SMT Techniques for Fast Predicate Abstraction

Shuvendu K. Lahiri[⋆], Robert Nieuwenhuis[⋆⋆], and Albert Oliveras[⋆⋆]

Abstract. Predicate abstraction is a technique for automatically extracting finite-state abstractions for systems with potentially infinite state space. The fundamental operation in predicate abstraction is to compute the best approximation of a Boolean formula φ over a set of *predicates* P. In this work, we demonstrate the use for this operation of a decision procedure based on the DPLL(T) framework for SAT Modulo Theories (SMT). The new algorithm is based on a careful generation of the set of all satisfying assignments over a set of predicates. It consistently outperforms previous methods by a factor of at least 20, on a diverse set of hardware and software verification benchmarks. We report detailed analysis of the results and the impact of a number of variations of the techniques. We also propose and evaluate a scheme for incremental refinement of approximations for predicate abstraction in the above framework.

1 Introduction

In many industrial verification problems, typical logical formulas consist of large sets of clauses such as:

$$p \quad\vee\quad \neg q \quad\vee\quad a{=}f(b-c) \quad\vee\quad read(s,\, f(b-c)\,){=}d \quad\vee\quad a-g(c)\ \leq 7$$

containing purely propositional atoms as well as atoms over (combined) theories, such as the integers, arrays, or Equality with Uninterpreted Functions (EUF). Deciding the satisfiability of such clause sets modulo the background theories is known as the *Satisfiability Modulo Theories* (SMT) problem, and the systems for doing so are called *SMT solvers*. Currently, SMT is a very active area of research, and efficient SMT solvers exist that can handle (combinations of) many such theories (see also the SMT problem library [TR05] and the SMT Competition [BdMS05]). One particular SMT solver used in this paper is the Barcelogic-Tools implementation of the DPLL(T) approach to SMT [GHN$^+$04,NO05a]. It consists of a Davis-Putnam-Loveland-Logemann-based DPLL(X) engine, whose parameter X can be instantiated with a specialized solver $Solver_T$ for the given (possibly combined) T under consideration, thus producing a DPLL(T) system.

[⋆] Microsoft Research, `research.microsoft.com/~shuvendu`.

[⋆⋆] Technical Univ. of Catalonia, Barcelona, `www.lsi.upc.es/~roberto|~oliveras`.
Partially supported by Spanish Min. of Educ. and Science through the LogicTools project (TIN2004-03382, both), FPU grant AP2002-3533 (Oliveras) and personal research grant "SMT Solvers for High-Level Hardware Verification" from Intel Corporation (Nieuwenhuis).

Predicate abstraction [GS97] (an instance of the more general theory of *abstract interpretation* [CC77]) is a technique for constructing finite-state abstractions from large or infinite-state systems. The resulting finite-state abstraction can be analyzed efficiently using Boolean techniques. Predicate abstraction has been applied successfully in various verification tools to analyze software [BMMR01,HJMS02,CCG$^+$03,FQ02], hardware [CKSY04] and high-level protocols [DDP99,LBC03].

Predicate abstraction involves approximating a concrete transition system or a set of concrete states using a set P of formulas, also called *predicates.* The predicates usually denote properties of the state and are expressed as formulas, modulo some background theory, over the state variables. The abstraction is defined by the value of these predicates in any concrete state of the system. The fundamental operation in predicate abstraction can be summarized as follows:

> Given a formula φ and a set of predicates P in a theory T, generate the most precise approximation of φ using P.

Depending on the nature of the problem domain, one may either want to generate (i) the best underapproximation of φ, i.e., the *weakest* Boolean combination of P that implies φ (denoted by $\mathcal{F}_P(\varphi)$) or (ii) the best overapproximation of φ, i.e., the *strongest* Boolean combination of P that is implied by φ (denoted by $\mathcal{G}_P(\varphi)$). Here, the notions of weakness, strength and implication are with respect to entailment in the given theory T. These operations are dual of each other — $\mathcal{G}_P(\varphi)$ is the same as $\neg\mathcal{F}_P(\neg\varphi)$, and therefore it suffices to provide a procedure to compute only one of them.

Example 1. Let T be the theory of the integers, and let φ be $x < y-2 \ \vee \ x > y$. Furthermore, let P be $\{p_1, p_2, p_3\}$ where p_1, p_2 and p_3 are $x < 0$ and $y = 2$ and $x \neq 4$, respectively.

W.l.o.g., we can express $\mathcal{F}_P(\varphi)$ as a DNF, i.e., as a disjunction of cubes. Here $\mathcal{F}_P(\varphi)$ is $p_1 p_2 \ \vee \ p_2 \overline{p_3}$. Clearly, both its cubes T-entail φ and hence their disjunction does too. Moreover it is as weak (modulo T) as possible: all other cubes that T-entail φ either contain one of these two or are T-inconsistent.

