Computing Interpolants without Proofs*

Hana Chockler, Alexander Ivrii, and Arie Matsliah

IBM Research – Haifa

Abstract. We describe an incremental algorithm for computing interpolants for a pair φ_A, φ_B of formulas in propositional logic. In contrast with the common approaches, our method does not require a proof of unsatisfiability of $\varphi_A \wedge \varphi_B$, and can be realized using any SAT solver as a black box. We achieve this by combining model enumeration with the ability to easily generate interpolants in the special case that one of the formulas is a cube.

1 Introduction

Craig's interpolation theorem [Cra57] states that for any pair of propositional formulas φ_A, φ_B, if φ_A implies $\neg\varphi_B$ ($\varphi_A \Rightarrow \neg\varphi_B$) then there exists a formula φ_I, so that $\varphi_A \Rightarrow \varphi_I \Rightarrow \neg\varphi_B$, and in addition $\mathrm{Vars}(\varphi_I) \subseteq \mathrm{Vars}(\varphi_A) \cap \mathrm{Vars}(\varphi_B)$. The formula φ_I is called a *Craig interpolant of* φ_A *and* φ_B.

Starting with the seminal work of McMillan [McM03], interpolants have a central role in formal verification (and beyond) – various application include hardware model checking [McM03, McM05], detection of functional dependency [LJHM07], Boolean function decomposition [LJH08], and model checking of sequential programs [McM10].

The most common technique for computing an interpolant for a pair of formulas φ_A, φ_B in propositional logic is based on a resolution refutation for $(\varphi_A \wedge \varphi_B)$ produced by a DPLL-like SAT solver [ZM03, ANORC10]. Once obtained, the proof can be transformed into an interpolant in the form of a Boolean circuit having the same structure as the proof itself [Kra97, Pud97, McM03, KW10].[1]

Even though this scheme is generally very successful in practice, its main limitation is the need for a refutation (proof of unsatisfiability) that is of manageable size. Since modern SAT solvers are not specifically aimed to produce short refutations, even for simple problems the interpolants produced are often too big to handle. In addition, the interpolants constructed in this way are usually highly redundant, and for practical applications it is often beneficial to minimize/simplify them. However, such minimization can be very costly – and thus in practice one might not succeed to construct a small interpolant even if

* This work is partially supported by the European Community under the call FP7-ICT-2009-5 – project PINCETTE 257647.

[1] There are efficient algorithms known to compute interpolants based on refutations in proof systems other than resolution (e.g., in Cutting Planes [Kra97]), but the one based on resolution is the canonical one from practical perspective.

A. Biere, A. Nahir, and T. Vos (Eds.): HVC 2012, LNCS 7857, pp. 72–85, 2013.

one exists. In the worst case, the input formula $(\varphi_A \wedge \varphi_B)$ might not have a resolution refutation of polynomial (in the number of variables) size at all.

The problem boils down to the dependency of the method on a particular type of SAT solving algorithm. In other words, even if the ultimate SAT solver was given to us as an oracle that answers any satisfiability query instantly, we would not know how to use it to produce interpolants efficiently.

1.1 Our Contribution

In this paper we present a SAT-based incremental algorithm for computing interpolants. Our algorithm can be easily implemented on top of any complete SAT-solver with the minimal interface to return *SAT* / *UNSAT* and a satisfying assignment in case of *SAT*, and in particular the solver does not need to produce proofs or be DPLL-based. Even though this minimal interface in principle suffices, for practical reasons we will want from the solver a bit more: the (now standard) incremental interface that allows adding clauses between calls and solving under assumptions (see Section 3).

The core idea of our algorithm is to compute an interpolant φ_I incrementally by taking the disjunction of "point" interpolants φ_p for all $p \in A$ (where A denotes the set of models of φ_A). Each point interpolant φ_p contains p, is disjoint from B, and is defined in terms of common variables of φ_A and φ_B. It follows that the union of these interpolants for all $p \in A$ contains all of A and is still disjoint from B – thus constituting a valid interpolant for (φ_A, φ_B). An important observation here is that computing a point interpolant, or more generally an interpolant between a cube $c = \ell_1 \wedge \cdots \wedge \ell_k$ (containing up to 2^{-k} fraction of the points in A) and any propositional formula φ_B is trivial: one can simply remove from c the literals referring to variables local to A. From the practical viewpoint, it is crucial to further generalize these cubes to cover as many points of A as possible while still being disjoint from B. The difference between our approach and the traditional (monolithic) interpolation is depicted graphically in Figure 1.

A more general method of computing an interpolant for (φ_A, φ_B) is the following two-stage process. First, we show that $\varphi_A \wedge \varphi_B$ is unsatisfiable by means of a certain *partition-based* algorithm (by this we mean an algorithm in which one solver is run on φ_A, another solver is run on φ_B, and the two solvers are allowed to exchange constraints consisting only of common variables). Second, we show how to construct an interpolant based on these exchanged constraints (see Section 3.3 for details).

