

PREDICATE ABSTRACTION VIA SYMBOLIC DECISION PROCEDURES*

SHUVENDU K. LAHIRI^a, THOMAS BALL^a, AND BYRON COOK^b

^a Microsoft Research, Redmond, WA 98052
e-mail address: {shuvendu,tball}@microsoft.com

^b Microsoft Research, Cambridge, United Kingdom
e-mail address: bycook@microsoft.com

ABSTRACT. We present a new approach for performing predicate abstraction based on symbolic decision procedures. Intuitively, a symbolic decision procedure for a theory takes a set of predicates in the theory and symbolically executes a decision procedure on all the subsets over the set of predicates. The result of the symbolic decision procedure is a shared expression (represented by a directed acyclic graph) that implicitly represents the answer to a predicate abstraction query.

We present symbolic decision procedures for the logic of Equality and Uninterpreted Functions (EUF) and Difference logic (DIFF) and show that these procedures run in pseudo-polynomial (rather than exponential) time. We then provide a method to construct symbolic decision procedures for simple mixed theories (including the two theories mentioned above) using an extension of the Nelson-Oppen combination method. We present preliminary evaluation of our Procedure on predicate abstraction benchmarks from device driver verification in SLAM.

1. INTRODUCTION

Predicate abstraction is a technique for automatically creating finite abstract models of finite and infinite state systems [GS97]. The method has been widely used in abstracting finite-state models of programs in SLAM [BMMR01] and numerous other software verification projects [HJMS02, CCG⁺04]. It has also been used for synthesizing loop invariants [FQ02] and verifying distributed protocols [DDP99, LBC03].

The fundamental operation in predicate abstraction can be summarized as follows: Given a set of predicates P describing some set of properties of the system state, and a formula e , compute the weakest Boolean formula $\mathcal{F}_P(e)$ over the predicates P that implies

2000 ACM Subject Classification: F.3.1, F.4.1.

Key words and phrases: predicate abstraction, decision procedures, formal verification, symbolic algorithms.

* An earlier version of the paper appeared in the proceedings of the Computer Aided Verification conference, 2005.

e ¹. Most implementations of predicate abstraction [GS97, BMMR01] construct $\mathcal{F}_P(e)$ by collecting the set of cubes (a conjunction of the predicates or their negations) over P that imply e . The implication is checked using a first-order theorem prover. This method may require making a very large ($2^{|P|}$ in the worst case) number of calls to a theorem prover and can be expensive.

We propose a new way to perform predicate abstraction based on *symbolic decision procedures*. A symbolic decision procedure for a theory T (SDP_T) takes sets of predicates G and E and symbolically executes a decision procedure for T on $G' \cup \{\neg e \mid e \in E\}$ ², for all the subsets G' of G . The output of $SDP_T(G, E)$ is a shared expression (an expression where common subexpressions can be shared) representing those subsets $G' \subseteq G$, for which $G' \cup \{\neg e \mid e \in E\}$ is unsatisfiable. We show that such a procedure can be used to compute $\mathcal{F}_P(e)$ for performing predicate abstraction.

We present symbolic decision procedures for the logic of Equality and Uninterpreted Functions (EUF) and Difference logic (DIF) and show that these procedures run in polynomial and pseudo-polynomial time respectively, and therefore produce compact shared expressions. We provide a method to construct SDP for a combination of two simple theories $T_1 \cup T_2$ (including EUF + DIF), by using an extension of the Nelson-Oppen combination [NO80] method. We use Binary Decision Diagrams (BDDs) [Bry86] to construct $\mathcal{F}_P(e)$ from the shared representations efficiently in practice.

We present a preliminary evaluation of our procedure on predicate abstraction benchmarks from device driver verification in SLAM, and show that our method outperforms existing methods for doing predicate abstraction.

The rest of the paper is organized as follows: Section 1.1 describes related work in predicate abstraction techniques. Section 2 describes the background concepts including predicate abstraction. Section 3 describes symbolic decision procedures, and instantiates it for two different theories (EUF and DIF). Section 4 describes a framework for modularly combining the SDPs for two theories that satisfy certain requirements, using an extension of the Nelson-Oppen combination method. Section 5 describes the implementation and the experimental evaluation of our technique. Finally, we present the conclusions and future work in Section 6.

1.1. Related Work. Several techniques have been suggested to improve the performance of predicate abstraction. The techniques can be broadly classified into three categories: In the first category, we classify methods that treat the decision procedures as a “black box”, and attempt to minimize the number of decision procedure calls during predicate abstraction. The second category consists of methods that use a quantifier elimination procedure to perform predicate abstraction. Finally, there are techniques that do not compute the most precise abstract directly; instead, they rely on counterexamples or proofs in the overall verification process to refine the abstraction. In the following paragraphs, we describe these techniques in more details.

The techniques that aim to reduce the number of calls to the theorem prover or decision procedure are mostly based on enumerating cubes over P in an increasing order of their size. Das et al. [DDP99] enumerates cubes over a tree, after fixing the order of predicates

¹The dual of this problem, which is to compute the strongest Boolean formula $\mathcal{G}_P(e)$ that is implied by e , can be expressed as $\neg \mathcal{F}_P(\neg e)$.

²Throughout this paper, we interpret a set of expressions to be a conjunction over the expressions in the set.

that appear in any path to the leaves. If a cube is found unsatisfiable, then all its subcubes (represented by the subtree) are pruned off. This method may require $2^{|P|+1}$ calls to the theorem prover in the worst case. Saidi and Shankar [SS99] relaxes the order on the predicates, and enumerate all possible cubes ($3^{|P|}$ of them) over the predicates. Flanagan and Qadeer [FQ02] provide an algorithm that searches over the $2^{|P|}$ clauses (disjunction of cubes over the predicates or their negations) of size $|P|$, but attempts to greedily grow the clause (by dropping literals) when such a clause is implied by the formula e . Their technique requires $|P| \cdot 2^{|P|}$ theorem prover calls in the worst case. Other techniques sacrifice precision to gain efficiency, by only considering cubes of some fixed length [BMMR01]. All these techniques may require an exponential number of theorem prover calls in the worst case, and demonstrate worst case behavior in practice. However, more importantly, since these queries are not incremental, the state of the prover has to be reset across each call, precluding any learning across calls.

Alternately, predicate abstraction can be formulated as a quantifier elimination problem. Lahiri et al. [LBC03] and Clarke et al. [CKSY04] perform predicate abstraction by reducing the problem of computing $\mathcal{F}_P(e)$ to Boolean quantifier elimination. The former method first transforms a first-order quantifier elimination problem into Boolean quantifier elimination by encoding first-order formulas into Boolean formulas; the latter assumes all variables are propositional. The method in [LBC03] first converts the quantifier-free first-order formula to a Boolean formula such that the translation preserves the set of satisfying assignments of the Boolean variables in the original formula. Both these techniques use incremental Boolean Satisfiability (SAT) techniques [CKSY04, McM02] to perform the Boolean quantifier elimination. These techniques have the benefit that the large number of calls to the theorem prover is avoided, and learning can be used to prune away the search space in the SAT solver. However, the translation from a first-order formula to a Boolean formula can result in a loss of structure (since the arithmetic operations are encoded as bitwise operations), and make the translation inefficient. Namjoshi and Kurshan [NK00] also proposed using quantifier elimination for first-order logic directly to perform predicate abstraction — however many theories (such as the theory of Equality with Uninterpreted Functions) do not admit quantifier elimination.

Most of the above approaches use decision procedures or SAT solvers as “black boxes”, at best in an incremental fashion, to perform predicate abstraction. We believe that having a customized procedure for predicate abstraction can help improve the efficiency of predicate abstraction on large problems.

Finally, there are a set of techniques to avoid computing the most precise abstraction upfront, and refine it only based on failed proof attempts in the verification tool. Das and Dill [DD01] and subsequently Ball et al. [BCDR04] use counterexamples to refine the predicate abstraction incrementally. Jhala and McMillan [JM05] use *interpolants* to refine the predicate abstraction. It is not clear if it is always preferable to compute the abstraction incrementally. But, we have observed that the refinement loop can often become the main bottleneck in these techniques (for example in SLAM), and limits the scalability of the overall system [BCDR04].

2. SETUP

Figure 1 defines the syntax of a quantifier-free fragment of first-order logic. An expression in the logic can either be a *term* or a *formula*. A *term* can either be a variable or an

$$\begin{aligned}
\textit{term} & ::= \textit{variable} \mid \textit{function-symbol}(\textit{term}, \dots, \textit{term}) \\
\textit{atomic-formula} & ::= \textit{term} = \textit{term} \mid \textit{predicate-symbol}(\textit{term}, \dots, \textit{term}) \\
\textit{formula} & ::= \textbf{true} \mid \textbf{false} \mid \textit{atomic-formula} \\
& \quad \mid \textit{formula} \wedge \textit{formula} \mid \textit{formula} \vee \textit{formula} \mid \neg \textit{formula}
\end{aligned}$$

Figure 1: Syntax of a quantifier-free fragment of first-order logic.

application of a function symbol to a list of terms. A *formula* can be the constants **true** or **false** or an atomic formula or Boolean combination of other formulas. Atomic formulas can be formed by an equality between terms or by an application of a predicate symbol to a list of terms.

