Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Satisfiability Modulo Theory Chains with DPLL(T)

Scott Cotton and Oded Maler

Verimag Research Report n^o TR2006-04

March 20, 2006

Satisfiability Modulo Theory Chains with DPLL(T)

Scott Cotton and Oded Maler

March 20, 2006

Abstract

We extend the DPLL(T) framework for satisfiability modulo theories to address richer theories by means of increased flexibility in the interaction between the propositional and theory-specific solvers. We decompose a rich theory into a chain of increasingly more complex subtheories, and define a corresponding propagation strategy which favors the simpler subtheories using two mechanisms. First, subtheory propagation is prioritized so that more expensive propagation is avoided whenever possible. Second, constraints are filtered along the path from simpler to more complex propagation, thus easing the task of propagation for each subtheory. We present this strategy formally in a refined abstract DPLL(T) system and provide a concrete algorithmic skeleton with a proof of correctness.

Keywords:

Reviewers:

Notes:

How to cite this report:

@techreport { ,
title = { Satisfiability Modulo Theory Chains with DPLL(T)},
authors = { Scott Cotton and Oded Maler},
institution = { Verimag Research Report },
number = {TR2006-04},
year = { },
note = { }
}

1 Introduction

The problem of determining the satisfiability of a Boolean combination of constraints is a fundamental and computationally-intractable problem with applications in many fields including model checking, theorem proving, scheduling, circuit analysis and planning. The field of *satisfiability modulo theories* (SMT) studies effective mechanisms for solving this problem in its full generality, that is, in a manner independent of the form of the constraints. Thus far, the field has focused on modular integration of standard decision procedures for propositional logic together with pluggable decision procedures for the theory that corresponds to the specific type of constraints.

Amongst a plethora of recent research on satisfiability modulo theories [20, 8, 21, 18, 16, 17, 1, 12, 11, 3], the DPLL(T) framework has proven successful [8, 16]. Within this framework, a major issue in the interaction between the DPLL procedure and the theory-specifc solver is that of *theory propagation*, that is, at what frequency and to what extent are the theory consequences of newly-assigned constraints computed. Since reasoning inside a theory is typically more expensive than purely-Boolean reasoning, different choices of theory propagation strategies may significanyly affect performance.

In this paper we present a prioritized strategy for theory propagation of richer theories which can be decomposed into chains of increasingly complex sub-theories. The strategy consists primarily of an abstract *constraint filter* which is described in terms of the roles that constraints play with respect to the subtheories in the chain. We also provide a simple, prioritized implementation of this filter given a chain of sub-solvers for each theory, and prove its correctness. This rest of this paper is organized as follows. In section 2, we review the DPLL(T) framework and discuss theory propagation strategies. Our new methodology is presented in section 3 as a refinement of an abstract DPLL(T) system. In section 4, we present a simple implementation of the abstract constraint filter, followed by a proof of its correctness in section 5. Conclusions and future work are presented briefly in section 6.

2 Background

2.1 DPLL(T)

DPLL(T) is a framework for determining the satisfiability of a Boolean combination of constraints, where the constraints belong to some specified theory. In particular it makes use of two solvers, a slightly modified Davis-Putnam-Loveland-Logeman (DPLL) style Boolean satisfiability solver [7, 6], and a solver for a given theory T. As the modified DPLL solver is parameterized on the theory solver, we refer to it as DPLL(X); the theory solver is called Solver$_T$.

The DPLL satisfiability checking procedure is concerned with propositional logic. Its input formula φ is assumed to be in conjunctive normal form. This is to say that φ is a conjunction of clauses, where each clause is a disjunct of literals and each literal is either a propositional variable or its negation. Given such a formula, a DPLL solver will search the space of truth assignments to the variables by incrementally building up partial assignments and backtracking whenever a partial assignment falsifies a clause. Assignments are extended both automatically by *unit propagation* and by guessing truth values. Unit propagation occurs by keeping track of the effect of partial assignments on the clauses in the input formula. For example, the clause $x \vee \neg y$ is solved under an assignment including $y \mapsto \texttt{false}$ and is reduced to the clause x under an assignment which includes $y \mapsto \texttt{true}$ but which contains no mapping for x. Whenever a clause is reduced to contain a single literal, *i.e.* it is in the form v or the form $\neg v$, the DPLL procedure deduces by unit propagation the assigment to v which solves the clause. If a satisfying assignment is found, the formula is satisfiable. If the procedure exhausts the assignment space without finding a satisfying assignment, the formula is not satisfiable.

Modern DPLL solvers are highly optimized, employing various dynamic variable ordering schemes [13, 9, 19], non-chronologic backtracking [10], efficient clause status tracking [13], and dynamic learning of clauses [2]. These procedures have made advances by leaps and bounds in the last years and are routinely able to solve real-world problems consisting of tens of thousands of variables and clauses within a few seconds. As DPLL(X) is a slightly modified version of DPLL, these optimizations may readily be employed in the DPLL(T) framework.

DPLL(X) interacts with a theory-specific solver Solver$_T$, which is a satisfiability solver for *conjunctions* of constraints in the language of a given theory T. The DPLL(X) procedure interprets every constraint

c as a propositional variable, and interacts with Solver$_T$ in the following way.

SetTrue. DPLL(X) calls Solver$_T$ after assigning a truth value of some constraint c. Solver$_T$ then performs a consistency check on the current assignment. If the assignment is consistent, Solver$_T$ may optionally return a list $I(c)$ of constraints which are implied by the theory under the current assignment. Otherwise, it indicates that the current assignment is unsatisfiable. In this case, Solver$_T$ may moreover identify a small subset of the assignment which is unsatisfiable. The DPLL(X) solver is expected to backtrack in response.

Explain. DPLL(X) may ask Solver$_T$ to provide a list of reasons R_c for any implied constraint c previously returned by `SetTrue`. This occurs when learning new clauses. The reasons R_c for c must not include c and it must be the case that c follows from the reasons under the theory, *i.e.* that $R_c \models_T c$. Moreover, every $c' \in R_c$ must have been assigned prior to c.

BackTrack. DPLL(X) calls Solver$_T$ every time backtracking occurs, indicating those constraints which are removed from the partial truth assignment.

In hopes of making this paper both self-contained and readably short, the high level description given here of DPLL, and DPPL(T) is prosaic and informal. The reader is referred to [10, 13] for a general overview of modern DPLL SAT solving and to [8, 20] for a general overview of DPLL(T) and SMT.