Our approach is somewhat similar in spirit to the classic algorithm that extracts a satisfying assignment from a SAT decision procedure, viewing the procedure as an oracle. In this algorithm, the variables are ordered, and then the first one is assigned at random. The algorithm then queries the SAT oracle for the existence of a satisfying assignment for the rest of the formula; it continues in this way until all variables are assigned.

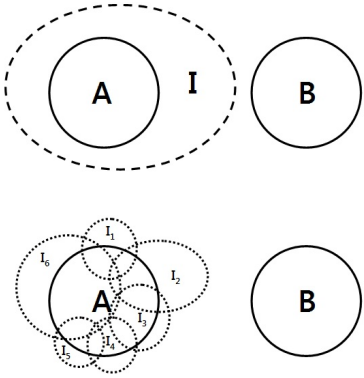

Fig. 1. Comparison between monolithic and incremental interpolants

1.2 Related Work

This work is tightly related to various methods for finding all models (satisfying assignments) of a given formula, or more specifically finding all assignments to the common variables of φ_A and φ_B possessing extensions satisfying φ_A. We refer to the papers [McM02, JS05, BKK11, GM12] containing efficient algorithms for this task and references to earlier work. In particular we also follow the widely used blocking clause approach to prevent the algorithm from discovering the same point again and again, and we try to generalize cubes as much as possible to get quick coverage of A (the set of models of φ_A). However, our setting allows an additional twist on the generalization process which makes convergence of interpolant computation quicker than that of computing all satisfying assignments: we can additionally generalize each cube as long as it remains disjoint from B, allowing wider coverage of points in A.

An alternative partition-based algorithm for detecting whether $\varphi_A \wedge \varphi_B$ is unsatisfiable appears in [PG00]. In the cited work, all the assignments to the common variables are checked, and $\varphi_A \wedge \varphi_B$ is unsatisfiable if and only if for every such assignment c either $c \wedge \varphi_A$ or $c \wedge \varphi_B$ is unsatisfiable. In contrast, our algorithm only considers those assignments to the common variables which admit a satisfiable extension for φ_A (and in practice this number is much smaller due to generalization).

Last but not least, an important source of inspiration for this work are the papers [BKK11, Bra11] which demonstrate the general power of an incremental approach for solving difficult problems. Note that in the context of model checking, IC3 (the recent breakthrough model-checking technique by Bradley [Bra11]) can be also viewed as a method which generates interpolants without proofs. However our setting is more general, allowing to compute an interpolant for any pair of propositional formulas.

2 Preliminaries

As usual, a *literal* is either a variable or its negation, a *clause* is a disjunction of literals, and a *cube* is a conjunction of literals. A *CNF* is a conjunction of clauses and a *DNF* is a disjunction of cubes.

Given a (propositional) formula φ_A, we denote by V_A the set of all variables that occur in φ_A. Given two formulas φ_A and φ_B, we denote by $V_{A\cap B} \triangleq V_A \cap V_B$ the set of *common* variables of φ_A and φ_B, and by $V_{A\setminus B} = V_A \setminus V_B$ the set of all variables *local* to φ_A.

With each formula φ_A we associate a subset of $\{0,1\}^{V_A}$ containing all assignments to V_A satisfying φ_A (models of φ_A). Slightly abusing notation, we sometimes refer to formulas as subsets and vice versa; in particular, $\varphi_A = \emptyset$ means that φ_A is unsatisfiable.

Definition 1 (Interpolant). [2] *Let φ_A, φ_B be a pair of formulas that cannot be simultaneously satisfied ($\varphi_A \wedge \varphi_B = \emptyset$). An* interpolant *$\varphi_I = Itp(\varphi_A, \varphi_B)$ of (φ_A, φ_B) is a formula satisfying: 1) $\varphi_A \Rightarrow \varphi_I$; 2) $\varphi_I \wedge \varphi_B = \emptyset$; and 3) $V_I \subseteq V_{A\cap B}$.*

Our algorithm makes use of a SAT solver to decide satisfiability of a propositional formula. For most of the paper this solver is viewed as a black box; the only requirements are that it should output *SAT* or *UNSAT* depending on the status of the formula, and in the case of *SAT* the solver should also return an assignment (model) satisfying the formula. Note that a model is a conjunction of unit clauses, or simply a cube.

We denote by $P^A_{A\cap B}$ the projection of the set (of all assignments satisfying) φ_A to $V_{A\cap B}$. $P^B_{A\cap B}$ is defined similarly. Clearly, $\varphi_A \wedge \varphi_B = \emptyset$ if and only if $P^A_{A\cap B} \cap P^B_{A\cap B} = \emptyset$.

3 Algorithm

In this section we present the main contribution of this paper – an incremental algorithm for computing an interpolant φ_I of (φ_A, φ_B). In the following subsections we describe a basic version of the algorithm, an efficient implementation of this algorithm on top of a modern CDCL solver (e.g. MiniSat), and various extensions and optimizations which seem crucial for real-life instances.