The function and predicate symbols can either be *uninterpreted* or can be defined by a particular theory. For instance, the theory of integer linear arithmetic defines the function-symbol “+” to be the addition function over integers and “<” to be the comparison predicate over integers. If an expression involves function or predicate symbols from multiple theories, then it is said to be an expression over *mixed* theories.

A formula F is said to be *satisfiable* if it is possible to assign values to the various symbols in the formula from the domains associated with the theories to make the formula **true**. A formula is *valid* if $\neg F$ is not satisfiable (or unsatisfiable). We say a formula A *implies* a formula B ($A \Rightarrow B$) if and only if $(\neg A) \vee B$ is valid.

We define a *shared expression* to be a Directed Acyclic Graph (DAG) representation of an expression where common subexpressions can be shared, by using names to refer to common subexpressions. For example, the intermediate variable t refers to the expression e_1 in the shared expression “**let** $t = e_1$ **in** $(e_2 \wedge t) \vee (e_3 \wedge \neg t)$ ”.

2.1. Predicate Abstraction. A *predicate* is an atomic formula or its negation³. If G is a set of predicates, then we define $\tilde{G} \doteq \{\neg g \mid g \in G\}$, to be the set containing the negations of the predicates in G . We use the term “predicate” in a general sense to refer to any atomic formula or its negation and should not be confused to only mean the set of predicates that are used in predicate abstraction.

Definition 2.1. For a set of predicates P , a *literal* l_i over P is either a predicate p_i or $\neg p_i$, where $p_i \in P$. A *cube* c over P is a conjunction of literals. A *clause* cl over P is a disjunction of literals. Finally, a *minterm* over P is a cube with $|P|$ literals, and exactly one of p_i or $\neg p_i$ is present in the cube.

Given a set of predicates $P \doteq \{p_1, \dots, p_n\}$ and a formula e , the main operation in predicate abstraction involves constructing the *weakest* Boolean formula $\mathcal{F}_P(e)$ over P such that $\mathcal{F}_P(e) \Rightarrow e$. The expression $\mathcal{F}_P(e)$ can be expressed as the set of all the minterms over P that imply e :

$$\mathcal{F}_P(e) = \bigvee \{c \mid c \text{ is a minterm over } P \text{ and } c \text{ implies } e\} \quad (2.1)$$

Proposition 2.2. *For a set of predicates P and a formula e , the following statements are true:*

³We always use the term “predicate symbol” (and not “predicate”) to refer to symbols like “<”.

$\frac{X = Y}{Y = X}$	$\frac{X = Y \quad X \neq Y}{\perp}$
$\frac{X = Y \quad Y = Z}{X = Z}$	$\frac{X_1 = Y_1 \quad \cdots \quad X_n = Y_n}{f(X_1, \dots, X_n) = f(Y_1, \dots, Y_n)}$

Figure 2: Inference rules for theory of equality and uninterpreted functions.

- (1) $\mathcal{F}_P(\neg e) \Rightarrow \neg \mathcal{F}_P(e)$,
- (2) $\mathcal{F}_P(e_1 \wedge e_2) \Leftrightarrow \mathcal{F}_P(e_1) \wedge \mathcal{F}_P(e_2)$, and
- (3) $\mathcal{F}_P(e_1) \vee \mathcal{F}_P(e_2) \Rightarrow \mathcal{F}_P(e_1 \vee e_2)$

Proof. These properties follow very easily from the definition of \mathcal{F}_P .

We know that $\mathcal{F}_P(e) \Rightarrow e$, by the definition of $\mathcal{F}_P(e)$. By contrapositive rule, $\neg e \Rightarrow \neg \mathcal{F}_P(e)$. But $\mathcal{F}_P(\neg e) \Rightarrow \neg e$. Therefore, $\mathcal{F}_P(\neg e) \Rightarrow \neg \mathcal{F}_P(e)$.

To prove the second equation, we prove that (i) $\mathcal{F}_P(e_1 \wedge e_2) \Rightarrow (\mathcal{F}_P(e_1) \wedge \mathcal{F}_P(e_2))$, and (ii) $(\mathcal{F}_P(e_1) \wedge \mathcal{F}_P(e_2)) \Rightarrow \mathcal{F}_P(e_1 \wedge e_2)$. Since $e_1 \wedge e_2 \Rightarrow e_i$ (for $i \in \{1, 2\}$), $\mathcal{F}_P(e_1 \wedge e_2) \Rightarrow \mathcal{F}_P(e_i)$. Therefore $\mathcal{F}_P(e_1 \wedge e_2) \Rightarrow (\mathcal{F}_P(e_1) \wedge \mathcal{F}_P(e_2))$. On the other hand, $\mathcal{F}_P(e_1) \Rightarrow e_1$ and $\mathcal{F}_P(e_2) \Rightarrow e_2$, $\mathcal{F}_P(e_1) \wedge \mathcal{F}_P(e_2) \Rightarrow e_1 \wedge e_2$. Since $\mathcal{F}_P(e_1 \wedge e_2)$ is the weakest expression that implies $e_1 \wedge e_2$, $\mathcal{F}_P(e_1) \wedge \mathcal{F}_P(e_2) \Rightarrow \mathcal{F}_P(e_1 \wedge e_2)$.

To prove the third equation, note that $\mathcal{F}_P(e_1) \vee \mathcal{F}_P(e_2) \Rightarrow e_1 \vee e_2$ and $\mathcal{F}_P(e_1 \vee e_2)$ is the weakest expression that implies $e_1 \vee e_2$. □

The operation $\mathcal{F}_P(e)$ does not distribute over disjunctions. Consider the example where $P \doteq \{x \neq 5\}$ and $e \doteq x < 5 \vee x > 5$. In this case, $\mathcal{F}_P(e) = x \neq 5$. However $\mathcal{F}_P(x < 5) = \mathbf{false}$ and $\mathcal{F}_P(x > 5) = \mathbf{false}$ and thus $(\mathcal{F}_P(x < 5) \vee \mathcal{F}_P(x > 5))$ is not the same as $\mathcal{F}_P(e)$.

The above properties suggest that one can adopt a two-tier approach to compute $\mathcal{F}_P(e)$ for any formula e :

- (1) Convert e into an *equivalent* Conjunctive Normal Form (CNF), which comprises of a conjunction of clauses, i.e., $e \equiv (\bigwedge_i cl_i)$.
- (2) For each clause $cl_i \doteq (e_1^i \vee e_2^i \dots \vee e_m^i)$, compute $r_i \doteq \mathcal{F}_P(cl_i)$ and return $\mathcal{F}_P(e) \doteq \bigwedge_i r_i$.

To obtain an equivalent CNF form, one cannot introduce auxiliary variables (to keep the size of the resulting formula linear in the size of the input formula), as is typically done during an equisatisfiable CNF translation. These auxiliary variables introduced have to be existentially quantified out to obtain an equivalent formula. In our case, the CNF representation of the formula can be exponentially large compared to the original formula. However, we can use recent techniques to obtain the CNF form lazily, by a method proposed by McMillan [McM02].

For the rest of the paper, we focus here on computing $\mathcal{F}_P(\bigvee_{e_i \in E} e_i)$ when e_i is a predicate. Unless specified otherwise, we always use e to denote $(\bigvee_{e_i \in E} e_i)$, a disjunction of predicates in the set E in the sequel.

3. SYMBOLIC DECISION PROCEDURES (SDP)

We now show how to perform predicate abstraction using symbolic decision procedures. We start by describing a saturation-based decision procedure for a theory T and then use it to describe the meaning of a symbolic decision procedure for the theory T . Finally, we show how a symbolic decision procedure can yield a shared expression of $\mathcal{F}_P(e)$ for predicate abstraction.

A set of predicates G (over theory T) is unsatisfiable if the formula $(\bigwedge_{g \in G} g)$ is unsatisfiable. For a given theory T , the decision procedure for T takes a set of predicates G in the theory and checks if G is unsatisfiable. A theory is defined by a set of *inference rules*. An inference rule R is of the form:

$$\frac{A_1 \quad A_2 \quad \dots \quad A_n}{A} \quad (\text{R})$$

which denotes that the predicate A can be derived from predicates A_1, \dots, A_n in one step. Each theory has at least one inference rule for deriving *contradiction* (\perp). We also use $g : -g_1, \dots, g_k$ to denote that the predicate g (or \perp , where $g = \perp$) can be derived from the predicates g_1, \dots, g_k using one of the inference rules in a single step. Figure 2 describes the inference rules for the theory of Equality and Uninterpreted Functions.

