Efficient Generation of Small Interpolants in CNF

Yakir Vizel[1], Vadim Ryvchin[2,3], and Alexander Nadel[3]

[1] Computer Science Department, The Technion, Haifa, Israel
[2] Information Systems Engineering Department, The Technion, Haifa, Israel
[3] Design Technology Solutions Group, Intel Corporation, Haifa, Israel

Abstract. Interpolation-based model checking (ITP) [14] is an efficient and complete model checking procedure. However, for large problems, interpolants generated by ITP might become extremely large, rendering the procedure slow or even intractable.

In this work we present a novel technique for interpolant generation in the context of model checking. The main novelty of our work is that we generate *small* interpolants in *Conjunctive Normal Form (CNF)* using a twofold procedure: First we propose an algorithm that exploits resolution refutation properties to compute an interpolant approximation. Then we introduce an algorithm that takes advantage of inductive reasoning to turn the interpolant approximation into an interpolant. Unlike ITP, our approach maintains only the relevant subset of the resolution refutation. In addition, the second part of the procedure exploits the properties of the model checking problem at hand, in contrast to the general-purpose algorithm used in ITP.

We developed a new interpolation-based model checking algorithm, called CNF-ITP. Our algorithm takes advantage of the smaller interpolants and exploits the fact that the interpolants are given in CNF. We integrated our method into a SAT-based model checker and experimented with a representative subset of the HWMCC'12 benchmark set. Our experiments show that, overall, the interpolants generated by our method are 42 times smaller than those generated by ITP. Our CNF-ITP algorithm outperforms ITP, and at times solves problems that ITP cannot solve. We also compared CNF-ITP to the successful IC3 [3] algorithm. We found that CNF-ITP outperforms IC3 [3] in a large number of cases.

1 Introduction

Model checking is a method for formally verifying that a system satisfies a predefined set of properties. A SAT-solver is a powerful decision procedure used in model checking. While in the early days SAT-based model checking was only used for bug-hunting, nowadays it is a complete procedure and can either prove or refute properties. One such complete SAT-based algorithm uses *Interpolation* [14].

We present a novel approach for interpolant computation in the context of SAT-based model checking. The main contribution of this work is the ability to produce *small* interpolants in *Conjunctive Normal Form* (CNF) efficiently. In order to compute an interpolant, our work takes advantage both of the properties of the resolution refutation, generated by the SAT solver, and of the structure of the model checking problem at hand. In addition, we present *CNF-ITP*, an enhanced version of the original

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 330–346, 2013.

interpolation-based model checking algorithm [14] (ITP). CNF-ITP makes use of the fact that interpolants are given in CNF.

Given a pair of inconsistent propositional formulas $A(X,Y)$ and $B(Y,Z)$, where X, Y and Z are sets of Boolean variables, an interpolant $I(Y)$ is a formula that fulfills the following properties: $A(X,Y) \Rightarrow I(Y)$; $I(Y) \wedge B(Y,Z)$ is unsatisfiable; and $I(Y)$ is a formula over the common variables of $A(X,Y)$ and $B(Y,Z)$ [6]. Modern SAT-solvers are capable of generating an unsatisfiability proof of an unsatisfiable formula. The proof is in the form of a resolution refutation [22,10,16]. It is possible to compute an *interpolant* from a resolution refutation of $A(X,Y) \wedge B(Y,Z)$ [17,14].

Interpolants are used in various domains. The work in [14] was the first to incorporate interpolants into model checking, creating a complete SAT-based algorithm referred to as ITP. ITP uses interpolants to over-approximate image computations. Since [14], interpolants have been applied in several model checking algorithms [11,12,15,20,21].

[14] presents a recursive procedure for interpolant generation from a proof. The procedure initially assigns a propositional formula to each one of the leaves in the resolution refutation (hypothesis clauses). It then recursively assigns a propositional formula to every node in the refutation by either conjoining or disjoining the propositional formulas of its predecessors. Choosing between conjunction or disjunction depends on whether the pivot variable is local to $A(X,Y)$ or not. The formula that is assigned for the empty clause represents the interpolant.

While this algorithm is linear in the size of the proof, the resulting interpolant is a non-CNF propositional formula that mirrors the structure of the resolution refutation. Thus, when the resolution refutation is large, so is the interpolant. Moreover, the resulting formula is often highly redundant, meaning that the interpolant can be simplified and be represented by a smaller formula.

ITP requires the interpolants to be fed back into the SAT solver for computing the next interpolant. Therefore, in those cases where the size of interpolants is large, the resulting SAT problem may be intractable.

We strive to solve this problem by natively generating small interpolants in CNF. One way to compute an interpolant is by *existential quantification*. Considering the unsatisfiable formula $A(X,Y) \wedge B(Y,Z)$, $I(Y) = \exists X(A(X,Y))$ is an interpolant. For a CNF formula $A(X,Y)$, $\exists X(A(X,Y))$ can be created by iteratively applying variable elimination[1] on X variables in $A(X,Y)$. The problem with this approach is that variable elimination is exponential, and, therefore impractical, given a large set of variables.

In this work, we provide a novel resolution-refutation-guided method for variable elimination to derive an interpolant in CNF. This procedure, while creating less clauses than naïve variable elimination procedures, might still result in an exponential blow-up.

Our solution is first to build an *approximated* interpolant $I_w(Y)$ for which $I_w(Y) \wedge B(Y,Z)$ may be satisfiable. We refer to such an interpolant as a B_{weak}*-interpolant*. Computing the B_{weak}-interpolant is based on the method of resolution-refutation-guided variable elimination but is far more efficient. The second stage of our method aims

[1] *Variable elimination* [7] is an operation that replaces all occurrences of a variable v from a CNF formula by replacing clauses containing v with the result of pairwise resolutions between clauses containing the literal v and those containing the literal $\neg v$.

at strengthening $I_w(Y)$ and transforming it into an interpolant $I(Y)$ where $I(Y) \wedge B(Y,Z)$ is unsatisfiable. We refer to this process as *B-Strengthing*.

In order to transform a B_{weak}-interpolant into an interpolant we need to make sure that $A(X,Y) \Rightarrow I_w(Y)$ and that $I_w(Y) \wedge B(Y,Z)$ is unsatisfiable. This can be done by finding all satisfying assignments $s(Y)$ to $I_w(Y) \wedge B(Y,Z)$ and conjoining $\neg s(Y)$ with $I_w(Y)$. Note that an assignment s is a conjunction of literals, and therefore its negation is a clause. By this we keep $I_w(Y)$ in CNF. The number of such assignments may be vast, and therefore this is an inefficient method.