2.2 A DPLL(T) Transition System

We describe our extension of DPLL(T) in terms of the abstract DPLL(T) transition system due to Nieuwenhuis *et. al.* [17]. The system is both more faithful to real implementations than the prosaic description above and at the same time facilitates formal reasoning about a wide class of DPLL(T) implementations. We describe this system briefly below and extend it in section 3, to better accomodate flexible theory propagation.

To begin at the beginning, a *theory* is a deductively closed set of valid formulas, each of which has no free variables. The formulas are expressed in a language consisting of variables, logical constants ($\wedge, \neg, \forall, \ldots$), and a set of function and relation symbols called a *signature*. Each theory has an associated class of *models* each of which is an interpretation of all the symbols in the signature over some set, called the *domain* of the model. One usually reasons about properties of models by means of the axioms under which the theory is deductively closed. For example, a theory of equality T_{eq} may be specified by a relation symbol "R", and a set of axioms expressing that "R" is reflexive, symmetric, and transitive. Any model $M = \langle D, R^D \rangle$ where R^D is in fact a reflexive, symmetric and transitive relation over D is a T_{eq}-model. In particular, a model M is called a T-model, also written $M \models T$, whenever for every formula $\varphi \in T$, M interprets φ in such a way that φ is true. The satisfaction relation $\models_T$ is then used to indicate whether T-models satisfy arbitrary formulas in the language of T. In particular, we write $M \models_T \varphi$ if $M \models T$ and $M \models \varphi$, *i.e.* if M is a T-model and there is an assignment mapping the free variables of φ to the domain of M such that φ is true. In this case, we say that φ is *satisfiable in* T. Similarly, the expression $\Gamma \models_T \varphi$, when Γ is a set of formulas in the language of T, denotes that for any T-model M such such that $M \models \Gamma$, $M \models \varphi$.

A *literal* is either a *propositional literal* or a *constraint*. A propositional literal is either a Boolean variable v or its negation $\neg v$. A *constraint* is an atom, possibly negated, in the language of the given theory T. For example, $x = 2$ is a constraint in the language of arithmetic. A *clause* C is a non-trivial disjunction $l_0 \vee l_1 \vee \ldots \vee l_k$ of literals. For any literal $l \in C$, we assume its negation is not a member of C.

With these definitions at hand, we briefly present the DPLL(T) framework as an abstract *state transition system* $\mathbb{D}_T$ with conditional transition rules. Each state in $\mathbb{D}_T$ consists of a partial truth assignment A in the form of an annotated set of literals, coupled with a set of clauses F. Every literal in a truth assignment either appears in F, or is in the form $\neg c$ for some constraint in F. There is a single initial state which consists of the empty truth assignment coupled with the inital set of clauses H. As transitions are taken, the partial assignment is built up, and the set of clauses may change, subject to the conditions in the transition rules. Every literal in a given partial assignment is labelled as either *guessed* or *not guessed*, depending on what transition rule is used to add it to the truth assignment. The transition rules are given in Figure 1.

The system $\mathbb{D}_T$ is, in general, non-deterministic because several transition rules may be enabled in a given state. We call the mechanism by which a solver chooses among transitions a *strategy*. One easy way

Rule	Transition			Conditions
U-prop	$A \circ F, C \vee l$	$\rightarrow$	$A, l \circ F, C \vee l$	$A \models_T \neg C$
T-prop	$A \circ F$	$\rightarrow$	$A, l \circ F$	$A \models_T l$
Guess	$A \circ F$	$\rightarrow$	$A, l_g \circ F$	-
Back	$A, l_g, A' \circ F, C$	$\rightarrow$	$A, l' \circ F, C$	$A, l_g, A' \models_T \neg C$ There exists a clause $C' \vee l'$ s.t. $A \models_T \neg C'$ $F, C \models_T C' \vee l'$
Learn	$A \circ F$	$\rightarrow$	$A \circ F, C$	$F \models_T C$
Forget	$A \circ F, C$	$\rightarrow$	$A \circ F$	$F \models_T C$
Fail	$A \circ F, C$	$\rightarrow$	fail	$A \models_T \neg C$ There are no guessed literals in A
Restart	$A \circ F$	$\rightarrow$	$\emptyset \circ F$	-

Figure 1: Transition relation $\rightarrow$ for DPLL(T) system $\mathbb{D}_T$, originating from [17]. We write $A \circ F$ for a state, indicating the assignment A and a set of clauses C. A clause $C \vee l$ is taken to be a possibly empty clause C, which does not include l or $\neg l$, disjoined with the literal l. We write l_g for guessed literals and simply l otherwise. We assume all literals or their negations occur in the original input formula and that any transition involving an assignment assigns only undefined literals.

to define a strategy is to define *priorities* for to the various transition rules. In general, some priorities are more or less universal. For example, the fail transition is almost always implemented as taking priority over all other rules, and the Guess transition is almost never implemented when the U-prop rule is applicable. On the other hand, the priority of the T-prop rule with respect to the U-prop and Guess rules is more complex and varied.

2.3 Theory Propagation Strategies

Theory propagation, *i.e.* application of the T-prop rule, is generally expensive in practice. However, it also has the benefit of reducing the guessing space of the whole DPLL(T) procedure. Every literal implied by theory propagation is a propositional variable whose truth value DPLL(X) need not guess; moreover every such literal may enable further unit propagation which, in turn, reduces the guessing space further and may also lead to more theory propagation. As increased guessing leads in the worst case to exponentially longer runs and hence even more theory specific processing, there is a balance to be struck between applying theory propagation and avoiding it. That balance is unfortunately both theory- and problem-specific.

As a result of this quandary, the role of theory propagation in SMT solvers is extremely varied. Some solvers, such as MathSat [11] don't make use of expensive theory propagation at all but rather simply perform theory consistency checks on the partial assignments. Others, such as MxSolver [15] and BarceLogicTools [16] (for difference logic) apply theory propagation whenever the T-prop rule is enabled to avoid expensive guessing. Between these two extremes, some solvers such as Jat [3] apply theory propagation lazily and incompletely and others, such as BarceLogicTools (for the logic of uninterpreted functions) apply an incomplete propagation procedure exhaustively. However, each of these strategies lack flexibility. The lazy strategy of [11] disallows theory propagation altogether. The eager strategy of [16] requires special treatment of theory consequences by the Boolean solver and requires that theory consequences be complete. The lazy round-robbin strategy of Jat makes incremental theory propagation difficult at best.

In all these cases, the theory propagation is monolithic in the sense that there is no decomposition of theory propagation into distinct subprocedures or subtheories. In addition, the relative priorities of unit propagation and theory propagation are hard-coded and in no way configurable. Finally, with the notable exception of the greedy strategy, the strategies make no distinction between the role of constraints with respect to theory propagation. In particular, T-consequences are subject to the same processing as any other constraint, although they are guaranteed to be T-consistent and, under reasonable assumptions, not to lead to more T-consequences.

