DOI 10.1007/s00165-009-0105-z
BCS © 2009
Formal Aspects of Computing (2009) 21: 485–494

Formal Aspects of Computing

A framework for Satisfiability Modulo Theories

Daniel Kroening[1] and Ofer Strichman[2]

[1] Computing Laboratory, Oxford University, Oxford, UK
[2] Information Systems Engineering, IE, Technion, 32000 Haifa, Israel. E-mail: ofers@ie.technion.ac.il

Abstract. We present a unifying framework for understanding and developing SAT-based decision procedures for Satisfiability Modulo Theories (SMT). The framework is based on a reduction of the decision problem to propositional logic by means of a deductive system. The two commonly used techniques, *eager* encodings (a direct reduction to propositional logic) and *lazy* encodings (a family of techniques based on an interplay between a SAT solver and a decision procedure) are identified as special cases. This framework offers the first generic approach for eager encodings, and a simple generalization of various lazy techniques that are found in the literature.

1. Introduction

Decision procedures for checking satisfiability of formulas are widely used in formal verification, as the rich set of theories supported by Satisfiability Modulo Theories (SMT) solvers permits convenient modeling of many verification problems.

Literally all competitive decision procedures in this domain rely on propositional SAT solvers. The renewed interest in this field in the last decade, especially since the introduction of Chaff [MMZ+01] in 2001 and the growing interest of the verification community in SAT-solving in general, led to the development of a large set of SAT-based SMT-solvers. As of 2005, these solvers compete in the annual SMT-COMP competition [BdMS05]. The terminology used in the literature distinguishes between two main strategies for using the SAT solver, namely the *eager* and *lazy* encodings, while recognizing that these two strategies are on the same continuum. The eager encoding is simply a reduction from the input formula to a formula in propositional logic. Many such reductions from various theories were suggested through the years (see, for example, [PRSS99, BV00] for equality logic with uninterpreted functions and [Str02, SSB02] for linear arithmetic and its restriction to difference logic). The state-of-the-art solvers for bit-vector arithmetic employ pre-processing techniques and an eager encoding into propositional logic.

On the other hand, the basic lazy approach is an iterative procedure that alternates between a SAT solver and a decision procedure for the underlying theory, which roughly works as follows:

1. Substitute each predicate in the input formula φ with a Boolean variable, to obtain a propositional formula φ_{sk}, the *Boolean skeleton* of φ.
2. If φ_{sk} is unsatisfiable, return 'Unsatisfiable'.
3. Make a decision and apply Boolean constraint propagation. Backtrack if necessary, until reaching a non-conflicting partial assignment α.

Correspondence and offprint requests to: O. Strichman, E-mail: ofers@ie.technion.ac.il
Supported by the European Union as part of the FP7 STREP MOGENTES.

4. Check if $\hat{Th}(\alpha)$, the conjunction of theory-literals corresponding to α, is satisfiable. This step can be performed with a decision procedure for a conjunction of predicates in the underlying theories.
5. If $\hat{Th}(\alpha)$ is inconsistent, conjoin φ_{sk} with a clause corresponding to the negation of α and go to Step 2.
6. If α is a full assignment, return 'Satisfiable'. Otherwise, go to Step 3.

There are numerous improvements over this basic algorithm, only some of which we discuss in this paper. The first tools based on this idea were developed and published almost simultaneously in 2002: CVC [SBD02], ICS- SAT [FORS01], MATHSAT [ABC+02], and a little later VERIFUN [FJOS03]. Since the introduction of this framework, it became main-stream and at least ten other solvers based on the same principles have been developed and published. In fact, all tools that participated in the SMT-COMP competitions in 2005-2008 belong to this category of solvers. One of the attractive features of this framework is its generality, especially in the DPLL($\mathcal{T}$) variant [GHN+04]: given an arbitrary decision procedure for a conjunction of predicates in some theory, only a simple interface is needed for integrating it into a DPLL($\mathcal{T}$) solver.

In this paper, we present a theoretical framework for understanding and developing decision procedures for SMT. The framework is based on a reduction to propositional logic, starting from a deductive decision procedure for the (potentially combined) input theory. More specifically, given a theory $\mathcal{T}$ for which there is a sound and complete deductive system, we describe a template for using this system to construct a propositional encoding of formulas in $\mathcal{T}$. Through variations on this template we show how it generalizes previously published work, including both the eager and lazy approaches. Our framework contributes in several ways:

- It offers the first generic approach for eager encodings,[1] as explained and demonstrated in Sect. 3. So far, eager encodings were designed in an ad hoc way for each input theory, which is considered one of the flaws of this approach in comparison to lazy encodings.
- It offers a generalization of most of the published variants of the lazy technique, including DPLL($\mathcal{T}$). Soundness and completeness of such procedures become easy to show. This is the subject of Sect. 4.
- It presents the eager and lazy approaches as variations of the same principle, which clarifies the relationship between them.

