-level logic minimization problem* is to find a cover for $f$ that minimizes a given cost function. In digital design, such a cover can be implemented as a minimum-cost sum-of-products (two-level) circuit. Here, the cost, or size, of a cover is often defined as the number of cubes in the cover. (Another popular cost function is the number of literals.)

The classic QUINE-MCCLUSKEY algorithm [7, 11] to solve the exact two-level minimization problem is based on the insight that the implicants in a minimum-cost cover can be restricted to prime implicants. The algorithm consists of two steps: $(i)$ generate the set of all prime implicants; and $(ii)$ select a minimum number of prime implicants such that each ON-set minterm is contained.

SCHERZO [4] is currently the state-of-the-art exact two-level logic minimization algorithm. Using *implicit* minimization techniques, i.e. using data structures (BDDs and ZBDDs) that facilitate the manipulation of a large number of objects simultaneously, SCHERZO is 10 to more than 100 times faster than the best previous minimization methods.

Since solving the exact two-level logic minimization problem involves computationally intractable problems, heuristic approaches have been developed as well. ESPRESSO-II [12, 1] is the state-of-the-art tool for *heuristic* two-level logic minimization. The output of ESPRESSO-II is a cover, which in practice is almost always near-minimum in cardinality. The tool is very efficient and is used worldwide. Recently, an alternative tool, called BOOM [6], that particularly addresses large problems has been introduced.

## 3. Heuristic Minimization Using SAT Checkers

Solving the two-level logic minimization problem can be computationally expensive. Hence, heuristic tools like ESPRESSO-II have been developed as powerful practical alternatives. While ESPRESSO-II almost always produces near-minimum solutions in practice, it takes a long time to solve large problems.

Our aim is to achieve high-quality approximations efficiently for large problems. We combine the strengths of ESPRESSO-II (quality of approximation) and SAT solvers (speed on large problems) by adopting the same basic strategies as ESPRESSO-II but by performing them efficiently using SAT-based algorithms.

ESPRESSO-II's strategies are implemented by various procedures called 'operators'. The major ESPRESSO-II operators are called EXPAND, IRREDUNDANT, REDUCE and ESSENTIALS (these are described in more detail in the following sub-section). ESPRESSO-II's run-time profile was analyzed for some large examples, taken from a recently-published benchmark suite [6]. Our analysis indicated that the major bottlenecks were REDUCE, ESSENTIALS and IRREDUNDANT (in that order). For the considered examples, the EXPAND operator was not a bottleneck and executed quickly even on large examples with 200 variables. Accordingly, our efforts were focused on developing SAT-based algorithms for the other three operators.

In the remainder of this section, we begin by giving background on ESPRESSO-II, since its basic minimization strategy is similar to the one that we use. We then develop SAT-based algorithms to implement the IRREDUNDANT, REDUCE and ESSENTIALS operators.

### 3.1. Background on ESPRESSO-II

ESPRESSO-II, developed in the early 1980s, is a very powerful tool for heuristic two-level logic minimization. The tool has been very successful, and the underlying ideas have also inspired tools for other domains, e.g. for a variety of problems in logic design [9] and asynchronous logic synthesis [15].

The input to ESPRESSO-II is the Boolean function to be minimized, specified in terms of its ON-set, OFF-set, and DON'T-CARE-set. Two of these sets are actually sufficient as the three sets partition $\mathbb{B}^n$. Each of the sets is specified in terms of an arbitrary set of implicants (e.g. all contained minterms, or possibly larger cubes), denoted $F^{ON}$, $F^{OFF}$, and $F^{DC}$, respectively. The set of implicants $F^{ON}$ represents an initial unoptimized cover, or solution. The output of ESPRESSO-II is a cover, which is in practice almost always near-minimum in cardinality.

ESPRESSO-II *iteratively* refines the cover by applying three operators in its main loop. This iteration continues until no further improvement is possible: *(i)* EXPAND enlarges each implicant of the current cover, in turn, into a prime implicant. *(ii)* IRREDUNDANT makes the current cover irredundant by deleting a maximal number of redundant implicants from the cover. *(iii)* REDUCE sets up a cover that is likely to be made smaller by the following EXPAND step. To achieve this goal, each cube in the current cover is maximally reduced, in turn, to a smaller cube such that the resulting set of cubes is still a cover.

ESPRESSO-II also employs additional operators, such as ESSENTIALS and LAST_GASP, which can be quite powerful. ESSENTIALS is used to identify all essential prime implicants before the main loop is entered, in order to simplify the covering problem. LAST_GASP is applied after the main loop is exited, to try to escape a suboptimal local minimum; if successful the main loop is entered again.