To overcome this, instead of adding a clause to $I_w(Y)$ we *generalize* it to a sub-clause so as to block a larger set of assignments. In order to perform an efficient generalization we use the structure of A. In the context of model checking, $A(V,V') = Q(V) \wedge TR(V,V')$ where V is the set of variables in the checked system and *TR* is the transition relation. Using this fact allows us to perform *inductive generalization* [3].

We implemented CNF-ITP, a model checking algorithm which is a variant of ITP [14], but which uses the above method to compute the interpolants. Our goal was to measure the impact of our interpolant computation method on the underlying model checking algorithm. However, CNF-ITP also exploits the fact that interpolants are given in CNF in order to improve the traditional ITP. Our improvements to ITP were inspired by [3].

For the experiments we used the HWMCC'12 benchmark set. The interpolants computed by our method, compared to those computed by the original ITP algorithm of [14], were much smaller in size overall in the vast majority of cases. Sometimes, the size was up to *two* orders of magnitude smaller. Our procedure significantly outperformed ITP and solved some test cases that ITP could not solve. To complete our experiments, we also compared CNF-ITP to the successful IC3 [3] algorithm. We found that CNF-ITP outperformed IC3 [3] in a large number of cases.

1.1 Related Work

A well-known problem of interpolants is their size. Several works try to deal with this problem. The work in [4] suggests dealing with the increasing size of interpolants by using circuit compaction. While this process can be efficient in some cases, it may consume considerable resources for very large interpolants. Moreover, compacting an interpolant does not result in a CNF formula, whereas our approach results in interpolants in CNF.

As we have already noted, an interpolant computed from a resolution refutation mirrors its structure. Several works [1,18] deal with reductions to the resolution refutation. Since our method uses resolution refutation it too can benefit from such an approach.

During interpolant computation, our approach only uses the relevant parts of the resolution refutation. The idea of holding and maintaining only the relevant parts of the resolution derivation was proposed and proved useful in [19] in the context of group-oriented minimal unsatisfiable core extraction.

Deriving interpolants in CNF was suggested in [12]. The authors suggest applying a set of reordering rules for resolution refutations so that the resulting interpolant will be in CNF. As the authors state in the paper, the described procedure does not always return an interpolant in CNF. Also, the reordering of a resolution refutation may result in an exponential blow up of the proof and, as stated in [8], reordering is not always possible.

In contrast to [12], our method does not rewrite the resolution refutation generated by the SAT solver.

The work in [5] suggests an interpolant computation method that does not use the generated resolution refutation. In addition, an interpolant that results from the use of that method is in a Disjunctive Normal Form (DNF). Our work, on the other hand, uses the resolution refutation and generates interpolants in CNF efficiently.

2 Preliminaries

Throughout the paper we denote the value *false* as $\bot$ and the value *true* as $\top$.

Let V be a set of Boolean variables. For $v \in V$, v' is used to denote the value of v after one time unit. The set of these variables is denoted by V'. In the general case V^i is used to denote the variables in V after i time units (thus, $V^0 = V$). For a propositional formula F over V we write F' to denote the same formula when substituting every occurrence of $v \in V$ in F with $v' \in V'$. In the general case, we write $F(V^i)$ to denote the substitution of every occurrence of $v^j \in V^j$ in F with $v^i \in V^i$ for some non-negative i, j. From now on, all formulas we refer to are *propositional formulas*, unless stated otherwise.

Definition 1. *A* finite transition system *is a triple* $M = (V, \mathrm{INIT}, \mathrm{TR})$ *where* V *is a set of boolean variables,* $\mathrm{INIT}(V)$ *is a formula over* V*, describing the initial states, and* $\mathrm{TR}(V, V')$ *is a formula over* V *and the next-state variables* V'*, describing the transition relation.*

In order to describe a path in a transition system M by means of propositional formulae we define: $path^{i,j} = TR(V^i, V^{i+1}) \wedge \ldots \wedge TR(V^{j-1}, V^j)$ where $0 \leq i < j$. Abusing notation somewhat, we sometimes refer to a propositional formula over V as a set of states in M.

Definition 2 (Conjunctive Normal Form (CNF)). *Given a set* U *of Boolean variables, a literal* l *is a variable* $u \in U$ *or its negation and a* clause *is a set of literals. A formula* F *in CNF is a conjunction of clauses.*

A SAT solver is a complete decision procedure that, given a set of clauses, determines whether the clause set is *satisfiable* or *unsatisfiable*. A clause set is said to be satisfiable if there exists a *satisfying assignment* such that every clause in the set is evaluated to $\top$. If the clause set is satisfiable then the SAT solver returns a satisfying assignment for it. Otherwise the solver produces a *resolution refutation* comprising the proof of unsatisfiability [22,10,16].

For a formula X, $\mathcal{V}(X)$ is the set of variables appearing in X.

Definition 3 (Local and Global Variable). *Let* (A, B) *be a pair of formulas in CNF. A variable* v *is* A*-local (*B*-local) iff* $v \in \mathcal{V}(A) \backslash \mathcal{V}(B)$ *(*$v \in \mathcal{V}(B) \backslash \mathcal{V}(A)$*);* v *is* (A, B)*-global or, simply, global, iff* $v \in \mathcal{V}(A) \cap \mathcal{V}(B)$*.*

Definition 4 (Interpolant). *Let* (A, B) *be a pair of formulas in CNF such that* $A \wedge B \equiv \bot$*. The* interpolant *for* (A, B) *is a formula* I *such that: (i)* $A \Rightarrow I$*. (ii)* $I \wedge B \equiv \bot$*. (iii)* $\mathcal{V}(I) \subseteq \mathcal{V}(A) \cap \mathcal{V}(B)$ *(all the variables in the interpolant are global).*

We will use the notions of weaker versions of interpolants that fulfill two out of three interpolant properties.

Definition 5 (B_{weak}-Interpolant). *Let (A, B) be a pair of formulas in CNF such that $A \wedge B \equiv \bot$. The* B_{weak}-interpolant *for (A, B) is a formula I such that: (i) $A \Rightarrow I$. (ii) $\mathcal{V}(I) \subseteq \mathcal{V}(A) \cap \mathcal{V}(B)$.*

Definition 6 (Non-Global-Interpolant). *Let (A, B) be a pair of formulas in CNF such that $A \wedge B \equiv \bot$. The* non-global-interpolant *for (A, B) is a formula I such that: (i) $A \Rightarrow I$. (ii) $I \wedge B \equiv \bot$.*

Next we provide some resolution-related definitions. The *resolution rule* states that given clauses $\alpha_1 = \beta_1 \vee v$ and $\alpha_2 = \beta_2 \vee \neg v$, where β_1 and β_2 are also clauses, one can derive the clause $\alpha_3 = \beta_1 \vee \beta_2$. Application of the resolution rule is denoted by $\alpha_3 = \alpha_1 \otimes^v \alpha_2$.

