Journal on Satisfiability, Boolean Modeling and Computation 1 (2006) 89-110

Extending Existential Quantification in Conjunctions of BDDs

Sean Weaver[*] fett@gauss.ececs.uc.edu
John Franco[*] franco@gauss.ececs.uc.edu
John Schlipf[*] schlipf@ececs.uc.edu
ECECS,
University of Cincinnati,
Cincinnati, OH 45221-0030, USA

Abstract

We introduce new approaches intended to speed up determining the satisfiability of a given Boolean formula φ expressed as a conjunction of Boolean functions. A common practice in such cases, when using constraint-oriented methods, is to represent the functions as BDDs, then repeatedly cluster BDDs containing one or more variables, and finally existentially quantify those variables away from the cluster. Clustering is essential because, in general, existential quantification cannot be applied unless the variables occur in only a single BDD. But, clustering incurs significant overhead and may result in BDDs that are too big to allow the process to complete in a reasonable amount of time.

There are two significant contributions in this paper. First, we identify elementary conditions under which the existential quantification of a subset of variables V' may be distributed over all BDDs without clustering. We show that when these conditions are satisfied, *safe assignments* to the variables of V' are automatically generated. This is significant because these assignments can be applied, as though they were inferences, to simplify φ.

Second, some efficient operations based on these conditions are introduced and can be integrated into existing frameworks of both *search-oriented* and *constraint-oriented* methods of satisfiability. All of these operations are relaxations in the use of existential quantification and therefore may fail to find one or more existing safe assignments.

Finally, we compare and contrast the relationship of these operations to autarkies and present some preliminary results.

Keywords: *satisfiability, autarky, safe assignment, inference*

Submitted November 2005; revised May 2006; published June 2006

1. Introduction

We are interested in determining the satisfiability of arbitrary Boolean formulae which are expressed as conjunctions (logical "and") of Boolean functions. Boolean formulae, arising from real applications, are often expressed as conjunctions of Binary Decision Diagrams (BDDs). BDDs are described in detail in section 2.3. The satisfiability of a formula can

[*] Supported by the U.S. Department of Defense under contracts MDA904-99-C-4547 and MDA904-02-C-1162.

be determined using either a search-oriented or constraint-oriented method. Most search-oriented methods used in practice employ backtracking as a variation of the DPLL algorithm [6]. Constraint-oriented methods primarily intend to create a *monolithic* BDD which represents the same function as the conjunction; if such a BDD is created, all paths to its *1* node represent satisfying solutions.

The main ideas presented in this paper came about while looking at an operation, called *existential quantification*, that is critical to the constraint-oriented method. Let v be a Boolean variable in the support of a Boolean function φ. Denote the assignment of value `True` (`False`) to variable v by $v \mapsto$ `True` ($v \mapsto$ `False`), respectively. Let $\varphi\,|_v$ ($\varphi\,|_{\overline{v}}$) denote the result of the assignment $v \mapsto$ `True` ($v \mapsto$ `False`), respectively. Existentially quantifying v away from φ means replacing φ with $\varphi\,|_v \vee \varphi\,|_{\overline{v}}$. Existential quantification is described in more detail in section 2.4. It is primarily used during the process of clustering BDDs to remove variables which occur in a single BDD and is the essential component of the *early quantification method*, first proposed in [5] and later specifically applied to satisfiability in [11]. Since then, the method of early quantification has been used numerous times (for example, see [10], [14], and [7]). It is generally thought that the existential quantification of a variable, since it is not distributive over a conjunction of functions, can be applied *only* when the variable occurs in just one BDD of the conjunction. We show that this is not always the case.

Our results depend on the notion of *safe assignment* which is formalized as follows. Note, if φ_1 and φ_2 are Boolean functions, then we use $\varphi_1 \equiv \varphi_2$ to mean "φ_1 and φ_2 are logically equivalent" and $\varphi_1 \rightarrow \varphi_2$ to mean $\overline{\varphi_1} \vee \varphi_2$.

Definition 1. *The assignment $v \mapsto$ `True` is* safe *in φ if any assignment satisfying $\varphi\,|_{\overline{v}}$ also satisfies $\varphi\,|_v$, i.e. if $(\varphi\,|_{\overline{v}} \rightarrow \varphi\,|_v) \equiv$ `True`, and the assignment $v \mapsto$ `False` is* safe *in φ if $(\varphi\,|_v \rightarrow \varphi\,|_{\overline{v}}) \equiv$ `True`.*

The next theorem shows why we use the term *safe* to describe assignments of Definition 1.

Theorem 1.

1. *If $v \mapsto$ `True` is safe in φ then $\varphi\,|_v$ is satisfiable if and only if φ is satisfiable.*
2. *If $v \mapsto$ `False` is safe in φ then $\varphi\,|_{\overline{v}}$ is satisfiable if and only if φ is satisfiable.*

Proof. Consider case 1. φ is satisfiable if and only if $\varphi\,|_v \vee \varphi\,|_{\overline{v}}$ is satisfiable. Since $\varphi\,|_{\overline{v}} \rightarrow \varphi\,|_v \equiv$ `True` then $(\varphi\,|_v \vee \varphi\,|_{\overline{v}}) \wedge (\varphi\,|_{\overline{v}} \rightarrow \varphi\,|_v) = \varphi\,|_v$ and φ is satisfiable if and only if $\varphi\,|_v$ is satisfiable. A similar argument applies to case 2. □

From Theorem 1 it is clear that if an assignment to v is safe, it may be applied instead of existentially quantifying away v. However, we do not propose to use safe assignments to replace existential quantification. Rather, we develop new tools which *help* the early quantification method by reducing on quantifications *before* a variable has been totally clustered into a single BDD.

In this article we present some elementary conditions under which a set of variables occurring in multiple BDDs may be existentially quantified away without clustering (that is, conjoining) those BDDs. Moreover, in the process of finding whether or not a set of variables may be existentially quantified away, we also find a safe assignment for those variables. Thus, our results allow the choice of existentially quantifying or safely assigning. We will show that sometimes one is preferred to the other. The operations we present for finding safe assignments can be used by constraint- or search-oriented methods.

Andersson et.al. [2] identify the same notion of safe assignment which they call *variable instantiation*. We believe the following points highlight the differences between their work and ours:

1. The method proposed in [2] requires two tests on a single variable: for a safe `True` and safe `False` value. Our method subsumes both tests in a single, distributed operation which may automatically reveal safe assignments. We note that for both methods it is possible that existing safe assignments are not revealed depending on how the formula is represented.

2. Our method is able to consider several variables simultaneously, sometimes finding a safe assignment that cannot be found considering those variables individually. The method proposed in [2] applies to single variables.

3. Our method distributes computation over many constraints, without conjoining them. In many cases this avoids having to deal with an unacceptably large intermediate constraint that may be a by product of the conjunction. It also allows the possibility of revealing a safe assignment before computation ends. The method stated in [2] creates a single, conjoined constraint before trying to reveal a safe assignment. Therefore, this method does not terminate early and may lose efficiency to large intermediate constraints.

As reported in [2], significant improvements to traditional BDD and DPLL techniques are possible when variable instantiation is employed. Therefore, we choose not to pay significant attention to such comparisons here and focus instead on the nature of our optimizations. Points 1., 2., 3., above are illustrated by examples given near the end of this article.

2. Background

2.1 Satisfiability

The question of determining whether there exists an assignment of values (`True` or `False`) to input variables causing a Boolean formula to evaluate to `True` is called the Satisfiability problem (or SAT for short). A solution, or satisfying assignment, to a Boolean formula is an assignment of values to variables which causes the conjunction to evaluate to `True`. If a solution exists the formula is said to be *satisfiable*; if no solution exists the formula is said to be *unsatisfiable*.

2.2 Conjunctive Normal Form

Search-oriented methods such as DPLL [6] have historically been intended for formulae in Conjunctive Normal Form (CNF): that is, a conjunction of Boolean functions, each of which is expressed as a simple disjunction of literals. Although few real problems are naturally expressed in CNF, it is well known that any propositional formula can be expressed as a CNF formula with only a constant factor increase in size, but perhaps requiring the introduction of considerably many variables that have nothing substantial to do with the given formula. An advantage of search-oriented methods is they can be terminated without exploring an entire search space if a solution is encountered early. Search-oriented methods originally suffered from having rather large search spaces due to their inherent tree-like structure. However, recent advances have replaced tree-like search spaces with DAG-like search spaces resulting in much better general performance, particularly on unsatisfiable inputs. A fundamental operation of search-oriented methods is assigning a value to a variable.