3.1 Basic Algorithm

The algorithm (described in Algorithm 1) accepts a pair (φ_A, φ_B) of propositional formulas, and, in the case under consideration that $\varphi_A \wedge \varphi_B$ is unsatisfiable, returns an interpolant φ_I (in DNF). The algorithm also detects the case that $\varphi_A \wedge \varphi_B$ is satisfiable, and can in principle return a satisfying assignment to $\varphi_A \wedge \varphi_B$.

[2] This definition slightly deviates from the original definition of Craig, but is now standard in the context of formal methods.

The interpolant φ_I is constructed incrementally; initially it is empty. Roughly speaking, the algorithm searches for points $p \in \varphi_A$ not yet covered by φ_I and then generalizes these points to cubes by omitting assignments to all variables local to φ_A and to as many common variables as possible, while still keeping these cubes disjoint from φ_B. Each such generalized cube represents a new incremental knowledge and is added to φ_I. For convenience we introduce the set $\varphi'_A = \varphi_A \wedge \neg\varphi_I$ corresponding to the set of assignments in φ_A not yet covered by φ_I; the algorithm terminates when φ'_A becomes empty.

Algorithm 1. Iterative computation of the interpolant

Input: A pair (φ_A, φ_B) of propositional formulas
Output: An interpolant φ_I for (φ_A, φ_B) (in DNF) if $\varphi_A \wedge \varphi_B = \emptyset$, or a satisfying assignment otherwise

```
 1: φ_I ← ∅
 2: φ'_A ← φ_A
 3: while TRUE do
 4:     if φ'_A is unsatisfiable then
 5:         return φ_I
 6:     else
 7:         Let p be a model of φ'_A
 8:         p' ← projection of p to V_{A∩B}
 9:         p'' ← generalization of p' w.r.t. φ_A
10:         if (p'' ∧ φ_B) is satisfiable then
11:             return SAT + model
12:         else
13:             p''' ← generalization of p'' w.r.t. φ_B
14:             φ_I ← φ_I ∨ p'''
15:             φ'_A ← φ'_A ∧ ¬p'''
16:         end if
17:     end if
18: end while
```

We now describe a single iteration of the main loop (lines 4-17) in detail. On line 4 we call a SAT solver to check whether φ'_A is empty. If so, then $\varphi_A \subseteq \varphi_I$ and the algorithm terminates providing φ_I as the final interpolant. Otherwise (line 7), the SAT solver returns a model p for φ'_A, that is an assignment to variables in V_A. First we project this assignment to $V_{A\cap B}$ by omitting the variables local to φ_A (line 8); this projection corresponds to the cube p'.

We can (on line 9) further generalize the cube p' to p'' as long as it satisfies the following property: for any extension $\tilde{p}'$ of p'' to $V_{A\cap B}$ there is a further extension $\tilde{p}$ of $\tilde{p}'$ to V_A which satisfies φ_A. In other words, consider the projection $P^A_{A\cap B}$ of all assignments satisfying φ_A to $V_{A\cap B}$. By construction, $p' \in P^A_{A\cap B}$, and we seek to generalize it to p'' so that as sets of assignments $p' \in p'' \subseteq P^A_{A\cap B}$, thus enumerating more than one (projection of a) satisfying assignment to φ_A to $V_{A\cap B}$ at once. We describe the existing methods for such a generalization in

Section 3.3. Also note that this generalization is performed with respect to the original set φ_A.

Next (on line 10), we make another call to a SAT solver to check whether $\varphi_B \wedge p''$ is satisfiable (note that the cube p'' can be passed to the solver as a set of unit assumptions). If this is the case, then $\varphi_A \wedge \varphi_B$ is satisfiable, and in fact we can obtain an explicit satisfying assignment to $\varphi_A \wedge \varphi_B$ by extending the assignment satisfying $p'' \wedge \varphi_B$ to $V_{A \setminus B}$ which satisfies φ_A (which is possible by the property above). If the generalization step on line 9 is omitted, a satisfying assignment to $\varphi_A \wedge \varphi_B$ can be obtained immediately by unifying the assignment to $p'' \wedge \varphi_B$ and the assignment p (since the two assignments match on the common variables).

In the main case under consideration, $p'' \wedge \varphi_B$ is unsatisfiable, and we seek (on line 13) to generalize p'' even further to p''' by dropping the assignments to some of the variables in $V_{A \cap B}$ while keeping p''' disjoint from φ_B. The difference between this generalization and the one on line 9 is that now we can let p''' represent non-$P^A_{A \cap B}$ points provided that they are also non-$P^B_{A \cap B}$ points (see definitions above). In particular, p''' can also describe additional $P^A_{A \cap B}$-points, not previously described by p''. From the practical viewpoint, this is a very important optimization (details follow).

