Using BDDs with Combinations of Theories *

Pascal Fontaine and E. Pascal Gribomont

University of Liège (Belgium)
{pfontain,gribomont}@montefiore.ulg.ac.be

Abstract. A Boolean formula is unsatisfiable if and only if its representing binary decision diagram (BDD) is reduced to the single leaf "false". When BDD variables represent first-order atoms including equalities between terms, uninterpreted predicates or linear arithmetic constraints, a path to the "true" leaf in the BDD might not give a model. So BDDs representing unsatisfiable quantifier-free first-order logic formulas may not reduce to the single leaf "false". Decision procedures for combinations of theories can be used to eliminate all those unsatisfiable paths. In a naive approach every path would be considered; this would be very inefficient. We provide efficient algorithms to find general constraints (connections) from unsatisfiable paths to "true" in the BDD. Adding those connections to the BDD will eliminate many paths to "true" at one go. This procedure also ensures that no unnecessary constraint is added.
In the context of invariant validation, this gives good results when using BDDs with a rich quantifier-free language.

1 Introduction

Binary decision diagrams (BDDs) [4] have been widely and successfully used as a powerful technique for representing very large propositional logic formulas. Unfortunately, their expressive power restricted to propositional logic is often a problem for using them in various fields, included their native application (hardware verification). In the presence of complex atoms[1], containing uninterpreted predicates, equalities, and uninterpreted functions, the problem is usually first reduced to propositional logic. A BDD of the result is then built to check for validity [13].

There were several attempts to extend the expressive power of BDDs towards quantifier-free first-order logic with equality. Goel et al. [9] allow uninterpreted functions and equality to be used in BDDs by introducing variables representing equalities, at the cost of losing canonicity[2]. Groote [13] refines this method by having a better integration of the decision diagrams with this new type of variables. In return the ability to deal with uninterpreted functions is lost. It also

* This work was partially funded by a grant of the "Communauté française de Belgique - Direction de la recherche scientifique - Actions de recherche concertées".

[1] An atom is a quantifier-free first-order logic formula which does not contain Boolean connectives. A literal is an atom or the negation of an atom.

[2] In Boolean logic, two logically equivalent Boolean formulas are represented by the same BDD. A (reduced ordered) binary decision diagram is a canonical representation of a formula.

implies modifications of BDD handling functions. In a similar way, Møller et al. [18] introduce Difference Decision Diagrams which allow to efficiently represent and manipulate a Boolean logic over inequalities of the form $x - y \leq c$. Several BDD based decision procedures have also been given for quantified first-order logic (without equality) [10, 21].

In the context of formal verification, uninterpreted predicates and functions are intensively used, but also interpreted predicates, and more particularly linear arithmetic. Our method combines classical BDD handling with usual first-order satisfiability procedures, based on the Nelson-Oppen combination framework [19, 24] and algorithm [20]. This will not only allow to deal with equality and non-interpreted functions, but also with interpreted terms for some important decidable theories (e.g. linear arithmetic). This technique relies on two efficient procedures to find very general constraints on variables in the BDD. One applies to quantifier-free logic with uninterpreted predicates and functions. The other is a general minimalization method for any incremental decision procedure for interpreted predicates and functions.

In the next section, binary decision diagrams are briefly introduced, as well as the context in which we use them. It will also be shown why the problem of using BDDs with atomic formulas as variables is related to the problem of extracting small unsatisfiable subsets from large sets of literals. After a brief presentation of the main decision procedures for combinations of theories, a modified version of the Nelson-Oppen congruence closure algorithm is introduced. Besides deciding unsatisfiability of conjunctions of first-order literals (with equality), our algorithm also provides a small unsatisfiable subset. Finally, we give a procedure to minimalize unsatisfiable sets of constraints (remove unnecessary constraints) for other decidable theories. This general method is suitable for any incremental decision procedure; more particularly, we present our method to deal with linear arithmetic. Some results will be given before we conclude.

2 BDDs beyond the propositional case

BDDs are nested "if-then-else" formulas represented as directed acyclic graphs. Every BDD has one root and two leaves[3] labeled by "true" and "false". Each node has a high son ("then" case) and a low son ("else" case). A BDD with proposition p labeling its root corresponds to formula "if p then A else B", where A and B are formulas corresponding to the high and low sons respectively.

