Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2013, Article ID 964682, 13 pages
http://dx.doi.org/10.1155/2013/964682

Research Article

A Unified Framework for DPLL(T) + Certificates

Min Zhou,[1,2,3,4] Fei He,[1,2,3] Bow-Yaw Wang,[5] Ming Gu,[1,2,3] and Jiaguang Sun[1,2,3]

[1] *Tsinghua National Laboratory for Information Science and Technology (TNList), Beijing 100084, China*
[2] *School of Software, Tsinghua University, Beijing 100084, China*
[3] *Key Laboratory for Information System Security, MOE, Beijing 100084, China*
[4] *Department of Computer Science and Technologies, Tsinghua University, Beijing 100084, China*
[5] *Institute of Information Science, Academia Sinica, Taipei 115, Taiwan*

Correspondence should be addressed to Min Zhou; zhoumin03@gmail.com

Received 6 February 2013; Accepted 8 April 2013

Academic Editor: Xiaoyu Song

Satisfiability Modulo Theories (SMT) techniques are widely used nowadays. SMT solvers are typically used as verification backends. When an SMT solver is invoked, it is quite important to ensure the correctness of its results. To address this problem, we propose a unified certificate framework based on DPLL(T), including a uniform certificate format, a unified certificate generation procedure, and a unified certificate checking procedure. The certificate format is shown to be simple, clean, and extensible to different background theories. The certificate generation procedure is well adapted to most DPLL(T)-based SMT solvers. The soundness and completeness for DPLL(T) + certificates were established. The certificate checking procedure is straightforward and efficient. Experimental results show that the overhead for certificates generation is only 10%, which outperforms other methods, and the certificate checking procedure is quite time saving.

1. Introduction

1.1. Background and Motivation. Satisfiability Modulo Theories (SMT) techniques are getting increasingly popular in many verification applications. An SMT solver takes as input an arbitrary formula given in a specific fragment of first order logic with interpreted symbols. It returns a judgment telling whether the formula is satisfiable. By satisfiable, it means that there exists at least one interpretation under which the formula evaluates to true.

SMT solvers are typically used as verification backends. During a verification process, SMT solver may be invoked many times. Each time the SMT solver is given a complex logical formula and is expected to give a correct judgment. However, SMT solvers employ many sophisticated data structures and tricky algorithms, which make their implementations prone to error. For instance, there are some solvers disqualified in SMT-COMP due to their incorrect judgments [1]. We therefore strongly believe that SMT solvers should be supported with a formal assessment of their correctness.

Ensuring the correctness of an SMT solver is never easy. Generally, there are two approaches: to verify the SMT solver or to certify their judgments. For the former approach, it proves directly that the solver is correct. So, its judgements are naturally correct. Though rigor in logic, this method is not generally practical because verifying an SMT solver requires great workload. Not only the algorithm but also the implementation should be verified in a formal way. To the best of our knowledge, no state-of-the-art SMT solver has been completely verified. More importantly, this approach is solver dependent. Even if a solver is completely verified, the whole verification has to be done again when the solver is modified.

The rationale of the latter approach is as follows: instead of proving the correctness of the solver, we prove its judgment. For this purpose, the solver is required to return a piece of evidence to support its judgment. This evidence is called *certificate* in this paper. Then, the correctness for the judgment is ensured by the certificate. If so, a rigorous and detailed proof can be constructed by referring to the certificate. Obviously, checking a certificate is much easier than

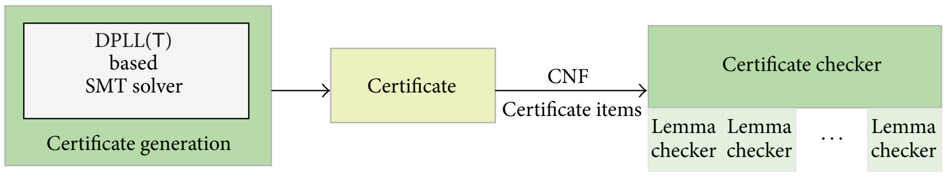

Figure 1: The road map of our approach.

verifying a solver. If the certificates are presented in a proper format, its checking algorithm could have low complexity. For instance, linear time is required to check certain certificates for unsatisfiable SAT problems. The certificate approach is solver independent. We need not to look into the solvers; the only requirement for solvers is to generate certificates. If the certificate format is unified, the certificate checking can be a uniform procedure. This benefit is vital since we need not to develop a certificate checker for each SMT solver.

There are some SMT solvers that support certificate generation, such as CVC3 [2] and Z3 [3]. However, their certificate formats are quite different. As a result, the generation and checking procedures are tool specific. Although SMT solvers differ in the algorithms, we believe that their formats of certificates could be unified. The basic idea comes from SAT. In an SAT community, people consider to generate certificates along with the DPLL framework, and the certificates (for unsatisfiability instances) are unified as chains of linear regular resolutions which can lead from the initial clauses to an empty clause [4, 5]. With this uniform format, the generation and checking procedures for certificates can also be unified.

This paper considers the DPLL(T) framework, which is extended from DPLL and has been widely used in state-of-the-art SMT solvers. A uniform format of SMT certificate is defined in this paper. The road map of our approach is shown in Figure 1. Note that the size of certificate can be exponential or even larger with respect to the input problem [6]. To reduce storage space, the certificate format is defined carefully in a concise way. Upon this format, a unified procedure for certificates generation is proposed. This procedure can be easily integrated into DPLL(T)-based SMT solvers. Moreover, a certificate checking procedure is also proposed in this paper. The checking procedure is easy, fast, and memory saving. To make it extensible, theory lemmas are checked by individual lemma checkers. Experimental results show that the average overhead for the certificates generation is about 10%.

1.2. Related Work. Propositional satisfiability problem (SAT) is a well-known decision problem. It was proven to be NP-complete by Cook [7] in 1971. All known algorithms for SAT have exponential complexity. State-of-the-art SAT solvers are based on DPLL [8] algorithm. The basic idea is to explore all valuations by depth-first search in a branch and bound manner. To reduce unnecessary search efforts, an optimization called *clause learning* is used, which learns a clause whenever it recovers from a conflict. This optimization significantly increases the performance and makes SAT algorithms practical. During the clause learning process, each learned clause is a resolvent of linear regular resolution from existing clauses [4].

SMT is an extension of SAT. As the name implies, the input formula may contain theory-specific predicates and functions. For instance, while only boolean variables are allowed in SAT, $x \geq y \wedge y \geq f(a) + x \vee x < 0$ is a well-formed SMT formula. The background theory T of SMT is usually a composed theory from several individual theories. In this paper, we focus on quantifier-free SMT problems, for which the popular algorithm is DPLL(T) [9]. The DPLL(T) algorithm is extended from DPLL. Since the DPLL(T) algorithm is described as a transition system, solving an SMT problem is actually seeking a path to the *success* or *fail* state while respecting the guard conditions on transitions. Although there are other heuristics and optimizations, currently it is almost standard to develop SMT solvers based on DPLL(T).

Certificates are, informally, a set of evidence that can be used to construct a rigorous proof for the judgment. For satisfiable cases, the certificate could be a model. What is more interesting is the certificates for unsatisfiable cases. Deduction steps from the input problem to a conflict should be reproducible by referring to the certificate. In this paper, when saying certificates, we refer to those for unsatisfiable cases.

Many of the state-of-the-art SAT solvers can generate certificates, but their logical formats are very distinct from each other. For example, the certificate generated by zChaff [10] and MiniSat [11] are quite different. PicoSAT [12] generates concise and well-defined certificates. A tool called TraceCheck can do certificate checking [12]. In PicoSAT's certificates, there are initial clauses and resolvents. Resolvents are obtained from linear regular resolution. If all the clauses form a directional acyclic graph and the empty clause is derived, then it implies that the initial clause set is unsatisfiable. As shown in [4, 5], clause learning in DPLL corresponds to a linear regular resolution. So, this certificate format is also adoptable by other DPLL-based solvers.