The framework, however, does not offer a general method for doing things faster, although in Sect. 5 we point to research directions in which it can be exploited for this purpose as well.

2. Propositional encodings with proofs

For simplicity of presentation, we focus on quantifier-free fragments of first-order theories, and assume that the formulas are given in negation normal form.

Definition 1 (Propositional encoding) Let φ denote a formula in an arbitrary theory, and let φ_P denote a propositional formula. The formula φ_P is called a *propositional encoding* of φ iff φ_P and φ are equisatisfiable (i.e., φ_p is satisfiable iff φ is).

Our generic construction of propositional encodings relies on deriving propositional formulas that represent the antecedent/consequent relations of deductive proofs.

2.1. Encoding proofs

A deductive proof is constructed using a pre-defined set of *proof rules*, which we assume to be sound. A proof rule consists of a set of antecedents $A_1, \ldots, A_k$, which are the premises that have to hold for the rule to be applicable, and a consequent C. Proof rules without antecedents are called *axioms*. A formalization of proofs as a set of proof steps follows.

Definition 2 (Proof step) A *proof step* s is a triple $(Rule, Consequent, Antecedent)$, where $Rule$ is a proof rule, $Consequent$ a proposition, and $Antecedent$ a (possibly empty) set of antecedents $A_1, \ldots, A_k$.

[1] We ignore here the generic option of reducing decision problems to SAT through a universal device such as a Turing machine, as this option is commonly considered impractical.

Definition 3 (Proof) A *proof* $P = \{s_1, \ldots, s_n\}$ is a set of proof steps in which the antecedence relation is acyclic.

Every proof can be associated with a proof graph, which is a DAG in which the nodes correspond to the steps, and there is an edge (x, y) if and only if the consequent of x represents an antecedent of step y.

We associate a new propositional variable with each unique proposition that is not a (propositional) variable in a given proof. This variable is called the *propositional encoder* of the proposition. Let $e(p)$ denote the propositional encoder of a proposition p. Using such encoders we define:

Definition 4 (Proof-step encoder) Let $s = (Rule, Consequent, Antecedent)$ denote a proof step, and let $Antecedent = \{A_1, \ldots, A_k\}$ be the set of antecedents of s. The *proof-step constraint* $psc(s)$ of s is a constraint enforcing that the antecedents imply the consequent of s:

$$psc(s) := \left(\bigwedge_{i=1}^{k} A_i \right) \Rightarrow Consequent. \tag{1}$$

We can now obtain the constraint for a whole proof by simply conjoining the constraints for all its steps.

Definition 5 (Proof constraint) Let $P = \{s_1, \ldots, s_n\}$ denote a proof. The *proof constraint* $\hat{P}$ induced by P is the conjunction of the constraints induced by its steps:

$$\hat{P} := \bigwedge_{i=1}^{n} psc(s_i). \tag{2}$$

The encoding of a proof constraint $\hat{P}$, denoted $e(\hat{P})$, is obtained by replacing each literal in $\hat{P}$ with its encoder. If a proof P is empty, the convention is that $\hat{P} = \text{TRUE}$ and $e(\hat{P}) = \text{TRUE}$.

2.2. Complete proofs

Our framework uses the *propositional skeleton* of the input formula φ:

Definition 6 (Propositional skeleton) Let $lit(\varphi)$ denote the set of literals in a given formula φ. The ith distinct literal in φ is denoted by $lit_i(\varphi)$. The *propositional skeleton* φ_{sk} of a formula φ is obtained by replacing each literal $a \in lit(\varphi)$ with its propositional encoder $e(a)$, i.e., a new Boolean identifier.

Two comments about this definition:

Encoding literals We encode literals rather than variables according to this definition. This is only a matter of convenience, for presenting the arguments later on. In practice one may encode the variables, as most SMT solvers indeed do.

Boolean literals We can either assume that Boolean literals are treated as any other (theory) literal and hence encode them with new variables, or leave them as is in the Boolean skeleton. The latter option complicates our definitions and thus, for simplicity, we either assume the former option or assume that there are no such variables in the input formula. The variables of φ_{sk}, thus, are all propositional encoders.