2.4 Overview of a New Methodology

In this paper we present a methodology for theory propagation as a refinement of the system $\mathbb{D}_T$, providing a framework for reasoning about a broad class of propagation strategies. We describe an implementation scheme for this methodology. The following concepts underly this work:

Decomposition. A theory T is decomposed into a chain of theories $T_1 \subset T_2 \subset \ldots \subset T_k$ with $T = T_k$. Each step in the chain properly extends the set of valid formulas and each T_i is a deductively closed theory. Decomposition provides a chain of refinements for a given formula in the language of T. As an example, the theory of the real field may be decomposed into a chain consisting of pure equality, difference logic, linear arithmetic, quadratic arithmetic, and arbitrary order arithmetic.

Prioritization. The decomposition above is prioritized: propagation for each theory T_i always takes priority over propagation for any T_j with $i < j$. We present a transition system $\mathbb{D}_{T^\circ}$, a variant of $\mathbb{D}_T$, which provides for modelling prioritization more flexibly, including for the case of a trivial decomposition. The system $\mathbb{D}_{T^\circ}$ can be mapped back to the original $\mathbb{D}_T$, thus preserving useful properties of $\mathbb{D}_T$ such as termination and correctness.

Identification of constraint roles. We refine the states of $\mathbb{D}_{T^\circ}$ by decorating constraints with labels that reflect their role with respect to theory propagation in each T_i. We use the constraint roles to build a *constraint filter* along the theory chain. This filter prevents theory-deduced constraints from undergoing a lot of redundant processing to which they are subject in the standard DPLL(T) framework, even in the case of a trivial decomposition.

3 Abstract Methodology

3.1 Preliminaries

Definition 3.1 (T-Decomposition)**.** *A decomposition T_k° of a theory T is a set of theories $\{T_i \mid 1 \leq i \leq k\}$ such that each $T_i \subset T_{i+1}$ and $T = T_k$. In addition, for each theory T_i with $i \in [1..k]$, we associate a language $\mathcal{L}_i$, a satisfaction relation $\models_i$ and a decision procedure $\vdash_i$. We require that each $\vdash_i$ be able to answer queries of the form $\Gamma \models_i c$ for a finite set $\Gamma \cup \{c\}$ of constraints in $\mathcal{L}_i$.*

The decomposition can be seen as a sequence of refinements. Just as DPLL(X) views constraints propositionally while Solver$_T$ views them in refined form as theory constraints, a decomposition allows one to view a constraint as a propositional literal, as a constraint in a simple theory, as well as a constraint in a more complex theory. The following lemma and corollary are central to our handling of decompositions.

Lemma 3.2 (Refinement)**.** *Given two theories T, T', if $T \subset T'$ then any T'-model is a T-model*

Proof. Any T'-model M satisfies every formula in T', and $T \subset T'$. Hence M satisfies every formula in T and so by definition is a T-model. □ □

Corollary 3.3 (Consequence lifting)**.** *Given two theories $T_i, T_j \in T_k^\circ$ with $i < j$ and a set of formula $\Gamma \cup \{\varphi\}$, if $\Gamma \models_i \varphi$, then $\Gamma \models_j \varphi$.*

Proof. Suppose that any T_i-model of Γ is a model of φ. We need to show that any T_j model M of Γ is also a model of φ. By lemma 3.2, since M is a T_j-model, M is also T_i-model, and so $M \models_i \Gamma$. By the hypothesis, M is a model of φ. □ □

Definition 3.4 (The set χ)**.** *Given a set of clauses F, we define the set χ as the smallest set containing every constraint c and its negation $\neg c$ such that c appears in some clause in F.*

The consequence finders ρ_χ^i defined below provide a means to model actual deduction procedures associated with a given theory T_i. A sound and *complete* consequence finder may be implemented using the decision procedure $\vdash_i$, simply by querying for every $c \in \chi$ whether $\Gamma \models_i c$, but we allow also incomplete finders to reflect more faithfully the working of real solvers.

Definition 3.5 (Consequence finders)**.** *Given a theory decomposition T_k° and the set χ, a consequence finder ρ_χ^i for $T_i \in T_k^\circ$ is an arbitrary function $\rho_\chi^i : 2^\chi \to 2^\chi$ satisfying the following three conditions.*

1. *Soundness. For every $i \in [1..k]$, $\rho_\chi^i(\Gamma) \subseteq \{c \in \mathcal{L}_i \mid \Gamma \cap \mathcal{L}_i \models_i c\}$.*
2. *Monotonicity. For every $i \in [1..k]$, and $\Gamma \subseteq \Gamma' \subseteq \chi$, $\rho_\chi^i(\Gamma) \subseteq \rho_\chi^i(\Gamma')$.*
3. *Chain closure. For every $i, j \in [1..k]$ with $j \leq i$, $\rho_\chi^i(\rho_\chi^j(\Gamma)) \subseteq \rho_\chi^i(\Gamma)$.*

Monotonicity simply states that as more constraints are handed to a consequence finder, more consequences are found. Chain closure which, by Corollary 3.3, holds for any sound and complete consequence finder, is not strictly necessary in our methodology but helps simplifying the results and their presentation.

3.2 Modelling Decompositions with Priorities

Given a T-decomposition T_k°, consider replacing the T-prop rule in $\mathbb{D}_T$ with one rule T-prop$_i$ for each $T_i \in T_k^\circ$:

$$A \circ F \to A, l \circ F \text{ if } A \models_i l$$

All the runs of the resulting transition system can easily be mapped back to runs in $\mathbb{D}_T$ simply by renaming applications of each T-prop$_i$ to the original T-prop rule. This follows directly from Corollary 3.3 as any T_i-propagation is a valid T-propagation. We refer to the modified trsansition system as $\mathbb{D}_{T^\circ}$ and view it as a model of DPLL(T) executions in which we can differentiate between the effects of different T_i consequences in a given T-decomposition.

However, we would also like to reason about priorities given to rules and hence need to take a few more things into consideration. In particular, for one propagation rule to take precedence over another, it is necessary to know when propagation by one rule is "finished" so that a lower priority rule may be applied. It is also necessary to know what constitutes the smallest possible unit of propagation for a given rule, so that propagation by a lower priority rule may defer to a higher priority rule, but only after having completed a conveniently small chunk of work.

It is reasonable to define a propagation by a rule as finished when that rule is no longer enabled. This in turn may be expressed by a closure condition associated with each propagation rule. In particular, we call a rule T-prop$_i$ *closed* in a state $A \circ F$ whenever $\rho_\chi^i(A) \subseteq A$, *i.e.* whenever it is not enabled. Similary, we consider the U-prop rule *closed* in a state $A \circ F$ if there are no unit clauses in F under the assignment A.