For general SMT problems, quantifiers may be used. Furthermore, theory-specific inference rules should be considered when deciding their satisfiability. Various logical frameworks are developed to provide flexible languages to write their proof, such reference could be found in [13–16]. If we focus on quantifier-free problems, the proof format and proof checking could be simplified. In practice, solvers such as CVC3 and Z3 can generate certificates, but their formats are also quite different because they use different proof rules. Proof rules are a set of inference rules with respect to the background theory T. To be better suited for certificate generation and checking, the certificate format should not depend heavily on theory-specific proof rules.

The overhead for generating certificates should be as small as possible. An approach with 30% overhead is presented in [17], which uses the Edinburgh Logical Framework with Side Condition (LFSC). In our approach, we focus on quantifier-free problems so that the overhead is further reduced. Certificate generation is related to unsatisfiable core computation. In [18], a simple and flexible way of computing small unsatisfiable core in SMT was proposed. They compute by the propositional abstraction of SMT problem and invoke an existing unsatisfiable core extractor for SAT. Although the idea might be similar, unsatisfiable core computation concerns which clauses lead to unsatisfiability, whereas certificate generation concerns how. It is convenient to generate unsatisfiable core from certificates, but not vice versa.

To check certificates, people can either check the certificate directly, just as in the SAT case, or translate them into inputs for some theorem provers such as HOL Light [19] or Isabelle/HOL [20]. There is also research in checking certificates using simple term rewriting [21]. A known fact is that if certificates are checked by translating them to proof items for theorem provers, it could take much longer time than certificate finding [20].

1.3. Contribution. In this paper, we present a certificate framework for quantifier-free formulas. Compared to other works, our approach has the following advantages.

(i) The certificate format is simple, clean, and extensible to different underlying theories.

(ii) The certificate generation procedure is well adapted to most DPLL(T)-based SMT solvers. We also implement it in our solver.

(iii) On average, the certificate generation induces about 10% overhead which is much less than other approaches.

(iv) The certificate checking procedure is simplified. It has better performance.

The rest of this paper is organized as follows. In Section 2, we recall the original DPLL(T) algorithm. The certificate format is defined in Section 3. The framework of DPLL(T) + certificate is described in Section 4, where the formal rules are defined in Section 4.1, necessary implementation issues are discussed in Section 4.2, the properties are discussed in Section 4.3, and a couple of examples are shown in Section 4.4. Section 5 discusses the certificate checking matters. Experiments' results are shown in Section 6. Finally, Section 7 concludes the paper.

2. The DPLL(T) Algorithm

The DPLL(T) algorithm was proposed in [9]. The input is a quantifier-free first order formula in conjunctive normal form (CNF) from the background theory T. DPLL(T) was given as a transition system. Most features and optimizations of SMT algorithms can be formalized in that way.

In first order logic, any CNF formula can be viewed as a set of clauses. Each clause is disjunction of literals, and each literal is a positive or negative form of an atom. A formula is satisfiable if there exists an interpretation under which the formula evaluates to true.

In DPLL(T), formula satisfiability is considered in some background theory T (all interpreted symbols should be from T). We use "$\vDash_{\mathsf{T}}$" to denote theory entailment in T. Given sets of formulas Σ and Γ, if any interpretation that satisfies all formulas in Σ also satisfies all formulas Γ, we write $\Sigma \vDash_{\mathsf{T}} \Gamma$. Similarly, "$\vDash$" denotes propositional entailment which consider each atomic formula syntactically as a propositional literal.

Each state is a 2-tuple "$M \parallel F$" where M is a stack of the currently asserted literals and F the set of clauses to be satisfied. The transition relations are given by the following rules.

(i) Decide:

Precondition

(1) l or $\neg l$ occurs in a clause of F;

(2) l is undefined in M,

$$M \parallel F \longmapsto M\, l^{\mathsf{d}} \parallel F. \tag{1}$$

(ii) UnitPropagate:

Precondition

(1) $M \vDash \neg C$;

(2) l is undefined in M,

$$M \parallel F, C \vee l \longmapsto M\, l \parallel F, C \vee l. \tag{2}$$

(iii) TheoryPropagate:

Precondition

(1) $M \vDash_{\mathsf{T}} l$;

(2) l or $\neg l$ occurs in F;

(3) l is undefined in M,

$$M \parallel F \longmapsto M\, l \parallel F. \tag{3}$$

(iv) T-Backjump:

Precondition

(1) $M\, l^{\mathsf{d}} N \vDash \neg C$;

(2) there is some clause $C' \vee l'$ such that:

(a) $F, C \vDash_{\mathsf{T}} C' \vee l'$ and $M \vDash \neg C'$;

(b) l' is undefined in M;

(c) l' or $\neg l'$ occurs in F or in $M\, l^{\mathsf{d}} N$,

$$M\, l^{\mathsf{d}} N \parallel F, C \longmapsto M\, l' \parallel F, C. \tag{4}$$

(v) T-Learn:

Precondition

(1) each atom of C occurs in F or in M;

(2) $F \vDash_{\mathsf{T}} C$,

$$M \parallel F \longmapsto M \parallel F, C. \quad (5)$$

(vi) T-Forget:

Precondition

$F \vDash_{\mathsf{T}} C$,

$$M \parallel F, C \longmapsto M \parallel F. \quad (6)$$

(vii) Restart:

Precondition

T

$$M \parallel F \longmapsto \emptyset \parallel F. \quad (7)$$

(viii) Fail:

Precondition

(1) $M \vDash \neg C$;

(2) M contains no decision literals,

$$M \parallel F, C \longmapsto \langle \text{Fail} \rangle . \quad (8)$$

In DPLL(T) algorithm, there is an assumption that there exists a T-solver that can check the consistency of conjunctions of literals given in T. This algorithm will terminate under weak assumption and is proven sound and complete [9].

3. The Certificate Format

If a formula is satisfiable, the certificate is simply an interpretation under which the formula evaluates to true. On the other hand, if it is unsatisfiable, the certificate could be much more complicated. It should be proven that "all attempts to find a model failed." More precisely, each branch of the search tree must be tried. Technically speaking, this *closed search tree* can be presented as a refutation procedure which contains sequences of resolution procedures that produce the empty clause [12]. It is imaginable that if the search tree is quite large so is the set of refutation clauses.

Remember that in first order logic (FOL), constants are considered as nullary functions. Then, a term is either a variable or a function with arguments which are also terms. For example, x, $f(x, y)$ and $\phi(s, \psi(t))$ are valid terms. An atom (or atomic formula) is a predicate with arguments being terms. Note that nullary predicates are actually propositional variables.

Definition 1 (T-atom, T-literal). Given a background theory T, a T-atom is an atom whose functions and predicates are in the signature of T. A T-literal is the positive or negative form of a T-atom.

For example, given T the theory of *Equality with Uninterpreted Functions*, if p is a propositional variable, then p, $x = f(y)$ are T-atoms and $p, \neg p$, $x = f(y)$ and $x \neq f(y)$ are T-literals.

A proof rule is an inference rule which has several T-literals as premises and one T-literal as the conclusion. For proof rules whose conclusion is conjunction of n T-literals, we can split it to n proof rules and one for each.) For example, $x = y$ implies that $f(x) = f(y)$ is a proof rule (the monotonicity of equality). This rule holds for all T-terms x, y and all function f. Each proof rule can be instantiated to a theory lemma. For instance, if s, t are two specific T-terms and g is a specific function, then $(s = t) \rightarrow (g(s) = g(t))$ is a theory lemma.

Definition 2 (clause item). A clause item is one of the following three forms.

(i) Init $l_1 \vee \cdots \vee l_n$: an initial clause $l_1 \vee \cdots \vee l_n$, where all l_i are T-literals.