2.3 Binary Decision Diagrams

In contrast to search-oriented methods, constraint-oriented methods maintain constraint structures and employ operations that combine those structures until a final constraint structure is produced. That structure determines the satisfiability of the given formula. The most common constraint structure used by these methods is the Binary Decision Diagram (BDD).

BDDs were introduced by Lee [12] and Akers [1]. A BDD is a rooted directed acyclic graph with nodes labeled by names of input variables, except for two special leaf nodes labeled *1* and *0*, corresponding to `True` and `False`, respectively. All nodes, except for the two leaf nodes, have two outgoing edges, one labeled *1* and one labeled *0*. The special nodes have no outgoing edges.

A BDD is a compact representation of a Boolean function on a particular ordering of input variables. Every path from root to leaf node represents an assignment of values to input variables, where each variable associated with a node on the path is assigned the value specified by the label of the edge taken on the path out of that node. A path ending in the *1* node causes the function to evaluate to `True`, all other paths cause the function to evaluate to `False`. In formulas of the type we consider in this paper there is one BDD per Boolean function and the ordering of variables is arbitrary and the same for all those BDDs. For convenience we will use comparisons on variables to indicate the relative order of those variables. Thus, $v_1 > v_2$ will mean that variable v_1 is "higher" in order than v_2. In other words, if both v_1 and v_2 associate with nodes on a BDD path from root to leaf, variable v_1 is considered to be closer (maybe the same as) the root. Also, $v_1 = v_2$ will mean that v_1 and v_2 are the same variable.

We are concerned with a particular form of BDDs called Reduced Ordered Binary Decision Diagrams (ROBDDs) [4]. ROBDDs are *canonical*: that is, for every Boolean function and fixed variable ordering, there is a *unique* logically equivalent ROBDD. In conformance with the literature, from now on we will use the term BDD in place of ROBDD.

There exist operations for building and manipulating BDDs. Many have been incorporated in the well-known suite of operations that has been available for some time in BDD packages such as CUDD [16]. The results of this paper use the following subset of BDD operations: **`tbr`**, **`fbr`**, **`var`**, **`and`**, **`or`**, **`not`**, **`ite`** and **`ite_constant`**. The **`tbr`** ("true-branch") and **`fbr`** ("false-branch") operations take a BDD b with root variable v, as input and return $b\,|_v$ and $b\,|_{\overline{v}}$, respectively. The **`var`** operation takes a Boolean variable v_1 as input and returns a BDD representing v_1. The **`and`** and **`or`** operation takes two BDDs b_1 and b_2 as input and returns a BDD representing $b_1 \wedge b_2$ and $b_1 \vee b_2$, respectively. The `not` operation takes a BDD b as input and returns $\bar{b}$. The `ite` ("if-then-else") operation takes as input three BDDs b_1, b_2, and b_3, and returns $(b_1 \wedge b_2) \vee (\overline{b_1} \wedge b_3)$. As used in this paper, the `ite_constant` operation takes as input three BDDs and returns `False` if the result of the `ite` operation, given the three input BDDs, is `False` and returns `True` otherwise. The `ite_constant` operation *does not* build a BDD, but only simultaneously and non-exhaustively traverses the three input BDDs. We use `ite_constant` below to make algorithms more efficient. These basic BDD operations are introduced and discussed in more depth in [3].

2.4 Existential Quantification

Both search-oriented and constraint-oriented methods of satisfiability use Boolean existential quantification. In this paper we use existential quantification to motivate our algorithms. In rough analogy to existential quantification in first order logic, for some variable v and Boolean function b depending on v,

$$\exists v(b) =_{\text{def}} b\mid_v \vee b\mid_{\overline{v}}.$$

The traditional aim of existential quantification is to remove a variable from a Boolean formula without affecting its satisfiability. Let $\{b_1, b_2, \cdots, b_m\}$ be a set of Boolean functions and suppose variable v appears in *only* one of them, say b_j. Then

$$\exists v(b_1 \wedge \cdots \wedge b_m) \quad \equiv \quad \exists v(b_j) \wedge b_1 \wedge \cdots \wedge b_{j-1} \wedge b_{j+1} \wedge \cdots \wedge b_m. \tag{1}$$

The right side of (1) can be computed in time proportional to the size of b_j and is therefore the preferred means of implementing existential quantification. In other words, variable v is *existentially quantified away* from b_j before conjoining $\{b_1, b_2, \cdots, b_m\}$.

The obvious extension of this idea to multiple BDDs is unwieldy. Suppose variable v occurs in several BDDs, w.l.g. say $b_1, ..., b_n$, $n \leq m$. To be able to compute the right side of (1), one might first *cluster* $\{b_1, ..., b_n\}$: that is, replace them with a single BDD $b = b_1\wedge, ..., \wedge b_n$. Then existentially quantify v away from b. The problem with clustering is the size of b may become, in the worst case, exponential in the number of BDD nodes of $b_1, ..., b_n$. Despite this, constraint-oriented methods existentially quantify variables away in this manner, aiming in the end to create a single, final monolithic BDD with functionality equivalent to that of the given formula. Sometimes, these methods fail due to the exponential explosion of the intermediate BDDs.

We show that clustering BDDs can be avoided in some cases with the added benefit of being able to safely assign values to some variables. This results in bypassing the exponential blowup that may be incurred by conjoining BDDs and, at the same time, simplification of the formula through variable assignments. Details are in the following sections.

In what follows, b is used to represent a Boolean function and a BDD, interchangeably, and φ is used to represent a Boolean formula, a conjunction of Boolean functions, and a conjunction of BDDs, interchangeably.

3. Single Variable Safe Assignments

This section presents some elementary conditions under which a single variable may be safely assigned a value in a Boolean formula, even if it occurs in multiple functions. We illustrate these results by introducing an operation called `safe_assign` which can find a safe assignment, if one exists, to a given variable v occurring in a given conjunction of Boolean functions φ, where all functions of φ are expressed as BDDs. As stated earlier, one way to a safely remove v from φ is to existentially quantify it away from φ. However, in general, v will be in support of many BDDs of φ and existential quantification may, in this case, result in an impractically large intermediate formula because all those BDDs must be clustered in order to apply the operation (Section 2.4). On the other hand, no clustering is required in order for `safe_assign` to be applied.

Sometimes clustering BDDs is desired because doing so can increase the number of safe assignments that can be found using `safe_assign`, even if the variables with safe assignments are in support of more than one BDD after clustering. Thus, using clustering in conjunction with `safe_assign` we can accomplish at least as much as existential quantification with a lower risk of producing overly large intermediate formulas.

Finally, we point out that the computation of `safe_assign` is *distributed* over only those BDDs which contain v. This ensures the efficiency of `safe_assign`. Conditions under which multiple variables may be safely assigned, even over multiple functions, are given in Section 4. The remainder of this section lays the groundwork for that section.

Consider, first, the special case where a variable v occurs in only one Boolean function b. If $(b\,|_v \equiv \exists v(b))$ or $(b\,|_{\overline{v}} \equiv \exists v(b))$ then v can be safely assigned the value `True` or `False`, respectively, instead of being existentially quantified away. The proof of this uses Lemmas 1 and 2. Although a shorter proof exists, we prefer to present the two Lemmas because their proofs reveal some insights which get used in subsequent proofs. In the proof and elsewhere in the article $\overline{\varphi}$ is used to denote the complement of Boolean function φ.