While closure is easily defined in the system $\mathbb{D}_{T^\circ}$, a flexible notion of a "conveniently small chunk of work" is not. Often, propagation procedures will find a set of consequences in reaction to an extension of a set of antecedents. For example, if $S = \{w < x, x < y\}$ is a set of antecedents and $S' = S \cup \{y < z\}$, then a consequence finder may find that $w < y$ is a consequence of S and then that $\{w < z, x < z\}$ are consequences of S'. In the latter case, the procedure has found more than one consequence in response to antecedent extension. But a transition $A \circ F \to A, l \circ F$ in the system $\mathbb{D}_{T^\circ}$ would require that only one consequence $l \in \{w < z, x < z\}$ become part of the assignment at a time. If some other propagation rule has higher priority, then the consequence finder should remember unnassigned consequences while other higher priority rules are brought to closure. This may be inconvenient since it is natural to implement consequence finders simply as a function of a set of antecedents. To allow for prioritized theory propagation based on antecedent extension rather than based on consequence extension, we define alternate T-prop$_i$ rules in below.

$$A \circ F \to A, B \circ F \text{ if } \begin{cases} \exists A' \subseteq A \,.\, B = \rho_\chi^i(A') \setminus A \\ B \neq \emptyset \end{cases} \tag{1}$$

With closure and antecedent based propagation rules, we can easily express realistic rule preferences for $\mathbb{D}_{T^\circ}$. If one rule R has a higher priority than R', then we add the closure condition of R to the set of preconditions of R'. We always consider that the Guess rule has lowest priority, and hence that it is conditioned on the closure of both theory and unit propagation.

Rule prioritization is effective in the DPLL(T) framework for two reasons. Consider the case that one rule R is much more expensive than a rule R'. If a literal is impliable under both R and R', then it need not be implied under R when R' has priority. In this case, one may "save" an application of the rule R

by applying the rule R' first. Secondly, and more profoundly, the DPLL(T) framework is a *backtracking search* framework. Consequently, there are many contexts in which applications of the cheaper R' may lead to a backtrack. In all these contexts, if R' has higher priority than R, then *no* applications of the more expensive rule occur.

3.3 Extended Constraint States

In every state of $\mathbb{D}_{T^\circ}$, there is an annotated partial truth assignment A which indicates whether or not constraints are assigned and whether or not assigned constraints are guessed. Below, we define *extended states* which refine the original states by indicating, in addition, the role of constraints with respect to each subtheory in a T-decomposition. This additional information is used by Solver$_T$ and is invisible to DPLL(X), hence all the good properties of $\mathbb{D}_{T^\circ}$ are retained.

To motivate this refinement we illustrate an inherent inefficieny in standard implementations of the DPLL(T) algorithm, manifested already in the case of a monolithic theory T. Suppose Solver$_T$ is invoked by DPLL(X) to find consequences of some newly-assigned constraint c, and it finds a consequence c'. It then hands c' to DPLL(X) which puts it in its implication queue. When the time comes for c' to be processed, DPLL(X) will hand c' back to Solver$_T$, where it will undergo a redundant consistency check, and moreover it will participate, together with its antecedents, in each and every further theory propagation although, being a consequent, it will *not* contribute *any new consequences*. This can needlessly increase the amount of work done by the consistency checker and the consequence finder. By identifying the role of constraints with respect to theory propagation, *i.e.* which constraints are consequences and which are antecedents with respect to each subtheory, we can avoid this processing, even in the case of a trivial decomposition.

To this end, for every constraint c we keep track of the role it plays in T_i propagation, namely, consequence, antecedent or no role at all. Whenever a constraint c is identified as a consequence in theory T_i, we consider it to be a consequence in T_j for all $j \geq i$. Likwise, whenever a constraint c is known to be an antecedent in a subtheory T_i, it is considered an antecedent in subtheories T_j with $j \leq i$. All this information is compactly represented by associating with each constraint c a pair of labels $\langle l_c, u_c \rangle$ with $l_c, u_c \in [0..k+1]$, such that c has participated as antecedent in all $\{T_i \mid i < l_c\}$, and as a consequence in all $\{T_i \mid i > u_c\}$. The role of c in $\{T_i \mid i \in [l_c, u_c]\}$ is not yet determined. We will maintain that whenever $u_c < l_c$, $u_c = l_c - 1$. This way, for every subtheory, a constraint may participate either as an antecedent or as a consequence but not both.

An extended state is then simply a state $A \circ F$ together with a labelling $\langle l_c, u_c \rangle$ of every constraint $c \in \chi$, satisfying certain properties detailed below. To express these properties, we make use of the following definition.

Definition 3.6 (T_i-basis and T_i-cobasis)**.** *In any given state in the system $\mathbb{D}_{T^\circ}$, we refer to the T_i-basis $\mathbf{b}_i$ as the set $\{c \in \chi \mid l_c > i\}$. Similarly, we write the T_i-cobasis $\bar{\mathbf{b}}_i$, denoting the set $\{c \in \chi \mid u_c < i\}$.*

Below we give several extended state properties which we require to hold throughout the transition system $\mathbb{D}_{T^\circ}$ with extended states. Taken together, these properties facilitate safe optimization of DPLL(T) for richer theories by identifying redundant constraints for each T_i, safely allowing a great deal of flexibility in when consequence finding and consistency checking occurs, for which constraints they occur, and guaranteeing that all possible consequence finding eventually occurs.

1. *Basis-cobasis assignment.* In any state $A \circ F$, $\bigcup_{i \in [1..k]} \mathbf{b}_i \subseteq A$, and $A \cap \chi \subseteq \bigcup_{i \in [0..k]} \mathbf{b}_i \cup \bar{\mathbf{b}}_i$. This property states the relationship between assignments and constraint roles. It simply requires that we know the role of all assigned constraints with respect to some subtheories, while allowing the latitude that the role of some unassigned constraints may also be known.

2. *Basis consistency.* In any state, for every $i \in [1..k]$, $\mathbf{b}_i \cap \mathcal{L}_i$ is T_i-consistent. This is a weak consistency condition, allowing the system to be T-inconsistent at times while requiring piecewise T_i-consistency. If some consistency checks are expensive, they can be temporarily avoided, allowing the possibility of arriving at an inconsistency by less expensive means.

