Challenges in Satisfiability Modulo Theories

Robert Nieuwenhuis, Albert Oliveras,
Enric Rodríguez-Carbonell, and Albert Rubio*

Abstract. Here we give a short overview of the DPLL(T) approach to Satisfiability Modulo Theories (SMT), which is at the basis of current state-of-the-art SMT systems. After that, we provide a documented list of theoretical and practical current challenges related to SMT, including some new ideas to exploit SAT techniques in Constraint Programming.

1 Introduction

Propositional satisfiability checkers (SAT solvers) are currently being applied in more and more contexts, including hardware and software verification, in Operations Research (planning, scheduling), as well as in Biology, Linguistics and Medicine. Most SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DP60,DLL62]. The performance of DPLL-based SAT solvers has improved spectacularly in the last years, due to better implementation techniques and conceptual enhancements such as *backjumping*, *conflict-driven lemma learning* and *restarts* [MSS99,MMZ$^+$01,ES03]. However, some practical problems are more naturally expressed in logics that are more expressive than propositional logic.

For example, for timed automata, a good choice is *difference logic*, where formulas contain atoms of the form $a - b \leq k$, which are interpreted with respect to a background theory T of the integers, rationals or reals. Similarly, for the verification of pipelined microprocessors it is convenient to consider a logic of *Equality with Uninterpreted Functions (EUF)*, where the background theory T specifies a congruence [BD94]. To mention just one other example, the conditions arising from program verification usually involve arrays, lists and other data structures, so it becomes very natural to consider satisfiability problems *modulo* the theory T of these data structures. In such applications, problems may contain thousands of clauses like

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

containing purely propositional atoms as well as atoms over (combined) theories. This is known as the *Satisfiability Modulo Theories* (SMT) problem for a theory T: given a formula F, determine whether F is T-satisfiable, i.e., whether there exists a model of T that is also a model of F.

SMT has become an extremely active area of research. A rapidly growing library of benchmarks for SMT with a formal syntax and semantics exists [RT03],

---

* Technical Univ. of Catalonia, Barcelona. All authors partially supported by Spanish Min. of Educ. and Science through the LogicTools project (TIN2004-03382) and Intel Corp. Research Grant: “SMT Solvers for High-Level Hardware Verification”.

as well as a yearly SMT competition (both SMT-LIB and SMT-COMP are easily found on the web).

The DPLL(T) approach to SMT is based on a general DPLL(X) engine, whose parameter X can be instantiated with specialized solvers $Solver_T$ for given theories T, thus producing a system DPLL(T). Once the DPLL(X) engine has been implemented, new theories can be dealt with by simply plugging in new theory solvers. These solvers must only be able to deal with *conjunctions* of theory literals and conform to a minimal and simple set of additional requirements.

In Sections 2, 3 and 4 of this paper, by means of a rewrite-rule-based framework called Abstract DPLL we first give a brief overview of DPLL, SMT, and the DPLL(T) approach to SMT (we refer to [NOT06] for all details and references). In Section 5 we describe a number of theoretical and practical challenges in SMT. Extensions for handling new theories and applications, including optimization and constraint programming are discussed, as well as for first-order theorem proving. Other challenges involve the design of efficient data structures and algorithms for implementing certain key parts of SMT solvers. All of them are closely related to the area of rewriting.

2 The DPLL Procedure

Let P be a fixed finite set of propositional symbols. If $p \in P$, then p and $\neg p$ are *literals* of P. The *negation* of a literal l, written $\neg l$, denotes $\neg p$ if l is p, and p if l is $\neg p$. A *clause* is a disjunction of literals $l_1 \vee \ldots \vee l_n$. A *unit clause* is a clause consisting of a single literal. A (finite, non-empty, CNF) *formula* is a conjunction of one or more clauses $C_1 \wedge \ldots \wedge C_n$. When it leads to no ambiguities, we sometimes also write such a formula in set notation $\{C_1, \ldots, C_n\}$ or simply replace $\wedge$ connectives by commas.

A (partial truth) *assignment* M is a set of literals such that $\{p, \neg p\} \subseteq M$ for no p. A literal l is *true* in M if $l \in M$, it is *false* in M if $\neg l \in M$, and l is *undefined* in M otherwise. M is *total* over P if no literal of P is undefined in M. A clause C is true in M if at least one of its literals is true in M. It is false in M if all its literals are false in M, and it is undefined in M otherwise. A formula F is true in M, or *satisfied* by M, denoted $M \models F$, if all its clauses are true in M. In that case, M is called a *model* of F. If F has no models then it is called *unsatisfiable*. If F and F' are formulas, we write $F \models F'$ if F' is true in all models of F. Then we say that F' *is entailed by* F, or is a *logical consequence* of F.

In what follows, (possibly subscripted or primed) lowercase l *always* denotes literals. Similarly C and D always denote clauses, F and G denote formulas, and M and N are assignments. If C is a clause $l_1 \vee \ldots \vee l_n$, we sometimes write $\neg C$ to denote the formula $\neg l_1 \wedge \ldots \wedge \neg l_n$.

Here a DPLL procedure is modeled by a transition relation over states. A state is either *FailState* or a pair $M \parallel F$, where F is a finite set of clauses and M is a sequence of literals that is seen as a partial assignment. Some literals l in M will be *annotated* as being *decision literals*; these are the ones added to

M by the Decide rule given below, and are sometimes written l^{d}. The transition relation is defined by means of rules.