Lemma 1. *Given a conjunction of Boolean functions $\varphi = b_1 \wedge \cdots \wedge b_m$ and variable v occurring in one or more Boolean functions of φ, let φ_1 be the conjunction of all Boolean functions in φ which contain v. Let $\psi_v = \overline{(\varphi_1\,|_v)} \wedge (\varphi_1\,|_{\overline{v}})$. Let $\psi_{\overline{v}} = (\varphi_1\,|_v) \wedge \overline{(\varphi_1\,|_{\overline{v}})}$.*

1. *If $\psi_v \equiv$ `False`, then $\varphi\,|_v$ is satisfiable if and only if φ is satisfiable.*
2. *If $\psi_{\overline{v}} \equiv$ `False`, then $\varphi\,|_{\overline{v}}$ is satisfiable if and only if φ is satisfiable.*

Proof. We consider only case 1; case 2 is analogous. Re-index the functions of φ so that $\varphi_1 = b_1 \wedge \cdots \wedge b_n$ where $n \leq m$. By definition,

$$\varphi\,|_v = \varphi_1\,|_v \wedge b_{n+1} \wedge \cdots \wedge b_m.$$

Existentially quantifying away v from φ gives

$$\exists v(\varphi) \equiv (\varphi_1\,|_v \vee \varphi_1\,|_{\overline{v}}) \wedge b_{n+1} \wedge \cdots \wedge b_m$$

which is well known to be satisfiable if and only if φ is satisfiable.

$$\begin{aligned} \varphi\,|_v \oplus \exists v(\varphi) &\equiv (\varphi\,|_v \wedge \overline{\exists v(\varphi)}) \vee (\overline{\varphi\,|_v} \wedge \exists v(\varphi)) \\ &\equiv \texttt{False} \vee \overline{(\varphi_1\,|_v \wedge b_{n+1} \wedge \cdots \wedge b_m)} \wedge \\ &\quad\ (\varphi_1\,|_v \vee \varphi_1\,|_{\overline{v}}) \wedge (b_{n+1} \wedge \cdots \wedge b_m) \\ &\equiv \overline{(\varphi_1\,|_v)} \wedge (\varphi_1\,|_{\overline{v}}) \wedge (b_{n+1} \wedge \cdots \wedge b_m) \\ &\equiv \psi_v \wedge (b_{n+1} \wedge \cdots \wedge b_m). \end{aligned} \qquad (2)$$

From (2), if ψ_v is `False`, then $\varphi\mid_v \equiv \exists v(\varphi)$. Therefore, if $\psi_v \equiv$ `False`, then $\varphi\mid_v$ is satisfiable if and only if φ is satisfiable. □

By Lemma 1, if $\psi_v \equiv$ `False` (respectively, $\psi_{\overline{v}} \equiv$ `False`), then assigning `True`(resp., `False`) to v will cause φ to reduce to $\exists v(\varphi)$, so the satisfiability of φ is not affected. We emphasize that the value of v is not necessarily inferred, yet it may be safely assigned as is shown by Lemma 2.

Lemma 2. *Given a conjunction of Boolean functions $\varphi = b_1 \wedge \cdots \wedge b_m$ and variable v occurring in one or more Boolean functions of φ, let φ_1 be the conjunction of all Boolean functions in φ which contain v. Let $\psi_v = \overline{(\varphi_1\mid_v)} \wedge (\varphi_1\mid_{\overline{v}})$. Let $\psi_{\overline{v}} = (\varphi_1\mid_v) \wedge \overline{(\varphi_1\mid_{\overline{v}})}$.*

1. *Every assignment of values to variables of $\varphi_1\mid_v$ satisfying $(\varphi_1\mid_{\overline{v}} \rightarrow \varphi_1\mid_v)$ falsifies ψ_v.*

2. *Every assignment of values to variables of $\varphi_1\mid_v$ satisfying $(\varphi_1\mid_v \rightarrow \varphi_1\mid_{\overline{v}})$ falsifies $\psi_{\overline{v}}$.*

Proof. For case 1: $\varphi_1\mid_{\overline{v}} \rightarrow \varphi_1\mid_v \;=_{\text{def}}\; \overline{\varphi_1\mid_{\overline{v}}} \vee \varphi_1\mid_v \;\equiv\; \overline{(\varphi_1\mid_{\overline{v}} \wedge \overline{\varphi_1\mid_v})} \;\equiv\; \overline{\psi_v}$. Case 2 is analogous. □

Theorem 2. *Let φ, φ_1, and v be defined as in Lemmas 1 and 2. If $\varphi_1\mid_v \equiv \exists v(\varphi_1)$ then $v \mapsto$ `True` is safe in φ. If $\varphi_1\mid_{\overline{v}} \equiv \exists v(\varphi_1)$ then $v \mapsto$ `False` is safe in φ.*

Proof. If $\varphi_1\mid_v \equiv \exists v(\varphi_1)$ then, for every assignment to variables of $\varphi_1\mid_v$, $(\varphi_1\mid_{\overline{v}} \rightarrow \varphi_1\mid_v)$. It follows from Lemmas 1 and 2 that $\varphi\mid_v$ is satisfiable if and only if φ is satisfiable and $v \mapsto$ `True` is safe. A similar statement shows when $v \mapsto$ `False` is safe. □

If Boolean functions are represented as BDDs, then the `safe_assign`$_0$ operation shown in Figure 1 may be used to find a safe assignment to a single variable occurring in one BDD. Once a safe assignment is found, standard BDD operations can be used to simplify φ accordingly. (As noted in Section 2.3, the calls to `ite_constant` in Algorithm `safe_assign`$_0$ are optimizations.) The proof of correctness of this operation relies on details contained in the proof of Lemma 1 and of 2. This is presented as Lemma 3.

Lemma 3. *Let b be any BDD, and v be any variable. Then* `safe_assign`$_0(b, v)$ *returns one of the following:*

1. `True` *if v does not occur in b,*

2. *a BDD consisting of a root corresponding to v, and leaves 0 and 1 (the returned BDD specifies a safe assignment for v),*

3. `False` *if v occurs in b but no safe assignment exists for v in b.*

Proof. By induction on the height of recursion.

At the bottom level, `True` is all that is returned by `safe_assign`$_0$ because, necessarily, v cannot be the root of a `True` or `False` BDD, so the first line of `safe_assign`$_0$ is executed in this case. Therefore, at the bottom level, v does not occur in the input BDD (parameter b) and `True` is returned so hypothesis *1.* only applies and is satisfied.

Suppose the hypotheses are true up to height k from the bottom. There are three cases to consider.

1. *Suppose v does not exist at or below the root of the input BDD*: There are two subcases.

    (a) *The index of v is greater than the index of the root variable*: The third line in `safe_assign`$_0$ (which tests for this) returns `True`. Hypothesis *1.* applies only and is satisfied.

    (b) *The index of v is less than that of the root variable*: Then `safe_assign`$_0$ returns the logical `and` of calls to `safe_assign`$_0$ on the two branches of the root. By the induction hypothesis these calls return `True`. Thus, `safe_assign`$_0$ returns `True`. Hypothesis *1.* applies only and is satisfied.

2. *Suppose v exists in the input BDD but is not at its root*: (if v does not exist in that portion of b then `safe_assign`$_0$ would have returned). There are three possible outcomes as return values from calls of `safe_assign`$_0$ on the true and false branches from the root.

    (a) *At least one value is* `False`: Then `safe_assign`$_0$ returns `False`. By the induction hypothesis, this happens if there is no safe assignment to v in either the left BDD or the right BDD. Hence there cannot be a safe assignment for v in the input BDD and hypothesis *3.* applies only and is satisfied.

    (b) *One value is* `True` *and the other is a safe assignment for one branch*: Then `safe_assign`$_0$ returns the logical `and` of `True` and a safe assignment for v in a branch below the root. This is also a safe assignment for the input BDD. Through implied BDD reductions, the logical `and` of `True` and a simple BDD is the simple BDD so the assignment is returned as a BDD with root and leaves only and, by the induction hypothesis, the root must be v. Therefore, hypothesis *2.* applies only and is satisfied.

    (c) *Two BDDs specifying opposite assignments to v are returned*: The logical `and` of these is `False` and that is what `safe_assign`$_0$ returns. Since v occurs in b, there cannot be both `True` and `False` safe assignments to v existing in b. Therefore, the return value of `False` satisfies hypothesis *3.* in this case.

    (d) *Two BDDs specifying the same assignment to v are returned*: The logical `and` of a BDD with itself is said BDD and that is what `safe_assign`$_0$ returns. This BDD has a root and leaves only and, by the induction hypothesis, the root must be v. This is also a safe assignment for the input BDD. Therefore, hypothesis *2.* applies only and is satisfied.

3. *Suppose v is the root of the input BDD*: `safe_assign`$_0$ returns at line 7 the value of `ite(var(v), not(e), not(r))`. Due to the use of `ite_constant` (see Section 2.3) and line 6 of `safe_assign`$_0$, `e`=`False` if $(\overline{b\,|_v}) \wedge (b\,|_{\overline{v}})$ is `False` and `e`=`True` otherwise; and, from `ite_constant` and line 5 of `safe_assign`$_0$, `r`=`False` if $(b\,|_v) \wedge (\overline{b\,|_{\overline{v}}})$ is `False` and `r`=`True` otherwise. Now consider the four sub-cases on values of `e` and `r`.