The need for efficient predicate abstraction has motivated a significant amount of work during the last years. For example, Clarke et al. [CKSY04], and Lahiri and Bryant [LBC03,LB04] perform predicate abstraction by Boolean quantifier elimination using SAT solvers for propositional and first-order logic respectively. The idea of using SMT solvers for predicate abstraction has also been explored repeatedly [DDP99,SS99,FQ02,BCLZ04], but differently from what we do here, in particular, concerning incrementality. The recent *symbolic decision procedure* approach of [LBC05] is a specialized method for predicate abstraction based on saturating a set of predicates; however, it imposes restrictions on the underlying theories, it requires an expensive transformation of the queries to a logically equivalent conjunctive normal form, and combination methods for non-convex theories would need to be devised for it.

In this paper, we show how to adapt SMT solvers to compute predicate abstraction efficiently. The key idea of the procedure is to use the SMT solver to enumerate all T-models over P of φ (or, sometimes, $\neg\varphi$). For this purpose, we have turned our DPLL(X) engine into an *AllSAT* engine, i.e., an engine that can enumerate all models. Several ways of doing so are discussed and compared experimentally in this paper, including some AllSAT techniques that are also useful for DPLL-based propositional SAT solvers. With no additional work one can now obtain an efficient tool for predicate abstraction modulo a theory T by simply instantiating our adapted DPLL(X) engine with the corresponding Solver_T as it is used for DPLL(T).

The key difference of our work with previous SMT solver-based predicate abstraction techniques is in the amount of incrementality achieved in enumerating all the solutions. We show that the incrementality, aggressive theory propagation and efficient conflict analysis present in the DPLL(T) framework are crucial for obtaining an efficient predicate abstraction engine.

In particular, according to our extensive experimental results on large sets of benchmarks from three completely different sources, using the BarcelogicTools SMT solver, we always obtain a speedup factor of at least 20 with respect to the method that was previously best on that benchmark family. This scheme of using SMT solvers for predicate abstraction is attractive because it allows us to leverage the advances in the development of SMT decision procedures for obtaining more efficient predicate abstraction procedures.

We also show how we can adapt the DPLL(T)-based *AllSAT* engine to compute a series of increasingly precise approximations of $\mathcal{G}_P(\varphi)$, where for some $k < n$, approximations only use cubes of size k. Given a fixed set of predicates P and a query φ, this allows a client of predicate abstraction to first explore coarser approximations (that can be generated fast) that might suffice for proving a desired property. In fact, our experiments reveal that (i) for small cube sizes, the computation times are extremely small, and that (ii) computing the full $\mathcal{G}_P(\varphi)$ in successive steps slightly increasing the cube size can be done almost as efficiently as computing it directly, if each step is done incrementally from the previous one. Although several approaches have been developed in recent years to compute coarser approximations [GS97,BMMR01,DD01], the process of refining the approximations is not incremental, and can sometimes be the main bottleneck in the verification [BCDR04].

The rest of the paper is structured as follows. We first give some background and definitions about SMT and DPLL(T) in Section 2. Section 3 is on the encoding in SMT of the under and upper approximation problems. Its Subsections 3.1 and 3.2 discuss the different ways of forcing the enumeration of all cubes over P and the variations with increasing cube sizes. Section 4 analyzes a large number of experiments on problems from three completely different applications. Finally, Section 5 lists future work and conclusions.

2 Background

2.1 Formal preliminaries

A *theory* T is a set of closed first-order formulas. A formula φ is *T-satisfiable* or *T-consistent* if $\varphi \wedge T$ is satisfiable in the first-order sense. Otherwise, it is called *T-unsatisfiable* or *T-inconsistent.* In this paper we will deal with (partial) assignments M, where M is a set (conjunction) of ground literals. If M is a T-consistent partial assignment and φ is a ground formula such that M is a model of φ in the propositional sense, then we say that *M is a T-model of φ*. The SMT problem for a theory T is the problem of determining, given a formula φ, whether φ is T-satisfiable, or, equivalently, whether φ has a T-model. As usual in SMT, here we only consider the SMT problem for *ground* (and hence quantifier-free) CNF formulas φ. If φ and ψ are formulas, then φ *T-entails* ψ, written $\varphi \models_T \psi$, if $\varphi \wedge \neg\psi$ is T-inconsistent. A *theory lemma* is a clause C such that $\emptyset \models_T C$.

2.2 The DPLL(T) approach to SMT

The so-called *lazy* approach to SMT, in its simplest form, initially considers each atom occurring in the input formula F simply as a propositional symbol, i.e., it "forgets" about the theory T. Then it sends the formula to a SAT solver. If the SAT solver reports propositional unsatisfiability, then F is also T-unsatisfiable. If it returns a propositional model of F, then this assignment is checked by a specialized T-solver that can only deal with conjunctions of literals. If the model is found T-consistent then it is a T-model of F. Otherwise, the T-solver builds a ground clause that is a logical consequence of T, i.e., a theory lemma, precluding that assignment. This lemma is added to F and the SAT solver is started again. This process is repeated until the SAT solver finds a T-model or returns unsatisfiable.

DPLL-based refinements of the lazy approach use *incremental T-solvers* that check the T-inconsistency of the partial models while they are being built. Moreover, the DPLL-based SAT solver is usually *on-line*: upon each T-inconsistent assignment it can then backjump to some point where the assignment was still T-consistent, instead of restarting the search from scratch.

