Interpolant based Decision Procedure for Quantifier-Free Presburger Arithmetic

Shuvendu K. Lahiri[1] Krishna K. Mehra[2]

October 18, 2005

Technical Report
MSR-TR-2005-121

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

Interpolant based Decision Procedure for Quantifier-Free Presburger Arithmetic

Shuvendu K. Lahiri[1] and Krishna K. Mehra[2]

[1] Microsoft Research
shuvendu@microsoft.com
[2] Department of Computer Science and Engineering,
Indian Institute of Technology, Kharagpur, India.
kmehra@iitkgp.ac.in

Abstract. Recently, there have been two popular approaches for SAT-based theorem proving — *eager* and *lazy*. Eager approaches are based on a satisfiability preserving translation to a Boolean formula, whereas the *lazy* approaches perform an incremental translation to SAT. Eager approaches are usually based on encoding integers as bit-vectors and suffer from lack of structure and sometime very large size for the bit-vectors. Lazy approaches suffer from large number of invocations of theory decision procedures and the complexity of the decision procedures for integer linear arithmetic.

In this paper, we present a decision procedure for Quantifier-free Presburger arithmetic that is based on alternately under and over-approximating a formula. We use Boolean interpolants to compute the overapproximation. The algorithm seems to address the bottlenecks of both eager and lazy approaches, and improve on both. The algorithm consistently outperforms approaches based on eager and lazy methods on a set of verification benchmarks.

1 Introduction

Decision procedures for quantifier-free theories are the cornerstones of many automated analysis tools for software and hardware. Software verification tools like SLAM [2] use decision procedures to perform automated predicate abstraction and refinement of programs; ESC-JAVA [12] uses decision procedures to discharge verification conditions in static analysis of programs. High-level hardware verification tools including UCLID [4] use decision procedures for checking first-order formulas arising from bounded model checking or invariant checking for models of microprocessor and cache coherence protocols. On the other hand, decision procedures are used to discharge quantifier-free formulas within more general first-order theorem provers like Simplify [10] or theorem provers for more expressible logics such as PVS [25].

The quantifier-free queries that arise in verification typically have a lot of Boolean structure in addition to theory constraints. This often requires an interplay between a search algorithm to case split on the Boolean structure and a decision procedure for the ground theory. In recent years, several approaches have emerged to leverage the rapid improvements in Boolean Satisfiability (SAT) solving [23]. These techniques differ mainly in how closely the SAT solver interacts with theory reasoning. The *eager* approaches rely on translating the quantifier-formula to an equisatisfiable Boolean formula and uses a SAT solver to solve the resultant Boolean formula [4, 33]. The *lazy* techniques create a Boolean abstraction of the first-order formula, and refine it based on assignments that are inconsistent with the underlying theory [1, 3, 11]. Finally, other approaches augment the Davis, Putnam, Logeman and Loveland (DPLL) [7, 8] algorithm for the Boolean SAT solvers with theory specific reasoning [13, 24].

Quantifier-free Presburger (QFP) arithmetic is the quantifier-free fragment of Presburger arithmetic [27], which deals with linear arithmetic constraints along with Boolean connectives. Kroening et al. [18] proposed a framework for integrating the Boolean and the theory reasoning for QFP, based on alternately under and over-approximating the input QFP formula. The algorithm searches for a satisfying solution in a sequence of increasingly large domains, lazily increasing the domain size when no satisfying solution is found in a smaller domain. The proof of unsatisfiability in a domain is used to construct an abstraction of the original formula. The abstractions are checked using a decision procedure for QFP. They illustrate that the procedure terminates and therefore constitutes a decision procedure for QFP.

In this work, we present an alternate implementation of the above framework that is based on Craig's *interpolants* [6, 28] to construct the abstraction, once a formula is unsatisfiable in a given domain. The algorithm also differs in other respects as we do not require a complete decision procedure for QFP to check the abstraction. Finally, we incorporate *learning* from the search in the smaller domains when moving to a larger domain. We compare our approach with Kroening et al.'s work in detail in Section 5.

At a high level, our algorithm works as follows: Consider a QFP formula ϕ. Given a domain D, it is possible to construct a Boolean formula ϕ_u whose satisfiability determines the existence of a satisfying solution for ϕ in the domain D. This Boolean formula underapproximates the original formula ϕ. If ϕ_u is unsatisfiable, then we use Boolean interpolant generation to construct a formula ϕ_o in QFP that is an overapproximation of ϕ (we defer the actual details of the procedure to Section 3). Intuitively, the overapproximation abstracts the reason why ϕ was unsatisfiable in the particular domain D, in terms of the constraints in ϕ. The overapproximation ϕ_o generated serves two purposes: first, if ϕ_o is unsatisfiable, then ϕ is unsatisfiable and secondly, we can generate some conflict clauses from ϕ_o that can be added to ϕ.

The theory of QFP has a *small-model* property for any formula ϕ in the theory, i.e., if ϕ is satisfiable, then it has a satisfying assignment in a finite domain D_{max} determined by the formula ϕ. This ensures that we can start with a small domain D and increase the size lazily until the maximum domain D_{max} is reached. The small-model property also allows us to use an incomplete decision procedure for QFP to check the satisfiability of ϕ_o. In our experience, the maximum domain size D_{max} is almost never reached.

We have implemented the algorithm in Zap theorem prover [34] and provide experimental evaluation of the approach on a set of verification benchmarks. The algorithm seems to consistently outperform approaches based on purely eager encodings (that use the maximum domain size D_{max} to encode an equisatisfiable formula to SAT) or lazy proof explicating frameworks.

The rest of the paper is organized as follows: In Section 2, we describe the background material including eager and lazy approaches for leveraging SAT solvers. We also describe basics of interpolants. In Section 3, we motivate the algorithm and present the details of the decision procedure for QFP. Section 4 describes the experimental evaluation of the approach. We describe related work in Section 5 and finally conclude. Appendix A presents a variation of the algorithm that does not require computing D_{max} for completeness.

2 Preliminaries

In this section, we provide some background on the logic QFP, eager and lazy approaches for solving QFP along with Boolean interpolants.