Definition 1. *The* DPLL system *with* Backtrack *consists of the four rules:*

UnitPropagate :

$$M \parallel F,\, C \vee l \quad \Longrightarrow \quad M\; l \parallel F,\, C \vee l \quad \textbf{if} \begin{cases} M \models \neg C \\ l \textit{ is undefined in } M \end{cases}$$

Decide :

$$M \parallel F \quad \Longrightarrow \quad M\; l^{\mathsf{d}} \parallel F \quad \textbf{if} \begin{cases} l \textit{ or } \neg l \textit{ occurs in a clause of } F \\ l \textit{ is undefined in } M \end{cases}$$

Fail :

$$M \parallel F,\, C \quad \Longrightarrow \quad \mathit{FailState} \quad \textbf{if} \begin{cases} M \models \neg C \\ M \textit{ contains no decision literals} \end{cases}$$

Backtrack :

$$M\; l^{\mathsf{d}}\; N \parallel F,\, C \quad \Longrightarrow \quad M\; \neg l \parallel F,\, C \quad \textbf{if} \begin{cases} M\; l^{\mathsf{d}}\; N \models \neg C \\ N \textit{ contains no decision literals} \end{cases}$$

One can use these rules for deciding the satisfiability of an input CNF F by simply generating an arbitrary derivation $\emptyset \parallel F \Longrightarrow \ldots \Longrightarrow S_n$, where S_n is irreducible by the rules. Such derivations are always finite, and
(i) F is unsatisfiable if, and only if, the final state S_n is *FailState*, and
(ii) if S_n is of the form $M \parallel F$ then M is a model of F.

These rules speak for themselves, providing a classical depth-first search with backtracking, where the Decide rule represents a case split: an undefined literal l is chosen from F, and added to M. The literal is annotated as a *decision literal*, to denote that, if $M\; l^{\mathsf{d}}$ cannot be extended to a model of F, then (by Backtrack) still the other possibility $M\; \neg l$ must be explored. In the following, if M is a sequence of the form $M_0\; l_1\; M_1 \ldots l_k\; M_k$, where the l_i are all the decision literals in M, then the literals of each $l_i\; M_i$ are said to *belong to decision level* i.

Example 2. In the following derivation, to improve readability we have denoted atoms by natural numbers, negation by overlining, and written decision literals in **bold** font:

$$\begin{array}{rcll}
\emptyset & \parallel & \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{1}\vee\overline{3}\vee 4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee 4 & \Longrightarrow \quad \text{(Decide)} \\
\mathbf{1} & \parallel & \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{1}\vee\overline{3}\vee 4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee 4 & \Longrightarrow \quad \text{(UnitPropagate)} \\
\mathbf{1}\,\overline{2} & \parallel & \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{1}\vee\overline{3}\vee 4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee 4 & \Longrightarrow \quad \text{(UnitPropagate)} \\
\mathbf{1}\,\overline{2}\,3 & \parallel & \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{1}\vee\overline{3}\vee 4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee 4 & \Longrightarrow \quad \text{(UnitPropagate)} \\
\mathbf{1}\,\overline{2}\,3\,4 & \parallel & \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{1}\vee\overline{3}\vee 4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee 4 & \Longrightarrow \quad \text{(Backtrack)} \\
\overline{1} & \parallel & \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{1}\vee\overline{3}\vee 4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee 4 & \Longrightarrow \quad \text{(UnitPropagate)} \\
\overline{1}\,4 & \parallel & \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{1}\vee\overline{3}\vee 4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee 4 & \Longrightarrow \quad \text{(Decide)} \\
\overline{1}\,4\,\mathbf{3} & \parallel & \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{1}\vee\overline{3}\vee 4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee 4 & \Longrightarrow \quad \text{(UnitPropagate)} \\
\overline{1}\,4\,\mathbf{3}\,2 & \parallel & \overline{1}\vee\overline{2},\ 2\vee 3,\ \overline{1}\vee\overline{3}\vee 4,\ 2\vee\overline{3}\vee\overline{4},\ 1\vee 4 & \qquad\quad \text{Final state:} \\
 & & & \qquad\qquad \text{model found.}
\end{array}$$

□

In modern DPLL-based SAT solvers instead of Backtrack a more general Backjump rule is considered, of which Backtrack is a particular case.

Definition 3. *The Backjump rule is defined as follows:*

$$M\ l^{\mathsf{d}}\ N \parallel F,\ C \implies M\ l' \parallel F,\ C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \quad F, C \models C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \end{cases}$$

We call the clause $C' \vee l'$ a backjump *clause.*

Example 4. The aim of this Backjump rule is to generalize backtracking by a better analysis of why the so-called *conflicting* clause C is false. Standard backtracking reverses the *last* decision, and adds it (as a non-decision literal) to the previous decision level. Backjumping generalizes this by adding a new literal to a possibly lower decision level. Consider:

$$\begin{array}{rcl}
\emptyset \parallel \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \implies & \text{(Decide)} \\
\mathbf{1} \parallel \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \implies & \text{(UnitPropagate)} \\
\mathbf{1}\ 2 \parallel \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \implies & \text{(Decide)} \\
\mathbf{1}\ 2\ \mathbf{3} \parallel \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \implies & \text{(UnitPropagate)} \\
\mathbf{1}\ 2\ \mathbf{3}\ 4 \parallel \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \implies & \text{(Decide)} \\
\mathbf{1}\ 2\ \mathbf{3}\ 4\ \mathbf{5} \parallel \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \implies & \text{(UnitPropagate)} \\
\mathbf{1}\ 2\ \mathbf{3}\ 4\ \mathbf{5}\ \overline{6} \parallel \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \implies & \text{(Backjump)} \\
\mathbf{1}\ 2\ \overline{5} \parallel \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \implies & \ldots
\end{array}$$

Before the Backjump step, the clause $6\vee\overline{5}\vee\overline{2}$ is conflicting: it is false in $\mathbf{1}\ 2\ \mathbf{3}\ 4\ \mathbf{5}\ \overline{6}$. The reason for its falsity is the literal 2 together with the decision **5** and its unit propagation $\overline{6}$. Therefore, one can infer that 2 is incompatible with the decision **5**. Indeed, the backjump clause $\overline{2} \vee \overline{5}$ is a logical consequence of the last two clauses. It allows us to return to the first decision level, adding there, as a unit propagation, the literal $\overline{5}$ (which plays the role of l' in the Backjump rule). □

Note that in the previous example an application of Backtrack instead of Backjump would have given a state with first component $\mathbf{1}\ 2\ \mathbf{3}\ 4\ \overline{5}$, even though the decision level $\mathbf{3}\ 4$ is unrelated with the reasons for the falsity of $6 \vee \overline{5} \vee \overline{2}$. Moreover, intuitively, the search state $\mathbf{1}\ 2\ \overline{5}$ reached after Backjump is more *advanced* than $\mathbf{1}\ 2\ \mathbf{3}\ 4\ \overline{5}$. This notion of "being more advanced" is formalized in Theorem 8 below.