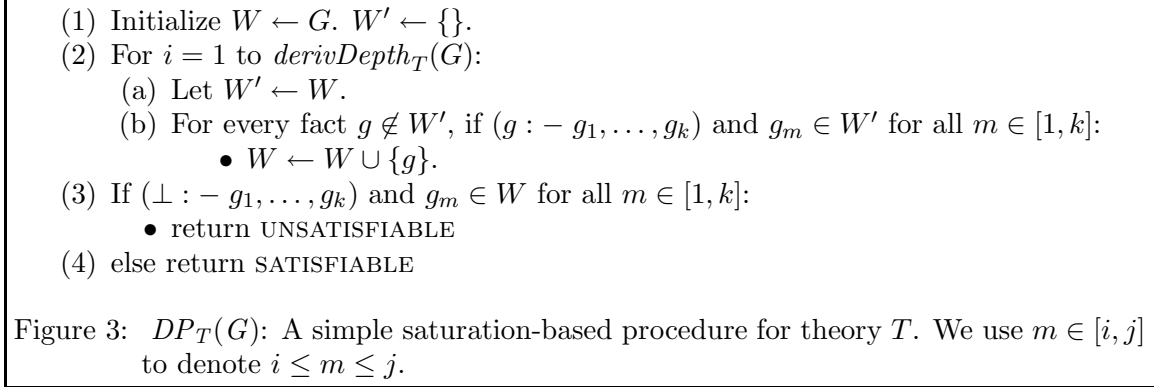
3.1. Saturation based decision procedures. Consider a simple saturation-based procedure DP_T shown in Figure 3, that takes a set of predicates G as input and returns SATISFIABLE or UNSATISFIABLE.

The algorithm maintains two sets: (i) W is the set of predicates derived from G up to (and including) the current iteration of the loop in step (2); (ii) W' is the set of all predicates derived before the current iteration. These sets are initialized in step (1). During each iteration of step (2), if a new predicate g can be derived from a set of predicates $\{g_1, \dots, g_k\} \subseteq W'$, then g is added to W . The loop terminates after a bound $derivDepth_T(G)$. In step (3), we check if *any* subset of facts in W can derive contradiction. If such a subset exists, the algorithm returns UNSATISFIABLE, otherwise it returns SATISFIABLE.

The parameter $d \doteq derivDepth_T(G)$ is a bound (that is determined solely by the set G for the theory T) such that if the loop in step (2) is repeated for at least d steps, then $DP_T(G)$ returns UNSATISFIABLE if and only if G is unsatisfiable. If such a bound exists for any set of predicates G in the theory, then DP_T procedure implements a decision procedure for T .

Definition 3.1. A theory T is called a *bounded saturation* theory, if the procedure DP_T described in Figure 3 implements a decision procedure for T .

In the rest of the paper, we only consider bounded saturation theories. Since there is no ambiguity, we will drop the term “bounded” in the rest of the paper and refer to such a theory as saturation theory. To show that a theory T is a saturation theory, it suffices to consider a decision procedure algorithm for T (say A_T) and show that DP_T implements A_T . This can be shown by deriving a bound on $derivDepth_T(G)$ for any set G in the theory.



3.2. Symbolic Decision Procedure. For a (saturation) theory T , a symbolic decision procedure for T (SDP_T) takes sets of predicates G and E as inputs, and symbolically simulates DP_T on $G' \cup \tilde{E}$, for every subset $G' \subseteq G$. The output of $SDP_T(G, E)$ is a symbolic expression representing those subsets $G' \subseteq G$, such that $G' \cup \tilde{E}$ is unsatisfiable. Thus with $|G| = n$, a single run of SDP_T symbolically executes 2^n runs of DP_T .

We introduce a set of Boolean variables $B_G \doteq \{b_g \mid g \in G\}$, one for each predicate in G . An assignment $\sigma : B_G \rightarrow \{\text{true}, \text{false}\}$ over B_G uniquely represents a subset $G' \doteq \{g \mid \sigma(b_g) = \text{true}\}$ of G .

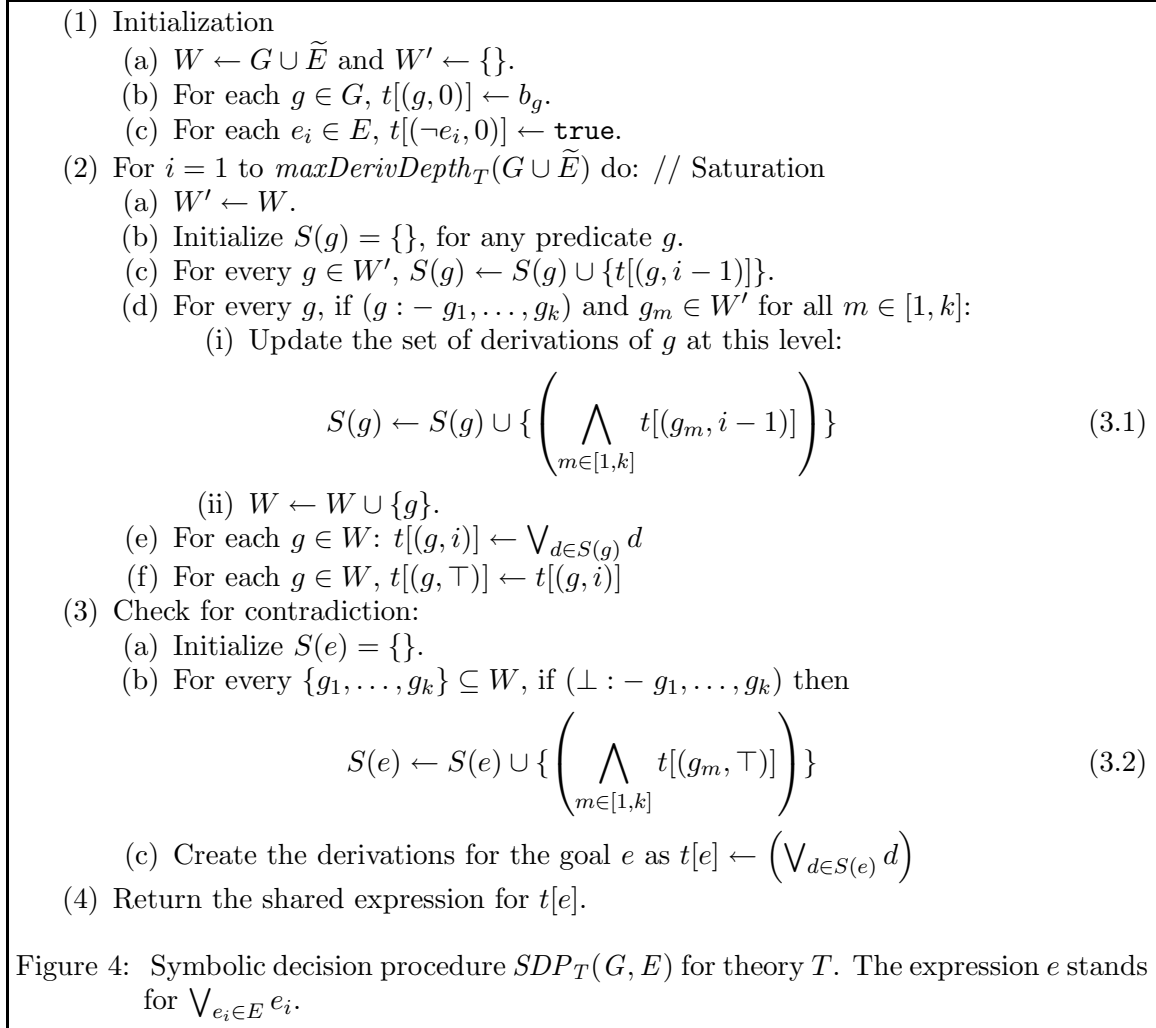
Figure 4 presents the symbolic decision procedure for a theory T , which symbolically executes the saturation based decision procedure DP_T on all possible subsets of the input component G . Just like the DP_T algorithm, this procedure also has three main components: *initialization*, *saturation* and *contradiction* detection. The algorithm also maintains sets W and W' , as the DP_T algorithm does.

