UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

Lazy Satisfiability Modulo Theories

Roberto Sebastiani

April 2007

Technical Report # DIT-07-022

Journal on Satisfiability, Boolean Modeling and Computation 1 (2006)

Lazy Satisfiability Modulo Theories

Roberto Sebastiani * roberto.sebastiani@dit.unitn.it
Dept. of Information and Communication Technologies (DIT), University of Trento, Italy.

Abstract

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first-order formula with respect to some decidable first-order theory $\mathcal{T}$ ($SMT(\mathcal{T})$). These problems are typically not handled adequately by standard automated theorem provers. SMT is being recognized as increasingly important due to its applications in many domains in different communities, in particular in formal verification. An amount of papers with novel and very efficient techniques for SMT has been published in the last years, and some very efficient SMT tools are now available.

Typical $SMT(\mathcal{T})$ problems require testing the satisfiability of formulas which are boolean combinations of atomic propositions and atomic expressions in $\mathcal{T}$, so that heavy boolean reasoning must be efficiently combined with expressive theory-specific reasoning. The dominating approach to $SMT(\mathcal{T})$, called *lazy approach*, is based on the integration of a SAT solver and of a decision procedure able to handle sets of atomic constraints in $\mathcal{T}$ ($\mathcal{T}$*-solver*), handling respectively the boolean and the theory-specific components of reasoning.

Unfortunately, neither the problem of building an efficient SMT solver, nor even that of acquiring a comprehensive background knowledge in lazy SMT, is of simple solution.

In this paper we present an extensive survey of SMT, with particular focus on the lazy approach. We survey, classify and analyze from a theory-independent perspective the most effective techniques and optimizations which are of interest for lazy SMT and which have been proposed in various communities; we discuss their relative benefits and drawbacks; we provide some guidelines about their choice and usage; we also analyze the features for SAT solvers and $\mathcal{T}$*-solvers* which make them more suitable for an integration.

The ultimate goals of this paper are to become a source of a common background knowledge and terminology for students and researchers in different areas, to provide a domain-independent reference guide for developers of SMT tools, and to stimulate the cross-fertilization of techniques and ideas among different communities.

Keywords: *Propositional Satisfiability (SAT), Satisfiability Modulo Theories (SMT), Decision Procedures*

[*] This survey work has benefited from the collaboration of and important discussions with Alessandro Armando, Clark Barrett, Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Anders Franzen, Leonardo de Moura, Fausto Giunchiglia, Enrico Giunchiglia, Alberto Griggio, Robert Nieuwenhuis, Albert Oliveras, Silvio Ranise, Marco Roveri, Ofer Strichman, Aaron Stump, Armando Tacchella, Cesare Tinelli, Stefano Tonetta, Moshe Y. Vardi, Michele Vescovi, to whom I am very grateful. A particular thank goes to all past and present members of the KSAT and MathSAT teams.
Some material discussed here was presented at the ESSLI'02 course and IJCAI'03 and CADE'03 Tutorials "SAT Beyond Propositional Satisfiability", whose slides can be downloaded from `www.dit.unitn.it/~rseba/DIDATTICA/Tutorials/Slides_tutorial_cade03.pdf`, and at the course "Efficient Boolean Reasoning" at 2003, 2004 and 2005 International Doctorate School on ICT of Trento and at 2006 International BIT School on ICT in Brixen, whose slides are available at `www.dit.unitn.it/~rseba/DIDATTICA/SAT_BASED06/`.

Contents

List of Figures

1. Introduction

In this paper we present an extensive survey of *Satisfiability Modulo Theories (SMT)*, with particular focus on the currently-most-effective approach to SMT, the *lazy approach.*

1.1 Satisfiability Modulo Theories - 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 ($\mathcal{LI}$)*. These problems are typically not handled adequately by standard automated theorem provers —like, e.g., those based on resolution calculus— because the latter cannot satisfactorily deal with the theory-specific interpreted symbols (i.e., constants, functions, predicates). [1.]

SMT is being recognized as increasingly important due to its applications in many domains in different communities, ranging from resource planning [WW99] and temporal reasoning [ACG99] to formal verification, the latter including verification of pipelines and of circuits at Register-Transfer Level (RTL) [BD94, PICW04, BBC+06a], of proof obligations in software systems [RD03], of compiler optimizations [BFG+05], of real-time embedded systems [ACKS02, dMRS02a, ABCS03].

An amount of papers with novel and very efficient techniques for SMT has been published in the last years, and some very efficient SMT tools are now available (e.g., Ario [SS05], BarceLogic [NO05a], CVCLite [BB04], DLSAT [MNAM02], haRVey [RD03], MathSAT [BBC+05a], Sateen [KS06], SDSAT [GTG06] Simplify [DNS05], TSAT++ [ACGM04], UCLID [LS04], Yices [DdM06], Verifun [FJOS03], Zapato [BCLZ04]). An amount of benchmarks, mostly derived from verification problems, is available at the *SMT-LIB* official page [RT06b, RT06c]. A workshop devoted to SMT [2.] and an official competition on SMT tools [3.] are run yearly.

1.2 Lazy SMT = SAT + $\mathcal{T}$*-solvers*

All applications mentioned above require testing the satisfiability of formulas which are (possibly-big) boolean combinations of atomic propositions and atomic expressions in some theory $\mathcal{T}$, so that heavy boolean reasoning must be efficiently combined with expressive theory-specific reasoning.

On the one hand, in the last decade we have witnessed an impressive advance in the efficiency of propositional satisfiability techniques, SAT [SS96, BS97, MMZ+01, GN02, ES04, EB05]). As a consequence, some hard real-world problems have been successfully solved by encoding them into SAT. SAT solvers are now a fundamental tool in most formal verification design flows for hardware systems, both for equivalence, property checking, and

[1.] E.g., even handling a simple formula in $\mathcal{LA}(\mathbb{Z})$ like $(x \leq y) \rightarrow (x \leq y + 1024)$ could be a problem for a resolution-based theorem prover, because it would need an axiomatic formalization of the interpreted symbols “1024”, “+” and “≤”. See, e.g., [RT06a] for some more discussion on this issue.

[2.] SMT'07, previously called “PDPAR”. See http://www.lsi.upc.edu/~oliveras/smt07/.

[3.] SMT-COMP05/06 [BdMS05]. See http://www.csl.sri.com/users/demoura/smt-comp/.

ATPG [BCCZ99, McM02, SBSV96]; other application areas include, e.g., the verification of safety-critical systems [SS90, Bor97], and AI planning in its classical formulation [KMS96], and in its extensions to non-deterministic domains [CGT03, HB05]. Plain boolean logic, however, is not expressive enough for representing many other real-world problems (including, e.g., the verification of pipelined microprocessors, of real-time and hybrid control systems, and the analysis of proof obligations in software verification); in other cases, such as the verification of RTL designs or assembly-level code, even if boolean logic is expressive enough to encode the verification problem, it does not seem to be the most effective level of abstraction (e.g., words in the data path are typically treated as collections of unrelated boolean variables).

On the other hand, decision procedures for much more expressive decidable logics have been conceived and implemented in different communities, like, e.g., automated theorem proving, operational research, knowledge representation and reasoning, AI planning, CSP, formal verification. In particular, since the pioneering work of Nelson & Oppen [NO79, NO80, Opp80] and Shostak [Sho79, Sho84], efficient procedures have been conceived and implemented which are able to check the consistency of sets/conjunctions of atomic expressions in decidable F.O. theories. (We call these procedures, *Theory Solvers* or *$\mathcal{T}$-solvers*.) To this extent, most effort has been concentrated in producing *$\mathcal{T}$-solvers* of increasing expressiveness and efficiency and, in particular, in combining them in the most efficient way (e.g., [NO79, NO80, Opp80, Sho79, Sho84, FORS01, BDS02b, SR02]). These procedures, however, deal only with conjunctions of atomic constraints, and thus cannot handle the boolean component of reasoning.

In the last ten years new techniques for efficiently integrating SAT solvers with logic-specific or theory-specific decision procedures have been proposed in different communities and domains, producing big performance improvements when applied (see, e.g., [GS96a, Hor98, PS98, ACG99, WW99, dMRS02a, BDS02a, ABC$^+$02a, Tin02, FJOS03, GHN$^+$04, BBC$^+$05a, SS06b]). Most such systems have been implemented on top of SAT techniques based on variants of the DPLL algorithm [DP60, DLL62, SS96, BS97, MMZ$^+$01, GN02, ES04, EB05]. [4.]

In particular, the dominating approach to $SMT(\mathcal{T})$, which underlies most state-of-the-art $SMT(\mathcal{T})$ tools, is based on the integration of a SAT solver and one (or more) *$\mathcal{T}$-solver*(s), respectively handling the boolean and the theory-specific components of reasoning: the SAT solver enumerates truth assignments which satisfy the boolean abstraction of the input formula, whilst the *$\mathcal{T}$-solver* checks the consistency in $\mathcal{T}$ of the set of literals corresponding to the assignments enumerated. This approach is called *lazy*, in contraposition of the *eager* approach to $SMT(\mathcal{T})$, consisting on encoding a SMT formula into an equivalently-satisfiable boolean formula, and on feeding the result to a SAT solver (see, e.g., [VB99, BLS02, SSB02, Str02, SLB03]). All the most extensive empirical evaluations performed in the last years [GHN$^+$04, dMR04, NO05a, BBC$^+$05b, BdMS05, SMT05, SMT06] confirm the fact that currently all the most efficient SMT tools are based on the lazy approach.

[4.] Notice that the idea of integrating decision procedures and DPLL, and many techniques for optimizing this integration, were conceived and implemented in the domain of modal and description logics [GS96a, GS96b, Hor98, PS98, GGST98], and have been imported into the SMT domain only lately [ACG99].

1.3 Motivations and goals of the paper

The writing of this survey paper is motivated by the following facts.

First, the problem of efficiently combining modern SAT solvers and state-of-the-art decision procedures into a lazy SMT solver is not of simple solution. In fact, the efficiency of a combined procedure does not come straightforwardly from the efficiency of its two components: a naive integration of extremely-efficient SAT solvers and $\mathcal{T}$-solvers may end up into very inefficient tools if the integration is not done properly. For instance, a naively-integrated SAT solver may cause big amounts of redundant calls to the $\mathcal{T}$-solver (see §6).

Moreover, with lazy SMT the choice of the suitable procedures for the SAT solvers or *$\mathcal{T}$-solvers* is not always straightforward. In fact, the features which make a SAT solver or a *$\mathcal{T}$-solver* suitable for an efficient integration are often different from those which make them efficient as standalone solvers. For instance, some features which contribute to improve the efficiency of a modern DPLL solver may have some drawbacks when used within a lazy SMT solver (see §7.2); moreover, some features of a *$\mathcal{T}$-solver* which allow for maximizing the synergy with the SAT solver (see §4.1) are often more important than the efficiency of the *$\mathcal{T}$-solver* itself.

Second, acquiring a comprehensive background knowledge in lazy SMT from the literature may be a complicate task. In fact, the information on techniques of interest is scattered into a plethora of papers from heterogeneous research communities (e.g., Automated Reasoning, CSP, EDA, Formal Verification, Knowledge Representation & Reasoning, Planning, Operational Research, SAT), because lazy SMT borrows ideas and techniques from many disciplines (e.g., automated reasoning in modal & description logics, F.O. theorem proving, graph algorithms, linear and integer programming, SAT, ...), some of which have hardly anything to do with logic or even with symbolic computation. For instance, for theories involving arithmetic (e.g., $\mathcal{LA}(\mathbb{Q})$, $\mathcal{LA}(\mathbb{Z})$, $\mathcal{DL}$) the most efficient *$\mathcal{T}$-solvers* are based on numerical algorithms borrowed from linear programming, integer programming and shortest-path (see §4.2), which have been adapted to work in a logic context.

Moreover, in many papers the description of the integration techniques is mixed up with (and often hidden by) lots of domain-specific information, and it is described with domain-specific notation and terminology, so that it may prevent or discourage researchers from other areas to access it. On the whole, there has been a general lack of cross-fertilization among different communities, which is witnessed by the fact that some techniques have been "reinvented" from scratch several times in different contests. For instance, the technique called "$\mathcal{T}$-backjumping" (see §6) has been invented for description logics [Hor98], and then "reinvented" in both the communities of resource planning [WW99] and formal verification [SBD02, dMRS02b], in different moments and without cross-citations.

In this paper we survey, classify and analyze from a theory-independent perspective the most effective techniques and optimizations which are of interest for lazy SMT and which have been proposed in various communities; we discuss their relative benefits and drawbacks; we provide some guidelines about their choice and usage; we also analyze the features for SAT solvers and *$\mathcal{T}$-solvers* which make them more suitable for an integration.

People with a background in SAT may learn from this paper how to extend SAT solvers to work with much more expressive logics; people with a background on decision procedures may learn how to handle boolean reasoning efficiently; people with background on neither area may learn about lazy SMT from scratch. To this extent, the paper is written with a didactic style, explaining basic concepts from scratch and presenting many examples, so that to be at the reach also of students and of researcher from other communities.

The ultimate goals of this paper are to become a source of a common background knowledge and terminology for students and researchers in different areas, to provide a domain-independent reference guide for developers of SMT tools, and to stimulate the cross-fertilization of techniques and ideas among different communities.

1.4 Content of the paper

The rest of the paper is organized as follows.

- In §2 we provide the necessary theoretical background. We recall some basic concepts about first-order logic and theories, and provide some formal definitions and results which justify the correctness and completeness of lazy SMT procedures.

- In §3 we report some basic concepts about SAT, surveying the main techniques and optimizations of modern DPLL solvers which are of interest for lazy SMT.

- In §4 we report some basic concepts about $\mathcal{T}$*-solvers*. We discuss the features of $\mathcal{T}$*-solvers* which are most important for lazy SMT, and briefly survey the most interesting theories and their relative $\mathcal{T}$*-solvers*.

- In §5 we introduce the basic integration schemata between DPLL and $\mathcal{T}$*-solvers*.

- In §6 we survey and analyze the most effective techniques and optimizations available in the literature, which allow for optimizing the interaction between DPLL and $\mathcal{T}$*-solver*.

- In §7 we provide some guidelines and tips about the choice and usage of such techniques. We also overview a list of problems one may encounter while implementing a lazy SMT tool on top of a modern DPLL implementation, and propose some solutions.

- In §8 we address the case where where $\mathcal{T}$ is the combination of two or more different theories. We present and discuss the main techniques for integration two or more $\mathcal{T}$*-solvers* into a lazy SMT tool.

- In §9 we present the related work. First, we present and discuss possible alternatives to the usage of DPLL in lazy SMT. Then we survey and discuss the main alternative approaches for SMT, including the rewrite-based approach, the eager approach, and some recent attempt of combining the lazy and eager approaches.

2. Theoretical background

In this section we recall some basic theoretical concepts, mostly from [SV98, GS00, Seb01, ABC$^+$02b, BBC$^+$06b], providing the theoretical background and terminology for this paper.

2.1 Background on first-order logic and theories

In order to make the paper self-contained, we recall some basic notions and terminology about first-order theories. We assume the usual syntactic notions of first-order logic with equality as defined, e.g., in [DJ90].