DPLL(T) is such a new modular lazy-like approach for SMT. It is based on a general DPLL(X) engine, whose parameter X can be instantiated with a specialized Solver_T for conjunctions of (ground) atoms, thus producing a system DPLL(T). Once the DPLL(X) engine has been implemented, this approach becomes very flexible: a DPLL(T) system for a theory T is obtained by simply plugging in the corresponding Solver_T. In DPLL(T), a special attention is devoted to *theory propagation*, a refinement that can have a crucial impact on performance. The idea is that the T-solver tells the DPLL(X) engine which literals can be set to true because they are T-consequences of the current partial assignment. For example, if T is the theory of equality and the current assignment contains the literals $b{=}d$, $f(b){=}d$ and $f(d){=}a$, then the T-solver may report $a{=}b$ as a T-consequence instead of letting DPLL(X) guess a truth value for it.

3 Predicate abstraction using SMT

For us, a *predicate* will be any ground formula. If P is a set of finite predicates, a *cube* over P is a conjunction $p_1 \wedge \ldots \wedge p_k \wedge \neg p'_1 \wedge \ldots \wedge \neg p'_{k'}$, where all p_i and p'_j are distinct predicates of P and $k + k'$ is the *size* of the cube. A *minterm* over P is a cube of size $|P|$. A *Boolean formula over* P is either a predicate of P or a conjunction, disjunction or negation of Boolean formulas over P.

Given a theory T, a set of predicates $P = \{p_1, \ldots, p_n\}$ and a ground formula φ, the central operation in predicate abstraction is to compute the weakest Boolean formula $\mathcal{F}_P(\varphi)$ over P that T-entails φ. Equally important is the dual operation, to compute the strongest Boolean formula $\mathcal{G}_P(\varphi)$ over P that is T-entailed by φ, but this operation can be reduced to the previous one, since it is not difficult to see that $\mathcal{F}_P(\varphi)$ is indeed $\neg\mathcal{G}_P(\neg\varphi)$.

$\mathcal{F}_P(\varphi)$ can be characterized as the disjunction of all the minterms over P that T-entail φ, that is:

$$\mathcal{F}_P(\varphi) \equiv \bigvee \{c \mid c \text{ is a minterm over } P \text{ and } c \models_T \varphi\}$$

Since each minterm T-entails φ, this formula clearly T-entails φ as well. Any other formula over P that T-entails φ, when expressed as a disjunction of minterms, consists only of minterms T-entailing φ, all of which belong to $\mathcal{F}_P(\varphi)$. Hence, $\mathcal{F}_P(\varphi)$ is also the weakest such formula. Now we can also easily characterize $\mathcal{G}_P(\varphi)$ by using its relation to $\mathcal{F}_P(\neg\varphi)$:

$$\begin{aligned}
\mathcal{G}_P(\varphi) &\equiv \neg\mathcal{F}_P(\neg\varphi)\\
&\equiv \neg\left(\bigvee\{c \mid c \text{ is a minterm over } P \text{ and } c \models_T \neg\varphi\}\right)\\
&\equiv \bigvee\{c \mid c \text{ is a minterm over } P \text{ and } c \not\models_T \neg\varphi\}\\
&\equiv \bigvee\{c \mid c \text{ is a minterm over } P \text{ and } c \wedge \varphi \text{ is } T\text{-satisfiable}\}
\end{aligned}$$

i.e., computing $\mathcal{G}_P(\varphi)$ amounts to enumerating all minterms over P that are T-satisfiable when conjoined with φ.

As in [LBC03,CKSY04], we can introduce a set B of n fresh propositional variables $\{b_1, \ldots b_n\}$ and consider the formula $\varphi \wedge \bigwedge_{i=1}^{n} b_i \Leftrightarrow p_i$. Given a T-model M of this formula, we collect the conjunction of all B-literals that are true in M, and replace each b_i by its corresponding formula p_i. The resulting minterm c, called the *projection of M onto P*, is over P and $c \wedge \varphi$ is T-satisfiable.

For enumerating *all* such c, in principle, *every* off-the-shelf SMT solver can be used, by adding, each time a T-model is found whose B-literals are $\{l_1, \ldots, l_n\}$, a *blocking clause* $\neg l_1 \vee \ldots \vee \neg l_n$ and then starting the SMT solver from scratch, and repeating this until no more T-models are found. The number of restarts is no more than the number of different minterms over P, that is, 2^n. Each model found can be stored, say, in a BDD, or in a file to be treated once the AllSAT procedure has finished. The computation of $\mathcal{F}_P(\varphi)$ can be done in a similar way.

3.1 AllSAT and AllSAT over *important* symbols

The complete black-box approach explained above is not very efficient. This is due to the restarts from scratch, where moreover the lemmas learned by the SMT solver are not re-used between restarts, and due to the worst-case exponential growth of the clause set (one additional blocking clause for each model found). We now show that these problems can be entirely overcome, without modifying the search behavior of the best DPLL implementations, i.e., using the best known conflict analysis techniques and conflict-driven backjumping, and without the need of keeping the blocking clauses or the learned lemmas.