Note that we can view p''' as an interpolant of p and φ_B. We update $\varphi_I \leftarrow \varphi_I \vee p'''$ (thus keeping φ_I in DNF) and prevent the solver from rediscovering points in $p''' \wedge \varphi_A$ (and in particular p) by adding the blocking clause $\neg p'''$ to φ'_A.

Claim 1. *(1) Algorithm 1 always terminates. (2) If $\varphi_A \wedge \varphi_B = \emptyset$ it outputs a valid interpolant φ_I for (φ_A, φ_B).*

Proof. (1) Initially $\varphi'_A = \varphi_A$, and in each iteration its size (as set of assignments) shrinks by ≥ 1. (2) By construction, $V_I \subseteq V_{A \cap B}$. In addition, φ_I is a disjunction of cubes that are disjoint from φ_B, hence $\varphi_I \wedge B = \emptyset$. To see that $\varphi_A \Rightarrow \varphi_I$, observe that the algorithm terminates only when $\varphi'_A \triangleq \varphi_I \setminus \varphi_A$ becomes empty.

3.2 Implementation Details

Now we describe how the algorithm proposed in the last section can be efficiently implemented on top of MiniSat or any other SAT solver that provides an interface to incrementally add new clauses into the solver, and to solve under a set of additional unit assumptions [ES03]. In the case of a satisfiable result, such a solver should return a model satisfying all of the clauses in the solver as well as all of the unit assumption literals. In the case of an unsatisfiable result, the solver should return a subset of the assumptions used in the proof of unsatisfiability.

We keep two instances of the SAT solver: *A-solver* holding the CNF for φ'_A and *B-solver* holding the CNF for φ_B. Thus the SAT call on line 4 corresponds to $A.Solve()$ and the SAT call on line 10 corresponds to $B.Solve(p'')$ with the cube p'' passed as the set of unit assumptions. In the simplest version of the algorithm we can skip the generalization on line 9, and the generalization on

line 13 of p'' to p''' is obtained for free by taking p''' to be the subset of the assumptions in p'' used for unsatisfiability. Finally, the strengthening of φ'_A on line 15 corresponds to $A.add(\neg p''')$.

We found that this (somewhat primitive) implementation already performs quite well on many instances.

3.3 Extensions

For some real-life instances originating from hardware model checking problems, the basic algorithm described above often takes too many iterations to converge in reasonable time. By experimentation, we found that the following heuristics/optimizations work well for those hard cases.

Exhaustive Generalization of p'' to p'''. Even though the MiniSat-like "solve under assumption" mechanism is highly successful at detecting which subset of the assumptions is important for unsatisfiability, this subset is very often far from minimal. Thus, after obtaining a reduced set from the solver's "final" conflict analysis, one can try to shrink this set further. It is natural to look for *minimal*[3] or even *minimum-sized* subsets.

We implemented the following greedy approach for finding a minimal subset of the conflicting assumptions, similar to a basic destructive algorithm for MUS computations [DGHP09, Nad10, SL11]. Remove one of the assumptions – if the remaining formula is satisfiable then this assumption is deemed as *necessary* and must be present in all minimal assumption subsets from this point on. If the remaining formula is unsatisfiable, then the assumption is *redundant* and is deleted from the set of assumptions under consideration. Also note that in the case of an unsatisfiable answer, one can immediately trim the set of non-processed assumptions further (when this functionality is supported by the solver). After all of the assumptions are processed, we end up with a minimal subset as required. We refer to this approach as *exhaustive B-generalization*.

In general this optimization has a significant overhead on the running time of a single iteration of the loop since in the worst case it resorts to one additional SAT call for each of the assumptions in the initial set. However as we will see in the experimental section, the smaller clauses produced by this minimization are of better quality and the algorithm takes significantly fewer iterations to converge (in the same spirit as generalization of counterexamples and inductive clauses in IC3).

Forall-Exists Generalization of p' to p''. Turning to generalization of p' with respect to φ_A (on line 9), several methods have been proposed in earlier works in the context of finding all satisfying assignments to a formula (see for example [JS05]) or existential quantification (e.g. [BKK11] and [GM12]).

We implemented several variations, based on [BKK11]. We apply the following "dual-rail" construction. For each of the common variables $v \in V_{A\cap B}$, we introduce two additional fresh variables v^+ and v^-, and we replace each occurrence

[3] I.e. the formula would become satisfiable if any of the assumptions were dropped.

of v in φ_A by v^+, and each occurrence of $\neg v$ in φ_A by v^-. In addition, we add the binary clause $(\neg v^+, \neg v^-)$ to prevent the solver from assigning both v^+ and v^- to *true*. In the simpler variant to which we refer as *trivial A-generalization* we create the cube p'' from the model p of φ'_A by including the literals v for which v^+ is set to *true*, and the literals $\neg v$ for which v^- is set to *true*. In the more complicated variants, we seek the shortest possible cubes p''. To this end, we create additional variables $v^{\pm} = v^+ \wedge v^-$ for $v \in V_{A \cap B}$, and put a sequential counter construction [Sin05] on top of $v^{\pm}$. This allows (passing additional assumptions to the A-solver) to look for cubes in A which are of size at most k, for any given k, and one can find a shortest cube by setting increasing values to $k = 1, 2, \ldots$. By experimenting with various parameter settings, we limit the maximum value of the counter to $\min(15, |V_{A \cap B}|)$ and we do a binary search to find the minimal $k \in [1..15]$ if it exists. In other words we are guaranteed to end up with a shortest cube whenever it has length at most 15, and otherwise we resort to the trivial A-generalization from above. We refer to this version as *counter-based A-generalization*.