Since $SDP(G, E)$ has to execute $DP_T(G' \cup \tilde{E})$ on all $G' \subseteq G$, the number of steps to iterate the saturation loop equals the maximum $\text{derivDepth}_T(G' \cup \tilde{E})$ for any $G' \subseteq G$. For a set of predicates S , we define the bound $\text{maxDerivDepth}_T(S)$ as follows:

$$\text{maxDerivDepth}_T(S) \doteq \max\{\text{derivDepth}_T(S') \mid S' \subseteq S\}$$

During the execution, the algorithm constructs a set of shared expressions with the variables over B_G as the leaves and temporary variables $t[\cdot]$ to name intermediate expressions. We use $t[(g, i)]$ to denote the expression for the predicate g after the iteration i of the loop in step (2) of the algorithm. We use $t[(g, \top)]$ to denote the top-most expression for g in the shared expression. Below, we briefly describe each of the phases of SDP_T :

- : *Initialization* [Step (1)]. The set W is initialized to $G \cup \tilde{E}$ and W' to $\{\}$. The leaves of the shared expression symbolically encode each subset $G' \cup \tilde{E}$, for every $G' \subseteq G$. For each $g \in G$, the leaf $t[(g, 0)]$ is set to b_g . For any $e_i \in E$, since $\neg e_i$ is present in all possible subset $G' \cup \tilde{E}$, we replace the leaf for $\neg e_i$ with **true**.
- : *Saturation* [Step (2)]. For each predicate g , $S(g)$ is the set of derivations of g from predicates in W' during any iteration. For any predicate g , we first add all the ways to derive g until the previous steps by adding $t[(g, i-1)]$ to $S(g)$. Every time g can be derived from some set of facts g_1, \dots, g_k such that each g_j is in W' , we add this derivation to $S(g)$ in Equation 3.1. At the end of the iteration i , $t[(g, i)]$ and $t[(g, \top)]$ are updated with the set of derivations in $S(g)$. The loop is executed $\text{maxDerivDepth}_T(G \cup \tilde{E})$ times.



: *Contradiction* [Steps (3,4)]. We know that if $G' \cup \tilde{E}$ is unsatisfiable, then G' implies e (recall, e stands for $\bigvee_{e_i \in E} e_i$). Therefore, each derivation of \perp from predicates in W gives a new derivation of e . The set $S(e)$ collects these derivations and constructs the final expression $t[e]$, which is returned in step (4).

The output of the procedure is the shared expression $t[e]$, where the leaves of the expression are the variables in B_G . The only operations in $t[e]$ are conjunction and disjunction; $t[e]$ is thus a Boolean expression (or a Boolean circuit) over B_G . The internal nodes in the expression are shared and can be inputs to multiple nodes in the subsequent level. We now define the evaluation of a (shared) Boolean expression inductively with respect to a subset $G' \subseteq G$.

Definition 3.2. For any Boolean expression $t[x]$ whose leaves are in set B_G , and a set $G' \subseteq G$, we define $eval(t[x], G')$ as the recursive evaluation of $t[x]$, after replacing each leaf b_g of $t[x]$ with \mathbf{true} if $g \in G'$ and with \mathbf{false} otherwise. The propositional connectives in the expression (\wedge and \vee) are interpreted using their standard meaning.

The following theorem explains the correctness of the symbolic decision procedure.

Theorem 3.3. *If $t[e] \doteq SDP_T(G, E)$, then for any set of predicates $G' \subseteq G$, $eval(t[e], G') = \mathbf{true}$ if and only if $DP_T(G' \cup \tilde{E})$ returns UNSATISFIABLE.*

To prove Theorem 3.3, we first describe an intermediate lemma about SDP_T . To disambiguate between the data structures used in DP_T and SDP_T , we use W_S and W'_S (corresponding to symbolic) to denote W and W' respectively for the SDP algorithm. Moreover, it is also clear that W' (respectively W'_S) at the iteration i ($i > 1$) is the same as W (respectively W_S) after $i - 1$ iterations.

Lemma 3.4. *For any set of predicates $G' \subseteq G$, at the end of i ($i \geq 0$) iterations of the loop in step (2) of $SDP_T(G, E)$ and $DP_T(G' \cup \tilde{E})$ procedures:*

- (1) $W \subseteq W_S$, and
- (2) $eval(t[(g, i)], G') = \mathbf{true}$ if and only if $g \in W$ for the DP_T algorithm.

Proof. We use an induction on i to prove this lemma, starting from $i = 0$.

For the base case (after step (1) of both algorithms), $W = G' \cup \tilde{E} \subseteq G \cup \tilde{E} \subseteq W_S$. Moreover, for this step, $eval(t[(g, 0)], G')$ for a predicate g can be \mathbf{true} in two ways.

- (1) If $g \in \tilde{E}$, then step (1) of SDP_T assigns it to \mathbf{true} . Therefore $eval(t[(g, 0)], G')$ is \mathbf{true} for any G' . But in step (1) of $DP_T(G' \cup \tilde{E})$, W contains all the predicates in $G' \cup \tilde{E}$, and therefore $g \in W$.
- (2) If $g \in G'$, then $eval(t[(g, 0)], G') = eval(b_g, G')$ which is \mathbf{true} , by the definition of $eval(\cdot, \cdot)$. Again $g \in W$ after step (1) of the DP_T algorithm too.

Let us assume that the inductive hypothesis holds for all values of i less than m . Consider the iteration number m . It is easy to see that if any fact g is added to W in this step, then g is also added to W_S ; therefore part (1) of the lemma is easily established.

To prove part (2) of the lemma, we will consider two cases depending of whether a predicate g was present in W before the m^{th} iteration:

- (1) Let us assume that after $m - 1$ iterations of $DP_T(G' \cup \tilde{E})$ procedure, $g \in W$. Since g is never removed from W during any step of DP_T , $g \in W$ after m iterations too. Now, by the inductive hypothesis, $eval(t[(g, m - 1)], G') = \mathbf{true}$. However, $t[(g, m - 1)] \implies t[(g, m)]$ (because $t[(g, m)]$ contains $t[(g, m - 1)]$ as one of its disjuncts in step 2(c) of the SDP_T algorithm). Therefore, $eval(t[(g, m)], G') = \mathbf{true}$.
- (2) We have to consider two cases depending on whether g can be derived in $DP_T(G' \cup \tilde{E})$ in step m .
 - (a) If g can't be derived in this step in DP_T algorithm, then there is no set $\{g_1, \dots, g_k\} \subseteq W'$ (of DP_T) such that $g : -g_1, \dots, g_k$. Since W' is the same as W after $m - 1$ iterations, we can invoke the induction hypothesis to show that there exists a predicate $g_j \in \{g_1, \dots, g_k\}$, $eval(t[(g_j, m - 1)], G') = \mathbf{false}$. Again, by the induction hypothesis, $eval(t[(g, m - 1)], G') = \mathbf{false}$, since $g \notin W$ after $m - 1$ steps. Thus $eval(t[(g, m)], G') = \mathbf{false}$.
 - (b) If g can be derived from $\{g_1, \dots, g_k\} \subseteq W'$ (of DP_T), then $\bigwedge_j t[(g_j, m - 1)]$ implies $t[(g, m)]$. But for each $g_j \in \{g_1, \dots, g_k\}$, $eval(t[(g_j, m - 1)], G') = \mathbf{true}$ and thus $eval(t[(g, m)], G') = \mathbf{true}$.

This completes the induction proof. □

We are now ready to complete the proof of Theorem 3.3.

Proof. Consider the situation where both $SDP_T(G, E)$ and $DP_T(G' \cup \tilde{E})$ have executed the loop in step (2) for $i = \maxDerivDepth_T(G \cup \tilde{E})$. We will consider two cases depending on whether \perp can be derived in $DP_T(G' \cup \tilde{E})$ in step (3).

- Suppose after i iterations, there is a set $\{g_1, \dots, g_k\} \subseteq W$, such that $\perp : -g_1, \dots, g_k$. This implies that $G' \cup \tilde{E}$ is unsatisfiable. By Lemma 3.4, we know that $eval(t[(g_j, \top)], G') = \mathbf{true}$ for each $g_j \in \{g_1, \dots, g_k\}$, and therefore $eval(t[e], G') = \mathbf{true}$.
- On the other hand, let $eval(t[e], G') = \mathbf{true}$. This implies that there exists a set $\{g_1, \dots, g_k\} \subseteq W_S$, such that $\perp : -g_1, \dots, g_k$ and $eval(t[(g_j, \top)], G') = \mathbf{true}$ for each $g_j \in \{g_1, \dots, g_k\}$. By Lemma 3.4, we know that $\{g_1, \dots, g_k\} \in W$, for the DP_T procedure too. This means that $DP_T(G' \cup \tilde{E})$ will return UNSATISFIABLE.