The following example shows how Backjump can be applied in practice, by finding an adequate backjump clause.

Example 5. Consider a state $M \parallel F$, where, among other clauses, F contains:

$$\overline{9}\lor\overline{6}\lor 7\lor\overline{8} \quad 8\lor 7\lor\overline{5} \quad \overline{6}\lor 8\lor 4 \quad \overline{4}\lor\overline{1} \quad \overline{4}\lor 5\lor 2 \quad 5\lor 7\lor\overline{3} \quad 1\lor\overline{2}\lor 3$$

and M is of the form: $\ldots 6 \ldots \overline{7} \ldots \mathbf{9}\ \overline{8}\ \overline{5}\ 4\ \overline{1}\ 2\ \overline{3}$. It is easy to observe how by six applications of UnitPropagate this state has been reached after the last decision **9**. For example, $\overline{8}$ is implied by 9, 6, and $\overline{7}$, due to the leftmost clause $\overline{9}\lor\overline{6}\lor 7\lor\overline{8}$. The ordered sequence of propagated literals is stored, each one of them together with the clause that caused it. In this state $M \parallel F$, the clause $1\lor\overline{2}\lor 3$ is conflicting, since M contains $\overline{1}$, 2 and $\overline{3}$. Now one can trace back the reasons for this conflicting clause. For example, $\overline{3}$ was implied by $\overline{5}$ and $\overline{7}$, due to the clause $5\lor 7\lor\overline{3}$. The literal $\overline{5}$ was in turn implied by $\overline{8}$ and $\overline{7}$, and so on. In this way, working backwards from the conflicting clause, and in the reverse order in which each literal was propagated, we get the following *(conflict) resolution* proof:

$$\frac{8\lor 7\lor\overline{\mathbf{5}} \qquad \frac{\overline{6}\lor 8\lor\mathbf{4} \qquad \frac{\overline{4}\lor\overline{\mathbf{1}} \qquad \frac{\overline{4}\lor 5\lor\mathbf{2} \qquad \frac{5\lor 7\lor\overline{\mathbf{3}} \qquad 1\lor\overline{2}\lor\mathbf{3}}{5\lor 7\lor 1\lor\overline{\mathbf{2}}}}{\overline{4}\lor 5\lor 7\lor\mathbf{1}}}{5\lor 7\lor\overline{\mathbf{4}}}}{\overline{6}\lor 8\lor 7\lor\mathbf{5}}}{8\lor 7\lor\overline{6}}$$

The process stops once it generates a clause with only one atom of the current decision level, in our example, the literal 8 in the clause $8\lor 7\lor\overline{6}$. This process is equivalent to the *implication-graph-based* conflict analysis, where this literal is called the *First Unique Implication Point (1UIP)*, see [MSS99,MMZ$^+$01]. The clause obtained is a backjump clause $C\lor l'$ where the 1UIP is the literal l'. □

In most modern DPLL implementations, the backjump clause is always *added to the clause set* as a *learned* clause (*conflict-driven clause learning*). In the previous example, learning the lemma $8\lor 7\lor\overline{6}$ will allow the application of UnitPropagate to any state where M contains the negation of two of its literals, hence preventing any conflict caused by having the negation of all three. Indeed, reaching such *similar* conflicts frequently happens in industrial problems having some regular structure, and learning such lemmas is known to be very effective. Since a lemma is aimed at preventing future similar conflicts, when such conflicts are not very likely to be found again the lemma can be removed. In practice this is done if its *activity* (e.g., the number of times it is involved in a conflict) has become low.

Definition 6. *The rules of* Learn *and* Forget *are the following ones:*

Learn :
$$M \parallel F \quad \Longrightarrow \quad M \parallel F, C \quad \textbf{if} \begin{cases} \textit{all atoms of } C \textit{ occur in } F \\ F \models C \end{cases}$$
Forget :
$$M \parallel F, C \quad \Longrightarrow \quad M \parallel F \quad \textbf{if} \ \big\{ F \models C$$

State-of-the-art SAT-solvers [MMZ+01,ES03] essentially apply these rules using efficient implementation techniques for UnitPropagate (e.g., watching two literals for unit propagation [MMZ+01]), and *activity-based* heuristics for selecting the decision literal for Decide: split on literals that occur in *recent* lemmas and conflicts. In addition, the DPLL procedure may periodically be *restarted* to escape from bad search behaviors. The rationale behind this is that upon each Restart (i.e., $M \parallel F \implies \emptyset \parallel F$), the newly learned lemmas will lead the heuristics for Decide to behave differently, and hopefully cause the procedure to explore the search space in a more compact way. We have the following results for the DPLL rules introduced so far (see [NOT06] for all details):

Theorem 7. *If* $\emptyset \parallel F \implies^* S$ *where* S *is irreducible w.r.t.* Decide, Backjump *and* Fail, *then (i)* S *is FailState if, and only if,* F *is unsatisfiable, and (ii) if* S *is of the form* $M \parallel F'$ *then* M *is a model of* F.