Definition 7 (Resolution Derivation). *A* resolution derivation *of a target clause α from a CNF formula $G = \{\alpha_1, \alpha_2, \ldots, \alpha_q\}$ is a sequence*
$\pi = (\alpha_1, \alpha_2, \ldots, \alpha_q, \alpha_{q+1}, \alpha_{q+2}, \ldots, \alpha_p \equiv \alpha)$, where each clause α_i for $i \leq q$ is initial *and α_i for $i > q$ is* derived *by applying the resolution rule to α_j and α_k, where $j, k < i$.*

A resolution derivation π can naturally be conceived of as a directed acyclic graph (DAG) whose vertices correspond to all the clauses of π and in which there is an edge from a clause α_j to a clause α_i iff $\alpha_i = \alpha_j \otimes \alpha_k$. A clause $\beta \in \pi$ is a *parent* of $\alpha \in \pi$ iff there is an edge from β to α. A clause $\beta \in \pi$ is *backward reachable* from $\gamma \in \pi$ if there is a path (of 0 or more edges) from β to γ. The set of all vertices backward reachable from $\beta \in \pi$ is denoted $\Gamma(\pi, \beta)$.

Definition 8 (Resolution Refutation). *A resolution derivation π of the empty clause $\square$ from a CNF formula G is called the resolution refutation of G.*

An interpolant can be produced out of a resolution refutation [14].

For this work, we will need a definition of an A-resolution refutation, that is, a projection of a given resolution refutation π to the clause set A:

Definition 9 (A-Resolution Refutation). *Let $\pi = (\alpha_1, \alpha_2, \ldots, \square)$ be a resolution refutation of the CNF formula $G = A \wedge B$. The* A-resolution refutation *$\pi_A \in \pi$ is constructed by applying the following operation for every clause $\alpha_i \in \pi$ in the order of appearence in π: α_i is appended to π_A iff either $\alpha_i \in A$ or $\alpha_i = \alpha_j \otimes^v \alpha_k$ and $\alpha_j \in \pi_A$ or $\alpha_k \in \pi_A$.*

In DAG terminology π_A is a sub-graph of π that contains only those vertices whose clauses belong to A, and the edges between such clauses. Note that a clause $\alpha \in \pi$ may have 0 or 2 parents, while a clause $\alpha \in \pi_A$ may also have 1 parent (if the second parent is implied only by the clauses of B).

We denote clauses containing the literal v/$\neg v$ in a given clause set by v^+/v^-, respectively. Given a CNF formula F and a variable $v \in \mathcal{V}(F)$, *variable elimination* [7] is an

operation that removes v from F by replacing clauses containing the variable v with the result of a pairwise resolution between v^+ and v^-. The resulting formula $VE(F, v)$ is equisatisfiable with F [7]. The groundbreaking DP algorithm for deciding propositional satisfiability [7] uses variable elimination until either the empty clause $\square$ is derived, in which case the formula is unsatisfiable, or all the variables appear in one polarity only, in which case the formula is satisfiable. It is well known that the original DP algorithm suffers from exponential blow-up.

A bounded version of variable elimination has been an essential contributor to the efficiency of modern SAT preprocessing algorithms (that is, algorithms that truncate the size of the CNF formula before embarking on the search) since the introduction of the SatELite preprocesor [9]. In *bounded variable elimination*, used in SatELite, a variable v is eliminated iff the operation does not increase the number of clauses.

3 Generating Interpolant Approximation in CNF

In this section we propose a method for generating a B_{weak}-interpolant (recall Def. 5) in CNF. First, we briefly describe two algorithms for generating interpolants in CNF. In practice, both algorithms are not applicable to all cases, because of exponential blow-up. Thereafter we introduce an efficient algorithm which is guaranteed to return a B_{weak}-interpolant in CNF, and which may for some cases return an interpolant in CNF.

Our first algorithm for generating an interpolant in CNF is based on naïve variable elimination. First it generates a resolution refutation of the given formula using a SAT solver. Then it initializes the interpolant by those clauses of A that are backward reachable from $\square$ (the empty clause). Note that at this stage I is a non-global-interpolant (recall Def. 6). Finally, the algorithm gradually turns the non-global-interpolant into an interpolant by applying variable elimination over all A-local variables. Consider the example in Fig. 1. Our algorithm would generate the following interpolant: $I = \{g_1 \vee g_2, g_1 \vee g_4, g_3 \vee g_2, g_3 \vee g_4\}$. Unfortunately, the algorithm suffers from the same drawback as the DP algorithm [7]: exponential blow-up when variables keep being eliminated.

Our next algorithm is based on the observation that to eliminate a variable v it is not necessary to apply resolution over all the pairs in v^+ and v^-, but rather only over those subsets that contribute to deriving a common ancestor in the resolution derivation. We need to introduce the notion of clause-interpolant.

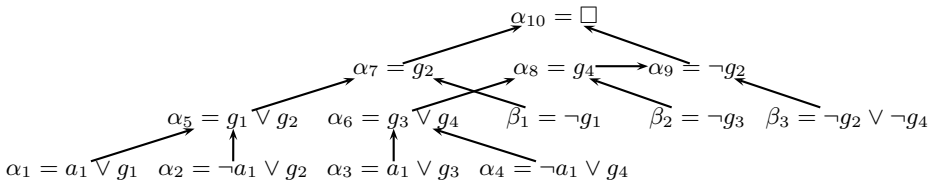

Fig. 1. An example of a resolution refutation. Assume $A = \{\alpha_1, \ldots, \alpha_4\}$ and $B = \{\beta_1, \ldots, \beta_3\}$.

Definition 10 (Clause-Interpolant). *Let (A, B) be an unsatisfiable pair of CNF formulas. Let α be a clause. Then, $I(\alpha)$ is a* Clause-Interpolant *of α iff:*
(i) $A \Rightarrow I(\alpha)$ (ii) $I(\alpha) \wedge B \Rightarrow \alpha$ (iii) $\mathcal{V}(I(\alpha)) \subseteq (\mathcal{V}(A) \cap \mathcal{V}(B)) \cup (\mathcal{V}(A) \cap \mathcal{V}(\alpha))$

A clause-interpolant is a generalization of an interpolant that allows one to associate an interpolant with every clause α in A-resolution refutation (recall Def. 9). As in the case of the standard interpolant, the clause-interpolant is implied by A. The conjunction of the clause-interpolant with B implies the clause α (instead of $\square$ for the standard interpolant). Finally, the clause-interpolant is allowed to contain global variables and A-local variables that appear in α. Note that a clause-interpolant of $\square$ is an interpolant.