One key reason for the efficiency of ESPRESSO-II is the so-called *unate recursive paradigm*, i.e., to decompose operations recursively leading to efficiently solvable sub-operations on unate functions (i.e. functions that are unate in all of their variables). A function is *unate in a variable* if changing the value of that variable from 0 to 1 either never changes the function's value from 0 to 1 or never changes the function's value from 1 to 0. A function would *not* be unate in a variable if changing the value of that variable from 0 to 1 sometimes changed the function's value from 0 to 1 and sometimes changed it from 1 to 0 depending on the values of the remaining variables.

## 4. Reduce

This section presents new SAT-based implementations of the REDUCE operator. In particular, we describe three methods in order of increasing performance advantage over the operator implementation in ESPRESSO-II. Our best implementation outperforms ESPRESSO-II's REDUCE by more than a factor of 100 on many of our large examples.

The purpose of the REDUCE operator is to modify the current cover so that its cardinality may be improved by the following EXPAND. Each implicant in a given cover is *maximally reduced* in size, i.e. reduced to the smallest cube such that the resulting set of implicants is still a cover. The end result of REDUCE depends on the order in which implicants are processed. Various heuristics have been developed to sort a cover before reducing its implicants. In ESPRESSO-II, implicants are weighted and then sorted in descending order of weight so as to first process those that are large and overlap many other implicants. Our implementation reuses the heuristics adopted by ESPRESSO-II.

### 4.1. Method 1:

Let us now consider how to compute a maximally reduced cube. We are given a (sorted) cover $F$ and an implicant $\alpha \in F$. Reducing $\alpha$ to the cube $\widetilde{\alpha}$ results in a new set of cubes $G = (F - \{\alpha\}) \cup \{\widetilde{\alpha}\}$. The goal is to find the smallest cube $\widetilde{\alpha}$ that makes $G$ a cover. Any $\widetilde{\alpha}$ that makes $G$ a cover must contain all of the ON-set minterms of $\alpha$ that are not contained in any other implicant of $F$. The smallest $\widetilde{\alpha}$ that contains all of these minterms is simply their $supercube$ (by definition).

Equivalently, we are looking for the supercube of all satisfying assignments of the following formula:

$$\alpha \cdot \prod_{\substack{\beta \in F \\ \beta \neq \alpha}} \overline{\beta} \cdot \left( \sum_{\gamma \in F^{ON}} \gamma \right) \qquad (1)$$

```
MAXIMALLY-REDUCE-SIMPLE ($\alpha$, $F$, $F^{ON}$)
1 {
2   $\Phi \leftarrow$ toCNF $\left(\sum_{\gamma \in F^{ON}} \gamma\right)$
3   $\Psi \leftarrow \alpha \cdot \left(\prod_{\beta \in F, \beta \neq \alpha} \overline{\beta}\right) \cdot \Phi$
4   $S \leftarrow \emptyset$
5   while (SAT_CHECK ($\Psi$, &$assignment$) = SATISFIABLE)
6   {
7     $S \leftarrow S \cup assignment$
8     $\Psi \leftarrow \Psi \cdot \overline{assignment}$
9   }
10  return SUPERCUBE($S$);
11 }
```

**Figure 1. Maximally-Reduce-Simple**

```
MAXIMALLY-REDUCE-FAST ($\alpha$, $F$, $F^{ON}$)
1 {
2   $\Phi \leftarrow$ toCNF $\left(\sum_{\gamma \in F^{ON}} \gamma\right)$
3   $\Psi \leftarrow \alpha \cdot \left(\prod_{\beta \in F, \beta \neq \alpha} \overline{\beta}\right) \cdot \Phi$
4   $\widetilde{\alpha} \leftarrow 0$
5   while (SAT_CHECK ($\Psi$, &$assignment$) = SATISFIABLE)
6   {
7     $\widetilde{\alpha} \leftarrow$ SUPERCUBE($\widetilde{\alpha}, assignment$)
8     $\Psi \leftarrow \Psi \cdot \widetilde{\alpha}'$
9   }
10  return $\widetilde{\alpha}$;
11 }
```

**Figure 2. Maximally-Reduce-Fast**

In Formula (1), $\alpha$ is the cube to be reduced, $F$ denotes the current cover, and $F^{ON}$ denotes the given ON-set. The formula characterizes minterms that *must* be included in the reduced cube. Each such minterm *(i)* must be covered by $\alpha$, *(ii)* must not be covered by any other cube $\beta$ of the current cover $F$, and *(iii)* must be in the ON-set of the function.

