From KSAT to Delayed Theory Combination: Exploiting DPLL Outside the SAT Domain*

Roberto Sebastiani

DIT, Università di Trento, via Sommarive 14, I-38050 Povo, Trento, Italy.
roberto.sebastiani@dit.unitn.it

Abstract. In the last two decades we have witnessed an impressive advance in the efficiency of propositional satisfiability techniques (SAT), which has brought large and previously-intractable problems at the reach of state-of-the-art SAT solvers. Most of this success is motivated by the impressive level of efficiency reached by current implementations of the DPLL procedure. Plain propositional logic, however, is not the only application domain for DPLL. In fact, DPLL has also been successfully used as a boolean-reasoning kernel for automated reasoning tools in much more expressive logics.
In this talk I overview a 12-year experience on integrating DPLL with logic-specific decision procedures in various domains. In particular, I present and discuss three main achievements which have been obtained in this context: the DPLL-based procedures for modal and description logics, the lazy approach to Satisfiability Modulo Theories, and Delayed Theory Combination.

1 Introduction

In the last two decades we have witnessed an impressive advance in the efficiency of propositional satisfiability techniques (SAT), which has brought large and previously-intractable problems at the reach of state-of-the-art SAT solvers. As a consequence, many hard real-world problems have been successfully solved by encoding into SAT. E.g., SAT solvers are now a fundamental tool in most formal verification design flows for hardware systems.

Most of the success of SAT technologies is motivated by the impressive level of efficiency reached by current implementations of the Davis-Putnam-Logemann-Loveland procedure (DPLL) [13, 12], in its most-modern variants (see, e.g., [43]).

Plain propositional logic, however, is not the only application domain for DPLL. In fact, DPLL has also been successfully used as a boolean-reasoning kernel for automated reasoning tools in much more expressive logics, including modal and description logics, and decidable subclasses of first-order logic. In most cases, this has produced a boost in the overall performances, which rely

* This work has been partly supported by ORCHID, a project sponsored by Provincia Autonoma di Trento, by the EU project S3MS "Security of Software and Services for Mobile System" contract n. 27004, and by a grant from Intel Corporation.

both on the improvements in DPLL technology and on clever integration between DPLL and the logic-specific decision procedures.

In this talk I overview a 12-year experience on integrating DPLL with logic-specific decision procedures in various domains. In particular, I present and discuss three main achievements which have been obtained in this context.

The first (§2) is the introduction of DPLL inside *satisfiability procedures for modal and description logics* [23, 24, 37, 27, 35, 28, 25, 21, 22, 26], which caused a boost in performances wrt. previous state-of-the-art procedures, which used Smullyan's analytic tableaux [39] as propositional-reasoning engine.

The second (§3) is the *lazy approach to Satisfiability Modulo Theories* (lazy SMT) [1, 41, 15, 3, 4, 18, 19, 6, 16], in which DPLL is combined with satisfiability procedures for (sets of literals in) expressive decidable first-order theories. Current lazy SMT tools have reached a high degree of efficiency, so that they are increasingly used in formal verification.

The third (§4) is *Delayed Theory Combination (*DTC*)* [7–9, 17, 14], a general method for tackling the problem of theory combination within the context of lazy *SMT*. DTC exploits the power of DPLL also for assigning truth values for the interface equalities that the $\mathcal{T}$*-solver*'s are not capable of inferring. Thus, it does not rely on (possibly very expensive) deduction capabilities of the component procedures —although it can fully benefit from them— and nicely encompasses the case of non-convex theories.

2 DPLL for Modal Logics

We assume the reader is familiar with the basic notions on modal logics and of first-order logic. Some very-basic background on SAT (see, e.g., [43]) and on decision procedures and their combination (see, e.g., [31]) is also assumed.

We adopt the following terminology and notation. We call an *atom* any formula which cannot be decomposed propositionally (e.g., A_1, $\Box_r(A_1 \wedge \Box_r A_2)$), and a *literal* an atom or its negation. We call a *truth assignment* μ for a formula φ any set/conjunction of top-level literals in φ. Positive literals $\Box_r\alpha_i$, A_k [resp. negative literals $\neg\Box_r\beta_i$, $\neg A_k$] mean that the corresponding atom is assigned to true [resp. false]. We say that a truth assignment μ for φ *propositionally satisfies* φ, written $\mu \models_p \varphi$, iff it tautologically entails φ. E.g., $\{A_1, \neg\Box_r(A_2 \wedge \Box_r A_3)\} \models_p (A_1 \wedge (A_2 \vee \neg\Box_r(A_2 \wedge \Box_r A_3)))$.

2.1 From Tableau-based to DPLL-based Procedures

We call "tableau-based" a system that implements and extends to other logics the Smullyan's propositional tableau calculus [39]. E.g., a typical Tableau-based procedure for modal K_m consists on some control strategy applied to the following rules:

$$\frac{\Gamma, \varphi_1 \wedge \varphi_2}{\Gamma, \varphi_1, \varphi_2}\ (\wedge) \qquad \frac{\Gamma, \varphi_1 \vee \varphi_2}{\Gamma, \varphi_1 \qquad \Gamma, \varphi_2}\ (\vee) \qquad \frac{\mu}{\alpha_1 \wedge \ldots \wedge \alpha_m \wedge \neg\beta_j}\ (\Box_r/\neg\Box_r) \qquad (1)$$

for each box-index $r \in \{1, ..., m\}$. Γ is an arbitrary set of formulas, and μ is a set of literals which includes $\neg\Box_r\beta_j$ and whose only positive $\Box_r$-atoms are $\Box_r\alpha_1, \ldots, \Box_r\alpha_m$.

We call "DPLL-based" any system that implements and extends to other logics the Davis-Putnam-Longeman-Loveland procedure (DPLL) [13, 12]. DPLL-based procedures basically consist on the combination of a DPLL procedure handling the purely-propositional component of reasoning, and some procedure handling the purely-modal component. Thus, for instance, in our terminology KSAT [23], FACT [27], DLP [35], RACER [26] are DPLL-based systems. [1] From a purely-logical viewpoint, it is possible to conceive a DPLL-based framework by substituting the propositional tableaux rules with some rules implementing the DPLL algorithms in a tableau-based framework [37]. A formal framework for representing DPLL and DPLL-based procedures has been proposed in [40, 33].

2.2 Basic Modal DPLL for K_m

The first DPLL-based procedure for a modal logic, KSAT, was introduced in [23, 25] (Figure 1). This schema evolved from that of the PTAUT procedure in [2], and is based on the "classic" DPLL procedure [13, 12]. KSAT takes in input a modal formula φ and returns a truth value asserting whether φ is K_m-satisfiable or not. KSAT invokes K-DPLL passing as arguments φ and (by reference) an empty assignment $\top$. K-DPLL tries to build a K_m-satisfiable assignment μ propositionally satisfying φ. This is done recursively, according to the following steps:

– (base) If $\varphi = \top$, then μ propositionally satisfies φ. Thus, if μ is K_m-satisfiable, then φ is K_m-satisfiable. Therefore K-DPLL invokes K-SOLVER(μ), which returns a truth value asserting whether μ is K_m-satisfiable or not.
– (backtrack) If $\varphi = \bot$, then μ does not satisfy φ, so that K-DPLL returns $False$.
– (unit) If a literal l occurs in φ as a unit clause, then l must be assigned $\top$. To obtain this, K-DPLL is invoked recursively with arguments the formula returned by *assign(l, φ)* and the assignment obtained by adding l to μ.
– (split) If none of the above situations occurs, then *choose-literal(φ)* returns an unassigned literal l according to some heuristic criterion. Then K-DPLL is first invoked recursively with arguments *assign(l, φ)* and $\mu \wedge l$. If the result is negative, then K-DPLL is invoked with *assign($\neg l, \varphi$)* and $\mu \wedge \neg l$.

K-DPLL is a variant of the "classic" DPLL algorithm [13, 12]. The K-DPLL schema differs from that of classic DPLL by only two steps.