Our second proposed algorithm for deriving an interpolant in CNF works as follows: it traverses the A-resolution refutation from the input clauses towards $\square$. It constructs a clause-interpolant for each traversed clause as follows. The clause-interpolant of each initial clause α is set to $\{\alpha\}$. For creating the clause-interpolant of a derived clause α, the algorithm first conjoins the clause-interpolants of α's parents. Then, if α was created by resolution over a local variable v, v is eliminated from the result. The clause-interpolant of $\square$ is returned as the interpolant. Consider again the example in Fig. 1. We have $I(\alpha_5) = \alpha_1 \otimes^{a_1} \alpha_2 = g_1 \vee g_2$; $I(\alpha_6) = \alpha_3 \otimes^{a_1} \alpha_2 = g_3 \vee g_4$; $I(\alpha_7) = I(\alpha_5)$; $I(\alpha_9) = I(\alpha_8) = I(\alpha_6)$. Finally, the interpolant is $I(\square) = I(\alpha_7) \cup I(\alpha_9) = \{g_1 \vee g_2, g_3 \vee g_4\}$. Note that for our example, the interpolant generated by the current algorithm is smaller than the one generated by our previous algorithm, which applies exhaustive variable elimination. In practice, however, the current algorithm is not always scalable either, due to the same problem – exponential blow-up caused by variable elimination. Also note that for our simple example the intepolant comprises a cut $\{\alpha_5, \alpha_6\}$ in the A-resolution refutation, where all the clauses are implied by A only. One can show that whenever such a cut exists it comprises an interpolant. Unfortunately, in the general case such cuts do not usually exist.

Now we are ready to present a scalable algorithm for approximating an interpolant by generating a B_{weak}-interpolant. The first stage of our algorithm traverses the resolution refutation to generate a non-global-interpolant. The second stage uses bounded variable elimination and then incomplete variable elimination (defined below), if required, to convert the non-global-interpolant to a B_{weak}-interpolant.

Definition 11 (Incomplete Variable Elimination). *Given a CNF formula F and a variable $v \in \mathcal{V}(F)$,* incomplete variable elimination *is an operation that removes v from F by replacing clauses containing the variable v with the set IVE(F, v) which contains* some *of the results of a pairwise resolution between v^+ and v^-, where two requirements are met:*

1. *$|IVE(F, v)| \leq |v^+| + |v^-|$*
2. *Let $\alpha \in v^+/v^-$ be a clause, such that there exists a clause $\beta \in v^-/v^+$, such that $\alpha \otimes^v \beta$ is not a tautology. Then, $\alpha \otimes^v \gamma \in IVE(F, v)$ for* at least one *clause $\gamma \in v^-/v^+$, such that $\alpha \otimes^v \gamma$ is not a tautology.*

The idea behind incomplete variable elimination is to omit some of the resolvents when eliminating the variable v in order not to increase the number of clauses, yet to guarantee that each clause containing v has some contribution to the generated set of clauses. Note

that while incomplete variable elimination is not sufficient to maintain unsatisfiability for all cases, it may be sufficient for some cases. Incomplete variable elimination is non-deterministic.

Before presenting our eventual algorithm, we need to introduce the notion of a non-global-clause-interpolant:

Definition 12 (Non-Global-Clause-Interpolant). *Let* (A, B) *be an unsatisfiable pair of CNF formulas. Let* α *be a clause. Then,* $I(\alpha)$ *is a* Non-Global-Clause-Interpolant *of* α *iff: (i)* $A \Rightarrow I(\alpha)$ *(ii)* $I(\alpha) \wedge B \Rightarrow \alpha$

Note that a non-global-clause-interpolant of $\square$ is a non-global-interpolant.

Consider now the algorithm described in Fig. 2. Its first part (lines 2-21) traverses the resolution refutation and asssociates a non-global-clause-interpolant with each clause. Consider a visited clause $\alpha_i = \alpha_j \otimes^v \alpha_k$ when v is local. First, the algorithm sets $I(\alpha_i)$ to be the union of $I(\alpha_j)$ and $I(\alpha_k)$. It eliminates the variable v if the following two conditions hold: First, that eliminating v does not increase the clause size of $I(\alpha_i)$ (as in the bounded variable elimination of SatELite [9]), and second, that variable elimination has been performed for all clauses backward reachable from α_i. (The second condition is ensured by using an auxiliary set *Skipped* for marking clauses for which variable elimination was skipped). The second stage of the algorithm (starting from line 22) uses bounded variable elimination and then incomplete variable elimination to convert the non-global-interpolant to the eventually returned B_{weak}-interpolant by eliminating A-local variables. Note that the bounded variable elimination stage is non-redundant even

```
1: function SIG(π_A = (α_1, α_2, ..., α_q, α_{q+1}, α_{q+2}, ..., α_p ≡ □))
2:     Skipped := {}
3:     for all i ∈ {1, 2, ..., q} do
4:         I(α_i) := {α_i}
5:     end for
6:     for all i ∈ {q + 1, q + 2, ..., p ≡ □} do
7:         if α_i has exactly one parent β then
8:             I(α_i) := I(β)
9:         else
10:            if α_i = α_j ⊗^v α_k, where v is global then
11:                I(α_i) := I(α_j) ∪ I(α_k)
12:            else      // α_i = α_j ⊗^v α_k, where v is A-local
13:                I(α_i) := I(α_j) ∪ I(α_k)
14:                if |VE(I(α_j) ∪ I(α_k), v)| ≤ |I(α_j) ∪ I(α_k)| and {α_j, α_k} ∩ Skipped = ∅ then
15:                    I(α_i) := VE(I(α_i), v)
16:                else
17:                    Skipped := Skipped ∪ {α_i}
18:                end if
19:            end if
20:        end if
21:    end for
22:    Apply bounded variable elimination for A-local variables over I(□)
23:    if I(□) then do not contain A-local variables
24:        return I(□)       // In this case I(□) is an interpolant
25:    else
26:        Apply incomplete variable elimination for A-local variables over I(□)
27:        return I(□)       // In this case I(□) is a B_weak-interpolant
28:    end if
29: end function
```

Fig. 2. B_{weak}-Interpolant Generation

```
1: function ITP(M,p)
2:     if INIT ∧ ¬p == SAT then
3:         return cex
4:     end if
5:     k = 1
6:     while true do
7:         result = ComputeReachable(M, p, k)
8:         if result == fixpoint then
9:             return Valid
10:        else if result == cex then
11:            return cex
12:        end if
13:        k = k + 1
14:    end while
15: end function
```

Fig. 3. Interpolation-Based Model Checking (ITP)

though bounded variable elimination was performed locally for resolution refutation clauses, since sometimes bounded variable elimination is possible given a large set of clauses while it is impossible given a subset of that set. Note also that the algorithm returns an interpolant rather than merely a B_{weak}-interpolant if all the A-local variables are succesfully removed before incomplete variable elimination is applied.