    (a) `e`=`False` *and* `r`=`False`: v does not exist in b and any assignment is safe. The value returned is `ite(var(v), True, True)` which is `True` as needed to satisfy hypothesis *1.*

```
Input:   A BDD $b$ and a variable $v$.
Output:  Returns:  True if $v$ does not occur in $b$;
                   else a safe assignment for $v$, if one exists;
                   else False.

BDD safe_assign$_0$(BDD $b$, variable $v$) {
   if $b$ is True or False then return True   // $v$ is not in $b$
   let $v_b =$ the root variable of $b$
   if $v > v_b$ then return True        // $v$ is not in $b$
   else if $v_b = v$ then {             // $v$ is the root of $b$
      let BDD $r$ := ite_constant(tbr($b$), not(fbr($b$)), False)
      let BDD $e$ := ite_constant(not(tbr($b$)), fbr($b$), False)
      return ite(var($v$), not($e$), not($r$))
   }
   let BDD $r :=$ safe_assign$_0$(tbr($b$), $v$)
   if $r =$ False then return False
   else return and($r$, safe_assign$_0$(fbr($b$), $v$))
}
```

Figure 1. Recursive pseudo-code, for finding a safe assignment for v in a single BDD.

(b) e=True *and* r=True: then Lemma 1 does not hold and no safe assignment can be determined for v. The value returned is `ite(var(v), False, False)` which is `False` as needed to satisfy hypothesis *3*.

(c) e=False *and* r=True: by Lemma 1 it is safe to assign `True` to v. The value returned is a BDD with root at v and the true branch incident with leaf *1* and the false branch incident with leaf *0* as needed to satisfy hypothesis *2*.

(d) e=True *and* r=False: by Lemma 1 it is safe to assign `False` to v. The value returned is a BDD with root at v and the true branch incident with leaf *0* and the false branch incident with leaf *1* as needed to satisfy hypothesis *2*.

□

The complexity of `safe_assign`$_0$ is linear in the number of nodes of b.

We are now ready to consider the general case where v occurs in more than one Boolean function.

Theorem 3 below provides the means to avoid clustering and instead distribute the computation of ψ_v and $\psi_{\overline{v}}$ over all Boolean functions depending on v. The cost of improved efficiency is that a safe assignment may not be always be determined via Theorem 3. The theorem depends on the next lemma.

Lemma 4. *Given a conjunction of Boolean functions $\varphi = b_1 \wedge \cdots \wedge b_m$ and a variable v occurring in one or more Boolean functions of φ, let φ_1 be the conjunction of all Boolean* *functions in φ which contain v. Without loss of generality, suppose $\varphi_1 = b_1 \wedge \cdots \wedge b_n$ where $n \leq m$. Let $\beta_v = (\overline{b_1 |_v} \wedge b_1 |_{\overline{v}}) \vee \cdots \vee (\overline{b_n |_v} \wedge b_n |_{\overline{v}})$. Let $\beta_{\overline{v}} = (b_1 |_v \wedge \overline{b_1 |_{\overline{v}}}) \vee \cdots \vee (b_n |_v \wedge \overline{b_n |_{\overline{v}}})$.*

1. *If $\beta_v \equiv$ `False`, then $\varphi |_v$ is satisfiable if and only if φ is satisfiable.*

2. *If $\beta_{\overline{v}} \equiv$ `False`, then $\varphi |_{\overline{v}}$ is satisfiable if and only if φ is satisfiable.*

Proof. We consider only case 1; case 2 is analogous. From Lemma 1, if $\psi_v \equiv$ `False` then $\varphi |_v$ is satisfiable if and only if φ is satisfiable. So, we show that $\beta_v \equiv$ `False` implies $\psi_v \equiv$ `False`.

$$\begin{aligned}
\overline{\beta_v} \rightarrow \overline{\psi_v} \quad &\equiv \quad \beta_v \vee \overline{\overline{(\varphi_1 |_v)} \wedge (\varphi_1 |_{\overline{v}})} \\
&\equiv \quad (\overline{b_1 |_v} \wedge b_1 |_{\overline{v}}) \vee \cdots \vee (\overline{b_n |_v} \wedge b_n |_{\overline{v}}) \vee \\
&\qquad \overline{(\overline{(b_1 |_v \wedge \cdots \wedge b_n |_v)} \wedge (b_1 |_{\overline{v}} \wedge \cdots \wedge b_n |_{\overline{v}}))} \\
&\equiv \quad (\overline{b_1 |_v} \wedge b_1 |_{\overline{v}}) \vee \cdots \vee (\overline{b_n |_v} \wedge b_n |_{\overline{v}}) \vee \\
&\qquad (b_1 |_v \wedge \cdots \wedge b_n |_v) \vee \overline{(b_1 |_{\overline{v}} \wedge \cdots \wedge b_n |_{\overline{v}})} \\
&\equiv \quad \overline{(b_1 |_v)} \vee \cdots \vee \overline{(b_n |_v)} \vee (b_1 |_v \wedge \cdots \wedge b_n |_v) \vee \overline{(b_1 |_{\overline{v}})} \vee \cdots \vee \overline{(b_n |_{\overline{v}})} \\
&\equiv \quad \overline{(b_1 |_v \wedge \cdots \wedge b_n |_v)} \vee (b_1 |_v \wedge \cdots \wedge b_n |_v) \vee \overline{(b_1 |_{\overline{v}})} \vee \cdots \vee \overline{(b_n |_{\overline{v}})} \\
&\equiv \quad \texttt{True}. \qquad (3)
\end{aligned}$$

Every assignment of values to variables of β_v which falsifies β_v also falsifies ψ_v. Therefore, if $\beta_v \equiv$ `False`, then $\varphi |_v$ is satisfiable if and only if φ is satisfiable. □

Lemma 4 is the basis for the following theorem.

Theorem 3. *Given a conjunction of Boolean functions $\varphi = b_1 \wedge \cdots \wedge b_m$ and variable v occurring in one or more Boolean functions of φ, let φ_1 be the conjunction of all Boolean functions in φ which contain v. Without loss of generality, suppose $\varphi_1 = b_1 \wedge \cdots \wedge b_n$ where $n \leq m$.*

1. *If for every $1 \leq j \leq n$, $(\overline{b_j |_v} \wedge b_j |_{\overline{v}}) \equiv$ `False`, then $\varphi |_v$ is satisfiable if and only if φ is satisfiable.*

2. *If for every $1 \leq j \leq n$, $(b_j |_v \wedge \overline{b_j |_{\overline{v}}}) \equiv$ `False`, then $\varphi |_{\overline{v}}$ is satisfiable if and only if φ is satisfiable.*

Proof. We consider only case 1; case 2 is analogous. From Lemma 4, if $\beta_v \equiv$ `False` then $\varphi |_v$ is satisfiable if and only if φ is satisfiable. For β_v to be `False`, every term in β_v must be `False`. Therefore, if $\forall_{1 \leq j \leq n}, (\overline{b_j |_v} \wedge b_j |_{\overline{v}}) \equiv$ `False`, then $\varphi |_v$ is satisfiable if and only if φ is satisfiable. □

If Boolean functions are represented as BDDs, then the `safe_assign` operation of Figure 2 may be used to find a safe assignment to a single variable occurring in multiple BDDs. The operation combines both cases of Theorem 3 and entails the application of `safe_assign`$_0$ (see Figure 1) to every BDD containing v, individually. If the same assignment for v is returned by every resulting BDD, then that assignment is safe for v; otherwise,

```
Input:  A set of BDDs $\varphi = \{b_1, \cdots, b_m\}$ and a variable $v$.
Output:  True if $v$ does not occur in $\varphi$,
        a safe assignment to variable $v$, if one exists,
        otherwise, False.

assignment safe_assign($\varphi, v$)
  let BDD $safeVal$ := True
  for($j$ := 1 to $m$) {
    $safeVal$ := and(safe_assign$_0$($b_j$, $v$), $safeVal$)
    if $safeVal$ = False then return False
  }
  return $safeVal$
}
```

Figure 2. Pseudo-code to search for a safe assignment to a variable in more than one BDD.

nothing can be said about v. The implementation may return before considering all functions of φ_1 because a safe assignment can be found only if every term of either β_v or $\beta_{\overline{v}}$ is `False`. Therefore, in some cases little work is done by `safe_assign`.

We remark that the thrust of this section parallels ideas arising from generalizations of the pure literal rule [6]: a literal whose complement does not exist in a conjunction of clauses is called *pure* and any pure literal can be assigned the value `True` for all extensions of the current assignment without affecting the correctness of the search.