[1] Notice that there is not an universal agreement on the terminology "tableau-based" and "DPLL-based". E.g., tools like FACT, DLP, and RACER are often called "tableau-based", although they use a DPLL-like algorithm instead of propositional tableaux for handling the propositional component of reasoning [27, 35, 28, 26].

```
function Ksat($\varphi$)
    return K-DPLL($\varphi, \top$);

function K-DPLL($\varphi, \mu$)
    if ($\varphi == \top$)                                    /* base      */
        then return K-Solver($\mu$);
    if ($\varphi == \bot$)                                    /* backtrack */
        then return False;
    if {a unit clause ($l$) occurs in $\varphi$}              /* unit      */
        then return K-DPLL($assign(l, \varphi), \mu \wedge l$);
    $l$ := choose-literal($\varphi$);                          /* split     */
    return  K-DPLL($assign(l, \varphi), \mu \wedge l$)   or
            K-DPLL($assign(\neg l, \varphi), \mu \wedge \neg l$);

/* $\mu$ is $\bigwedge_i \Box_1 \alpha_{1i} \wedge \bigwedge_j \neg\Box_1 \beta_{1j} \wedge \ldots \wedge \bigwedge_i \Box_m \alpha_{mi} \wedge \bigwedge_j \neg\Box_m \beta_{mj} \wedge \bigwedge_k A_k \wedge \bigwedge_h \neg A_h$ */
function K-Solver($\mu$)
    for each box index $r \in \{1...m\}$ do
        for each literal $\neg\Box_r \beta_{rj} \in \mu$ do
            if not (Ksat($\bigwedge_i \alpha_{ri} \wedge \neg\beta_{rj}$))
                then return False;
    return True;
```

Fig. 1. The basic version of Ksat algorithm. *assign*(l, φ) substitutes every occurrence of l in φ with $\top$ and evaluates the result.

The first is the "base" case: when standard DPLL finds an assignment μ which propositionally satisfies the input formula, it simply returns "$True$". K-DPLL, instead, is also supposed to check the K_m-satisfiability of the corresponding set of literals, by invoking K-Solver on μ. If the latter returns $true$, then the whole formula is satisfiable and K-DPLL returns $True$ as well; otherwise, K-DPLL backtracks and looks for the next assignment.

The second is in the fact that in K-DPLL the pure-literal step [12] is removed. [2] In fact the sets of assignments generated by DPLL with pure-literal might be incomplete and might cause incorrect results. This fact is shown by the following example.

Example 1. Let φ be the following formula:

$$(\Box_1 A_1 \vee A_1) \wedge (\Box_1(A_1 \to A_2) \vee A_2) \wedge (\neg\Box_1 A_2 \vee A_2) \wedge (\neg A_2 \vee A_3) \wedge (\neg A_2 \vee \neg A_3).$$

φ is K_m-satisfiable, because $\mu = \{A_1, \neg A_2, \Box_1(A_1 \to A_2), \neg\Box_1 A_2\}$ is a K_m-consistent assignment propositionally satisfying φ. It is easy to see that no satisfiable assignment propositionally satisfying φ assigns $\Box_1 A_1$ to true. As $\Box_1 A_1$ occurs only positively in φ, DPLL with the pure literal rule would assign $\Box_1 A_1$ to true as first step, which would lead the procedure to return $False$.

[2] Alternatively, the application of the pure-literal rule is restricted to atomic propositions only.

With these simple modifications, the embedded DPLL procedures works as an enumerator of a complete set of assignments, whose K_m-satisfiability is recursively checked by K-Solver. K-Solver is a straightforward application of the $(\Box_r/\neg\Box_r)$-rule in (1).

The above schema has lately been extended to other modal and description logics [27, 28, 22]. Moreover, the schema has been lately adapted to work with modern DPLL procedures, and many optimizations have been conceived. Some of them will be described in §3.3 in the context of Satisfiability Modulo Theories.

2.3 DPLL-based vs. Tableaux-based procedures

[23–25, 20, 21, 28, 29] presented extensive empirical comparisons, in which DPLL-based procedures outperformed tableau-based ones, with orders-of-magnitude performance gaps. (Similar performance gaps between tableau-based vs. DPLL-based procedures were obtained lately also in a completely-different context [1].) Remarkably, most such results were obtained with tools implementing variants of the "classic" DPLL procedure of §2.2, still very far from the efficiency of current DPLL implementations.

Both tableau-based and DPLL-based procedures for K_m-satisfiability work (i) by enumerating truth assignments which propositionally satisfy the input formula φ and (ii) by recursively checking the K_m-satisfiability of the assignments found. As both algorithms perform the latter step in the same way, the key difference relies in the way they handle propositional inference. In [24, 25] we remarked that, regardless the quality of implementation and the optimizations performed, DPLL-based procedures do not suffer from two intrinsic weaknesses of tableau-based procedures which significantly affect their efficiency, and whose effects are amplified up to exponentially when using them in modal inference. We consider these weaknesses in turn.

Syntactic vs. semantic branching. In a propositional tableaux truth assignments are generated as branches induced by the application of the $\vee$-rule to disjunctive subformulas of the input formula φ. Thus, they perform *syntactic branching* [24], that is, the branching in the search tree is induced by the syntactic structure of φ. As discussed in [11], an application of the $\vee$-rule generates two subtrees which can be mutually consistent, i.e., which may share propositional models. [3] Therefore, the set of truth assignments enumerated by a propositional tableau grows exponentially with the number of disjunctions occurring positively in φ, regardless the fact that it may contain up to exponentially-many duplicated and/or subsumed assignments.

Things get even worse in the modal case. When testing K_m-satisfiability, unlike the propositional case where they look for *one* assignment satisfying the

[3] As pointed out in [11], propositional tableaux rules are unable to represent *bivalence*: "every proposition is either true or false, *tertium non datur*". This is a consequence of the elimination of the cut rule in cut-free sequent calculi, from which propositional tableaux are derived.

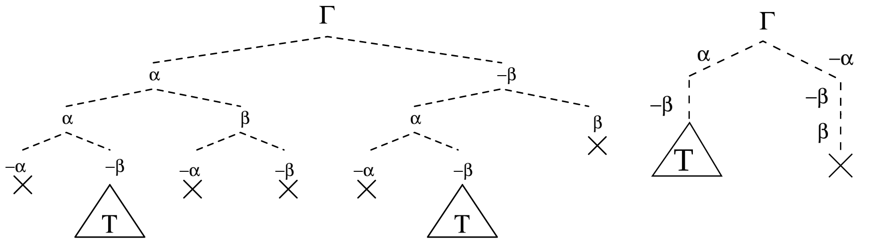

Fig. 2. Search trees for the formula $\Gamma = (\alpha \vee \neg\beta) \wedge (\alpha \vee \beta) \wedge (\neg\alpha \vee \neg\beta)$. Left: a tableau-based procedure. Right: a DPLL-based procedure.

input formula, the propositional tableaux are used to enumerate up to *all* satisfying assignments, which must be recursively checked for K_m-consistency. This requires checking recursively possibly-many sub-formulas of the form $\bigwedge_i \alpha_{ri} \wedge \neg\beta_j$ of depth $d-1$, for which a propositional tableau will enumerate all satisfying assignments, and so on. At every level of nesting, a redundant truth assignment introduces a redundant modal search tree. Thus, with modal formulas, the redundancy of the propositional case propagates up-to-exponentially with the modal depth.

DPLL instead, performs a search which is based on *semantic branching* [24], i.e., a branching on the *truth value* of sub-formulas ψ of φ (typically atoms): [4]

$$\frac{\varphi}{\varphi[\psi/\top] \qquad \varphi[\psi/\bot]},$$

where $\varphi[\psi/\top]$ is the result of substituting with $\top$ all occurrences of ψ in φ and then simplify the result. Thus, every branching step generates two *mutually-inconsistent* subtrees. Thus, DPLL always generates non-redundant sets of assignments. This avoids search duplications and, in the case of modal search, the recursive exponential propagation of redundancy.

Example 2. Consider the formula $\Gamma = (\alpha \vee \neg\beta) \wedge (\alpha \vee \beta) \wedge (\neg\alpha \vee \neg\beta)$, where α and β are modal atoms s.t. $\alpha \wedge \neg\beta$ is K_m-inconsistent, and let d be the depth of Γ. The only assignment propositionally satisfying Γ is $\mu = \alpha \wedge \neg\beta$. Consider Figure 2, left. Two distinct but identical open branches are generated, both representing the assignment μ. Then the tableau expands the two open branches in the same way, until it generates two identical (and possibly-big) closed modal sub-trees T of modal depth d, each proving the K_m-unsatisfiability of μ.

This phenomenon may repeat itself at the lower level in each sub-tree T, and so on. For instance, if $\alpha = \Box_1((\alpha' \vee \neg\beta') \wedge (\alpha' \vee \beta'))$ and $\beta = \Box_1(\alpha' \wedge \beta')$, then at the lower level we have a formula Γ' of depth $d-1$ analogous to Γ. This propagates exponentially the redundancy with the depth d.