3. *Full consistency.* In any state $A \circ F$ which precedes a Guess or is a final state, for every $i \in [1..k]$, $A \cap \mathcal{L}_i$ is T_i-consistent. This is a periodic strong consistency condition which ensures that the system is consistent, but only in the states in which full T-consistency is required.
4. *Cobasis irrelevancy.* In any state $A \circ F$, for every $i \in [1..k]$ and $j \geq i$, $A \setminus \bar{\mathbf{b}}_i \models_j A$. This is a safety condition which allows the system to correctly filter out any constraint in $\bar{\mathbf{b}}_i$ from all T_i specific processing.
5. *Basis-cobasis closure.* In any state, for every $i \in [1..k]$, $\rho^i_\chi(\mathbf{b}_i) \subseteq \mathbf{b}_i \cup \bar{\mathbf{b}}_i$. This property requires that all consequence finding of each basis has occurred. As a T_i-local property, it ensures that all T_i consequences of $\mathbf{b}_i$ have been identified.
6. *Full closure.* In any state $A \circ F$ which precedes a Guess or is a final state, for every $i \in [1..k]$, $\rho^i_\chi(A) \subseteq A$. This property states that all possible theory propagation must have occurred before the system solves a problem or Guesses.

4 Concrete Methodology

We present a relatively simple skeleton implementation of DPLL(X) and Solver$_T$ which realizes the system $\mathbb{D}_{T^\circ}$ with extended states, satisfying all extended state properties and using the theory propagation rules based on antecedent extension as presented in equation 1. In addition, this implementation enforces prioritization of propagation rules which respect the decomposition T°_k. We assume that consequence finding in theory T_i is less expensive than consequence finding in T_{i+1}. We also assume that consequence finding in T_i is comensurate with consistency checks in T_{i+1}, coupling consistency checks in T_{i+1} with propagation in T_i. We present these implementations in terms of of changes to the original DPLL(T) system described in section 2.1.

4.1 DPLL(X)

Recall that the DPLL(X) engine presented in section 2.1 interacts with Solver$_T$ via the three methods `SetTrue`, `Backtrack`, and `Explain`. In this framework, Solver$_T$ is unable to distinguish between states of the DPLL(X) engine. In particular, Solver$_T$ has no way of knowing whether or not the DPLL(X) engine will guess if Solver$_T$ does not apply the T-prop rule.

We remedy this situation by adding a method `TheoryProp` to Solver$_T$. This method is called by DPLL(X) as a last-resort form of constraint propagation. In particular, as in a standard DPLL engine, applicable instances of the propagation rules are kept in an implication queue, and processed in a first-in-first-out manner. Processing consists of checking for new instances of the U-prop rule and calling `SetTrue` for each assigned constraint. If the queue becomes empty and no inconsistencies are found, our DPLL(X) engine calls `TheoryProp`, which in turn may re-fill the implication queue with some instances of the T-prop rule. If `TheoryProp` does not re-fill the implication queue, DPLL(X) will guess because there are no more instances of propagation rules. Also, as in `SetTrue`, the method `TheoryProp` may indicate that an inconsistency has occurred. In this case, it also identifies an inconsistent subset of the assigment. Unlike `SetTrue`, the method `TheoryProp` is not associated with any particular constraint. The presence of `TheoryProp` simply enables Solver$_T$ to know in which states the U-prop rule is not enabled, so that theory propagation can occur at a lower priority than unit propagation.

Secondly, we modify a standard DPLL procedure to communicate the event that a constraint is removed from the implication queue as a result of a backtrack. A standard DPLL empties the implication queue when a conflict is found, just prior to backtracking. We assume this behavior and add a call to a new method `Dequeue`(c) for each constraint c that is removed from the implication queue upon backtracking. The DPLL(X) procedure makes no further accomodations for our methodology and in fact, may be readily used for lazy theory propagation [3] or standard DPLL(T)-style theory propagation.

4.2 A Generic Solver$_{T^\circ}$

We present a generic implementation of Solver$_{T^\circ}$ which is provided with a T-decomposition T°_k. Solver$_{T^\circ}$ allocates an extended state constraint label $\langle l_c, u_c \rangle$ for each constraint $c \in \chi$. Initially, Solver$_{T^\circ}$ identifies $\rho^i_\chi(\emptyset)$ for

each $i \in [1..k]$. and constraints c are labelled as follows. All l_c are assigned 0. If c is a member of some $\rho^i_\chi(\emptyset)$, then u_c is assigned $j-1$ where j is the least j such that $c \in \rho^j_\chi(\emptyset)$. Otherwise, u_c is assigned k. All constraints with $u_c < k$ are added to the DPLL(X) implication queue.

After initialization all constraint relabelling occurs via the methods `SetTrue`, `Backtrack`, `Dequeue`, and `TheoryProp`[1]. The method `SetTrue` occurs as a constraint c becomes assigned. The procedure first checks whether or not $u_c < 1$. If $u_c < 1$ or $c \notin \mathcal{L}_1$, no consistency check is performed. Otherwise the T_1-constistency of $\mathbf{b}_0 \cap \mathcal{L}_i \cup \{c\}$ is tested. If $u_c < 1$ or the consistency check succeeds, l_c is assigned the value 1. The methods `Backtrack` and `Dequeue` relabel every constraint c which become unassigned or leave the implication queue as a result of backtracking. Each such constraint c is labelled $\langle 0, k\rangle$. The method `TheoryProp` performs all relabelling which establishes the basis and cobasis of each T_i, and is detailed in figure 2.

$R \leftarrow \emptyset$
while $\{c \in R \mid l_c = 0\} = \emptyset$ and $\exists l_c \in [1..k]\ .\ u_c \geq l_c$ **do**
  select some c with the minimal $l_c \in [1..k]$ such that $u_c \geq l_c$
  $i \leftarrow l_c$
  if $c \in \mathcal{L}_{i+1} \wedge \mathbf{b}_i \cap \mathcal{L}_{i+1} \models_{i+1} \neg c$ **then**
    return $\langle \bot, \texttt{Explain}(\neg c) \cup \{c\}\rangle$
  $l_c \leftarrow i+1$
  $R \leftarrow \rho^i_\chi(\mathbf{b}_i) \cap \{c \in \chi \mid l_c \in \{0, i\} \wedge u_c \geq l_c\}$
  for $c \in R$ **do**
    $u_c \leftarrow i-1$
return $\langle \top, \{c \in R \mid l_c = 0\}\rangle$

Figure 2: `TheoryProp` returns a pair $\langle ok, \Gamma\rangle$. If ok is true, then Γ is a set of T-consequences of the current assignment, otherwise Γ is a small inconsistent subset of the current assignment. `TheoryProp` relabels constraints c one at a time. Each such constraint satisfies $l_c \leq u_c$ and minimizes l_c, prior to the relabelling. Let $i = l_c$ before c is relabelled. If $c \in \mathcal{L}_{i+1}$ then T_{i+1}-consistency of $\mathbf{b}_i \cap \mathcal{L}_{i+1} \cup \{c\}$ is tested. If the test succeeds, then l_c is assigned $i+1$ and then the consequence finder ρ^i_χ is invoked. For every consequence c, u_c is assigned $i-1$. If new consequences c with $l_c = 0$ are found, they are unassigned and the procedure returns them to the DPLL(X) engine to add to its implication queue.