This completes the proof. \square

Corollary 3.5. *For a set of predicates P , if $t[e] \doteq SDP_T(P \cup \tilde{P}, E)$, then for any $P' \subseteq (P \cup \tilde{P})$ representing a minterm over P (i.e. $p_i \in P'$ iff $\neg p_i \notin P'$), $eval(t[e], P') = eval(\mathcal{F}_P(e), P')$.*

Hence $t[e]$ is a shared expression for $\mathcal{F}_P(e)$, where e denotes $\bigvee_{e_i \in E} e_i$. An explicit representation of $\mathcal{F}_P(e)$ can be obtained by first computing $t[e] \doteq SDP_T(P \cup \tilde{P}, E)$ and then enumerating the cubes over P that make $t[e]$ **true**.

In the following sections, we will instantiate T to be the EUF and DIF theories and show that SDP_T exists for such theories. For each theory, we only need to determine the value of $\maxDerivDepth_T(G)$ for any set of predicates G .

Example 3.6. Figure 5 demonstrates the working of the $SDP(G, E)$ for a simple example. The predicates in $G \doteq \{a = b, b = c, a = d, d = c\}$ and $E \doteq \{a = c\}$ are limited to equality and disequality predicates. For this theory T , $\maxDerivDepth_T(G \cup \tilde{E})$ equals the $\lg(m)$, where m is the number of terms in $G \cup \tilde{E}$. We do not show this result for equality theory in this paper, but prove it for the more general theory of difference logic in Section 3.4. Therefore, we need to iterate Step (2) of the algorithm, for $lg(\{a, b, c, d\}) = 2$ steps in Figure 4.

First, a Boolean variable b_g is introduced for each of the predicate $g \in G$. These variables represent $t[(g, 0)]$ for each $g \in G$. For each $e_i \in \tilde{E}$, we use **true** to represent $t[(e_i, 0)]$. Then the Step (2) of the algorithm is repeated for 2 steps. At each step, new derivations are produced from the existing set of predicates at the level. The nodes at each level denotes the set W for the particular iteration. Each derivation from two predicates in W is represented as the conjunction of the two predicates (using the diamond connective), and multiple derivations for a predicate (e.g. 3 ways to derive $a = c$ for $i = 2$) are represented with multiple incoming edges to a node.

Finally, the contradiction inference rule is used to derive contradictions (\perp) at the last level. Since the only way to derive contradiction in this example is using $a = c$ and $a \neq c$, this is the only derivation of \perp . The expression $t[e]$ represents the acyclic graph rooted at \perp , whose leaves are symbols in B_G . The expression $t[e]$ intuitively represents all the derivations of $a = c$ from G . More precisely, it represents all the subsets of G that are inconsistent with $a \neq c$.

There are a couple of observations that one can make from the previous example:

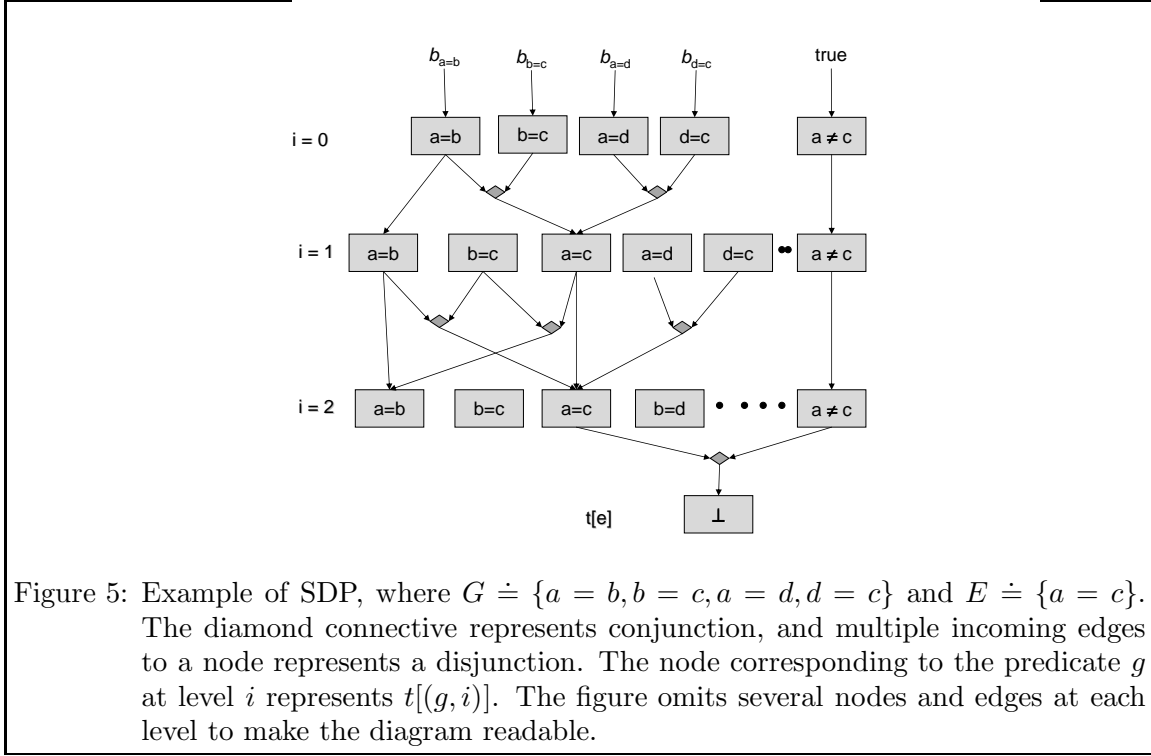


Figure 5: Example of SDP, where $G \doteq \{a = b, b = c, a = d, d = c\}$ and $E \doteq \{a = c\}$. The diamond connective represents conjunction, and multiple incoming edges to a node represents a disjunction. The node corresponding to the predicate g at level i represents $t[(g, i)]$. The figure omits several nodes and edges at each level to make the diagram readable.

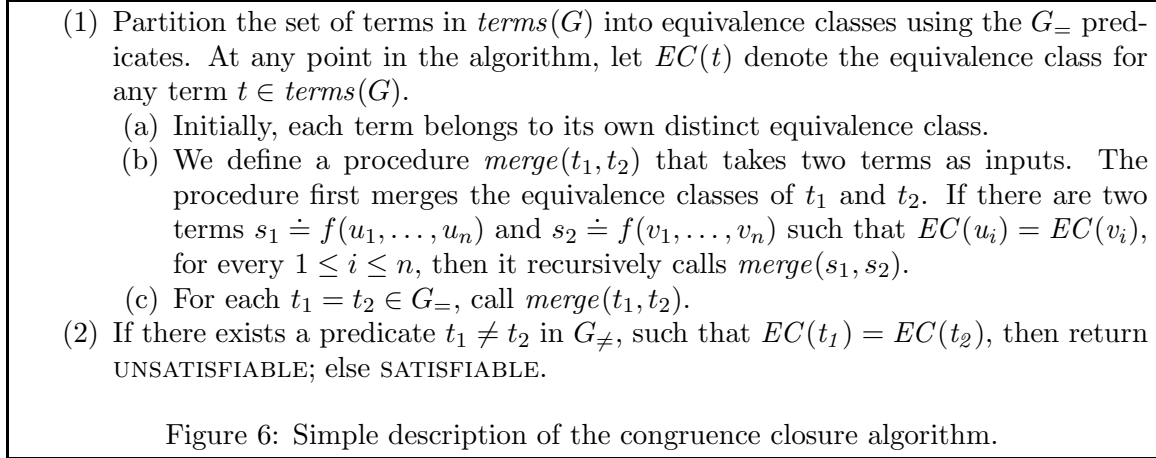
- (1) The expression $t[e]$ is a Boolean formula with B_G as inputs and an alternation of AND and OR operations. There are no negations (NOT) in the formula.
- (2) Even for this simple example, there are several redundant derivations. For example, consider the node $a = b$ in level $i = 2$. At this level, $a = b$ can either be derived from $a = b$ or from $b = c$ and $a = c$, in the previous level. However, the derivation of $a = c$ in level $i = 1$ already uses $a = b$ (at level $i = 0$) for one of its derivations. This means that the set of derivations of $a = b$ in level $i = 2$ contains redundant derivations. These derivations do not affect the correctness of the procedure, but simply increases the size of $t[e]$. However, as we will see in the next two sections, the size of the graph for $t[e]$ is still (pseudo) polynomially bounded for interesting theories.

Remark 3.7. It may be tempting to terminate the loop in step (2) of $SDP_T(G, E)$ once the set of predicates in W does not change across two iterations. However, this would lead to an incomplete procedure and the following example demonstrates this.

Example 3.8. Consider an example where G contains a set of predicates that denotes an “almost” fully connected graph over vertices x_1, \dots, x_n . G contains an equality predicate between every pair of variables except the edge between x_1 and x_n . Let $E \doteq \{x_1 = x_n\}$.