Obviously, if φ is satisfiable, then so is φ_{sk}. A stronger claim is the following:

Theorem 1 If φ is satisfiable, then for any proof P, $\varphi_{sk} \wedge e(\hat{P})$ is satisfiable.

Theorem 1 is useful if we find a proof P such that $\varphi_{sk} \wedge e(\hat{P})$ is unsatisfiable. In such a case, the theorem implies the unsatisfiability of φ. In other words, we would like to restrict ourselves to proofs with the following property:

Definition 7 (Complete proof) A proof P is called *complete with respect to* φ if $\varphi_{sk} \wedge e(\hat{P})$ and φ are equisatisfiable.

Note that Theorem 1 implies that if the formula is satisfiable, then any proof is complete. Our focus is then on unsatisfiable formulas.

Let Σ denote a signature of some theory. We now discuss the question of how to obtain suitable proofs for unsatisfiable Σ-formulas.

$$\frac{}{a < succ^i(a)}\ (\text{Ordering I}) \qquad \frac{x < y \qquad y < x}{\text{FALSE}}\ (\text{Ordering II})$$

$$\frac{x \neq x}{\text{FALSE}}\ (\text{Eq-Contradiction}) \qquad \frac{x = a \qquad P}{P[x/a]}\ (\text{Substitution})$$

Fig. 1. Several inference rules. Ordering I is an axiom schema, which uses $succ^i(a)$ to denote the ith successor, $i > 0$, of a.

	Consequent	*Rule*	$e(psc(s))$
1.	$x = 5$	premise	
2.	$x \neq 5$	premise	
3.	$x < 0$	premise	
4.	$5 < 0$	Substitution, 1, 3	$e(x = 5) \wedge e(x < 0) \implies e(5 < 0)$
5.	$0 < 5$	Ordering I ($i = 5$)	$e(0 < 5)$
6.	FALSE	Ordering II, 4, 5	$e(5 < 0) \wedge e(0 < 5) \implies$ FALSE
7.	$5 \neq 5$	Substitution, 1, 2	$e(x = 5) \wedge e(x \neq 5) \implies e(5 \neq 5)$
8.	FALSE	Eq-Contradiction, 7	$e(5 \neq 5) \implies$ FALSE

Fig. 2. Proof of unsatisfiability of $\varphi:\ x = 5 \wedge (x < 0 \vee x \neq 5)$, using the rules in Fig. 1. The only premises are the literals in the formula. The proof steps are annotated in the right column with the constraints that they induce.

Theorem 2 Given a sound and complete deductive decision procedure for a conjunction of Σ-literals, there exists an algorithm for deriving a complete proof for every Σ-formula.

Proof (*sketch*) Let φ' be the DNF representation of a Σ-formula φ. Let $DP_{\mathcal{T}}$ be a deductive, sound, and complete decision procedure for a conjunction of Σ-literals. We use $DP_{\mathcal{T}}$ to prove each of the conjunctions in φ'. The union of the steps in these proofs (together with a proof step for case-splitting) constitutes a complete proof for φ'.□

The goal, however, is to find complete proofs with smaller (practical) complexity than performing such splits (there is no point in propositional encoding if the encoding itself is as complex as performing the proof directly). Our strategy is to find deductive proofs that begin from the literals of the input formulas, while leaving it for the SAT solver to deal with the Boolean structure.

Example 1 Consider the unsatisfiable formula

$$\varphi:\ x = 5 \wedge (x < 0 \vee x \neq 5). \tag{3}$$

The skeleton of φ is:

$$\varphi_{sk} = e(x = 5) \wedge (e(x < 0) \vee e(x \neq 5)). \tag{4}$$

(Clearly, an efficient implementation would encode the first and third literals with one variable and different phases. It is more convenient for us to present it this way because it simplifies the presentation later on.)