In the following, let Σ be a first-order signature containing function and predicate symbols with their arities, and $\mathcal{V}$ be a set of variables. A 0-ary function symbol c is called a *constant*. A 0-ary predicate symbol A is called a *boolean atom*. A *Σ-term* is either a variable in $\mathcal{V}$ or it is build by applying function symbols in Σ to Σ-terms. If $t_1, \ldots, t_n$ are Σ-terms and P is a predicate symbol, then $P(t_1, \ldots, t_n)$ is a *Σ-atom*. If l and r are two Σ-terms, then the Σ-atom $l = r$ is called a *Σ-equality* and $\neg(l = r)$ (also written as $l \neq r$) is called a *Σ-disequality*. A *Σ-formula* φ is built in the usual way out of the universal and existential quantifiers $\forall, \exists$, the boolean connectives $\wedge, \neg$, and Σ-atoms. We use the standard boolean abbreviations: "$\varphi_1 \vee \varphi_2$" for "$\neg(\neg\varphi_1 \wedge \neg\varphi_2)$", "$\varphi_1 \rightarrow \varphi_2$" for "$\neg(\varphi_1 \wedge \neg\varphi_2)$", "$\varphi_1 \leftarrow \varphi_2$" for "$\neg(\neg\varphi_1 \wedge \varphi_2)$", "$\varphi_1 \leftrightarrow \varphi_2$" for "$\neg(\varphi_1 \wedge \neg\varphi_2) \wedge \neg(\varphi_2 \wedge \neg\varphi_1)$", "$\top$" [resp. "$\bot$"] for the true [resp. false] constant. A *Σ-literal* is either a Σ-atom (a *positive literal*) or its negation (a *negative literal*). The set of Σ-atoms and Σ-literals occurring in φ are denoted by *$Atoms(\varphi)$* and *$Lits(\varphi)$* respectively. We call a Σ-formula *quantifier-free* if it does not contain quantifiers, and a *sentence* if it has no free variables. A quantifier-free formula is in *conjunctive normal form (CNF)* if it is written as a conjunction of disjunctions of literals. A disjunction of literals is called a *clause*.

Notationally we use the Greek letters φ, ψ to represent Σ-formulas, the capital letters A_i's and B_i's to represent boolean atoms, and the Greek letters α, β, γ to represent Σ-atoms in general, the letters l_i's to represent Σ-literals. If l is a negative Σ-literal $\neg\beta$, then by "$\neg l$" we conventionally mean β rather than $\neg\neg\beta$. We sometimes write a clause in the form of an implication: $\bigwedge_i l_i \rightarrow \bigvee_j l_j$ for $\bigvee_i \neg l_i \vee \bigvee_j l_j$ and $\bigwedge_i l_i \rightarrow \bot$ for $\bigvee_i \neg l_i$.

We also assume the usual first-order notions of interpretation, satisfiability, validity, logical consequence, and theory, as given, e.g., in [End72]. We write $\Gamma \models \varphi$ to denote that the formula φ is a logical consequence of the (possibly infinite) set Γ of formulae. A *Σ-theory* is a set of first-order sentences with signature Σ. All the theories we consider are first-order theories *with equality*, which means that the equality symbol $=$ is a predefined predicate and it is always interpreted as a relation which is reflexive, symmetric, transitive, and it is also a congruence. Since the equality symbol is a predefined predicate, it will not be included in any signature Σ considered in this paper. A Σ-structure $\mathcal{I}$ is a model of a Σ-theory $\mathcal{T}$ if $\mathcal{I}$ satisfies every sentence in $\mathcal{T}$. A Σ-formula is *satisfiable in $\mathcal{T}$* (or *$\mathcal{T}$-satisfiable*) if it is satisfiable in a model of $\mathcal{T}$. We write $\Gamma \models_{\mathcal{T}} \varphi$ to denote $\mathcal{T} \cup \Gamma \models \varphi$. Two Σ-formulas φ and ψ are *$\mathcal{T}$-equisatisfiable* iff φ is $\mathcal{T}$-satisfiable iff ψ is $\mathcal{T}$-satisfiable. We call *Satisfiability Modulo (the) Theory $\mathcal{T}$, $SMT(\mathcal{T})$*, the problem of establishing the $\mathcal{T}$-satisfiability of Σ-formulae, for some background theory $\mathcal{T}$. The $SMT(\mathcal{T})$ problem is NP-hard, since it subsumes the problem of checking the satisfiability of boolean formulae.

In this paper we restrict our attention to *quantifier-free* Σ-formulae on some Σ-theory $\mathcal{T}$. [5] We call a *theory solver* for $\mathcal{T}$ (*$\mathcal{T}$-solver*) any procedure establishing whether any given finite conjunction of quantifier-free Σ-literals (or equivalently, any given finite set of Σ-literals) is $\mathcal{T}$-satisfiable or not.

Henceforth, for simplicity and if not specified otherwise, we may omit the "Σ-" prefix from term, formula, theory, models, etc. Moreover, by "formulas", "atoms" and "literals" we implicitly refer to *quantifier-free* formulas, atoms and literals respectively.

2.1.1 Combination of theories

A theory $\mathcal{T}$ is *stably-infinite* iff for each $\mathcal{T}$-satisfiable formula φ, there exists a model of $\mathcal{T}$ whose domain is infinite and which satisfies φ. A conjunction Γ of literals is *convex* in a theory $\mathcal{T}$ iff for each disjunction $\bigvee_{i=1}^{n} x_i = y_i$ (where x_i, y_i are variables and $i = 1, ..., n$) we have that $\Gamma \models_{\mathcal{T}} \bigvee_{i=1}^{n} x_i = y_i$ iff $\Gamma \models_{\mathcal{T}} x_i = y_i$ for some $i \in \{1, ..., n\}$. A theory $\mathcal{T}$ is *convex* iff all the conjunctions of literals are convex in $\mathcal{T}$. Notice that any convex theory whose models are non-trivial (i.e., the domains of the models have all cardinality strictly greater than one) is stably-infinite.

In the sequel, let Σ_1 and Σ_2 be two disjoint signatures (i.e., $\Sigma_1 \cap \Sigma_2 = \emptyset$) and $\mathcal{T}_i$ will be a theory in Σ_i for $i = 1, 2$. We consider $\Sigma := \Sigma_1 \cup \Sigma_2$ and $\mathcal{T} := \mathcal{T}_1 \cup \mathcal{T}_2$. We call *$SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$* the problem of establishing the $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiability of $\Sigma_1 \cup \Sigma_2$-formulae. [6]

A $\Sigma_1 \cup \Sigma_2$-term t is an *i-term* iff either it is a variable or it has the form $f(t_1, ..., t_n)$, where f is in Σ_i. Notice that a variable is both a 1-term and a 2-term. A non-variable subterm s of an i-term t is *alien* if s is a j-term, and all superterms of s in t are i-terms, where $i, j \in \{1, 2\}$ and $i \neq j$. An i-term is *i-pure* if it does not contain alien subterms. An atom (or a literal) is *i-pure* if it contains only i-pure terms and its predicate symbol is either equality or in Σ_i. A $\Sigma_1 \cup \Sigma_2$-formula φ is said to be *pure* if every atom occurring in the formula is i-pure for some $i \in \{1, 2\}$. Intuitively, φ is pure if each atom can can be seen as belonging to one theory $\mathcal{T}_i$ only.

Every non-pure $\Sigma_1 \cup \Sigma_2$-formula φ can be converted into an $\mathcal{T}_1 \cup \mathcal{T}_2$-equisatisfiable one by recursively replacing each alien subterm t by a new variable v_t and adding the equality $v_t = t$ to φ. E.g.:

$$(f(x+3y) = g(2x-y)) \implies (f(v_{x+3y}) = g(v_{2x-y})) \wedge (v_{x+3y} = x+3y) \wedge (v_{2x-y} = 2x-y).$$

This process is called *purification*, and the size of the resulting formula is linear in that of φ. If φ is a pure $\Sigma_1 \cup \Sigma_2$-formula, then v is an *interface variable* for φ iff it occurs in both 1-pure and 2-pure atoms of φ. An equality $(v_i = v_j)$ is an *interface equality* for φ iff v_i, v_j are interface variables for φ. Henceforth we denote the interface equality $(v_i = v_j)$ by "e_{ij}".

[5] Notice that in $SMT(\mathcal{T})$, the variables are implicitly existentially quantified, and hence equivalent to Skolem constants.

[6] For simplicity in this paper we refer to combinations of two theories only, but all the discourse can be easily generalized to combination of many signature-disjoint theories $\mathcal{T}_1 \cup ... \cup \mathcal{T}_n$.

2.2 Truth assignments and propositional satisfiability in $\mathcal{T}$

We consider a generic quantifier-free decidable First-Order Theory $\mathcal{T}$ on a signature Σ. Notationally, we will often use the prefix "$\mathcal{T}$-" to denote "in the theory $\mathcal{T}$": e.g., we call a "$\mathcal{T}$-formula" a formula in (the signature of) $\mathcal{T}$, "$\mathcal{T}$-model" a model in $\mathcal{T}$, and so on.

We call a *truth assignment* μ for a $\mathcal{T}$-formula φ a truth value assignment to all the $\mathcal{T}$-atoms of φ. A truth assignment is *total* if it assigns a value to all atoms in φ, *partial* otherwise. Syntactically identical instances of the same $\mathcal{T}$-atom are always assigned identical truth values; syntactically different $\mathcal{T}$-atoms, e.g., $(t_1 \geq t_2)$ and $(t_2 \leq t_1)$, are treated differently and may thus be assigned different truth values.

To this extent, we introduce a bijective function $\mathcal{T2B}$ ("Theory-to-Boolean") and its inverse $\mathcal{B2T} := \mathcal{T2B}^{-1}$ ("Boolean-to-Theory"), s.t. $\mathcal{T2B}$ maps boolean atoms into themselves and non-boolean $\mathcal{T}$-atoms into fresh boolean atoms —so that two atom instances in φ are mapped into the same boolean atom iff they are syntactically identical— and distributes with sets and boolean connectives. $\mathcal{T2B}$ and $\mathcal{B2T}$ are also called *boolean abstraction* and *boolean refinement* respectively.

We represent a truth assignment μ for φ as a set of $\mathcal{T}$-literals

$$\{\alpha_1, \ldots, \alpha_N, \neg\beta_1, \ldots, \neg\beta_M, A_1, \ldots, A_R, \neg A_{R+1}, \ldots, \neg A_S\}, \tag{1}$$

α_i's, β_j's being Σ-atoms and A_i's being boolean propositions. Positive literals α_i, A_k mean that the corresponding atom is assigned to true, negative literals $\neg\beta_i$, $\neg A_k$ mean that the corresponding atom is assigned to false. If $\mu_2 \subseteq \mu_1$, then we say that μ_1 *extends* μ_2 and that μ_2 *subsumes* μ_1. Sometimes we represent a truth assignment (1) also as the formula given by the conjunction of its literals:

$$\alpha_1 \wedge \ldots \wedge \alpha_N \wedge \neg\beta_1 \wedge \ldots \wedge \neg\beta_M \wedge A_1 \wedge \ldots \wedge A_R \wedge \neg A_{R+1} \wedge \ldots \wedge \neg A_S. \tag{2}$$

Notationally, we use the Greek letters μ, η to represent truth assignments.

We say that a total truth assignment μ for φ *propositionally satisfies* φ, written $\mu \models_p \varphi$, if and only if $\mathcal{T2B}(\mu) \models \mathcal{T2B}(\varphi)$, that is, for all sub-formulas φ_1, φ_2 of φ:

$$\begin{array}{lcl} \mu \models_p \varphi_1,\ \varphi_1 \in Atoms(\varphi) & \Longleftrightarrow & \varphi_1 \in \mu, \\ \mu \models_p \neg\varphi_1 & \Longleftrightarrow & \mu \not\models_p \varphi_1, \\ \mu \models_p \varphi_1 \wedge \varphi_2 & \Longleftrightarrow & \mu \models_p \varphi_1 \ and\ \mu \models_p \varphi_2. \end{array}$$

We say that a partial truth assignment μ *propositionally satisfies* φ if and only if all the total truth assignments for φ which extend μ propositionally satisfy φ. (Henceforth, if not specified, when dealing with propositional $\mathcal{T}$-satisfiability we do not distinguish between total and partial assignments.)

Intuitively, if we consider a $\mathcal{T}$-formula φ as a propositional formulas in its atoms, then $\models_p$ is the standard satisfiability in propositional logic. Thus, for every φ_1 and φ_2, we say that $\varphi_1 \models_p \varphi_2$ if and only if $\mu \models_p \varphi_2$ for every μ s.t. $\mu \models_p \varphi_1$. We say that φ is *propositionally satisfiable* if and only if there exist an assignment μ s.t. $\mu \models_p \varphi$. We also say that $\models_p \varphi$ (φ is *propositionally valid*) if and only if $\mu \models_p \varphi$ for every assignment μ for φ. Thus $\varphi_1 \models_p \varphi_2$ if and only if $\models_p \varphi_1 \rightarrow \varphi_2$, and $\models_p \varphi$ iff $\neg\varphi$ is propositionally unsatisfiable.

Notice that $\models_p$ is stronger than $\models_{\mathcal{T}}$, that is, if $\varphi_1 \models_p \varphi_2$, then $\varphi_1 \models_{\mathcal{T}} \varphi_2$, but not vice versa. E.g., $(x_1 \leq x_2) \wedge (x_2 \leq x_3) \models_{\mathcal{LA}} (x_1 \leq x_3)$, but $(x_1 \leq x_2) \wedge (x_2 \leq x_3) \not\models_p (x_1 \leq x_3)$.

Example 2.1. *Consider the following $\mathcal{LA}(\mathbb{Q})$-formula φ and it boolean abstraction $\mathcal{T}2\mathcal{B}(\varphi)$:*

$$
\begin{array}{rlcrl}
\varphi := & \{\neg(2x_2 - x_3 > 2) \vee A_1\} & \qquad & \mathcal{T}2\mathcal{B}(\varphi) := & \{\neg B_1 \vee A_1\} \\
\wedge & \{\neg A_2 \vee (x_1 - x_5 \leq 1)\} & & \wedge & \{\neg A_2 \vee B_2\} \\
\wedge & \{(3x_1 - 2x_2 \leq 3) \vee A_2\} & & \wedge & \{B_3 \vee A_2\} \\
\wedge & \{\neg(2x_3 + x_4 \geq 5) \vee \neg(3x_1 - x_3 \leq 6) \vee \neg A_1\} & & \wedge & \{\neg B_4 \vee \neg B_5 \vee \neg A_1\} \\
\wedge & \{A_1 \vee (3x_1 - 2x_2 \leq 3)\} & & \wedge & \{A_1 \vee B_3\} \\
\wedge & \{(x_2 - x_4 \leq 6) \vee (x_5 = 5 - 3x_4) \vee \neg A_1\} & & \wedge & \{B_6 \vee B_7 \vee \neg A_1\} \\
\wedge & \{A_1 \vee (x_3 = 3x_5 + 4) \vee A_2\} & & \wedge & \{A_1 \vee B_8 \vee A_2\}
\end{array}
$$

We consider the partial truth assignment μ:

$$\{\neg(2x_2 - x_3 > 2), \neg A_2, (3x_1 - 2x_2 \leq 3), \neg(3x_1 - x_3 \leq 6), (x_2 - x_4 \leq 6), (x_3 = 3x_5 + 4)\},$$

which is the boolean refinement of the assignment $\mathcal{T}2\mathcal{B}(\mu) = \{\neg B_1, \neg A_2, B_3, \neg B_5, B_6, B_8\}$. (Notice that the two occurrences of $(3x_1 - 2x_2 \leq 3)$ in rows 3 and 5 of φ are both assigned true.) μ is a partial assignment which propositionally satisfies φ (i.e., $\mathcal{T}2\mathcal{B}(\mu) \models \mathcal{T}2\mathcal{B}(\varphi)$), as it assigns to true one literal of every disjunction in φ.