Formula (1) will be fed to a SAT checker. To do so, the part of the formula $\left(\sum_{\gamma \in F^{ON}} \gamma\right)$ must be converted into CNF (conjunctive normal form).

To perform this conversion, we first introduce a new variable $v_\gamma$ for each product $\gamma$ in the sum-of-products. Then, for each $\gamma$, we express $(v_\gamma \leftrightarrow \gamma)$ in clause form. Finally, all the clauses so obtained are ANDed together with $\left(\sum_{\gamma \in F^{ON}} v_\gamma\right)$.

For example, if $\Psi = ab + \overline{b}c$, then we introduce the variables $v_{ab}$ and $v_{\overline{b}c}$ and form $\Psi_{CNF}$ as follows:

$$\begin{aligned}
\Psi_{CNF} &= (v_{ab} + v_{\overline{b}c}) \cdot (v_{ab} \leftrightarrow ab) \cdot (v_{\overline{b}c} \leftrightarrow \overline{b}c) \\
&= (v_{ab} + v_{\overline{b}c}) \cdot (v_{ab} + \overline{a} + \overline{b})(\overline{v_{ab}} + a)(\overline{v_{ab}} + b) \\
&\quad \cdot (v_{\overline{b}c} + b + \overline{c})(\overline{v_{\overline{b}c}} + \overline{b})(\overline{v_{\overline{b}c}} + c)
\end{aligned}$$

$\Psi_{CNF}$ is not equivalent to $\Psi$, but is satisfiable if and only if $\Psi$ is satisfiable.

As an optimization, from the summation part of Formula (1), we can exclude those $\gamma$ that are disjoint from $\alpha$. Disjointness of implicants can be computed efficiently using bitwise operators.

Finally, it remains to be ensured that $\widetilde{\alpha}$ does not intersect the OFF-set. This follows from the fact that the smallest cube that contains a set of minterms (i.e. their supercube) is a subset of any other cube that contains those minterms. Thus, $\widetilde{\alpha}$ is guaranteed to be a subset of $\alpha$, which itself does not intersect the OFF-set.

ESPRESSO-II computes maximally reduced cubes by using the aforementioned *unate recursive paradigm* (Section 3.1).

In contrast, our method is based on SAT-solving. Formula (1) suggests one simple approach — find all the satisfying assignments of (1) using a SAT checker, and then compute their supercube

This approach, called MAXIMALLY-REDUCE-SIMPLE, is shown in Figure 1. The function SAT_CHECK() takes a CNF formula as its first parameter and determines whether it is SATISFIABLE. If so, a satisfying assignment is returned by modifying the second parameter (passed-by-pointer). The function SUPERCUBE computes and returns the supercube of its argument cubes.

In each iteration of the while loop, it is determined if $\Psi$ is satisfiable. If it is, then a satisfying assignment is returned via $assignment$, $\Psi$ is modified to 'block' out $assignment$ (line 8), and $assignment$ is added to the collection $S$ of assignments found so far (line 7). Continuing in this way, eventually all satisfying assignments are found, and their supercube is computed and returned in line 10.

## 4.2. Method 2:

A modified version can be found that is often much faster than Method 1. The modified algorithm MAXIMALLY-REDUCE-FAST (Figure 2) maintains a 'running total' supercube $\widetilde{\alpha}$ of all the assignments found so far, rather than the set of satisfying assignments that has been discovered. Each time a satisfying assignment is found in the while loop of lines 5–9, Method 2 updates the supercube $\alpha$ and then blocks out the updated supercube $\widetilde{\alpha}$ from $\Psi$. This is in contrast to Method 1, which updates the *collection* S and blocks out each individual *assignment* (lines 7 and 8). In the end the algorithm simply returns the running total so far.

Intuitively, the modified algorithm limits the number of satisfying assignments that need to be found. In each iteration an assignment is found that differs from the running total in at least one literal. Therefore, each iteration extends the running total in at least one more dimension. Hence, if the reduced cube has $k$ dimensions, the maximum number of satisfying assignments that need to be found is $k$. (Obviously, it can be as few as two assignments in the best case, if the two 'opposite corners' of the final reduced cube are the first two assignments that are found. In fact, ordering heuristics can be geared toward such cases.)

### 4.3. Method 3:

It is possible to further speed up the algorithm just discussed. The improved algorithm MAXIMALLY-REDUCE-FASTER, shown in Figure 3, is the one that was actually implemented. Unlike the algorithm in Figure 2, the improved algorithm does not make multiple calls to SAT_CHECK, obtaining satisfying assignments one at a time. Instead, it makes a *single call* to a modified SAT checker NEW-SAT_CHECK, which computes all the solutions of $\Psi$ and returns their 'running total' supercube.