(ii) Res $\{C_1, \ldots, C_n\}$: a clause obtained from a linear resolution of $C_1, \ldots, C_n$, where all C_i are clauses items.

(iii) Lemma $l_1 \wedge \cdots \wedge l_n \rightarrow l'$: a theory lemma where all l_i and l' are T-literals. The defined clause is $\neg l_1 \vee \cdots \vee \neg l_n \vee l'$.

A resolution chain $C_1, C_2, \ldots, C_n$ is called a *linear regular resolution* chain if there exists a sequence of clauses $D_1, D_2, \ldots, D_{n-1}$ such that: (1) D_1 is the resolvent of C_1 and C_2; (2) for $2 \leq i < n$, D_i is the resolvent of D_{i-1} and C_{i+1}; (3) all resolutions are applied on different T-atoms.

A clause item gives the syntax form for a clause. For each clause item C, if it is well defined; the concrete form of C (given as disjunction of T-literals) can be calculated. It is called *a concrete clause*, denoted by $[\![C]\!]$. We do not distinguish C and $[\![C]\!]$ when no ambiguity is caused.

Definition 3 (certificate item). A certificate item is either of the following:

(i) Define ClauseItem,

(ii) Forget ClauseItem.

Definition 4 (certificate). A certificate P with respect to an SMT instance is a sequence of certificate items. The size of P, denoted by $|P|$, is the number of certificate items contained in P. $P[i]$ is the ith element in P, for $i = 1, 2, \ldots, |P|$.

Definition 5 (context). Given a certificate P and a number $0 \leq i \leq |P|$, the context with respect to i, denoted by $\lfloor P \rfloor$, is a set of clause items such that:

$$\lfloor P \rfloor_i = \begin{cases} \emptyset & \text{if } i = 0 \\ \lfloor P \rfloor_{i-1} \cup \{C\} & \text{if } i > 0 \text{ and } P[i-1] \text{ is Define}C \\ \lfloor P \rfloor_{i-1} \setminus \{C\} & \text{if } i > 0 \text{ and } P[i-1] \text{ is Forget}C. \end{cases} \quad (9)$$

Specially, denote $\lfloor P \rfloor_{|P|}$ by $\lfloor P \rfloor$.

Table 1: A certificate example.

	Clause item			$[\![C]\!]$	Lemma
C_1	Define	Init	$\neg l_1 \vee \neg l_2$	$\neg l_1 \vee \neg l_2$	
C_2	Define	Init	l_1	l_1	
C_3	Define	Lemma	$l_1 \rightarrow l_2$	$\neg l_1 \vee l_2$	The property of "="
C_4	Define	Res	C_1, C_3, C_2	$\square$	The empty clause

Example 6. Assume that $l_1 : x = y, l_2 : f(x) = f(y)$ and $C_1 : \neg l_1 \vee \neg l_2, C_2 : l_1$. A certificate for the unsatisfiability of the clause set $\{C_1, C_2\}$ is presented in Table 1. In this example, $\{C_1, C_2\}$ is intuitively unsatisfiable. Note that C_2 entails l_1 which implies l_2 in $\mathsf{T}_{\mathrm{EUF}}$, while $C_1, C_2 \vDash \neg l_2$.

Definition 7 (well-formed certificate). A certificate P is well formed if for all $1 \le i \le |P|$:

(i) *if* $P[i]$ *is a* Define *item then one of the following conditions hold:*

  (a) it is of Init type;

  (b) it is of Res $\{C_1, \ldots, C_n\}$ type where $C_k \in \lfloor P \rfloor_{i-1}$ for all $1 \le k \le n$, and there should be a linear regular resolution from $\{[\![C_1]\!], [\![C_2]\!], \ldots, [\![C_n]\!]\}$ to $P[i]$;

  (c) it is of Lemma type and the concrete clause is valid in theory T, that is, $\vDash_{\mathsf{T}} [\![P[i]]\!]$;

(ii) *otherwise* $P[i]$ *is a Forget item and the referred clause must be defined in* $\lfloor P \rfloor_{i-1}$.

4. Certificate Generation with DPLL(T)

The certificate generation algorithm is described in this section. We first describe the abstract rule and implementation issues, then prove some important properties, and finally give a couple of examples.

4.1. The CDPLL(T) Algorithm. We describe here a certificate generation procedure that can be well adapted to the DPLL(T) algorithm. The extension of DPLL(T) with certificate generation is called CDPLL(T).

Definition 8 (certificate refinement). The certificate obtained by appending a certificate item C to the end of the certificate P is denoted by $P \lhd C$.

We use a transition system to model the CDPLL(T) algorithm. Each state is represented as a 3-tuple "$M \parallel F \oplus P$," where M is the literal stack, F the clause set to be satisfied, and P the current certificate. The transition rules in CDPLL(T) are follows.

(i) Decide:

Precondition

(1) l or $\neg l$ occurs in a clause of F;

(2) l is undefined in M,

$$M \parallel F \oplus P \longmapsto M\, l^{\mathsf{d}} \parallel F \oplus P. \quad (10)$$

(ii) UnitPropagate:

Precondition

(1) $M \vDash \neg C$;

(2) l is undefined in M,

$$M \parallel F, C \vee l \oplus P \longmapsto M\, l \parallel F, C \vee l \oplus P. \quad (11)$$

(iii) TheoryPropagate:

Precondition

(1) $M \vDash_{\mathsf{T}} l$;

(2) l or $\neg l$ occurs in F;

(3) l is undefined in M,

$$M \parallel F \oplus P \longmapsto M\, l \parallel F \oplus P. \quad (12)$$

(iv) T-Backjump:

Precondition

(1) $M\, l^{\mathsf{d}} N \vDash \neg C$;

(2) there exists a clause $C' \vee l'$ such that: (a) $F, C \vDash_{\mathsf{T}} C' \vee l'$ and $M \vDash \neg C'$, (b) l' is undefined in M, and (c) l' or $\neg l'$ occurs in F or in $M\, l^{\mathsf{d}} N$;

(3) $R = \{C_1, C_2, \ldots, C_n\}$ is a set of clause items such that: $C_i \in \lfloor P \rfloor$ for all i, and there exists a linear regular resolution from $\{[\![C_1]\!], [\![C_2]\!], \ldots, [\![C_n]\!]\}$ to $C' \vee l'$,

$$M\, l^{\mathsf{d}} N \parallel F, C \oplus P \longmapsto M\, l' \parallel F, C \oplus (P \lhd \mathsf{Res}\,(R)). \quad (13)$$

(v) T-Learn(I): this rule models the clause learning procedure.

Precondition

(1) Each atom of C occurs in F or in M;

(2) $F \vDash C$;

(3) $R = \{C_1, C_2, \ldots, C_n\}$ is a set of clause items such that: $C_i \in \lfloor P \rfloor$ for all i, and there exists a linear regular resolution from $\{[\![C_1]\!], [\![C_2]\!], \ldots, [\![C_n]\!]\}$ to C,

$$M \parallel F \oplus P \longmapsto M \parallel F, C \oplus (P \lhd \mathsf{Res}\,(R)). \quad (14)$$

(vi) T-Learn(II): this rule models the learning of theory lemma.