2.1 Quantifier-free Presburger (QFP) Arithmetic

Presburger arithmetic is the first-order theory of structure $\langle \mathbb{N}, 0, 1, +, \leq \rangle$, where $\mathbb{N}$ denote the set of natural numbers. Since every integer variable can be expressed as the difference of two natural numbers, we assume that the underlying domain is the set of integers $\mathbb{Z}$. Quantifier-free Presburger Arithmetic (QFP) is the quantifier-free fragment of Presburger arithmetic. Let X be a set of integer variables. An atomic formula (*atomic-formula*) in this theory (also refered to as a linear constraint) is an expression of the form:

$$\sum_i a_i * x_i \leq c_i,$$

where $x_i \in X$ and the coefficients a_i and c_i are constants in $\mathbb{Z}$. A *formula* in this QFP is a Boolean combination of atomic formulas:

$$\begin{aligned} \mathit{formula} ::= & \ \texttt{true} \mid \texttt{false} \mid \mathit{atomic\text{-}formula} \\ \mid & \ \mathit{formula} \wedge \mathit{formula} \mid \mathit{formula} \vee \mathit{formula} \mid \neg \mathit{formula} \end{aligned}$$

Observe that the other relational operators $\{=, \neq, <, >, \geq\}$ can be expressed in QFP.

A formula ϕ in QFP is *satisfiable* if there is an assignment $\rho : X \rightarrow \mathbb{Z}$ that maps each $x_i \in X$ to an integer value, such that the evaluation of ϕ under ρ is `true`. A formula ϕ is unsatisfiable if there is no assignment ρ under which ϕ evaluates to `true`.

A *monome* is a conjunction of atomic formulas in QFP. Checking the satisfiability of a monome in QFP is NP-complete [26]. However, efficient algorithms based on branch-and-bound heuristics are implemented in various Integer Linear Programming (ILP) solvers like LP_SOLVE [21] and commercial tools like CPLEX [15] to solve this fragment. The complexity of checking the satisfiability of QFP formulas is no worse than checking satisfiability of a conjunction of atomic formulas. Previous studies have indicated that ILP solvers do not perform well for QFP formulas with a significant Boolean structure, that arise from verification problems [30, 18]. To circumvent this problem, two methods have been proposed in recent years to leverage backtracking search of modern Boolean satisfiability solvers:

2.2 Lazy Methods

The *lazy* methods [1, 29, 3] leverage the backtracking of modern Boolean Satisfiability (SAT) solvers to case-split on the Boolean structure in the QFP and use a ILP solver to check the satisfiability of a conjunction of linear constraints. The algorithms can be loosely described as follows:

1. The QFP formula ϕ is abstracted to a Boolean formula ϕ_a by replacing each atomic constraint with a Boolean variable.
2. The SAT solver enumerates satisfying solutions over ϕ_a and uses the ILP solver to validate the solution over the QFP theory.
3. If the satisfying solution is consistent with the QFP theory, the procedure returns satisfiable. Otherwise, the solver returns a conflict clause (a tautology in QFP) that rules out the current assignment. Typically, the literals that appear in the proof of unsatisfiability of the current assignment, constitute the conflict clause [3, 11]. This helps towards finding a "minimal" unsatisfiable core to rule out more than just the present satisfying assignment. The conflict clause is added to ϕ_a and Step 2 is repeated.

2.3 Eager Methods

Eager methods (e.g. implemented in UCLID [20]) are based on translating a QFP formula ϕ to an *equisatisfiable* Boolean formula ϕ_{bool}, such that ϕ is satisfiable if and only if ϕ_{bool} is satisfiable. Since the problem of deciding the satisfiability of QFP is in NP, QFP enjoys a *small-model* property — a QFP formula ϕ is satisfiable over $\mathbb{Z}$ if and only if it is satisfiable over a finite domain $\mathbb{D} \subseteq \mathbb{Z}$. The measure $\log(\mathbb{D})$ denotes the number of Boolean variables to represent the domain $\mathbb{D}$.

Let m be the number of atomic formulas in ϕ and n be the number of variables in X. When the set of atomic constraints in ϕ is restricted to *difference logic* constraints (where the atomic formulas are restricted to $x_i - x_j \leq c$), the size of the domain $\log(\mathbb{D})$ is $O(\log(n) + \log(c_{max}))$, where c_{max} is the absolute value of largest constant c that appears in any of the constraints. Seshia and Bryant [30] show that for general linear constraints, $\log(\mathbb{D})$ is bound by

$$O(\log(n) + \log(m) + \log(c_{max}) + k * (\log(a_{max}) + \log(w))), \qquad (1)$$

where a_{max} is the absolute value of any k is the number of non-difference atomic formulas in ϕ and w is the maximum number of variables in any linear constraint. When the number of non-difference constraints in ϕ is small, the size of $\log(\mathbb{D})$ is almost logarithmic in n and m.

The above bounds suggest that one can encode each variable $x \in X$ using $\log(\mathbb{D})$ Boolean variables and translate ϕ to a Boolean formula. The arithmetic operator $+$ can is encoded as an word adder, $*$ is implemented as a shift operator since only variables are multiplied with constants. Similarly, the relational operator $\leq$ is encoded as word comparator. The size of the resultant Boolean formula ϕ_{bool} incur a polynomial blowup (typically $\log(\mathbb{D})$) over ϕ. Finally, the resultant formula can be checked using any state-of-the-art SAT solvers. We refer to this encoding as *small-model* encoding of a first-order formula.

2.4 Boolean Interpolants

Consider two satisfiable Boolean formulas A and B such that $A \wedge B = \texttt{false}$. Let V be the set of Boolean variables shared by both A and B. An (Boolean) *interpolant* [6] of the pair of formulas (A, B) is a Boolean formula I, such that:

1. $A \Rightarrow I$,
2. $I \wedge B = \texttt{false}$, and
3. The set of variables in I is a subset of V.

Pudlak [28] showed that given the proof of unsatisfiability of $A \wedge B$, an interpolant I can be obtained in time linear to the size of the proof. A description of the algorithm when both A and B are present in conjunctive normal form (CNF) has been described by McMillan [22]. The motivation for using the interpolant I of (A, B) is usually two-fold: (a) it is an abstraction of A that is sufficient to prove the unsatisfiability with B and (b) it is over the common variables of A and B.

3 Interpolant-based Decision Procedure for QFP

3.1 Comparison of Eager and Lazy methods

In this section, we first address the following strength and weaknesses of the eager and the lazy approaches.