Before explaining NEW-SAT_CHECK, we first briefly review how state-of-the-art SAT checkers work. To determine the satisfiability of a given propositional Boolean formula on $n$ variables ($\Psi$ in our case), SAT checkers perform a sophisticated backtracking search of the Boolean space $\mathbb{B}^n$. During the search process, the SAT engine maintains a partial assignment, which is constructed a few variables at a time in the hope of finding a satisfying assignment (or a proof of unsatisfiability). While constructing the assignment, conflicts may be discovered, i.e. situations where the entire subcube represented by a partial assignment has been unsuccessfully searched without finding a satisfying assignment. In that case, the search *backtracks*. Modern SAT checkers implement a *learning mechanism* for conflicts - a clause is added to the original formula so that the same unsuccessful situation is not repeated.

Our modified SAT checker mainly differs in its behavior when finding a satisfying assignment. Unlike the original SAT_CHECK, NEW-SAT_CHECK does not immediately terminate when a satisfying instance is found. Instead, it treats this situation as a conflict - re-using zChaff's (the employed SAT engine) own internal functions. The resulting state of the search is as though we had started out with the found assignment blocked out. In particular, first, the formula being tested is adjusted and then the backtracking search is resumed, either continuing at the appropriate depth $blevel$ in the search tree, or stopping if $blevel$ turns out to be the root level of the search tree indicating that the search space has been exhausted. Note that the search tree is not maintained explicitly but is traversed implicitly.

One major advantage of using NEW-SAT_CHECK is that it allows us to avoid repeatedly restarting the backtracking search from scratch. This not only leads to a more efficient algorithm but also translates into an implementation efficiency — we no longer need to repeatedly set up and tear down the state needed by SAT_CHECK. Thus, a lot of redundant work is eliminated.

Note that the new algorithm makes the same adjustments to the formula $\Psi$ and supercube $\widetilde{\alpha}$ as does the previous algorithm (Method 2). The major difference is that in the new algorithm, the adjustments are made by the SAT checker, whereas in the previous algorithm they were made by the calling function.

```
MAXIMALLY-REDUCE-FASTER ($\alpha$, $F$, $F^{ON}$)
{
  $\Phi \leftarrow$ TOCNF $\left(\sum_{\gamma \in F^{ON}} \gamma\right)$
  $\Psi \leftarrow \alpha \cdot \left(\prod_{\substack{\beta \in F \\ \beta \neq \alpha}} \overline{\beta}\right) \cdot \Phi$
  $\widetilde{\alpha} \leftarrow 0$
  NEW-SAT_CHECK ($\Psi$, &$\widetilde{\alpha}$)
  /* NEW-SAT_CHECK always returns UNSATISFIABLE */
  return $\widetilde{\alpha}$;
}

Inside the SAT engine, whenever a satisfying assignment $\phi$ is found:
⋮
  $\widetilde{\alpha} \leftarrow$ SUPERCUBE($\widetilde{\alpha}, \phi$)
  $\Psi \leftarrow \Psi \cdot \widetilde{\alpha}'$
  $blevel$ = DETERMINE-BACKTRACK-LEVEL()
  BACKTRACK($blevel$);
  if (CURRENT-LEVEL () <= ROOT_LEVEL)
  {
    /* search space is exhausted; return from SAT engine */
    return UNSATISFIABLE
  }
⋮
```

**Figure 3. Maximally-Reduce-Faster**

## 5. Irredundant

The IRREDUNDANT operator takes a cover produced by EXPAND and tries to reduce its cardinality to a local minimum. ESPRESSO-II's IRREDUNDANT operator sets up and solves an optimization problem to find a largest subset of implicants that can be removed from the given cover without making it invalid.

Currently, our implementation uses a simple-minded algorithm where we test each implicant $\alpha$ in a cover $F$ for relative essentiality, i.e. if it contains an ON-set minterm of $f$ that is not contained in any other implicant of $F$. Each implicant that is not relatively essential is immediately removed from $F$. This algorithm does produce an irredundant cover, but the resulting quality may be suboptimal. In practice, we observed that this suboptimality is almost always negligible, and does not negatively influence the outcome of the logic minimization algorithm.

In our algorithm, SAT is employed in the test for relative essentiality. For $\alpha \in F$ to be relatively essential, there must exist a witness ON-set minterm that is contained in $\alpha$ but not in any other implicant of $F$. Hence, Formula (1), presented in the previous section, can be used to test if a cube is relatively essential. The formula is satisfiable for a cube $\alpha$ if and only if $\alpha$ is relatively essential. Note that in contrast to the previous section, it is not necessary to seek more than one satisfying assignment of the formula.