Precondition

(1) Each atom of C occurs in F or in M;

(2) $\vDash_{\mathsf{T}} C$,

$$M \parallel F \oplus P \longmapsto M \parallel F, C \oplus (P \lhd \mathsf{Lemma}\,(C)). \quad (15)$$

(vii) T-Forget:

Precondition

$F \models_T C$,

$$M \parallel F, C \oplus P \longmapsto M \parallel F \oplus (P \lhd \mathsf{Forget}(C)). \quad (16)$$

(viii) Restart:

Precondition

T

$$M \parallel F \oplus P \longmapsto \emptyset \parallel F \oplus P. \quad (17)$$

(ix) Fail:

Precondition

(1) $M \models \neg C$;

(2) M contains no decision literals;

(3) $R = \{C_1, C_2, \ldots, C_n\}$ is a set of clause items such that: $C_i \in \lfloor P \rfloor$ for all i, and there exists a linear regular resolution from $\{[\![C_1]\!], [\![C_2]\!], \ldots, [\![C_n]\!]\}$ to $\Box$,

$$M \parallel F, C \oplus P \longmapsto \langle \text{Fail} \rangle \oplus (P \lhd \mathsf{Res}(R)). \quad (18)$$

The initial state is $M \parallel F \oplus P_0$ where P_0 is the certificate that defines all initial clauses. When a final state $\langle \text{Fail} \rangle \oplus P$ is reached, P is the obtained certificate for the unsatisfiability judgment.

The major differences between CDPLL(T) and DPLL(T) are as follows. (1) Certificate generation is added. Notice the modifications to rules T-Backjump, T-Learn, T-Forget, and T-Fail. (2) In order to generate well-formed certificates, the original T-Learn rule is split into 2 cases, one for the learning of resolvents (corresponds to clause learning) and the other for the learning of theory lemmas (corresponds to deductions in T). The first three rules are unchanged except the certificate part.

4.2. Implementation of CDPLL(T). In a real implementation of CDPLL(T), additional information need to be collected to construct the certificate items. In this section, we discuss these details.

Lemma 9. *For any reachable state $M \parallel F \oplus P$ in CDPLL(T) and any clause $C \in F$, there exists a clause item $D \in \lfloor P \rfloor$ such that $[\![D]\!] = C$.*

Proof. We prove that by induction. Initially, this property holds because all clauses in F_0 are initial clauses and P_0 consists of all initial clause items (by definition). At each valid transition step, the clause set F and the certificate P may be modified. However, each rule ensures that whenever a clause C is added to F, there is always a clause item D such that $[\![D]\!] = C$ added to P, for example, in the rule T-Learn(I). Furthermore, whenever a clause item is removed from P, the corresponding clause is removed in F, for example, in the rule T-Forget. Thus, the property holds on all valid trace of CDPLL(T). □

4.2.1. Record Reasons. For any literal l in M, it is either added by the Decide rule or forced by a clause or theory lemma. For the latter case, we need to record a clause item which is a deduction of the lemma and from which the literal l can be enforced. We call this item the *reason* for literal l being in M.

Only rules UnitPropagate, TheoryPropagate, and T-Backjump can enforce new literal to M. We discuss in the following the reasons recorded by these three rules.

(i) UnitPropagate: the reason is a certificate item D such that $[\![D]\!] = C \vee l \in F$. By Lemma 9, such item always exists.

(ii) TheoryPropagate: assume that $M = l_1 \wedge l_2 \wedge \cdots \wedge l_m$ and $M \models_T l$; then, the reason is a certificate item D such that $[\![D]\!] = \neg l_1 \vee \neg l_2 \vee \neg \cdots \vee \neg l_m \vee l$.

(iii) T-Backjump: the reason for l' is a certificate item D such that $[\![D]\!] = C' \vee l'$. We will show in the following that such certificate item always exists when T-Backjump is applicable.

For convenience, a subscript is used to denote the reason, for example, $l_{(D)}$ denotes the literal l asserted by the reason $[\![D]\!]$.

4.2.2. Learn Clauses. In rules T-Backjump and Fail, once there is a conflict, a backjump clause needs be learned. Furthermore, if there are some branches that have not been explored, T-Backjump is applied; otherwise, Fail is applied. Each clause learning procedure introduces some new certificate items.

We use implication graph to analyze the clause learning process. Remember that in DPLL all literals are propositional atoms, the implication graph is simply a direct acyclic graph (DAG) where each edge corresponds to a deduction step. In CDPLL(T), the implication graph has two kinds of deductions: propositional deduction (which is similar to DPLL) and theory deduction. Each deduction step in the implication graph is labelled with a reason C. This graph can be viewed as a *generalized implication graph*. Without causing ambiguity, we still call it an implication graph.

In the implication graph of DPLL, each cut containing the conflict literals corresponds to a learned clause (by resolving all the associated clauses of edges in this cut) [4]. No matter which criteria (e.g., 1-UIP) is used, the rule for clause learning is applicable. For CDPLL(T), when considering the generalized implication graphs, the clause learning rule is also applicable.

There are two situations where T-Backjump can be applied. Given a state $M \parallel F \oplus P$, the first situation is that there exists a clause $C \in F$ which is falsified by M, that is, $\not\models M, C$. Then, C is called the *conflict clause*. As in the rule, some backjump clause $C' \vee l'$ should be learned by analyzing the conflict. Usually, $C' \vee l'$ is a resolvent of a linear resolution. A certificate item with resolution type will be learned.

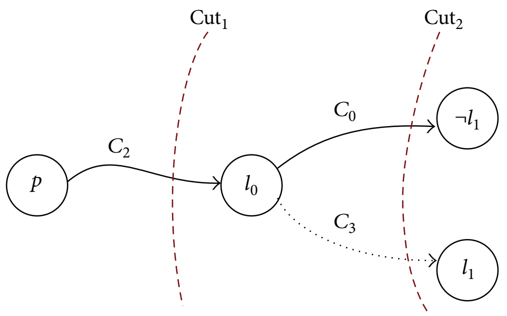

Figure 2: Example of a generalized implication graph.

The other situation is that M itself becomes T-inconsistent. Then, a subset $\{l_1, l_2, \ldots, l_m\}$ of M is unsatisfiable, that is, $\vDash_{\mathsf{T}} \neg l_1 \vee \neg l_2 \vee \cdots \vee \neg l_m$. This is also a conflict clause. It may not belong to F, but it can definitely be learned from F along with some theory lemmas. The fact that $l_1, \ldots, l_m$ leads to conflict is represented in the generalized implication graph. Furthermore, the clause $C' \vee l'$ that is required by the rule T-Backjump can also be learned by generalized conflict clause analysis.

In both situations, the backjump clauses are learned by clause learning. The certificate for unsatisfiability is actually a well-organized collection of these clause items. If Fail is applicable, no literal in M is decision variable; then, the learnt clause is the empty clause which shows $F \vDash_{\mathsf{T}} \Box$.

Example 10. A generalized implication graph is shown in Figure 2, where the literals are as follows: $l_0 : x = y$, $l_1 : f(x) = f(y)$, and p is a propositional literal. Clauses are as follows:

$$\begin{aligned} &C_0 : \neg l_0 \vee \neg l_1 := \neg(x = y) \vee (f(x) = f(y)), \\ &C_1 : l_0 \vee p := x = y \vee p, \\ &C_2 : l_0 \vee \neg p := x = y \vee \neg p. \end{aligned} \tag{19}$$

Assume the current assignment $M = p, l_0, \neg l_1$. Then, there is a conflict between $\neg l_1$ and l_1. In all, there are 3 deductions in the implication graph: $p \rightarrow l_0$ forced by C_2, $l_0 \rightarrow \neg l_1$ by C_0 and $l_0 \rightarrow l_1$ by the theory lemma C_3. Among those reasons, solid lines correspond to existing clauses in F, while dashed lines correspond to theory lemmas. C_3 is actually a theory lemma $l_0 \rightarrow l_1 (x = y \rightarrow f(x) = f(y))$.

In the clause learning procedure, we start from the conflict $(\neg(l_1 \wedge \neg l_1) = \neg l_1 \vee l_1)$ and trace back (apply a sequence of resolutions). If the 1st UIP schema [4] is used, it is possible to learn $\neg p$ or $\neg l_0$ (corresponds to cut$_1$ and cut$_2$, resp.). The resolution steps for learning $\neg p$ are shown in Figure 3. Actually, $\neg p$ is the resolvent of $\{C_3, C_0, C_2\}$ which are exactly the reasons labeled on the edges from the conflict to the cut. Among those, C_0, C_2 are normal clauses, and C_3 is a theory lemma.