Theorem 8. *Any derivation* $\emptyset \parallel F \implies S_1 \implies \ldots$ *is finite if it contains only finitely many consecutive* Learn *and* Forget *steps and* Restart *is applied only with increasing periodicity.*

3 Satisfiability Modulo Theories

Now let the set P over which formulas are built be a fixed finite set of *ground* (i.e., variable-free) first-order atoms (instead of propositional symbols as before). A *theory* T is a set of closed first-order formulas that is satisfiable in the first-order sense. A formula F is T*-satisfiable* or T*-consistent* if $F \wedge T$ is satisfiable in the first-order sense. If M is a T-consistent partial assignment and F is a formula such that $M \models F$, i.e., M is a (propositional) model of F, then we say that M *is a* T*-model of* F. The SMT problem for a theory T is the problem of determining, given a formula F, whether F is T-satisfiable, or, equivalently, whether F has a T-model. Note that, as usual in SMT, here we only consider the SMT problem for *ground* (and hence quantifier-free) CNF formulas F. Also note that F may contain constants that are free in T, which, as far as satisfiability is concerned, can equivalently be seen as existentially quantified variables. We will consider here only theories T such that the T-satisfiability of conjunctions of such ground literals is decidable, and a decision procedure for doing so is called a T*-solver.* If F and G are formulas, then F *entails* G *in* T, written $F \models_T G$, if $F \wedge \neg G$ is T-inconsistent.

The *eager* approach to SMT is based on sophisticated satisfiability-preserving translations from SMT into SAT. But on many practical problems the translation or the SAT solver run out of time or memory. Alternatively, translating into DNF and using a T-solver for deciding the satisfiability of conjunctions of theory literals is also too inefficient due to the exponential blowup of the DNF.

Therefore, the *lazy* approach tries to combine specialized T-solvers with state-of-the-art SAT solvers for dealing with the boolean structure of the formulas. It initially considers each atom as a propositional symbol, i.e., it “forgets” about the theory T. If a SAT solver reports propositional unsatisfiability, then F is also T-unsatisfiable. If it returns a propositional model of F, then this model (a conjunction of literals) is checked by a T-solver. If it is T-satisfiable then it is a T-model of F. Otherwise, the T-solver builds a ground clause, called a *theory lemma*, a clause C such that $\emptyset \models_T C$, precluding that model. This lemma is added to F and the SAT solver is started again. This process is repeated until the SAT solver finds a T-satisfiable model or returns unsatisfiable. The lazy approach is quite flexible, by combining any SAT solver with any T-solver. See [NOT06] for a more detailed comparison of approaches and references.

Example 9. Assume we are deciding the satisfiability of a large EUF formula, i.e., the background theory T is equality, and assume that the model M found by the SAT solver contains, among many others, the literals: $b = c$, $f(b) = c$, $a \neq g(b)$, and $g(f(c)) = a$. Then the T-solver detects that M is not a T-model, since $\quad b = c \quad \wedge \quad f(b) = c \quad \wedge \quad g(f(c)) = a \quad \models_T \quad a = g(b)$. Therefore, the lazy procedure has to be restarted after the corresponding theory lemma has been added to the clause set. In principle, one can take as theory lemma simply the negation of M, that is, the disjunction of the negations of all the literals in M. However, this clause may therefore have thousands of literals, and the lazy approach will behave much more efficiently if the T-solver is able to generate a small *explanation* of the T-inconsistency of M, which in this example could be the clause $\quad b \neq c \quad \vee \quad f(b) \neq c \quad \vee \quad g(f(c)) \neq a \quad \vee \quad a = g(b)$. □

3.1 DPLL Modulo Theories

We now adapt the abstract DPLL framework for the propositional case presented in the previous section. Here Learn, Forget and Backjump are slightly modified in order to work modulo theories: in these rules, entailment between formulas now becomes entailment in T:

Definition 10. *The rules T-Learn, T-Forget and T-Backjump are:*

T-Learn :

$$M \parallel F \quad \Longrightarrow \quad M \parallel F, C \quad \textbf{if} \begin{cases} \textit{every atom of } C \textit{ occurs in } F \textit{ or in } M \\ F \models_T C \end{cases}$$

T-Forget :

$$M \parallel F, C \quad \Longrightarrow \quad M \parallel F \quad \textbf{if} \begin{cases} F \models_T C \end{cases}$$

T-Backjump :

$$M\, l^{\mathsf{d}}\, N \parallel F, C \Longrightarrow M\, l' \parallel F, C \quad \textbf{if} \begin{cases} M\, l^{\mathsf{d}}\, N \models \neg C, \textit{ and there is} \\ \textit{some clause } C' \vee l' \textit{ such that:} \\ \quad F, C \models_T C' \vee l' \textit{ and } M \models \neg C', \\ \quad l' \textit{ is undefined in } M, \textit{ and} \\ \quad l' \textit{ or } \neg l' \textit{ occurs in } F \textit{ or in } M\, l^{\mathsf{d}}\, N \end{cases}$$

The naive lazy approach to SMT is modeled as follows using the rules. Each time a state $M \parallel F$ is reached that is irreducible with respect to Decide, Fail and T-Backjump, M can be T-consistent or not. If it is, then M is indeed a T-model of F. If it is not, then there exists a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \ldots \vee \neg l_n$. By one T-Learn step, this theory lemma $\neg l_1 \vee \ldots \vee \neg l_n$ can be learned and then Restart can be applied. If these theory lemmas are never removed by the T-Forget rule, this strategy is terminating, sound and complete in a similar sense as in the previous section.

Several important enhancements of the lazy approach can now easily be modeled using the rules:

Incremental T-solver. The T-consistency of the model can be checked incrementally, while the model is being built by the DPLL procedure, i.e., without delaying the check until a propositional model has been found, thus saving useless work. Assume a state $M \parallel F$ has been reached such that M is T-inconsistent. Then, as in the naive lazy approach, there exists a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \ldots \vee \neg l_n$. This theory lemma is then learned, reaching the state $M \parallel F,\ \neg l_1 \vee \ldots \vee \neg l_n$. As in the previous case, then Restart can be applied.