4 Using B_{weak}-Interpolants In Model Checking

In this section we describe a model checking algorithm that uses B_{weak}-interpolants. Our algorithm is composed of two main stages. Recall that by Def. 5, a B_{weak}-interpolant fulfills two out of the three conditions of an interpolant. Therefore, the first stage attempts to transform the B_{weak}-interpolant into an interpolant.

The second stage uses interpolants computed by the first stage. In essence, the second stage is a modification of the original ITP and is called CNF-ITP. Besides the fact that CNF-ITP uses interpolants in CNF, it further takes advantage of this fact by applying optimizations which are possible only as a result of using interpolants in CNF.

Before going into the details of CNF-ITP, we describe ITP.

4.1 Interpolation-Based Model Checking Revisited

ITP [14] is a complete SAT-based model checking algorithm. It uses interpolation to over-approximate the reachable states in a transition system M with respect to a property p. ITP uses nested loops where the outer loop increases the depth of unrolling and the inner loop computes the reachable states. ITP is described in Fig. 3

Definition 13. *Let k and n be the depth of unrolling used in the outer loop and the iteration of the inner loop of ITP respectively. We define $R_n^k = \text{INIT} \vee I_1^k \vee I_2^k \vee \ldots \vee I_n^k$ to be the set of reachable states computed by the inner loop of ITP after n iterations and with respect to unrolling depth k. For a given $1 \leq j \leq n$, I_j^k is the interpolant computed in the j-th iteration of the inner loop.*

From this point and on, k and n refer to the depth of unrolling used in the outer loop and the iteration of the inner loop of ITP respectively.

16: **function** COMPUTEREACHABLE(M,p, k)
17: $R_0^k = INIT$, $I_0^k = INIT$, $n = 1$
18: **if** $I_0^k \wedge \text{path}^{0,k} \wedge (\neg p(V^1) \vee \ldots \vee \neg p(V^k))$ == SAT **then**
19: **return** cex
20: **end if**
21: **repeat**
22: $A = I_{n-1}^k(V^0) \wedge TR(V^0, V^1)$
23: $B = \text{path}^{1,k} \wedge (\neg p(V^1) \vee \ldots \vee \neg p(V^k))$
24: $I_n^k = \text{GETINTERPOLANT}(A, B)$
25: **if** $I_n^k \Rightarrow R_{n-1}^k$ **then**
26: **return** $fixpoint$
27: **end if**
28: $R_n^k = R_{n-1}^k \vee I_n^k$
29: $n = n + 1$
30: **until** $I_{n-1}^k \wedge \text{path}^{0,k} \wedge (\neg p(V^1) \vee \ldots \vee \neg p(V^k))$ == SAT
31: **end function**

Fig. 4. Inner loop of ITP

In general, the inner loop checks a fixed-bound BMC [2] formula where at each iteration only the initial states are replaced with an interpolant computed at a previous iteration (line: 30). This is done until the BMC formula becomes SAT (line: 30) or until a fixpoint is reached (lines: 25-27). In the former case, the outer loop increases the unrolling depth by 1[2] (line: 13) in order to either increase the precision of the over-approximations or to find a counterexample.

Lemma 1. $R_n^k(V^0) \wedge path^{0,k-1} \wedge (\bigvee_{j=0}^{k-1} \neg p(V^j))$ *is unsatisfiable.*

The above lemma is derived directly from the interpolant definition and from the way R_n^k is computed in ITP. R_n^k is also referred to as $(k-1)$-adequate.

4.2 Transforming a B_{weak}-Interpolant into an Interpolant Using Inductive Reasoning

As we have shown in Sec. 3, given a pair of formulas (A, B) such that $A \wedge B$ is unsatisfiable, a B_{weak}-interpolant I_w can be computed. By Def. 5, $A \Rightarrow I_w$ and $\mathcal{V}(I_w) \subseteq \mathcal{V}(A) \cap \mathcal{V}(B)$, but it is not guaranteed that $I_w \wedge B$ is unsatisfiable. Intuitively, we can think of I_w as being too over-approximated and therefore needing strengthening with respect to B.

Definition 14 (B-adequate). *Let (A, B) be a pair of formulas s.t. $A \wedge B \equiv \bot$ and let I_w be a B_{weak}-interpolant for (A, B). We say that I_w is* B-adequate *iff $I_w \wedge B \equiv \bot$.*

Following the above definition, our purpose is to make a B_{weak}-interpolant I_w B-adequate. We refer to this procedure as *B-Strengthening*.

The purpose of this section is to demonstrate the use of B_{weak}-interpolants for model checking, in particular in the context of ITP.

[2] Some works choose different ways of increasing k. For example, k can be increased by the number of iterations executed in the inner loop: $k = k + n$. In our experiments $k = k + 1$ yielded better results.

Definition 15 (k-n-pair). *Given the formulas $A = I^k_{n-1}(V^0) \wedge \mathrm{TR}(V^0, V^1)$ and $B = path^{1,k} \wedge (\bigvee_{i=1}^{k} \neg p(V^i))$. The pair (A, B) is called a* k-n-pair. *When $A \wedge B \equiv \bot$ we call (A, B) an* inconsistent k-n-pair.

Consider a run of ITP for a given k and n. We aim at computing I^k_n. Let (A, B) be an inconsistent k-n-pair and let I_w be the B_{weak}-interpolant for (A, B). If I_w is B-adequate then it is an interpolant and therefore I^k_n can be defined to be I_w. If I_w is not B-adequate we are required to apply B-Strengthening and transform I_w into an interpolant.

Let us assume that I_w is not B-adequate and that $I_w(V^1) \wedge B$ is satisfiable. There exists a state $s \in I_w$ such that $s(V^1) \wedge B$ is satisfiable. Intuitively, in order to make I_w B-adequate, and by that an interpolant, we would like to remove s from it.