Based on the above discussions, the related rules can be interpreted more precisely as follows.

$$\dfrac{\dfrac{C_3\colon \neg l_0 \vee l_1 \qquad C_0\colon \neg l_0 \vee \neg l_1}{\neg l_0}\, l_1 \qquad C_2\colon \neg p \vee l_0}{\neg p}\, l_0$$

Figure 3: Resolution steps for learning $\neg p$.

(i) TheoryPropagate:

$$M \parallel F \oplus P \longmapsto M\, l_{(C)} \parallel F \oplus P. \tag{20}$$

The precondition requires that $M \vDash_{\mathsf{T}} l$. Let M' be a subset of M which entails l (possibly $M' = M$). Let $C = \neg M' \vee l$, then $\vDash_{\mathsf{T}} C$. Furthermore, $M, C \vDash_{\mathsf{T}} l$, which means C is a unit clause under the assignment M. Thus, C is the reason of l.

(ii) UnitPropagate:

$$M \parallel F, C \vee l \oplus P \longmapsto M\, l_{(C \vee l)} \parallel F, C \vee l \oplus P. \tag{21}$$

The precondition requires that $M \vDash \neg C$; thus, $C \vee l$ is a unit clause under M. So, the reason of l is $C \vee l$.

(iii) T-Backjump:

$$M\, l^{\mathsf{d}} N \parallel F, C \oplus P \longmapsto M\, l'_{(C' \vee l')} \parallel F, C \oplus (P \lhd \mathsf{Res}(R)). \tag{22}$$

The precondition requires that there exits some clause $C' \vee l'$ such that $F, C \vDash_{\mathsf{T}} C' \vee l'$ and $M \vDash \neg C'$. The clause $C' \vee l'$ is then used as the reason for l'. As the clause learning is always applicable [9], this clause always exists. If the assignment $M\, l^{\mathsf{d}} N$ is T-consistent, then there must be some conflict clause $C \in F$. As explained above, the clause learning will then be applied, and the learned clause will be $C' \vee l'$. On the other hand, when $M\, l^{\mathsf{d}} N$ is T-inconsistent, clause learning is also applicable in the generalized implication graph and generates a candidate $C' \vee l'$. In both cases, $C' \vee l'$ can be learned by applying linear regular resolutions on a clause set R [4]. Moreover, such R is the union of a subset of reasons in $M\, l^{\mathsf{d}} N$ and a group of theory lemmas.

(iv) Fail:

$$M \parallel F, C \oplus P \longmapsto \langle \mathsf{Fail} \rangle \oplus (P \lhd \mathsf{Res}(R)). \tag{23}$$

The Fail rule is similar to T-Backjump except that the learned clause is the empty clause.

*4.3. Properties of CDPLL(*T*).* The soundness and completeness of CDPLL(T) are proven in this subsection.

Lemma 11. *A well-formed certificate modified by* T-Backjump, T-Learn(I), T-Learn(II), Fail, *or* T-Forget *is still well formed.*

Proof. Each of the 4 rules appends a new item to the certificate. Assume that the certificate P is modified to $P' = P \lhd I$ where I is the appended certificate item. The well-formed property of P' is checked by case studying the rules applied.

(i) For T-Backjump: a certificate item of resolution type is appended. According to the precondition, the certificate items referred by I are already defined in $\lfloor P \rfloor$. So, P' is still well formed.

(ii) For T-Learn(I): similar to the previous case. The dependency of appended certificate item is satisfied.

(iii) For T-Learn(II): a theory lemma item C is appended. It is required that $\vDash_{\mathrm{T}} C$. Thus, P' is still well formed.

(iv) For Fail: similar to the T-Backjump case.

(v) For T-Forget: the rule requires that the forgotten clause is in the clause set. By Lemma 9, we know that there is a certificate item in P which defines C. Thus, the well formedness is ensured. □

With the help of this lemma, it can be proven that CDPLL(T) procedure always generates well-formed certificates.

Lemma 12. *Given any reachable state $M \parallel F \oplus P$ in any CDPLL(T) procedure, P is a well-formed certificate.*

Proof. First of all, the initial certificate P_0 is well formed because it only contains initial items. Secondly, the certificate is only modified in T-Backjump, T-Learn(I), T-Learn(II), Fail, and T-Forget. By Lemma 11, these rules will preserve well formedness. So, following any path of CDPLL(T) will always generate a well-formed certificate. That completes the proof. □

Theorem 13 (soundness). *Given a clause set F_0, for any certificate P, if $M_x \parallel F_x \oplus P_x$ is reachable in a CDPLL(T) procedure, then $M_x \parallel F_x$ is also reachable in a DPLL(T) procedure. Specially, if $\langle Fail \rangle \oplus P$ is reachable in CDPLL(T), then $\langle Fail \rangle$ is also reachable in DPLL(T).*

Proof. For each transition step in CDPLL(T):

$$M \parallel F \oplus P \longrightarrow_{\mathrm{CDPLL(T)}} M' \parallel F' \oplus P', \tag{24}$$

it holds that

$$M \parallel F \longrightarrow_{\mathrm{DPLL(T)}} M' \parallel F'. \tag{25}$$

So, given a CDPLL(T) trace, a trace in DPLL(T) can be obtained by removing the certificate part. If $M_x \parallel F_x \oplus P_x$ is reachable in CDPLL(T) so is $M_x \parallel F_x$ in DPLL(T). □

Theorem 14 (completeness). *Given a clause set F_0, if the state $M_x \parallel F_x$ is reachable in a DPLL(T) procedure, then there must be a certificate P_x such that $M_x \parallel F_x \oplus P_x$ is reachable in a DPLL(T) procedure. Furthermore, if $\langle Fail \rangle$ is reachable in a DPLL(T) procedure, then $\langle Fail \rangle \oplus P_x$ is reachable in a CDPLL(T) and $\square \in \lfloor P_x \rfloor$.*

Proof. For rules Decide, TheoryPropagate, UnitPropagate, T-Learn(II), T-Forget and Restart, the preconditions are equal to those in CDPLL(T). So, if there is a transition in DPLL(T) labelled with one of these rules, there could also be the same transition in CDPLL(T).

For other rules, T-Backjump, T-Learn(I), and Fail, we need to prove that: if $M_x \parallel F_x$ is reachable, then there is some certificate P_x such that $M_x \parallel F_x \oplus P_x$ is reachable. We prove that by induction. Initially, this condition holds because the initial state $M_0 \parallel F_0$ corresponds to $M_0 \parallel F_0 \oplus P_0$. Inductively, if

$$M \parallel F \longrightarrow_{\mathrm{DPLL(T)}} M' \parallel F' \tag{26}$$

is a possible transition in DPLL(T), and there is some P such that $M \parallel F \oplus P$ is reachable in CDPLL(T).

(i) If it is labelled with T-Learn(I), because the learned clause is from clause learning, and clause learning corresponds to linear regular resolution. It is always possible that necessary theory lemmas in the implication graph are learned at first by Learn(II) and get to a state $M \parallel F \oplus P''$, on which the preconditions of Learn(I) are satisfied. Then, $M' \parallel F' \oplus P'$ is reached.

(ii) If it is labelled with T-Backjump or Fail, the situation is similar. □

Our approach can be well adapted to any DPLL(T)-based SMT solver. It is sound and complete regardless of the theory learning scheme used or the order of the decision procedure. It works well as long as clause learning is based on the generalized implication graph.

4.4. Examples. A couple of examples are discussed in this subsection. The first example is given as a CNF formula on propositional variables. In this case, the SMT problem reduces to an SAT problem. No theory deduction is involved in this example; so, we can get a general idea of the structure of certificates.