Incremental T-solver and on-line SAT solver. When a T-inconsistency is detected by the incremental T-solver, the DPLL procedure can simply backtrack to the last point where the assignment was still T-consistent, instead of restarting from scratch. As in the previous case, if a T-inconsistency is detected, a state $M \parallel F,\ \neg l_1 \vee \ldots \vee \neg l_n$ is reached. But now the procedure *repairs* the T-inconsistency of the partial model by exploiting the fact that $\neg l_1 \vee \ldots \vee \neg l_n$ is a conflicting clause, and hence either Fail or T-Backjump applies.

Theory propagation. In the approach presented so far, the T-solver provides information only *after* a T-inconsistent partial assignment has been generated. In this sense, the T-solver is used only to *validate* the search a posteriori, not to *guide* it a priori. In order to overcome this limitation, the T-solver can also be used in a given DPLL state $M \parallel F$ to detect literals l occurring in F such that $M \models_T l$, allowing the DPLL procedure to move to the state $M\,l \parallel F$. This is called *theory propagation*. It requires the following additional rule Theory Propagate:

$$M \parallel F \implies M\,l \parallel F \ \textbf{ if } \begin{cases} M \models_T l \\ l \text{ or } \neg l \text{ occurs in } F \\ l \text{ is undefined in } M \end{cases}$$

Exhaustive Theory Propagation. For some theories it even pays off, for every state $M \parallel F$, to eagerly detect and propagate *all* literals l occurring in F such that $M \models_T l$. Then, in every state $M \parallel F$ the model M will be T-consistent, and hence the T-solver will never (need to) detect any T-inconsistencies. It is modeled simply by assuming that Theory Propagate is applied eagerly.

Similar correctness, termination, and completeness results apply as given in the previous section for the propositional case (see [NOT06] for details).

4 The DPLL(T) approach

DPLL(T) is based on a general DPLL engine, called DPLL(X), combined with a module $Solver_T$ that can handle conjunctions of literals in T. This is similar to the $CLP(X)$ scheme for constraint logic programming: a clean and modular, but efficient, use of specialized solvers within a general-purpose engine. DPLL(T) combines the advantages of the eager and lazy approaches to SMT. As soon as the theory starts playing a significant role, DPLL(T) is very efficient (see the SMT-COMP results), and it has the flexibility of the lazy approaches, by simply plugging in other solvers that conform to a minimal interface.

Here we describe the DPLL(T) approach without exhaustive theory propagation (see [NO05a] for an exhaustive approach for *difference logic*). For the initial setup of DPLL(T), $Solver_T$ reads the input CNF, stores the list of all literals occurring in it, and hands it over to DPLL(X), who treats it as a purely propositional CNF. After that, DPLL(T) implements the rules as follows:

- Each time DPLL(X) communicates to $Solver_T$ that another literal l is added to the partial model M (e.g., due to UnitPropagate or to Decide), $Solver_T$ answers indicating whether M is still T-consistent. If not, $Solver_T$ returns a (preferably small) *explanation* why, that is, a subset $\{l_1, \ldots, l_n\}$ of M that becomes T-inconsistent by adding l to it. DPLL(X) then handles $\neg l_1 \vee \ldots \vee \neg l_n$ as a conflicting clause, applying T-Backjump or Fail.
- DPLL(X) can also ask $Solver_T$ to return a (possibly incomplete) list of literals that are T-consequences, to which it then applies Theory Propagate.
- DPLL(X) applies Fail or T-Backjump if after UnitPropagate a conflict is detected. For T-Backjump, the backjump clause is built as in Example 5, but with an important difference: a literal l can now be in M not only by Decide or by UnitPropagate, but also due to an application of Theory Propagate. In the last case, the conflict resolution process requires that $Solver_T$ must be able to also give *explanations* of theory propagations, that is, to recover a (preferably small) subset of literals $\{l_1, \ldots, l_n\}$ of M that T-entailed l. DPLL(X) then treats l in the resolution process as if $\neg l_1 \vee \ldots \vee \neg l_n \vee l$ had caused a unit propagation of l.
- At each T-Backjump application, T-Learn learns the backjump clause (which is a T-consequence of the current formula). DPLL(X) also tells $Solver_T$ how many literals of the partial interpretation have been unassigned in the backjump, which allows $Solver_T$ to undo them.
- DPLL(X) applies Decide only if none of Theory Propagate, UnitPropagate, Fail or T-Backjump is applicable. An activity-based heuristic for choosing the decision literal as in propositional DPLL is used.
- In a typical DPLL(T) implementation, DPLL(X) applies Restart when certain system parameters reach some prescribed limits, such as the number of conflicts or lemmas, the number of new units derived, etc. T-Forget can be applied, e.g., after each restart, removing part of the lemmas according to their activity (number of times involved in a conflict, etc.). Usually the newest lemmas are not removed.

5 Challenges

We now describe a number of theoretical and practical challenges in SMT.

First we consider extensions for improving the solvers for some of the most important theories: equality, linear arithmetic, and bitvectors. This involves questions of both theoretical and practical nature.

After that, we discuss some challenges arising in the context of the extension of SMT for handling formulas with universal quantifiers, i.e., for first-order theorem proving.

Finally we discuss new ideas for extending SMT to other application areas including optimization and constraint programming. This also involves the development of solvers for new theories.

5.1 Challenges for improving current theory solvers

Let us first discuss T-solvers for EUF logic, where the theory is just equality (a congruence). As for any T-solver, its requirements are as explained in the previous section: each time an additional literal comes in, it must check whether the conjunction remains T-consistent, and, if not, give an *explanation* (a T-inconsistent subset of the literals); it must also be able to find theory propagations and, when demanded, give explanations of these too; and it must be capable of backtracking, i.e., undoing (dis)equalities.

Positive equality literals can be propagated efficiently by congruence closure (CC) [DST80]. In [NO07] an incremental, backtrackable CC algorithm is given which can also efficiently retrieve explanations from CC, which is non-trivial.