4. Multiple Variable Safe Assignments

This section presents some elementary conditions under which a subset of variables may be safely assigned values in a formula expressed as a conjunction of Boolean functions. We also introduce an operation called `safe_search` that is able to find safe assignments for a subset of variables when the Boolean functions are expressed as BDDs.

Finding safe assignments for subsets of variables provides an additional, potentially useful tool for both constraint-oriented and search-oriented SAT solving. For example, if all variables of V' are in a single BDD, a constraint-oriented method might existentially quantify away all variables in V', individually but, if a safe assignment M' is known for V', it may be advantageous to apply some or all of those assignments to variables of V' instead of existentially quantifying all of them away since assignments will reduce an expression to a simpler one. Perhaps surprisingly, we show that M' can be computed without determining whether $\varphi\mid_{M'} \equiv \exists V'(\varphi)$ for all possible assignments to V'.

First, we need to say what we mean by a safe assignment for a subset of variables.

Definition 2. *Let V be the set of Boolean variables occurring in φ, let V' be a subset of V, suppose $|V'| = k$, let φ_1 be the conjunction of all Boolean functions of φ containing at least one variable in V', and let $\mathcal{M}' = \{M'_1, \cdots, M'_{2^k}\}$ be the set of all possible truth assignments* *to the variables in V'. For all $1 \leq i \leq 2^k$ define*

$$\hat{\varphi}_i = (\varphi_1 \mid_{M_1'} \vee \cdots \vee \varphi_1 \mid_{M_{i-1}'} \vee \varphi_1 \mid_{M_{i+1}'} \vee \cdots \vee \varphi_1 \mid_{M_{2^k}'}).$$

Extending Definition 1 to multiple variables, we say M_i' is a safe assignment *for V' in φ if $(\hat{\varphi}_i \rightarrow \varphi_1 \mid_{M_i'}) \;\equiv\;$* `True`.

If $(\hat{\varphi}_i \rightarrow \varphi_1 \mid_{M_i'}) \;\equiv\;$ `True` then applying M_i' to φ will not affect the Satisfiability of φ since the variables in V' exist only in φ_1. Therefore, Definition 2 is sufficient to encapsulate the concept of 'safe' assignments for a subset of variables.

We show how to find safe assignments for variables of V' collectively for two cases: 1) the variables of V' occur in only one function, and 2) the variables of V' occur in multiple functions. Case 1 is considered first. The next two lemmas provide the counterparts to Lemmas 1 and 2 of Section 3. In the proof and elsewhere in the article we use $\varphi_1 \oplus \varphi_2$ to mean $(\varphi_1 \wedge \overline{\varphi_2}) \vee (\overline{\varphi_1} \wedge \varphi_2)$

Lemma 5. *Given $\varphi = b_1 \wedge \cdots \wedge b_m$ and a subset of variables $V' = \{v_1, \cdots, v_k\}$ occurring in φ, let $\mathcal{M}' = \{M_1', \cdots, M_{2^k}'\}$ be the set of all possible truth assignments to the variables in V'. Recall that φ_1 denotes the conjunction of all b_is containing at least one variable in V'. For any $1 \leq i \leq 2^k$, if $(\overline{\varphi_1 \mid_{M_i'}} \wedge \hat{\varphi}_i) \equiv$* `False` *then $\varphi \mid_{M_i'}$ is satisfiable if and only if φ is satisfiable, where $\hat{\varphi}_i$ is given in Definition 2.*

Proof. Re-index the functions of φ so that $\varphi_1 = b_1 \wedge \cdots \wedge b_n$ where $n \leq m$. For any $1 \leq i \leq 2^k$, let $\varphi \mid_{M_i'} = \varphi_1 \mid_{M_i'} \wedge b_{n+1} \wedge \cdots \wedge b_m$. Recall that

$$\exists V'(\varphi) \;\equiv\; (\varphi_1 \mid_{M_1'} \vee \cdots \vee \varphi_1 \mid_{M_{2^k}'}) \wedge b_{n+1} \wedge \cdots \wedge b_m$$

which is satisfiable if and only if φ is satisfiable (Page 94). Then

$$\begin{aligned}
(\varphi \mid_{M_i'} \oplus \exists V'(\varphi)) \;&\equiv\; ((\varphi \mid_{M_i'} \wedge \overline{\exists V'(\varphi)}) \vee (\overline{\varphi \mid_{M_i'}} \wedge \exists V'(\varphi)) \\
&\equiv\; \texttt{False} \vee ((\overline{\varphi_1 \mid_{M_i'} \wedge b_{n+1} \wedge \cdots \wedge b_m}) \wedge \\
&\quad\; (\varphi_1 \mid_{M_1'} \vee \cdots \vee \varphi_1 \mid_{M_{2^k}'}) \wedge (b_{n+1} \wedge \cdots \wedge b_m)) \\
&\equiv\; (\varphi_1 \mid_{M_1'} \vee \cdots \vee \varphi_1 \mid_{M_{i-1}'} \vee \varphi_1 \mid_{M_{i+1}'} \vee \cdots \vee \varphi_1 \mid_{M_{2^k}'}) \wedge \\
&\quad\; \overline{(\varphi_1 \mid_{M_i'})} \wedge (b_{n+1} \wedge \cdots \wedge b_m) \\
&\equiv\; (\overline{\varphi_1 \mid_{M_i'}} \wedge \hat{\varphi}_i) \wedge (b_{n+1} \wedge \cdots \wedge b_m). \qquad (4)
\end{aligned}$$

From (4), for any $1 \leq i \leq 2^k$, if $(\overline{\varphi_1 \mid_{M_i'}} \wedge \hat{\varphi}_i) \equiv$ `False` then $\varphi \mid_{M_i'} \equiv \exists V'(\varphi)$.

It follows that for any $1 \leq i \leq 2^k$, if $(\overline{\varphi_1 \mid_{M_i'}} \wedge \hat{\varphi}_i) \equiv$ `False` then $\varphi \mid_{M_i'}$ is satisfiable if and only if φ is satisfiable. □

Lemma 6. *Let M_i' be an assignment satisfying the hypothesis of Lemma 5. Then M_i' is a safe assignment for V'.*

Proof. Analogous to Lemma 2. □

A BDD-based operation for finding a set of k variables that has a safe assignment may be implemented based on Lemma 5. However, there are two problems with such an approach. First, there are $\binom{|V|}{k}2^k$ possible assignments to check. Second, and potentially much more serious, as in the case of Lemma 1, every BDD in φ_1 must be clustered into one BDD.

Fortunately, something can be done about the latter point; we next show how to find a safe assignment of k variables *without clustering any* of the BDDs containing them. This is accomplished by *distributing* the existential quantification of all variables of V' over every BDD of φ_1. This is interesting because existential quantification cannot generally be distributed in this way. In other words, we present a way to achieve more than existential quantification (namely, the safe assignment of values) without incurring a penalty due to its inherit inefficiency. The next lemma is independent of BDDs but we use it to implement an improved algorithm for finding a safe assignment to a subset of variables when the input formula is expressed as a collection of BDDs.

Lemma 7. *Given $\varphi = b_1 \wedge \cdots \wedge b_m$ and a subset of variables $V' = \{v_1, \cdots, v_k\}$ occurring in φ, let M'_d be an assignment to $\{v_1, ..., v_{d-1}\}$ which is safe for $\varphi'_d = \exists v_{d+1}, ..., v_k(b_1) \wedge \exists v_{d+1}, ..., v_k(b_2) \wedge ... \wedge \exists v_{d+1}, ..., v_k(b_m)$. Then, for $1 \leq d \leq k$,*

1. *If $v_d \mapsto$ `True` is safe for $\varphi'_d \mid_{M'_d} = \exists v_{d+1}, ..., v_k(b_1 \mid_{M'_d}) \wedge \exists v_{d+1}, ..., v_k(b_2 \mid_{M'_d}) \wedge ... \wedge \exists v_{d+1}, ..., v_k(b_m \mid_{M'_d})$, then $M'_d \cup \{v_d \mapsto$ `True`$\}$ is safe for φ'_d.*

2. *If $v_d \mapsto$ `False` is safe for $\varphi'_d \mid_{M'_d} = \exists v_{d+1}, ..., v_k(b_1 \mid_{M'_d}) \wedge \exists v_{d+1}, ..., v_k(b_2 \mid_{M'_d}) \wedge ... \wedge \exists v_{d+1}, ..., v_k(b_m \mid_{M'_d})$, then $M'_d \cup \{v_d \mapsto$ `False`$\}$ is safe for φ'_d.*