Clearly, $A \wedge s(V^1)$ is unsatisfiable; otherwise $A \wedge B$ would have been satisfiable. Thus, B-Strengthening can be done by iterating all assignments for $I_w(V^1) \wedge B$, extracting a state $s \in I_w$ from an assignment and blocking it in I_w. This is an inefficient way to perform B-Strengthening since the number of such assignments may be too large.

To overcome this, we use knowledge about the problem at hand. Namely, we consider the fact that A is of the following form: $A = I^k_{n-1}(V) \wedge \mathit{TR}(V, V')$.

Definition 16 (Relatively Inductive). *Let R and Q be propositional formulas and M a transition system. We say that Q is* relatively inductive *with respect to R and M if $(R(V) \wedge Q(V)) \wedge TR(V, V') \Rightarrow Q(V')$. When M is clear from the context we omit it.*

Recall that by Def. 13 R^k_n represents an over-approximation of all reachable states after up to n transitions and it is $(k-1)$-adequate (Lemma 1).

Lemma 2. *Let (A, B) be an inconsistent k-n-pair. Let I_w be the B_{weak}-interpolant for (A, B) . If s is an assignment to V s.t. $s(V^1) \wedge B$ is satisfiable, then $R^k_{n-1} \Rightarrow \neg s$ and $R^k_{n-1} \wedge \mathrm{TR} \Rightarrow \neg s'$ hold.*

The above lemma states that if a state s can reach a bad state in up to $k-1$ transitions, it cannot be a state in the set R^k_{n-1}. If we consider a B_{weak}-interpolant derived from the pair (A, B), assuming that $s \in I_w$ (derived from the satisfying assignment to $I_w(V^1) \wedge B$), then s follows the condition in Lemma 2. Therefore, $R^k_{n-1} \Rightarrow \neg s$ and $R^k_{n-1} \wedge \mathit{TR} \Rightarrow \neg s'$ hold and by that $(R^k_{n-1} \wedge \neg s) \wedge \mathit{TR} \Rightarrow \neg s$ holds. By Def. 16 $\neg s$ is relatively inductive with respect to R^k_{n-1}. Therefore, $\neg s$ can be inductively generalized [3].

Inductive generalization results in a sub-clause c of $\neg s$ such that $(R^k_{n-1} \wedge c) \wedge \mathit{TR} \Rightarrow c'$ and $\mathit{INIT} \Rightarrow c$. c can then be used to strengthen I_w and R^k_{n-1}. Adding the clause c to I_w removes s from I_w. This process is then iterated until I_w becomes B-adequate and hence an interpolant. The algorithm for finding the clauses that make I_w B-adequate is described in Fig. 5.

Theorem 1. *Let (A, B) be an inconsistent k-n-pair. Let I_w be a B_{weak}-interpolant and let $c_1, \ldots, c_m$ be clauses s.t.* INIT $\Rightarrow c_i$ *and c_i is relatively inductive w.r.t. R^k_{n-1} for $1 \leq i \leq m$. If $(I_w \wedge \bigwedge_{j=1}^{m} c_j) \wedge B \equiv \bot$ then $I_w \wedge \bigwedge_{j=1}^{m} c_j$ is an interpolant w.r.t $A = (I^k_{n-1}(V^0) \wedge \bigwedge_{j=1}^{m} c_j(V^0)) \wedge \mathrm{TR}(V^0, V^1)$ and $B = path^{1,k} \wedge (\bigvee_{i=1}^{k} \neg p(V^i))$.*

4.3 CNF-ITP: Using B_{weak}-Interpolants in ITP

Above we described how a B_{weak}-interpolant is transformed into an interpolant efficiently for model checking. In this section we present CNF-ITP, a model checking algorithm that is based on ITP. CNF-ITP uses the method described above to compute interpolants. In addition, it uses optimizations that are possible as a result of using interpolants in CNF.

Like the original ITP, our version consists of two nested loops. Since the computation of interpolants is performed in the inner loop, this is where we have made most of our modifications and optimizations. Recall that in the inner loop a BMC formula of a fixed-bound is checked iteratively, where at each iteration only the initial states are replaced by the interpolants computed in a previous iteration. Our modified version of the inner loop appears in Fig. 6

In what follows we consider k to be the unrolling depth used in the inner loop and n to be the iteration during the execution of the inner loop.

The beginning of the loop is similar to the original inner loop of ITP. First, a counterexample of length k is checked (lines: 44-46). If no counterexample exists the pair (A, B) is defined and a B_{weak}-interpolant I_w is computed (line: 50). Then, two optimizations are applied. First, clauses are pushed forward (line: 51). Second, previously computed interpolant is conjoined to the currently computed B_{weak}-interpolant (line: 52). Since I_w may not be B-adequate, the B-Strengthening process may need to add clauses to it (to strengthen it). Adding clauses to I_w before B-Strengthening results in a more efficient B-Strengthening. Moreover, after pushing clauses forward and adding clauses from the previously computed interpolant, I_w may become B-adequate, thereby rendering B-Strengthening redundant.

After applying the two optimizations, B-Strengthening is invoked (line 53). Then the clauses learned during this process are conjoined with R^k_{n-1} and I^k_{n-1} (line 56), and I_w (line 57). After conjoining the clauses, I^k_n is an interpolant. The rest of the loop is identical to the original inner loop of ITP.

We now describe the optimizations in more detail.

Pushing Clauses Forward. Let us consider the interpolant I^k_n computed during the n-th iteration of the inner loop. Since I^k_n is given in CNF, assume that $I^k_{n-1} = \{c_1, \ldots, c_m\}$ where c_i is a clause for every $1 \leq i \leq m$.

```
32: function FindMissingClauses(R,I_w , B, n)
33:     C = ∅
34:     while (I_w ∧ C)(V^1) ∧ B == SAT do      // When C = ∅ it is evaluated as ⊤
35:         Get s ∈ I_w from the SAT assignment
36:         c = InductiveGeneralization(R, s, C)
37:         C = C ∪ c
38:     end while
39:     StoreClauses(n)
40:     return C
41: end function
```