[4] Notice that the notion of "semantic branching" introduced in [24] is stronger than that lately used in [27, 28]; the former coarsely corresponds to the latter plus the usage of unit-propagation.

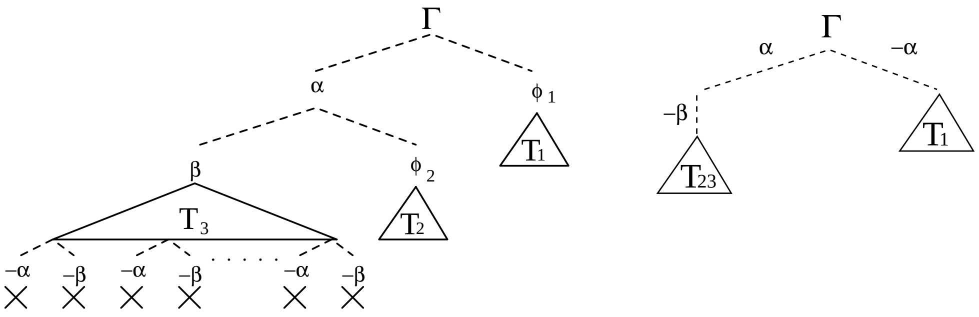

Fig. 3. Search trees for the formula $\Gamma = (\alpha \vee \phi_1) \wedge (\beta \vee \phi_2) \wedge \phi_3 \wedge (\neg\alpha \vee \neg\beta)$. Left: a tableau-based procedure. Right: a DPLL-based procedure.

Finally, notice that if we considered the formula $\Gamma^K = \bigwedge_{i=1}^{K} (\alpha_i \vee \neg\beta_i) \wedge (\alpha_i \vee \beta_i) \wedge (\neg\alpha_i \vee \neg\beta_i)$, the tableau would generate 2^K identical truth assignments $\mu^K = \bigwedge_i \alpha_i \wedge \neg\beta_i$, and things would get exponentially worse.

Look at Figure 2, right. A DPLL-based procedure branches asserting $\alpha = \top$ or $\alpha = \bot$. The first branch generates $\alpha \wedge \neg\beta$, whilst the second gives $\neg\alpha \wedge \neg\beta \wedge \beta$, which immediately closes. Therefore, only one instance of $\mu = \alpha \wedge \neg\beta$ is generated. The same applies to μ^K.

Detecting constraint violations. A propositional formula φ can be seen as a set of constraints for the truth assignments which possibly satisfy it. For instance, a clause $A_1 \vee A_2$ constrains every assignment not to set both A_1 and A_2 to $\bot$. Unlike tableaux, DPLL prunes a branch as soon as it violates some constraint of the input formula. (For instance, in KSAT this is done by the function *assign*.)

Example 3. Consider the formula $\Gamma = (\alpha \vee \phi_1) \wedge (\beta \vee \phi_2) \wedge \phi_3 \wedge (\neg\alpha \vee \neg\beta)$, α and β being atoms, ϕ_1, ϕ_2 and ϕ_3 being sub-formulas, such that $\alpha \wedge \beta \wedge \phi_3$ is propositionally satisfiable and $\alpha \wedge \phi_2$ is K_m-unsatisfiable. Look at Figure 3, left. Again, assume that, in a tableau-based procedure, the $\vee$-rule is applied in order, left to right. After two steps, the branch α, β is generated, which violates the constraint imposed by the last clause $(\neg\alpha \vee \neg\beta)$. A tableau-based procedure is not able to detect such a violation until it explicitly branches on that clause, that is, only after having generated the whole sub-tableau T_3 for $\alpha \wedge \beta \wedge \phi_3$, which may be rather big. DPLL instead (Figure 3, right) avoids generating the violating assignment detects the violation and immediately prunes the branch.

3 Integrating DPLL and Theory Solvers: Lazy SMT

Satisfiability Modulo Theories is the problem of deciding the satisfiability of a first-order formula with respect to some decidable first-order theory $\mathcal{T}$ ($SMT(\mathcal{T})$). Examples of theories of interest are, those of Equality and Uninterpreted Functions ($\mathcal{EUF}$), Linear Arithmetic ($\mathcal{LA}$), both over the reals ($\mathcal{LA}(\mathbb{Q})$) and the integers ($\mathcal{LA}(\mathbb{Z})$), its subclasses of Difference Logic ($\mathcal{DL}$) and Unit-Two-Variable-

Per-Inequality ($\mathcal{UTVPI}$), the theories of bit-vectors ($\mathcal{BV}$), of arrays ($\mathcal{AR}$) and of lists ($\mathcal{LI}$).

Efficient *SMT* solvers have been developed in the last five years, called *lazy SMT solvers*, which combine DPLL with decision procedures (*$\mathcal{T}$-solvers*) for many theories of interest (e.g., [1, 41, 15, 3, 4, 18, 19, 6, 16]).

3.1 Theory Solvers

In its simplest form, a *Theory Solver* for $\mathcal{T}$(*$\mathcal{T}$-solver*) is a procedure which takes as input a collection of $\mathcal{T}$-literals μ and decides whether μ is $\mathcal{T}$-satisfiable. In order to be effectively used within a lazy *SMT* solver, the following features of *$\mathcal{T}$-solver* are often important or even essential.

Model generation: when *$\mathcal{T}$-solver* is invoked on a $\mathcal{T}$-consistent set μ, it is able to produce a $\mathcal{T}$-model $\mathcal{I}$ witnessing the consistency of μ, i.e., $\mathcal{I} \models_{\mathcal{T}} \mu$.

Conflict set generation: when *$\mathcal{T}$-solver* is invoked on a $\mathcal{T}$-inconsistent set μ, it is able to produce the (possibly minimal) subset η of μ which has caused its inconsistency. η is called a *theory conflict set* of μ.

Incrementality: *$\mathcal{T}$-solver* "remembers" its computation status from one call to the other, so that, whenever it is given in input a set $\mu_1 \cup \mu_2$ such that μ_1 has just been proved $\mathcal{T}$-satisfiable, it avoids restarting the computation from scratch.

Backtrackability: it is possible for the *$\mathcal{T}$-solver* to undo steps and return to a previous status on the stack in an efficient manner.

Deduction of unassigned literals: when *$\mathcal{T}$-solver* is invoked on a $\mathcal{T}$-consistent set μ, it can also perform a set of deductions in the form $\eta \models_{\mathcal{T}} l$, s.t. $\eta \subseteq \mu$ and l is a literal on a not-yet-assigned atom in φ.

Deduction of interface equalities: when returning Sat, *$\mathcal{T}$-solver* can also perform a set of deductions in the form $\mu \models_{\mathcal{T}} e$ (if $\mathcal{T}$ is convex) or $\mu \models_{\mathcal{T}} \bigvee_j e_j$ (if $\mathcal{T}$ is not convex) s.t. $e, e_1, ..., e_n$ are equalities between variables or terms occurring in atoms in μ. We denote the equality $(v_i = v_j)$ by e_{ij}, and we call *e_{ij}-deduction* a deduction of (disjunctions of) e_{ij}'s. A *$\mathcal{T}$-solver* is *e_{ij}-deduction-complete* if it always capable to inferring the (disjunctions of) e_{ij}'s which are entailed by the input set of literals. Notice that here the deduced equalities need not occur in the input formula φ.