For instance BDD on Figure 1 represents the formula "if p then (if q then $\neg r$) else q", or in disjunctive normal form $(p \wedge q \wedge \neg r) \vee (\neg p \wedge q)$. Plain lines correspond to the high son ("then" case), whereas dotted lines correspond to the low son ("else" case). We work only with Reduced Ordered BDDs (see [4] for detailed informations on ROBDD).

We use BDDs for the validation of invariants of concurrent algorithms [11, 12]. A formula I is an invariant of a transition system if I initially holds and if every transition τ of the system preserves the invariant, i.e. if formula $\{I\}\tau\{I\}$ is valid. In this context, we are often faced with formulas (called verification conditions) of the form $(h_1 \wedge \ldots h_n) \Rightarrow C$ where $h_1, \ldots h_n, C$ are relatively small quantified

[3] Except when BDDs are reduced to one single leaf, in the special cases of unsatisfiable and valid formulas respectively reduced to the "false" and "true" leaves.

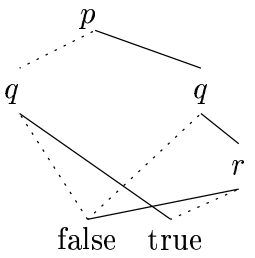

Fig. 1. Propositional

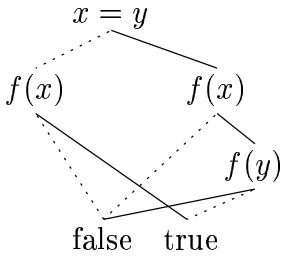

Fig. 2. Unsat. path

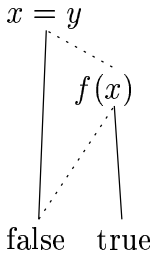

Fig. 3. Reduced

prenex formulas. The number n of hypotheses may be large. To validate such verification conditions our system CAVEAT first skolemizes each subformula, then guesses instances $c_1, \ldots c_m$ of the conclusion C, and finally tries to refute $h_1 \wedge \ldots h_n \wedge \neg(c_1 \vee \ldots c_m)$ where the formula $\neg(c_1 \vee \ldots c_m)$ is quantifier-free.

Here, variables in BDDs can represent any first-order logic atomic formula. We proceed incrementally using an evolving BDD b. Initially b represents formula $\neg(c_1 \vee \ldots c_m)$. While b is not reduced to the single leaf "false", formulas are repeatedly added (that is, *anded*) to this BDD, i.e., $b :=$ bdd_and$(b, \text{formula})$. Those formulas are either instances of hypotheses $h_1, \ldots h_n$ or quantifier-free first-order constraints (for example $x \neq y \vee f(x) \vee \neg f(y)$). The set of all formulas added to b (included the initial formula $\neg(c_1 \vee \ldots c_m)$) will be noted S_b. If the procedure manages to reduce b to the single leaf "false", then S_b is unsatisfiable, and $h_1 \wedge \ldots h_n \wedge \neg(c_1 \vee \ldots c_m)$ has been refuted. Otherwise the procedure gives up when the total number of formulas in S_b reaches a given limit.

Formulas are added to b as follow. If b is not the single leaf "false" there is at least one path to "true" in the BDD. Let P be the set of literals in this path. Either P is satisfiable or it is not. For example consider BDD on Figure 2. There are two paths to the leaf "true". One corresponds to $x \neq y \wedge f(x)$, the other to $x = y \wedge f(x) \wedge \neg f(y)$. The first path is satisfiable, the second is not.

If P is satisfiable, it represents a set of first-order models for the conjunctive set of formulas S_b. The set of literals P is then used as a goal for a hypothesis instantiation procedure. A new instance is generated from one of the hypotheses $h_1, \ldots h_n$, and added to b. In our example, the instantiation procedure would be informed that the next hypothesis instance should help to refute $x \neq y \wedge f(x)$.