Fig. 5. Find the clauses needed for the B_{weak}-interpolant I_w to be B-adequate

```
42: function ComputeReachableCNF(M,p, k)
43:     R_0^k = INIT, I_0^k = INIT, n = 1
44:     if I_0^k ∧ path^{0,k} ∧ (¬p(V^1) ∨ ... ∨ ¬p(V^k)) == SAT then
45:         return cex
46:     end if
47:     repeat
48:         A = I_{n-1}^k(V^0) ∧ TR(V^0, V^1)
49:         B = path^{1,k} ∧ (¬p(V^1) ∨ ... ∨ ¬p(V^k))
50:         I_w = GetBWeakInterpolant(A, B)
51:         PushInductiveClauses(I_w, n − 1)
52:         I_w = I_w ∧ I_n^{k−1}      // For k = 1, I_n^0 = ⊤
53:         C = FindMissingClauses(R_{n-1}^k, I_w, B)
54:         I_n^k = I_w
55:         for all c ∈ C do
56:             R_{n-1}^k = R_{n-1}^k ∧ c      // Implicitly conjoining c with I_{n-1}^k
57:             I_n^k = I_n^k ∧ c
58:         end for
59:         if I_n^k ⇒ R_{n-1}^k then
60:             return fixpoint
61:         end if
62:         R_n^k = R_{n-1}^k ∨ I_n^k
63:         n = n + 1
64:     until I_{n-1}^k ∧ path^{0,k} ∧ (¬p(V^1) ∨ ... ∨ ¬p(V^k)) == SAT
65: end function
```

Fig. 6. Inner loop of CNF-ITP

Definition 17. *Let M be a transition system and let $F = \{c_1, \ldots, c_m\}$ be a formula in CNF where c_i is a clause over V for every $1 \leq i \leq m$. A clause c_i for some $1 \leq i \leq n$ is said to be* pushable *if $F(V) \wedge \mathrm{TR}(V, V') \Rightarrow c_i(V')$ holds.*

After the computation of a B_{weak}-interpolant, we try to find pushable clauses. Those clauses can be made part of the new interpolant. Adding the pushable clauses to the B_{weak}-interpolant strengthens it.

Incremental Interpolants. The outer loop of CNF-ITP (and ITP) increases the unrolling depth when a more precise over-approximation is needed. Let I_1^1 be the interpolant computed in the first iteration of the inner loop for $k = 1$ and let I_1^2 be the interpolant computed in the first iteration of the inner loop for $k = 2$. Clearly, since both I_1^1 and I_1^2 over-approximate the states reachable in one transition from the initial states, $I_1^1 \wedge I_1^2$ is also an over-approximation of the same set of states. Usually, the size of the interpolants is an issue. Therefore, whenever the inner loop terminates and the bound is increased, all computed interpolants are discarded and are not re-used [13]. Since our method produces interpolants in CNF that are usually small, this conjunction does not create huge CNF formula. This re-use of previously computed interpolants increases the efficiency of CNF-ITP as compared to ITP.

4.4 CNF-ITP: The Best of ITP and IC3

CNF-ITP uses key elements of ITP and IC3. On the one hand, like ITP, CNF-ITP uses the resolution refutation to get information about the reachable states. This information is only partial, and therefore CNF-ITP also uses *inductive generalization*, a key element of IC3, to complete the computation of reachable states. Since the reachable

states are computed by means of over-approximations, there are cases in which the precision of these approximations must be increased. To do so, CNF-ITP uses unrolling, like in ITP. In addition, it uses the fact that interpolants are given in CNF and tries to reuse clauses that have already been learnt (both by pushing the clauses forward and by using previously computed interpolants). CNF-ITP can be viewed as a hybridization of the monolithic approach (ITP) and the incremental approach (IC3). We believe that there are well-founded grounds for comparing the three algorithms, and that further development can bring about an even tighter integration of ITP and IC3. This discussion, however, is outside the scope of this paper.

5 Experimental Results

Our approach includes two major parts. The first part computes a B_{weak}-interpolant from a resolution refutation, and the second part applies B-Strengthening and a model checking algorithm CNF-ITP. The computation of B_{weak}-interpolants was implemented on top of *MiniSAT 2.2*. CNF-ITP and ITP were implemented in a closed-source model checker. For IC3 we used the publicly available ABC framework[3]. In the results we also include the runtime for ABC's ITP implementation in order to show the efficiency of our implementation.

To evaluate our method we used a representative subset of the HWMCC'12 benchmark set. We chose all valid benchmarks that either ITP or CNF-ITP could prove in the given time frame (56 cases). Table 1 presents 30 out of 56 these cases. All experiments were conducted on a system with an Intel E5-2687W running at 3.1GHz with 32GB of memory. Timeout was set to 900 seconds. As mentioned, we sought to test two aspects: the size of the resulting interpolants and the impact on model checking.

Consider Table 1. Our method generates significantly smaller interpolants[4] in almost every case. Summarizing the average size of all computed interpolants shows that CNF-ITP generates interpolants that are *42 times smaller* than those generated by ITP. Note that average interpolant computation time is nearly the same for both methods.

Another interesting aspect of the comparison between CNF-ITP and ITP is the convergence bound. We can see that in many cases the bound is different. This indicates that the strength of the interpolants computed by the two methods is different and affects the results of the model checking algorithm.

Comparing the run-time of the model checking algorithms shows that our CNF-ITP algorithm outperforms ITP and IC3 in terms of the overall run-time. CNF-ITP outperforms ITP in 21 instances, where in 4 of these instances ITP times out. ITP outperforms CNF-ITP only in 7 cases only. CNF-ITP outperforms IC3 in 11 cases, but IC3 is preferable in 16 cases. CNF-ITP is the absolutely best algorithm in 8 cases.

Analysis of the results in the table shows that whenever the number of clauses in the interpolants computed by CNF-ITP is significantly smaller than the number of clauses in the interpolants computed by ITP, the former performs better.

In the cases where the size of interpolants is fairly the same, ITP performs better. This can be explained by the fact that ITP computes small interpolants when the

[3] `https://bitbucket.org/alanmi/abc`

[4] For ITP, the number of clauses is after translation of the interpolants to CNF.

Table 1. Experiment parameters on part of the benchmarks. *Name*: property name; ♯*Vars*: number of state variables in the cone of influence; k is the bound of the outer loop at which fixpoint was found; total_n is the *total* number of iterations executed by the inner loop; $\text{clauses}_{\text{Avg}}$ is the average number of clauses representing each computed interpolant; Extract[s] is the average time to compute an interpolant in seconds; MC[s] is the total runtime of the algorithm in seconds. Values in boldface are the best of all three. Underlined runtime is for cases where CNF-ITP outperforms ITP and *Italic* is for cases where CNF-ITP outperforms IC3.