Proof. Suppose M'_d is safe for φ'_d. Then $\varphi'_d \mid_{M'_d}$ is satisfiable if φ'_d is. If $v_d \mapsto$ `True` is safe for $\varphi'_d \mid_{M'_d}$ then there is an assignment to variables of φ'_d which includes $M'_d \cup \{v_d \mapsto$ `True`$\}$ and satisfies φ'_d. Similarly for $v_d \mapsto$ `False`. □

Lemma 7 tells us what we can do if v_d is safe for $\varphi'_d \mid_{M'_d}$ but does not say how to determine if v_d is safe for $\varphi'_d \mid_{M'_d}$. We use Theorem 3 for that purpose. Based on Lemma 7 and Theorem 3, an algorithm, called `safe_search`, for finding a safe assignment to a subset of variables in multiple BDDs is given in Figure 3. The `safe_search` algorithm calls the recursive `find_safe_assign` procedure (also shown in Figure 3) which extends a potentially safe assignment for $\{v_1, ..., v_{d-1}\}$ to a safe assignment for $\{v_1, ..., v_d\}$. The `find_safe_assign` procedure performs a search capable of testing all assignments to variables of V'. But each test is relatively efficient because its utilizes the memoized φ'_d functions which themselves are constructed by distributing computation over input BDDs.

Actually, `safe_search` is intended to answer the question: Given a set of BDDs and a set of variables, is satisfiability preserved if the existential quantification of the variables is distributed over the BDDs containing them? If the answer is yes, then *as an artifact* `safe_search` returns a safe assignment to these variables. The user may choose whether to apply that assignment directly, distribute the existential quantification of these variables, or do a mixture of the two. This point is revisited in the last paragraph of this subsection. If the answer is no, safe search returns "unknown."

Theorem 4. *If an assignment τ to V' is returned by* `find_safe_assign`*, and therefore by* `safe_search`*, then τ is a safe assignment for V' in φ.*

Proof. The φ'_d of `safe_search`, computed prior to the first invocation of `find_safe_assign`, are the same φ'_d of Lemma 7 except that the M'_d assignment has not been set. By induction, on the dth recursion of `find_safe_assign`, M'_d is an assignment to $\{v_1, ..., v_{d-1}\}$ which is safe for φ'_d provided the assignment made to v_d is safe for $\varphi'_d\mid_{M'_d}$. But this is decided by the first `for` loop of `find_safe_assign` plus one line above it. But this is the same code as the `safe_assign` code of Figure 2. If the value of *safeVal* is not `False`, it is either *i.)* a 3-node BDD rooted at v_d which expresses a value for v_d or *ii.)* `True` which expresses that both values, `True` and `False`, are safe values for v_d. By Theorem 3 and the fact that the code is applied to $b_j\mid_{M'_d}$ for all j, this value is safe for $\varphi'_d\mid_{M'_d}$. □

The complexity of `safe_search` is $O(k^2m)$ for preprocessing and $O(2^km)$ for finding the safe assignment.

The following remarks apply to `safe_search` and `find_safe_assign`.

1. The precomputation and memoization of the existential quantifications in `safe_search` is important since each recursive step would take a lot longer if the quantifications were not precomputed.

2. If `safe_search` returns "unknown" and V' consists of all variables of φ then φ is unsatisfiable. In fact, in this case, `safe_search` performs a search over all assignments and therefore can act as a complete SAT solver.

3. `safe_search` may miss finding a safe assignment when at least one exists. An example is shown in Figure 4.

4. In `safe_search`, variables are considered one at a time in any arbitrary order (that is, the search for a safe assignment does not need to consider all possible orders).

5. The order in which `safe_assign` considers variables may affect the speed with which a safe assignment is found.

6. We reiterate, it may be possible, using `safe_search`, to find safe assignments to a set of variables that could not be found using `safe_assign` on all those variables individually. An example is shown in Figure 5.

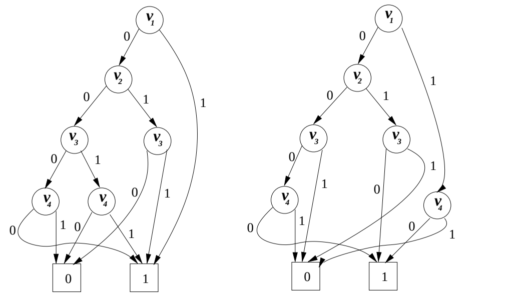
We emphasize that the results of this section afford the choice, for each variable in a given subset, of existentially quantifying it away or applying its safe assignment, *interchangeably.* For example, let $V' = \{v_1, v_2\}$ and let $M'_1 = \{v_1 \mapsto \texttt{True},\ v_2 \mapsto \texttt{False}\}$ where M'_1 is a safe assignment to φ with $\varphi_1 = b_1 \wedge b_2$ and both b_1 and b_2 contain at least one variable of V'. Then

$$\begin{aligned} \varphi\mid_{M'_1} &\equiv b_1\mid_{M'_1} \wedge b_2\mid_{M'_1} \wedge \cdots \\ &\equiv \exists v_1, v_2(b_1) \wedge \exists v_1, v_2(b_2) \wedge \cdots \\ &\equiv \exists v_1(b_1\mid_{\overline{v_2}}) \wedge \exists v_2(b_2\mid_{v_1}) \wedge \cdots, \end{aligned}$$

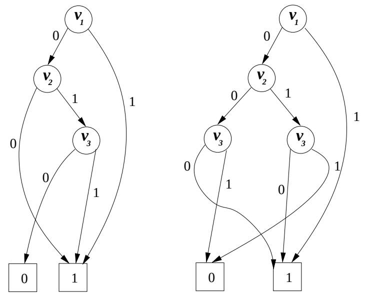
so any permutation of existential quantifications and assignments of variables of V' applied to each individual function of φ_1 is valid.

```
Input:   A set of BDDs $\varphi = \{b_1, \cdots, b_m\}$,
        a set of variables $V' = \{v_1, \cdots, v_k\}$,
Output:  A safe assignment to variables $V'$, if one is
        found, ‘‘unknown’’ otherwise.

assignment safe_search($\varphi$, $V'$) {
  for(d := 1 to k-1) {
    let $\varphi'_d := \{b_{1,d} :=$exist$(\{v_{d+1}, \cdots, v_k\}, b_1), \cdots,$
              $b_{m,d} :=$exist$(\{v_{d+1}, \cdots, v_k\}, b_m)\}$
  }
  let $\varphi'_k := \varphi$
  let $\Phi := \{\varphi'_1, \cdots, \varphi'_k\}$
  let $d := 1$
  let $M'_0$ be an empty assignment to $V'$
  return find_safe_assign($\Phi$, $V'$, $M'_0$, $d$) {
}

assignment find_safe_assign($\Phi$, $V'$, $M'_d$, $d$) {
  if($d = k + 1$) return $M'_d$
  apply $M'_d$ to $\{v_1, ..., v_{d-1}\}$ of $V'$
  let BDD $safeVal :=$ True
  for($j :=$ 1 to $m$) {
    $safeVal :=$ and(safe_assign$_0$($b_{j,d}$,$v_d$), $safeVal$)
    if($safeVal =$ False) return ‘‘unknown’’
  }
  if(and($safeVal$ , var($v_d$)) = var($v_d$)){
    let $M'' :=$ find_safe_assign($\Phi$, $V'$, ($M'_d$ $\cup$ $\{v_d \mapsto$ True$\}$), $d + 1$)
    if($M'' \neq$ ‘‘unknown’’) return $M''$
  }
  if(and($safeVal$, not(var($v_d$)))) = not(var($v_d$)))){
    let $M'' :=$ find_safe_assign($\Phi$, $V'$, ($M'_d$ $\cup$ $\{v_d \mapsto$ False$\}$), $d + 1$)
    if($M'' \neq$ ‘‘unknown’’) return $M''$
  }
  return ‘‘unknown’’
}
```

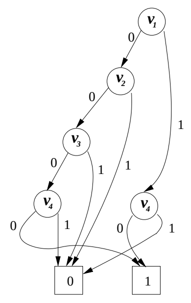
Figure 3. Recursive pseudo-code to search for a safe assignment to a set of variables V' in a conjunction of BDDs φ.