In general, the dual-rail construction has a negligible overhead, but searching for a shortest cube is expensive, both due to the extra logic pertaining to the sequential counter construction and the increased number of SAT-calls.

Exchanging Roles of the Two Solvers. In certain cases φ_B has fewer satisfying assignments than φ_A or it is easier to enumerate them. Then it might be easier to solve the "dual" problem first: compute φ_J – interpolant for (φ_B, φ_A), and then set $\varphi_I \leftarrow \neg\varphi_J$. This way the final interpolant is in CNF, but in most applications the precise form of the interpolant is not important (in case it is, see [BKK11] for an efficient method to convert between the two forms).

Exchanging Clauses of Common Variables. One can consider a general algorithm which uses the partitioning of $\varphi_A \wedge \varphi_B$ into (φ_A, φ_B) and allows to exchange learned clauses between the two solvers as long as they consist of common variables only. In particular, such an algorithm might use a scenario when the roles of the two solvers are switched periodically, or a scenario with the two solvers running in parallel, each producing satisfying assignments and/or blocking the satisfying assignments found by the other solver.

Of course, if the clauses are passed from φ_A to φ_B and back freely, more care should be taken when assembling the final interpolant. Luckily this is not too hard due to the following (here φ_G and φ_H correspond to (sets of) clauses learnt from φ_A and φ_B respectively).

Claim 2. *Suppose that* $\varphi_A \Rightarrow \varphi_G$ *and* $\mathrm{Vars}(\varphi_G) \subseteq V_{A \cap B}$. *Then an interpolant* $\varphi_I = Itp(\varphi_A, \varphi_B)$ *can be computed as* $\varphi_I = Itp(\varphi_A, \varphi_B \wedge \varphi_G) \wedge \varphi_G$.

Proof. Let $\varphi_J = Itp(\varphi_A, \varphi_B \wedge \varphi_G)$. By definition, $\varphi_A \Rightarrow \varphi_J$ and $\varphi_A \Rightarrow \varphi_G$, hence $\varphi_A \Rightarrow \varphi_I$. Also, φ_J is disjoint from $\varphi_B \wedge \varphi_G$, hence $\varphi_J \wedge \varphi_G$ is disjoint from φ_B.

Claim 3. *Suppose that* $\varphi_B \Rightarrow \varphi_H$ *and* $\mathrm{Vars}(\varphi_H) \subseteq V_{A\cap B}$. *Then an interpolant* $\varphi_I = Itp(\varphi_A, \varphi_B)$ *can be computed as* $\varphi_I = Itp(\varphi_A \wedge \varphi_H, \varphi_B) \vee \neg\varphi_H$.

Proof. Let $\varphi_J = Itp(\varphi_A \wedge \varphi_H, \varphi_B)$. $\varphi_A \wedge \varphi_H \Rightarrow \varphi_J$, hence $\varphi_A \Rightarrow \varphi_J \vee \neg\varphi_H = \varphi_I$. Also, since both φ_J and $\neg\varphi_H$ are disjoint from φ_B, so is φ_I.

We can keep track of the sets of clauses passed from φ_A to φ_B and vice versa and to reconstruct the interpolant by following the two rules above. This procedure leads to more general definitions of interpolants (not only CNF or DNF). However, if the only clauses passed from φ_A to φ_B are unit clauses, then the interpolant can be still computed in DNF as $(\vee C_i) \wedge (x) = \vee(C_i \wedge x)$ by distributivity.

Remark 1. Note that the blocking clauses $\neg p'''$ of Algorithm 1 are implied by φ_B, and thus can be viewed as clauses passed from φ_B to φ_A. In other words, Algorithm 1 can be seen as employing the incremental interpolant computation dictated by Claim 3 only.

We implemented a variant of this technique which periodically instructs the A-solver to look for unit clauses of common variables (by running the A-solver with a small time-limit and a small backtrack-limit) and passing these clauses to the B-solver. In many cases we saw a big reduction in the total number of iterations (in several cases passing unit clauses from φ_A to φ_B made φ_B directly unsatisfiable). The cons of this technique is that not all of the units passed from φ_A to φ_B are really required for unsatisfiability, while our construction adds all of these units into the interpolant. We refer to this optimization as *flp* (since it is reminiscent of failed literal probing in SAT-solving).