Loss of structure: Small-model encoding based eager methods suffer from the fact that the translation to a Boolean formula results in a loss of structure of the formula. For example, consider the simple formula:

$$\phi = x \leq y \wedge y \leq z \wedge z < x$$

There is a polynomial algorithm for deciding the satisfiability of such conjucntion of linear arithmetic formulas, based on negative cycle detection [5]. However, converting to a Boolean formula introduces a lot of disjunctions resulting from the encoding of $\leq$ as a circuit. This results in SAT solver to perform a lot of case splits before detecting unsatisfiability.

Large small-model size: The appeal of small-model based encoding to SAT lies in the fact that it provides only a polynomial blowup when translating a linear arithmetic formula to a Boolean formula. However, the blowup can be linear in the size in the size of the number of constraints and variables in the first-order formula, when the number of non-difference constraints is large. The small model size also explodes in the presence of large constants in the formula. We provide evidence of this blowup on some examples in Section 4.

Large number of complex theory decision procedure calls: The lazy approaches (e.g. Verifun [11]) suffer from the need to invoke a decision procedure for the first-order theory a very large number of times in the presence of a lot of Boolean structure (i.e. disjunctions) in the formula. Moreover, since the decision procedures take over only after the SAT solver has found a Boolean model, the monome handed to the theory can have a lot of theory literals. When the monome is unsatisfiable in the theory, if often happens because of very small subset of literals in the monome. Although the decision procedure can figure out the core reason for unsatisfiability (using the proof of unsatisfiability), the decision procedure is often overwhelmed with the size of the monome that it obtains from the SAT solver.

Handling Integer theories: The presence of integer linear arithmetic (a non-convex theory) results in a NP-complete decision procedure for a monome. The lazy approaches rely on a Nelson-Oppen architecture to construct a decision procedure for mixed theories. The combination requires each theory to generate a disjunction of equalities that is an additional overhead. Even for the theory of difference logic, generating disjunction of equalities is NP-complete [19].

Producing satisfying assignments: The eager method has the advantage that it can generate satisfying solutions easily (even in the presence of other theories) since one can map back an assignment to the Boolean formula to an assignment that satisfies the original QFP formula. Moreover, if the method searches for a solution by gradually growing the domain size, there is a good chance of producing small satisfying assignments. However, there is no good way to learn the conflicts that resulted in the smaller domain when moving to a larger domain.

3.2 Interpolant based algorithm

In this section, we describe an algorithm to decide the satisfiability of a QFP formula ϕ. The algorithm alternates between two phases that check the satisfiability of an underapproximation and overapproximation of ϕ, respectively. In the underapproximation phase, the algorithm creates a formula ϕ_u that is an underapproximation of ϕ by *restricting* the domain of each variable that appear in the formula. If the formula ϕ_u has a satisfying assignment within the small bounds, it reports satisfiable. Otherwise, it computes an abstraction ϕ_o of ϕ by using Boolean interpolants. The abstract formula ϕ_o is then checked for (un) satisfiability. If ϕ_o is unsatisfiable, then ϕ is unsatisfiable. Otherwise, we repeat the phases with an increased domain for the variables. The algorithm adds additional clauses that it discovers while checking ϕ_o to the formula ϕ, to speed up the underapproximation phase in the further iterations. We describe the algorithm in details in the next few paragraphs.

1. **[Input]**. Given a QFP formula ϕ.
2. **[Encoding]**. Construct the Boolean structure of ϕ by replacing an atomic formula e with a Boolean variable b_e in ϕ. The resultant formula ϕ_b will be referred to as the (Boolean) *skeleton* of ϕ. We introduce a set of fresh variables $B = \{b_e \mid e \in atoms(\phi)\}$ and create a formula

$$\phi_{th} = \left(\bigwedge_{e \in atoms(\phi)} b_e \Leftrightarrow e \right),$$

called the *theory* portion of ϕ. Finally, the formula representing the conjunction of the Boolean skeleton and the theory component is called ϕ_{bt}:

$$\phi_{bt} \doteq \phi_b \wedge \phi_{th}$$

3. **[Initialize]**. Compute the maximum model size D_{max} for each variable using Equation 1. The initial domain for each variable is restricted to D, i.e. each variable $v \in vars(\phi)$ takes values in $(-D, \ldots, D]$.
4. **[Boolean UNSAT]**. We first check if the skeleton of ϕ_{bt} is unsatisfiable using the SAT solver. If ϕ_b is unsatisfiable, the algorithm returns UNSATISFIABLE.
5. **[Underapproximation]**. We construct a Boolean formula for ϕ_{th} by encoding each variable in ϕ as a symbolic bit vector with $\log(2 * D)$ Boolean variables. The arithmetic operators $+$ and $*$ are modeled as Boolean adders and multipliers. Since we do not have non-linear expressions, the multiplication is implemented as as shifter circuit. Finally, the relational operations $\{\leq, =\}$ are modeled as Boolean comparators. We use $BE(\phi_{th}, D)$ to denote the Boolean translation. Let the resultant Boolean formula be ϕ_u, an underapproximation of ϕ_{bt}:

$$\phi_u \doteq \phi_b \wedge BE(\phi_{th}, D)$$

We check if ϕ_u is satisfiable using a SAT solver. If ϕ_u is satisfiable, the algorithm returns SATISFIABLE. If ϕ_u is unsatisfiable and $D \geq D_{max}$, we return UNSATISFIABLE.
6. **[Overapproximation]**. If ϕ_u is unsatisfiable, compute the Boolean interpolant ϕ_I of ϕ_b and $BE(\phi_{th}, D)$. We construct the formula ϕ_o which is an overapproximation of ϕ_{bt} as follows:

$$\phi_o \doteq \phi_I \wedge \phi_{th}$$

We check the satisfiability of ϕ_o using a conflict clause generator $CCG()$ for QFP (described later in this section). If $CCG(\phi_o)$ returns UNSATISFIABLE, the algorithm UNSATISFIABLE. Otherwise, $CCG(\phi_o)$ returns a set of (conflict) clauses ϕ_c that are tautologies in the theory of QFP.

We augment the conflict clauses ϕ_c to ϕ_b to create a more constrained Boolean skeleton of ϕ_{bt}:

$$\phi_b \leftarrow \phi_b \wedge \phi_c$$

7. [**Repeat**]. Increase the bound D for each variable by some predetermined amount $\delta > 0$, i.e. $D = D + \delta$. Goto step 4.

Let us first observe a couple of points about this algorithm.