Challenge 1: This challenge was first discussed at this conference in 2005 [NO05b]. It is widely understood that small explanations tend to behave better in practice. Finding for CC an explanation with the *minimum number of literals* is NP-hard (Ashish Tiwari, personal communication). Hence minimality w.r.t. $\subseteq$ is considered. The explanations produced in [NO07] may, in a small percentage of cases, contain redundant equations. How to get irredundant ones, or small(er) ones in some other sense? Studying this may produce useful new insights, although it may only have a limited practical impact on the performance of SMT solvers.

Challenge 2: Determine the exact complexity of CC. The aforementioned CC algorithms are $O(n \log n)$, but it is still unknown whether this is optimal. Some researchers conjecture that something like $O(n\ \alpha(n,n))$, as in Union-Find, might be possible for CC, and hence also for the ground word problem.

Challenge 3: The development of proof-producing SMT solvers is an important research topic. How to do efficient CC proof mining? For more details on this challenge, see [ST05], where Stump and Tan (two years ago at RTA) gave an elegant rewrite-based approach for equivalence closure; see also [SL06], an ingredient for its extension to CC.

Together with EUF logic, so far the most important classes of T-solvers are those for (fragments of) linear arithmetic over the integer or real numbers (see for instance the list of logics in SMT-LIB).

Challenge 4: It is well-known that, for many problems arising from the real world, a non-negligible percentage of the literals in linear arithmetic actually falls into difference logic[1] (see also [BBC$^+$05]). This observation cries out for techniques for linear constraint solving that are "difference-logic-aware". In this direction, promising results have already been accomplished in [DdM06b], thanks to, among others, a simplex procedure with a particular treatment of bound constraints, i.e., of the form $a \leq k$ or $a \geq k$, which in some cases is even faster than specialized tools for difference logic. However, for dense difference-logic problems, such as those coming from scheduling, there is room for improvement [DdM06a].

Challenge 5: There is practical evidence that a good way to handle equalities (respectively, disequalities) is by splitting these constraints as conjunctions (respectively, disjunctions) of inequalities. For instance, in the case of the integers, the satisfiability of a conjunction of difference logic disequalities and inequalities is NP-complete, whereas by restricting to inequalities the problem becomes polynomial; thus, splitting allows one to pass the NP-hardness of the solver to the boolean engine, which is designed to be efficient in handling the search space, as explained in previous sections. In the case of linear real arithmetic, in order to detect inconsistencies with disequalities, it is necessary to detect all implicit equalities implied by the constraints in the assignment, which may entangle a costly overhead; state-of-the-art solvers confirm this fact experimentally [DdM06b]. Therefore, boolean splittings can be exploited to improve efficiency. A natural problem is thus whether there exist new better ways of using the boolean engine in order to simplify the theories and so get faster solvers.

Challenge 6: So far, all SMT tools for full linear arithmetic employ infinite-precision numbers to guarantee the soundness of the results (since most of them are applied in verification applications). Although there exist sophisticated numerical libraries for this purpose, e.g., GMP[2], the involved overhead must not be neglected. A challenge would be to employ non-precise arithmetic so as to obtain more efficient solvers, as done in the context of Operations Research [ILO]. Is there any clever way of using an efficient non-precise off-the-shelf solver, and then only do a few checks with infinite-precision to guarantee soundness? A possibility could also be to develop solvers based on interior-point algorithms [Ter96,RTV97], which can only be implemented efficiently by means of floating-point arithmetic.

Finally, one of the most challenging theories in SMT, mainly due to its application to hardware verification, is the one of bitvectors. Elements of this domain can be viewed as arrays of bits, to which bitwise logical operators can be ap-

---

[1] See `eecs.berkeley.edu/~sseshia/research/uclid.html`.

[2] See `http://gmplib.org/`.

plied; but they can also be seen as integers, requiring support for the elementary arithmetic operations.

This inherent duality is also reflected on the existing techniques. On the one hand, translating the problem into propositional logic (known as *bit-blasting*) is well-suited for problems where bitwise operators dominate. On the other hand, when the problem has a prevailing arithmetic component, encoding it in linear integer arithmetic is the method of choice.

Challenge 7: Unfortunately, when there is no significant dominance none of the current methods is satisfactory. The challenge is to obtain hybrid procedures that combine the benefits of both approaches, e.g., by bit-blasting only very lazily.

Challenge 8: Are there any fragments of the theory of bitvectors that can be handled more efficiently, but are still useful for certain practical applications?

5.2 Challenges for SMT with quantifiers

SMT is typically considered to be the problem of checking the satisfiability of a ground first-order formula modulo a background theory T. If a T-solver for this particular theory is available, no quantifier reasoning is necessary at all. However, for several reasons, this is sometimes a too optimistic setting:

- In some applications, the ground fragment is not expressive enough and one needs to introduce first-order quantifiers in the formula. This is the case, just to give an example, in proof obligations arising from software verification where loop invariants may contain quantifiers.
- It is not always the case that a T-solver for the theory under consideration is available. In this case, a possible solution is to work with a finite axiomatization of T (if it exists), and apply generic first-order theorem proving techniques such as resolution or paramodulation.

Hence, it is necessary to develop techniques and tools that support quantifiers. Although some initial work has already been carried out, we believe there is still a lot of space for improvement.

Challenge 9: For dealing with non-ground formulas, the underlying idea of the existing techniques is based on Herbrand's theorem. That is, the unsatisfiability of a formula is to be detected by generating an unsatisfiable set of ground instances. In order to only generate a small but still sufficient set of instances, one first considers the congruence E generated by all equalities between ground terms in the current partial model. Then, given a non-ground term t occurring in the formula, its relevant ground instances $t\sigma$ are those such that $t\sigma =_E s$ for some ground term $s \in E$.

For given t, s, and E, checking whether such a σ exists is called the *E-matching* problem of t with s. It is well-known to be NP-hard even for fixed s and E: if E is the congruence generated by the 10 ground equations $and(0,0) = 0$, $and(0,1) = 0$, $\ldots$ representing the truth tables of *and*, *or* and *not*, then a propositional formula (a term with variables built over *and*, *or* and *not*) is satisfiable if, and only if, it E-matches with 1.