Using the proof rules in Fig. 1, we show a contradiction using the proof P, which appears in Fig. 2. Note that P uses only literals as antecedents. The proof constraint $e(\hat{P})$ is:

$$\begin{array}{ll} & (e(x = 5) \wedge e(x < 0) \Rightarrow e(5 < 0)) \\ \wedge & (e(0 < 5)) \\ \wedge & (e(5 < 0) \wedge e(0 < 5) \Rightarrow \text{FALSE}) \\ \wedge & (e(x = 5) \wedge e(x \neq 5) \Rightarrow e(5 \neq 5)) \\ \wedge & (e(5 \neq 5) \Rightarrow \text{FALSE}). \end{array} \tag{5}$$

The conjunction of φ_{sk} and $e(\hat{P})$ (Eq. 4 and 5) is unsatisfiable, and thus, due to Theorem 1, φ is unsatisfiable. □

How can we find proofs that only use the literals of φ as premises? Our framework generalizes the eager and lazy approaches in order to obtain such proofs efficiently. This is the subject of the next two sections.

3. Eager encodings

Algorithm 1 (EAGER- ENCODING) computes a propositional encoding of a given formula φ in a single step. All the proof steps that might be necessary are assumed to be performed by the DEDUCTION procedure before the propositional engine is called.

Algorithm 1 *Input*: A Formula φ. *Output*: 'Satisfiable' if φ is satisfiable and 'Unsatisfiable' otherwise.

1: **function** EAGER- ENCODING(φ)
2: $P :=$ DEDUCTION($lit(\varphi)$);
3: $\varphi_E := \varphi_{sk} \wedge e(\hat{P})$;
4: **return** SAT- SOLVER(φ_E);

The SAT- SOLVER procedure returns in line 4 one of {'Satisfiable','Unsatisfiable'} according to whether the input formula is satisfiable or not. Thus, the result of EAGER- ENCODING equals the result returned by SAT-SOLVER, as φ and φ_E are equisatisfiable. It is left to describe sufficient conditions for complete proofs. In other words, it is enough to prove that a given implementation of DEDUCTION fulfills any one of these conditions in order to establish completeness of the procedure.

3.1. Criteria for complete proofs

Let α be either a partial or full truth assignment to φ_{sk}. For a propositional encoder e unassigned by α we write $\alpha(e) = \perp$. The following notation is used:

- The literal corresponding to the assignment of α to its propositional encoder:

$$Th(lit_i, \alpha) := \begin{cases} lit_i & : \ \alpha(e(lit_i)) = \text{TRUE} \\ \neg lit_i & : \ \alpha(e(lit_i)) = \text{FALSE} \\ \text{TRUE} & : \ \alpha(e(lit_i)) = \perp \end{cases} \tag{6}$$

- We denote by $Th(\alpha)$ the set of literals $Th(lit_i, \alpha)$ for all literals $lit_i \in lit(\varphi)$.
- We write $Th(\alpha) \longrightarrow_P \text{FALSE}$ if a proof P leads to FALSE using $Th(\alpha)$ as premises[2].

The following example demonstrates the use of this notation.

Example 2 Let $lit_1 = (x_1 > x_2)$ and $lit_2 = (x_2 \leq x_1)$ be the literals of a formula φ. Now consider the assignment

$$\alpha(e(x_1 > x_2)) = \text{TRUE}, \quad \alpha(e(x_2 \leq x_1)) = \text{FALSE}. \tag{7}$$

Thus, we have

$$Th(lit_1, \alpha) = x_1 > x_2 \quad Th(lit_2, \alpha) = \neg\,(x_2 \leq x_1) = x_2 > x_1 \tag{8}$$

and

$$Th(\alpha) = \{x_1 > x_2,\ x_2 > x_1\}. \tag{9}$$

We use the following proof rules:

$$\frac{x_i > x_j \qquad x_j > x_k}{x_i > x_k}\ (>\text{- TRANS}) \qquad \frac{x_i > x_i}{\text{FALSE}}\ (>\text{- CONTR}) \tag{10}$$

and $Th(\alpha)$ as the set of premises. Let P be the following proof:

$$\begin{aligned} P : \{ &(>\text{- TRANS}, \quad x_1 > x_1, \quad Th(\alpha)), \\ &(>\text{- CONTR}, \quad \text{FALSE}, \quad x_1 > x_1)\}. \end{aligned} \tag{11}$$

This proof shows that $Th(\alpha)$ is inconsistent, i.e., $Th(\alpha) \longrightarrow_P \text{FALSE}$. □

[2] An alternative definition is: there is a node in the proof-graph of P whose consequent is FALSE and its supporting leaves are a subset of $Th(\alpha)$.

The following theorem defines a sufficient condition for the completeness of a proof.