- When ϕ_u is unsatisfiable in step 5, we know that the two components of ϕ_u, namely ϕ_b and $BE(\phi_{th}, D)$ are each satisfiable in isolation. This is because step 4 ensures that ϕ_b is satisfiable and $BE(\phi_{th}, D)$ is always satisfiable for any domain.
- The common variables in ϕ_b and $BE(\phi_{th}, D)$ are only the variables from B. This allows us to construct an interpolant in terms of the B variables, independent of the integer variables or the variables introduced during translating an integer variable to a symbolic bit vector.

3.3 Conflict Clause Generator (*CCG*)

The conflict clause generator algorithm takes as input the QFP formula $\phi_o \doteq \phi_I \wedge \phi_{th}$, where ϕ_I is a Boolean formula over B variables and checks for the unsatisfiability of the formula:

- If it returns UNSATISFIABLE, then the formula ϕ_o is unsatisfiable.
- Otherwise, it returns a conjunction of clauses $\phi_c \doteq \bigwedge_i c_i$, such that $\phi_o \implies \phi_c$ and each c_i is a disjunction of literals over B.

The conflict clauses generator can be implemented as a simple variation of a lazy SAT-based theorem proving framework as follows: Initially, ϕ_c is assigned `true`. The SAT solver enumerates assignments to B variables that satisfy ϕ_I and checks if the assignment satisfies ϕ_{th} using a (possibly incomplete) decision procedure for a conjunction of linear arithmetic constraints.

If the assignment is found unsatisfiable by the linear arithmetic decision procedure, the decision procedure returns a "proof core", a subset of constraints that are inconsistent. The negation of the proof core is a clause c_i that is added to ϕ_c (after mapping an atomic expression e to the corresponding b_e variable). The clause c_i is also added to the ϕ_I to prevent it from generating the same assignment. However, if the linear arithmetic decision procedure returns satisfiable, we simply add the negation of the assignment to ϕ_I and repeat the loop. The process is continued until the number of satisfying assignments (that are also satisfiable in the linear arithmetic theory) does not exceed some threshold. We currently limit this threshold to be 5, i.e. after obtaining more than 5 satisfying solutions, we return ϕ_c.

The usefulness of the conflict clause generator comes from the fact that it adds some structure back to the Boolean formula ϕ_u. The conflict clauses added aid the SAT-solver to avoid some case-splits when it is checking the satisfiability of the formula ϕ_u in the subsequent iterations with larger D values.

3.4 Correctness

It is not hard to show that the algorithm is sound and complete. The algorithm returns with SATISFIABLE or UNSATISFIABLE in steps 4, 5, and 6. We use the following invariant on the algorithm:

Lemma 1. *At any point in the algorithm, ϕ_{bt} is equisatisfiable with ϕ. In step 5, $\phi_u \implies \phi_{bt}$, and in step 6, $\phi_{bt} \implies \phi_o$.*

Proof. The proof follows simply by induction on the number of iterations of the algorithm.

For the first iteration, ϕ_{bt} is equisatisfiable with ϕ since $\phi = \exists B : \phi_{bt}$. Since ϕ_u simply restricts the domain of each variable in ϕ_{bt}, clearly $\phi_u \implies \phi_{bt}$. In step 6, assume ϕ_u is unsatisfiable. Recall that ϕ_b is not unsatisfiable because we have ensured that in step 4. We also know that the formula $BE(\phi_{th}, D)$ is satisfiable. This is because it just says that the value of b_e is the same as the value of e. There are no constraints on any b_e variables or the atomic constraints e. This explains the reason why we have step 4 explicitly in the algorithm. Moreover, recall that ϕ_u is a Boolean formula. Therefore, we can construct a Boolean interpolant ϕ_I for the pair $(\phi_b, BE(\phi_{th}, D))$. Since $\phi_b \implies \phi_I$, $\phi_{bt} \implies \phi_o$.

Now, let us assume that the lemma holds for some iteration. We want to prove that it holds for the next iteration. Recall, that the only step in which ϕ_b changes is in step 6. We also know that $\phi_o \implies \phi_c$ (from the property of the conflict clause generator), and hence $\phi_{bt} \implies \phi_c$. This in turn implies that $\phi_{bt} \implies (\phi_{bt} \wedge \phi_c)$. Moreover, $(\phi_c \wedge \phi_{bt} \implies \phi_{bt}$. Hence $(\phi_c \wedge \phi_{bt}) \Leftrightarrow \phi_{bt}$. Therefore, the formula ϕ_{bt} is equivalent across all iterations steps. This preserves other parts of the lemma.

Theorem 1. *The algorithm described above is sound and complete for QFP.*

Proof. It is easy to see that the algorithm terminates, since D is monotonically increased and finally crosses D_{max}. Observe that the algorithm returns SATISFIABLE, if and only when it finds ϕ_u SATISFIABLE. Lemma 1 ensures soundness of the algorithm. The algorithm returns UNSATISFIABLE if and only if a formula weaker than ϕ_{bt} is unsatisfiable. The weaker formulas in step 4, step 5 and step 6 are ϕ_b, ϕ_u (only when $D \geq D_{max}$) and ϕ_o respectively.

3.5 Example

In this section, we illustrate the working of the algorithm on a simple example. Consider the following formula ϕ:

$$\begin{aligned} \phi \doteq\ & (x < y) \wedge (y < z) \wedge (z < w \vee z < x) \wedge \\ & (\neg (z < w) \vee x < 2.w + 1) \wedge (\neg (z < w) \vee \neg (x < 2.w + 1)) \end{aligned} \tag{2}$$

Let us introduce Boolean variables $\{b_{x<y},\ b_{y<z},\ b_{z<w},\ \ldots\}$ for the atomic constraints in ϕ. Let ϕ_{th} denote the constraint $[\bigwedge_e b_e \Leftrightarrow e]$ as before and ϕ_b be the Boolean skeleton of ϕ.

Let us start with a domain D where each variable takes values in $\{0, 1\}$. The underapproximation of ϕ created by restricting each variable to D is unsatisfiable. Let

$$\phi_I = (b_{x<y} \wedge b_{y<z} \wedge (b_{z<w} \vee b_{z<x}))$$

be the interpolant generated in step 6 of the algorithm. In this domain, the formula $(x < y \wedge y < z \wedge (z < w \vee z < x))$ is unsatisfiable. The monome $x < y \wedge y < z \wedge z < w$ can't be satisfied because each variable can only take two values, and the monome $x < y \wedge y < z \wedge z < x$ is always unsatisfiable.