## 6. Essentials

The ESSENTIALS operator is intended to simplify the minimization problem. Essential prime implicants must be present in any prime cover of the given function. Therefore, they should be identified at the outset so that the subsequent main loop of ESPRESSO-II only has to deal with non-essential primes.

A prime implicant $\alpha$ of a function $f$ is essential if there exists at least one *witness* ON-set minterm that is contained in $\alpha$ but not in any other prime implicant of $f$. Consequently, to compute the essentials of a function, it is sufficient to compute their witnesses. To get the actual essentials given the witnesses, we can use the EXPAND operator on each witness.

### 6.1. Characterization of Witnesses

We now derive some facts, which will help to identify witnesses ($f$ denotes a given Boolean function). Let $e = l_1 l_2 \ldots l_n$ be a minterm, then there are $n$ minterms adjacent to it, each of which is obtained by negating one literal in $e$:

$$\overline{l_1} l_2 \ldots l_n, \qquad l_1 \overline{l_2} \ldots l_n, \qquad \ldots \qquad l_1 l_2 \ldots \overline{l_n}$$

Let $Adj(e)$ denote the set of minterms adjacent to $e$. Then, we define:

$$\begin{aligned}
Adj^{ON}(e) &\equiv Adj(e) \cap \text{ON-set}(f) \\
&\qquad \text{(i.e. ON-set minterms adjacent to } e) \\
Adj^{DC}(e) &\equiv Adj(e) \cap \text{DC-set}(f) \\
&\qquad \text{(i.e. DC-set minterms adjacent to } e) \\
Adj^{OFF}(e) &\equiv Adj(e) \cap \text{OFF-set}(f) \\
&\qquad \text{(i.e. OFF-set minterms adjacent to } e)
\end{aligned}$$

It holds that $Adj^{ON}(e)$, $Adj^{DC}(e)$ and $Adj^{OFF}(e)$ are pairwise disjoint; and $Adj^{ON}(e) \cup Adj^{DC}(e) \cup Adj^{OFF}(e) = Adj(e)$.

Given a cube $\alpha$, we also define $Adj^{\{\alpha\}}(e)$ to be the set of all minterms contained in $\alpha$ that are adjacent to $e$. In this section, we denote the (uniquely defined) supercube of a set $S$ of minterms by $supercube(S)$.

**Fact 6.1** *If $\alpha$ is a Boolean $k$-cube, and $e$ is a minterm in $\alpha$, then it holds $\alpha = supercube(\{e\} \cup Adj^{\{\alpha\}}(e))$*

Since $\alpha$ has $k$ dimensions, there are $k$ minterms in $\alpha$ that are adjacent to $e$ (i.e. $|Adj^{\{\alpha\}}(e)| = k$). Since the supercube under consideration must contain $e$ and all of these $k$ adjacent minterms, it itself must have dimensionality of at least $k$. In fact, since $\alpha$ has $k$ dimensions, the supercube must also have exactly $k$ dimensions (by definition). Further, for the supercube to be uniquely defined, we must have that $\alpha$ is the supercube.

**Fact 6.2** *Given an ON-set minterm $e$ and a prime implicant $\alpha$ containing it, $e$ is a witness of essentiality (with $\alpha$ its essential prime) if and only if all minterms in $Adj^{ON}(e) \cup Adj^{DC}(e)$ are contained in $\alpha$.*

('only if'): Assume there is an $e' \in Adj^{ON}(e) \cup Adj^{DC}(e)$ that is *not* contained in $\alpha$. Then since $e$ and $e'$ are adjacent, they form a binary cube. Further, neither minterm is in the OFF-set. Hence, the set $\{e, e'\}$ represents an implicant, which we will denote as $\beta$ for brevity. Let $\beta'$ be a prime implicant containing $\beta$. Then $\beta'$ contains both $e$ and $e'$. But this means $\beta'$ is a prime implicant that contains $e$ but is distinct from $\alpha$ ($\alpha$ does not contain $e'$ by assumption). Hence, $e$ is not a witness and $\alpha$ is not an essential.