After one iteration of the SDP_T algorithm on this example, W will contain an equality between every pair of variables including x_1 and x_n since $x_1 = x_n$ can be derived from $x_1 = x_i, x_i = x_n$, for every $1 < i < n$. Therefore, if the SDP_T algorithm terminates once the set of predicates in W stabilizes, the procedure will terminate after two steps.

Now, consider the subset $G' = \{x_1 = x_2, x_2 = x_3, \dots, x_i = x_{i+1}, \dots, x_{n-1} = x_n\}$ of G . For this subset of G , $DP_T(G' \cup E)$ requires $\lg(n) > 1$ (for $n > 2$) steps to derive the



fact $x_1 = x_n$. Therefore $SDP_T(G, E)$ does not simulate the action of $DP_T(G' \cup \tilde{E})$. More formally, we can show that $eval(t[e], G') = \mathbf{false}$, but $G' \cup \tilde{E}$ is unsatisfiable.

3.3. SDP for Equality and Uninterpreted Functions. The terms in this logic can either be variables or application of an uninterpreted function symbol to a list of terms. A predicate in this theory is $t_1 \sim t_2$, where t_i is a term and $\sim \in \{=, \neq\}$. For a set G of EUF predicates, $G_=$ and G_{\neq} denote the set of equality and disequality predicates in G , respectively. Figure 2 describes the inference rules for this theory.

Let $terms(\phi)$ denote the set of syntactically distinct terms in an expression (a term or a formula) ϕ . For example, $terms(f(h(x)))$ is $\{x, h(x), f(h(x))\}$. For a set of predicates G , $terms(G)$ denotes the union of the set of terms in any $g \in G$.

A decision procedure for EUF can be obtained by the *congruence closure* algorithm [NO80], described in Figure 6.

For a set of predicates G , let $m = |terms(G)|$. We can show that if we iterate the loop in step (2) of $DP_T(G)$ (shown in Figure 3) for at least $3m$ steps, then DP_T implement the congruence closure algorithm. More precisely, for two terms t_1 and t_2 in $terms(G)$, the predicate $t_1 = t_2$ will be derived within $3m$ iterations of the loop in step 2 of $DP_T(G)$ if and only if $EC(t_1) = EC(t_2)$ after step (1) of the congruence closure algorithm (see proof below).

Proposition 3.9. *For a set of EUF predicates G , if $m \doteq |terms(G)|$, then the value of $maxDerivDepth_T(G)$ for the theory is bound by $3m$.*

Proof. We first determine the $derivDepth_T(G)$ for any set of predicates in this theory.

Given a set of EUF predicates G , and two terms t_1 and t_2 in $terms(G)$, we need to determine the maximum number of iterations in step (2) of $DP_T(G)$ to derive $t_1 = t_2$ (if $G_=$ implies $t_1 = t_2$).

Recall that the congruence closure algorithm (described in Figure 6) is a decision procedure for the theory of EUF. At any point in the algorithm, the terms in G are partitioned into a set of equivalence classes. The operation $EC(t_1) = EC(t_2)$ is used to determine if t_1 and t_2 belong to the same equivalence class.

One way to maintain an equivalence class $C \doteq \{t_1, \dots, t_n\}$ is to keep an equality $t_i = t_j$ between every pair of terms in C . At any point in the congruence closure algorithm, the

set of equivalence classes corresponds to a set of equalities $C_=$ over terms. Then $EC(u) = EC(v)$ can be implemented by checking if $u = v \in C_=$. Although this is certainly not an efficient representation of equivalence classes, this representation allows us to build SDP_T for this theory.

Let us implement the $C'_= \doteq merge(C_=, t_1, t_2)$ operation that takes in the current set of equivalence classes $C_=$, two terms t_1 and t_2 that are merged and returns the set of equalities $C'_=$ denoting the new set of equivalence classes. This can be implemented using the step (2) of the DP_T algorithm as follows:

- (1) $C'_= \leftarrow C_= \cup \{t_1 = t_2\}$.
- (2) For every term $u \in EC(t_1)$, (i.e. $u = t_1 \in C_=$), add the predicate $u = t_2$ to $C'_=$ by the transitive rule $u = t_2 : - u = t_1, t_1 = t_2$. Similarly, for every $v \in EC(t_2)$, add the predicate $v = t_1$ to $C'_=$ by $v = t_1 : - v = t_2, t_2 = t_1$. All these steps can be performed in one iteration of step 2.
- (3) For every $u \in EC(t_1)$ and every $v \in EC(t_2)$, add the edge $u = v$ to $C'_=$ by either of the two transitive rules ($u = v : - u = t_2, t_2 = v$) or ($u = v : - u = t_1, t_1 = v$).
- (4) Return $C'_=$

If there are m distinct terms in G , then there can be at most m merge operations, as each merge reduces the number of equivalence classes by one and there were m equivalence classes at the start of the congruence closure algorithm. Each merge requires three iterations of the step (2) of the DP_T algorithm to generate the new equivalence classes. Hence, we will need at most $3m$ iterations of step (2) of DP_T to derive any fact $t_1 = t_2$ that is implied by $G_=$.

Observe that this decision procedure DP_T for EUF does not need to derive a predicate $t_1 = t_2$ from G , if both t_1 and t_2 do not belong to $terms(G)$. Otherwise, if one generates $t_1 = t_2$, then the infinite sequence of predicates $f(t_1) = f(t_2), f(f(t_1)) = f(f(t_2)), \dots$ can be generated without ever converging.

Again, since $maxDerivDepth_T(G)$ is the maximum $derivDepth_T(G')$ for any subset $G' \subseteq G$, and any G' can have at most m terms, $maxDerivDepth_T(G)$ is bounded by $3m$. We also believe that a more refined counting argument can reduce it to $2m$, because two equivalent classes can be merged simultaneously in the DP_T algorithm. \square

3.3.1. Complexity of SDP_T . The run time and size of expression generated by SDP_T depend both on $maxDerivDepth_T(G)$ for the theory and also on the maximum number of predicates in W at any point during the algorithm. The maximum number of predicates in W can be at most $m(m-1)/2$, considering equality between every pair of term. The disequalities are never used except for generating contradictions. It is also easy to verify that the size of $S(g)$ (used in step (2) of SDP_T) is polynomial in the size of input. Hence the run time of SDP_T for EUF and the size of the shared expression returned by the procedure is polynomial in the size of the input.

3.4. SDP for Difference Logic. Difference logic is a simple yet useful fragment of linear arithmetic, where predicates are of the form $x \bowtie y + c$, where x, y are variables, $\bowtie \in \{<, \leq\}$ and c is a real constant. Any equality $x = y + c$ is represented as a conjunction of $x \leq y + c$ and $y \leq x - c$. The variables x and y are interpreted over real numbers. The function

	$X < Y + C$	$Y \bowtie X + D$	$C + D \leq 0$	
$\frac{X \leq Z + C \quad Z \bowtie Y + D}{X \bowtie Y + (C + D)}$	\perp			(c)
(A)	$X \leq Y + C$	$Y \leq X + D$	$C + D < 0$	
$\frac{X < Z + C \quad Z \bowtie Y + D}{X < Y + (C + D)}$	\perp			(D)
(B)	$X \leq Y$	$Y \leq X$		(E)
	$X = Y$			

Figure 7: Inference rules for Difference logic.

symbol “+” and the predicate symbols $\{<, \leq\}$ are the interpreted symbols of this theory. Figure 7 presents the inference rules for this theory⁴.

Given a set G of difference logic predicates, we can construct a graph where the vertices of the graph are the variables in G and there is a directed edge in the graph from x to y , labeled with (\bowtie, c) if $x \bowtie y + c \in G$. We will use a predicate and an edge interchangeably in this section.

Definition 3.10. A simple cycle $x_1 \bowtie x_2 + c_1, x_2 \bowtie x_3 + c_2, \dots, x_n \bowtie x_1 + c_n$ (where each x_i is distinct) is “illegal” if the sum of the edges is $d = \sum_{i \in [1, n]} c_i$ and either (i) all the edges in the cycle are \leq edges and $d < 0$, or (ii) at least one edge is an $<$ edge and $d \leq 0$.

It is well known [CLR90] that a set of difference predicates G is unsatisfiable if and only if the graph constructed from the predicates has a simple illegal cycle. Alternately, if we add an edge (\bowtie, c) between x and y for every simple path from x to y of weight c (\bowtie determined by the labels of the edges in the path), then we only need to check for simple cycles of length two in the resultant graph. This corresponds to the rules (C) and (D) in Figure 7.

For a set of predicates G , a predicate corresponding to a simple path in the graph of G can be derived within $lg(m)$ iterations of step (2) of DP_T procedure, where m is the number of variables in G (see proof below).

Proposition 3.11. For a set of DIF predicates G , if m is the number of variables in G , then $maxDerivDepth_T(G)$ for the DIF theory is bound by $lg(m)$.