Now $\phi_o \doteq \phi_I \wedge \phi_{th}$ is fed to the conflict clause generator. Even though ϕ_o is satisfiable, it generates a conflict clause

$$\phi_c \doteq (\neg b_{x<y} \vee \neg b_{y<z} \vee \neg b_{z<x}) \, .$$

Once we add the conflict clause ϕ_c to ϕ_b, we obtain unsatisfiable in the Boolean skeleton.

This example illustrates the interesting case where the unsatisfiability of ϕ is detected in step 4 of the algorithm.

3.6 Discussion

In this section, we illustrated a decision procedure for QFP (or in general for a theory T) that uses a sound procedure for deciding a formula in QFP (or in theory T) as part of the conflict clause generator. The $CCG()$ decision procedure need not be complete for the theory. One can use a fast decision procedure for a subset of integer linear arithmetic (e.g. based on negative cycle detection algorithms), coupled with simple rules for more general arithmetic. This enables us to plug in any fast decision procedure for a subset of QFP (e.g. difference logic) as the $CCG()$. To ensure the completeness of the procedure, we need to compute the small-model size D_{max} for a formula in the theory and search this domain in the worst case.

However, we can avoid the need to compute the small-model size D_{max}, if we have a sound and complete decision procedure for QFP as the conflict clause generator. In this case, one can ensure that the sequence of overapproximations $\phi_o^1, \phi_o^2, \ldots,$ (where ϕ_o^i denotes the overapproximation generated during the iteration i of the above algorithm) for the input formula ϕ, will eventually converge to ϕ. Details of such a decision procedure is described in Appendix A.

4 Experiments

We have implemented a prototype of the technique in the theorem prover Zap [34]. We compare the following three variants of eager and lazy decision procedure implemented in Zap.

Verifun. This is the implementation of the lazy proof-explicating theorem prover described in Section 2.2 based on the Verifun architecture [11]. The linear arithmetic is restricted to Unit Two Variable Per Inequality (UTVPI) constraints [16], for efficiency. We do not consider examples where Verifun returns satisfiable because of the incompleteness of the procedure. The UTVPI decision procedure suffices to prove a lot of examples of our example set even in the presence of more linear arithmetic.

Eager. This implements a small-domain encoding of a quantifier-free formula to a Boolean formula [4] using the eager encoding technique mentioned in Section 2.3. For encoding QFP, it uses the small model size computed by Seshia et al. [30] reported in Equation 1.

EaZI. This is the implementation of the algorithm *Eager Zap with Interpolants* (EaZI) described in Section 3. The initial size of the domain is restricted to assign 3 bits for each integer variable. Each subsequent iteration increases the number of bits per variable by 2. For the $CCG()$ generation, it uses the Verifun procedure described above. Observe that since the Verifun procedure is restricted to UTVPI, $CCG()$ is not complete for general linear arithmetic.

In all the variants, we used SharpSAT as the Boolean SAT solver. SharpSAT is a variant of ZChaff [23] developed by Lintao Zhang at Microsoft. We ran the experiments on a 3GHz machine running Windows with 1 GB of memory. A timeout of 300 seconds was set for each benchmark.

4.1 Benchmarks

The set of benchmarks fall into two categories:

- *Mathsat*. This a set of QFP benchmarks obtained from the timed automata verification problems [1]. An analysis of the formulas in previous study [9] suggests that the coefficients of the QFP formulas are restricted to $\{-1, 0, 1\}$, with the number of integer variables ranging from around 10 to around 150. Most of the variables are Booleans.
- *SAL*. This set of benchmarks [9] consists of formulas derived from bounded model checking of linear and hybrid systems and from test-case generation for embedded systems. Most of the benchmarks have significant linear arithmetic constraints and the number of integer variables range from tens to hundreds of variables.

4.2 Results

In this section, we present the results of running the three approaches on the Mathsat and SAL benchmarks. We analyze the results for different options:

Lazy vs. Eager Figure 1 compares the Verifun and the Eager approach on the benchmarks. Verifun scales better than Eager on the Mathsat benchmarks whereas Eager outperforms Verifun on SAL benchmarks. We believe the difference in performance of Verifun on the two sets of benchmarks can be explained by the number of times it invokes a theory decision procedure. Column 3 of

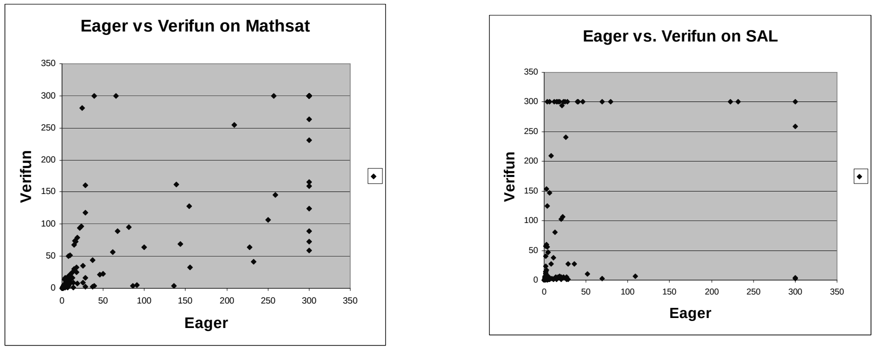


Fig. 1. Eager vs. Verifun on Mathsat and Sal benchmarks.

Figure 4.2 (marked "# Verifun Loops" for "Verifun") shows that the number of calls to the theory decision procedure for SAL benchmarks is at least an order greater than in the case for Mathsat benchmarks. The results suggest that neither Eager nor Verifun is robust enough for a large set of benchmarks.

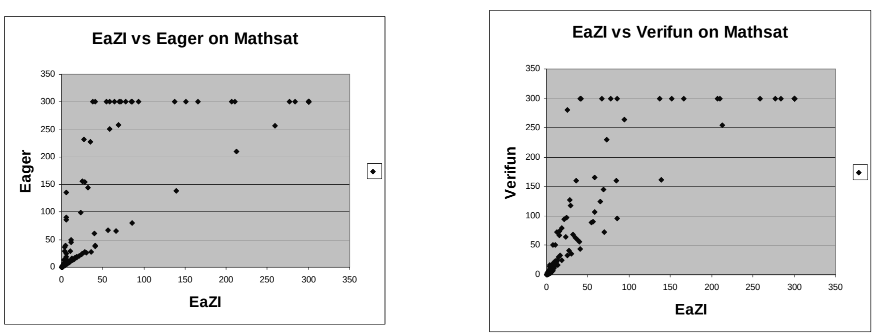


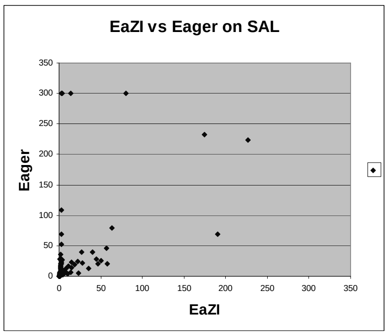
Fig. 2. Eazi vs. Eager and Verifun on Mathsat Benchmarks.