If P is unsatisfiable, there is at least one Boolean model for the BDD which does not correspond to any first-order model. In our example, formula $x = y \wedge f(x) \wedge \neg f(y)$ is unsatisfiable. The BDD can be simplified. A Boolean constraint which states that P is unsatisfiable has to be added to b. In our example this constraint is $\neg \bigwedge_{\ell \in P} \ell$. Indeed $\{x = y, f(x), \neg f(y)\}$ is a minimal unsatisfiable set (any proper subset is satisfiable). After adding this constraint we get the BDD on Figure 3.

Taking as a constraint the (negation of the) full conjunction of literals from an unsatisfiable path to the leaf "true" eliminates this path only. Every path would have to be examined. In practice the problem would be intractable with large BDDs, as the number of paths explodes with the size of the BDD. For example, if in a BDD there are two paths to the leaf "true" corresponding to

$x = y \wedge f(x) \wedge \neg f(y) \wedge g(x)$ and $x = y \wedge f(x) \wedge \neg f(y) \wedge h(x)$, both paths could be suppressed by the general constraint $\neg (x = y \wedge f(x) \wedge \neg f(y))$. In our context, practical cases (see Figure 7) show that the same unsatisfiable subpath is often shared by hundreds of paths.

Our methods allow to efficiently extract from a large unsatisfiable set of literals a general connection, containing few literals which could be removed keeping it unsatisfiable. In that sense, a most general connection is a minimal unsatisfiable set. A minimal connection will suppress all paths to "true" which are unsatisfiable because of the same unsatisfiable subpath.

Several different minimal unsatisfiable subsets for the same path may exist. In practice, this occurs rarely: a path to "true" is examined every time a hypothesis is added. Hypotheses are small formulas so they rarely introduce enough information in the BDD for a path to get several different minimal unsatisfiable subsets. If even a path contains two minimal unsatisfiable subpaths S_1 and S_2, there is no strong reason to prefer one or the other[4]: it is very unlikely that every path which contains S_1 also contains S_2. If some path exists which contains S_1 and not S_2, it is useful to give S_1 as a constraint to the BDD.

In our context, generating general constraints enables to use BDDs with variables representing complex atomic formulas, as the number of such constraints to be added to the BDD remains small.

3 Decision procedures for combinations of theories

Before considering the problem of finding small connections in unsatisfiable conjunctive sets, it is useful to remind some facts about the somewhat simpler problem of deciding whether a conjunctive set of literals is satisfiable or not.

In 1954 Ackermann [1] showed that quantifier-free first-order logic with equality is decidable, but did not give a practical algorithm. It is only in the late seventies that this problem has been better understood and that a usable decision procedure has been found by Nelson, Oppen, Downey, Sethi, and Tarjan. It is known as the Nelson-Oppen algorithm (See [20] for the algorithm and early references). This decision procedure and those mentioned below are *restricted to conjunctive sets of literals*. General quantifier-free formulas have first to be put in conjunctive normal forms.

In the meantime, Nelson and Oppen also managed [19] to combine (some) decidable theories with the decision procedures for quantifier-free first-order logic with equality. This allows linear arithmetic to be used together with functions symbols from some other decidable theories and with uninterpreted predicates and functions. This is known as the Nelson-Oppen combination framework.

Shostak [23] improved the Nelson-Oppen algorithm. His new algorithm also internally combined the decision procedure for uninterpreted predicates and functions with some frequently met decidable theories, i. e., theories which have both a canonizer and a solver. Although worst case complexity is not better for Shostak's algorithm than for the Nelson-Oppen algorithm and combination framework, it is agreed that Shostak's algorithm gives better results in practice [7]. But Shostak's algorithm has also got drawbacks. Some theories cannot be

[4] A heuristic could be to take the smallest of both constraints but this would only be a (costly) heuristic.

combined with Shostak's algorithm (although they can be in the Nelson-Oppen framework); for example it is not suitable for linear arithmetic on integers with inequalities. In Shostak's original implementation linear arithmetic inequalities were treated externally in the Nelson-Oppen style. Recent validation tools (for example the Stanford Validity Checker, SVC [16]) try to take the best of both Nelson-Oppen's and Shostak's worlds, and actual works tend to find a better integration between Shostak's algorithm and the Nelson-Oppen combination framework [2, 8]. Finally, it should be noted that completeness of Shostak's algorithm is not trivial. Subtle mistakes in the original algorithm motivated several new versions until very recently [22].