Proof. It is not hard to see that if there is a simple path $x \bowtie_1 x_1 + c_1, x_1 \bowtie_2 x_2 + c_2, \dots, x_{n-1} \bowtie_n y + c_n$ in the original graph of G , then after $lg(m)$ iterations of the loop in step (2), there is a predicate $x \bowtie' y + c$ in W ; where $c = \sum_{i \in [1, n-1]} c_i$ and \bowtie' is $<$ if at least one of \bowtie_i is $<$ and \leq otherwise. This is because if there is a simple path between x and y through edges in G with length (number of edges from G) between 2^{i-1} and 2^i , then the algorithm DP_T generates a predicate for the path during iteration i .

However, DP_T can produce a predicate $x \bowtie y + c$, even though none of the simple paths between x and y add up to this predicate. These facts are generated by the non-simple paths that go around cycles one or more times. Consider the set $G \doteq \{x < y + 1, y < x - 2, x <$

⁴Constraints like $x \bowtie c$ are handled by adding a special variable x_0 to denote the constant 0, and rewriting the constraint as $x \bowtie x_0 + c$ [SSB02].

$z - 1, \dots\}$. In this case we can produce the fact $y < z - 3$ from $y < x - 2, x < z - 1$ and then $x < z - 2$ from $y < z - 3, x < y + 1$.

To prove the correctness of the DP_T algorithm, we will show these additional facts can be safely generated. Consider two cases:

- Suppose there is an illegal cycle in the graph. In that case, after $lg(m)$ steps, we will have two facts $x \bowtie y + c$ and $y \bowtie x + d$ in W such that they form an illegal cycle. Thus DP_T returns unsatisfiable.
- Suppose there are no illegal cycles in the original graph for G . For simplicity, let us assume that there are only $<$ edges in the graph. A similar argument can be made when \leq edges are present.

In this case, every cycle in the graph has a strictly positive weight. A predicate $x \bowtie y + d$ can be generated from non-simple paths only if there is a predicate $x \bowtie y + c \in G$ such that $c < d$. The predicate $x \bowtie y + d$ can't be a part of an illegal cycle, because otherwise $x \bowtie y + c$ would have to be part of an illegal cycle too. Hence DP_T returns satisfiable.

Note that we do not need any inference rule to weaken a predicate, $X < Y + D : - X < Y + C$, with $C < D$. This is because we use the predicates generated only to detect illegal cycles. If a predicate $x < y + c$ does not form an illegal cycle, then neither does any weaker predicate $x < y + d$, where $d \geq c$. \square

3.4.1. *Complexity of SDP_T .* Let c_{max} be the absolute value of the largest constant in the set G . We can ignore any derived predicate in of the form $x \bowtie y + C$ from the set W where the absolute value of C is greater than $(m - 1) * c_{max}$. This is because the maximum weight of any simple path between x and y can be at most $(m - 1) * c_{max}$. Again, let $const(g)$ be the absolute value of the constant in a predicate g . The maximum weight on any simple path has to be a combination of these weights. Thus, the absolute value of the constant is bound by:

$$C \leq \min\{(m - 1) * c_{max}, \Sigma_{g \in G} const(g)\}$$

The maximum number of derived predicates in W can be $2 * m^2 * (2 * C + 1)$, where a predicate can be either \leq or $<$, with m^2 possible variable pairs and the absolute value of the constant is bound by C . This is a *pseudo polynomial* bound as it depends on the value of the constants in the input.

However, many program verification queries use a subset of difference logic where each predicate is of the form $x \bowtie y$ or $x \bowtie c$. For this case, the maximum number of predicates generated can be $2 * m * (m - 1 + k)$, where k is the number of different constants in the input.

4. COMBINING SDP FOR SATURATION THEORIES

In this section, we provide a method to construct a symbolic decision procedure for the combination of saturation theories T_1 and T_2 , given SDP for T_1 and T_2 . The combination is based on an extension of the Nelson-Oppen (N-O) framework [NO79] that constructs a decision procedure for the theory $T_1 \cup T_2$ using the decision procedures of T_1 and T_2 .

We assume that the theories T_1 and T_2 have disjoint signatures (i.e., they do not share any function symbol), and each theory T_i is *convex* and *stably infinite*⁵. Let us briefly explain the N-O method for combining decision procedures before explaining the method for combining *SDP*.

4.1. Nelson-Open method for Combining Decision Procedures. Given two theories T_1 and T_2 , and the decision procedures DP_{T_1} and DP_{T_2} , the N-O framework constructs the decision procedure for $T_1 \cup T_2$, denoted as $DP_{T_1 \cup T_2}$.

To decide an input set G , the first step in the procedure is to *purify* G into sets G_1 and G_2 such that G_i only contains symbols from theory T_i and G is satisfiable if and only if $G_1 \cup G_2$ is satisfiable. Consider a predicate $g \doteq p(t_1, \dots, t_n)$ in G , where p is a theory T_1 symbol. The predicate g is purified to g' by replacing each subterm t_j whose top-level symbol does not belong to T_1 with a fresh variable w_j . The expression t_j is then purified to t'_j recursively. We add g' to G_1 and the *binding predicate* $w_j = t'_j$ to the set G_2 . We denote the latter as binding predicate because it binds the fresh variable w_j to a term t'_j .

Let V_{sh} be the set of *shared* variables that appear in $G_1 \cap G_2$. A set of equalities Δ over variables in V_{sh} is maintained; Δ records the set of equalities implied by the facts from either theory. Initially, $\Delta = \{\}$.

Each theory T_i then alternately decides if $DP_{T_i}(G_i \cup \Delta)$ is unsatisfiable. If any theory reports UNSATISFIABLE, the algorithm returns UNSATISFIABLE; otherwise, the theory T_i generates the new set of equalities over V_{sh} that are implied by $G_i \cup \Delta$ ⁶. These equalities are added to Δ and are communicated to the other theory. This process is continued until the set Δ does not change. In this case, the method returns SATISFIABLE. Let us denote this algorithm as $DP_{T_1 \cup T_2}$.

Theorem 4.1 ([NO79]). *For convex, stably infinite and signature-disjoint theories T_1 and T_2 , $DP_{T_1 \cup T_2}$ is a decision procedure for $T_1 \cup T_2$.*

There can be at most $|V_{sh}|$ irredundant equalities over V_{sh} , therefore the N-O loop terminates after $|V_{sh}|$ iterations for any input.

4.2. Combining SDP using Nelson-Open method. We will briefly describe a method to construct the $SDP_{T_1 \cup T_2}$ by combining SDP_{T_1} and SDP_{T_2} . As before, the input to the method is the pair (G, E) and the output is an expression $t[e]$. The facts in E are also purified into sets E_1 and E_2 and the new binding predicates are added to either G_1 or G_2 .

Our goal is to symbolically encode the runs of the N-O procedure for $G' \cup \tilde{E}$, for every $G' \subseteq G$. For any equality predicate δ over V_{sh} , we maintain an expression ψ_δ that records all the different ways to derive δ (initialized to **false**). We also maintain an expression ψ_e to record all the derivations of e (initialized to **false**).

The N-O loop operates just like the case for constructing $DP_{T_1 \cup T_2}$. The SDP_{T_i} for each theory T_i now takes $(G_i \cup \Delta, E_i)$ as input, where Δ is the set of equalities over V_{sh} derived so far. In addition to computing the (shared) expression $t[e]$ as before, SDP_{T_i} also

⁵We need these restrictions only to exploit the N-O combination result. The definition of convexity and stably infiniteness can be found in [NO79].

⁶We assume that each theory has an inference rule for deriving equality between variables in the theory, and DP_T also returns a set of equality over variables.

returns the expression $t[(\delta, \top)]$, for each equality δ over V_{sh} that can be derived in step (2) of the SDP_T algorithm.

The leaves of the expressions $t[e]$ and $t[(\delta, \top)]$ are $G_i \cup \Delta$ (since leaves for \widetilde{E}_i are replaced with **true**). We substitute the leaves for any $\delta \in \Delta$ with the expression ψ_δ , to incorporate the derivations of δ until this point. We also update $\psi_\delta \leftarrow (\psi_\delta \vee t[(\delta, \top)])$ to add the new derivations of δ . Similarly, we update $\psi_e \leftarrow (\psi_e \vee t[e])$ with the new derivations.

The N-O loop iterates $|V_{sh}|$ number of times to ensure that it has seen every derivation of a shared equality over V_{sh} from any set $G'_1 \cup G'_2 \cup \widetilde{E}_1 \cup \widetilde{E}_2$, where $G'_i \subseteq G_i$.