Lazy vs. EaZI Figure 2 and Figure 3 compares EaZI with Eager and Verifun on Mathsat and SAL benchmarks respectively.

EaZI outperforms Verifun consistently on both the set of benchmarks. To understand the improvement, we extracted some information for a subset of benchmarks. Figure 4.2 compares the two approaches in terms of three metric (a) number of times a theory decision procedure was invoked, (b) the average size of the number of constraints involved in the monome passed to the decision procedure and (c) total time spent in the theory decision procedure.

We observed the following characteristics across a wide set of benchmarks:

– The number of times a theory decision procedure is invoked is usually much smaller in the case of EaZI. This can be explained because the abstraction of



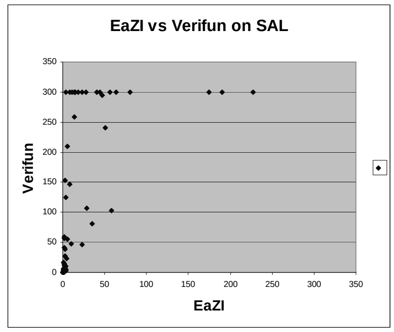


Fig. 3. Eazi vs. Eager and Verifun on SAL Benchmarks.

Example		# Verifun Loops		Avg. Monome size		Total theory time(secs)	
		Verifun	EaZI	Verifun	EaZI	Verifun	EaZI
sal	Carpark2-t1-4	200	2	755	1	222.27	0.06
	fischer3-mutex-5	372	72	399	66	116.84	2.45
	fischer6-mutex-3	143	18	426	19	39.28	0.17
	fischer9-mutex-3	221	40	603	15	107.14	0.25
	inf-bakery-invalid-10	180	0	211	-	47.90	0
	inf-bakery-mutex-9	426	66	200	38	149.77	1.11
	lpsat-goal-12	233	45	2019	19	87.75	0.34
	tgc_io-safe-7	198	189	484	96	181.26	11.28
	windowreal-safe-4	185	14	328	22	13.14	0.17
	windowreal-safe2-4	254	8	328	23	19.14	0.17
mathsat	FISCHER10-5-fair	19	3	969	49	30.46	0.34
	FISCHER11-5-fair	21	25	1072	113	42.54	11.53
	FISCHER3-8-fair	93	0	478	-	66.89	0
	FISCHER5-7-fair	81	52	682	93	108.69	8.85
	FISCHER8-6-fair	36	40	935	108	94.72	14.29
	PO3-10-PO3	18	0	686	-	18.89	0
	PO3-9-PO3	38	3	618	20	25.53	0.07

Fig. 4. Analysis of EaZI vs. Verifun on Mathsat and SAL.

the original formula ϕ_o has lot more conflicts in the Boolean structure and thus comes to theory less often.
- The size of monome and the total time spent in the theory is reduced at least by an order of magnitude in most cases. This is because the abstraction ϕ_o contains a lot few constraints that ϕ.
- The main bottleneck of the EaZI method shifts from the theory decision procedure to SAT. In most cases, the interpolant generation consumes more than 80% of the total time. We believe the performance of our algorithm will further improve with improvement in the the interpolant generation implementation in SharpSAT.

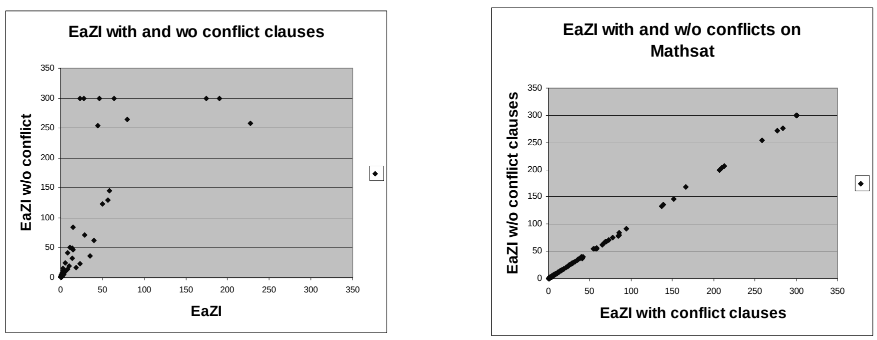


Fig. 5. Effect on conflict clauses in Eazi for SAL and Mathsat Benchmarks.

Eager vs. EaZI Figure 2 and Figure 3 suggests that EaZI also outperforms the Eager method in most of the examples in this set. There following observations help explain the speedup:

- The small model size for the most examples calculated by Equation 1 requires the Eager method to encode each variable with more than 15 bits in most case. In some cases with large number non-difference constraints, the number of bits to encode each variable exceeds 100 bits.
- Although the EaZI method relies on the small model size for completeness, it never reaches this maximum domain size for either proving unsatisfiability or satisfiability. For satisfiable instances, it finds a satisfying solution using a small number bits for most examples. For unsatisfiable cases, the procedure exits from the $CCG()$ procedure with UNSATISFIABLE, or the conflict clauses generated from $CCG()$ makes the Boolean part of the formula ϕ_{bt} unsatisfiable. Even for the unsatisfiable instances, we need to increase the number of bits to at most 8 in most cases.
- The Eager approach outperforms the EaZI approach on some of the SAL benchmarks. This is primarily because the number of bits to encode each

variable is less than 20 and the overhead of computing interpolants in EaZI offsets the advantage of learning from smaller domain size.

Adding Conflict Clauses We conjectured that the advantage of EaZI over Eager comes from the fact that we add more structure (the conflict clauses from $CCG()$) to ϕ_b and thereby aids the SAT solver to prune away the search space efficiently. Figure 5 compares the EaZI approach with and without the conflict clauses from $CCG()$. The conflict clauses make marked difference in the results for the SAL benchmarks. Since the dominant time in the SAL experiments is in solving the Boolean formula ϕ_u, addition of the conflict clauses help the SAT solver. However, adding the conflict clauses made almost no difference for the Mathsat examples. For most of the Mathsat examples, we found that the abstraction ϕ_o was proved unsatisfiable by the conflict clause generator after the first iteration. Hence the conflict clauses do not play a part in most of these examples.