Interpolant Strength. We make a theoretical digression. In the discussion above we strove to generalize p' as much as possible, that is to describe the largest set in the projection. The motivation for this is clear – a smaller blocking clause can potentially block more points of φ'_A, thus allowing the algorithm to converge faster. However for various applications the loosest possible interpolant might not be good, and it could be helpful to compute the *largest* blocking clause which blocks the same points in φ'_A as $\neg p'''$. In other words, we can seek for a subcube q with $p' \subseteq q \subseteq p'''$ and $\varphi'_A \wedge \neg q = \varphi'_A \wedge \neg p'''$. After such subcube q is found, we can modify the line 14 of the algorithm to include q instead.

We illustrate this on an example. Suppose $\{x_1, x_2, x_3, x_4, x_5\} \subset V_{A\cap B}$ is a subset of common variables, $p'' = x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5$, and $p''' = x_1 \wedge x_2$. Suppose further that $p''' \wedge P^{A'}_{A\cap B}$ consists of the three points $(1,1,1,1,1,\cdot)$, $(1,1,1,0,1,\cdot)$, $(1,1,0,1,1,\cdot)$, where $\cdot$ represents the remaining variables in $V_{A\cap B}$. Note that the variable x_5 takes the same value on $P^{A'}_{A\cap B}$ and thus we can use $q = x_1 \wedge x_2 \wedge x_5$ instead of $x_1 \wedge x_2$ in the interpolant.

Finding the maximal set of variables which are constant on $p''' \wedge \varphi'_A$ can be done in at most $|p''| - |p'''|$ SAT calls. We illustrate this procedure on our example. We ask the SAT solver whether $\varphi'_A \wedge x_1 \wedge x_2 \wedge \neg(x_3 \wedge x_4 \wedge x_5)$ is satisfiable, that is we are looking for a point in $p''' \wedge P^{A'}_{A\cap B}$ with at least one different value out of

$\{x_3 = 1, x_4 = 1, x_5 = 1\}$. Let's say that the solver returns the point $(1, 1, 1, 0, 1, \cdot)$ which means that the variable x_4 is not constant on $p''' \wedge P^{A'}_{A\cap B}$. We refine the query asking whether $\varphi_A \wedge x_1 \wedge x_2 \wedge \neg(x_3 \wedge x_5)$ is satisfiable. Now the solver returns the point $(1, 1, 0, 1, 1, \cdot)$ which means that the variable x_3 is also not constant on $p''' \wedge P^{A'}_{A\cap B}$. Finally the query $\varphi_A \wedge x_1 \wedge x_2 \wedge \neg x_5$ is unsatisfiable, and the remaining set of variables are constant on $p''' \wedge P^{A'}_{A\cap B}$.

4 Experiments

Before discussing concrete experimental results, let us think when we expect the suggested approach to succeed. As mentioned before, Algorithm 1 and its variations might perform well if enumerating all satisfying assignments of φ_A (or φ_B) is not too hard, or whenever there exists a successful partition-based algorithm for solving $\varphi_A \wedge \varphi_B$. In particular we expect our algorithm to be successful for simple formulas with a small number $|V_{A\cap B}|$ of common variables[4].

We have evaluated our algorithm on the 465 single-property benchmarks used in the 2011 Hardware Model Checking Competition. For each of these benchmarks we unrolled the design for 11 cycles to represent the bounded model checking formula $J(x_0) \wedge T(x_0, x_1) \wedge \bigwedge_{i=1}^{10} T(x_i, x_{i+1}) \wedge \bigvee_{i=1}^{11} \neg P(x_i)$, where J, T, and P denote respectively the initial states of the design, the transition relation and the property being verified (see [McM03] for details). We define[5] $\varphi_A = \bigwedge_{i=1}^{10} T(x_i, x_{i+1}) \wedge \bigvee_{i=1}^{11} \neg P(x_i)$, $\varphi_B = J(x_0) \wedge T(x_0, x_1)$, and cnf-ize these propositional formulas using a variant of the approach described in [CMV09]. As the underlying SAT solver we use *Mage*, an IBM SAT solver which supports both the incremental interface of MiniSat and the ability to compute interpolants from proofs. In all of the experiments, the time-limit was set to 1800 seconds.

In the following tables we compare the performance of standard interpolation (building an interpolant from the proof of unsatisfiability of $\varphi_A \wedge \varphi_B$, in this and only in this case the proof generation capabilities of the solver are turned on) and various schemes based on Algorithm 1. We distinguish between three versions of generalization with respect to φ_A: no generalization at all (no A-gen), trivial A-generalization (triv A-gen) and counter-based A-generalization (cntr A-gen). The latter two versions are described in Section 3.3 and require dual-rail encoding. We also consider three versions of generalization with respect to φ_B: no generalization at all (no B-gen), the generalization based on the conflicting assumptions returned by the solver (std B-gen) and exhaustive B-generalization from Section 3.3 (exh B-gen). The results are summarized in Table 1. The second, third and fourth columns respectively denote the numbers of satisfied, unsatisfied and time-out instances, and the last column denotes the total time of the run. First of all, we see that generalizing with respect to B is crucial and that counter-based A-generalization is mostly unhelpful (in fact,

[4] In the worst case, all interpolants (expressed in terms of $V_{A\cap B}$) might be of size exponential in $|V_{A\cap B}|$: consider formulas φ_A, φ_B which enforce the parity of assignments to $V_{A\cap B}$ to be even and odd, respectively.