Theorem 3 (Sufficient condition #1 for completeness) Let φ be an unsatisfiable formula. A proof P is complete with respect to φ if for every full assignment α to φ_{sk},

$$\alpha \models \varphi_{sk} \quad \Rightarrow \quad Th(\alpha) \longrightarrow_P \text{FALSE}. \tag{12}$$

The premise of this theorem can be weakened, which leads to a stronger theorem. In the following theorem we use the term *prime implicants*, which, for a given formula φ, are minimal partial assignments such that any of their extensions satisfy φ.

Theorem 4 (Sufficient condition #2 for completeness) Let φ be an unsatisfiable formula. A proof P is complete with respect to φ if for every prime implicant α of φ_{sk}, $Th(\alpha) \longrightarrow_P \text{FALSE}$.

Now consider a yet weaker requirement for complete proofs:

Theorem 5 (Sufficient condition #3 for completeness) Let φ be an unsatisfiable formula. A proof P is complete with respect to φ if for every prime implicant α of φ_{sk}, for some unsatisfiable core $Th^{uc}(\alpha) \subseteq Th(\alpha)$, $Th^{uc}(\alpha) \longrightarrow_P$ FALSE.

Note that there is at least one unsatisfiable core because $Th(\alpha)$ must be unsatisfiable if $\alpha \models \varphi_{sk}$ and φ is unsatisfiable.

It is not hard to see that Theorem 5 implies Theorem 4, which, in turn, implies Theorem 3. Hence, we only prove Theorem 5.

Proof Let φ be an unsatisfiable formula. Assume that $\varphi_{sk} \wedge e(\hat{P})$ is satisfiable, where P satisfies the premise of Theorem 5, i.e., for each prime implicant α of φ_{sk}, it holds that $Th^{uc}(\alpha) \longrightarrow_P$ FALSE for some unsatisfiable core $Th^{uc}(\alpha) \subseteq Th(\alpha)$. Let α' be the satisfying assignment, and let α be a prime implicant of φ_{sk} that can be extended to α'. Let $Th^{uc}(\alpha) \subseteq Th(\alpha)$ denote an unsatisfiable core of $Th(\alpha)$ such that $Th^{uc}(\alpha) \longrightarrow_P$ FALSE. This implies that $e(\hat{P})$ evaluates to FALSE when the premises of P corresponding to Th^{uc} are evaluated to α. This implies that $\varphi_{sk} \wedge e(\hat{P})$ evaluates to FALSE under α, a contradiction. □

The problem, now, is to find a proof P that fulfills one of these sufficient conditions.

3.2. Algorithms for generating complete proofs

Recall that by Theorem 2, a sound and complete deductive decision procedure for a conjunction of terms can be used for generating complete proofs, simply by case-splitting and conjoining the proof steps. As discussed earlier, this type of procedure misses the point, as we want to find such proofs with less effort than splitting. We now study, then, strategies for modifying such procedures so they generate complete proofs from disjunctive formulas, with potentially less effort than splitting. The procedures that we study in this section are generic, and fulfill conditions much stronger than required by the premises of Theorems 3–5.

We need the following definition:

Definition 8 (Saturation) Let Γ be an inference system (i.e., a set of inference rules and axioms, including schemas). We say that the process of applying Γ to a set of premises *saturates* if no new facts can be derived based on these premises and previously derived facts. Γ is said to be *saturating* if applying it to any set of premises saturates.

In this section, we consider decision procedures whose underlying inference system is saturating—other classes are left for future work. Many popular decision procedures belong to this class. For example, Simplex [Dan63], Fourier–Motzkin [BW94], and the Omega-test [Pug91], which can all be presented as based on deduction (see Nelson [Nel81] and Ruess and Shankar [RS04] for a deductive version of Simplex), belong to this class.

Let dp be a deductive decision procedure in this class for conjunction of $\mathcal{T}$-terms (where $\mathcal{T}$ is some theory), and let Γ be the set of inference rules that it can use. Let φ be a disjunctive $\mathcal{T}$-formula. Now consider the following procedure:

Apply the rules in Γ *to* $lit(\varphi)$ *until saturation.*

Since every inference that is possible after case-splitting is also possible here, this procedure clearly generates a complete proof. Note that the generality of this variant comes with the price of completely ignoring the inference

strategy applied by the original decision procedure dp, which entails a sacrifice in efficiency and the size of the resulting proof. Nevertheless, even with this general scheme, the number of inferences is expected to be much smaller than using case-splitting, because the same inference is never repeated (whereas it can be repeated an exponential number of times with case-splitting).