Example 1. Consider the following clause set:

$$\begin{aligned} C_1 &= \neg l_1 \vee \neg l_2 \vee \neg l_3, \\ C_2 &= \neg l_1 \vee \neg l_2 \vee l_3, \\ C_3 &= \neg l_1 \vee l_2, \\ C_4 &= l_1 \vee \neg l_2, \\ C_5 &= l_1 \vee l_2. \end{aligned} \tag{27}$$

Let $F_0 = \{C_1, C_2, C_3, C_4, C_5\}$; assume that P_0 defines everything in F_0 as initial clause; then, a possible CDPLL(T) procedure is shown in Table 2, where "~" means this field is the same as that in the previous row.

In step 4, $C_2 = \neg l_1 \vee \neg l_2 \vee l_3$ is falsified. By analysing the implication graph, a clause is learned from C_1, C_2, C_3. Among'the referred clauses, C_2 is a falsified clause, and C_1, C_3 are in the reasons. In step 7, C_4 is falsified, but there is no decision variable. By analysing the implication graph, we can find a resolution from C_4, C_5, C_6 to the empty clause. Among the referred clauses, C_4 is a falsified clause, and C_5, C_6 are in the reasons.

Table 2: The CDPLL(T) procedure for Example 1.

No.	$M\|$	F	$\oplus P$	Reason	Explanation
1	$\emptyset\|$	F_0	$\oplus P_0$	$\emptyset$	Initial state
2	$l_1^{\mathrm{d}}\|$	$\sim$	$\oplus \sim$	$\emptyset$	Decide
3	$l_1^{\mathrm{d}}, l_2\|$	$\sim$	$\oplus \sim$	$l_2 \mapsto C_3$	UnitPropagate
4	$l_1^{\mathrm{d}}, l_2, \neg l_3\|$	$\sim$	$\oplus \sim$	$l_2 \mapsto C_3$ $\neg l_3 \mapsto C_1$	UnitPropagate now C_2 is falsified
5	$\neg l_1\|$	F_0 $C_6 = \neg l_1$	$\oplus \begin{bmatrix} P_0 \lhd \\ \mathsf{Res}\{C_1, C_2, C_3\} = C_6 \end{bmatrix}$	$\neg l_1 \mapsto C_6$	$C_6 = \neg l_1$ is learned T-Backjump
6	$\neg l_1, \neg l_2\|$	$\sim$	$\oplus \sim$	$\neg l_1 \mapsto C_6$ $\neg l_2 \mapsto C_4$	UnitPropagate
7	$\langle\text{Fail}\rangle\|$	$\sim$	$\oplus \begin{bmatrix} P_0 \lhd \\ \mathsf{Res}\{C_1, C_2, C_3\} = C_6 \\ \mathsf{Res}\{C_4, C_5, C_6\} = \square \end{bmatrix}$	$\emptyset$	Fail

Example 2. Consider the background theory $\mathsf{T}_{\mathrm{EUF}}$ and a clause set F containing the following:

$$\begin{aligned} &(C_1)\ a = b \vee g(a) \neq g(b), \\ &(C_2)\ h(a) \neq h(c) \vee p, \\ &(C_3)\ g(a) = g(b) \vee \neg p, \\ &(C_4)\ h(a) = h(c) \vee a = c, \\ &(C_5)\ f(a) \neq f(b), \\ &(C_6)\ b = c. \end{aligned} \tag{28}$$

Its boolean structure is

$$\begin{aligned} C_1 &= l_1 \vee \neg l_5, \\ C_2 &= \neg l_6 \vee p, \\ C_3 &= l_5 \vee \neg p, \\ C_4 &= l_6 \vee l_2, \\ C_5 &= \neg l_4, \\ C_6 &= l_3, \end{aligned} \tag{29}$$

where

$$\begin{aligned} &l_1\text{: } a = b, \\ &l_2\text{: } a = c, \\ &l_3\text{: } b = c, \\ &l_4\text{: } f(a) = f(b), \\ &l_5\text{: } g(a) = g(b), \\ &l_6\text{: } h(a) = h(c). \end{aligned} \tag{30}$$

A possible CDPLL(T) procedure is shown in Table 3, where the referred certificates are as follows:

$$\begin{aligned} P_1 &= \begin{bmatrix} P_0 \lhd \\ \mathsf{Lemma}\,(\neg l_1 \vee l_4) = C_7 \end{bmatrix}, \\ P_2 &= \begin{bmatrix} P_0 \lhd \\ \mathsf{Lemma}\,(\neg l_1 \vee l_4) = C_7 \\ \mathsf{Lemma}\,(l_2 \wedge l_3 \longrightarrow l_1) = C_8 \end{bmatrix}, \\ P_3 &= \begin{bmatrix} P_0 \lhd \\ \mathsf{Lemma}\,(\neg l_1 \vee l_4) = C_7 \\ \mathsf{Lemma}\,(l_2 \wedge l_3 \longrightarrow l_1) = C_8 \\ \mathsf{Res}\,\{C_8, C_7, C_4, C_6, C_2, C_3, C_1\} = \square \end{bmatrix}. \end{aligned} \tag{31}$$

In step 4, M becomes T-inconsistent. The generalized implication graph is shown in Figure 4. A theory lemma $C_7 = \neg l_1 \vee l_4$ is learned (instantiated the monotonicity property of the equality) firstly, then the backjump rule is applicable. In steps 11 and 12, M becomes T-inconsistent again. A theory lemma $C_8 = l_2 \wedge l_3 \rightarrow l_1$ is then learned (transitivity of equality), and then clause learning is performed which learns the empty clause. The implication graph is shown in Figure 5.

5. The Certificate Checking

Lemma 15. *For any initial clause set F_0, given a well-formed certificate P and $0 \le i \le |P|$, if $P[i]$ is a certificate item that defines a clause, then $F_0 \vDash_{\mathsf{T}} [\![P[i]]\!]$.*

Proof. We prove that by induction. The base case is easy to prove because for any initial item $P[i]$ that defines a clause, $P[i] \in F_0$. Therefore, it is trivial that $F_0 \vDash [\![P[i]]\!]$. Thus, $F_0 \vDash_{\mathsf{T}} [\![P[i]]\!]$.

(i) For resolution certificate items, assume that $P[i] = \mathsf{Res}\{C_1, C_2, \ldots, C_n\}$. By definition, all C_i are all defined in $\lfloor P \rfloor_{i-1}$. Then by the inductive hypothesis,

Table 3: A possible CDPLL(T) procedure.

No.	$M\|$	F	$\oplus P$	Reason	Explanation
1	$\emptyset\|$	F_0	$\oplus P_0$	$\emptyset$	Initial state
2	$\neg l_4\|$	~	$\oplus$ ~	$\neg l_4 \mapsto C_5$	UnitPropagation
3	$\neg l_4, l_3\|$	~	$\oplus$ ~	$\neg l_4 \mapsto C_5$ $l_3 \mapsto C_6$	UnitPropagation
4	$\neg l_4, l_3, l_1^d\|$	~	$\oplus$ ~	~	Decide M becomes T-inconsistent
5	~$\|$	$F_0 \cup \{C_7\}$	$\oplus P_1$	~	T-Learn(II), C_7 is learned
6	$\neg l_4, l_3, \neg l_1\|$	~	$\oplus$ ~	$\neg l_4 \mapsto C_5$ $l_3 \mapsto C_6$ $\neg l_1 \mapsto C_7$	T-Backjump
7	$\neg l_4, l_3, \neg l_1, \neg l_5\|$	~	$\oplus$ ~	$\neg l_4 \mapsto C_5$ $l_3 \mapsto C_6$ $\neg l_1 \mapsto C_7$ $\neg l_5 \mapsto C_1$	UnitPropagation
8	$\neg l_4, l_3, \neg l_1, \neg l_5, \neg p\|$	~	$\oplus$ ~	$\neg l_4 \mapsto C_5$ $l_3 \mapsto C_6$ $\neg l_1 \mapsto C_7$ $\neg l_5 \mapsto C_1$ $\neg p \mapsto C_3$	UnitPropagation
9	$\neg l_4, l_3, \neg l_1, \neg l_5, \neg p, \neg l_6\|$	~	$\oplus$ ~	$\neg l_4 \mapsto C_5$ $l_3 \mapsto C_6$ $\neg l_1 \mapsto C_7$ $\neg l_5 \mapsto C_1$ $\neg p \mapsto C_3$ $\neg l_6 \mapsto C_2$	UnitPropagation
10	$\neg l_4, l_3, \neg l_1, \neg l_5, \neg p, \neg l_6, l_2\|$	~	$\oplus$ ~	$\neg l_4 \mapsto C_5$ $l_3 \mapsto C_6$ $\neg l_1 \mapsto C_7$ $\neg l_5 \mapsto C_1$ $\neg p \mapsto C_3$ $\neg l_6 \mapsto C_2$ $l_2 \mapsto C_4$	UnitPropagation M becomes T-inconsistent
11	~$\|$	$F_0 \cup \{C_7, C_8\}$	$\oplus P_2$	~	T-Learn(II), C_8 is learned
12	~$\|$	$F_0 \cup \{C_7, C_8\}$	$\oplus P_3$	~	Fail, $\square$ derived