[5] The benefits of this "opposite" splitting are discussed later.

even the version of A-generalization with the unrestricted counter size on average only removes at most 5% - 10% of the literals of p', while the exhaustive B-generalization usually removes 90% of the literals and more). We also note that the standard interpolation performs best in terms of time (however, there was in fact one testcase where the version (no A-gen, std B-gen) finished in 1184 seconds, while the standard interpolation timed out). It should be noted that in this and subsequent experiments all the observed phenomena are consistent across individual instances (and not only in the bulk).

Table 1. Comparison of runtimes on 465 single property benchmarks from HWMCC11

Variant	**SAT**	**UNSAT**	**TO**	**Total Running Time** (s)
standard interpolation	14	429	22	49,153
no A-gen, no B-gen	1	22	442	767,678
no A-gen, std B-gen	14	420	31	79,686
no A-gen, exh B-gen	14	421	30	79,754
triv A-gen, no B-gen	1	25	439	762,135
triv A-gen, std B-gen	13	419	33	83,759
triv A-gen, exh B-gen	14	420	31	79,599
cntr A-gen, no B-gen	1	17	447	783,311
cntr A-gen, std B-gen	10	393	62	124,170
cntr A-gen, exh B-gen	10	399	56	115,312

We omit the inferior configurations and restrict to the test cases on which each configuration returned with a SAT or UNSAT answer - there are 430 such configurations. The comparison of the total number of iterations (column 2), the total interpolant size (column 3) and the total running time (column 4) are provided in Table 2. For the standard interpolation the number of iterations is meaningless and the size represents the number of gates in the *non-optimized* circuits (i.e. no structural hashing, etc. has been performed). For the remaining configurations the size represents the total number of literals in the computed interpolants (in CNF). Even though it is clear that the 4 schemes based on Algorithm 1 are on average 7 times slower than standard interpolation, it is interesting to note that they produce interpolants of much smaller size (up to 900 times). In particular, they require no need for further minimization. Next, it seems that on our test cases the extra time spent by a round of exhaustive generalization is compensated by fewer iterations required for the algorithm to converge. Finally, the dual-rail encoding and the trivial version A-generalizations seem to have a small positive impact on the size of the interpolant.

We have performed an additional experiment to see the value of periodically passing unit clauses from φ_A to φ_B (the flp technique described in Section 3.3). To this end we compare the (no A-gen, exh B-gen) configuration with a version of itself, where at the start and every 100 iterations the A-solver is instructed to look for unit clauses of common variables and to pass them to the B-solver. The results are summarized in Table 3. As usual, we restrict only to the benchmarks

Table 2. Comparison of numbers of iterations and interpolant sizes on 430 benchmarks

Variant	**Total #iters**	**Total itp size**	**Total Running Time** (s)
standard interpolation	0	90,922,242	3,842
no A-gen, std B-gen	129,506	837,273	19,138
no A-gen, exh B-gen	80,764	119,975	22,603
triv A-gen, std B-gen	133,903	857,081	22,286
triv A-gen, exh B-gen	77,752	117,105	22,637

where both versions complete – there are 407 such test cases. The interpolant size now measures the total number of literals in all the passed clauses (this corresponds to the previous definition when flp is disabled, and includes the number of unit clauses when flp is enabled). Activation of flp increases the running times (and the number of time-outs), but reduces the total number of iterations by about 4 times. On the other hand, most of the unit clauses detected during flp are irrelevant for the algorithm, and they increase the interpolant size.

Table 3. Measuring the effect of flp on 407 benchmarks

Variant	**Total #iters**	**Total itp size**	**Total Running Time** (s)
no A-gen, exh B-gen	73,071	147,358	15,532
no A-gen, exh B-gen, flp	18,685	184,619	43,374

A couple of additional remarks are in order. First, the size of the final interpolant can serve as a rough estimate for the total memory consumption of an algorithm. Second, in our experience enabling proof-logging techniques for the standard interpolation takes a very small overhead (around 5%), while the overhead of recording blocking (or more generally exchanged) clauses is absolutely negligible. Thus, the running times really represent a comparison between showing unsatisfiability of $\varphi_A \wedge \varphi_B$ using a single call to a SAT-solver and using various variants of a partition-based algorithm. Finally, these experiments should be taken only as a proof of concept of the methods presented. In fact, the current setup benefits our approach in two ways: 1) φ_B is a more restricted formula and so potentially has less satisfying assignments than φ_A, and 2) φ_A is a much simpler formula and so potentially allows for shorter explanation of the inconsistency between a satisfying assignment to φ_B and φ_A, that is for shorter cubes p'''. Indeed, the experiments with roles of φ_A and φ_B reversed resulted in inferior performance (nearly on all instances).