Our decision procedure is based on the Nelson-Oppen framework and algorithm. Inequalities are very often used in the context of program verification, and the Nelson-Oppen framework is the classical way to treat them. The arguments in favour of the Nelson-Oppen congruence algorithm (that is, against the Shostak's algorithm) are more practical. Our problem is more general than deciding satisfiability of a conjunctive set of literals. Slight modifications to the Nelson-Oppen algorithm make it very useful to extract small connections from the set of literals given to the decision procedure. The algorithm we present next is the key to find general constraints in unsatisfiable paths in the BDD, and so avoid exploration of a very large number of paths.

4 An enhanced congruence closure algorithm

Congruence closure algorithm is the cornerstone for a decision procedure for conjunctions of first-order literals (with equality). Given a conjunctive set S of atomic formulas, the algorithm partitions terms and atomic formulas into classes such that two terms (or atomic formulas) are in the same class if and only if the equalities in S entail the equality of the two terms. Once this has been done, S is unsatisfiable if it contains a strict inequality with both members in the same class, or if it contains a pair of complementary literals, that is a positive literal and a negative one with both atoms in the same class.

The congruence algorithm provides one information; two terms are equal according to the equalities in S if and only if they are put in the same class by the algorithm. Changes we propose aim to provide one more information; given two terms in the same class, our version of the algorithm also provides a subset of all equalities which is enough to entail the equality of the two terms.

The technique main procedure, MERGE (see Figure 4), is called in sequence with every equation[5] in S and builds the congruence. It implicitly makes use of an evolving partition $\mathcal{P}$ of all terms. The algorithm interacts with partition $\mathcal{P}$ through function FIND and procedure UNION. FIND(u) returns the *class* of u, i.e. the set in $\mathcal{P}$ which contains u. A call to procedure UNION with u and v as arguments merges the classes for u and v in $\mathcal{P}$.

Before the first call to function MERGE, $\mathcal{P}$ contains a set $\{t\}$ for every term t. Every term has its own class. After function MERGE has been called successively with equations $E_1, \ldots E_n$, two terms t_1, t_2 are in the same class if and only if $E_1, \ldots E_n \models t_1 = t_2$. This is quite straightforward. Partition

[5] MERGE is said to be called with equation $t_1 = t_2$ as argument, if MERGE is called with t_1 as first argument, t_2 as second argument, and set $\{t_1 = t_2\}$ as third argument.

$\mathcal{P}$ determines a reflexive, symmetric and transitive relation between terms. A call to UNION with both members of an equality puts them in the same set in $\mathcal{P}$. Equality propagates also to parents (if they are not already in the same class, FIND(x) $\neq$ FIND(y)) through the recursive call to MERGE (on line 8); P_u = PREDECESSORS(u) is simply the set of all terms which have a direct subterm in the class of u. The call to CONGRUENT(x, y) ensures x and y can be said equal, that is they have the same top function symbol ($\lambda(x) = \lambda(y)$) with the same arity ($\delta(x) = \delta(y)$), and for every i, the (corresponding) i-th terms $x[i]$ and $y[i]$ belong to the same class.

```
procedure MERGE(u, v, S_e);
 1: begin
 2:     if FIND(u)=FIND(v) then return;
 3:     P_u := PREDECESSORS(u);
 4:     P_v := PREDECESSORS(v);
 5:     UNION(u, v) || R_e := R_e ∪ {(u, v, S_e)};
 6:     for each x ∈ P_u do for each y ∈ P_v do
 7:         if FIND(x) ≠ FIND(y) ∧ CONGRUENT(x, y) then
 8:             MERGE(x, y, CONDITIONS(x, y));
 9: end
```

Fig. 4. Procedure MERGE

Our version of the Nelson-Oppen congruence algorithm differs from the original by a new third argument to procedure MERGE, as well as the use of supplementary relation R_e and the new procedure CONDITIONS. In the Nelson-Oppen original congruence closure there is no structure to memorize the given equalities. Beside maintaining the partition $\mathcal{P}$ of terms, our version of the algorithm will also maintain the ternary relation R_e. This relation has some properties useful to obtain a small set of equations which entails equality between two given terms in the same class.