		IC3_{ABC}	ITP_{ABC}	ITP					CNF-ITP				
Name	♯Vars	MC[s]	MC[s]	k	total_n	$\text{clauses}_{\text{Avg}}$	Extract[s]	MC[s]	k	total_n	$\text{clauses}_{\text{Avg}}$	Extract[s]	MC[s]
beembkry1b1	76	**4.68**	758	15	72	94495	3.14	792	20	83	**1830**	0.65	248
beemcoil1b1	132	**11.77**	TO	9	51	45563	2.09	577	11	52	**487**	0.28	201
beemexit5f1	246	**7.11**	106	25	218	15792	1.21	611	25	255	**792**	0.3	466
beemfish4f1	94	**4.41**	TO	15	50	63423	3.26	TO	14	59	**1348**	0.72	85
beemfwt5f2	3045	543.32	14.68	5	9	854	0.009	2.2	5	10	**22**	0.00	**2.01**
beemfwt1b1	1214	321.86	396	4	18	5721	0.23	**10.58**	4	15	**62**	38.14	TO
beemndhm2f2	251	13.53	138	7	49	29051	1.07	213	5	10	**143**	0.14	**2.92**
beempgmprot1b1	1025	**8.03**	97	33	218	1121	0.82	TO	19	113	**61**	0.00	27.65
beempgmprot7b1	1033	591.49	204	27	168	2717	1.23	TO	18	153	**237**	0.00	**36.21**
beemtlphn5f1	249	29.81	TO	12	60	73460	5.27	TO	20	66	**1197**	0.06	TO
beemtrngt2b1	170	**1.55**	TO	29	193	31942	1.95	TO	15	154	**618**	0.08	23.42
beemtrngt4b1	228	44.71	TO	29	196	22144	1.56	TO	30	281	**371**	0.71	TO
bob05	2404	**7.5**	275	24	121	962	0.37	221	24	136	**198**	0.38	113
bob1u05cu	4377	**7.66**	235	24	124	3116	0.45	251	24	147	**272**	0.19	152
eijkbs3330	246	**7.2**	TO	3	6	764550	22.74	TO	3	9	**5873**	10.44	154
6s38	1931	TO	TO	10	33	84988	1.19	TO	7	22	**4299**	15.7	**423**
6s108	782	**4.83**	TO	8	43	89493	3.67	TO	7	25	**1787**	0.56	42.65
6s120	58	**0.71**	4.1	3	6	**365**	0.34	6.5	3	8	373	0.12	2.92
6s121	419	821.54	TO	24	214	4542	0.08	46.25	18	98	**389**	0.04	**14.42**
6s132	139	**2.87**	7.5	7	13	35973	2.88	85	5	13	**4221**	3.5	76
6s136	3342	TO	3.1	20	58	2471	0.005	4.4	20	53	**16**	0.00	**1.94**
6s151	150	TO	TO	14	515	**1998**	0.22	**461**	10	122	6226	3.15	TO
6s159	252	0.03	7.8	15	143	656	0.01	4.9	10	60	**27**	0.00	0.34
6s164	198	8.96	TO	18	77	753	0.02	3.7	18	85	**135**	0.006	**2.43**
6s181	607	TO	26.2	8	19	63509	4.58	**232**	6	10	**2556**	6.45	TO
intel021	365	TO	TO	18	316	2503	0.05	**51.3**	18	489	**331**	0.14	*117*
intel022	530	TO	TO	21	435	18818	0.5	**629**	20	555	**585**	1.05	TO
intel024	357	TO	TO	15	233	3087	0.04	**34.5**	15	341	**206**	0.12	*83*
intel031	531	TO	TO	21	268	3465	0.15	114	18	235	**183**	0.03	**61**
intel034	3297	TO	TO	16	425	1477	0.02	**85**	16	432	**83**	0.19	*119*
Total		10544	16672			1469009	59	12535			34928	83	**7854**

resolution refutation is small. Therefore, computing the interpolants in ITP is more efficient in these cases since it only requires linear traversal over the resolution refutation. In contrast, our method requires B-Strengthening, a process that is in some cases expensive. We conclude that when the resulting interpolants in ITP are large, CNF-ITP has a significant advantage in the vast majority of cases.

6 Conclusion

We have presented a novel approach for deriving interpolants for SAT-based model checking. Our procedure generates small interpolants in CNF using a twofold scheme: First, an interpolant approximation is computed with an algorithm that exploits resolution refutation properties. Following this, inductive reasoning is used to complete transforming the approximation into an interpolant. Our experiments show that our approach generates interpolants that are much smaller (by 42 times overall) than those generated by the classical ITP approach.

In addition, we have implemented CNF-ITP, a model checking algorithm that uses the above method to compute interpolants. CNF-ITP significantly outperformed ITP and outperformed IC3 in a large number of cases. We believe that this approach may be further developed and enhanced, yielding an even more efficient model checking algorithm.

Acknowledgments. The authors would like to thank Håkan Hjort and Paul Inbar for their valuable comments.

References

1. Bar-Ilan, O., Fuhrmann, O., Hoory, S., Shacham, O., Strichman, O.: Linear-time reductions of resolution proofs. In: Chockler, H., Hu, A.J. (eds.) HVC 2008. LNCS, vol. 5394, pp. 114–128. Springer, Heidelberg (2009)
2. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)
3. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
4. Cabodi, G., Murciano, M., Nocco, S., Quer, S.: Stepping forward with interpolants in unbounded model checking. In: ICCAD, pp. 772–778 (2006)
5. Chockler, H., Ivrii, A., Matsliah, A.: Computing interpolants without proofs. In: HVC (2012)
6. Craig, W.: Linear reasoning. a new form of the herbrand-gentzen theorem. J. Symb. Log. 22(3) (1957)
7. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM 7(3), 201–215 (1960)
8. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 129–145. Springer, Heidelberg (2010)
9. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
10. Goldberg, E., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: DATE, pp. 886–891 (2003)
11. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL (2004)
12. Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 39–51. Springer, Heidelberg (2005)
13. Marques-Silva, J.: Interpolant learning and reuse in SAT-based model checking. Electr. Notes Theor. Comput. Sci. 174(3), 31–43 (2007)
14. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
15. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
16. McMillan, K.L., Amla, N.: Automatic Abstraction without Counterexamples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)
17. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symb. Log. 62(3) (1997)

18. Rollini, S.F., Bruttomesso, R., Sharygina, N.: An efficient and flexible approach to resolution proof reduction. In: Raz, O. (ed.) HVC 2010. LNCS, vol. 6504, pp. 182–196. Springer, Heidelberg (2010)
19. Ryvchin, V., Strichman, O.: Faster extraction of high-level minimal unsatisfiable cores. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 174–187. Springer, Heidelberg (2011)
20. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: FMCAD (2009)
21. Vizel, Y., Grumberg, O., Shoham, S.: Intertwined forward-backward reachability analysis using interpolants. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 308–323. Springer, Heidelberg (2013)
22. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In: SAT (2003)