We say that a collection $\mathcal{M} := \{\mu_1, \ldots, \mu_n\}$ of (possibly partial) assignments propositionally satisfying φ is *complete* if and only if,

$$\models_p \varphi \leftrightarrow \bigvee_{\mu_j \in \mathcal{M}} \mu_j. \tag{3}$$

Property 2.2. *[SV98] A collection $\mathcal{M} := \{\mu_1, \ldots, \mu_n\}$ of partial assignments propositionally satisfying φ is complete if and only if, for every total assignment η s.t. $\eta \models_p \varphi$, there exists $\mu_j \in \mathcal{M}$ s.t. $\mu_j \subseteq \eta$.*

Proof.
[If] For every η s.t. $\eta \models_p \varphi$, $\mu_j \subseteq \eta$ for some $\mu_j \in \mathcal{M}$. Thus, $\eta \models_p \mu_j$, and hence $\eta \models_p \bigvee_{\mu_j \in \mathcal{M}} \mu_j$. Thus, $\models_p \varphi \rightarrow \bigvee_{\mu_j \in \mathcal{M}} \mu_j$. Vice versa, as $\mu_j \models_p \varphi$ for every $\mu_j \in \mathcal{M}$, then $\models_p \varphi \leftarrow \bigvee_{\mu_j \in \mathcal{M}} \mu_j$. Thus $\mathcal{M}$ is complete.
[Only if] Let $\mathcal{M}$ be complete. As $\eta \models_p \varphi$, then $\eta \models_p \bigvee_{\mu_j \in \mathcal{M}} \mu_j$. Hence $\eta \models_p \mu_j$ for some $\mu_j \in \mathcal{M}$, so that $\eta \supseteq \mu_j$. □

$\mathcal{M}$ can be seen thus as a compact representation of the whole set of total assignments propositionally satisfying φ.

Proposition 2.3. *[SV98] Let φ be a $\mathcal{T}$-formula and let $\mathcal{M} := \{\mu_1, \ldots, \mu_n\}$ be a complete collection of truth assignments propositionally satisfying φ. Then, φ is $\mathcal{T}$-satisfiable if and only if μ_j is $\mathcal{T}$-satisfiable for some $\mu_j \in \mathcal{M}$.*

Proof.
[If] $\mathcal{I} \models_{\mathcal{T}} \mu$ for some $\mathcal{T}$-model $\mathcal{I}$ and $\mu \models_p \varphi$, so that $\mu \models_{\mathcal{T}} \varphi$. Thus $\mathcal{I} \models_{\mathcal{T}} \varphi$.
[Only if] $\mathcal{I} \models_{\mathcal{T}} \varphi$ for some $\mathcal{T}$-model $\mathcal{I}$. Let

$$\mu_{\mathcal{I}} := \{\alpha \textit{ s.t. } \alpha \in \textit{Atoms}(\varphi) \textit{ and } \mathcal{I} \models_{\mathcal{T}} \alpha\} \cup \{\neg\beta \textit{ s.t. } \beta \in \textit{Atoms}(\varphi) \textit{ and } \mathcal{I} \models_{\mathcal{T}} \neg\beta\}.$$

By construction, $\mu_{\mathcal{I}}$ is a total truth assignment s.t. $\mu_{\mathcal{I}} \models_p \varphi$ and $\mu_{\mathcal{I}}$ is $\mathcal{T}$-consistent. By definition of $\mathcal{M}$, there exists a $\mu_j \in \mathcal{M}$ s.t. $\mu_j \subseteq \mu_{\mathcal{I}}$ and $\mu_j \models_p \varphi$. μ_j is $\mathcal{T}$-consistent because $\mu_{\mathcal{I}}$ is $\mathcal{T}$-consistent. □

Finally, we notice the following fact.

Proposition 2.4. *[Seb01] Let α be a non-boolean atom occurring only positively [resp. negatively] in φ. Let $\mathcal{M}$ be a complete set of assignments satisfying φ, and let*

$$\mathcal{M}' := \{\mu_j \setminus \{\neg\alpha\} | \ \mu_j \in \mathcal{M}\} \qquad [resp.\ \{\mu_j \setminus \{\alpha\} \mid \mu_j \in \mathcal{M}\}].$$

Then (i) for every $\mu'_j \in \mathcal{M}'$, $\mu'_j \models_p \varphi$, and (ii) φ is $\mathcal{T}$-satisfiable if and only if there exist a $\mathcal{T}$-satisfiable $\mu'_j \in \mathcal{M}'$.

Proof. (sketch) (i) As $\mu'_j = \mu_j \setminus \{\neg\alpha\}$ for some μ_j and $\mu_j \models_p \varphi$ and α occurs only positively in φ, it is easy to show by induction on φ that $\mu'_j \models_p \varphi$.
(ii) [If] $\mathcal{I} \models_{\mathcal{T}} \mu'_j$ for some $\mathcal{T}$-model $\mathcal{I}$ and $\mu'_j \models_p \varphi$ by (i), so that $\mu'_j \models_{\mathcal{T}} \varphi$. Thus $\mathcal{I} \models_{\mathcal{T}} \varphi$.
[Only if] If φ is $\mathcal{T}$-satisfiable, then there is a $\mathcal{T}$-satisfiable $\mu_j \in \mathcal{M}$ s.t. $\mu_j \models_p \varphi$ because $\mathcal{M}$ is complete. Let $\mu'_j := \mu_j \setminus \{\neg\alpha\}$. Then $\mu'_j \in \mathcal{M}'$ and μ'_j is $\mathcal{T}$-satisfiable because μ_j is $\mathcal{T}$-satisfiable. □

2.3 Enumerators and $\mathcal{T}$-solvers

By proposition 2.3, the problem of establishing the $\mathcal{T}$-satisfiability of φ can be decomposed into two orthogonal components: one *boolean component*, consisting in searching for (up to a complete set of) propositional models μ's propositionally satisfying φ, and one *theory-dependent component*, consisting in checking the $\mathcal{T}$-consistence of μ (that is, for the set of $\mathcal{T}$-literals in μ). This suggests that an *SMT*($\mathcal{T}$) solver can be seen as a combination of two basic ingredients: a *Truth Assignment Enumerator* and a *Theory Solver* for $\mathcal{T}$.

We call a *Truth Assignment Enumerator* (ENUMERATOR henceforth) a total function which takes as input a $\mathcal{T}$-formula φ and returns a complete collection $\mathcal{M} := \{\mu_1, \ldots, \mu_n\}$ of assignments propositionally satisfying φ.

As in §2.1, we call a *Theory Solver* for $\mathcal{T}$ (*$\mathcal{T}$-solver*) a procedure which takes as input a collection of $\mathcal{T}$-literals μ and decides whether μ is $\mathcal{T}$-satisfiable; optionally, it can return a $\mathcal{T}$-model satisfying μ, or *Null* if there is none. (It can return also some other information, which we will discuss in §4.1.)

Examples of calls to *$\mathcal{T}$-solver* for different theories $\mathcal{T}$ are: [7]

$\mathcal{DL}$: $\mathcal{DL}$-SOLVER($\{(x - y = 3), (y - z \le 4), \neg(x - z \le 8)\}$) returns Unsat;

$\mathcal{EUF}$: $\mathcal{EUF}$-SOLVER($\{a = b, b = f(c), \neg(g(a) = g(f(c)))\}$) returns Unsat;

$\mathcal{LA}(\mathbb{Q})$: $\mathcal{LA}(\mathbb{Q})$-SOLVER($\{(x - 2y = 3), (4y - 2z < 9), \neg(x - z \le 7)\}$) returns Sat;

$\mathcal{LA}(\mathbb{Z})$: $\mathcal{LA}(\mathbb{Z})$-SOLVER($\{(x - 2y = 3), (4y - 2z < 9), \neg(x - z \le 7)\}$) returns Unsat;

$\mathcal{BV}$: $\mathcal{BV}$-SOLVER({`(w[31:0]>>`$_{16}$`) != 0`$_{16}$`:w[31:16]` }) returns Unsat;

[7] These theories will be described in details in §4.2.

$\mathcal{AR}$: $\mathcal{AR}$-SOLVER({ ¬(`a1=a2`,), ¬(`read(M,a2)=read(write(M,a1,x),a2)`) }) returns Unsat.

Notice that $\mathcal{T}$ can be a combination of sub-theories, and hence $\mathcal{T}$*-solver* be a combined solver, as, e.g., in [NO79, Sho79, FORS01, BDS02b, SR02].

Remark 2.5. *For better readability, in all the examples of this paper we will use the theory of linear arithmetic on rational numbers ($\mathcal{LA}(\mathbb{Q})$) because of its intuitive semantics. Nevertheless, analogous examples can be built with all other theories of interest.*

```
SatValue DPLL (Bool_formula $\varphi$, assignment & $\mu$) {
    if (preprocess($\varphi,\mu$)==Conflict);
        return Unsat;
    while (1) {
        decide_next_branch($\varphi,\mu$);
        while (1) {
            status = deduce($\varphi,\mu$);
            if (status == Sat)
                return Sat;
            else if (status == Conflict) {
                blevel = analyze_conflict($\varphi,\mu$);
                if (blevel == 0)
                    return Unsat;
                else backtrack(blevel,$\varphi,\mu$);
            }
            else break;
} } }
```

Figure 1. Schema of a modern DPLL engine.

3. Basics on SAT solvers

A SAT solver is a procedure which decides whether an input boolean formula φ is satisfiable, and returns a satisfying assignment if this is the case. Notice the difference between a SAT solver and a truth assignment enumerator: the former has to find *only one* satisfying assignment —or to decide there is none— while the latter has to find a *complete collection* of satisfying assignments.

Most state-of-the-art SAT procedures are evolutions of the *Davis-Putnam-Longeman-Loveland (*DPLL*)* procedure [DP60, DLL62].

3.1 Modern DPLL

Unlike with "classic" representation of DPLL [DP60, DLL62], modern DPLL implementation are non-recursive, and are based on very efficient, destructive data structures to handle boolean formulas and assignments. They benefit of sophisticated search techniques, smart decision heuristics, highly-engineered data structures and cute implementation tricks, and smart preprocessing techniques. (We refer the reader to [ZM02] for an overview.) A high-level schema of a modern DPLL engine, adapted from [ZM02], is reported in Figure 1. The boolean formula φ is in CNF; the assignment μ is initially empty, and it is updated in a stack-based manner.

`preprocess`(φ,μ) simplifies φ into a simpler and equi-satisfiable formula, and updates μ if it is the case. [8] If the resulting formula is unsatisfiable, then DPLL returns Unsat.

[8] More precisely, if φ, μ, φ', μ' are the formula and the assignment before and after preprocessing respectively, then $\varphi' \wedge \mu'$ is equisatisfiable to $\varphi \wedge \mu$.

In the main loop, `decide_next_branch`(φ, μ) chooses an unassigned literal l from φ according to some heuristic criterion, and adds it to μ. (This operation is called *decision*, l is called *decision literal* end the number of decision literals in μ after this operation is called the *decision level* of l.)

In the inner loop, `deduce`(φ, μ) iteratively deduces literals l deriving from the current assignments (i.e., $\varphi \wedge \mu \models_p l$) and updates φ and μ accordingly; this step is repeated until either μ satisfies φ, or μ falsifies φ, or no more literals can be deduced, returning Sat, Conflict and Unknown respectively. (The iterative application of boolean deduction steps in `deduce` is also called *Boolean Constraint Propagation, BCP.*)

In the first case, DPLL returns Sat. If the second case, `analyze_conflict`(φ, μ) detects the subset η of μ which caused the conflict (*conflict set*) and the decision level `blevel` to backtrack. If `blevel==0`, then a conflict exists even without branching, so that DPLL returns Unsat. Otherwise, `backtrack(blevel,`φ, μ`)` adds $\neg\eta$ to φ (*learning*) and backtracks up to `blevel` (*backjumping*), updating φ and μ accordingly. In the third case, DPLL exits the inner loop, looking for the next decision.

We look at these steps with some more detail.

`preprocess` implements simplification techniques like, e.g., detecting and inlining boolean equivalences among literals, applying resolutions steps to selected pairs of clauses, detecting and dropping subsumed clauses (see, e.g., [Bra01, BW03, EB05]). It may also apply BCP if this is the case.

`decide_next_branch` implements the key non-deterministic step in DPLL, for which many heuristic criteria have been conceived. Old-style heuristics like MOMS and Jeroslow-Wang [JW90] used to select a new literal at each branching point, picking the literal occurring most often in the minimal-size clauses (see, e.g., [HV95]). The heuristic implemented in SATZ [LA97] selects a candidate set of literals, perform BCP, chooses the one leading to the smallest clause set; this maximizes the effects of BCP, but introduces big overheads. When formulas derive from the encoding of some specific problem, it is sometimes useful to allow the encoder to provide to the DPLL solver a list of "privileged" variables on which to branch on first (e.g., action variables in SAT-based planning [GMS98], primary inputs in bounded model checking [Str00]). Modern DPLL solvers adopt evolutions of the VSIDS heuristic [MMZ$^+$01, GN02, ES04], in which decision literals are selected according to a score which is updated only at the end of a branch, and which privileges variables occurring in recently-learned clauses; this makes `decide_next_branch` state-independent (and thus much faster, because there is no need to recomputing the scores at each decision) and allows it to take into account search history, which makes search more effective and robust.

`deduce` is mostly based on the iterative application of unit propagation. Highly-engineered data structures and cute implementation tricks (like the *two-watched-literal scheme* [MMZ$^+$01]) allow for extremely efficient implementations. Other forms of deductions (and formula simplification) are, e.g., *pure literal rule* (now obsolete), on-line equivalence reasoning [Li00], and variable and clause elimination [EB05].

It is important to notice that most modern implementations DPLL solver do not return Sat when all clauses are satisfied, but only when all variables are assigned truth values. [9.] As

[9.] This is mostly due to the fact that the two-watched-literal scheme [MMZ$^+$01] does not allow for an easy check of clause satisfaction. (E.g., if a non-watched literal l in the clause $C \vee l$ is true, then the clause it satisfied but DPLL is not informed of this fact.)

a consequence, modern SAT solvers typically return *total* truth assignments, even though the formulas are satisfied by *partial* ones. (We will further discuss this issue in §7.2.1.)