We suppose every call to MERGE(u, v, S_e) is such that $S_e \models u = v$ (the third argument is a set of equations which entails equality between the first two arguments). Relation R_e keeps track of those sets. The new function CONDITIONS (see Figure 4) is used to compute such a set for the recursive call to MERGE. Given two terms t_1 and t_2 such that CONGRUENT(t_1, t_2) is true, a call to CONDITIONS(t_1, t_2) will return a small set of formulas making $t_1 = t_2$ true (Notation $P_{u,v}$ is defined below).

Those basic notations are used in the following. The relation $R_e^\star$ is the reflexive, symmetric and transitive closure of the relation $R'_e =_{\text{def}} \{(u, v) \mid \exists S : (u, v, S) \in R_e\}$. A path from a_1 to a_n in R_e is a sequence of nodes $a_1, \ldots a_n$ such that for every $i \in \{1..n-1\}$, either $(a_i, a_{i+1}) \in R'_e$ or $(a_{i+1}, a_i) \in R'_e$. A

function CONGRUENT(x, y) : Boolean;

1: **var** i : integer;
2: **begin**
3: **return** $\lambda(x) = \lambda(y) \wedge \delta(x) = \delta(y) \wedge$
$\forall i [1 \leq i \leq \delta(x) \Rightarrow \text{FIND}(x[i]) = \text{FIND}(y[i])]$;
4: **end**

Fig. 5. Function CONGRUENT

function CONDITIONS(x, y) : **set of** equations;

1: **var** i : integer;
2: **begin**
3: **return** $\bigcup_{1 \leq i \leq \delta(x)} P_{x[i],y[i]}$;
4: **end**

Fig. 6. Function CONDITIONS

non-looping path is a path $a_1, \ldots a_n$ which does not contains twice (or more) the same node.

Theorem 1. *The following property is an invariant of the algorithm; given two nodes a and b, $(a, b) \in R_e^\star$ if and only if a and b are in the same class.*

Proof. Partition $\mathcal{P}$ of all terms and relation R_e are both changed during the parallel statement of line 5 in MERGE. If terms a and b are in the same class before execution of this statement, they remain in the same class after (as a call to UNION only merges sets in $\mathcal{P}$). Similarly if $(a, b) \in R_e^\star$ before execution of the statement, (a, b) also belongs to $R_e^\star$ after, as no element is removed in R_e.

If the classes of a and b are the same after the union statement but not before, it means that a and b belonged to the class of u or the class of v. Suppose that a and u belonged to the same class (and similarly for b and v). Then (a, u) and (v, b) belonged to $R_e^\star$ before the union statement. As (u, v, S_e) is added to R_e, (a, b) belongs to $R_e^\star$ after this statement. □

Theorem 2. *Given two nodes a and b, there is a path in R_e from a to b if and only if a and b are in the same class. There is at most one non-looping path in R_e going from a to b.*

Proof. The first part is trivial using Theorem 1 and the definitions of class, path, and $R_e^\star$. The second part is easily verified by induction: thanks to condition on line 2 in MERGE, u and v are not in the same class before statement on line 5. So there is no path between u and v. The statement adds a unique direct path between u and v. If it merges classes of a and b, then it adds a unique non-looping path from a to b which is the concatenation of the unique non-looping paths from a to u, from u to v, and from v to b (we supposed a was in the same class as u). □

Given two nodes a_1 and a_n in the same class there is a unique path $a_1, \ldots a_n$ from a_1 to a_n. For each $i \in \{1..n-1\}$, there exists a set S_i such that either $(a_i, a_{i+1}, S_i) \in R_e$ or $(a_{i+1}, a_i, S_i) \in R_e$.

Definition 1. *Given a relation R_e, the set of labels from a_1 to a_n is defined as* $P_{a_1,a_n} =_{\text{def}} \cup_{1 \leq i < n} S_i$.