The implementation above exhibits some "instant" optimizations which are not part of a standard DPLL(T) system. First, no constraints c with $u_c < k$ are subject to processing in any T_i with $i > u_c$. These constraints do not trigger T_i-consistency checks, do not trigger T_i consequence finding, and do not form part of the T_i-basis $\mathbf{b}_i$. Thus in comparison to a standard DPLL(T) system, fewer consistency checks occur, fewer calls to theory propagation occur, and theory propagation is itself a function of a smaller set. This optimization holds for single theories as well and its correctness follows from the correctness of extended state properties, which we address in the next section. In practice, the dramatic effect of these optimizations for the case of the (undecomposed) theory of difference constraints was demonstrated empirically by the `Jat` solver [4].

5 Correctness of Extended State Properties

We now give proofs showing how the required extended state properties of section 3.3 are realized by the implementation.

Theorem 5.1 (Basis-cobasis assignment)**.** *In any state $A \circ F$, and for any $i \in [1..k]$ 1) the T_i basis is assigned ($\mathbf{b}_i \subseteq A$) and 2) the assignment is contained in the union of the T_i-basis and T_i-cobasis ($A \cap \chi \subseteq \bigcup_{i\in[0..k]}(\mathbf{b}_i \cup \bar{\mathbf{b}}_i)$).*

[1]The method `Explain` does not perform any relabelling. Nonetheless, it is worth mentioning that `Explain` is always called with a constraint c which is implied by some $\mathbf{b}_i \cap \mathcal{L}_i$. This set may be used as a set of reasons for c in case a particular implementation is unable to provide a smaller set of reasons.

Proof. 1) Initially, each $\mathbf{b}_i$ is empty, since each $l_c = 0$. Whenever a constraint c becomes a member of $\mathbf{b}_0$ (l_c is relabelled from 0 to some greater value), it is done in `SetTrue` where it is guaranteed to be assigned by the DPLL(X) engine. Only constraints in $\mathbf{b}_i$ are subsequently members of $\mathbf{b}_{i+1}$. Hence it will suffice to show that after any constraint becomes unassigned, it is not in any $\mathbf{b}_i$, *i.e* that each such constraint c has $l_c = 0$. All constraints which becomes unassigned are passed to the procedure `Backtrack`, where they are relabelled $\langle 0, k\rangle$.
2) It suffices to show that in any state, every member of

$$\{c \in \chi \mid l_c = 0 \wedge u_c \geq k\}$$

is an unassigned constraint, since all constraints c labelled otherwise are in some basis or co-basis and there are no other possible labellings. Initially, all constraints are unassigned, and this is trivially true. The `Backtrack` and `Dequeue` procedures are the only procedure which labels constraints c so that $l_c = 0$ or $u_c \geq k$, and they do so only for constraints which are unassigned. □ □

Lemma 5.2. *In every state $A \circ F$ which precedes a Guess or is final, for every $c \in \chi$ either $l_c = u_c + 1$ and c is assigned, or $l_c = 0$, $u_c = k$ and c is unassigned.*

Proof. The DPLL(X) procedure ensures that before a guess and in every final state, `TheoryProp` returns no consequences and no inconsistency. `TheoryProp`, in turn only produces such a result when there is no c such that $l_c \leq u_c$ and $l_c \in [1..k]$. (see line 2, figure 2). The only other possible labellings have $l_c = 0$, $l_c = k + 1$ or $l_c > u_c$. When $l_c = k + 1$, it must be that $l_c > u_c$, since nothing ever assigns $u_c > k$. Hence the only labellings have $l_c = 0$ or $l_c > u_c$. In the latter case, we are guaranteed that $l_c = u_c + 1$ since every time some l_c is relabelled to a non-zero value, it is incremented by one and $l_c \leq u_c$; and whenever the label u_c takes a value and $l_c > 0$, it is assigned by theorem 5.1 since $l_c > 0$.

We then consider the case that $l_c = 0$. Every time l_c is assigned the value 0, u_c is assigned k and c is unassigned. Hence we need only consider what happens when u_c is assigned some value other than k. In this context, c is placed in the DPLL(X) implication queue, since $l_c = 0$. However, the implication queue is empty prior to any guess and in any final state. So c must have left the implication queue in such a state. This may occur by a call to `SetTrue`, which assigns $l_c > 0$, and hence is not relevant to this case. The only other way a constraint leaves the implication queue is via `Dequeue`, which assigns $u_c = k$ and leaves c unassigned. □ □

Theorem 5.3 (Basis-cobasis closure)**.** *In any state, every $\rho^i_\chi(\mathbf{b}_i) \subseteq \mathbf{b}_i \cup \bar{\mathbf{b}}_i$*

Proof. After Solver$_{T^\circ}$ initialization, each $\mathbf{b}_i = \emptyset$ and $\rho^i_\chi(\emptyset) \subseteq \bar{\mathbf{b}}_i$. We now prove this by induction on the relabellings that occur in the methods `SetTrue`, `TheoryProp`, `Explain`, `Backtrack`, and `Dequeue`. Since `TheoryProp`, `Backtrack`, and `Dequeue` are the only methods which change the set of constraints labelled $\mathbf{b}_i$ or $\bar{\mathbf{b}}_i$ for $i \in [1..k]$, we can restrict our attention to them.

- `TheoryProp`
  In this procedure, every update $\mathbf{b}'_i = \mathbf{b}_i \cup \{c\}$ with $c \in \mathbf{b}_{i-1}$ is followed by the update $\bar{\mathbf{b}}'_i = \bar{\mathbf{b}}_i \cup \rho^i_\chi(\mathbf{b}'_i) \cap \{c \in \chi \mid l_c \in \{i, 0\} \wedge l_c \leq u_c\}$. We then must show that $\rho^i_\chi(\mathbf{b}'_i) \subseteq \mathbf{b}'_i \cup \bar{\mathbf{b}}'_i$ after any such pair of updates.