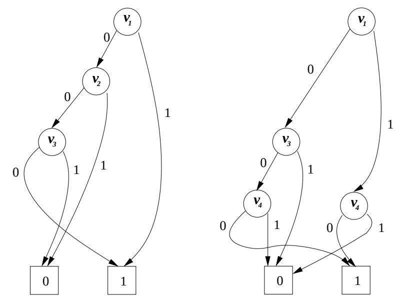
a) Suppose the two BDDs shown above represent φ_1 with $V' = \{v_2, v_4\}$. Call the left BDD b_1 and the right BDD b_2.



b) Result of applying `safe_assign` to φ_1 of a) above. First, $\exists v_4(b_1)$ (shown left) and $\exists v_4(b_2)$ (shown right) are computed. The return values of `safe_assign`$_0$ on these are $v_2 =$ `False` (left) and `False` (right). Therefore, `safe_assign` does not find a safe assignment for φ_1.

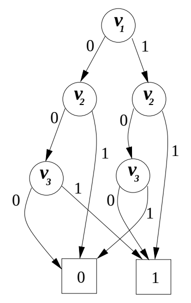


c) However, if b_1 and b_2 are conjoined, φ_1 is the single BDD shown above...



d) then eliminating v_4 gives the BDD on the left and `safe_assign`$_0$ returns $v_2 =$ `False`. Applying this to the BDD of c) gives the BDD shown on the right, on which `safe_assign`$_0$ returns $v_4 =$ `False`. Therefore, there is a safe assignment, namely $v_2 = v_4 =$ `False` in this case.

Figure 4. Example showing that `safe_search` may miss finding a safe assignment when at least one exists. Frames a) and b) show the result of applying `safe_search` to a simple example: no safe assignment is found. Frames c) and d) demonstrate that a safe assignment exists, and that `safe_search` finds it if the BDDs of φ_1 are conjoined first.



Suppose the BDD shown to the left is the only BDD of φ. `safe_assign` will not find any safe assignments given v_1, v_2, or v_3. `safe_search` given $V' = \{v_1, v_2\}$ will find the safe assignment $v_1 = v_2 =$ `True`. `safe_search` given $V' = \{v_1, v_3\}$ will find the safe assignment $v_1 =$ `True` and $v_3 =$ `False`. `safe_search` given $V' = \{v_2, v_3\}$ will not find any safe assignments.

Figure 5. Example showing that, in certain cases, it is possible, using `safe_search`, to find safe assignments to a set of variables that could not be found using `safe_assign` on all those variables individually.

5. Relationship to Autarkies

The notion of an autark assignment first appeared for CNF formulae in [13]. Let φ be a CNF formula containing literals taken from a set V of variables. Suppose M' is some assignment of values to a subset $V' \subset V$ of variables and that φ' is the CNF formula (said to be a residual formula) that remains after clauses satisfied by and literals falsified by M' are removed from φ. Then an assignment M'' of values to a subset $V'' \subseteq V \setminus V'$ of variables is said to be *autark with respect to* M' if all clauses of φ' which are not satisfied by M'' contain no literals taken from V''.

The importance of autark assignments during search is that, if one is found with respect to the assignment yielding the current residual formula φ', then the values of the variables of the autark assignment can be fixed for all extensions of that assignment, not including the variables of the autark assignment. An elementary example of this is known as the pure literal rule, briefly discussed in Section 3. Observe this is different from existentially quantifying the autark variables away in that an actual assignment to those variables is imposed. Yet, observe also that the values of autark variables are not inferred.

One straightforward way to extend the notion of autarkies to conjunctions of BDDs could require that an assignment satisfy every BDD containing a variable in the assignment. The safe assignments described in this paper do not require each entire BDD to be satisfied, just the paths of each BDD which are dependent upon the assignment.

Another extension of autarkies to BDDs involves representing a set of clauses as an individual BDD. Any autarky that exists for a set of CNF clauses will also exist as a safe assignment in the corresponding BDD. However, since BDDs are canonical (Page 92), a safe assignment may exist in a BDD but not exist in some of the, possibly many, corresponding logically equivalent sets of CNF clauses. This means that if a safe assignment exists for a Boolean function represented as a BDD, it will be found immediately. Whereas the autarky is not guaranteed to be found using a CNF representation of the same Boolean function. Therefore, the idea of finding safe assignments to a set of variables in conjunctions of BDDs is potentially more powerful than finding autarkies in CNF formulae.

6. Experimental Validation

We wish to show that the safe assignment tools described here help the early quantification method by reducing on a variable before it has been totally clustered into a single BDD. Early quantification entails the existential quantification of variables while clustering BDDs in search for the monolithic BDD. Doing so is useful because it tends to control the growth of intermediate BDDs during the clustering process. In this section we show, by experiment, that the methods of safely assigning values to some of these variables can control intermediate BDD growth even further without introducing significant overhead.

The particular clustering schedule we used is a greedy one in which a variable v occurring in the *least* number of BDDs is chosen and clustering continues with focus on getting that variable into one BDD so it can be existentially quantified away. If more than one variable occurs in the same number of BDDs, the tie is broken by choosing a variable at random from among the pool. If clustering is successful, v is existentially quantified away from its BDD. Clustering may not be successful if an intermediate BDD is created whose size exceeds a certain threshold which is determined by the amount of memory available on the host. If no further clustering is possible, it is restarted, possibly several times, each time removing all nodes that are not in the original set of BDDs, recording and applying all newly learned implications, and increasing the threshold if no new implications have recently been found. Controlling the threshold in this way allows for the near exhaustion of possible implications while also keeping memory usage low. By using this clustering schedule, we found it becomes possible to cluster BDDs with a large number of variables.

Runs were made on a collection of benchmarks as listed in Table 1. These are taken from the ISCAS'85 suite. They are miter circuits which are used for checking whether two functions are equivalent. A good property of these benchmarks is that the maximum number of variables in an input BDD is rather limited. This allows us to collect results without complications due some ancillary operations that would otherwise become necessary, such as splitting BDDs. Since the benchmarks represent circuits, numerous variables, corresponding to internal circuit points, are dependent and therefore, intuitively, can be easily existentially quantified away. Thus, the benchmarks are good candidates for testing early quantification and safe assignments. The benchmarks used are all unsatisfiable.

For each benchmark, 100 runs were made. Each run was started with a different random seed for tie breaking, as described above. Doing so mitigates the effect of clustering choices on the results. The results under the heading **w/o S** are obtained using the above clustering schedule without making safe assignments. There are three columns of results: the average runtime of completed runs, the number of completed runs (in parentheses), and the average size of an intermediate BDD in any of the 100 runs (under the heading **BDD Size**).

To test the effect of safe assignments, we interleave the `safe_assign()` operation with clustering, as described above, in the following way: immediately after any two BDDs have been clustered, we run `safe_assign()` on all the variables occurring in the combined BDD. Safe assignments are recorded as implications and are inferred immediately after each restart. Intuitively, the clustering algorithm above supports safe assignments in this way; however, the extra implications delay the threshold increase, sometimes causing an increase in execution time. Moreover, the interleaving of clustering and making safe assignments is easy to implement and seems to work well. However, we emphasize that safe assignments can be used in any schema involving early quantification. Results are shown in Table 1 under the heading **w/ S**. In addition, there is a column with the heading **Avg. safe assigns** where the average number of safe assignments found is reported. This may be contrasted with the column of heading **# vars** which shows the number of variables in each benchmark.

The results of Table 1 show that between 1% and 5% of these benchmark's total variables can be safely assigned, using the technique described in this paper, during early quantification as we have implemented it. Probably the most significant effect of applying these safe assignments is that roughly twice as many normal implications were found during the clustering process. On average, this kept the number of variables per BDD lower by up to 30%. Since smaller BDDs were created, RAM usage was kept generally lower, solving times were generally significantly better on hard benchmarks which were generally able to be solved more successfully.

The effectiveness of safe assignments can be better appreciated by the results shown in Figure 6 which compare RAM usage and inference generation as the c7552-s benchmark is being solved. The results show that RAM usage with safe assignments levels off at about the same time that finding safe assignments and making inferences from them begin to snowball. This behavior is typical of other benchmarks. Without safe assignments RAM usage rises unabated and in some cases becomes too high to proceed to completion.