Theorem 3. *Given two nodes a and b in the same class, $P_{a,b} \models a = b$.*

Proof. This is easily verified by induction. □

Definition 2. *An unsatisfiable set is* minimal *if every proper subset is satisfiable*

Theorem 4. *In the function-free case, $P_{a,b} \cup \{a \neq b\}$ is a minimal unsatisfiable set.*

Proof. In the function-free case, no recursive call is done to function MERGE. Every element in R_e is a triplet of the form $(u, v, \{u = v\})$. To each element $u = v$ in $P_{a,b}$ corresponds an element $(u, v, \{u = v\})$ in R_e. Let $R_e^2 = R_e \setminus \{(u, v, \{u = v\})\}$. As the path from a to b in R_e is unique, there is no path in R_e^2 from a to b. It follows from Theorem 1 that for any $u = v$ in $P_{a,b}$, $P_{a,b} \setminus \{u = v\} \not\models a = b$. □

This last theorem states that the changes made to the original algorithm are more than heuristics; if two elements a and b are found equal, the newly introduced relation R_e allows to find a small set S of equations such that $S \models a = b$. In the function-free case, the set S is minimal. But when functions are used, this is not necessarily the case anymore. Suppose function MERGE is successively called with equations $f(a) = f(b)$, $a = b$, $f(b) = b$ as arguments. Elements $f(a)$ and a are then found equal, and the path from a to $f(a)$ in R_e is labeled by all three equations, although $f(a) = f(b)$ is not necessary to conclude equality between $f(a)$ and a. Anyway it is not mandatory to get a minimal connection as long as the connection does not include too many unnecessary equations. In practice, the connections obtained contain very few elements. The overhead implied by further computation to minimalize them using the following general minimalization algorithm would decrease the overall performance.

5 Finding minimal unsatisfiable sets

Preceding section introduces an enhanced congruence closure algorithm which allows to extract a subset (from a given set of literals) sufficient to prove a given equality. This is one part of a decision procedure for combination of theories which not only decides unsatisfiability of a conjunctive set of literals, but also gives an small unsatisfiable subset. The second part is a method to find small unsatisfiable subsets of literals containing interpreted predicates and functions from a given decidable theory.

The following method applies on any incremental decision procedure for a given theory. For instance, for integer linear arithmetic, we use the LASH tool [15] based on the results in [25]. It provides an incremental decision procedure; to decide if a conjunctive set of linear arithmetic constraints is unsatisfiable, constraints are included one by one to an automaton representing the set of vectors of integers which satisfy already included constraints. This is quite similar to the way a conjunctive set of Boolean formulas is validated using BDDs.

The fact that this decision procedure is incremental gives us a direct method to find a minimal unsatisfiable set of constraints. Let $H = \{h_1, \ldots h_n\}$ be the unsatisfiable set of linear arithmetic constraints from which we would like to extract a minimal unsatisfiable set. Let $H_0 = \emptyset$. For all $i \geq 0$, let k_i be the smallest j such that $H_i \cup \{h_r : 1 \leq r \leq j\}$ is unsatisfiable. An incremental decision procedure directly provides this information; all constraints in H_i are first added, then $h_1, \ldots h_n$ are added one by one until the set of all added constraints is found unsatisfiable. The last added constraint is thus h_{k_i}. Let H_{i+1} be $H_i \cup \{h_{k_i}\}$.

Theorem 5. *For all i,*

- *H_i contains i elements;*
- *$H_i \cup \{h_r : 1 \leq r \leq k_i\}$ is unsatisfiable;*
- *$H_{i+1} \cup \{h_r : 1 \leq r < k_i\}$ is unsatisfiable;*
- *for every $h \in H_{i+1}$, the set $(H_{i+1} \setminus \{h\}) \cup \{h_r : 1 \leq r < k_i\}$ is satisfiable.*
- *the sequence $k_0, k_1, \ldots$ is strictly decreasing.*

Proof. If $H = \{h_1, \ldots h_n\}$ is unsatisfiable, the theorem is true for $i = 0$. Indeed from the definition of k_0, $\{h_r : 1 \leq r \leq k_0\}$ is unsatisfiable. As $H_1 = \{h_{k_0}\}$, the set $H_1 \cup \{h_r : 1 \leq r < k_0\}$ is also unsatisfiable. Finally, $\{h_r : 1 \leq r < k_0\}$ is satisfiable, otherwise k_0 is not the smallest j such that $\{h_r : 1 \leq r \leq j\}$ is unsatisfiable.