  Suppose for a contradiction this is not the case. Then there is some $d \in \rho^i_\chi(\mathbf{b}'_i)$ such that $d \notin \mathbf{b}'_i \cup \bar{\mathbf{b}}'_i$. Since $\mathbf{b}_i \subseteq \mathbf{b}'_i$ and $\bar{\mathbf{b}}_i \subseteq \bar{\mathbf{b}}'_i$, and $d \notin \mathbf{b}'_i \cup \bar{\mathbf{b}}'_i$, it follows that $d \notin \mathbf{b}_i \cup \bar{\mathbf{b}}_i$. Hence before the updates, $d \in \{c \in \chi \mid i \in [l_c..u_c]\}$, because there are no other possible labellings. We now consider two cases.

  Case 1. Suppose $l_d \in \{0, i\}$ before the first update. Then $l'_d \in \{0, i\}$ after the first update, since otherwise $l'_d = i + 1$ and $d \notin \mathbf{b}'_i$. Since by assumption $c \in \rho^i_\chi(\mathbf{b}'_i)$, and the second update assigns $\bar{\mathbf{b}}'_i = \bar{\mathbf{b}}_i \cup \rho^i_\chi(\mathbf{b}'_i) \cap \{c \in \chi \mid l_c \in \{0, i\}\}$, it must be the case that $d \in \bar{\mathbf{b}}'_i$ after the second update, a contradiction.

  Case 2. Suppose $l_d \in [1..i)$ before the update. The procedure `TheoryProp` disallows this possibility whenever $l_d \leq u_d$, since it chooses a constraint c which minimizes l_c, and $i = l_c$ before the update. Hence $l_d > u_d$ and $c \in \bar{\mathbf{b}}_i$. Since $\bar{\mathbf{b}}'_i \supseteq \bar{\mathbf{b}}_i$, it must be that $c \in \bar{\mathbf{b}}'_i$, contracting the assumption.

- `Backtrack`, `Dequeue`
  Both methods are called during every DPLL(X) backtrack. A backtrack is guaranteed to reach a state $b = A, l \circ F$ such that the state $g = A \circ F'$ was previously visited, g was followed by a Guess, and every $l \in A$ remained assigned in the path from g to b. By lemma 5.2, in every state prior to a guess, every c has $l_c = 0$ and is unassigned or $u_c < l_c$ and is assigned. No constraints c with $u_c < l_c$ are relabelled as long as they are assigned. Therefore, by the induction hypothesis, it will suffice to show that all constraints relabelled in the path from g to b are not in any $\mathbf{b}_i$ or $\bar{\mathbf{b}}_i$ with $i \in [1..k]$ in state b. The `Backtrack` procedure relabels all constraints which were assigned in the path from g to b with $\langle 0, k\rangle$. The only remaining constraints which are relabelled are those which were passed to DPLL(X) via `TheoryProp` but were not yet assigned in the state preceding the backtrack. The `Dequeue` procedure relabels all such constraints $\langle 0, k\rangle$.

□ □

Theorem 5.4 (Cobasis irrelevancy)**.** *In any state $A \circ F$ and for every $i, j \in [1..k]$ with $j \geq i$, $A \setminus \bar{\mathbf{b}}_i \models_j A$.*

Proof. Every time a constraint c becomes a member of some $\bar{\mathbf{b}}_i$, $c \in \rho_\chi^h(\mathbf{b}_h)$ for some $h \leq i$. Since each ρ_χ^h is sound, we have that $\mathbf{b}_h \models_h \bar{\mathbf{b}}_i$. By theorem 5.1, $\mathbf{b}_h \subseteq A$ and hence $A \models_h \bar{\mathbf{b}}_i$. By consequence lifting (corollary 3.3), then $A \models_j \bar{\mathbf{b}}_i$ for any $j \geq h$. Since $h \leq i$, we have $A \models_j \bar{\mathbf{b}}_i$ for any $j \geq i$ as well. □ □

Theorem 5.5 (Basis consistency)**.** *In any state every set $\mathbf{b}_i \cap \mathcal{L}_i$ is T_i-consistent.*

Proof. We assume each T_i is consistent, and prove a stronger proposition: each $\mathbf{b}_i \cap \mathcal{L}_{i+1}$ is T_{i+1}-consistent. We proceed by induction on constraint relabelling. Initially, every $\mathbf{b}_i$ is empty and the result is trivial. We observe that before any $\mathbf{b}_i$ is extended with a constraint c, $c \in \mathbf{b}_j$ for every $j < i$, since l_c is always incremented by one. By the induction hypothesis, each such $\mathbf{b}_j \cap \mathcal{L}_j$ is T_{j+1}-consistent and so we need only show that $\mathbf{b}_i \cup \{c\} \cap \mathcal{L}_j$ is T_{i+1}-consistent. However, every time a constraint $c \in \mathcal{L}_j$ becomes a member of $\mathbf{b}_i$, a T_{i+1}-consistency check succeeds on the set $\mathbf{b}_i \cup \{c\} \cap \mathcal{L}_i$. Hence there is a T_{i+1} model of $\mathbf{b}_i \cup \{c\} \cap \mathcal{L}_j$. By lemma 3.2, any such model is a T_i-model and so $\mathbf{b}_i \cup \{c\} \cap \mathcal{L}_j$ is T_i-consistent. □ □

Theorem 5.6 (Full consistency)**.** *In any state $A \circ F$ which precedes a guess or is final, every $A \cap \mathcal{L}_i$ is T_i-consistent.*

Proof. In any state which precedes a guess or is final, every assigned constraint c satisfies $c \in \mathbf{b}_i \cup \bar{\mathbf{b}}_i$ for every $i \in [1..k]$, since every assigned constraint has $l_c = u_c + 1$ (lemma 5.2). It follows that $A \cap \chi \subset \mathbf{b}_i \cup \bar{\mathbf{b}}_i$ for every $i \in [1..k]$. This together with theorem 5.1 and lemma 5.2 gives us $A \cap \chi = \mathbf{b}_i \cup \bar{\mathbf{b}}_i$, and it follows that $A \cap \mathcal{L}_i = (\mathbf{b}_i \cup \bar{\mathbf{b}}_i) \cap \mathcal{L}_i$. By the soundness of each consequence finder, $\mathbf{b}_i \cap \mathcal{L}_i \models_i (\mathbf{b}_i \cup \bar{\mathbf{b}}_i) \cap \mathcal{L}_i$ and so $\mathbf{b}_i \cap \mathcal{L}_i \models_i A \cap \mathcal{L}_i$. Then we have that $A \cap \mathcal{L}_i \models_i \bot \implies \mathbf{b}_i \cap \mathcal{L}_i \models_i \bot$ by transitivity of $\models_i$. Contraposing this statement gives $\mathbf{b}_i \cap \mathcal{L}_i \not\models_i \bot \implies A \cap \mathcal{L}_i \not\models_i \bot$. By theorem 5.5, each $\mathbf{b}_i \cap \mathcal{L}_i$ is T_i-consistent, hence $A \cap \mathcal{L}_i$ is T_i-consistent. □ □