5 Related Work

In this section, we compare the interpolant based decision procedure for QFP for other decision procedures for QFP or its restricted fragments. The use of interpolants has been recently explored for finite-state model checking [22] and in refining the abstractions for software verification [14, 17].

Eager approaches for translating a QFP formula to an equisatisfiable Boolean formula employ either the small-model encoding discusses in Section 2.3 or add all the theory constraints to the formula [33]. The latter approach can result in an exponential number of constraints being added to the original formula. This translation is often the bottleneck in the method. In [31], the authors encode disjoint set of constraints in a formula using either the small-model encoding or by adding all the theory constraints. The approach was restricted to the difference logic fragment of QFP, and use the number of constraints in the formula to determine the encoding. In contrast, our approach uses small-encoding but increases the size of encoding lazily starting from a small size. Beside, we only add a very small set of theory constraints as conflict clauses in a more lazy manner.

Lazy approaches [3, 11] use the lazy proof-exlicating framework described in Section 2.2. The main bottleneck of these approaches appear to be the large number of invocations of the theory decision procedure. The monomes passed to the decision procedure are often large, even though the reason for unsatisfiability of the monome is often simple and small. Our approach addresses this problem by creating an abstraction of the original formula by using the interpolants and using the lazy approach as a mean to generate conflict clauses. Our experiments indicate that the size of the monome passed to the decision procedure is reduced by more than an order of magnitude and the time spent in the theory decision procedure is reduced considerably. Mathsat [1] use a *layered* approach, where a sequence of increasingly complete (and therefore more complex) decision procedures to decide a monome. However, to our knowledge, the approach still suffers from passing large monomes to the linear arithmetic decision procedures.

DPLL(T) [13, 24] uses a closer integration of the theory decision procedure with the SAT solver. In addition to the Boolean constraint propagation in the SAT solver, the theory participates in the constraint propagation by adding all the theory facts implied by the theory in the current context. This enables detection of early unsatisfiability than the lazy approaches. However, the decision procedures for the theories become more complex as they need to support the generation of all the facts that are implied by a set of constraints. This may increase the complexity of the ILP decision procedures that already have an exponential worst-case complexity. A promising approach has been suggested by Sheini and Sakallah [32], where they integrate an UTVPI decision procedure in DPLL(T) framework and use the general ILP solver in a lazy framework. They ILP solver is moreover equipped to generate some UTVPI facts implied by a set of general linear constraints. More experiments are required to understand the scalability of this approach for benchmarks containing significant number of non-UTVPI constraints.

The approach closest to our work is Kroening et al.'s [18] work on deciding QFP formulas using Boolean proof of unsatisfiability. Kroening et al. construct a equisatisfiable clausal representation of the original QFP formula by introducing additional variables. The clausal form is encoded to a Boolean formula by performing small-model encoding starting with a small size. An abstraction of the original clausal formula is constructed by choosing a subset of clauses that appear in the proof of unsatisfiability of the Boolean formula. This abstraction is checked using a sound and complete decision procedure for QFP. The sequence is repeated with increasing small-model encoding size. Although similar in many aspects, our approach differ from this work in several ways. First, this method appears to require a clausal representation of the original formula (that can introduce auxiliary variables), and the abstraction is limited to be a subset of the clauses in this representation. The abstraction is obtained in a fairly syntactic fashion, by considering the subset of clauses for which there is a corresponding clause in the proof of unsatisfiability for the Boolean formula. We believe that the use of interpolants results in a more semantic method to construct the abstraction. This might often result in more concise abstractions being generated. Secondly, unlike their approach we do not require a complete decision procedure for QFP (for checking the abstraction) to ensure the completeness of the procedure. Their experiments (although on a different set of benchmarks) indicate that the lazy decision procedure for ILP dominates the total time of the procedure. Finally, we add conflict clauses from the abstractions generated from smaller domain sizes; this allows us to prune the search space when searching in a larger domain.

6 Conclusion

In this paper, we present a framework for using simultaneous under and over approximations to decide a quantifier-free first-order formula. The small-model encoding is used to create the underapproximation of the formula and the interpolant generation from the proofs of unsatisfiability gives an overapproximation of the formula. The method also demonstrates a mechanism to leverage some conflict clauses learned during searching in a smaller domain size, to prune the search space during the later search. One of the directions of future work is to implement Kroening et al.'s scheme and empirically compare the two approaches on the set of benchmarks.

References

1. G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *LNCS*, pages 195–210. Springer Verlag, July 2002.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, Snowbird, Utah, June, 2001. *SIGPLAN Notices,* 36(5), May 2001.
3. C. W. Barrett, D. L. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 236–249, July 2002.
4. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.
5. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.
6. W. Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *J. Symbolic Logic*, 22:250–268, 1957.
7. M. Davis, G.eorge Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
8. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
9. Leonardo M. de Moura and Harald Rueß. An experimental evaluation of ground decision procedures. In *Computer Aided Verification (CAV '04)*, LNCS 3114, pages 162–174. Springer-Verlag, 2004.
10. D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical report, HPL-2003-148, 2003.
11. C. Flanagan, R. Joshi, X. Ou, and J. Saxe. Theorem Proving usign Lazy Proof Explication. In *Computer-Aided Verification (CAV 2003)*, LNCS 2725, pages 355–367. Springer-Verlag, 2003.
12. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.

13. Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast Decision Procedures. In *Computer Aided Verification (CAV '04)*, LNCS 3114, pages 175–188. Springer-Verlag, 2004.
14. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Symposium on Principles of programming languages (POPL '04)*, pages 232–244. ACM Press, 2004.
15. ILOG CPLEX. Available at `http://ilog.com/products/cplex`.
16. J. Jaffar, M. J. Maher, P. J. Stuckey, and H. C. Yap. Beyond Finite Domains. In Alan Borning, editor, *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94*, volume 874 of *LNCS 874*, pages 86–94. Springer-Verlag, 1994.
17. Ranjit Jhala and Kenneth L. McMillan. Interpolant-based transition relation approximation. In *Computer Aided Verification, 17th International Conference, CAV 2005*, volume 3576 of *LNCS*, pages 39–51. Springer, 2005.
18. Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, and Ofer Strichman. Abstraction-based satisfiability solving of presburger arithmetic. In *Computer Aided Verification (CAV '04)*, LNCS 3114, pages 308–320. Springer-Verlag, 2004.
19. Shuvendu K. Lahiri and Madanlal Musuvathi. An efficient decision procedure for utvpi constraints. In Bernhard Gramlich, editor, *Frontiers of Combining Systems, FroCos 2005*, volume 3717 of *LNCS*, pages 168–183. Springer, 2005.
20. Shuvendu K. Lahiri and Sanjit A. Seshia. The uclid decision procedure. In *Computer Aided Verification (CAV '04)*, LNCS 3114, pages 475–478. Springer-Verlag, 2004.
21. LP_SOLVE. Available at `http://groups.yahoo.com/group/lp_solve/`.
22. K.L. McMillan. Interpolation and sat-based model checking. In *CAV 03: Computer-Aided Verification*, LNCS 2725, pages 1–13. Springer-Verlag, 2003.
23. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC '01)*, 2001.
24. R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In *Computer Aided Verification, 17th International Conference, CAV 2005*, LNCS, pages 321–334. Springer, 2005.
25. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, June 1992.
26. Christos H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, 1981.
27. M. Presburger. Uber die Vollstfindigkeit eines gewissea Systems Arithmetic ganzer Zahlen in welchem die Addition als eiilzige Operation hervortritt. In *Comptes-rendus du I Congr s de Mathematiciens de Pays Slaves*, pages 395:92–101, 1930.
28. P. Pudl'ak. Lower bounds for resolution and cutting planes proofs and monotone computations. *J. of Symbolic Logic*, 62(3):981–998, 1995.
29. H. Rueß and N. Shankar. Solving linear arithmetic constraints. Technical Report CSL-SRI-04-01, SRI International, January 2004.
30. S. A. Seshia and R. E. Bryant. Deciding quantifier-free presburger formulas using parameterized solution bounds. In *19th IEEE Symposium of Logic in Computer Science(LICS '04)*. IEEE Computer Society, July 2004.
31. S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A Hybrid SAT-based Decision Procedure for Separation Logic with Uninterpreted Functions. In *Proc. Design Automation Conference (DAC)*, pages 425–430, June 2003.
32. Hossein M. Sheini and Karem A. Sakallah. A scalable method for solving satisfiability of integer linear arithmetic logic. In *Theory and Applications of Satisfiability*

Testing, 8th International Conference, SAT 2005*, volume 3569 of *LNCS*, pages 241–256. Springer, 2005.
33. Ofer Strichman. On solving presburger and linear arithmetic with sat. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 160–170, 2002.
34. Zap Project. Available at `http://www.research.microsoft.com/tvm`.

A Termination in the absence of D_{max}

In the previous section, the termination argument required computing a maximum domain size D_{max}. When $D \geq D_{max}$, then the formula ϕ_u is equisatisfiable with ϕ_{bt}, and therefore also with ϕ. In this section, we show that we can modify the algorithm slightly to ensure that the computation terminates even in the absence of an upper bound.

The main change required from the previous algorithm is to replace the conflict clause generator (*CCG*) with a sound and complete lazy SAT-based decision procedure for QFP. Since the conflict clauses generated from the *CCG* are simply adding more structure to ϕ_b and not required for completeness, we only required a sound decision procedure QFP for the previous algorithm.

The new algorithm is identical to the previous algorithm except for step 3 (where we do not compute D_{max}), step 6 and step 7. The new step 6 and step 7 becomes (for simplicity, we ignore the conflict clauses ϕ_c):

- **[Overapproximation]**. If ϕ_u is unsatisfiable, compute the Boolean interpolant ϕ_I of ϕ_b and $BE(\phi_{th}, D)$. We construct the formula ϕ_o which is an overapproximation of ϕ_{bt} as follows:

$$\phi_o \doteq \phi_I \wedge \phi_{th}$$

  We check the satisfiability of ϕ_o using a lazy SAT-based decision procedure for QFP. If the decision procedure returns UNSATISFIABLE, the algorithm returns UNSATISFIABLE. If the decision procedure returns SATISFIABLE, it also returns D', the maximum value of any variable in the satisfying assignment. If $\phi_I \Leftrightarrow \phi_b$, return SATISFIABLE.
- **[Repeat]**. Make $D = D'$ and go to step 4.

Lemma 2. *For any two distinct iterations i and j of the above algorithm, let ϕ_I^i and ϕ_I^j be the interpolants computed in step 6 of the algorithm in iterations i and j respectively. Then $\phi_I^i \not\Leftrightarrow \phi_I^j$.*

Proof. Let us assume $\phi_I^i \Leftrightarrow \phi_I^j$. Let us assume w.l.o.g. that $i < j$. Let D_k and D'_k be the value of D at the start of the iterations k (for $k \in \{i, j\}$). Let us also assume that both the iterations reach step 6. This means that $\phi_I^k \wedge BE(\phi_{th}, D_k)$ (for $k \in \{i, j\}$) is unsatisfiable. Moreover $\phi_I^i \wedge BE(\phi_{th}, D'_i)$ is satisfiable. Since $j > i$, $D_j \geq D'_i$. Therefore $\phi_I^j \wedge BE(\phi_{th}, D_j)$ is clearly satisfiable, and therefore ϕ_I^j can't the interpolant during the iteration j. Hence we reach a contradiction.

Theorem 2. *The algorithm described in this section that does not precompute D_{max} is sound and complete for QFP.*

Proof. Since there can only be a finite number of distinct Boolean functions over $|B|$ variables and $\phi_b \implies \phi_I$ for any iteration, Lemma 2 ensures that the algorithm above terminates. Therefore, by Theorem 1, the modified algorithm is sound and complete.

In the above algorithm, we have assumed that the decision procedure for QFP used in the overapproximation step (step 6) can construct a satisfying assignment (and therefore provide D'). We can relax this restriction and only increase D by a fixed amount δ as before and still obtain termination. This is because, using the same idea as Lemma 2, we can ensure that if ϕ_I^i is the interpolant at step i, and $j > i + (D'_i - D_i)/\delta$, then ϕ_I^j is distinct from ϕ_I^i. Although this does not allow us to bound the number of iterations (without refering D_{max}), we can ensure that in the limit some interpolant ϕ_I is going to be identical to ϕ_b for some finite iteration.