In this section we focus on the propositional case, since all these results on AllSAT immediately extend to AllSAT in the SMT case, i.e., for enumerating all T-models using any of the modern DPLL-based SMT solvers, including DPLL(T).

For the general AllSAT problem, i.e., without considering a subset of important symbols, the idea is as follows. Each time a model $\{l_1, \ldots, l_n\}$ is found, store it and do (e.g., 1UIP) conflict-driven backjumping as if the blocking clause $\neg l_1 \vee \ldots \vee \neg l_n$ (which is conflicting in the current DPLL state) belonged to the clause set; see [ZMMM01,NO05a]. Keeping the *lemmas learned* in backjump steps (or the blocking clauses) is optional: as usual, they can be kept only as long as they are *active* pruning the search.

Property 1. This AllSAT procedure terminates and enumerates all models.

As pointed out in [JHS05], for Chaff's DPLL procedure this result easily follows from the proofs in [ZM03]. For essentially any practical DPLL strategy or variant, it follows from [NOT05], where the termination proof uses a well-founded ordering on DPLL search states, based on a lexicographic comparison of the number of literals in each *decision level*. Roughly, the intuition is that a search state is *more advanced* than another one if it has more information at lower decision levels, i.e., for some i, it has set more literals at decision level i, and it has the same number of literals at all decision levels lower than i.

An important variant of the AllSAT problem is, given a subset P of *distinguished* or *important* symbols (in this paper, the predicates), to enumerate all (sub)models over P that can be extended to total models over all symbols. This also has important applications to, e.g., model checking [GSY04]. For this, the same procedure applies by removing from the blocking clause the non-important literals, and the same correctness proof based on [NOT05] holds.

Property 2. This AllSAT procedure with distinguished symbols P terminates and enumerates all models over P that can be extended to total models over all symbols.

We have not found this observation elsewhere in the literature. E.g., [GSY04] does chronological backtracking on the important literals, and forces the decision heuristic to split on important literals first.

Quite surprisingly, if, as proposed here, not all blocking clauses or lemmas are kept, some model may be found more than once. However, according to our experiments, this phenomenon appears to be rare and has a very low impact on performance, see Section 4.

3.2 Incrementally refining the approximation

The approaches for predicate abstraction of [DDP99,SS99,FQ02] aim at explicitly asking an SMT solver for each cube c whether $\varphi \wedge c$ is T-satisfiable or not. For reducing the number of calls to the SMT solver, they work by starting with smaller cube sizes. For example, if P is $\{p_1, \ldots, p_n\}$, one can first send $\varphi \wedge p_1$ to the SMT solver, and only if this is T-satisfiable, try with $\varphi \wedge p_1 \wedge p_2$ and with $\varphi \wedge p_1 \wedge \neg p_2$, and so on. But this still leads to a large exponential number of independent calls to the SMT solver, without much incrementality to be exploited.

In this subsection we also work with increasing cube sizes, but in a completely different way and with a completely different purpose. Rather than directly computing the strongest overapproximation $\mathcal{G}_P(\varphi)$, we want to compute a sequence $\mathcal{G}_P^{k_1}(\varphi), \mathcal{G}_P^{k_2}(\varphi), \ldots, \mathcal{G}_P^{k_m}(\varphi)$ of successively stronger (i.e., each one T-entails the previous ones) overapproximations over P of φ, where the last one is $\mathcal{G}_P(\varphi)$. Here we will compute each $\mathcal{G}_P^{k}(\varphi)$ by collecting cubes of size k, and the sizes will be such that $k_1 < \ldots < k_m = |P|$.

The motivation for doing this is that for certain applications, some of the first few $\mathcal{G}_P^{k_i}(\varphi)$'s may already suffice. Moreover, our experiments reveal that for small k, computing $\mathcal{G}_P^{k}(\varphi)$ is very fast. In addition, computing the whole sequence with small increments of k can be done almost as efficiently as computing $\mathcal{G}_P(\varphi)$, if each step is done incrementally from the previous one.

In the following, let *restr* be any function such that, given a minterm c over P and an integer k with $k \leq |P|$, $restr(c, k)$ returns a subcube of size k of c.

Theorem 1. *For every $k_1 < \ldots < k_m = |P|$, the sequence $\mathcal{G}_P^{k_1}(\varphi), \ldots, \mathcal{G}_P^{k_m}(\varphi)$ is such that*

- *$\mathcal{G}_P^{k_i}(\varphi)$ is T-entailed by φ for all i in $\{1, \ldots, m\}$,*
- *$\mathcal{G}_P^{k_{i+1}}(\varphi) \models_T \mathcal{G}_P^{k_i}(\varphi)$ for all i in $\{1, \ldots, m-1\}$, and*
- *$\mathcal{G}_P^{k_m}(\varphi)$ is $\mathcal{G}_P(\varphi)$.*

if, for all i in $\{1, \ldots, m\}$, the following two conditions hold:

1. *$\mathcal{G}_P^{k_i}(\varphi) \equiv \bigvee\{restr(c, k_i) \mid c$ is a minterm over P and $c \wedge \varphi$ is T-satisfiable$\}$*
2. *For each minterm c over P with $c \wedge \varphi$ T-satisfiable there exists a minterm c' over P with $c' \wedge \varphi$ T-satisfiable such that $restr(c, k_i) \supset restr(c', k_{i-1})$.*

Proof. (sketch) Each $\mathcal{G}_P^{k_i}(\varphi)$ is T-entailed by φ, since the disjunction of all minterms c over P with T-satisfiable $c \wedge \varphi$ is T-entailed by φ, and for each one of these c there is some subcube in $\mathcal{G}_P^{k_i}(\varphi)$. The increasing strength follows in a similar way from the second condition, since each disjunct of $\mathcal{G}_P^{k_i}(\varphi)$ contains a disjunct of $\mathcal{G}_P^{k_{i-1}}(\varphi)$. Finally, $\mathcal{G}_P^{k_m}(\varphi)$ is $\mathcal{G}_P(\varphi)$ if k_m is $|P|$ due to the characterization of $\mathcal{G}_P(\varphi)$ given at the beginning of this section. $\square$

This theorem gives us a way to use our algorithm of the previous subsections for computing the successive $\mathcal{G}_P^{k_i}(\varphi)$'s. A difference is that, since here we collect cubes c_k of size k instead of the whole minterms, we can, after c_k has been collected, do the conflict analysis with the corresponding blocking clause $\neg c_k$ of size k. Note that this may preclude some minterms c extending c_k from later consideration, but we are still safe since for these c we can assume $restr(c, k)$ to be c_k. Again, as in Properties 1 and 2, termination follows from [NOT05].

In this algorithm, one could use a function $restr$ such that $restr(c, k)$ always returns the subset of literals of c over $\{p_1, \ldots, p_k\}$, i.e., the first k elements of P. However, having in each $\mathcal{G}_P^k(\varphi)$ only predicates of $\{p_1, \ldots, p_k\}$ may not be very useful. This is because in most cases no subset of P will suffice to construct strong invariants of interest. We believe that it is much more useful to have most predicates of P appear in some of the k-size cubes. Therefore, in our implementation we use a random $restr$ function. The second condition of Theorem 1 can be enforced as follows if one remembers the previous $\mathcal{G}_P^{k_{i-1}}(\varphi)$: for each minterm c considered in the computation of $\mathcal{G}_P^{k_i}(\varphi)$, we know that a subcube of c of size k_{i-1} must belong to $\mathcal{G}_P^{k_{i-1}}(\varphi)$. This subcube was added to $\mathcal{G}_P^{k_{i-1}}(\varphi)$ as the restriction $restr(c', k_{i-1})$ for some[1] minterm c'. The only thing we have to impose is that $restr(c, k_i)$ includes $restr(c', k_{i-1})$. This is all clearer in the example below:

Example 2. Let φ be the formula $x < y - 2 \quad \vee \quad x > y$ and P be $\{p_1, p_2, p_3\}$ where p_1 is $x < 0$, p_2 is $y{=}2$ and p_3 is $x{=}4$. We will construct the sequence of approximations $\mathcal{G}_P^1(\varphi), \mathcal{G}_P^2(\varphi)$, $\mathcal{G}_P^3(\varphi)$. For a better understanding of the algorithm, let us present the set of all minterms c such that $c \wedge \varphi$ is T-satisfiable: $\{p_1p_2\overline{p}_3,\ p_1\overline{p}_2\overline{p}_3,\ \overline{p}_1p_2p_3,\ \overline{p}_1p_2\overline{p}_3,\ \overline{p}_1\overline{p}_2p_3,\ \overline{p}_1\overline{p}_2\overline{p}_3\}$.

For the computation of $\mathcal{G}_P^1(\varphi)$, the AllSAT procedure first finds the minterm $p_1p_2\overline{p}_3$ and restricts it to $\overline{p}_3$. After adding the blocking clause p_3, the minterm $\overline{p}_1p_2p_3$ is found and restricted to $\overline{p}_1$. Then, p_1 is added as a blocking clause and since there are no more minterms to be found we finish with $\mathcal{G}_P^1(\varphi) \equiv \overline{p}_3 \vee \overline{p}_1$.

For $\mathcal{G}_P^2(\varphi)$ we start with the minterms already computed in the previous step. We can restrict $p_1p_2\overline{p}_3$ to $p_2\overline{p}_3$ (note that, due to condition 2 of Theorem 1, p_1p_2 would not have been a correct restriction), and similarly restrict $\overline{p}_1p_2p_3$ to $\overline{p}_1p_2$. After adding the blocking clauses $\overline{p}_2 \vee p_3$ and $p_1 \vee \overline{p}_2$, the AllSAT procedure starts the search. First, it finds the minterm $p_1\overline{p}_2\overline{p}_3$, and restricts it to $p_1\overline{p}_3$ (again due to condition 2 of Theorem 1, $p_1\overline{p}_2$ would not have been a correct choice). Then, after the blocking clause $\overline{p}_1 \vee p_3$ is added, $\overline{p}_1\overline{p}_2p_3$ is found and restricted to $\overline{p}_1\overline{p}_2$. Since the blocking clauses preclude any other possible minterm, $\mathcal{G}_P^2(\varphi)$ is $p_2\overline{p}_3 \vee \overline{p}_1p_2 \vee p_1\overline{p}_3 \vee \overline{p}_1\overline{p}_2$.