we know $F_0 \vDash_T [\![C_i]\!]$. By the property of resolution, it is true that $F_0 \vDash_T [\![P[i]]\!]$.

(ii) For theory lemma, it is true that $\vDash_T [\![P[i]]\!]$. So, $F_0 \vDash_T [\![P[i]]\!]$. □

Theorem 16. *Given a clause set F_0, if a certificate P for F_0 is well formed and $\square \in \lfloor P \rfloor$, then F_0 is unsatisfiable.*

Proof. By Lemma 15, we know that $F_0 \vDash_T \square$ in this case. □

By Theorem 16, certificate checking is actually checking whether the certificate is well formed and contains the empty clause. Checking the well formedness of a certificate can directly follow Definition 7. Only one traverse from the first item to the last one is needed. Moreover, the following properties make the checking process even easier.

(i) Only checking for theory lemma is performed in theory T. Once a theory lemma is proved valid, it can be treated as a propositional clause.

(ii) For each individual proof rule, we have a dedicated checker to check if the theory lemma is an instance of the rule. In this way, the certificate checker is extensible. Given a new proof rule, we need only to add the corresponding lemma checker. Notice that each proof rule defines an atomic deduction step; the checking effort is much less than solving a constraint

Table 4: Experiment results.

Case	Input		CDPLL (T)		Certificate			Checking	
	Lit	Clause	Learnt	Time	Lemmas	Res	Forget	Mem	Time
1	2	3	0	0.01 s	2	2	0	4	0.01 s
2	10	15	1	0.01 s	4	8	3	10	0.00 s
3	17	25	3	0.01 s	8	14	4	17	0.00 s
4	66	95	1069	0.08 s	1024	550	490	66	0.05 s
5	87	125	9537	0.71 s	8192	4146	4043	87	0.18 s
6	1157	2611	5197	1.39 s	22383	3711	2635	1051	0.23 s
7	1582	2173	16735	21.30 s	859886	8529	7438	1148	1.16 s
8	811	1037	5164	5.80 s	248159	4286	3580	555	0.41 s
9	4687	9433	759	1.00 s	1021	751	430	617	0.03 s
10	167	308	60	0.04 s	934	77	48	79	0.02 s
11	447	883	1464	0.53 s	15894	965	995	399	0.09 s
12	3930	6584	3046	2.22 s	8555	2468	1617	1639	0.14 s
13	3362	5154	2310	1.68 s	6204	1916	1251	1467	0.13 s
14	4615	7036	3244	3.71 s	9521	2308	1633	1776	0.13 s
15	2422	3726	828	0.43 s	587	616	509	428	0.03 s
16	5226	8614	9781	7.85 s	6305	3457	1723	2199	0.12 s
17	3073	4821	128729	87.65 s	74289	85711	84269	5899	5.16 s
18	2415	3710	2511	1.29 s	2165	1781	1377	1367	0.27 s
19	4151	6492	32991	21.26 s	26011	21015	20792	4497	1.24 s

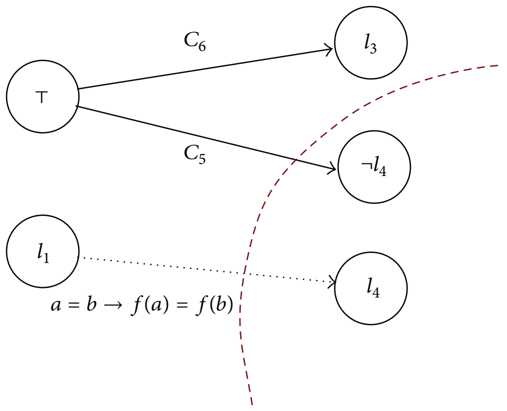

Figure 4: The implication graph in step 4.

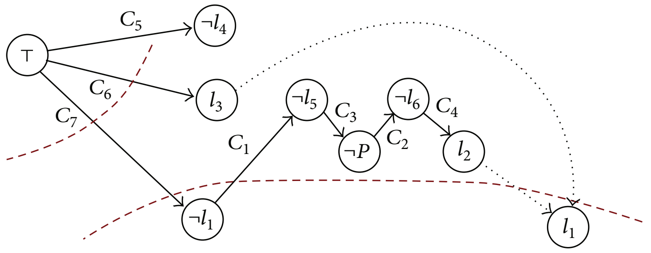

Figure 5: The implication graph in steps 11 and 12.

in the theory T. For instance, only pattern matching is needed to check if a theory lemma is an instance of the monotonicity property of equality.

6. Experiments

Two criteria are adopted to assess our certificate approach: the overhead for generating certificates, and the cost for certificate checking. We implemented an SMT solver aCiNO based on the CDPLL(T) algorithm. It uses the Nelson-Oppen framework to solve combined theory. Currently, $\mathsf{T}_{\mathrm{EUF}}$ and $\mathsf{T}_{\mathrm{LRA}}$ are considered. The experiments are carried out on a machine with a E7200 dual core CPU (2.53 GHz per core) and 2.0 GB RAM.

All test cases are taken from SMT-LIB [22]. Our approach is tested on 19 unsatisfiable instances from different folders (in order to test it on different kinds of problem instances). Experiment results are shown in Table 4.

In Table 4, the first column-group describes the scale of the input problem. The input is transformed to its equivalent conjunctive normal form, and the number of literals and clauses are counted. In the CDPLL(T) procedure, once a closed branch is encountered, a new clause will be learned. The number of learned clauses is listed in the second column-group. This number is equal to that in DPLL(T) since certificate generation will not affect the search procedure of the SMT problem. As in the DPLL(T) algorithm, the number

Table 5: Overhead of our approach.