3.2 Lazy Satisfiability Modulo Theories

We adopt the following terminology and notation. The bijective function $\mathcal{T}2\mathcal{B}$ ("theory-to-propositional"), called *boolean abstraction*, maps propositional variables into themselves, ground $\mathcal{T}$-atoms into fresh propositional variables, and is homomorphic w.r.t. boolean operators and set inclusion. The function $\mathcal{B}2\mathcal{T}$ ("propositional-to-theory"), called *refinement*, is the inverse of $\mathcal{T}2\mathcal{B}$. The symbols φ, ψ denote $\mathcal{T}$-formulas, and μ, η denote sets of $\mathcal{T}$-literals; φ^p, ψ^p denote propositional formulas, μ^p, η^p denote sets of propositional literals (i.e., truth

```
SatValue $\mathcal{T}$-DPLL ($\mathcal{T}$-formula $\varphi$, $\mathcal{T}$-assignment & $\mu$) {
   if ($\mathcal{T}$-preprocess($\varphi, \mu$) == Conflict);
      return Unsat;
   $\varphi^p$ = $\mathcal{T}2\mathcal{P}$($\varphi$); $\mu^p$ = $\mathcal{T}2\mathcal{P}$($\mu$);
   while (1) {
      $\mathcal{T}$-decide_next_branch($\varphi^p, \mu^p$);
      while (1) {
         status = $\mathcal{T}$-deduce($\varphi^p, \mu^p$);
         if (status == Sat) {
            $\mu$ = $\mathcal{P}2\mathcal{T}$($\mu^p$);
            return Sat; }
         else if (status == Conflict) {
            blevel = $\mathcal{T}$-analyze_conflict($\varphi^p, \mu^p$);
            if (blevel == 0)
               return Unsat;
            else $\mathcal{T}$-backtrack(blevel,$\varphi^p, \mu^p$);
         }
         else break;
} } }
```

Fig. 4. Schema of $\mathcal{T}$-DPLL based on modern DPLL.

assignments) and we often use them as synonyms for the boolean abstraction of φ, ψ, μ, and η respectively, and vice versa (e.g., φ^p denotes $\mathcal{T}2\mathcal{B}(\varphi)$, μ denotes $\mathcal{B}2\mathcal{T}(\mu^p)$). If $\mathcal{T}2\mathcal{B}(\varphi) \models \bot$, then we say that φ is *propositionally unsatisfiable*, written $\varphi \models_p \bot$.

Figure 4 represent the schema of a $\mathcal{T}$-DPLL procedure based on a modern DPLL engine. This schema evolved from that of the DPLL-based procedures for modal logics, see §2.2. The input φ and μ are a $\mathcal{T}$-formula and a reference to an (initially empty) set of $\mathcal{T}$-literals respectively. The DPLL solver embedded in $\mathcal{T}$-DPLL reasons on and updates φ^p and μ^p, and $\mathcal{T}$-DPLL maintains some data structure encoding the set *Lits*(φ) and the bijective mapping $\mathcal{T}2\mathcal{P}/\mathcal{P}2\mathcal{T}$ on literals.

$\mathcal{T}$-`preprocess` simplifies φ into a simpler formula, and updates μ if it is the case, so that to preserve the $\mathcal{T}$-satisfiability of $\varphi \wedge \mu$. If this process produces some conflict, then $\mathcal{T}$-DPLL returns Unsat. $\mathcal{T}$-`preprocess` combines most or all the boolean preprocessing steps for DPLL with some theory-dependent rewriting steps on the $\mathcal{T}$-literals of φ. (The latter are described in §3.3.)

$\mathcal{T}$-`decide_next_branch` selects the next literal like in standard DPLL (but it may consider also the semantics in $\mathcal{T}$ of the literals to select).

$\mathcal{T}$-`deduce`, in its simplest version, behaves similarly to standard BCP in DPLL: it iteratively deduces boolean literals l^p deriving propositionally from the current assignment (i.e., s.t. $\varphi^p \wedge \mu^p \models l^p$) and updates φ^p and μ^p accordingly, until one of the following facts happens:

(i) μ^p propositionally violates φ^p ($\mu^p \wedge \varphi^p \models \bot$). If so, $\mathcal{T}$-`deduce` behaves like `deduce` in DPLL, returning Conflict.
(ii) μ^p propositionally satisfies φ^p ($\mu^p \models \varphi^p$). If so, $\mathcal{T}$-`deduce` invokes *$\mathcal{T}$-solver* on μ: if the latter returns Sat, then $\mathcal{T}$-`deduce` returns Sat; otherwise, $\mathcal{T}$-`deduce` returns Conflict.
(iii) no more literals can be deduced. If so, $\mathcal{T}$-`deduce` returns Unknown. A slightly more elaborated version of $\mathcal{T}$-`deduce` can invoke *$\mathcal{T}$-solver* on μ at this intermediate stage: if *$\mathcal{T}$-solver* returns Unsat, then $\mathcal{T}$-`deduce` returns Conflict. (This enhancement, called *early pruning*, is discussed in §3.3.)

A much more elaborated version of $\mathcal{T}$-`deduce` can be implemented if *$\mathcal{T}$-solver* is able to perform deductions of unassigned literals $\eta \models_{\mathcal{T}} l$ s.t. $\eta \subseteq \mu$, as in §3.1. If so, $\mathcal{T}$-`deduce` can iteratively deduce and propagate also the corresponding literal l^p. (This enhancement, called *$\mathcal{T}$-propagation*, is discussed in §3.3.)

$\mathcal{T}$-`analyze_conflict` is an extensions of `analyze_conflict` of DPLL [42, 43]: if the conflict produced by $\mathcal{T}$-`deduce` is caused by a boolean failure (case (i) above), then $\mathcal{T}$-`analyze_conflict` produces a boolean conflict set η^p and the corresponding value of `blevel`; if the conflict is caused by a $\mathcal{T}$-inconsistency revealed by *$\mathcal{T}$-solver* (case (ii) or (iii) above), then $\mathcal{T}$-`analyze_conflict` produces the boolean abstraction η^p of the theory conflict set $\eta \subseteq \mu$ produced by *$\mathcal{T}$-solver*, or computes a mixed boolean+theory conflict set by a backward-traversal of the implication graph starting from the conflicting clause $\neg\eta^p$ (see §3.3). Once the conflict set η^p and `blevel` have been computed, $\mathcal{T}$-`backtrack` behaves analogously to `backtrack` in DPLL: it adds the clause $\neg\eta^p$ to φ^p, either temporarily or permanently, and backtracks up to `blevel`. (These features, called *$\mathcal{T}$-backjumping* and *$\mathcal{T}$-learning*, are discussed in §3.3.)

$\mathcal{T}$-DPLL differs from the standard DPLL [42, 43] because it exploits:

– an extended notion of *deduction of literals*: not only *boolean deduction* ($\mu^p \wedge \varphi^p \models l^p$), but also *theory deduction* ($\mu \models_{\mathcal{T}} l$);
– an extended notion of *conflict*: not only *boolean conflict* ($\mu^p \wedge \varphi^p \models_p \bot$), but also *theory conflict* ($\mu \models_{\mathcal{T}} \bot$), or even *mixed boolean+theory conflict* ($(\mu \wedge \varphi) \models_{\mathcal{T}} \bot$).

Example 4. Consider the $\mathcal{LA}(\mathbb{Q})$-formulas φ and its boolean abstraction φ^p of Figure 5. Suppose $\mathcal{T}$-`decide_next_branch` selects, in order, $\mu^p := \{\neg B_5, B_8, B_6, \neg B_1\}$ (in c_4, c_7, c_6, and c_1). $\mathcal{T}$-`deduce` cannot unit-propagate any literal. By the enhanced version of step (iii), it invokes *$\mathcal{T}$-solver* on $\mu := \{\neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2 - x_4 \leq 6), \neg(2x_2 - x_3 > 2)\}$. The enhanced $\mathcal{T}$-solver not only returns Sat, but also it deduces $\neg(3x_1 - 2x_2 \leq 3)$ (c_3 and c_5) as a consequence of the first and last literals. The corresponding boolean literal $\neg B_3$, is added to μ^p and propagated ($\mathcal{T}$-propagation). Hence A_1, A_2 and B_2 are unit-propagated from c_5, c_3 and c_2.

Let μ'^p be the resulting assignment $\{\neg B_5, B_8, B_6, \neg B_1, \neg B_3, A_1, A_2, B_2\}$). By step (iii), $\mathcal{T}$-`deduce` invokes *$\mathcal{T}$-solver* on μ': $\{\neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2-$

$\varphi =$	$\varphi^p =$
$c_1 : \neg(2x_2 - x_3 > 2) \vee A_1$	$\neg B_1 \vee A_1$
$c_2 : \neg A_2 \vee (x_1 - x_5 \leq 1)$	$\neg A_2 \vee B_2$
$c_3 : (3x_1 - 2x_2 \leq 3) \vee A_2$	$B_3 \vee A_2$
$c_4 : \neg(2x_3 + x_4 \geq 5) \vee \neg(3x_1 - x_3 \leq 6) \vee \neg A_1$	$\neg B_4 \vee \neg B_5 \vee \neg A_1$
$c_5 : A_1 \vee (3x_1 - 2x_2 \leq 3)$	$A_1 \vee B_3$
$c_6 : (x_2 - x_4 \leq 6) \vee (x_5 = 5 - 3x_4) \vee \neg A_1$	$B_6 \vee B_7 \vee \neg A_1$
$c_7 : A_1 \vee (x_3 = 3x_5 + 4) \vee A_2$	$A_1 \vee B_8 \vee A_2$

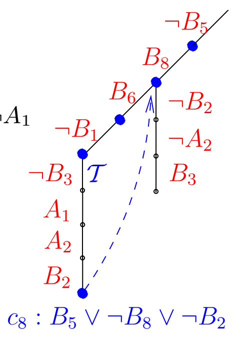

Fig. 5. Boolean search (sub)tree in the scenario of Example 4. (A diagonal line, a vertical line and a vertical line tagged with "$\mathcal{T}$" denote literal selection, unit propagation and $\mathcal{T}$-propagation respectively; a bullet "●" denotes a call to $\mathcal{T}$-*solver*.)