Finally, for $\mathcal{G}_P^3(\varphi)$ we start with the four minterms already computed and then the AllSAT procedure will compute the two missing ones, namely $\overline{p}_1p_2\overline{p}_3$ and $\overline{p}_1\overline{p}_2\overline{p}_3$. $\square$

The interesting aspect hereby is that, at each incremental step, we reuse from previous step(s):

[1] Note that we cannot assume c' to be c because, due to the use of blocking clauses, c might not have been considered in the computation of $\mathcal{G}_P^{k_{i-1}}(\varphi)$.

1. all lemmas learned by DPLL(T) that are T-consequences of φ, which helps to speed up the search.
2. all minterms c already computed.

We finish this section with three remarks about the quality of these approximations. First, let us note that the strongest disjunction of cubes over P of size k that is T-entailed by φ does not always exist. Second, in CNF it does exist: the strongest conjunction of clauses over P of size k that is T-entailed by φ is a well-defined concept. Third, let us remark that there are formulas for which our algorithm would compute a stronger approximation than this CNF and viceversa.

4 Experimental evaluation

4.1 Benchmarks and their source

The set of benchmarks for evaluating our technique has been generated from three completely different verification tasks:

1. **SLAM**: This category contains a set of 665 predicate abstraction queries generated from Windows device driver verification in SLAM [BMMR01]. In SLAM, predicate abstraction is used to abstract a Boolean program from a C program. This set has been previously used to evaluate the predicate abstraction technique in [LBC05].
2. **UCLID Suite**: This category contains $\mathcal{G}_P(\varphi)$ queries generated during the verification of high-level description of microprocessors, cache-coherence protocols and other distributed algorithms [LB04]. Each benchmark in this category contains around 6 to 19 predicate abstraction queries denoting the different image computation steps.
3. **Recursive Data Structures (RDS)**: This is a set of benchmarks generated from the verification of programs manipulating linked lists inside UCLID [LQ06]. Each benchmark contains a set of $\mathcal{G}_P(\varphi)$ queries for different abstract image computation steps.

The theories used in all the three categories are combinations of EUF and difference logic (constraints of the form $x \leq y + c$). For the latter two classes of benchmarks for UCLID and RDS, more complex theories are axiomatized using quantifiers. However, these quantifiers are eliminated upfront using simple (but sufficient to prove the properties in the examples) quantifier instantiation within UCLID, to generate a quantifier-free predicate abstraction query.

4.2 Results and Analysis

We have implemented the procedure described in Section 3.1 on top of the BarcelogicTools implementation of the DPLL(T) approach for SMT. Each minterm was stored in a BDD immediately after finding it. For this, the CUDD [CUD]

BDD package was used. The result was read from the BDD as a disjunction of prime implicants. This significantly reduced the number of disjuncts (see the table below) especially when many minterms were stored. For each of the benchmarks described above, our resulting system was compared with the best existing competitor by running experiments on a 2GHz 512 MB Pentium 4.

For the SLAM benchmarks we compared our system with the symbolic decision procedure approach of [LBC05]. This set of benchmarks posed little difficulty to our approach: the whole set of benchmarks (655 queries) was processed in less than 5 seconds, whereas the symbolic decision procedure implementation took 273 seconds. This is a first indication that our approach is superior, but, since the running time for each single query is negligible, we will concentrate on analyzing the results obtained from the other two families.

For the UCLID and Recursive Data Structures benchmarks, the table below lists the number of queries for each family, predicates each query consists of, and the total number of minterms of the $\mathcal{G}_P(\varphi)$'s to be computed. Finally, for UCLID [LBC03] we give the aggregated running time in seconds and for our BarcelogicTools implementation, we give the running time, the speedup factor w.r.t. UCLID, and the number of cubes in the answers. In order to be more confident about the results, we used CUDD to check whether the output of our tool was equivalent to UCLID's output.

Benchmark family	**#queries**	**#prds.**	**#minterms**	**UCLID**	**BCLT**		
				time	**time**	**speedup**	**#cubes**
UCLID Suite:							
aodv	7	21	2916	657	4.6	143x	458
bakery	19	32	426	245	11	22x	294
BRP	10	22	30	3.5	0.1	35x	24
cache_ibm	10	16	326	34	1.3	26x	123
cache_bounded	18	26	2238	1119	23	49x	1022
DLX	6	23	38080	335	13	26x	2704
OOO	10	25	10728	921	36	26x	242
Rec. Data Struct.:							
reverse_acyclic	7	16	91	20	0.6	33x	44
set_union	6	24	334	22	0.7	31x	60
simple_cyclic	5	15	110	3.7	0.11	34x	20
sorted_int	10	21	2465	765	19	40x	250

Independently of the benchmark, BCLT is always at least 20 times faster. Hence, it is also very robust, i.e. it is not tailored towards any specific type of benchmark.