Specific decision procedures that belong to this class can be changed in a way that results in a more efficient procedure, however. We consider here the case of projection-based decision-procedures, and present it through an example, namely the Fourier–Motzkin (FM) procedure for linear arithmetic.

Consider the following rules:

$$\frac{UB \geq x \qquad x \geq LB}{UB \geq LB}\ (\textsc{Project}) \qquad \frac{l > u}{\text{FALSE}}\ (\textsc{Constants- Contradiction}) \tag{13}$$

where UB and LB are linear constraints that do not include x, and l, u are any two constants such that $l \leq u$. Given a conjunction of linear arithmetic predicates ϕ, FM's strategy is, informally, the following:

1. If $var(\phi) = \emptyset$ return 'Satisfiable';
2. Choose a variable $v \in var(\phi)$;
3. For every upper bound UB and a lower bound LB on x, apply rule Project;
4. Simplify the resulting constraints by accumulating the coefficients of the same variable;
5. Remove all the constraints that contain x;
6. If rule Constants- contradiction is applicable, return 'Unsatisfiable';
7. Goto Step 2;

Note that the result of each step is a new formula without the eliminated variable, which is equisatisfiable to the previous one.

Now consider the following variation of this procedure, which is meant for generating complete proofs (rather than deciding a formula) starting from a disjunctive linear arithmetic formula φ.

> Apply FM to $lit(\varphi)$ with the following difference: Replace Step 6 with:
> 6. If rule Constants- contradiction is applicable, apply it;

This procedure, which first appeared in [Str02] (and, motivated the generalization presented in the current work), generates complete proofs. It preserves the following condition: The set of facts that are derived from each inconsistent set of literals is inconsistent. We call a projection that has this property, a *strong projection*. It is not hard to see that any strong projection method preserves the premise of Theorem 3.

4. Lazy encodings

An eager encoding, which is based on the entire set of literals, may result in more deduction steps than needed, due to two reasons: First, algorithms for generating complete proofs as presented in Sect. 3.2 are wasteful because they ignore the Boolean structure of the formula and hence deduce facts from literals that do not have to be satisfied simultaneously. Second, frequently only a small part of the input formula is necessary for showing that it is unsatisfiable. This motivates the idea of a *lazy encoding*: Instead of building a propositional encoding in a single step, a series of increasingly stronger formulas is built, starting from φ_{sk}. Termination is typically achieved with fewer deduction steps than required for the eager encoding. The main algorithmic question is how to choose the additional deduction steps to perform for the next formula. We do not provide a new solution for this problem, but only propose a framework that generalizes existing algorithms for tackling it.

4.1. An algorithm for computing lazy encodings

Algorithm 2 (DPLL($\mathcal{T}$)- E) takes φ as input and decides whether it is satisfiable. It does so by iteratively solving a propositional formula, starting from φ_{sk}, and gradually strengthening it with proof constraints. Theorem 1 implies that every such intermediate formula is never stronger than a propositional encoding of φ. Tinelli's original DPLL($\mathcal{T}$) abstract calculus [Tin02] is general enough to represent many algorithmic variants. Algorithm 2 extends

one such variant (in a matter soon to be explained), in which *theory propagation* (learning new facts due to theory implications) is applied after full Boolean Constraint Propagation (BCP). In the context of the current paper, there is no importance to this restriction—we merely want to illustrate how deductive proofs can be used for theory propagation, regardless of the algorithmic variant.

DPLL is integrated in Algorithm 2 in a way that enables an incremental solving process [Sht01, WKS01]. In particular, the function ADDCLAUSES adds clauses to a given instance without restarting the DPLL process, hence conflict clauses are maintained in the solver's clause database. After every decision and its implications (through BCP) that does not end in a conflict, DEDUCTION is called in Line 11 with $Th(\alpha)$ as argument, where α is the current partial assignment. The DEDUCTION procedure returns a proof constraint $\hat{P}$. This generalizes *theory propagation* [GHN+04, NO05]—not its essence, only the common practice—in the sense that it does not restrict the returned constraint to implications of the current partial assignment α nor to existing literals. If $Th(\alpha)$ is inconsistent, $e(\hat{P})$ should, as a minimum, refute the current partial assignment α (see the discussion on termination in the next subsection).