('if'): Assume all minterms in $Adj^{ON}(e) \cup Adj^{DC}(e)$ are contained in $\alpha$. Then we must have that $Adj^{ON}(e) \cup Adj^{DC}(e) \subseteq Adj^{\{\alpha\}}(e)$. Now let $\beta$ be an arbitrary prime implicant containing $e$. Since $\beta$ cannot intersect the OFF-set, every minterm in $\beta$ must be either an ON-set minterm or a DC-set minterm. This implies that $Adj^{\{\beta\}}(e) \subseteq Adj^{ON}(e) \cup Adj^{DC}(e)$. Hence we must have that $Adj^{\{\beta\}}(e) \subseteq Adj^{\{\alpha\}}(e)$. This in turn means that

$$\begin{aligned}
\beta &= supercube(\{e\} \cup Adj^{\{\beta\}}(e)) \qquad \text{(Fact 6.1)} \\
&\subseteq supercube(\{e\} \cup Adj^{\{\alpha\}}(e)) \\
&= \alpha
\end{aligned}$$

Thus, $\beta \subseteq \alpha$. But since $\beta$ is also a prime implicant and hence cannot be contained in any other implicant, this means that $\beta = \alpha$. Therefore, there is exactly one prime implicant that contains $e$. Thus, $e$ is a witness to essentiality and $\alpha$ is an essential prime, which completes the proof.

**Fact 6.3** *An ON-set minterm $e$ is a witness of essentiality if and only if $supercube(\{e\} \cup Adj^{ON}(e) \cup Adj^{DC}(e))$ does not intersect the OFF-set.*

If $supercube(\{e\} \cup Adj^{ON}(e) \cup Adj^{DC}(e))$ does not intersect the OFF-set, then it is an implicant which contains all minterms in $Adj^{ON}(e) \cup Adj^{DC}(e)$. The cube is a prime implicant as expanding it in any additional direction would include one minterm from $Adj^{OFF}(e)$ and thus overlap the OFF-set. Hence, $e$ is a witness of essentiality (Fact 6.2). If $supercube(\{e\} \cup Adj^{ON}(e) \cup Adj^{DC}(e))$ intersects the OFF-set, no implicant can be found to contain all minterms in $Adj^{ON}(e) \cup Adj^{DC}(e)$. Hence, $e$ is not a witness (Fact 6.2).

## 6.2. SAT-Formula for Witnesses

Fact 6.3 suggests a procedure, given below, for deciding whether a minterm $e$ is a witness. First, we adopt the following notation:

- Let $f^1 : \mathbb{B}^n \to \mathbb{B}$ denote the Boolean function that evaluates to 1 for all ON-set minterms and 0 otherwise. We express $f^1$ by the formula $\sum_{\gamma \in F^{ON}} \gamma$.
- Let $f^{1*} : \mathbb{B}^n \to \mathbb{B}$ denote the Boolean function that evaluates to 0 for all OFF-set minterms and 1 otherwise. We express $f^{1*}$ by the formula $\prod_{\beta \in F^{OFF}} \overline{\beta}$.

Procedure for deciding whether a given minterm $e = (l_1, l_2, \ldots, l_n)$ is a witness:

1. Ensure that $f^1(e) = 1$. If not, $e$ is not a witness.
2. For each $i$ in $1 \ldots n$:
   Let $e_i = (l_1, l_2, \ldots, l_{i-1}, \overline{l_i}, l_{i+1}, \ldots, l_n)$.
   Let $p_i = 1$ if $f^{1*}(e_i) = 0$; otherwise let $p_i = 0$.
3. Compute supercube of $e$ with all $e_i$ for which $p_i = 0$, i.e. $f^{1*}(e_i) = 1$.
4. If this supercube does not intersect the OFF-set, then $e$ is a witness. Otherwise, $e$ is not a witness.

We can 'program' the above procedure into a single Boolean formula over the $2n$ variables $l_1, l_2, \ldots, l_n, p_1, p_2, \ldots, p_n$. Here, the $p$-variables are implied variables, and the formula is satisfiable for a minterm $e = (l_1, l_2, \ldots, l_n)$ if and only if $e$ is a witness to an essential prime implicant:

$$f^1(l_1, l_2, \ldots, l_n) \quad (2)$$
$$\cdot \quad \prod_{i=1}^{n} \left[ \overline{p_i} \leftrightarrow f^{1*}(l_1, l_2, \ldots, l_{i-1}, \overline{l_i}, l_{i+1}, \ldots, l_n) \right] \quad (3)$$
$$\cdot \quad f_p^{1*}(l_1, l_2, \ldots, l_n, p_1, p_2, \ldots, p_n) \quad (4)$$