As mentioned in Section 3.1, a possible drawback of our AllSAT approach is that the same minterm can be listed more than once. Therefore, we also tried a mixed approach where, in order to preclude most repeated minterms, blocking clauses were kept while they were active (as it is done with learned lemmas). This did not produce any observable improvement. This is probably due to the fact that even for the `DLX` and `OOO` families, where the number of minterms to be enumerated is significant, the number of repeated minterms did not account for more than 3 percent of the total.

Profiling shows that the BDD operations (including the computation of prime implicants) take negligible runtime compared to the rest of the procedure. A typical distribution of the running time for these benchmarks is to spend 50 percent of the total time in the Boolean reasoning part, 30 percent in the theory reasoning and the rest mainly on the branching heuristic.

4.3 Results on alternative settings and analysis

In order to understand the reasons behind the performance of our Barcelogic-Tools implementation (called *good* in the table below), we also ran it with three different settings.

One of them, (*black-box* in the table) was to use the black-box approach explained at the beginning of Section 3, where each time a T-model is found a blocking clause is added and the search is restarted from scratch, with no possibility to reuse the lemmas already computed. This setting was modified by allowing the lemmas to be reused (*naive* in the table). Finally, in order to analyze the role of theory propagation [NO05b], we also ran BCLT with the more advanced enumeration algorithm but with theory propagation turned off (*noTP* in the table).

Benchmark family	**#queries**	**#preds.**	**#minterms**	**BCLT**			
				good	**black-box**	**naive**	**no TP**
UCLID Suite:							
aodv	7	21	2916	4.6	24	11	11
bakery	19	32	426	11	19	13	14
BRP	10	22	30	0.1	0.12	0.13	0.2
cache_ibm	10	16	326	1.3	2.3	2	2.5
cache_bounded	18	26	2238	23	63	31	32
DLX	6	23	38080	13	242	63	15
OOO	10	25	10728	36	176	57	615
Rec. Data Struct.:							
reverse_acyclic	7	16	91	0.6	0.7	0.7	1
set_union	6	24	334	0.7	1	0.8	1
simple_cyclic	5	15	110	0.11	0.16	0.13	0.2
sorted_int	10	21	2465	19	38	24	154

Clearly, the black-box approach is not very competitive. It significantly improves if we allow the reusability of lemmas among the enumeration of models (*naive*). In fact, it is not much slower than *good*: since the number of minterms is not too large, the useless restarts are not too frequent, and also the explosion in the formula size does not show up in its full extent. Only in the families with many minterms (`DLX` and `OOO`), one starts noticing the benefits of a better AllSAT algorithm.

Concerning the role of theory propagation in these benchmarks, we can see that in some families (`OOO` and `sorted_int`) it is crucial for the success of the method. This is interesting because these two families are the ones which use arithmetic symbols most heavily among all. Moreover, applying theory propagation never increases the runtime, because the overhead in time it produces is always compensated by a reduction in the search space. This confirms, in another application area, the results presented in [NO05b] with respect to the importance of theory propagation.

4.4 Results on the incremental refinements of the approximation

The procedure presented in Section 3.2 has also been implemented in order to evaluate its feasibility. The table below includes, in its third column, the time (in seconds) needed to directly compute $\mathcal{G}_P(\varphi)$, as explained in Section 3.1. The other columns are in groups of two, using different increment steps, comparing the incremental version of the procedure (`incr` in the table), with a non-incremental version (`n-incr`, not reusing lemmas nor minterms from previous steps). For example, the column `incr` below `step 2` contains the time needed to compute, with the incremental algorithm, the whole set of approximations $\mathcal{G}_P^2(\varphi), \mathcal{G}_P^4(\varphi), \mathcal{G}_P^6(\varphi), \ldots$, until finally computing the exact $\mathcal{G}_P(\varphi)$.

Benchmark			**step 1**		**step 2**		**step 5**	
family	**#preds.**	**exact**	**incr.**	**n-incr.**	**incr.**	**n-incr.**	**incr.**	**n-incr.**
UCLID Suite:								
aodv	21	4.6	15	47	10	24	7.2	13
bakery	32	11	28	159	21	86	16	40
BRP	22	0.1	1.1	1.7	0.6	1	0.3	0.5
cache_ibm	16	1.3	3	8.6	2.2	5.1	1.7	2.8
cache_bounded	26	23	71	333	51	185	40	88
DLX	23	13	37	84	26	42	18	21
OOO	25	36	67	368	50	193	43	102
Rec. Data Struct.:								
reverse_acyclic	16	0.6	1.1	2.4	0.9	1.5	0.7	1
set_union	24	0.7	1.7	4.8	1.2	2.7	0.9	1.6
simple_cyclic	15	0.11	0.4	0.7	0.3	0.4	0.2	0.3
sorted_int	21	19	25	113	20	63	19	36

The first important thing to note is that even when we use an increment of 1, which means that more than 20 approximations are computed on average, the time needed in the incremental version only increases by a factor of 2 or 3 with respect to the time required to directly compute $\mathcal{G}_P(\varphi)$. One can also notice that this factor is reduced when we use a bigger increment step. This shows that in a situation where one wants to use some of these approximations but they do not suffice, the time needed to compute $\mathcal{G}_P(\varphi)$ will not be much worse than if we had tried to directly compute it.