`analyze_conflict` and `backtrack` work as follows [SS96, BS97, ZMMM01]. Each non-decision literal l in μ is tagged by a link to the clause C_l causing its unit-propagation (called the *antecedent clause* of l). When a conflict occurs on a clause C (called the *conflicting clause*), a conflict clause is computed starting from C by iteratively resolving the current clause C with the antecedent clause C_l of one non-decision literal l occurring in C. Depending on the strategy adopted, this process may terminate, e.g., as soon as C contains no non-decision literal of the current decision level (the *last UIP strategy*) or at most one non-decision literal of the current decision level (*the 1st UIP strategy*).

Graphically, building a conflict set/clause corresponds to (implicitly) building and analyzing the *implication graph* corresponding to the current assignment. An implication graph is a DAG s.t. each node represents a variable assignment (literal), the node of a decision literal has no incoming edges, all edges incoming into a non-decision-literal node l are labeled with the antecedent clause s.t. C_l, s.t. $l_1 \overset{C_l}{\longmapsto} l, \ldots, l_n \overset{C_l}{\longmapsto} l$ if and only if $C_l = \neg l_1 \vee \ldots \vee \neg l_n \vee l$. When both l and $\neg l$ occur in the implication graph, we have a *conflict*, and a partition of the graph with all decision literals on one side and the conflict on the other represents a conflict set. A node l in an implication graph is an *unique implication point (UIP)* for the last decision level iff any path from the last decision node to both the conflict nodes passes through l; the most recent decision node is an UIP (*last UIP*); the most-recently-assigned UIP is called the 1st UIP. E.g., the last [resp. 1st] UIP strategy corresponds to using as conflict set the partition corresponding to the last [resp. 1st] UIP and decision literals.

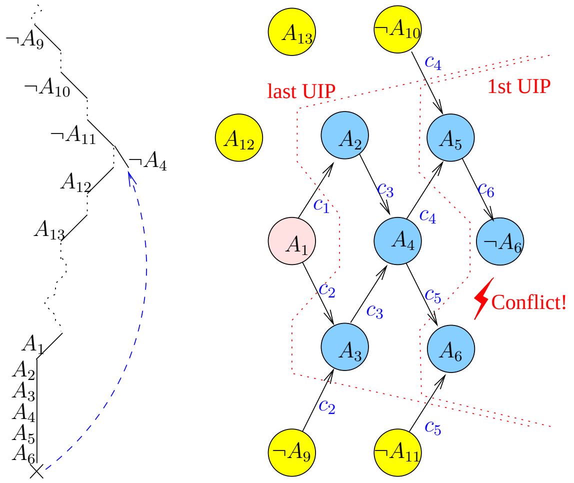
After `analyze_conflict` has computed the conflict clause C and added it to the formula, `backtrack` pops all assigned literals out of μ up to a decision level `blevel` deriving from C, which is computed by `analyze_conflict` according to different strategies. In the most modern implementations, DPLL backtracks to the highest point in the stack where one literal l in the learned clause $\neg\mu'$ is not assigned, and unit propagates l. We refer the reader to [ZMMM01] for an overview on backjumping and learning strategies.

Example 3.1. *Consider a boolean formula containing the clauses $c_1 \ldots c_9$ in Figure 2, and assume at some point $\mu := \{\ldots, \neg A_9, \ldots, \neg A_{10}, \ldots, \neg A_{11}, \ldots, A_{12}, \ldots, A_{13}, \ldots, A_1\}$. After applying BCP on $c_1 \ldots c_8$ a conflict on c_6 occurs. Starting from the conflicting clause c_6, the conflict clause/set is computed by iteratively resolving the current clause C with the antecedent clause of one non-decision literal l in C, until it contains at most one non-decision literal assigned at the current decision level (1st UIP):*

$$\dfrac{\overbrace{\neg A_4 \vee A_5 \vee A_{10}}^{c_4} \qquad \dfrac{\overbrace{\neg A_4 \vee A_6 \vee A_{11}}^{c_5} \qquad \overbrace{\neg A_5 \vee \neg A_6}^{conflicting\ clause}}{\neg A_4 \vee \neg A_5 \vee A_{11}}\ (A_6)}{\underbrace{\neg A_4}_{1st\ UIP} \vee \underbrace{A_{10} \vee A_{11}}_{decision\ lits}}\ (A_5)$$

This corresponds to the 1st UIP cut of the implication graph in Figure 2. Then DPLL *learns the conflict clause $c_{10} := A_{10} \vee A_{11} \vee \neg A_4$, and backtracks up to below $\neg A_{11}$, it unit-propagates $\neg A_4$ on c_{10}, and proceeds.*

$c_1 : \{\neg A_1 \vee A_2\}$ $\surd$
$c_2 : \{\neg A_1 \vee A_3 \vee A_9\}$ $\surd$
$c_3 : \{\neg A_2 \vee \neg A_3 \vee A_4\}$ $\surd$
$c_4 : \{\neg A_4 \vee A_5 \vee A_{10}\}$ $\surd$
$c_5 : \{\neg A_4 \vee A_6 \vee A_{11}\}$ $\surd$
$c_6 : \{\neg A_5 \vee \neg A_6\}$ $\times$
$c_7 : \{A_1 \vee A_7 \vee \neg A_{12}\}$ $\surd$
$c_8 : \{A_1 \vee A_8\}$ $\surd$
$c_9 : \{\neg A_7 \vee \neg A_8 \vee \neg A_{13}\}$
...



$\Longrightarrow$Conflict set: $\{\neg A_{10}, \neg A_{11}, A_4\}$, learn $c_{10} := A_{10} \vee A_{11} \vee \neg A_4$

Figure 2. Example of learning and backjumping based on the 1st UIP stategy.

Learning must be used with some care, because it may cause an explosion in the size of φ. To avoid this problem, modern DPLL tools implement techniques for *discharging* learned clauses when necessary [SS96, BS97]. Moreover, in order to avoid getting stuck into hard portions of the search space, most DPLL tools *restart* the search from scratch in a controlled manner [GSK98]; the clauses which have been learned avoid exploring the same search tree again. Clause discharging and restarts are substantially orthogonal to our discussion on SMT and come for free by using state-of-the-art DPLL solvers, so that they will not be discussed any further.

Modern DPLL procedures can be used as truth assignment enumerators, by modifying rows 8-9 in Figure 1 so that, when a satisfying assignment μ, is generated, DPLL stores μ and backtracks. This issue will be discussed in §5.3.

3.2 The Abstract-DPLL logical framework

[Tin02, NOT05, NO05a, NOT06] proposed an abstract rule-based formulation of DPLL (*Abstract* DPLL) and of DPLL-based lazy SMT systems (*Abstract* DPLL *Modulo Theories, or DPLL($\mathcal{T}$)*), which are represented as control strategies applied to a set of formal rules. This allows for expressing and reasoning about most variants of these procedures in a formal way.

In the Abstract-DPLL framework, DPLL is modeled as a transition system. A state is either $fail$ or a pair $\langle \mu \mid \varphi \rangle$, φ being a CNF boolean formula and μ being a set of annotated literals, representing the current truth assignment. All DPLL steps are seen as transitions in the form $\langle \mu \mid \varphi \rangle \Rightarrow \langle \mu' \mid \varphi' \rangle$, and are applications of the conditioned transition rules described in Figure 3. [10.]

[10.] The formalization of the rules in [NOT05, NO05a, NOT06] changes slightly from paper to paper. Here we report the most-recent one from [NOT06]. We have adapted the notation to that used in this paper.

$$
\begin{array}{llll}
\text{Unit Propagate:} & \langle \mu \mid \varphi, C \vee l \rangle & \Rightarrow \langle \mu, l \mid \varphi, C \vee l \rangle & if \begin{cases} \mu \models \neg C \\ l \ is \ undefined \ in \ \mu \end{cases} \\
\text{Decide:} & \langle \mu \mid \varphi \rangle & \Rightarrow \langle \mu, l^d \mid \varphi \rangle & if \begin{cases} l^d \ or \ \neg l^d \ occurs \ in \ \varphi \\ l^d \ is \ undefined \ in \ \mu \end{cases} \\
\text{Fail:} & \langle \mu \mid \varphi, C \rangle & \Rightarrow fail & if \begin{cases} \mu \models \neg C \\ \mu \ contains \ no \ decision \ literals \end{cases} \\
\text{Backjump:} & \langle \mu, l^d, \mu' \mid \varphi, C \rangle & \Rightarrow \langle \mu, l' \mid \varphi, C \rangle & if \begin{cases} \mu, l^d, \mu' \models \neg C \\ there \ is \ some \ clause \ C' \vee \ l' \ s.t.: \\ \quad \varphi, C \models C' \vee l' \ and \ \mu \models \neg C' \\ \quad l' \ is \ undefined \ in \ \mu \\ \quad l' \ or \ \neg l' \ occurs \ in \ \varphi \ or \\ \quad\ in \ \mu \cup \{l^d\} \cup \mu' \end{cases} \\
\text{Learn:} & \langle \mu \mid \varphi \rangle & \Rightarrow \langle \mu \mid \varphi, C \rangle & if \begin{cases} all \ atoms \ in \ C \ occur \ in \ \varphi \ or \ in \ \mu \\ \varphi \models C \end{cases} \\
\text{Discharge:} & \langle \mu \mid \varphi, C \rangle & \Rightarrow \langle \mu \mid \varphi \rangle & if \ \{ \ \varphi \models C \\
\text{Restart:} & \langle \mu \mid \varphi \rangle & \Rightarrow \langle \emptyset \mid \varphi \rangle &
\end{array}
$$

Figure 3. The Abstract-DPLL logical framework from [NOT06]. l^d denotes a decision literal. In the Backjump rule, C and $C' \vee l'$ represent the conflicting and the conflict clause respectively.

The first five rules represent respectively the unit-propagation step of `deduce`, the literal selection in `decide_next_branch`, the failure step of row 12-13 in Figure 1, the backjumping and learning mechanisms of `analyze_conflict` and `backtrack`. The last two rules represent the discharging and restart mechanisms described, e.g., in [BS97] and [GSK98].

The only non-obvious rule is Backjump, which deserves some more explanation: if a branch $\mu \cup \{l^d\} \cup \mu'$ falsifies one clause C (the conflicting clause), and a conflict clause $C' \vee l'$ [11] can be computed from C s.t. the corresponding conflict set $\neg(C' \vee l')$ falsifies $\varphi \wedge C$, $\neg C' \subseteq \mu$, $l' \notin \mu$, and l' or $\neg l'$ occur in φ or in $\mu \cup \{l^d\} \cup \mu'$, then it is possible to backjump up to μ, and hence unit-propagate l' on the conflict clause $(C' \vee l')$.

Example 3.2. *Consider the problem in Example 3.1 and Figure 2. The execution can be represented in Abstract-*DPLL *as follows:*

$$
\begin{array}{lll}
... & & \\
\langle .., \neg A_9, .., \neg A_{10}, .., \neg A_{11}, .., A_{12}, .., A_{13}, .. & | c_1, ..., c_9 \rangle & \Rightarrow (Decide \ A_1) \\
\langle .., \neg A_9, .., \neg A_{10}, .., \neg A_{11}, .., A_{12}, .., A_{13}, .., A_1 & | c_1, ..., c_9 \rangle & \Rightarrow (UnitP. \ A_2) \\
\langle .., \neg A_9, .., \neg A_{10}, .., \neg A_{11}, .., A_{12}, .., A_{13}, .., A_1, A_2 & | c_1, ..., c_9 \rangle & \Rightarrow (UnitP. \ A_3) \\
... & & \\
\langle .., \neg A_9, .., \neg A_{10}, .., \neg A_{11}, .., A_{12}, .., A_{13}, .., A_1, A_2, A_3, A_4, A_5, A_6 & | c_1, ..., c_9 \rangle & \Rightarrow (Learn \ c_{10}) \\
\langle .., \neg A_9, .., \neg A_{10}, .., \neg A_{11}, .., A_{12}, .., A_{13}, .., A_1, A_2, A_3, A_4, A_5, A_6 & | c_1, ..., c_9, c_{10} \rangle & \Rightarrow (Backjump) \\
\langle .., \neg A_9, .., \neg A_{10}, .., \neg A_{11}, \neg A_1 & | c_1, ..., c_9, c_{10} \rangle & \Rightarrow (...) \\
... & &
\end{array}
$$

[11] Also called the *backjump clause* in [NOT06].

$c_1, ..., c_{10}$ *being the clauses in Figure 2.*

If a finite sequence $\langle \emptyset \mid \varphi \rangle \Rightarrow \langle \mu_1 \mid \varphi_1 \rangle \Rightarrow \ldots \Rightarrow fail$ is found, then the formula is unsatisfiable; if a finite sequence $\langle \emptyset \mid \varphi \rangle \Rightarrow \ldots \Rightarrow \langle \mu_n \mid \varphi_n \rangle$ is found so that no rule can be further applied, then the formula is satisfiable. Different strategies in applying the rules correspond to different variants of the algorithm. [NOT05, NOT06] provides a group of results about termination, correctness and completeness of various configurations. Importantly, notice that only the first four rules are strictly necessary for correctness and completeness [NOT05]. We refer the reader to [NOT05, NO05a, NOT06] for further details. The Abstract-DPLL Modulo Theories/DPLL($\mathcal{T}$) framework will be described in §5.4.

4. Basics on theory solvers

As stated in §2, in its simplest form a $\mathcal{T}$-*solver* is a procedure establishing whether any given finite set/conjunction of quantifier-free Σ-literals is $\mathcal{T}$-satisfiable or not. Starting from the pioneering works by Nelson, Oppen and Shostak [NO79, NO80, Sho79, Sho84], many algorithms have been conceived for $\mathcal{T}$-*solvers* in many theories of interest. In this section we discuss the features of $\mathcal{T}$-*solvers* which are most important for $SMT(\mathcal{T})$, and briefly survey the most interesting theories and the relative $\mathcal{T}$-*solvers*.

As it will be made clear in the next sections, two main features of a $\mathcal{T}$-*solver* concur in achieving the maximum efficiency of a $SMT(\mathcal{T})$ solver: the *effectiveness* of its synergical interaction with the DPLL solver, and its *efficiency* in time and memory. The effectiveness of the interaction depends on the capability of the $\mathcal{T}$-*solver* of producing, exchanging and exploiting fruitful information with DPLL, and will be discussed in §4.1. The efficiency in time and memory of $\mathcal{T}$-*solver* strongly depends on the theory $\mathcal{T}$. (E.g., the problem of deciding the $\mathcal{T}$-satisfiability of sets of literals is $O(n \cdot log(n))$ for $\mathcal{EUF}$ and NP-complete for $\mathcal{LA}(\mathbb{Z})$.) This will be briefly discussed in §4.2 for some theories of interest.

4.1 Important features of $\mathcal{T}$-*solvers*

In this section we overview the main capabilities of a general $\mathcal{T}$-*solver* which are of interest for their usage within an SMT procedure.

4.1.1 Model generation

A key issue for $\mathcal{T}$-*solver*, whenever it is invoked on an $\mathcal{T}$-consistent assignment μ, is its ability to produce a $\mathcal{T}$-model $\mathcal{I}$ for μ witnessing its consistency, i.e., $\mathcal{I} \models_{\mathcal{T}} \mu$. Substantially most $\mathcal{T}$-*solvers* for all theories of interest are able to produce a model on demand.