The idea of using E-matching for generating a sufficient yet small set of ground instances was first used in the Simplify theorem prover [DNS96] and it has recently been adapted in other SMT solvers such as Yices [DdM06a] or CVC3 [BT07]. A challenging task, partially studied in [dMB07], is to develop efficient data structures and algorithms that support all necessary operations for E-matching in the context of an SMT solver.

Challenge 10: As already mentioned, the generation of suitable instances is done via E-matching. Since some function and predicate symbols have a predefined semantics given by the theory T, it would be worth considering at least part of this semantics when matching terms. For example, could one use the fact that, when working modulo the theory of linear arithmetic, the function symbol $+$ is associative and commutative and thus generate instances using AC-matching?

Challenge 11: Despite the well-known severe theoretical limitations, it would be interesting to identify fragments and theories for which refutational completeness can be obtained. Even more, by using appropriate redundancy techniques, would it be possible to detect satisfiability in some particular cases?

5.3 SMT for Constraint Programming (CP) and Optimization

In CP (in a broad sense), relations between variables over given domains can be stated in the form of constraints, and the aim is to find values for these variables that satisfy these constraints and/or to optimize some objective function. CP modeling and solving techniques are being applied to problems in a large and broad variety of fields in engineering, (hardware and software) verification, timetabling, traffic and logistics, or finance, among others.

Today it is becoming clearer that SAT and CP techniques share many technological similarities and applications (see the "CP 2006 Workshop on the Integration of SAT and CP techniques"). SAT techniques, when applicable, have the advantage of being very efficient, robust, and highly automatic. On the other hand, the low-level language of propositional logic makes modeling tedious and difficult, and produces non-compact SAT problems, even with extensions such as (weighted) MAX-SAT or pseudo-Boolean constraints that can express optimization. In CP, elegant general formalisms facilitate modeling, and sophisticated special-purpose filtering and propagation algorithms exist for a large diversity of expressive global constraints. But CP implementations are frequently sensitive to variations in the input problem, and tend to need *tuning* by hand to find good heuristics.

Our aim here is to outline several ideas for using SMT, and in particular, our DPLL(T) approach, to combine SAT and CP techniques, hopefully getting the advantages of both and the drawbacks of none.

One lesson most SAT and CP researchers have learned is that techniques that work well on artificial or random problems may not do so on real-world problems, and vice versa[3]. In the SAT world, this is not surprising, since lemma learning is crucial for exploiting the "structure" of real-world problems, a structure that does not exist in random problems. Indeed, on real-world SAT problems, the complete DPLL procedure outperforms incomplete local search methods even for satisfiable problems (see `www.satcompetition.org`). We now compare complete systematic search methods for SAT and CP on four basic aspects.

Backtracking, backjumping and lemmas:
SAT: Conflict analysis techniques allow one to backjump, and at the same time provide the lemmas (in the form of new clauses, i.e., in the same input language!) for preventing similar conflicts in the future.
CP: Techniques for going beyond chronological backtracking exist, as well as notions of lemmas (*nogoods*) for pruning portions of the search space that are known to contain no solutions, but frequently the generality and diversity of the language makes this too difficult. Also the complexity of the constraint filtering and propagation algorithms frequently impedes it, since a notion of *explanation* is required for a precise conflict analysis (as we have seen), and there is no uniform representation language for nogoods in CP, and no uniform conflict analysis and backjump technique (these aspects are highly implementation-dependent).

Heuristics:
SAT: One single, robust, general-purpose heuristic is used, based on literal activity (roughly, split on the literal with the highest number of recent occurrences in conflicts and lemmas). One can see this as "working off" *locally* one constraint "cluster" at a time, and, while doing this, extracting lemmas from it, which are kept only while they are *active* (i.e., useful in pruning the search).
CP: Typical heuristics are based on the *first-fail* principle (e.g., *minimum domain*). In practice, tuning is usually needed to find a good heuristic for a given problem, or problem instance. On industrial SAT problems such heuristics behave poorly, consecutively visiting rather unrelated points in the search space, and thus also making it difficult to keep enough useful active lemmas.

Propagation/pruning:
SAT: Essentially, only unit propagation is used. Other techniques such as 2-literal-clause reasoning are usually found too expensive.
CP: Sophisticated techniques for propagating and filtering many types of constraints (aimed at important applications) have been developed, maintaining different degrees of (arc, bound, etc.) consistency.

Data structures:
SAT: Refined data structures exist for unit propagation (two-watched literals), clause representation, and bookkeeping for the heuristics.

[3] But still, many experiments of CP techniques for real-world applications are being carried out on artificial problems, and problems are sometimes called "non-artificial" because they are translations of, e.g., graph problems which were random or hand-crafted!

CP: Again, the generality and diversity of the language makes it hard to develop such data structures. Even for the simple language of propositional CNF, it has taken years of research in SAT solving to reach the current state of the art.

Challenge 12: Develop an SMT system with the advantages of one of CP's sophisticated global constraint propagation algorithms and the robustness and efficiency of SAT's backjumping, lemmas and heuristics. The idea is to express the global constraints as a theory. For instance, we are currently working on the following system.

Example 11. Consider the typical (academic) CP problem of Quasi-Group Completion (QGC, also known as Latin squares). It is important in practice because it appears hidden in many real-world (e.g., scheduling) problems. The question is whether an $n \times n$ table like this one can be completed such that each row and column contains the numbers $1 \dots n$ (in this case $n = 5$):

	3	4		
3	4	5		
4	5			
5				

Currently a good (possibly the best) technique for QGC is the so-called *3-D encoding* into SAT [KRA+01], where a propositional variable x_{ijk} means "row i column j has value k", and the following clauses are given:

1. *At least one k per* $[i, j]$: clauses like $x_{ij1} \vee \dots \vee x_{ijn}$, and *at most one k per* $[i, j]$: 2-literal clauses like $\neg x_{ij1} \vee \neg x_{ij2}$.
2. The analogous clauses for exactly *one j per* $[i, k]$ and *one i per* $[j, k]$.
3. One unit clause per filled-in value, e.g., x_{313}.