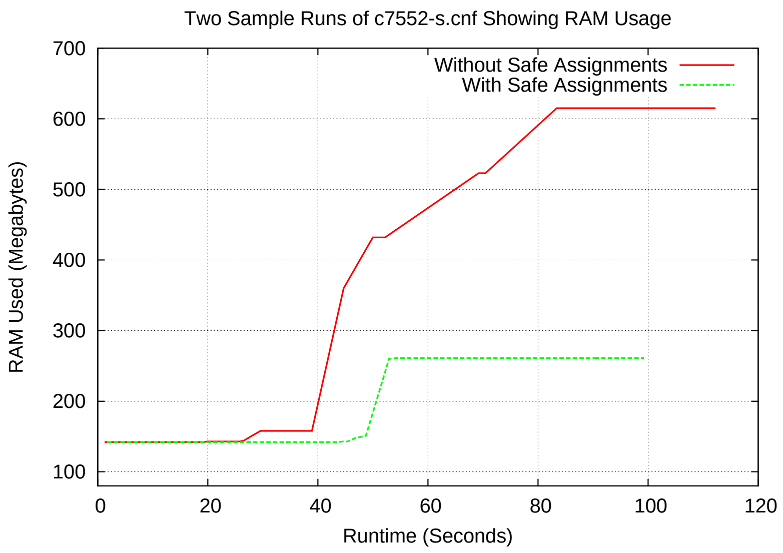
Our experiments have been run using our own BDD tool which is designed to support research and not for speed. In particular, a number of features which make production BDD tools fast are missing, including implementation of complemented edges, and optimal variable ordering algorithms, among others. The results we report here are not intended to show that our BDD tool is competitive. Rather, they are intended to demonstrate the usefulness of adding safe assignments to any BDD tool.

Table 1. Results of experiments on solving a family of benchmarks with and without making safe assignments.

		Avg. safe	**Avg. runtime (s)**				**BDD Size**		**Impl.**	
Benchmark	**# vars**	**assigns**	**w/ S**		**w/o S**		**w/ S**	**w/o S**	**w/ S**	**w/o S**
c499.cnf	606	33.3	0.90	(100)	0.87	(100)	22.0	21.4	454	248
c499-s.cnf	606	33.5	0.86	(100)	0.87	(100)	21.8	21.4	456	242
c880.cnf	957	42.3	6.14	(98)	8.19	(99)	25.0	29.2	605	393
c880-s.cnf	957	41.7	5.74	(99)	9.78	(100)	25.5	29.8	602	392
c1355.cnf	1294	65.4	2.56	(100)	1.32	(100)	22.1	20.7	1067	298
c1355-s.cnf	1294	66.2	2.53	(100)	1.38	(100)	22.3	20.2	1055	306
c1908.cnf	1917	18.7	62.5	(99)	154	(95)	45.3	56.0	1450	1004
c1908-s.cnf	1919	20.5	33.2	(100)	106	(91)	39.2	55.0	1493	1002
c2670.cnf	2703	98.6	5.72	(100)	12.1	(100)	25.9	35.4	2207	1462
c2670-s.cnf	2940	101	6.09	(100)	5.83	(100)	18.8	29.2	2399	1653
c5315.cnf	5399	215	41.47	(100)	31.2	(100)	23.5	28.5	4308	2726
c5315-s.cnf	5408	201	40.48	(99)	36.6	(100)	23.4	28.5	4274	2714
c7552.cnf	7652	216	315	(65)	287	(17)	31.9	43.8	6353	3990
c7552-s.cnf	7767	220	98.8	(95)	122	(73)	27.9	39.3	6535	3938

7. Conclusion

This paper describes some elementary conditions under which one or more variables may be safely assigned values even though no values may be inferred for those variables. Some practical operations based on these conditions are presented. These operations can be integrated into the existing framework of both search-oriented and constraint-oriented methods of satisfiability. An operation suitable for search-oriented methods, and which attempts to



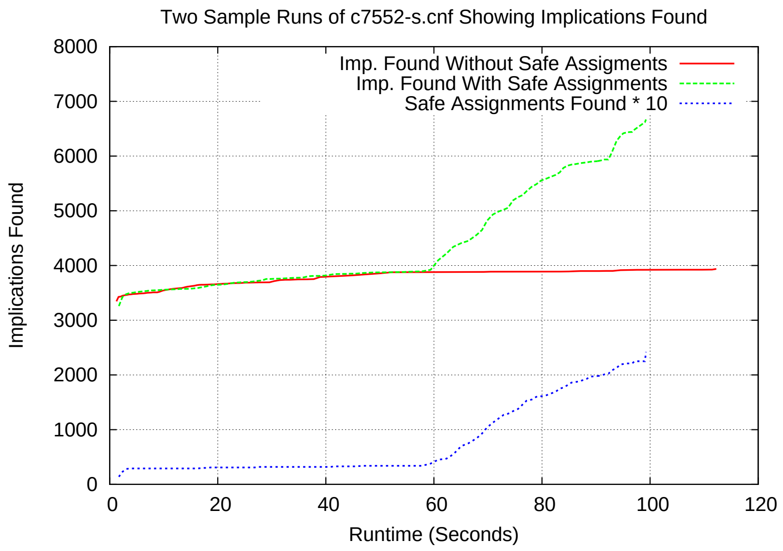


Figure 6. Progressive RAM usage and inference generation for the c7552-s benchmark with and without safe assignments. The top graph shows that RAM usage, both with and without safe assignments, is about the same early in the computation. Then a split occurs. For some time this split diverges as RAM usage for both cases increases. But at some point the RAM usage with safe assignments levels off while RAM usage without safe assignments continues to grow. The reason for this can be seen in the bottom graph where, at the same time the RAM usage levels off, the number of inferences due to safe assignments increases dramatically. The number of safe assignments found is displaced by a factor of 10 so that it can be shown on the graph. Approximately 100 safe assignments appear responsible for each additional 1000 inferences. Data points were collected during restarts.

find a safe assignment to a set of variables, is also presented. All of these operations are relaxations in the use of existential quantification and therefore may fail to find one or more existing safe assignments.

There is virtually no research targeted at finding values for existentially quantified variables before all BDDs have been clustered. We have introduced an efficient operation that can find some values for quantified variables prior to creating the monolithic BDD. Combining this idea with BDD restarting has potential for being a powerful BDD manipulation tool that can be used in SAT preprocessing as well as BDD solving.

References

[1] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers* **C-27**(6): 509–516, 1978.

[2] G. Andersson, , P. Bjesse, B. Cook, and Z. Hanna. A Proof Engine Approach to Solving Combinational Design Automation Problems. *Proc. 39th ACM/IEEE Design Automation Conf.* 725–730, 2002.

[3] K.S. Brace, R.R. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. *Proc. 27th ACM/IEEE Design Automation Conf.* 40–45, 1990.

[4] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.* **C-35**(8): 677–691, 1986.

[5] J. Burch, E. Clark, and D. Long: Symbolic model checking with partitioned transitions relations. In: *Intnl. Conf. on VLSI* (Halaas, A., and Denyer, P.B., eds.), IFIP Transactions, North-Holland 49–58, 1991.

[6] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the Association of Computing Machinery* **5**: 394–397, 1962.

[7] M. Dransfield, and R.E. Bryant. Using ordered binary decision diagrams to solve highly structured satisfiability problems. Unpublished technical report CMU-CS-1996, Carnegie Mellon University, 1996.

[8] J. Franco, M. Dransfield, W.M. Vanfleet, and J.S. Schlipf. State-based propositional Satisfiability Solver. US Patent Application Serial No. 10/164,203, 2005.

[9] J. Franco, M. Kouril, J.S. Schlipf, J. Ward, S. Weaver, M. Dransfield, and W.M. Vanfleet. SBSAT: a state-based, BDD-based Satisfiability solver. *Lecture Notes in Computer Science* **2919**, Springer, New York: 398–410, 2004.

[10] J. Franco, M. Kouril, J.S. Schlipf, S. Weaver, M. Dransfield, and W.M. Vanfleet. Function-complete lookahead in support of efficient SAT search heuristics. *Journal of Universal Computer Science* **12**: 1655–1692, 2004.

[11] J.F. Groote. Hiding propositional constants in BDDs. *Formal Methods in System Design* **8**: 91–96, 1996.

[12] C.Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal* **38**: 985–999, 1959.

[13] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics* **10**: 117–133, 1983.

[14] G. Pan and M.Y. Vardi. Search vs. symbolic techniques in satisfiability solving. In: *Proc. Seventh International Conference on Theory and Applications of Satisfiability Testing* (SAT 2004).

[15] A. San Miguel Aguirre and M.Y. Vardi. Random 3-SAT and BDDs: The plot thickens further. In: *Principles and Practice of Constraint Programming*: 121-136, 2001.

[16] F. Somenzi. Colorado University Decision Diagram package. Available from http://vlsi.colorado.edu/~fabio/CUDD/.