Algorithm 2 *Input*: A formula φ. *Output*: 'Satisfiable' if the formula is satisfiable and 'Unsatisfiable' otherwise.

```
1:  function DPLL(T)- E
2:      AddClauses(cnf(e(φ)));
3:      if BCP() = 'conflict' then return 'Unsatisfiable';
4:      while (TRUE) do
5:          if ¬Decide() then return 'Satisfiable';                ▷ Full assignment
6:          repeat
7:              while (BCP() = 'conflict') do
8:                  backtrack-level := Analyze- Conflict();
9:                  if backtrack-level < 0 then return 'Unsatisfiable';
10:                 else BackTrack(backtrack-level);
11:             P := Deduction(Th(α));                  ▷ Returns TRUE if saturated
12:             AddClauses(e(P̂));
13:         until P̂ = ∅
```

4.2. Termination

The procedure described above is not guaranteed to terminate, even if DEDUCTION always does. This is because DEDUCTION is permitted to return proof steps with new literals, and there is nothing to prevent an infinite series of these. In order to guarantee termination, the proof P that is returned by DEDUCTION has to be restricted in some way. Note that, as was indicated earlier, if $Th(\alpha)$ is inconsistent, then $e(\hat{P})$ must refute α.

Before discussing specific restrictions, let us make the following link between termination and completeness of proofs: for any theory with an algorithm that generates complete proofs, there is a lazy algorithm that terminates. This is not hard to achieve, because one only needs to restrict the proof steps that are generated by DEDUCTION to the proof steps that are possibly included in the corresponding complete proof. The other direction holds as well: If we have a lazy decision procedure that terminates, then there exists a complete proof. Indeed, the set of proof steps generated by DEDUCTION forms a complete proof. Hence, there is a clear connection between complete proofs in the eager approach, and termination in the lazy approach. This result is not constructive, however: it does not specify how DEDUCTION should be restricted in order to guarantee termination. We continue, therefore, with concrete conditions on termination.

A trivial way to guarantee termination is to require DEDUCTION to restrict the proof steps to the literals of φ: This restriction guarantees that no new variables are introduced. We may want to introduce new literals, however, in order to prevent redundant work between the different invocations of DEDUCTION, and thus, propose a weaker condition. The following simple generalization captures this condition.

Theorem 6 (Sufficient condition for termination) If P is a proof such that the literals in all antecedents and all consequents of P are contained in a finite set of literals, Algorithm LAZY- ENCODING terminates.

This condition is not difficult to fulfill. In fact, it corresponds to the class of decision procedures that we referred to in Sect. 3.2, for which we showed a method for generating complete proofs. This type of restriction

was also posed by Barrett et al. in [BNOT06] in their extension to DPLL($\mathcal{T}$) that allows the introduction of new variables by the theory solvers.

In the general case, in which there is no implicit bound on the number of facts that can be generated as in these procedures, one may put a bound on their number (i.e., after inferring a certain predefined number of new facts, restricting further consequents to existing predicates) and hence fulfill the condition of Theorem 6.

5. Conclusion

We have presented a generic framework for propositional encoding of $\mathcal{T}$-formulas starting from a deductive sound and complete decision procedure for a theory $\mathcal{T}$. We established conditions on soundness and completeness for programs that compute eager and lazy encodings by instantiating this framework.

The main idea is to change the decision procedure so it takes as premises the set of literals of the input formula, and let the SAT solver reason about its Boolean structure. We showed that the eager and lazy encodings, and some previously published optimizations thereof, are instantiations of this framework. Although we have not done so in this paper, it should be fairly easy to show that these optimizations are correct by the fact that they fulfill one of the various tests for completeness that we presented in Theorems 3, 4, and 5.

This framework offers two directions for future research on how to make SMT solvers work faster: (1) the introduction of new variables during the encoding (already partially explored by previous publications, e.g., [BNOT06]), and (2) exploiting the unified interface of the generic algorithm that we presented in Algorithm 2.

References

[ABC+02] Audemard G, Bertoli P, Cimatti A, Kornilowicz A, Sebastiani R (2002) A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In: Voronkov A (ed) Automated deduction (CADE 2002), Lecture notes in computer science, vol 2392. Springer, Berlin

[BdMS05] Barrett C, de Moura L, Stump A (2005) Design and results of the 1st satisfiability modulo theories competition (SMT-COMP 2005). J Autom Reason 35(4):373–390

[BNOT06] Barrett C, Nieuwenhuis R, Oliveras A, Tinelli C (2006) Splitting on demand in SAT modulo theories. In: Logic for programming, artificial intelligence, and reasoning (LPAR 2006). Lecture notes in computer science, vol 4246. Springer, Berlin, pp 512–526