The other important conclusion is that it is essential to use an incremental algorithm, e.g., using an increment step of 1 and a non-incremental algorithm requires about 10 times as much time as directly computing $\mathcal{G}_P(\varphi)$.

5 Conclusions and further work

In this paper, we have demonstrated the use of an SMT solver based on the DPLL(T) framework for efficient predicate abstraction. The algorithm is based on a careful generation of the set of all satisfying assignments over a set of predicates, and we have illustrated the impact of the various factors such as theory propagation, backjumping and incrementality on this approach. We also show how the technique can be adapted to compute increasingly precise approximations with respect to a given set of predicates in an incremental fashion, which provides an alternate method for refining predicate abstractions with a fixed set of predicates [DD01,JM05].

We are currently investigating exploiting incrementality when computing an abstraction over an monotonically growing set of predicates, which can be useful for creating Boolean programs [BMMR01] incrementally. Another area of future work is to extend a minterm c over P to a larger cube *on-the-fly*, before starting the search for a new minterm — this could impact the performance of queries (e.g. `OOO`, `DLX` and `sorted_int`) that have a very large #minterms/#cubes ratio.

References

[BCDR04] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining Approximations in Software Predicate Abstraction. In *TACAS'04*, LNCS 2988, pages 388–403.

[BCLZ04] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *CAV'04*, LNCS 3114, pages 457–461. Springer, 2004.

[BdMS05] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In *CAV'05*, LNCS 3576, pages 20–23. Springer, 2005.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'01*, *SIGPLAN Notices,* 36(5), May 2001.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : A Unified Lattice Model for the Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*. ACM Press, 1977.

[CCG+03] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In *ICSE'03)*, pages 385–395. IEEE Computer Society 2003, May 2003.

[CKSY04] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *FMSD'04*, 25, 2004.

[CUD] CUDD: CU Decision Diagram Package. Available at `http://vlsi.colorado.edu/fabio/CUDD/cuddIntro.html`.

[DD01] S. Das and D. Dill. Successive approximation of abstract transition relations. In *LICS'01*, pages 51–60. IEEE Computer Society, June 2001.

[DDP99] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV'99*, LNCS 1633, pages 160–171. Springer, 1999.

[FQ02] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL'02*, pages 191–202. ACM, 2002.

[GSY04] O. Grumberg, A. Schuster and A. Yadgar. Memory efficient all-solutions sat solver and its application for reachability analysis. In *FMCAD'04*, LNCS 3312, pages 275–289. Springer, 2004.

[GHN+04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *CAV'04*, LNCS 3114, pages 175–188. Springer, 2004.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV'97*, LNCS 1254, pages 72–83. Springer, 1997.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL'02*, pages 58–70. ACM Press, 2002.

[JHS05] H. Jin, H. Han, and F. Somenzi. Efficient conflict analysis for finding all satisfying assignments of a boolean circuit. In *TACAS'05*, LNCS 3440, pages 287–300. Springer, 2005.

[JM05] R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In *CAV'05*, LNCS 3576, pages 39–51. Springer, 2005.

[LB04] S. K. Lahiri and R. E. Bryant. Constructing Quantified Invariants via Predicate Abstraction. In *VMCAI'04*, LNCS 2937, pages 267–281, 2004.

[LBC03] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *CAV'03*, LNCS 2725, pages 141–153. Springer, 2003.

[LBC05] S. K. Lahiri, T. Ball, and B. Cook. Predicate abstraction via symbolic decision procedures. In*CAV'05*, LNCS 3576, pages 24–38. Springer, 2005.

[LQ06] S. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL'06*, pages 115–126. ACM, 2006.

[NO05a] R. Nieuwenhuis and A. Oliveras. Decision procedures for SAT, SAT Modulo Theories and Beyond. The BarcelogicTools. (Invited Paper). In *LPAR'05*, LNCS 3835, pages 23–46. Springer, 2005.

[NO05b] R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In *CAV'05* LNCS 3576, pages 321–334. Springer, 2005.

[NOT05] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories. *LPAR'04*, LNCS 3452, pp. 36–50. Springer, 2005.

[SS99] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *CAV'99*, LNCS 1633, pages 443–454. Springer, 1999.

[TR05] C. Tinelli and S. Ranise. SMT-LIB: The Satisfiability Modulo Theories Library, July 2005. http://goedel.cs.uiowa.edu/smtlib/.

[ZM03] L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker In *DATE 2003*, pages 10880–10885. IEEE Computer Society, 2003.

[ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *ICCAD'01*, pages 279–285, 2001.