Subformula (2) 'performs' Step 1 of the procedure outlined above; subformula (3) performs Step 2; and subformula (4) combines Steps 3 and 4. The formula for $f_p^{1*}$ is constructed by taking the formula for $f^{1*}$, and then replacing each occurrence of the literal $x_i$ with $p_i x_i$ and each occurrence of the literal $\overline{x_i}$ with $p_i \overline{x_i}$. Intuitively, the supercube must not intersect the OFF-set. That is, for each cube $\beta \in F^{OFF}$, the supercube must contain one literal whose negated literal is contained in $\beta$. The latter can be reformulated for a witness. The witness must contain one literal *(i)* whose negated literal is contained in $\beta$ and *(ii)* which does not correspond to a literal that is removed by the supercube operation, i.e. for which $p_i$ is 0. $f^{1*}$ negates each OFF-set cube $\beta$ obtaining a CNF clause which guarantees (i), and (ii) is established by using the $p$-variables to mask out literals which are not included in the supercube.

| Name | ESPRESSO-II | BOOM | SAT-ESPRESSO |
|---|---|---|---|
| 50/100 | 17.79 | 8.48 | 3.04 |
| 50/150 | 48.40 | 31.57 | 6.69 |
| 50/200 | 138.55 | 109.03 | 23.13 |
| 100/50 | 9.23 | 0.63 | 3.11 |
| 100/200 | 1198.20 | 165.83 | 64.16 |
| 150/100 | 175.43 | 13.66 | 16.59 |
| 150/200 | 1320.30 | 1212.21 | 260.36 |
| 200/50 | 18.44 | 10.63 | 3.12 |
| 200/100 | 204.49 | 30.40 | 22.15 |
| 200/150 | 1265.68 | 186.52 | 56.05 |
| 200/200 | 2178.11 | 2626.39 | 134.19 |

**Table 1. Run-time comparison (in sec)**

The entire formula can be converted to CNF using the technique described in Section 4.2.

As an optimization, one can compute an over-approximation $S$ to the set of essential primes, and then constrain the above formula to represent only those minterms that are contained in $S$. The over-approximation $S$ can be computed by taking any two covers of the given function and computing their intersection. The two covers are obtained by using EXPAND with different heuristics for expanding cubes. In practice, this optimization was very useful because in all of the examples we tested, our over-approximation $S$ turned out to be the empty set, indicating that no essentials are present, constraining the formula to FALSE, and thus allowing us to skip the SAT checking phase entirely.

## 7. Experimental Results

Table 1 compares the overall run-times of SAT-ESPRESSO with ESPRESSO-II and BOOM on a recently published benchmark suite [6]. BOOM is a recently developed heuristic minimizer that specializes in large examples. SAT-ESPRESSO was found to be typically 5–20 times faster than ESPRESSO-II and 3–5 times faster than BOOM.

Table 2 compares per-operator run-times of SAT-ESPRESSO with ESPRESSO-II for the operators REDUCE, ESSENTIALS and IRREDUNDANT. Here, we observed speed-ups of one to two orders of magnitude (REDUCE: 16–400 times; ESSENTIALS: 37–270 times; IRREDUNDANT: 7–110 times).

Note that SAT-ESPRESSO uses the same overall strategy as ESPRESSO-II, and thus obtains the same resulting covers. Hence, the tables in this section primarily focus on run-time.

ESPRESSO-II and SAT-ESPRESSO both iterate until no further improvement of the cover can be achieved. In contrast, BOOM iterates until the cover satisfies a given criteria, e.g. the number of literals is smaller than a given bound. Hence, we ran BOOM after ESPRESSO-II and SAT-ESPRESSO, using the obtained number of literals from these runs as the bound to be achieved by BOOM for termination.

In SAT-ESPRESSO, we use as SAT checker a modified

| Name ($i/p$) | ESSENTIALS | | | IRREDUNDANT | | | REDUCE | | |
|---|---|---|---|---|---|---|---|---|---|
| | ESPRESSO-II | SAT-ESPRESSO | Factor | ESPRESSO-II | SAT-ESPRESSO | Factor | ESPRESSO-II | SAT-ESPRESSO | Factor |
| 50/100 | 4.86 | 0.08 | 60.75 | 2.99 | 0.38 | 7.87 | 7.77 | 0.39 | 19.92 |
| 50/150 | 15.32 | 0.22 | 69.64 | 6.08 | 0.69 | 8.81 | 21.97 | 0.74 | 29.69 |
| 50/200 | 35.28 | 0.37 | 95.35 | 18.64 | 1.40 | 13.31 | 65.00 | 1.68 | 38.69 |
| 100/50 | 2.65 | 0.07 | 37.86 | 1.11 | 0.16 | 6.94 | 2.68 | 0.17 | 15.76 |
| 100/200 | 212.70 | 1.10 | 193.36 | 134.17 | 2.12 | 63.29 | 793.85 | 3.64 | 218.09 |
| 150/100 | 31.10 | 0.46 | 67.61 | 20.39 | 0.72 | 28.32 | 109.63 | 1.18 | 92.91 |
| 150/200 | 511.33 | 1.87 | 273.44 | 103.75 | 0.95 | 109.21 | 448.51 | 1.20 | 373.76 |
| 200/50 | 6.67 | 0.18 | 37.06 | 2.35 | 0.26 | 9.04 | 7.11 | 0.32 | 22.22 |
| 200/100 | 84.44 | 0.64 | 131.94 | 14.88 | 0.56 | 26.57 | 84.65 | 0.74 | 114.39 |
| 200/150 | 251.70 | 1.47 | 171.22 | 94.92 | 1.82 | 52.15 | 869.72 | 3.79 | 229.48 |
| 200/200 | 567.99 | 2.84 | 200.00 | 183.58 | 1.70 | 107.99 | 1301.07 | 3.27 | 397.88 |