5 Conclusions and Future Work

We described an incremental algorithm for computing Craig interpolants for a pair of mutually unsatisfiable formulas. The most significant advantage of this algorithm is its simplicity – it does not depend on the underlying solver's ability

to produce refutations and thus can be quickly implemented on top of any SAT-solver. In particular, it has the advantage of immediately benefiting from rapid improvements of modern SAT solvers which do not produce proofs.

At this stage, the main contribution of this work is theoretical, rather than practical. We have observed the need for better partition-based algorithms. We have suggested several heuristics towards this goal, but the experimental results show inferior performance (in terms of runtime) compared to a single monolithic call. If more efficient partition-based algorithms are discovered, this work shows how an interpolant may be easily and efficiently reconstructed afterwards. We have also described a technique to vary the strength of the computed interpolant.

On ther other hand, our algorithms are much lighter in terms of memory consumption (even though the size of a proof is linear in the running time of the solver, such proofs are usually huge), and as seen in experiments, the sizes of the interpolants produced are several orders of magnitude smaller than the sizes of the interpolants constructed from proofs. With this in mind (and in the spirit of [PG00]), we can view the algorithm as *the last resort* for computing interpolants, when all of the conventional techniques have failed.

One especially interesting direction for further study is to see how much the proposed technique for computing interpolants can be used inside the original interpolation algorithm for model checking [McM03]. The source of inspiration for this is the success of the IC3 technique [Bra11], which shows that it is often possible to efficiently characterize an over-approximation to states reachable within a certain number of cycles as a conjunction of clauses defined on state-variables only. Note that by splitting the bounded model checking formula as we described – with $\varphi_A = \bigwedge_{i=1}^{k} T(x_i, x_{i+1}) \wedge \bigvee_{i=1}^{k+1} \neg P(x_i)$, and $\varphi_B = J(x_0) \wedge T(x_0, x_1)$, the interpolant φ_I for (φ_A, φ_B) is computed as a DNF, and hence $\neg\varphi_A$ representing an over-approximation of states reachable in one step is precisely in CNF form.

References

[ANORC10] Achá, R.J.A., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Practical algorithms for unsatisfiability proof and core generation in SAT solvers. AI Commun. 23(2-3), 145–157 (2010)

[BKK11] Brauer, J., King, A., Kriener, J.: Existential quantification as incremental SAT. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 191–207. Springer, Heidelberg (2011)

[Bra11] 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)

[CMV09] Chambers, B., Manolios, P., Vroon, D.: Faster SAT solving with better CNF generation. In: DATE, pp. 1590–1595 (2009)

[Cra57] Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. 22(3), 250–268 (1957)

[DGHP09] Desrosiers, C., Galinier, P., Hertz, A., Paroz, S.: Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems. J. Comb. Optim. 18(2), 124–150 (2009)

[ES03] Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)

[GM12] Goldberg, E., Manolios, P.: Quantifier elimination by dependency sequents. CoRR, abs/1201.5653 (2012)

[JS05] Jin, H., Somenzi, F.: Prime clauses for fast enumeration of satisfying assignments to Boolean circuits. In: DAC, pp. 750–753 (2005)

[Kra97] Krajícek, J.: Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. J. Symb. Log. 62(2), 457–486 (1997)

[KW10] Kroening, D., Weissenbacher, G.: Verification and falsification of programs with loops using predicate abstraction. Formal Asp. Comput. 22(2), 105–128 (2010)

[LJH08] Lee, R.-R., Jiang, J.-H.R., Hung, W.-L.: Bi-decomposing large Boolean functions via interpolation and satisfiability solving. In: DAC, pp. 636–641 (2008)

[LJHM07] Lee, C.-C., Jiang, J.-H.R., Huang, C.-Y., Mishchenko, A.: Scalable exploration of functional dependency by interpolation and incremental SAT solving. In: ICCAD, pp. 227–233 (2007)

[McM02] McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 250–264. Springer, Heidelberg (2002)

[McM03] 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)

[McM05] McMillan, K.L.: Applications of Craig interpolants in model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 1–12. Springer, Heidelberg (2005)

[McM10] McMillan, K.L.: Lazy annotation for program testing and verification. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 104–118. Springer, Heidelberg (2010)

[Nad10] Nadel, A.: Boosting minimal unsatisfiable core extraction. In: FMCAD, pp. 221–229 (2010)

[PG00] Park, T.J., Van Gelder, A.: Partitioning methods for satisfiability testing on large formulas. Inf. Comput. 162(1-2), 179–184 (2000)

[Pud97] Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symb. Log. 62(3), 981–998 (1997)

[Sin05] Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)

[SL11] Marques-Silva, J., Lynce, I.: On improving MUS extraction algorithms. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 159–173. Springer, Heidelberg (2011)

[ZM03] Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE, pp. 10880–10885 (2003)