Example 4.1. *Let μ be $\{\neg(2v_2 - v_3 > 2), (3v_1 - 2v_2 \leq 3), (v_3 = 3v_5 + 4)\}$. A $\mathcal{LA}(\mathbb{Q})$-solver decides that μ is $\mathcal{LA}(\mathbb{Q})$-satisfiable, and may return $\mathcal{I} := \{v_1 = v_2 = v_3 = 0, v_5 = -4/3\}$.*

Notice that sometimes producing a model may require extra computation effort. This is mostly due to the fact that the $\mathcal{T}$-*solvers* may perform satisfiability-preserving transformations on the input literals, which must be reversed when building the model.

4.1.2 Conflict set generation

Given a $\mathcal{T}$-unsatisfiable assignment μ, we call a *theory conflict set* (simply "conflict set" when this causes no ambiguity) a $\mathcal{T}$-unsatisfiable sub-assignment $\mu' \subseteq \mu$; we say that μ' is a *minimal theory conflict set* if all strict subsets of μ' are $\mathcal{T}$-consistent. [12.] (E.g., in Example 2.1, (4) is a minimal conflict set for μ.) A key efficiency issue for $\mathcal{T}$-*solver*, whenever it is invoked on an $\mathcal{T}$-inconsistent assignment μ, is its ability to produce the (possibly minimal) conflict set of μ which has caused its inconsistency.

Example 4.2. *Consider the $\mathcal{LA}(\mathbb{Q})$-formula φ in Example 2.1, and suppose $\mathcal{LA}(\mathbb{Q})$-solver is called on μ. As μ is not $\mathcal{LA}(\mathbb{Q})$-satisfiable, $\mathcal{LA}(\mathbb{Q})$-solver will return* Unsat*, and may also*

[12.] Theory conflict sets are also called *reasons*, *proofs* or *infeasible sets* by some authors; minimal theory conflict sets are also called *irreducible infeasible sets* in [YM06].

return a (minimal) conflict set causing the conflict:

$$\{(3x_1 - 2x_2 \le 3), \neg(2x_2 - x_3 > 2), \neg(3x_1 - x_3 \le 6)\}. \tag{4}$$

For instance, there exist conflict-set-producing variants for the Bellman-Ford algorithm for $\mathcal{DL}$, [CG99], for the Simplex LP procedures for $\mathcal{LA}(\mathbb{Q})$ [BB01] and for the congruence closure algorithm for $\mathcal{EUF}$ [NO03]. (See §4.2.)

4.1.3 Incrementality and Backtrackability

It is often the case that $\mathcal{T}$*-solver* is invoked sequentially on *incremental* assignments, in a stack-based manner, like in the following trace (left column first, then right) [BBC$^+$05a]:

$$\begin{array}{llll}
\mathcal{T}\textit{-solver}\ (\mu_1) & \Longrightarrow \mathsf{Sat} & \text{Undo } \mu_4,\ \mu_3,\ \mu_2 & \\
\mathcal{T}\textit{-solver}\ (\mu_1 \cup \mu_2) & \Longrightarrow \mathsf{Sat} & \mathcal{T}\textit{-solver}\ (\mu_1 \cup \mu_2') & \Longrightarrow \mathsf{Sat} \\
\mathcal{T}\textit{-solver}\ (\mu_1 \cup \mu_2 \cup \mu_3) & \Longrightarrow \mathsf{Sat} & \mathcal{T}\textit{-solver}\ (\mu_1 \cup \mu_2' \cup \mu_3') & \Longrightarrow \mathsf{Sat} \\
\mathcal{T}\textit{-solver}\ (\mu_1 \cup \mu_2 \cup \mu_3 \cup \mu_4) & \Longrightarrow \mathsf{Unsat} & \ldots &
\end{array}$$

Thus, a key efficiency issue of $\mathcal{T}$*-solver* is that of being *incremental* and *backtrackable.* [13] *Incremental* means that $\mathcal{T}$*-solver* "remembers" its computation status from one call to the other, so that, whenever it is given in input an assignment $\mu_1 \cup \mu_2$ such that μ_1 has just been proved $\mathcal{T}$-satisfiable, it avoids restarting the computation from scratch by restarting the computation from the previous status. *Backtrackable* means that it is possible to undo steps and return to a previous status on the stack in an efficient manner.

For instance, there are incremental and backtrackable versions of the congruence closure algorithm for $\mathcal{EUF}$ [NO03], of the Bellman-Ford algorithm for $\mathcal{DL}$ [CG99, NO05a], and of the Simplex LP procedure for $\mathcal{LA}(\mathbb{Q})$ [BB01, DdM06]. (See §4.2.)

4.1.4 Deduction of unassigned literals

For many theories it is possible to implement $\mathcal{T}$*-solver* so that, when returning Sat, 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. We say that $\mathcal{T}$*-solver* is *deduction-complete* if it can perform all possible such deductions, or say that no such deduction can be performed.

Example 4.3. *Consider the $\mathcal{LA}(\mathbb{Q})$-formula φ in Example 2.1, and suppose $\mathcal{LA}(\mathbb{Q})$-solver is called on $\{\neg(2x_2 - x_3 > 2), \neg(3x_1 - x_3 \le 6), (x_3 = 3x_5 + 4)\}$ (1st, 4th and 7th rows in φ); then $\mathcal{LA}(\mathbb{Q})$-solver returns* Sat *and may perform and return the deduction*

$$\{\neg(2x_2 - x_3 > 2), \neg(3x_1 - x_3 \le 6)\} \models_{\mathcal{LA}(\mathbb{Q})} \neg(3x_1 - 2x_2 \le 3). \tag{5}$$

For instance, for $\mathcal{EUF}$, the computation of congruence closure allows for efficiently deducing positive equalities [NO03]; for $\mathcal{DL}$, a very efficient implementation of a deduction-complete $\mathcal{T}$*-solver* has been presented by [NO05a, CM06a]; for $\mathcal{LA}$ the task is much harder, and only $\mathcal{T}$-solvers capable of incomplete forms of deduction have been presented [DdM06]. (See §4.2.)

Notice that, in principle, every $\mathcal{T}$*-solver* has deduction capabilities, as it is always possible to call $\mathcal{T}$*-solver*$(\mu \cup \{\neg l\})$ for every unassigned literal l. We call this technique, *plunging* [DNS05]. In practice, apart from some application [ACG99], plunging is very inefficient.

[13] The latter feature is also called *resettable* in other contests (e.g., in [NO79]).

4.1.5 Deduction of interface equalities

Similarly to deduction of unassigned literals, for most theories it is possible to implement $\mathcal{T}$*-solver* so that , when returning Sat, it can also perform a set of deductions in the form $\mu \models_{\mathcal{T}} e$ (if $\mathcal{T}$ is convex) or in the form $\mu \models_{\mathcal{T}} \bigvee_j e_j$ (if $\mathcal{T}$is not convex) s.t. e, $e_1, ..., e_n$ are equalities between variables occurring in μ. In accordance with the notation in §2.1.1, and because typically e, $e_1, ..., e_n$ are interface equalities, we call these forms of deductions *e_{ij}-deductions*, and we say that a $\mathcal{T}$*-solver* is *e_{ij}-deduction-complete* if it can perform all possible such deductions, or say that no such deduction can be performed.

Example 4.4. *If an e_{ij}-deduction complete $\mathcal{LA}(\mathbb{Z})$-solver is given as input a consistent assignment $\{\ldots, (v_1 \geq 0), (v_1 \leq 1), (v_3 = 0), (v_4 = 1), \ldots\}$, then it will deduce from it the disjunction of equalities $(v_1 = v_3) \vee (v_1 = v_4)$.*

Notice that, unlike with the deduction of unassigned literal described in §4.1.4, here the deduced equalities need not occur in the input formula φ.

e_{ij}-deduction is often (implicitly) implemented by means of *canonizers* [Sho84]. Intuitively, a *canonizer* $\mathtt{canon}_{\mathcal{T}}$ for a theory $\mathcal{T}$ is a function which maps a term t into another term $\mathtt{canon}_{\mathcal{T}}(t)$ in *canonical* form, that is, $\mathtt{canon}_{\mathcal{T}}$ maps terms which are semantically equivalent in $\mathcal{T}$ into the same term. Thus, if x_{t_1}, x_{t_2} are interface variables labeling the terms t_1 and t_2 respectively, then the interface equality $(x_{t_1} = x_{t_2})$ can be deduced in $\mathcal{T}$ if and only if $\mathtt{canon}_{\mathcal{T}}(t_1)$ and $\mathtt{canon}_{\mathcal{T}}(t_2)$ are syntactically identical.

4.2 Some relevant theories and $\mathcal{T}$*-solvers*

We briefly overview some of the theories of interest, with some information about the relative $\mathcal{T}$*-solvers*. (See also [MZ03].)

As stated in §2, all the theories we consider are first-order theories with equality, in which "=" is a predefined predicate and it is always interpreted as a relation which is reflexive, symmetric, transitive, and it is also a congruence. Thus, the following equality (6) and congruence (7) axioms are implicit in all theories, for every function symbol f and predicate symbol P:

$$\forall x.\ (x = x), \qquad \forall x, y.\ (x = y \to y = x), \qquad \forall x, y, z.\ ((x = y \wedge y = z) \to x = z) \tag{6}$$

$$\begin{array}{l} \forall x_1, ..., x_n, y_1, ..., y_n.\ ((\bigwedge_{i=1}^{n} x_i = y_i) \to f(x_1, ..., x_n) = f(y_1, ..., y_n)) \\ \forall x_1, ..., x_n, y_1, ..., y_n.\ ((\bigwedge_{i=1}^{n} x_i = y_i) \to (P(x_1, ..., x_n) \leftrightarrow P(y_1, ..., y_n))). \end{array} \tag{7}$$

For simplicity and w.l.o.g., we can assume that identities like $t = t$ are simplified into $\top$ and atoms like $t_1 = t_2$ have been sorted s.t. $t_1 \prec t_2$ for some total order $\prec$. Thus the first two axioms in (6) are useless.

4.2.1 Equality and Uninterpreted Functions

The *theory of Equality and Uninterpreted Functions ($\mathcal{EUF}$)* [14.] is the quantifier-free F.O. theory with equality with no restrictions on Σ. Semantically, no axioms other than (6) and (7) are provided. If Σ contains no uninterpreted functions or predicates, then (7) are

[14.] Simply called "the theory of equality" by some authors (e.g., [MZ03]).

not needed, and we denote the resulting restricted theory by $\mathcal{E}$. $\mathcal{EUF}$ is stably-infinite and convex. The $\mathcal{EUF}$-satisfiability of sets of quantifier-free literals is decidable and polynomial [Ack54].

An $\mathcal{E}$-solver can be simply implemented on top of the standard Union-Find algorithm (see, e.g., [NO05b]). Efficient $\mathcal{EUF}$-solvers have been implemented on top of congruence-closure data structures, they are incremental and backtrackable, can efficiently perform conflict-set generation and deduction of (positive) unassigned equalities and of interface equalities (see, e.g., [DNS05, NO03, NO05b]). The algorithm in [NO03] extends $\mathcal{EUF}$ with offset values (that is, it can represent expression like $(t1 = t2 + k)$, $t1$, t_2 being $\mathcal{EUF}$ terms and k being a constant integer value) and it is probably the most efficient algorithm currently available.

4.2.2 Linear arithmetic

The *theory of Linear Arithmetic ($\mathcal{LA}$)* on the rationals ($\mathcal{LA}(\mathbb{Q})$) and on the integers ($\mathcal{LA}(\mathbb{Z})$) is the quantifier-free F.O. theory with equality whose atoms are written in the form $(a_1 \cdot x_1 + ... + a_n \cdot x_n \bowtie a_0)$, s.t. $\bowtie \in \{\leq, <, \neq, =, \geq, >\}$, the a_is are (interpreted) constants, each labeling one value in $\mathbb{Q}$ and $\mathbb{Z}$ respectively. The atomic expressions are interpreted according to the standard semantics of linear arithmetic on $\mathbb{Q}$ and $\mathbb{Z}$ respectively. (See, e.g., [MZ03] for a more formal definition of $\mathcal{LA}(\mathbb{Q})$ and $\mathcal{LA}(\mathbb{Z})$.)

$\mathcal{LA}(\mathbb{Q})$ is stably-infinite and convex. The $\mathcal{LA}(\mathbb{Q})$-satisfiability of sets of quantifier-free literals is decidable and polynomial [Kha79]. The main algorithms used are variant of the well-known Simplex and Fourier-Motzkin algorithms. Efficient incremental and backtrackable algorithms for $\mathcal{LA}(\mathbb{Q})$-solvers have been conceived, which can efficiently perform conflict-set generation and deduction of unassigned equalities and of interface equalities (see, e.g., [BMSX97, dRS04, DNS05, RS04, DdM06]).

$\mathcal{LA}(\mathbb{Z})$ is stably-infinite and non-convex. [15.] The $\mathcal{LA}(\mathbb{Z})$-satisfiability of sets of quantifier-free literals is decidable and NP-complete [Pap81]. Many algorithms have been conceived, involving techniques like Euler's reduction, Gomory-cuts application, Fourier-Motzkin algorithm, branch-and-bound [LD60]. Notably, the OMEGA library [Ome] provides a $\mathcal{LA}(\mathbb{Z})$-solver based on a combination of Euler's reduction, Fourier-Motzkin algorithm, and smart optimizations like real and dark shadow [Pug91]. Incremental and backtrackable algorithms for $\mathcal{LA}(\mathbb{Z})$-solvers have been conceived, which can perform conflict-set generation and deduction of interface equalities (see, e.g., [BGD03, DdM06]).

There are two main relevant sub-theories of $\mathcal{LA}$: the theory of *differences* and the *Unit-Two-Variable-Per-Inequality* theory.

4.2.3 Difference logic

The *theory of differences ($\mathcal{DL}$)* [16.] on the rationals ($\mathcal{DL}(\mathbb{Q})$) and the integers ($\mathcal{DL}(\mathbb{Z})$) is the sub-theory of $\mathcal{LA}(\mathbb{Q})$ [resp. $\mathcal{LA}(\mathbb{Z})$] whose atoms are written in the form $(x_1 - x_2 \bowtie a)$,

[15.] E.g., let μ be $\{x - z \geq 0, x - z \leq 1, x_0 - z = 0, x_1 - z = 1\}$. Thus, $\mu \models_{\mathcal{LA}(\mathbb{Z})} ((x = x_0) \vee (x = x_1))$, but $\mu \not\models_{\mathcal{LA}(\mathbb{Z})} (x = x_0)$ and $\mu \not\models_{\mathcal{LA}(\mathbb{Z})} (x = x_1)$.

[16.] Also called *difference logic* or *separation logic*. Notice the overlapping with the notion of "separation logic" defined in [Rey02], where this term indicates an extension of Hoare logic for reasoning about programs that use shared mutable data structures.

s.t. $\bowtie \in \{\leq, <, \neq, =, \geq, >\}$, and the a is an (interpreted) constant labeling one value in $\mathbb{Q}$ and $\mathbb{Z}$ respectively. All such literals can be easily rewritten in terms of *positive difference inequalities* $(x - y \leq a)$ only. [17.]