After the N-O iteration terminates, ψ_e contains all the derivations of e from G . However, at this point, there are two kind of predicates in the leaves of ψ_e ; the purified predicates and the binding predicates. If g' was the purified form of a predicate $g \in G$, we replace the leaf for g' with b_g . The leaves of the binding predicates are replaced with **true**, as the fresh variables in these predicates are really names for subterms in any predicate, and thus their presence does not affect the satisfiability of a formula. Let $t[e]$ denote the final expression for ψ_e that is returned by $SDP_{T_1 \cup T_2}$. Observe that the leaves of $t[e]$ are variables in B_G .

Theorem 4.2. *For two convex, stably-infinite and signature-disjoint theories T_1 and T_2 , if $t[e] \doteq SDP_{T_1 \cup T_2}(G, E)$, then for any set of predicates $G' \subseteq G$, $eval(t[e], G') = \mathbf{true}$ if and only if $DP_{T_1 \cup T_2}(G' \cup E)$ returns UNSATISFIABLE.*

Since the theory of EUF and DIF satisfy all the restrictions of the theories of this section, we can construct an SDP for the combined theory that still runs in pseudo-polynomial time.

5. IMPLEMENTATION AND RESULTS

We have implemented a prototype of the symbolic decision procedure for the combination of EUF and DIF theories. To construct $\mathcal{F}_P(e)$, we first build a BDD (using the CUDD [CUD] BDD package) for the expression $t[e]$ (returned by $SDP_T(P \cup \widetilde{P}, E)$) and then enumerate the cubes from the BDD.

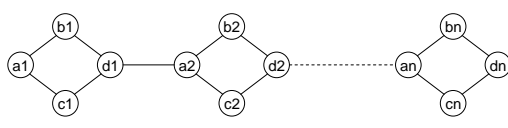
Creating the BDD for the shared expression $t[e]$ and enumerating the cubes from the BDD can have exponential complexity in the worst case. This is because the expression for $\mathcal{F}_P(e)$ can involve an exponential number of cubes (e.g. the example in Fig 8). However, most problems in practice have a few cubes in $\mathcal{F}_P(e)$. Secondly, as the number of leaves of $t[e]$ (alternately, number of BDD variables) is bound by $|P|$, the size of the overall BDD is usually small, and is computed efficiently in practice. Finally, by generating only the *prime implicants*⁷ of $\mathcal{F}_P(e)$ from the BDD, we obtain a compact representation of $\mathcal{F}_P(e)$.

We report preliminary results evaluating our symbolic decision procedure based predicate abstraction method on a set of software verification benchmarks. The benchmarks are generated from the predicate abstraction step for constructing Boolean Programs from C programs of Microsoft Windows device drivers in SLAM [BMMR01].

We compare our method with two other methods for performing predicate abstraction:

- DP-based: This method uses the decision procedure ZAPATO [BCLZ04] to enumerate the set of cubes that imply e . Various optimizations (e.g. considering cubes in

⁷ For any Boolean formula ϕ over variables in V , prime implicants of ϕ is a set of cubes $C \doteq \{c_1, \dots, c_m\}$ over V such that $\phi \Leftrightarrow \bigvee_{c \in C} c$ and two or more cubes from C can't be combined to form a larger cube.



n	$ P $	SDP_T time (s)	UCLID time (s)
3	14	0.20	19.37
4	19	0.43	656
5	24	0.65	-
10	49	5.81	-
12	59	12.28	-

Figure 8: Result on diamond examples with increasing number of diamonds. The expression e is $(a1 = dn)$. A “-” denotes a timeout of 1000 seconds.

increasing order of size) are used to prevent enumerating exponential number of cubes in practice.

- : UCLID-based: This method performs quantifier-elimination using incremental SAT-based methods [LBC03]. The procedure works by first converting the problem into an existential quantifier elimination problem in first-order logic and then reducing it to Boolean quantifier elimination by using an encoding to Boolean logic. Finally, it uses SAT-based methods for performing Boolean quantification.

To compare with the DP-based method, we generated 665 predicate abstraction queries from the verification of device-driver programs. Most of these queries had between 5 and 14 predicates in them and are fairly representative of queries in SLAM. The run time of DP-based method was 27904 seconds on a 3 GHz. machine with 1GB memory. The run time of SDP -based method was 273 seconds. This gives a little more than 100X speedup on these examples, demonstrating that our approach can scale much better than decision procedure based methods. We have not been able to run UCLID-based method on these particular SLAM benchmarks; the UCLID-based tool is no longer actively maintained, and we had trouble translating these SLAM benchmarks to input of UCLID. From our earlier experience of using UCLID on similar benchmarks (Fig. 3 in [LBC03]), we believe that most of these benchmarks can be solved within a few seconds, and the total runtime would not differ by more than 2–3X (in favor of the current technique).

To compare with UCLID-based approach, we generated different instances of a problem (see Figure 8 for the example) where P is a set of equality predicates representing n diamonds connected in a chain and e is an equality $a1 = dn$. We generated different problem instances by varying the size of n . For an instance with n diamonds, there are $5n - 1$ predicates in P and 2^n cubes in $\mathcal{F}_P(e)$ to denote all the paths from $a1$ to dn . Figure 8 shows the result comparing both the methods. We should note that UCLID method was run on a slightly slower 2GHz machine. The results illustrate that our method scales much better than the SAT-based enumeration used in UCLID for this example. Intuitively, UCLID-based approach grows exponentially with the number of predicates ($2^{|P|}$), whereas our approach only grows exponentially with the number of diamonds (2^n) in the result.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented the concept of symbolic decision procedures and showed its use for predicate abstraction. We have provided an algorithm for synthesizing a SDP for any bounded saturation theory. We show that such SDP exists for interesting theories such

as EUF and difference logic. These SDP construct a shared expression and run with polynomial and pseudo-polynomial complexity respectively. Finally, we have provided a method for constructing the SDP for simple mixed theories using an extension of the Nelson-Oppen combination framework. Preliminary results comparing it some of the existing approaches are encouraging.

There are several avenues of future work, some of which are outlined below:

- First, it is interesting to find out how to construct a SDP for other theories, including the theory of linear arithmetic (over rationals). For linear arithmetic, one can perform a “symbolic” Fourier-Motzkin [DE73] elimination procedure to construct an SDP — the inference rule would eliminate a variable from all the predicates in a given level. However, it is not clear how to generate implied equalities from such a procedure to combine the SDP with SDP for other theories.
- Second, as the example in Figure 5 illustrated, there are a lot of redundant derivations present in the resultant expression. The algorithm will benefit from optimizations that can minimize such redundant derivations.
- Extend the combination of SDPs to non-convex theories.

REFERENCES

- [BCDR04] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining Approximations in Software Predicate Abstraction. In Kurt Jensen and Andreas Podelski, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, LNCS 2988, pages 388–403. Springer-Verlag, 2004.
- [BCLZ04] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic Theorem Proving for Software Predicate Abstraction Refinement. In R. Alur and D. Peled, editors, *Computer Aided Verification (CAV '04)*, LNCS 3114. Springer-Verlag, 2004.
- [BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, pages 203–213, Snowbird, Utah, June, 2001. *SIGPLAN Notices*, 36(5), May 2001.
- [Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [CCG⁺04] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.
- [CKSY04] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, September–November 2004.
- [CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [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 *IEEE Symposium of Logic in Computer Science (LICS '01)*, pages 51–60. IEEE Computer Society, June 2001.
- [DDP99] S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification (CAV '99)*, LNCS 1633, pages 160–171. Springer-Verlag, July 1999.
- [DE73] G.B. Dantzig and B. C. Eaves. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory*, A(14):288–297, 1973.
- [FQ02] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In Launchbury and Mitchell [LM02], pages 191–202.
- [GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer-Aided Verification (CAV '97)*, LNCS 1254. Springer-Verlag, June 1997.
- [HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In Launchbury and Mitchell [LM02], pages 58–70.

- [JM05] R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 39–51. Springer, 2005.
- [LBC03] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In W. A. Hunt, Jr. and F. Somenzi, editors, *Computer-Aided Verification (CAV 2003)*, LNCS 2725, pages 141–153. Springer-Verlag, 2003.
- [LM02] John Launchbury and John C. Mitchell, editors. *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL '02)*. ACM Press, 2002.
- [McM02] K. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In E. Brinksma and K. G. Larsen, editors, *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 250–264, July 2002.
- [NK00] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In A. Emerson and P. Sistla, editors, *Computer Aided Verification*, LNCS 1855, pages 435–449, 2000.
- [NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):245–257, 1979.
- [NO80] G. Nelson and D. C. Oppen. Fast decision procedures based on the congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.
- [SS99] H. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwegs and D. Peled, editors, *Computer-Aided Verification*, volume 1633 of *LNCS*, pages 443–454. Springer-Verlag, July 1999.
- [SSB02] O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding Separation Formulas with SAT. In E. Brinksma and K. G. Larsen, editors, *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 209–222, July 2002.