**Table 2. Time spent in operators ESSENTIALS, IRREDUNDANT and REDUCE.** $i/p$-number of inputs/number implicants in the initial cover. All functions have 5 outputs.

version of zChaff [10]. Our implementation involves a lot of redundant file I/O, so there is room for improvement regarding the performance.

ESPRESSO-II and SAT-ESPRESSO were run on a Dell OptiPlex GX300 workstation using a 733MHz Intel Pentium III processor, 512MB system memory and the Linux operating system. BOOM was run on a Dell OptiPlex GX1 500 MTbr+ workstation using a 500MHz Pentium III processor, 512MB system memory and the Windows 2000 operating system. This arrangement was due to the fact that BOOM was only available for Windows whereas zChaff (the SAT checker used) was not. To estimate the difference due to workstation type, ESPRESSO-II was also run on both platforms for seven of the examples with an observed average speed-up from Windows to Linux of $1.46 \pm 0.15$.

All considered examples were taken from a recently published benchmark set of large examples [6]. However, it turns out that this benchmark set shows some unexpected unifying behavior: the examples do not have essential prime implicants and the IRREDUNDANT operator never succeeds. For these reasons, further experimentation with other sets of large benchmarks will be necessary to better evaluate our new algorithms.

## 8. Conclusions

We have presented an approach to efficiently achieve high-quality approximations on large two-level logic minimization problems, by combining the strengths of ESPRESSO-II (quality of approximation) and SAT checkers (speed on large problems). Preliminary experiments show significant reductions in run-time when compared to other minimizers.

## References

[1] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, 1984.

[2] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[3] E. M. Clarke, P. Chauhan, S. Sapra, J. Kukula, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *FMCAD*, pages 33–51, 2002.

[4] O. Coudert. Two-level logic minimization: an overview. *Integration, the VLSI journal*, 17:97–140, 1994.

[5] S. Devadas, A. Ghosh, and K. Keutzer. *Logic Synthesis*. McGraw-Hill, 1994.

[6] J. Hlavicka and P. Fiser. BOOM - a heuristic boolean minimizer. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 439–442, 2001.

[7] E. McCluskey. *Logic Design Principles*. Prentice-Hall, 1986.

[8] K. L. McMillan. Applying sat methods in unbounded symbolic model checking. In *Proc. Computer Aided Verification*, pages 250–264, 2002.

[9] G. D. Micheli. *Synthesis And Optimization Of Digital Circuits*. McGraw-Hill, 1994.

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. ACM/IEEE Design Automation Conference*, pages 530–535, 2001.

[11] R. Rudell. Logic synthesis for VLSI design. Technical Report UCB/ERL M89/49, Berkeley, 1989.

[12] R. Rudell and A. Sangiovanni-Vincentelli. Multiple valued minimization for PLA optimization. *IEEE Transactions on CAD*, CAD-6(5):727–750, September 1987.

[13] J. P. M. Silva and K. A. Sakallah. Grasp: A new search algorithm for satisfiability. Technical Report CSE-TR-292-96, Computer Science and Engineering Division, Department of EECS, Univ. of Michigan, 1996.

[14] M. Theobald. *Efficient Algorithms for the Design of Asynchronous Control Circuits*. PhD thesis, Department of Computer Science, Columbia University, 2002.

[15] M. Theobald and S. M. Nowick. Fast heuristic and exact algorithms for two-level hazard-free logic minimization. *IEEE Transactions on Computer-Aided Design*, Nov. 1998.

[16] H. Zhang. Sato: An efficient propositional prover. In *Proceedings of the Conference on Automated Deduction (CADE'97)*, pages 272–275, 1997.