[BV00] Bryant RE, Velev M (2000) Boolean satisfiability with transitivity constraints. In: Proceedings of the 12th international conference on computer aided verification (CAV 2000). Lecture notes in computer science, vol 1855, pp 85–98

[BW94] Bik AJC, Wijshoff HAG (1994) Implementation of Fourier–Motzkin elimination. Technical report 94-42. Department of Computer Science, Leiden University

[Dan63] Danzig GB (1963) Linear programming and extensions. Princeton University Press, NJ

[FJOS03] Flanagan C, Joshi R, Ou X, Saxe JB (2003) Theorem proving using lazy proof explication. In: Proceedings of the 15th international conference on computer aided verification (CAV 2003). Lecture notes in computer science, vol 2725. Springer, Berlin, pp 355–367

[FORS01] Filliatre JC, Owre S, Rueb H, Shankar N (2001) ICS: integrated canonizer and solver. In: Berry G, Comon H, Finkel A (eds) Proceedings of the 13th international conference on computer aided verification (CAV 2001). Lecture notes in computer science, vol 2102. Springer, Berlin, pp 246–249

[GHN+04] Ganzinger H, Hagen G, Nieuwenhuis R, Oliveras A, Tinelli C (2004) DPLL(T): fast decision procedures. In: Proceedings of the 16th international conference on computer aided verification (CAV 2004). Lecture notes in computer science, vol 3114. Springer, Berlin, pp 175–188

[MMZ+01] Moskewicz M, Madigan C, Zhao Y, Zhang L, Malik S (2001) Chaff: engineering an efficient SAT solver. In: Design automation conference (DAC 2001). ACM Press, New York, pp 530–535

[Nel81] Nelson G (1981) Techniques for program verification. Technical report. Xerox Palo Alto Research Center (CSL-81-10)

[NO05] Nieuwenhuis R, Oliveras A (2005) DPLL(T) with exhaustive theory propagation and its application to difference logic. In: Proceedings of the 17th internatioal conference on computer aided verification (CAV 2005). Lecture notes in computer science, vol 3576. Springer, Berlin, pp 321–334

[PRSS99] Pnueli A, Rodeh Y, Shtrichman O, Siegel M (1999) Deciding equality formulas by small-domains instantiations. In: Proceedings of the 11th international conference on computer aided verification (CAV'99). Lecture notes in computer science, vol 1633. Springer, Berlin, pp 455–469

[Pug91] Pugh W (1991) The Omega test: a fast and practical integer programming algorithm for dependence analysis. In: Proceedings of the 1991 ACM/IEEE conference on supercomputing, pp 4–13

[RS04] Ruess H, Shankar N (2004) Solving linear arithmetic constraints. Technical Report (SRI-CSL-04-01), SRI

[SBD02] Stump A, Barrett C, Dill D (2002) CVC: a cooperating validity checker. In: Brinksma E, Larsen KG (eds) Proceedings of the 14th international conference on computer aided verification (CAV'02). Lecture notes in computer science, vol 2404. Springer, Berlin, pp 500–504

[Sht01] Shtrichman O (2001) Pruning techniques for the sat-based bounded model checking problem. In: Margaria T, Melham TF (eds) Correct hardware design and verification methods (CHARME 2001). Lecture notes in computer science, vol 2144. Springer, Berlin, pp 58–70

[SSB02] Strichman O, Seshia SA, Bryant RE (2002) Deciding separation formulas with SAT. In: Brinksma E, Larsen KG (eds) Proceedings of the 14th international conference on computer aided verification (CAV'02). Lecture notes in computer science, vol 2404. Springer, Berlin, pp 209–222

[Str02] Strichman O (2002) On solving Presburger and linear arithmetic with SAT. In: Aagaard M, O'Leary JW (eds) Formal methods in computer-aided design (FMCAD 2002). Lecture notes in computer science, vol 2517. Springer, Portland, pp 160–170

[Tin02] Tinelli C (2002) A DPLL-based calculus for ground satisfiability modulo theories. In: Proceedings of the 8th European conference on logics in artificial intelligence. Lecture notes in artificial intelligence, vol 2424. Springer, Berlin, pp 308–319

[WKS01] Whittemore J, Kim J, Sakallah K (2001) SATIRE: a new incremental satisfiability engine. In: Design automation conference (DAC). ACM Press, New York, pp 542–545

Received 15 March 2006

Accepted in revised form 8 October 2008 by C.B. Jones

Published online 21 February 2009