With this encoding, in our 5x5 example, DPLL's UnitPropagate infers no value. But `alldifferent` constraint filtering on the first three columns and the first row $v_{11}, v_{12}, v_{13}, v_{14}, v_{15}$ reveals that v_{11} and v_{12} consume values 1 and 2 and hence v_{13} must be 3.

Consider an SMT system using this 3-D encoding and where T is the theory of `alldifferent`. As usual in SMT, the T-solver knows what the x_{ijk}'s mean. From time to time, one can invoke the T-solver for doing Theory Propagate, but one should apply cheap SAT rules first: UnitPropagate, Backjump, etc. In this case, the T-solver does incremental filtering [Rég94] but must be able to produce *explanations*. In our example, the theory-propagated literal x_{133} (meaning $v_{13} = 3$) is entailed by $\{\ \neg x_{113}\ \ \neg x_{114}\ \dots\ \neg x_{135}\ \}$. □

In this way, the specialized filtering algorithms only need to be extended for generating explanations, but the remaining machinery can be used as it is in DPLL(T): one uniform language (clauses) for expressing no-goods, the conflict analysis mechanism, etc. SAT's heuristics and unit propagation mechanisms will do what they are good at, which is carrying out the actual search, i.e., the labeling. Learned lemmas help transferring knowledge from the theory to the DPLL(X) engine, which handles it efficiently.

Challenge 13: In the previous example we have considered the `alldifferent` constraint. Develop explanation-generating T-solvers for other typical global constraints.

Challenge 14: In the previous example, we used a complete underlying encoding into SAT of the QGC problem. Try to exploit the same ideas, but using the boolean part of SMT only for an incomplete encoding.

Challenge 15: In [NO06] we have shown how to model in SMT optimization problems (Max-SAT and Max-SMT) by expressing as an (increasingly stronger) theory T the best solution so far in a branch-and-bound search. How can lower bounds be more effectively applied in that framework?

6 Concluding remark

We hope that the reader has become challenged and motivated for helping develop this exciting research area and/or for applying SMT techniques and tools.

References

[BBC+05] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. System description: MathSAT 3. In Robert Nieuwenhuis, editor, *Proceedings of the 20th Conference on Automated Deduction*, LNCS 3632, pages 315–321, Tallinn, Estonia, 2005. Springer.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Procs. 6th Int. Conf. Computer Aided Verification (CAV)*, LNCS 818, pages 68–80, 1994.

[BT07] C. Barrett and C. Tinelli. CVC3. In *Computer Aided Verification, 19th International Conference, (CAV)*. Springer LNCS, 2007. To appear.

[DdM06a] B. Dutertre and L. de Moura. The YICES SMT Solver, 2006. `http://yices.csl.sri.com/tool-paper.pdf`.

[DdM06b] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *Int. Conf. Computer Aided Verification (CAV)*, pages 81–94. Springer LNCS 4144, 2006.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Comm. of the ACM*, 5(7):394–397, 1962.

[dMB07] L. de Moura and N. Bjorner. Efficient E-matching for SMT Solvers. In *xx*, 2007. Submitted.

[DNS96] D. L. Detlefs, G. Nelson, and J. Saxe. Simplify: the ESC theorem prover. Technical report, Compaq, December 1996.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[DST80] Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpressions problem. *J. of the Association for Computing Machinery*, 27(4):758–771, 1980.

[ES03] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2003.

[ILO] ILOG. "ilog cplex". `http://www.ilog.com/products/cplex`.

[KRA+01] Henry A. Kautz, Yongshao Ruan, Dimitris Achlioptas, Carla P. Gomes, Bart Selman, and Mark E. Stickel. Balance and Filtering in Structured Satisfiable Problems. In Bernhard Nebel, editor, *17th International Joint Conference on Artificial Intelligence, IJCAI'01*, pages 351–358. Morgan Kaufmann, 2001.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. 38th Design Automation Conference (DAC'01)*, 2001.

[MSS99] Joao Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, may 1999.

[NO05a] Robert Nieuwenhuis and Albert Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In *17th Int. Conf. on Computer Aided Verification, (CAV)*, Springer LNCS 3576, pages 321–334, 2005.

[NO05b] Robert Nieuwenhuis and Albert Oliveras. Proof-Producing Congruence Closure. In *16th Int. Conf. on Term Rewriting and Applications (RTA)*, Springer LNCS 3467, pages 453–468, 2005.

[NO06] Robert Nieuwenhuis and Albert Oliveras. On sat modulo theories and optimization problems. In *Theory and Applications of Satisfiability Testing (SAT), LNCS 4121,*, pages 156–169, 2006.

[NO07] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, April 2007.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.

[Rég94] J-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of 12th National Conference on AI (AAAI'94)*, volume 1, pages 362–367, Seattle, July 31 - August 4 1994.

[RT03] Silvio Ranise and Cesare Tinelli. The SMT-LIB Format: An Initial Proposal. In *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, Miami, 2003.

[RTV97] C. Roos, T. Terlaky, and J. P. Vial. *Theory and Algorithms for Linear Optimization: An Interior Point +Approach*. Wiley, 1997.

[SL06] Aaron Stump and Bernd Löchner. Knuth-bendix completion of theories of commuting group endomorphisms. *Inf. Process. Lett.*, 98(5):195–198, 2006.

[ST05] Aaron Stump and Li-Yang Tan. The algebra of equality proofs. In *16th Int. Conf. on Term Rewriting and Applications (RTA)*, Springer LNCS 3467, pages 469–483, 2005.

[Ter96] T. Terlaky, editor. *Interior Point Methods of Mathematical Programming*, volume 5 of *Applied Optimization*. Kluwer Academic Publishers, 1996.