$\mathcal{DL}(\mathbb{Q})$ is stably-infinite and convex. The $\mathcal{DL}(\mathbb{Q})$-satisfiability of sets of quantifier-free difference inequalities is decidable and polynomial. The main algorithms encode the problem into that of finding negative cycles into a weighted oriented graph, called *constraint graph*, using variants of the well-known Bellman-Ford a shortest-path algorithm [CG99]. [18.] Efficient incremental algorithms for $\mathcal{DL}(\mathbb{Q})$-solvers have been conceived, which can efficiently perform conflict-set generation [CG99], deduction of unassigned literals [NO05a, CM06a] and of interface equalities (see, e.g., [LM06]).

$\mathcal{DL}(\mathbb{Z})$ is stably-infinite and non-convex. [19.] As with $\mathcal{DL}(\mathbb{Q})$, the $\mathcal{DL}(\mathbb{Z})$-satisfiability of sets of quantifier-free difference inequalities is decidable and polynomial, whilst problem the deduction of (disjunctions of) interface equalities is NP-complete [LM05]. The algorithms used for $\mathcal{DL}(\mathbb{Z})$-solvers are the same as for $\mathcal{DL}(\mathbb{Q})$ [CG99, NO05a, CM06a], except for the fact that we are not aware of any specialized algorithm for $\mathcal{DL}(\mathbb{Z})$-solvers able to efficiently deduce disjunctions of interface equalities. Interesting results have also been obtained by implementing "mixed" techniques combining a lazy approach with "eager" encodings of $\mathcal{DL}(\mathbb{Z})$ into SAT [WIGG05, GTG06, KS06].

4.2.4 Unit-Two-Variable-Per-Inequality

The *Unit-Two-Variable-Per-Inequality ($\mathcal{UTVPI}$)* theory is a subcase of $\mathcal{LA}(\mathbb{Z})$ whose atoms can be written in the form $(\pm x_2 \pm x_1 \leq c)$. Notice that $\mathcal{DL}(\mathbb{Z})$ is a sub-theory of $\mathcal{UTVPI}$. $\mathcal{UTVPI}$ is stably-infinite and non-convex. [20.] The $\mathcal{DL}(\mathbb{Q})$-satisfiability of sets of quantifier-free difference inequalities is decidable and polynomial, and $\mathcal{UTVPI}$ is the most expressive sub-theory of $\mathcal{LA}(\mathbb{Z})$ with this feature (see [HS97]).

Many $\mathcal{UTVPI}$-solvers are based on the iterative transitive closure [HS97]: as soon as a new constraint is added into the system, all possible consequences of the input are computed, until either a pair of contradictory constraints are generated, or a fixpoint is reached. [LM05] proposed instead one non-incremental though asymptotically faster algorithm based on negative cycle detection on an extended constraint graph. [21.] This algorithm can efficiently perform conflict-set generation and deduction of unassigned equalities, and some form of

[17.] First, all literals are straightforwardly rewritten into boolean combinations of difference inequalities, by applying rules like, e.g., $(x - y > a) \Longrightarrow \neg(x - y \leq a)$, $(x - y \neq a) \Longrightarrow (\neg(x - y \leq a) \vee \neg(y - x \leq -a))$. Negated differences are then rewritten into positive ones by applying $\neg(x - y \leq a) \Longrightarrow (y - x \leq -a - 1)$ ($\mathcal{DL}(\mathbb{Z})$ case) or $\neg(x - y \leq a) \Longrightarrow (y - x \leq -a - \epsilon)$ ($\mathcal{DL}(\mathbb{Q})$ case), for a sufficiently small ϵ [ACGM04]. Notice that negated equalities may require being split into the disjunction of two difference inequalities.

[18.] Intuitively, a node of the graph represents univocally one variable x_i, and a labeled arc $x_1 \overset{a}{\longmapsto} x_2$ represents the difference inequality $(x_2 - x_1 \leq a)$, meaning "the length of the shortest path from x_1 to x_2 is smaller or equal to a". The graph represents an inconsistent set of difference inequalities if it contains a cycle $x_0 \overset{a_0}{\longmapsto} x_1 \overset{a_1}{\longmapsto} \dots \overset{a_{n-1}}{\longmapsto} x_n \overset{a_n}{\longmapsto} x_0$ s.t. $\sum_{i=0}^{n} a_i < 0$, corresponding to a $\mathcal{DL}$-inconsistent subset $\bigcup_{i=0}^{n-1}\{x_{i+1} - x_i \leq a_i\} \cup \{x_0 - x_n \leq a_n\}$.

[19.] Same example as in Footnote 15..

[20.] Same example as in Footnote 15..

[21.] Intuitively, the set of $\mathcal{UTVPI}$ constraints is encoded into an equisatisfiable set of $\mathcal{DL}(\mathbb{Z})$ problems by introducing two variables x^+ and x^- for each original variable x, representing x and $-x$ respectively, and encoding each constraint with a pair of $\mathcal{DL}(\mathbb{Z})$ constraint (e.g., $(x^+ - y^- \leq a)$ and $(y^+ - x^- \leq a)$ for $(x + y \leq a)$).

deduction of interface equalities. As with $\mathcal{DL}(\mathbb{Z})$, we are not aware of any specialized algorithm for $\mathcal{UTVPI}$-solvers able to efficiently deduce disjunctions of interface equalities.

4.2.5 Bit vectors

The *theory of fixed-width bit vectors ($\mathcal{BV}$)* [22.] is a F.O. theory with equality which aims at representing Register Transfer Level (RTL) hardware circuits, so that components such as data paths or arithmetical sub-circuits are considered as entities as a whole, rather than being encoded into purely propositional sub-formulae ("*bit blasting*").

In $\mathcal{BV}$ terms indicate fixed-width bit vectors, and are built from variables (e.g., $\mathbf{x}^{[32]}$ indicates a bit vector x of 32 bits) and constants (e.g., $\mathbf{0}^{[16]}$ denotes a vector of 16 0's) by means of interpreted functions representing standard RTL operators: word concatenation (e.g., $\mathbf{0}^{[16]} \circ \mathbf{z}^{[16]}$), sub-word selection (e.g., $(\mathbf{x}^{[32]}[20:5])^{[16]}$), modulo-n sum and multiplication (e.g., $\mathbf{x}^{[32]} +_{32} \mathbf{y}^{[32]}$ and $\mathbf{x}^{[16]} \cdot_{16} \mathbf{y}^{[16]}$), bitwise-operators $\mathbf{and}_n$, $\mathbf{or}_n$, $\mathbf{xor}_n$, $\mathbf{not}_n$ (e.g., $\mathbf{x}^{[16]}\mathbf{and}_{16}\ \mathbf{y}^{[16]}$), left and right shift $<\!<_n$, $>\!>_n$ (e.g., $\mathbf{x}^{[32]}<\!<_4$). Atomic expressions can be built from terms by applying interpreted predicates like $\leq_n$, $<_n$ (e.g., $\mathbf{0}^{[32]} \leq_{32} \mathbf{x}^{[32]}$) and equality.

$\mathcal{BV}$ is non-convex and non-stably infinite. The $\mathcal{BV}$-satisfiability of sets of quantifier-free literals is decidable and NP-complete. Different approaches for $\mathcal{BV}$-satisfiability have been proposed: some authors (e.g, [FDK98, ZKC01, BD02, PICW04, BBC$^+$06a]) encode bits, bit vectors and (most of) their operators into $\mathcal{LA}(\mathbb{Z})$; [BCF$^+$07] propose a layered (see §4.3) $\mathcal{BV}$-solver base on a hierarchy of rewriting steps, and a final call to a $\mathcal{LA}(\mathbb{Z})$-solver; others [BD94, CMR97, MR98, BDL98] provide canonizers and $\mathcal{T}$-solvers explicitly for (sub-theories of) $\mathcal{BV}$, which are integrated by means of Nelson-Oppen/Shostak-style schema (see §8.2). The quest for suitable algorithms for efficient $\mathcal{BV}$-solvers is currently a hot research topic.

4.2.6 Other theories of interest

We very briefly recall some other theories of interest, in particular in the filed of software verification. Here we provide only a very high-level description, and point the reader to the reported bibliography (e.g., to [MZ03]) for a more detailed description.

The *theory of arrays ($\mathcal{AR}$)* [23.] aims at modeling the behavior of arrays/memories. The signature consists in the two interpreted function symbols *write* and *read*, s.t. $write(a, i, e)$ represents (the state of) the array resulting from storing an element e into the location of address i of an array a, and $read(a, i)$ represents the element contained in the array a at location i. $\mathcal{AR}$ is formally characterized by the following axioms (see [MZ03]):

$$\forall a.\forall i.\forall e.\ (read(write(a, i, e), i) = e), \tag{8}$$

$$\forall a.\forall i.\forall j.\forall e.\ ((i \neq j) \rightarrow read(write(a, i, e), j) = read(a, j)), \tag{9}$$

$$\forall a.\forall b.\ (\forall i.(read(a, i) = read(b, i)) \rightarrow (a = b)). \tag{10}$$

(8) and (9), called *McCarthy's axioms*, characterize the intended meaning of *write* and *read*, whilst (10), called the *extensionality axiom*, requires that, if two arrays contain the

[22.] We should better say the *theories* of bit vectors, because many variants of $\mathcal{BV}$ have been proposed. Like in [MZ03], here we simply aim at summing up into one theory the main concepts related to these theories. The same comment holds also for the theories mentioned in §4.2.6.

[23.] See Footnote 22..

same values in all locations, than they must be the same array. Theories of arrays are called *extensional* if they include (10), *non-extensional* otherwise. The $\mathcal{AR}$-satisfiability of sets of quantifier-free literals is decidable and NP-complete [SDBL01]. $\mathcal{AR}$ is typically handled in combination with other theories, by means of Nelson-Oppen/Shostak-style integration schema (see §8.2). Decision procedures for $\mathcal{AR}$ have been presented, e.g., in [NO79, DNS05, SDBL01]. We refer the reader to [SDBL01] for an overview.

The *theory of lists ($\mathcal{LI}$)* aims at modeling the behavior of lists. The signature consists in the three interpreted function symbols *cons*, *car*, *cdr* representing the standard LISP constructor and selectors for lists. $\mathcal{LI}$ is formally characterized by the following axioms (see [MZ03]):

$$\forall x.\ (cons(car(x), cdr(x)) = x), \tag{11}$$

$$\forall x.\forall y.\ (car(cons(x, y)) = x),\ \forall x.\forall y.\ (cdr(cons(x, y)) = y), \tag{12}$$

$$\forall x.(car(x) \neq x),\ \forall x.(cdr(x) \neq x),\ \forall x.(car(car(x)) \neq x),\ \forall x.(car(cdr(x)) \neq x), \ldots . \tag{13}$$

(11) and (12), called *construction* and *selection axioms* respectively, characterize the intended meaning of *cons*, *car* and *cdr*, whilst (the infinite sequence of) the *acyclicity axioms* (13) force the list to be acyclic. The $\mathcal{LI}$-satisfiability of sets of quantifier-free literals is decidable and linear in time [Opp80]. $\mathcal{LI}$ is a subcase of the *theory of recursive datatypes ($\mathcal{RDT}$)*, which introduce more general kinds of constructors and selectors, and for which decision procedures have been developed. We refer the readers, e.g., to [Opp80, BST06, BE06] for more details.

4.3 Layered $\mathcal{T}$-*solvers*

In many calls to $\mathcal{T}$-*solver*, a general solver for $\mathcal{T}$ is not needed: very often, the unsatisfiability of the current assignment μ can be established in less expressive, but much easier, sub-theories. Thus, $\mathcal{T}$-*solver* may be organized in a *layered hierarchy* of solvers of increasing solving capabilities [ABC$^+$02a, BBC$^+$05a, SS05, CM06b, BCF$^+$07]. If a higher level solver finds a conflict, then this conflict is used to prune the search at the boolean level; if it does not, the lower level solvers are activated.

The general idea consists in stratifying the problem over N layers $L_0, L_1, \ldots, L_{N-1}$ of increasing complexity, and searching for a solution "at a level as simple as possible". In our view, each level considers only an abstraction of the problem which interprets a subgrammar $G_0, G_1, \ldots, G_{N-1}$ of the original problem, G_{N-1} being the grammar G of the problem. Since L_n refines L_{n-1}, if the problem does not admit a solution at level L_n, then it does not at $L_0, \ldots, L_{n-1}$. If indeed a solution S exists at L_n, either n equals $N-1$, in which case S solves the problem, or a refinement of S must be searched at L_{n+1}. In this way, much of the reasoning can be performed at a high level of abstraction. This results in an increased efficiency in the search of the solution, since low-level searches, which are often responsible for most of the complexity, are avoided whenever possible.

The simple and general idea above maps to an N-layered architecture of the solver. In general, a layer L_n is called by layer L_{n-1} to refine a (maybe partial) solution S of the problem. L_n must check for unsatisfiability of S and (a) return failure if no refinement can be found, or (b) invoke L_{n+1} upon a refinement S', unless n equals $N-1$. An explanation for failure can be added in case (a), to help higher levels "not to try the same wrong solution

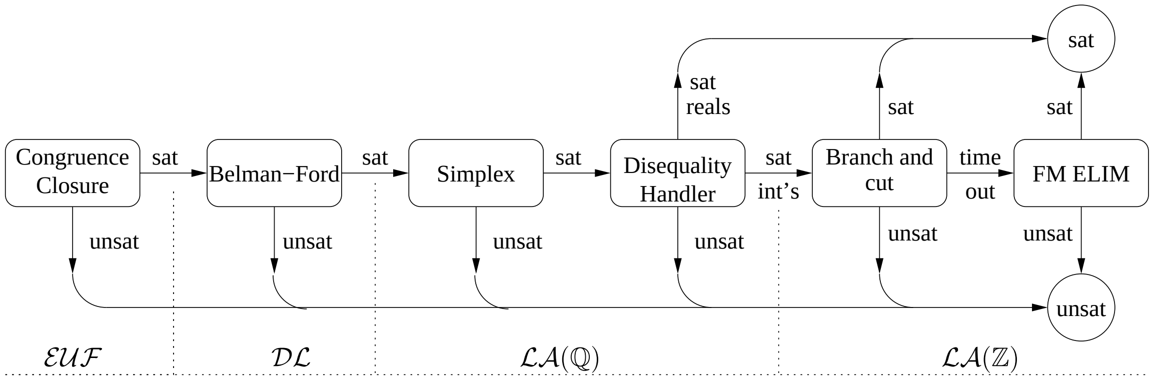


Figure 4. Example of a layered $\mathcal{T}$-*solver*: the $\mathcal{LA}(\mathbb{Z})$-solver in MathSAT [BBC+05b].

twice". The schema can be further enhanced by allowing each layer L_i infer novel equalities and inequalities and to pass them down to the next layer L_{i+1}, so that to better drive its search [SS05, SS06a, CM06b].