By induction, the theorem is also true for $i > 0$. The inductive hypothesis states that $H_i \cup \{h_r : 1 \leq r < k_{i-1}\}$ is unsatisfiable, so k_i exists and is strictly smaller than k_{i-1}. The definition of k_i states that $H_i \cup \{h_r : 1 \leq r \leq k_i\}$ is unsatisfiable. The sets $H_i \cup \{h_r : 1 \leq r \leq k_i\}$ and $H_{i+1} \cup \{h_r : 1 \leq r < k_i\}$ are equal.

Finally, given $h \in H_{i+1}$, either $h \in H_i$ or $h = h_{k_i}$. If $h \in H_i$, the set $H_{i+1} \setminus \{h\} \cup \{h_r : 1 \leq r < k_i\}$ is satisfiable as it is included in $H_i \setminus \{h\} \cup \{h_r : 1 \leq r < k_{i-1}\}$ which is satisfiable. If $h = h_{k_i}$, then $H_{i+1} \setminus \{h\} = H_i$, and $H_{i+1} \setminus \{h\} \cup \{h_r : 1 \leq r < k_i\}$ is satisfiable from the definition of k_i. □

Corollary 1. *Let H_f be the last element of the sequence of sets H_i. H_f is a minimal unsatisfiable set and it contains f elements.*

The practical success of this procedure is due to the fact that, in our case, minimal unsatisfiable sets do contain very few constraints. If m is the maximum number of those necessary constraints in a set H, and if t is the time to decide if H is unsatisfiable, then the time to get minimal set from H is $m \times t$ in the worst case[6]. As m is small, this remains acceptable.

6 Results

Figure 7 presents some results[7] for the validation of formulas coming from the verification of some algorithms. Burns, Dijkstra, Fisher, Ricart & Agrawala, Szymanski are parameterized mutual exclusion algorithms often used as benchmarks for verification tools (See [6, 17] for more information on those algorithms). GRC is a generalized railroad crossing example [14] which has been widely used as benchmark for formal systems (See for example [3]). t_1 and t_2 are total verification times (in seconds) without using our procedures to find small connections,

[6] The procedure is order-sensitive. The worst case corresponds to all necessary constraints being added late.

[7] We used a Pentium 1 GHz running Linux.

and using them respectively (connections are the full unsatisfiable sets in the first case). The verification of one of those algorithm (R & A) is possible in a reasonable time only with our enhancements. The time gain t_1/t_2 is not really representative of the progress brought by our methods, as t_2 also (and mainly) contains instantiation and formula transformation times. The ratio of the number of calls to the decision procedure (n_1/n_2) is a better measure of the improvement; each "minimalized" connection eliminates between 30 and 600 paths to "true" in BDDs. The average number of literals ($\overline{m}_i$) in one path is around 20, with peaks ($m_{i,\max}$) to 55 literals. In these sets of literals, the selected unsatisfiable subset is very small, i.e., 3 elements in average ($\overline{m}_o$), the largest connections ($m_{o,\max}$) contain 5 literals. Those results show that in the case of invariant validation, our method leads to significant improvements.

Algorithm	t_1(s)	t_2(s)	t_1/t_2	n_1	n_2	n_1/n_2	$\overline{m}_i$	$\overline{m}_o$	$m_{i,\max}$	$m_{o,\max}$
Burns	21	4	5.25	5607	40	140	19.65	3	29	4
Dijkstra	75	20	3.75	46651	289	161	15.75	2.79	51	4
Fisher	60	4	15	12033	23	523	10.34	2.95	15	4
R & A	? (> 1 h)	23	?	?	276	?	26.00	2.68	55	5
Szymanski	13	10	1.3	840	25	33	30.40	2.96	47	4
GRC	980	94	10.42	11149	18	619	9.66	3.05	25	4

Fig. 7. Some practical results

7 Conclusion

The success of binary decision diagrams often seems restricted to Boolean logic, or nearly Boolean logic. Indeed, when variables in BDDs represent first-order atomic formulas (and not only Boolean propositions), there can be paths to the "true" leaf which are unsatisfiable. To inspect all those paths may be a huge task, as the number of paths explodes with respect to the size of the BDDs.