Corollary 5.7. *Full consistency occurs even if consistency checks only occur with respect to T_k.*

Theorem 5.8 (Full closure)**.** *In any state which precedes a Guess or is final every $\rho_\chi^i(A) \subseteq A$.*

Proof. By theorem 5.3, the implementation ensures that at every state and for every $i \in [1..k]$, $\rho_\chi^i(\mathbf{b}_i) \subseteq \mathbf{b}_i \cup \bar{\mathbf{b}}_i$. By lemma 5.2, every constraint c with $u_c < k$ is assigned, hence every $\bar{\mathbf{b}}_i \subseteq A$. It follows immediately that $\rho_\chi^i(\mathbf{b}_i) \subseteq \mathbf{b}_i \cup A$. By theorem 5.1, in addition each $\mathbf{b}_i \subseteq A$, and it follows immediately that each $\rho_\chi^i(\mathbf{b}_i) \subseteq A$.

By chain-closure and monotonicity of each ρ_χ^i, it follows that $\rho_\chi^i(\mathbf{b}_i) = \rho_\chi^i(\mathbf{b}_i \cup \bar{\mathbf{b}}_i)$. Consequently, $\rho_\chi^i(\mathbf{b}_i \cup \bar{\mathbf{b}}_i) \subseteq A$. By lemma 5.2, every assigned constraint c has $l_c = u_c + 1$. Consequently, for every $i, j \in [1..k]$, $\mathbf{b}_i \cup \bar{\mathbf{b}}_i = \mathbf{b}_j \cup \bar{\mathbf{b}}_j = A \cap \chi$. Hence it follows that $\rho_\chi^i(A \cap \chi) \subseteq A$. Since each $l \in A \setminus \chi$ has no interpretation in any T_i, $\rho_\chi^i(\mathbf{b}_i \cup (A \setminus \chi)) = \rho_\chi^i(\mathbf{b}_i)$. We conclude each $\rho_\chi^i(A) \subseteq A$. □ □

Corollary 5.9. *If ρ_χ^k is complete then the procedure never guesses an inconsistent constraint.*

6 Conclusions

We presented a method for extending the DPLL(T) procedure to richer theories by means of a prioritized constraint filter. Classical approaches to theory *combination* such as Nelson-Oppen's method [14] may also be used to enrich theories. However, these approaches almost invariably require the signatures of the theories to be disjoint. This restriction inherently precludes the exploitation of simple sub-theories and practically excludes disjoint theories intended to operate over the same domain (such as the additive and multiplicative groups in arithmetic). By contrast, while our methodology doesn't address signature-disjoint theories, it does exploit the simplicity of sub-theories, and allows multiple theories which share the same domain. We hence believe that the DPLL(T) methodology presented in this paper complements theory combination methods.

Nonetheless, several questions remain open with regard DPLL(T) theory chains in general. Perhaps foremost amongst these are empirical questions. We have only begun to test theory chains in DPLL(T). Thus far, experiments suggest that prioritization and filtering are effective mechanisms which reduce calls to the most expensive procedures and also reduces their complexity in general. However, our experiments are thus far limited to trivial decompositions and so our evidence is far from conclusive. Other questions of interest are whether or not formula transforms can be used to boost the effictiveness of theory chains. For example, the Nelson-Oppen procedure performs a *purification* transform on formulas which is satisfiability preserving and may produce a formula in which every constraint has only one relation or function symbol. Perhaps this procedure would also artificially enlarge the set of constraints belonging to less complex theories in a chain, thus making the filter more effective. Finally, it would be interesting to address the question of composing Nelson-Oppen with theory chains to arrive at theory trees. We would like to study these issues in future work.

References

[1] A. Armando, C. Castellini, E. Giunchiglia, M. Idini, and M. Maratea. Tsat++: an open platform for satisfiability modulo theories. In *PDPAR'04*, 2004. 1

[2] R. J. Jr. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *AAAI'97*, pages 203–208, 1997. 2.1

[3] S. Cotton. Satisfiability checking with difference constraints. Master's thesis, Max Planck Institute, 2005. 1, 2.3, 4.1

[4] S. Cotton and O. Maler. Fast and flexible difference logic propagation for DPLL(T). In *Submitted for publication.*, 2006. 4.2

[5] S. Cotton and O. Maler. Satisfiability modulo theory chains with DPLL(T). In *Verimag Technical Report TR-2006-04*, 2006.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. In *Communications of the ACM*, volume 5(7), pages pages 394–397, 1962. 2.1

[7] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(1):201–215, 1960. 2.1

[8] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *CAV'04*, pages 175–188, 2004. 1, 2.1

[9] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver. In *DATE'02*, pages 142–149, 2002. 2.1

[10] J. P. Marquez-Silva and K. A. Sakallah. Grasp – a new search algorithm for satisfiability. In *CAV'96*, pages 220–227, November 1996. 2.1

[11] M.Bozzano, R.Bruttomesso, A.Cimatti, T.Junttila, P.v.Rossum, S.Schulz, and R.Sebastiani. Mathsat: Tight integration of SAT and mathematical decision procedures. *Journal of Automated Reasoning*, Special Issue on SAT, 2005. 1, 2.3

[12] M.Bozzano, R.Bruttomesso, A.Cimatti, T.Junttila, S.Ranise, P.v.Rossum, and R.Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *CAV'05*, volume LNCS 3576, pages pages 335–349, 2005. 1

[13] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC'01*, 2001. 2.1

[14] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. In *ACM Trans. on Programming Languages and Systems*, volume 1(2), pages 245–257, Oct. 1979. 6

[15] P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, O. Maler, and N. Jain. Verification of timed automata via satisfiability checking. In *LNCS*, volume Volume 2469, pages 225 – 243, Jan 2002. 2.3

[16] R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV'05*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer, 2005. 1, 2.3

[17] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *LPAR'04*, volume 3452 of *LNCS*, pages 36–50, 2005. 1, 2.2, 1

[18] R. Sebastiani. On efficiently integrating boolean and theory-specific solving procedures. 1

[19] J. P. M. Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proc. of the 9th Portuguese Conference on Artificial Intelligence*, pages 62–74, 1999. 2.1

[20] C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Proc.of the 8th European Conf. on Logics in Artificial Intelligence*, volume 2424 of *LNCS*, 2002. 1, 2.1

[21] O. Tveretina. DPLL-based procedure for equality logic with uninterpreted functions. 1