$x_4 \leq 6), \neg(2x_2 - x_3 > 2), \neg(3x_1 - 2x_2 \leq 3), (x_1 - x_5 \leq 1)\}$ which is inconsistent because of the 1st, 2nd, and 6th literals, so that returns Unsat, and hence $\mathcal{T}$`-deduce` returns Conflict. Then $\mathcal{T}$`-analyze_conflict` and $\mathcal{T}$`-backtrack` learn the corresponding boolean conflict clause

$$c_8 =_{def} B_5 \vee \neg B_8 \vee \neg B_2$$

and backtrack, popping from μ^p all literals up to $\{\neg B_5, B_8\}$, and then unit-propagate $\neg B_2$ on c_8 ($\mathcal{T}$-backjumping and $\mathcal{T}$-learning). Then, starting from $\{\neg B_5, B_8, \neg B_2\}$, also $\neg A_2$ and B_3 are unit-propagated on c_2 and c_3 respectively.

As in standard DPLL, an excessive number of $\mathcal{T}$-learned clauses may cause an explosion in size of φ. Thus, many lazy *SMT* tools introduce techniques for *discharging* $\mathcal{T}$-learned clauses when necessary. Moreover, like in standard DPLL, $\mathcal{T}$-DPLL can be *restarted* from scratch in order to avoid dead-end portions of the search space. The learned clauses prevent $\mathcal{T}$-DPLL to redo the same steps twice. Most lazy *SMT* tools implement restarting mechanisms as well.

3.3 Enhancements

In the schema of Figure 4, even assuming that the DPLL engine and the $\mathcal{T}$-*solver* are extremely efficient as a stand-alone procedures, their combination can be extremely inefficient. This is due to a couple of intrinsic problems.

- The DPLL engine assigns truth values to (the boolean abstraction of) $\mathcal{T}$-atoms in a blind way, receiving no information from $\mathcal{T}$-*solver* about their semantics. This may cause up to an huge amount of calls to $\mathcal{T}$-*solver* on assignments which are obviously $\mathcal{T}$-inconsistent, or whose $\mathcal{T}$-inconsistency could have been easily derived from that of previously-checked assignments.

- The $\mathcal{T}$*-solver* is used as a memory-less subroutine, in a master-slave fashion. Therefore $\mathcal{T}$*-solver* may be called on assignments that are subsets of, supersets of or similar to assignments it has already checked, with no chance of reusing previous computations.

Therefore, it is essential to improve the integration schema so that the DPLL solver is driven in its boolean search by $\mathcal{T}$-dependent information provided by $\mathcal{T}$*-solver*, whilst the latter is able to take benefit from information provided by the former, and it is given a chance of reusing previous computation.

We describe some of the most effective techniques which have been proposed in order to optimize the interaction between DPLL and $\mathcal{T}$*-solver*. (We refer the reader to [36] for a much more extensive and detailed survey.) Some of them, like Normalizing $\mathcal{T}$-atoms, Early pruning, $\mathcal{T}$-backjumping and pure-literal filtering, derive from those developed in the context of DPLL-based procedures for modal logics.

Normalizing $\mathcal{T}$-atoms. In order to avoid the generation of many trivially-unsatisfiable assignments, it is wise to preprocess $\mathcal{T}$-atoms so that to map as many as possible $\mathcal{T}$-equivalent literals into syntactically-identical ones. This can be achieved by applying some rewriting rules, like, e.g.:

- *Drop dual operators*: $(x_1 < x_2)$, $(x_1 \geq x_2) \Rightarrow \neg(x_1 \geq x_2)$, $(x_1 \geq x_2)$.
- *Exploit associativity*: $(x_1 + (x_2 + x_3) = 1)$, $((x_1 + x_2) + x_3) = 1) \Rightarrow (x_1 + x_2 + x_3 = 1)$.
- *Sort*: $(x_1 + x_2 - x_3 \leq 1)$, $(x_2 + x_1 - 1 \leq x_3) \Rightarrow (x_1 + x_2 - x_3 \leq 1))$.
- *Exploit* $\mathcal{T}$*-specific properties*: $(x_1 \leq 3)$, $(x_1 < 4) \Rightarrow (x_1 \leq 3)$ if $x_1 \in \mathbb{Z}$.

The applicability and effectiveness of these mappings depends on the theory $\mathcal{T}$.

Static learning. On some specific kind of problems, it is possible to quickly detect a priori short and "obviously $\mathcal{T}$-inconsistent" assignments to $\mathcal{T}$-atoms in $Atoms(\varphi)$ (typically pairs or triplets). Some examples are:

- *incompatible values* (e.g., $\{x = 0, x = 1\}$),
- *congruence constraints* (e.g., $\{(x_1 = y_1), (x_2 = y_2), \neg(f(x_1, x_2) = f(y_1, y_2))\}$),
- *transitivity constraints* (e.g., $\{(x - y \leq 2), (y - z \leq 4), \neg(x - z \leq 7)\}$),
- *equivalence constraints* ($\{(x = y), (2x - 3z \leq 3), \neg(2y - 3z \leq 3)\}$).

If so, the clauses obtained by negating the assignments (e.g., $\neg(x = 0) \vee \neg(x = 1)$) can be added a priori to the formula before the search starts. Whenever all but one literals in the inconsistent assignment are assigned, the negation of the remaining literal is assigned deterministically by unit propagation, which prevents the solver generating any assignment which include the inconsistent one. This technique may significantly reduce the boolean search space, and hence the number of calls to $\mathcal{T}$*-solver*, producing very relevant speed-ups [1, 6].

Intuitively, one can think to static learning as suggesting a priori some small and "obvious" $\mathcal{T}$-valid lemmas relating some $\mathcal{T}$-atoms of φ, which drive DPLL in its boolean search. Notice that the clauses added by static learning refer only to atoms which already occur in the original formula, so that the boolean search space is not enlarged.

Early pruning. Another optimization, here generically called *early pruning* – *EP*, is to introduce an intermediate call to $\mathcal{T}$*-solver* on intermediate assignment μ. (I.e., in the $\mathcal{T}$-DPLL schema of Figure 4, this is represented by the "slightly more elaborated" version of step (iii) of $\mathcal{T}$-deduce.) If $\mathcal{T}$*-solver*(μ) returns Unsat, then all possible extensions of μ are unsatisfiable, so that $\mathcal{T}$-DPLL returns Unsat and backtracks, avoiding a possibly big amount of useless search.

In general, EP may introduce a drastic reduction of the boolean search space, and hence of the number of calls to $\mathcal{T}$*-solvers*. Unfortunately, as EP may cause useless calls to $\mathcal{T}$*-solver*, the benefits of the pruning effect may be partly counterbalanced by the overhead introduced by the extra EP calls. To this extent, many different improvements to EP and strategies for interleaving calls to $\mathcal{T}$*-solvers* and boolean reasoning steps [41, 19, 3, 6, 10] have been proposed.

$\mathcal{T}$-propagation. As discussed in §3.1, for some theories it is possible to implement $\mathcal{T}$*-solver* so that a call to $\mathcal{T}$*-solver*(μ) returning Sat can also perform one or more deduction(s) in the form $\eta \models_{\mathcal{T}} l$, s.t. $\eta \subseteq \mu$ and l is a literal on an unassigned atom in φ. If this is the case, then $\mathcal{T}$*-solver* can return l to $\mathcal{T}$-DPLL, so that l^p is added to μ^p and unit-propagated [1, 3, 19]. This process, which is called $\mathcal{T}$-propagation, may induce a beneficial loop with unit-propagation. As with early-pruning, there are different strategies by which $\mathcal{T}$-propagation can be interleaved with unit-propagation [1, 3, 19, 6, 10, 33].