We provide methods to extract very general constraints from given (large) unsatisfiable conjunctive sets of literals with equalities, non-interpreted and interpreted predicates and functions. A first procedure presented here is based on an extension of the Nelson-Oppen congruence closure algorithm. It applies on sets of literals containing equalities and uninterpreted predicates and functions. A second procedure is a general minimalization algorithm which applies on any incremental decision procedure. Used together in the Nelson-Oppen combination framework, they provide a simple efficient way to extract small unsatisfiable subsets out of large unsatisfiable sets of literals as easily as checking the (large) set for unsatisfiability. When added to the BDD, those general constraints eliminate many unsatisfiable paths to the "true" leaf at one go. This ensures that a small number of constraints is needed. It also ensures that every constraint found is necessary (as constraints are found from the unsatisfiable paths). And so it provides an efficient way to deal with BDDs on a rich quantifier-free language.

This method has been implemented in our verification tool CAVEAT. Our results clearly show that it is particularly suitable for invariant verification.

References

1. W. Ackermann. *Solvable Cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. C. W. Barrett, D. L. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In *Frontiers of Combining Systems (FROCOS)*, volume 2309 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
3. N. S. Bjørner, Z. Manna, H. B. Sipma, and T. E. Uribe. Deductive verification of real-time systems using STeP. *TCS: Theoretical Computer Science*, 253, 2001.
4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.
5. W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *Proc. 9th Conf. Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 316–327. Springer-Verlag, 1997.
6. K. M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, Reading, Massachusetts, 1988.
7. D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In *Proc. 13th Int. Conf. on Automated Deduction*, volume 1104 of *Lecture Notes in Computer Science*, pages 463–477, New Brunswick, NJ, 1996. Springer-Verlag.
8. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In *Proc. 13th Conf. Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer-Verlag, 2001.
9. A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In *Proc. 10th Conf. Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 244–255. Springer-Verlag, 1998.
10. J. Goubault. Proving with BDDs and control of information. In *Proc. 12th Conf. on Automated Deduction*, volume 814 of *Lecture Notes in Computer Science*, pages 499–513. Springer-Verlag, 1994.
11. E. P. Gribomont and D. Rossetto. CAVEAT : technique and tool for Computer Aided VErification And Transformation. In *Proc. 7th Conf. on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 70–83, Liege, Belgium, 1995. Springer-Verlag.
12. E. P. Gribomont and N. Salloum. Using OBDD's for the validation of Skolem verification conditions. In *Proc. 16th Int. Conf. on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 222–226, Trento, Italy, 1999. Springer-Verlag.
13. J. F. Groote and J. van de Pol. Equational binary decision diagrams. In *Logic Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Computer Science*, pages 161–178. Springer-Verlag, 2000.
14. C. Heitmeyer and N. A. Lynch. The generalized railroad crossing — a case study in formal verification of real-time systems. In *Proceedings 15th IEEE Real-Time Systems Symposium,* San Juan, Puerto Rico, pages 120–131, Dec. 1994.
15. The Liège Automata-based Symbolic Handler (LASH). Available at `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.
16. J. R. Levitt. *Formal Verification Techniques for Digital Systems.* PhD thesis, Stanford University, December 1998.
17. N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, San Francisco, CS, 1996.
18. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 1999.

19. G. Nelson and D. C. Oppen. Simplifications by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.
20. G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.
21. J. Posegga and P. H. Schmitt. Automated deduction with shannon graphs. *Journal of Logic and Computation*, 5(6):697–729, Dec. 1995.
22. H. Rueß and N. Shankar. Deconstructing shostak. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS-01)*, pages 19–28, Los Alamitos, CA, 2001. IEEE Computer Society.
23. R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, Jan. 1984.
24. C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop (Munich, Germany)*, pages 103–120. Kluwer Academic Publishers, 1996.
25. P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *Proc. 6th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, March 2000. Springer-Verlag.