Test Case	With cert (sec.)	Without cert (sec.)	Overhead (%)
NEQ.NEQ004_size4.smt2	0.63	0.60	4.62
NEQ.NEQ004_size5.smt2	19.87	18.27	8.74
PEQ.PEQ002_size5.smt2	5.29	4.85	9.07
PEQ.PEQ020_size5.smt2	1.29	1.20	7.47
QG-classification.loops6.dead_dnd001.smt2	1.34	1.20	11.94
QG-classification.loops6.gensys_brn004.smt2	1.06	0.95	11.81
QG-classification.loops6.gensys_icl001.smt2	2.59	2.22	16.62
QG-classification.loops6.iso_brn005.smt2	0.18	0.14	27.33
QG-classification.loops6.iso_icl030.smt2	3.32	3.05	8.68
QG-classification.qg5.dead_dnd007.smt2	35.62	34.55	3.09
QG-classification.qg5.gensys_brn007.smt2	0.74	0.71	4.52
QG-classification.qg5.gensys_icl100.smt2	14.29	13.60	5.07
SEQ.SEQ004_size5.smt2	0.68	0.54	26.93
SEQ.SEQ050_size3.smt2	0.19	0.18	7.03

of clause learning can be quite large (e.g., the 17th case). The time used in CDPLL(T) is also presented. The third column-group summarises the generated certificates. It is obvious that the number of theory lemma items is usually very large. The column of "Forget" is the number of forget items that forgets an initial or resolution certificate item. All theory lemmas are eventually forgotten, these items are not counted here. For SMT problem, this is not surprising since the major work of SMT solvers is in reasoning in the background theory. The resources and time required to check the certificates are listed in the *Checking* column-group. All certificates are checked to be well formed. The "Mem" column is not the size of memory consumed but the maximal number of clause items that are stored in the memory during checking. Also, the time for certificate checking is listed besides.

Regarding the certificate checking, the number of initial clauses and memory consumption is compared in Figure 6. It is rather interesting that although the certificate itself could be exponentially large with respect to the input problem, the memory consumption will not grow in the same way. The reason is that we have explicitly forgotten many items. In particular, many theory lemma are referred locally. They are only available in a short period of time, after which they are forgotten. With the technique of forgetting items, the certificate checking becomes more efficient. This is also supported by data from the "Time" column in Table 4.

Towards the overhead of our approach, the experiment is shown in Table 5. Tested cases are also from SMT-LIB. In order to suppress the inaccurate measurement of time, we intentionally selected time consuming cases (e.g., longer than 0.01 sec). The maximal overhead is about 27.33%, and the average overhead is about 10.5%. It is much smaller compared to other approaches [17, 21].

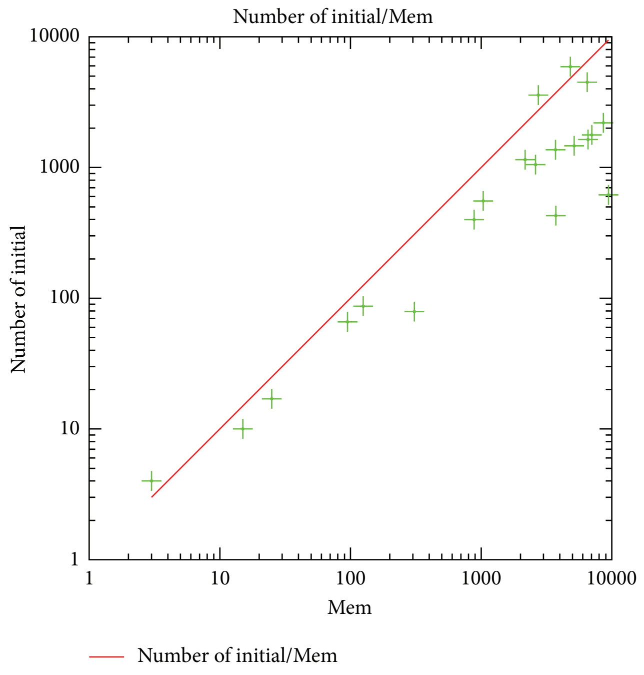

Figure 6: Comparison between the number of initial clauses and memory usage.

7. Conclusion

In this paper, a unified certificate framework for quantifier-free SMT instances was presented. The certificate format is simple, clean, and extensible to other background theories. The certificate generation procedure can be easily integrated to any DPLL(T)-based SMT solver. Soundness and completeness of the extension of DPLL(T) with the certificate generation procedure were established. Experimental results show that our certificate framework outperforms others in both certificate generation and certificate checking.

Acknowledgments

This work was supported by the Chinese National 973 Plan under Grant no. 2010CB328003; the NSF of China under Grants nos. 61272001, 60903030, and 91218302; the Chinese

National Key Technology R&D Program under Grant no. SQ2012BAJY4052; the Tsinghua University Initiative Scientific Research Program.

References

[1] C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump, "6 Years of SMT-COMP," *Journal of Automated Reasoning*, vol. 50, no. 3, pp. 243–277, 2013.

[2] C. Barrett and C. Tinelli, "CVC3," in *Proceedings of the 19th International Conference on Computer Aided Verification*, pp. 298–302, Springer, July 2007.

[3] L. Moura and N. Bjrner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, Berlin, Germany, 2008.

[4] P. Beame, H. Kautz, and A. Sabharwal, "Understanding the power of clause learning," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1194–1201, Citeseer, Acapulco, Mexico, August 2003.

[5] J. Silva, "An overview of backtrack search satisfiability algorithms," in *Proceedings of the 5th International Symposium on Artificial Intelligence and Mathematics*, Citeseer, January 1998.

[6] P. Beame and T. Pitassi, "Propositional proof complexity: past, present, and future," in *Bulletin of the European Association for Theoretical Computer Science*, The Computational Complexity Column, pp. 66–89, 1998.

[7] S. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pp. 151–158, ACM, Shaker Heights, Ohio, USA, 1971.

[8] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, pp. 394–397, 1962.

[9] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: from an abstract davis—putnam—logemann—loveland procedure to DPLL(T)," *Journal of the ACM*, vol. 53, no. 6, Article ID 1217859, pp. 937–977, 2006.

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Proceedings of the 38th Design Automation Conference*, pp. 530–535, June 2001.

[11] N. Een and N. Sorensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, pp. 333–336, Springer, Berlin, Germany, 2004.

[12] A. Biere, "PicoSAT essentials," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, article 45, 2008.

[13] M. Boespug, Q. Carbonneaux, and O. Hermant, "The $\lambda\pi$-calculus modulo as a universal proof language," in *Proceedings of the 2nd International Workshop on Proof Exchange for Theorem Proving (PxTP '12)*, June 2012.

[14] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli, "SMT proof checking using a logical framework," *Formal Methods in System Design*, vol. 42, no. 1, pp. 91–118.

[15] D. Deharbe, P. Fontaine, B. Paleo et al., "Quantifier inference rules for SMT proofs," 1st International Workshop on Proof eXchange for Theorem Proving (PxTP '11), 2011.

[16] P. Fontaine, J. Y. Marion, S. Merz, L. Nieto, and A. Tiu, "Expressiveness + automation + soundness: towards combining SMT solvers and interactive proof assistants," in *Tools and Algorithms for the Construction and Analysis of Systems*, H. Hermanns and J. Palsberg, Eds., vol. 3920 of *Lecture Notes in Computer Science*, pp. 167–181, Springer, Berlin, Germany, 2006.

[17] D. Oe, A. Reynolds, and A. Stump, "Fast and flexible proof checking for SMT," in *Proceedings of the 7th International Workshop on Satifiability Modulo Theories (SMT '09)*, pp. 6–13, ACM, August 2009.

[18] A. Cimatti, A. Griggio, and R. Sebastiani, "A simple and exible way of computing small unsatisfiable cores in SAT modulo theories," in *Theory and Applications of Satisfiability Testing SAT*, J. Marques-Silva and K. Sakallah, Eds., vol. 4501 of *Lecture Notes in Computer Science*, pp. 334–339, Springer, Berlin, Germany, 2007.

[19] Y. Ge and C. Barrett, "Proof translation and SMT-LIB benchmark certification: a preliminary report," in *Proceedings of International Workshop on Satisfiability Modulo Theories (SMT '08)*, August 2008.

[20] S. Bohme, "Proof reconstruction for Z3 in Isabelle/HOL," in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT '9)*, August 2009.

[21] M. Moskal, "Rocket-fast proof checking for SMT solvers," in *Tools and Algorithms For the Construction and Analysis of Systems*, C. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*, pp. 486–500, Springer, Berlin, Germany, 2008.

[22] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard: version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, Edinburgh, UK, 2010.