Notice that $\mathcal{T}$*-solver* can return the deduction(s) performed $\eta \models_{\mathcal{T}} l$ to $\mathcal{T}$-DPLL, which can add the deduction clause $(\eta^p \rightarrow l^p)$ to φ^p, either temporarily and permanently. The deduction clause will be used for the future boolean search, with benefits analogous to those of $\mathcal{T}$-learning (see §3.3).

$\mathcal{T}$-backjumping and $\mathcal{T}$-learning. Modern implementations inherit the backjumping mechanism of current DPLL tools: $\mathcal{T}$-DPLL learns the conflict clause $\neg\eta^p$ and backtracks to the highest point in the stack where one $l^p \in \eta^p$ is not assigned, and unit propagates $\neg l^p$ on $\neg\eta^p$. Intuitively, DPLL backtracks to the highest point where it would have done something different if it had known in advance the conflict clause $\neg\eta^p$ from the $\mathcal{T}$*-solver*.

As hinted in §3.2, it is possible to use either a theory conflict η (i.e., $\neg\eta$ is a $\mathcal{T}$-valid clause) or a *mixed boolean+theory conflicts sets* η', i.e., s.t. an inconsistency can be entailed from $\eta' \wedge \varphi$ by means of a combination of boolean and theory reasoning ($\eta' \wedge \varphi \models_{\mathcal{T}} \bot$). Such conflict sets/clauses can be obtained starting from the theory-conflicting clause $\neg\eta^p$ by applying the backward-traversal of the implication graph, until one of the standard conditions (e.g., 1UIP) is achieved. Notice that it is possible to learn *both* clauses $\neg\eta$ and $\neg\eta'$.

Example 5. The scenario depicted in Example 4 represents a form of $\mathcal{T}$-backjumping and $\mathcal{T}$-learning, in which the conflict clause c_8 used is a $\mathcal{LA}(\mathbb{Q})$-conflict clause (i.e., $\mathcal{P}2\mathcal{T}(c_8)$ is $\mathcal{LA}(\mathbb{Q})$-valid). However, $\mathcal{T}$`-analyze_conflict` could instead look for a mixed boolean+theory conflict clause by treating c_8 as a conflicting clause and backward-traversing the implication graph, that is, by resolving backward c_8 with c_2 and c_3, (i.e., with the antecedent clauses of B_2 and A_2) and with the deduction clause c_9 (which "caused" the propagation of $\neg B_3$):

$$\cfrac{\cfrac{\cfrac{\overbrace{B_5 \vee \neg B_8 \vee \neg B_2}^{c_8:\ \textit{theory conflicting clause}} \qquad \overbrace{\neg A_2 \vee B_2}^{c_2}}{B_5 \vee \neg B_8 \vee \neg A_2}\,(B_2) \qquad \overbrace{B_3 \vee A_2}^{c_3}}{B_5 \vee \neg B_8 \vee B_3}\,(\neg A_2) \qquad \overbrace{B_5 \vee B_1 \vee \neg B_3}^{c_9}}{\underbrace{B_5 \vee \neg B_8 \vee B_1}_{c_8':\ \textit{mixed boolean+theory conflict clause}}}\,(B_3)$$

finding the mixed boolean+theory conflict clause $c_8' : \ B_5 \vee \neg B_8 \vee B_1$. (Notice that, $\mathcal{P}2\mathcal{T}(c_8') = (3x_1 - x_3 \leq 6) \vee \neg(x_3 = 3x_5 + 4) \vee (2x_2 - x_3 > 2)$ is not $\mathcal{LA}(\mathbb{Q})$-valid.) If so then $\mathcal{T}$`-backtrack` pops from μ^p all literals up to $\{\neg B_5, B_8\}$, and then unit-propagates B_1 on c_8', and hence A_1 on c_1.

As with static learning, the clauses added by $\mathcal{T}$-learning refer only to atoms which already occur in the original formula, so that no new atom is added. [18] proposed an interesting generalization of $\mathcal{T}$-learning, in which learned clause may contain also new atoms. [7, 8] used a similar idea to improve the efficiency of Delayed Theory Combination (see §4).

Pure-literal filtering. If we have non-boolean $\mathcal{T}$-atoms occurring only positively [resp. negatively] in the input formula, we can safely drop every negative [resp. positive] occurrence of them from the assignment to be checked by $\mathcal{T}$*-solver* [41, 22, 3, 6, 36]. [5] We call this technique, *pure-literal filtering*

There are two potential benefits for this behavior. Let μ' be the reduced version of μ. First, μ' might be $\mathcal{T}$-satisfiable despite μ is $\mathcal{T}$-unsatisfiable. If so, and if μ propositionally satisfies φ, then $\mathcal{T}$-DPLL can stop, potentially saving a lot of search. Second, if μ' (and hence μ) is $\mathcal{T}$-unsatisfiable, then checking the consistency of μ' rather than that of μ can be faster and cause smaller conflict sets, so that to improve the effectiveness of $\mathcal{T}$-backjumping and $\mathcal{T}$-learning.

Moreover, this technique is particularly useful in some situations. For instance, many $\mathcal{T}$*-solvers* for $\mathcal{DL}(\mathbb{Z})$ and $\mathcal{LA}(\mathbb{Z})$ cannot efficiently handle disequalities (e.g., $(x_1 - x_2 \neq 3)$), so that they are forced to split them into the disjunction of strict inequalities $(x_1 - x_2 > 3) \vee (x_1 - x_2 < 3)$. This causes an enlargement of the search, because the two disjuncts must be investigated

[5] If both $\mathcal{T}$-propagation and pure-literal filtering are implemented, then the filtered literals must be dropped not only from the assignment, but also from the list of literals which can be $\mathcal{T}$-deduced, so that to avoid the $\mathcal{T}$-propagation of literals which have already been filtered away.

separately. In many problems, however, it is very frequent that most equalities $(t_1 = t_2)$ occur with positive polarity only. If so, then pure-literal filtering avoids adding $(t_1 \neq t_2)$ to μ when $(t_1 = t_2)^p$ is assigned to false by $\mathcal{T}$-DPLL, so that no split is needed [3].

4 DPLL for Theory Combination: DTC

We consider the *SMT* problem in the case of combined theories, *SMT*$(\mathcal{T}_1 \cup \mathcal{T}_2)$. In the original Nelson-Oppen method [31] and its variant due to Shostak [38] (hereafter referred as *deterministic N.O.* [6]) the two $\mathcal{T}$*-solvers* cooperate by inferring and exchanging equalities between shared terms (interface equalities), until either one $\mathcal{T}$*-solver* detects unsatisfiability (Unsat case), or neither can perform any more entailment (Sat case). In case of a non-convex theory $\mathcal{T}_i$, the $\mathcal{T}_i$*-solver* may generate a disjunction of interface equalities; consequently, a $\mathcal{T}_i$*-solver* receiving a disjunction of equalities from the other one is forced to case-split on each disjunct. Deterministic N.O. requires that each $\mathcal{T}$*-solver* is always capable to inferring the (disjunctions of) equalities which are entailed by the input set of literals (see §3.1). Whilst for some theories this feature can be implemented very efficiently (e.g., $\mathcal{EUF}$ [32]), for some others it can be extremely expensive (e.g., $\mathcal{DL}(\mathbb{Z})$ [30]).

*Delayed Theory Combination (*Dtc*)* is a general method for tackling the problem of theory combination within the context of lazy *SMT* [7, 8]. As with N.O., we assume that $\mathcal{T}_1$, $\mathcal{T}_2$ are two signature-disjoint stably-infinite theories with their respective $\mathcal{T}_i$*-solvers*. Importantly, no assumption is made about the e_{ij}-deduction capabilities of the $\mathcal{T}_i$*-solvers* (§3.1): for each $\mathcal{T}_i$*-solver*, every intermediate situation from complete e_{ij}-deduction (like in deterministic N.O.) to no e_{ij}-deduction capabilities (like in non-deterministic N.O.) is admitted.

In a nutshell, in Dtc the embedded DPLL engine not only enumerates truth assignments for the atoms of the input formula, but also assigns truth values for the interface equalities that the $\mathcal{T}$*-solver*'s are not capable of inferring, and handles the case-split induced by the entailment of disjunctions of interface equalities in non-convex theories. The rationale is to exploit the full power of a modern DPLL engine by delegating to it part of the heavy reasoning effort previously due to the $\mathcal{T}_i$*-solvers*.