For example, Figure 4 describes the control flow of the $\mathcal{LA}(\mathbb{Z})$-solver in MathSAT [BBC+05b]. Four logical components can be distinguished. First, the current assignment μ is passed to the $\mathcal{EUF}$-solver, which implements the congruence closure algorithm in [NO03]. The solver considers as uninterpreted all arithmetical functions and predicates in the atoms, detecting inconsistencies due to the properties of equality and to the congruence properties (e.g., $\{(x = y), (z = x + w), (v = y + w), (z \neq v)\}$). Second, $\mathcal{T}$-*solver* tries to find a conflict over the $\mathcal{DL}$ inequalities by means of a Bellman-Ford procedure. Third, if this solver does not find a conflict, $\mathcal{T}$-*solver* tries to find a conflict over the equalities and inequalities in $\mathcal{LA}(\mathbb{Q})$ by means of a Simplex procedure; if this is not enough, an ad hoc device is invoked which applies also to disequalities. Finally, if the current assignment is also satisfiable over the reals and the variables are to be interpreted over the integers, Euler reduction and a simple form of branch-and-cut are carried out, to search for solutions over the integers. If branch-and-cut does not find either an integer solution or a conflict within a small, predetermined amount of search, the Omega constraint solver [Pug91, Ome], based on Fourier-Motzkin elimination, is called on the current assignment.

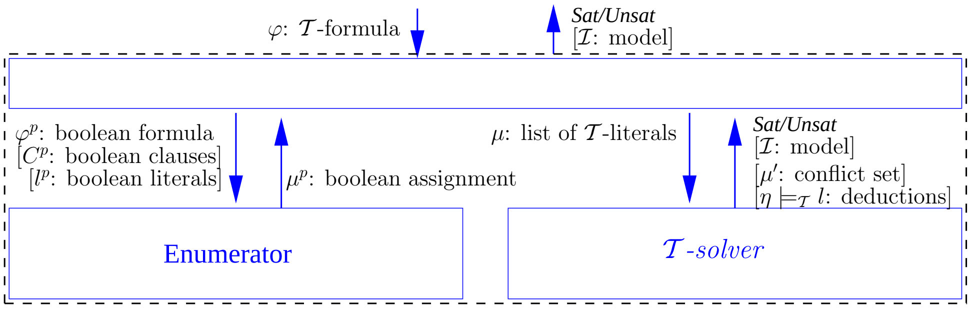

φ: $\mathcal{T}$-formula
Sat/Unsat
[$\mathcal{I}$: model]
φ^p: boolean formula
[C^p: boolean clauses]
[l^p: boolean literals]
μ^p: boolean assignment
μ: list of $\mathcal{T}$-literals
Sat/Unsat
[$\mathcal{I}$: model]
[μ': conflict set]
[$\eta \models_{\mathcal{T}} l$: deductions]
Enumerator
$\mathcal{T}$-*solver*

Figure 5. Basic architectural schema of a lazy $SMT(\mathcal{T})$ procedure.

5. Integrating DPLL and $\mathcal{T}$-*solvers*

Different representations and variants of SMT procedures integrating DPLL and $\mathcal{T}$-*solvers* have been presented. Following [FJOS03], we partition all such procedures into two main categories: the *offline* procedures, in which DPLL is used as a SAT solver which is reinvoked from scratch each time an assignment is found $\mathcal{T}$-unsatisfiable (e.g., [BDS02a, dMRS02b]), and the *online* procedures, in which DPLL is modified to be used directly as an enumerator (e.g., [GS96a, WW99, ACG99, ABC$^+$02a, FJOS03, GHN$^+$04, BBC$^+$06b]). We also recall from [NOT05, NO05a] the Abstract-DPLL Modulo Theories DPLL ($\mathcal{T}$) logical framework, which allows for expressing and reasoning about most variants of lazy DPLL-based procedures in a formal way.

In §5.1 we present a general, high-level integration schema between DPLL and $\mathcal{T}$-*solver*. In §5.2 and §5.3 we present the offline and online integration schemata respectively. (The offline and online schemata will be further discussed in §7.1.2.) Finally, in §5.4 we recall the main concepts of the Abstract-DPLL Modulo Theories logical framework.

5.1 A basic integration schema

A basic architectural schema of a typical lazy DPLL-based $SMT(\mathcal{T})$ procedure, which we generically call $\mathcal{T}$-DPLL henceforth, is reported in Figure 5. $\mathcal{T}$-DPLL takes as input a $\mathcal{T}$-formula φ, and builds its boolean abstraction $\varphi^p =_{def} \mathcal{T}2\mathcal{B}(\varphi)$. (Notationally, we frequently use the superscript ^{p} to denote boolean abstractions, i.e., given a $\mathcal{T}$-expression e, we write e^p to denote $\mathcal{T}2\mathcal{B}(e)$.) Notice that both $\mathcal{T}2\mathcal{B}$ and $\mathcal{B}2\mathcal{T}$ can be implemented so that to require constant time to map a literal from one representation to the other. [24.]

φ^p is given in input to Enumerator, which enumerates truth assignments in a complete collection $\{\mu_1^p, .., \mu_n^p\}$ for φ^p. Each time a new μ^p is generated, its corresponding list of $\mathcal{T}$-literals $\mu =_{def} \mathcal{B}2\mathcal{T}(\mu^p)$ is fed to $\mathcal{T}$-*solver*. (If μ contains also boolean literals, then they are dropped because they do not contribute to the $\mathcal{T}$-satisfiability of μ.) If μ is found $\mathcal{T}$-satisfiable, then the procedure returns Sat, possibly returning also the model $\mathcal{I}$ produced. If not, a new assignment is generated by Enumerator. The process is repeated until

[24.] In practice, it is often necessary or convenient that $\mathcal{B}2\mathcal{T}$ maps *literals* rather than *atoms*, i.e., that $\mathcal{B}2\mathcal{T}(\neg A_i)$ is syntactically different from $\neg\mathcal{B}2\mathcal{T}(A_i)$. E.g., if $\mathcal{T}$ is $\mathcal{DL}(\mathbb{Z})$, we may have that $\mathcal{B}2\mathcal{T}(A_i) = (x - y \leq 3)$ and that $\mathcal{B}2\mathcal{T}(\neg A_i) = (y - x \leq -4)$.

```
1.   SatValue $\mathcal{T}$-DPLL ($\mathcal{T}$-formula $\varphi$) {
2.       $\varphi^p = \mathcal{T}2\mathcal{B}(\varphi)$;
3.       while (DPLL($\varphi^p, \mu^p$) == Sat) {
4.           if ($\mathcal{T}$-solver($\mathcal{B}2\mathcal{T}(\mu^p)$) == Sat)
5.               return Sat;
6.           $\varphi^p = \varphi^p \wedge \neg\mu^p$;
7.       };
8.       return Unsat;
9.   };
```

Figure 6. A simplified offline integration schema for lazy $SMT(\mathcal{T})$ procedures.

either one $\mathcal{T}$-satisfiable assignment is found, or no more assignments are generated by Enumerator. In the former case φ is $\mathcal{T}$-satisfiable, in the latter it is not.

As discussed in §4.1, $\mathcal{T}$-*solver* can also return one or more theory conflict set(s) μ' (if μ is $\mathcal{T}$-unsatisfiable) or one or more deduction(s) $\eta \models_{\mathcal{T}} l$ (if μ is $\mathcal{T}$-satisfiable). If so, boolean clauses like $C^p =_{def} \mathcal{T}2\mathcal{B}(\neg\mu')$ or $C^p =_{def} \mathcal{T}2\mathcal{B}(\neg\eta \vee l)$, and deduced boolean literals $l^p =_{def} \mathcal{T}2\mathcal{B}(l)$, can be passed back to Enumerator, so that to drive its boolean search. Moreover, $\mathcal{T}$-*solver* can be invoked also on intermediate assignments during their construction, so that to prune the boolean search. These issues will be discussed in detail in §6 and §7.

5.2 The offline approach to integration

Offline schemata for integrating DPLL and $\mathcal{T}$-*solver* have been independently proposed by [BDS02a] and by [dMRS02b]. [25.] In its naivest form, the idea works as described in Figure 6.

The propositional abstraction φ^p of the input formula φ is given as input to a SAT solver, which either decides that φ^p is unsatisfiable, and hence φ is $\mathcal{T}$-unsatisfiable, or it returns a satisfying assignment μ^p; in the latter case, $\mathcal{B}2\mathcal{T}(\mu^p)$ is given as input to $\mathcal{T}$-*solver*. If $\mathcal{B}2\mathcal{T}(\mu^p)$ is found $\mathcal{T}$-consistent, than φ is $\mathcal{T}$-consistent. If not, $\neg\mu^p$ is added as a clause to φ^p [26.], and the SAT solver is *restarted from scratch* on the resulting formula. Notice that here DPLL is used as a black-box SAT solver, and that the loop 3.-7. works as a (non-redundant) Enumerator, because step 6. prevents DPLL from finding the same assignment more than once.

A way more efficient form is when $\mathcal{T}$-*solver* is able to return the conflict set η which caused the $\mathcal{T}$-inconsistency of $\mathcal{B}2\mathcal{T}(\mu^p)$. If this is the case, then $\mathcal{T}2\mathcal{B}(\neg\eta)$ is added as a clause to φ instead of $\neg\mu^p$. As typically the former is way smaller than the latter, this drastically reduces the search space. This and other optimizations will be discussed in §6.

[25.] The offline approach is also called *lemmas on demand* approach in [dMRS02b].

[26.] $\neg\mu^p$ is called *blocking clause*, because it blocks the future generation of every assignment containing μ^p, or *cube*, because it represents a (hyper-)cube of counter-assignments.

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

Figure 7. An online schema of $\mathcal{T}$-DPLL based on modern DPLL.

5.3 The online approach to integration

Several procedures exploiting the online integration schema have been proposed in different communities and domains (see, e.g., [GS96a, WW99, ACG99, ABC+02a, FJOS03, GHN+04, BBC+06b]). In the online integration schema, $\mathcal{T}$-DPLL is a variant of the DPLL procedure, modified to work as an enumerator of truth assignments, whose $\mathcal{T}$-satisfiability is checked by a *$\mathcal{T}$-solver*.

Figure 7 represent the schema of an online $\mathcal{T}$-DPLL procedure based on a modern DPLL engine, like that of Figure 1. 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 $Lits(\varphi)$ and the bijective mapping $\mathcal{T}2\mathcal{B}/\mathcal{B}2\mathcal{T}$ on literals. [27.]

$\mathcal{T}$-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 described in §3.1 with some theory-dependent rewriting steps on the $\mathcal{T}$-literals of φ. (The latter will be described in details in §6.1. and §6.2.)

Example 5.1. *Suppose that initially* $\varphi = (x > 0) \wedge (A_1 \vee (x \leq 0)) \wedge (\neg A_1 \vee (x \leq 0))$*, and* $\mu = \emptyset$*. Then* $\mathcal{T}$-preprocess *may rewrite the literal* $(x > 0)$ *into* $\neg(x \leq 0)$*, so that* φ *is*

[27.] Hereafter we implicitly assume that all functions called in $\mathcal{T}$-DPLL have direct access to $Lits(\varphi)$ and to $\mathcal{T}2\mathcal{B}/\mathcal{B}2\mathcal{T}$, and that both $\mathcal{T}2\mathcal{B}$ and $\mathcal{B}2\mathcal{T}$ require constant time for mapping each literal.

rewritten into $\neg(x \leq 0) \wedge (A_1 \vee (x \leq 0)) \wedge (\neg A_1 \vee (x \leq 0))$*, and hence find a boolean conflict by applying BCP. Thus* φ *is* $\mathcal{T}$*-unsatisfiable.*

$\mathcal{T}$`-decide_next_branch` plays the same role as `decide_next_branch` in DPLL (see Figure 1), but it may take into consideration also the semantics in $\mathcal{T}$of the literals to select. (This will be discussed with more details in §7.2.5.)

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

(i) μ^p propositionally violates φ^p ($\mu^p \wedge \varphi^p \models_p \bot$). If so, $\mathcal{T}$`-deduce` behaves like `deduce` in DPLL, returning Conflict.

(ii) μ^p propositionally satisfies φ^p ($\mu^p \models_p \varphi^p$). If so, $\mathcal{T}$`-deduce` invokes $\mathcal{T}$*-solver* on $\mathcal{B}2\mathcal{T}(\mu^p)$: 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 $\mathcal{B}2\mathcal{T}(\mu^p)$ also at this intermediate stage: if $\mathcal{T}$*-solver* returns Unsat, then $\mathcal{T}$`-deduce` returns Conflict. (This enhancement, called *early pruning*, will be discussed with more details in §6.3.)

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. $\eta \subset \mu$, as in §4.1.4. If so, $\mathcal{T}$`-deduce` can iteratively deduce also literals l^d which can be inferred *in* $\mathcal{T}$ (i.e., s.t. $\mathcal{B}2\mathcal{T}(\mu^p) \models_{\mathcal{T}} \mathcal{B}2\mathcal{T}(l^d)$). (This enhancement, called $\mathcal{T}$*-propagation*, will be discussed with more details in §6.4.)

$\mathcal{T}$`-analyze_conflict` is an extensions of `analyze_conflict` of DPLL in §3.1: 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`, as described in §3.1; if instead 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 as a conflict set the boolean abstraction η^p of the theory conflict set η produced by $\mathcal{T}$*-solver* (i.e., $\eta^p := \mathcal{T}2\mathcal{B}(\eta)$), or computes a mixed boolean+theory conflict set by a backward-traversal of the implication graph starting from the conflicting clause $\neg\mathcal{T}2\mathcal{B}(\eta)$ (see §6.5). If $\mathcal{T}$*-solver* is not able to return a theory conflict set, the whole assignment μ may be used, after removing all boolean literals from μ. 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 and backtracks up to `blevel`. (These features, called $\mathcal{T}$*-backjumping* and $\mathcal{T}$*-learning*, will be discussed with more details in §6.5 and §6.6.)

On the whole, $\mathcal{T}$-DPLL differs from the DPLL schema of Figure 1 because it exploits:

- an extended notion of *deduction of literals*: not only *boolean deduction* ($\mu^p \wedge \varphi^p \models_p l^p$), but also *theory deduction* ($\mathcal{B}2\mathcal{T}(\mu^p) \models_{\mathcal{T}} \mathcal{B}2\mathcal{T}(l^p)$);

- an extended notion of *conflict*: not only *boolean conflict* ($\mu^p \wedge \varphi^p \models_p \bot$), but also *theory conflict* ($\mathcal{B}2\mathcal{T}(\mu) \models_{\mathcal{T}} \bot$), or even *mixed boolean+theory conflict* ($\mathcal{B}2\mathcal{T}(\mu^p \wedge \varphi^p) \models_{\mathcal{T}} \bot$). See 6.5.

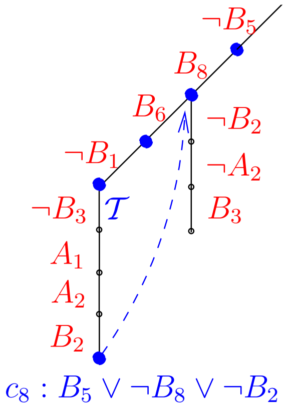


Figure 8. Boolean search (sub)tree in the scenario of Example 5.2. (A diagonal line, a vertical line and a vertical line tagged with "$\mathcal{T}$" denote literal selection, unit propagation and $\mathcal{T}$-propagation respectively; a bullet "●" denotes a call to $\mathcal{T}$*-solver*.)