An implementation of Dtc [8, 9] is based on the schema of Figure 4, exploiting early pruning, $\mathcal{T}$-propagation, $\mathcal{T}$-backjumping and $\mathcal{T}$-learning. Each of the two $\mathcal{T}_i$*-solvers* interacts only with the DPLL engine by exchanging literals via the truth assignment μ in a stack-based manner, so that there is no direct exchange of information between the $\mathcal{T}_i$*-solvers*. Let $\mathcal{T}$ be $\mathcal{T}_1 \cup \mathcal{T}_2$. The $\mathcal{T}$-DPLL algorithm is modified to the following extents [8, 9]: [7]

– $\mathcal{T}$-DPLL must be instructed to assign truth values not only to the atoms in φ, but also to the interface equalities not occurring in φ. $\mathcal{P}2\mathcal{T}$ and $\mathcal{T}2\mathcal{P}$

[6] We also call *nondeterministic N.O.* the non-deterministic variant of N.O. method first presented in [34].

[7] For simplicity, we assume φ is pure, although this condition is not necessary.)

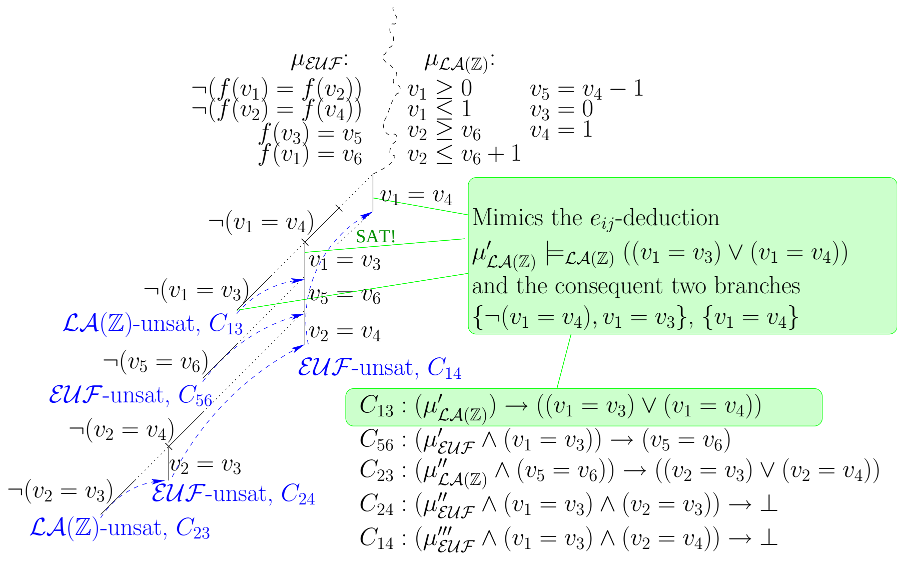
$\mu_{\mathcal{EUF}}$:
$\neg(f(v_1) = f(v_2))$
$\neg(f(v_2) = f(v_4))$
$f(v_3) = v_5$
$f(v_1) = v_6$
$\mu_{\mathcal{LA}(\mathbb{Z})}$:
$v_1 \geq 0$
$v_1 \leq 1$
$v_2 \geq v_6$
$v_2 \leq v_6 + 1$
$v_5 = v_4 - 1$
$v_3 = 0$
$v_4 = 1$
$v_1 = v_4$
$\neg(v_1 = v_4)$
SAT!
$v_1 = v_3$
$\neg(v_1 = v_3)$
$v_5 = v_6$
$\mathcal{LA}(\mathbb{Z})$-unsat, C_{13}
$v_2 = v_4$
$\neg(v_5 = v_6)$
$\mathcal{EUF}$-unsat, C_{14}
$\mathcal{EUF}$-unsat, C_{56}
$\neg(v_2 = v_4)$
$v_2 = v_3$
$\neg(v_2 = v_3)$
$\mathcal{EUF}$-unsat, C_{24}
$\mathcal{LA}(\mathbb{Z})$-unsat, C_{23}
Mimics the e_{ij}-deduction
$\mu'_{\mathcal{LA}(\mathbb{Z})} \models_{\mathcal{LA}(\mathbb{Z})} ((v_1 = v_3) \vee (v_1 = v_4))$
and the consequent two branches
$\{\neg(v_1 = v_4), v_1 = v_3\}$, $\{v_1 = v_4\}$
$C_{13} : (\mu'_{\mathcal{LA}(\mathbb{Z})}) \rightarrow ((v_1 = v_3) \vee (v_1 = v_4))$
$C_{56} : (\mu'_{\mathcal{EUF}} \wedge (v_1 = v_3)) \rightarrow (v_5 = v_6)$
$C_{23} : (\mu''_{\mathcal{LA}(\mathbb{Z})} \wedge (v_5 = v_6)) \rightarrow ((v_2 = v_3) \vee (v_2 = v_4))$
$C_{24} : (\mu''_{\mathcal{EUF}} \wedge (v_1 = v_3) \wedge (v_2 = v_3)) \rightarrow \bot$
$C_{14} : (\mu'''_{\mathcal{EUF}} \wedge (v_1 = v_3) \wedge (v_2 = v_4)) \rightarrow \bot$

Fig. 6. The DTC search tree for Example 6 on $\mathcal{LA}(\mathbb{Z}) \cup \mathcal{EUF}$, with no e_{ij}-deduction. $v_1, \ldots, v_6$ are interface terms. $\mu'_{\mathcal{T}_i}$, $\mu''_{\mathcal{T}_i}$, $\mu'''_{\mathcal{T}_i}$ denote generic subsets of $\mu_{\mathcal{T}_i}$, $\mathcal{T} \in \{\mathcal{EUF}, \mathcal{LA}(\mathbb{Z})\}$.

are modified accordingly. In particular, $\mathcal{T}$-`decide_next_branch` is modified to select also new interface equalities not occurring in the original formula.

- μ^p is partitioned into three components $\mu^p_{\mathcal{T}_1}$, $\mu^p_{\mathcal{T}_2}$ and μ^p_e, s.t. $\mu_{\mathcal{T}_i}$ is the set of i-pure literals and μ_e is the set of interface (dis)equalities in μ.
- $\mathcal{T}$-`deduce` is modified to work as follows: for each $\mathcal{T}_i$, $\mu^p_{\mathcal{T}_i} \cup \mu^p_e$, is fed to the respective $\mathcal{T}_i$*-solver*. If both return Sat, then $\mathcal{T}$-`deduce` returns Sat, otherwise it returns Conflict.
- Early-pruning is performed; if some $\mathcal{T}_i$*-solver* can deduce atoms or single interface equalities, then $\mathcal{T}$-propagation is performed. If one $\mathcal{T}_i$-solver performs the e_{ij}-deduction $\mu^* \models_{\mathcal{T}_i} \bigvee_{j=1}^{k} e_j$, s.t. $\mu^* \subseteq \mu_{\mathcal{T}_i} \cup \mu_e$, each e_j being an interface equality, then the deduction clause $\mathcal{T}2\mathcal{B}(\mu^* \rightarrow \bigvee_{j=1}^{k} e_j)$ is learned.
- $\mathcal{T}$-`analyze_conflict` and $\mathcal{T}$-`backtrack` are modified so that to use the conflict set returned by one $\mathcal{T}_i$*-solver* for $\mathcal{T}$-backjumping and $\mathcal{T}$-learning. Importantly, such conflict sets may contain interface equalities.

In order to achieve efficiency, other heuristics and strategies have been further suggested in [7–9], and more recently in [17, 14].

Example 6. [9] Consider the set of $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Z})$-literals $\mu =_{def} \mu_{\mathcal{EUF}} \cup \mu_{\mathcal{LA}(\mathbb{Z})}$ of Figure 6. We assume that both the $\mathcal{EUF}$- and $\mathcal{LA}(\mathbb{Z})$-solvers have no e_{ij}-deduction capabilities (like with non-deterministic N.O.). For simplicity, we also assume that both $\mathcal{T}_i$*-solvers* always return conflict sets which do not contain redundant interface disequalities $\neg e_{ij}$. (We adopt here a strategy for DTC which is described in detail in [9].) In short, $\mathcal{T}$-DPLL performs a boolean search on the e_{ij}'s, backjumping on the $\mathcal{T}$-conflicting clauses C_{13}, C_{56}, C_{23}, C_{24} and C_{14}, which in the end causes the unit-propagation of $(v_1 = v_4)$. Then, $\mathcal{T}$-DPLL selects a sequence of $\neg e_{ij}$'s without generating conflicts, and concludes that the formula is $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiable. Notice that the backjumping steps on the clauses C_{13}, C_{56}, and C_{25} mimic the effects of performing e_{ij}-deductions.

By adopting $\mathcal{T}$*-solvers* with different e_{ij}-deduction power, one can trade part or all the e_{ij}-deduction effort for extra boolean search. [9] shows that, if the $\mathcal{T}$*-solvers* have full e_{ij}-deduction capabilities, then no extra boolean search on the e_{ij}'s is required; otherwise, the boolean search is controlled by the quality of the conflict sets returned by the $\mathcal{T}$*-solvers*: the more redundant $\neg e_{ij}$'s are removed from the conflict sets, the more boolean branches are pruned. If the conflict sets do not contain redundant $\neg e_{ij}$'s, the extra effort is reduced to one branch for each deduction saved, as in Example 6.

Variants of DTC are currently implemented in the MATHSAT [8], YICES [17], and Z3 [14] lazy *SMT* tools.

4.1 Splitting on Demand

The idea of delegating to the DPLL engine part of the heavy reasoning effort previously due to the $\mathcal{T}_i$*-solvers* is pushed even further in the *Splitting on demand* technique proposed in [5]. This work is built on top of the observation that for many theories, in particular for non-convex ones, $\mathcal{T}$*-solvers* must perform lots of internal case-splits in order to decide the satisfiability of a set of literals. Unfortunately most $\mathcal{T}$*-solvers* cannot handle boolean search internally, so that they cannot do anything better then doing naive case-splitting on all possible combinations of the alternatives.

With splitting on demand, whenever the $\mathcal{T}$*-solver* encounters the need of a case-split, it gives back the control to the DPLL engine by returning (the boolean abstraction of) a clause encoding the alternatives, which is learned and split upon by the DPLL engine. (Notice that the atoms encoding the alternatives in the learned clause may not occur in the original formula.) This is repeated until the $\mathcal{T}$*-solver* can decide the $\mathcal{T}$-satisfiability of its input literals without case-splitting. Therefore the $\mathcal{T}$*-solver* delegates the boolean search induced by the case-splits to the DPLL solver, which presumably handles it in a much more efficient way.

References

1. A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proc. European Conference on Planning, CP-99*, 1999.
2. A. Armando and E. Giunchiglia. Embedding Complex Decision Procedures inside an Interactive Theorem Prover. *Annals of Mathematics and Artificial Intelligence*, 8(3–4):475–502, 1993.

3. G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In *Proc. CADE'2002.*, volume 2392 of *LNAI*. Springer, July 2002.
4. C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *14th International Conference on Computer-Aided Verification*, 2002.
5. C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on Demand in SAT Modulo Theories. In *Proc. LPAR'06*, volume 4246 of *LNAI*. Springer, 2006.
6. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P.van Rossum, S. Schulz, and R. Sebastiani. MathSAT: A Tight Integration of SAT and Mathematical Decision Procedure. *Journal of Automated Reasoning*, 35(1-3), October 2005.
7. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Ranise, and R. Sebastiani. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In *Proc. CAV 2005*, volume 3576 of *LNCS*. Springer, 2005.
8. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Ranise, and R. Sebastiani. Efficient Theory Combination via Boolean Search. *Information and Computation*, 204(10), 2006.
9. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. Delayed Theory Combination vs. Nelson-Oppen for Satisfiability Modulo Theories: a Comparative Analysis. In *Proc. LPAR'06*, volume 4246 of *LNAI*. Springer, 2006.
10. S. Cotton and O. Maler. Fast and Flexible Difference Logic Propagation for DPLL(T). In *Proc. SAT'06*, volume 4121 of *LNCS*. Springer, 2006.
11. M. D'Agostino and M. Mondadori. The Taming of the Cut. *Journal of Logic and Computation*, 4(3):285–319, 1994.
12. M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.
13. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
14. L. de Moura and N. Bjorner. Model-based Theory Combination. In *Proc. 5th workshop on Satisfiability Modulo Theories, SMT'07*, 2007. To appear.
15. L. de Moura, H. Rueß, and M. Sorea. Lemmas on Demand for Satisfiability Solvers. Proc. SAT'02, 2002.
16. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proc. CAV'06*, volume 4144 of *LNCS*. Springer, 2006.
17. B. Dutertre and L. de Moura. System Description: Yices 1.0. In *Proc. on 2nd SMT competition, SMT-COMP'06*, 2006. Available at `yices.csl.sri.com/yices-smtcomp06.pdf`.
18. C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem Proving Using Lazy Proof Explication. In *Proc. CAV 2003*, LNCS. Springer, 2003.
19. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *Proc. CAV'04*, volume 3114 of *LNCS*. Springer, 2004.
20. E. Giunchiglia, F. Giunchiglia, R. Sebastiani, and A. Tacchella. More evaluation of decision procedures for modal logics. In *Proc. KR'98*, Trento, Italy, 1998.
21. E. Giunchiglia, F. Giunchiglia, R. Sebastiani, and A. Tacchella. SAT vs. Translation based decision procedures for modal logics: a comparative evaluation. *Journal of Applied Non-Classical Logics*, 10(2):145–172, 2000.
22. E. Giunchiglia, F. Giunchiglia, and A. Tacchella. SAT Based Decision Procedures for Classical Modal Logics. Journal of Automated Reasoning. Special Issue: Satisfiability at the start of the year 2000, 2001.

23. F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. CADE'13*, LNAI, New Brunswick, NJ, USA, August 1996. Springer.
24. F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for ALC. In *Proc. KR'96*, Cambridge, MA, USA, November 1996.
25. F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K(m). *Information and Computation*, 162(1/2), October/November 2000.
26. V. Haarslev and R. Moeller. RACER System Description. In *Proc. of International Joint Conference on Automated reasoning - IJCAR-2001*, volume 2083 of *LNAI*, Siena, Italy, July 2001. Springer-verlag.
27. I. Horrocks. Using an expressive description logic: FaCT or fiction? In *Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 636–647, 1998.
28. I. Horrocks and P. F. Patel-Schneider. Optimizing Description Logic Subsumption. *Journal of Logic and Computation*, 9(3):267–293, 1999.
29. I. Horrocks, P. F. Patel-Schneider, and R. Sebastiani. An Analysis of Empirical Testing for Modal Decision Procedures. *Logic Journal of the IGPL*, 8(3):293–323, May 2000.
30. S. K. Lahiri and M. Musuvathi. An Efficient Decision Procedure for UTVPI Constraints. In *Proc. of 5th International Workshop on Frontiers of Combining Systems (FroCos '05)*, volume 3717 of *LNCS*. Springer, 2005.
31. G. Nelson and D.C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, 1979.
32. R. Nieuwenhuis and A. Oliveras. Proof-Producing Congruence Closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications, RTA'05*, volume 3467 of *LNCS*. Springer, 2005.
33. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.
34. Derek C. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, 12:291–302, 1980.
35. P. F. Patel-Schneider. DLP System Description. In *Proc. Int. Workshop on Description Logics, DL'98*, 1998.
36. R. Sebastiani. Lazy Satisfiability Modulo Theories. Technical Report dtr-07-022, DIT, University of Trento, Italy, April 2007. Available at `http://eprints.biblio.unitn.it/archive/00001196/01/dtr-07-022.pdf`.
37. R. Sebastiani and A. Villafiorita. SAT-based decision procedures for normal modal logics: a theoretical framework. In *Proc. AIMSA'98*, volume 1480 of *LNAI*. Springer, 1998.
38. R.E. Shostak. Deciding Combinations of Theories. *Journal of the ACM*, 31:1–12, 1984.
39. R. M. Smullyan. *First-Order Logic*. Springer-Verlag, NY, 1968.
40. C. Tinelli. A DPLL-based Calculus for Ground Satisfiability Modulo Theories. In *Proc. JELIA-02*, volume 2424 of *LNAI*, pages 308–319. Springer, 2002.
41. S. Wolfman and D. Weld. The LPSAT Engine & its Application to Resource Planning. In *Proc. IJCAI*, 1999.
42. L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.
43. L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proc. CAV'02*, number 2404 in LNCS, pages 17–36. Springer, 2002.