Example 5.2. *Suppose* $\mathcal{T}$-DPLL *implements all the enhancements described above, and consider the* $\mathcal{LA}(\mathbb{Q})$*-formula* φ *in Example* 2.1*, (which we report here for convenience):*

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

Look as Figure 8*. Suppose* $\mathcal{T}$`-decide_next_branch` *selects, in order,* $\neg B_5, B_8, B_6, \neg B_1$ *(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* $\mathcal{B}2\mathcal{T}(\{\neg B_5, B_8, B_6, \neg B_1\})$*:*

$$\{\neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2 - x_4 \leq 6), \neg(2x_2 - x_3 > 2)\}.$$

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*.*

By step (iii), $\mathcal{T}$`-deduce` *invokes* $\mathcal{T}$*-solver on* $\mathcal{B}2\mathcal{T}(\{\neg B_5, B_8, B_6, \neg B_1, \neg B_3, A_1, A_2, B_2\})$*:*

$$\{\neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2 - x_4 \leq 6), \neg(2x_2 - x_3 > 2), \neg(3x_1 - 2x_2 \leq 3), (x_1 - x_5 \leq 1)\}$$

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*

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

$$
\begin{array}{llll}
\mathcal{T}\text{-Propagate:} & \langle \mu \mid \varphi \rangle & \Rightarrow \langle \mu, l \mid \varphi \rangle & \textit{if} \begin{cases} \mu \models_{\mathcal{T}} l \\ l \textit{ is undefined in } \mu \\ l \textit{ or } \neg l \textit{ occurs in } \varphi \end{cases} \\
\mathcal{T}\text{-Backjump:} & \langle \mu, l^d, \mu' \mid \varphi, C \rangle & \Rightarrow \langle \mu, l' \mid \varphi, C \rangle & \textit{if} \begin{cases} \mu, l^d, \mu' \models_p \neg C \\ \textit{there is some clause } C' \vee \ l' \textit{ s.t.}: \\ \quad \varphi, C \models_{\mathcal{T}} C' \vee l' \textit{ and } \mu \models_p \neg C' \\ \quad l' \textit{ is undefined in } \mu \\ \quad l' \textit{ or } \neg l' \textit{ occurs in } \varphi \textit{ or} \\ \quad\ \textit{in } \mu \cup \{l^d\} \cup \mu' \end{cases} \\
\mathcal{T}\text{-Learn:} & \langle \mu \mid \varphi \rangle & \Rightarrow \langle \mu \mid \varphi, C \rangle & \textit{if} \begin{cases} \textit{all atoms in } C \textit{ occur in } \varphi \\ \varphi \models_{\mathcal{T}} C \end{cases} \\
\mathcal{T}\text{-Discharge:} & \langle \mu \mid \varphi, C \rangle & \Rightarrow \langle \mu \mid \varphi \rangle & \textit{if} \ \{ \ \varphi \models_{\mathcal{T}} C
\end{array}
$$

Figure 9. The Abstract-DPLL Modulo Theories logical framework from [NOT06]. l^d denotes a decision literal. In the $\mathcal{T}$-Backjump rule, C and $C' \vee l'$ represent the conflicting and the conflict clause respectively.

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, and the search proceeds from there.

5.4 The Abstract-DPLL Modulo Theories logical framework: DPLL($\mathcal{T}$)

As hinted in §3.2, [Tin02, NOT05, NO05a, NOT06] proposed an abstract rule-based formulation of DPLL-based lazy SMT systems, the *Abstract* DPLL *Modulo Theories*, also known as *DPLL($\mathcal{T}$)*. This allows for expressing and reasoning about most variants of these procedures in a formal way. In particular, [GHN$^+$04, NO05a] used such a framework to describe new DPLL-based procedures.

The Abstract-DPLL Modulo Theories extends the Abstract-DPLL framework of §3.2 by adding the set of rules Figure 9 [NOT05] to these of Figure 3. (Notice that here all literals, assignments and formulas are in the language of $\mathcal{T}$, and that here the symbol "$\models$" in Figure 3 must be interpreted as "$\models_p$".) The first three rules match the notion of $\mathcal{T}$-propagation, $\mathcal{T}$-backjumping and $\mathcal{T}$-learning introduced in §5.3 (but see also §6.4, §6.5 and §6.6). The fourth rule matches the fact that $\mathcal{T}$-learned clauses may be discharged when necessary, in order to avoid an explosion in size of the input formula. [28] The only rule deserving some more explanation is $\mathcal{T}$-Backjump: if a branch $\mu \cup \{l^d\} \cup \mu'$ falsifies propositionally one clause C (the conflicting clause), and a conflict clause $C' \vee l'$ [29] can be computed from C s.t. the corresponding conflict set $\neg(C' \vee l')$ falsifies $\varphi \wedge C$ in $\mathcal{T}$, $\neg C' \subseteq \mu$,

[28] As already remarked in §3.2, the formalization of the rules in [NOT05, NO05a, NOT06] changes slightly from paper to paper. Here we report the most-recent one from [NOT06]. We have adapted the notation to that used in this paper.

[29] Also called the *backjump clause* in [NOT06].

$l' \notin \mu$, and l' or $\neg l'$ occur in φ or in $\mu \cup \{l^d\} \cup \mu'$, then it is possible to backjump up to μ, and hence unit-propagate l' on the conflict clause $(C' \vee l')$.

Example 5.3. *Consider the formula and the scenario of Example 5.2. The execution can be represented in Abstract-*DPLL *Modulo Theories/DPLL($\mathcal{T}$) as follows:*

$\langle \quad |\varphi\rangle$

Decide $\neg B_5, B_8, B_6, \neg B_1 \;\Rightarrow$

$\langle \neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2 - x_4 \leq 6), \neg(2x_2 - x_3 > 2) \quad |\varphi\rangle$

Theory Propagate $\neg B_3 \;\Rightarrow$

$\langle \neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2 - x_4 \leq 6), \neg(2x_2 - x_3 > 2), \neg(3x_1 - 2x_2 \leq 3) \quad |\varphi\rangle$

Unit Propagate $A_1, A_2, B_2 \;\Rightarrow$

$\langle \neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2 - x_4 \leq 6), \neg(2x_2 - x_3 > 2), \neg(3x_1 - 2x_2 \leq 3), A_1, A_2, (x_1 - x_5 \leq 1) |\varphi\rangle$

$\mathcal{T}$-Learn $c_8 \;\Rightarrow$

$\langle \neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2 - x_4 \leq 6), \neg(2x_2 - x_3 > 2), \neg(3x_1 - 2x_2 \leq 3), A_1, A_2, (x_1 - x_5 \leq 1) |\varphi, c_8\rangle$

$\mathcal{T}$-Backjump $\;\Rightarrow$

$\langle \neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), \neg(x_1 - x_5 \leq 1) \quad |\varphi, c_8\rangle$

Unit Propagate $\neg A_2,\ B_1 \;\Rightarrow$

$\langle \neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), \neg(x_1 - x_5 \leq 1), \neg A_2, (3x_1 - 2x_2 \leq 3) \quad |\varphi, c_8\rangle$

$\ldots$

As in Abstract-DPLL, if a finite sequence $\langle \emptyset \mid \varphi \rangle \Rightarrow \langle \mu_1 \mid \varphi_1 \rangle \Rightarrow \ldots \Rightarrow fail$ is found, then the formula is unsatisfiable; if a finite sequence $\langle \emptyset \mid \varphi \rangle \Rightarrow \ldots \Rightarrow \langle \mu_n \mid \varphi_n \rangle$ is found so that no rule can be further applied, then the formula is satisfiable. Different strategies in applying the rules correspond to different variants of the algorithm. [NOT05, NOT06] provide a group of results about termination, correctness and completeness of various configurations. We refer the reader to [NOT05, NO05a, NOT06] for further details.

A DPLL($\mathcal{T}$) system consists of a general *DPLL(X)* engine, very similar in nature to a SAT solver, whose parameter "X" can be instantiated with a *$\mathcal{T}$-solver* for the theory $\mathcal{T}$of interest. [GHN$^+$04] provides a common application programming interface (API) for the *$\mathcal{T}$-solvers*: once the DPLL(X) engine has been implemented, new theories can be dealt with by simply plugging in new *$\mathcal{T}$-solvers* which conform to the API. We refer the reader to [GHN$^+$04] for further details.

6. Optimizing the integration of DPLL and $\mathcal{T}$*-solvers*

In the basic integration schema of §5.1 (and hence in these of §5.2 and §5.3), even assuming ENUMERATOR and $\mathcal{T}$*-solver* are extremely efficient as a stand-alone procedures, their combination can be extremely inefficient. This is due to a couple of intrinsic problems.

- ENUMERATOR 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.

- $\mathcal{T}$*-solver* is used as a memoryless 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.

It is essential to improve the basic integration schema of §5.1 so that the ENUMERATOR (DPLL) 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.

In this section we describe in detail the most effective techniques which have been proposed in various communities in order to optimize the interaction between DPLL and $\mathcal{T}$*-solver*. (Some techniques, like *normalizing* $\mathcal{T}$*-atoms* (§6.1), *early pruning* (§6.3), $\mathcal{T}$*-propagation* (§6.4), $\mathcal{T}$*-backjumping* (§6.5) and $\mathcal{T}$*-learning* (§6.6), have already been introduced in part in §5.3 and §5.4.) We coarsely distinguish four main categories of techniques.

Preprocessing Rewrite the input $\mathcal{T}$-formula φ into an equivalent or equivalently-satisfiable one which is supposedly easier to solve for $\mathcal{T}$-DPLL. Among them we have *normalizing* $\mathcal{T}$*-atoms* (§6.1) and *static learning* (§6.2).

Look-ahead Analyze the current status of the search and get from it as much information as possible which is useful to prune the remaining search space. Among them we have *early pruning* (§6.3), $\mathcal{T}$*-propagation* (§6.4), and *branching heuristics* (§7.2.5).

Look-back When recovering from a failure, try to understand the cause of that failure and use such an information to improve future search. Among them we have $\mathcal{T}$*-backjumping* (§6.5) and $\mathcal{T}$*-learning* (§6.6).

Assignment simplification $\mathcal{T}$-DPLL can provide useful information to make the assignment smaller or simpler for $\mathcal{T}$*-solver*. Among them we have *clustering* (§6.8), *reduction of assignments to prime implicants* (§6.9), *pure-literal filtering* (§6.10) and $\mathcal{T}$*-literal filtering* (§6.11).

Remark 6.1. *The techniques described in this section have been collected from an heterogeneous bibliography (including, e.g., some on modal and description logics), "cleaned" from any information related to the specific theory/logic, and grouped according to the form of interaction between* DPLL *and* $\mathcal{T}$*-solver. To this extent, we remark a few facts. First, techniques focused only on pure boolean reasoning or on pure theory-specific reasoning have been* *briefly discussed in* §*3 and* §*4, and they are not further considered here. Second, the names we adopted here for the various techniques may differ from those used by some of the authors. (E.g.,* $\mathcal{T}$-propagation *(*§*6.4) is called* forward reasoning *in [ACG99],* enhanced early pruning *in [ABC*$^+$*02a],* theory propagation *in [NOT05, NO05a],* theory-driven deduction or $\mathcal{T}$-deduction *in [BBC*$^+$*05a].) Third, we may present separately techniques which some authors present as one technique. (E.g.,* early pruning *(*§*6.3) and* $\mathcal{T}$-propagation *(*§*6.4) are sometimes described as one technique [GHN*$^+$*04].) Finally, the description of some technique may differ significantly from that given by some authors. (E.g., [NOT05, NO05a] describe their procedures in terms of inference rules and control strategies; most authors instead prefer a pseudo-code description.)*

6.1 Normalizing $\mathcal{T}$-atoms.

The idea of normalizing the $\mathcal{T}$-atoms was introduced in the very first DPLL-based procedures for modal logics [GS96a], and it is adopted to some extent in substantially all lazy SMT procedures.

One potential source of inefficiency for $\mathcal{T}$-DPLL is the occurrence in the input $\mathcal{T}$-formula of pairs of syntactically-different $\mathcal{T}$-atoms which are $\mathcal{T}$-equivalent (e.g., $(x_1 + (x_3 - x_2) = 1)$ and $((x_1 + x_3) - 1 = x_2))$, or s.t. one is $\mathcal{T}$-equivalent to the negation of the other (e.g. $(2x_1 - 6x_2 \le 4)$ and $(3x_2 + 2 < x_1)$). If two $\mathcal{T}$-atoms ψ_1, ψ_2 are s.t. $\psi_1 \neq \psi_2$ and $\models_{\mathcal{T}} \psi_1 \leftrightarrow \psi_2$ [resp. $\psi_1 \neq \neg\psi_2$ and $\models_{\mathcal{T}} \psi_1 \leftrightarrow \neg\psi_2$], then they are mapped into distinct boolean atoms $B_1 =_{def} \mathcal{T}2\mathcal{B}(\psi_1)$ and $B_2 =_{def} \mathcal{T}2\mathcal{B}(\psi_2)$, which may be assigned different [resp. identical] truth values by Enumerator. This may cause the useless generation of many $\mathcal{T}$-unsatisfiable assignments and the corresponding useless calls to $\mathcal{T}$-*solver* (e.g., up to $2^{|Atoms(\varphi)|-2}$ calls on assignments like $\{(2x_1 - 6x_2 \le 4), (3x_2 + 2 < x_1), ...\}$).

In order to avoid these problems, it is wise to preprocess 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*: $(x_1 < x_2), (x_1 \ge x_2) \Longrightarrow \neg(x_1 \ge x_2), (x_1 \ge x_2)$.
- *Exploit associativity*: $(x_1 + (x_2 + x_3) = 1), ((x_1 + x_2) + x_3) = 1) \Longrightarrow (x_1 + x_2 + x_3 = 1)$.
- *Sort*: $(x_1 + x_2 - x_3 \le 1), (x_2 + x_1 - 1 \le x_3) \Longrightarrow (x_1 + x_2 - x_3 \le 1))$.
- *Exploiting specific properties of* $\mathcal{T}$: $(x_1 \le 3), (x_1 < 4) \Longrightarrow (x_1 \le 3)$ if $\mathcal{T}$ is $\mathcal{LA}(\mathbb{Z})$.

The applicability and effectiveness of these mappings depends on the theory addressed.

Although rather straightforward, normalizing atoms is an essential step which may drastically reduce the global amount of search [GS96a]. As we described in §5.3, it can be effectively combined with standard boolean preprocessing (like in Example 5.1).

6.2 Static learning

The following idea was proposed by [ACG99] for a lazy SMT procedure for $\mathcal{DL}$. Similar such techniques were generalized and used in [ACKS02, BBC$^+$05b, YM06].

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 value assignments* (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 [ACG99, ACKS02, BBC$^+$05b, YM06].

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, unlike the extra clauses added in "per-constraint" eager approaches [SSB02, SLB03] (see §9.3), 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, and they are not needed for correctness and completeness: rather, they are used only for pruning the boolean search space.

6.3 Early pruning

The following family of optimizations, here generically called *early pruning – EP*,[30.] was introduced by [GS96a] in procedures for modal logics; [WW99, ABC$^+$02a, BDS02a, GHN$^+$04] developed similar ideas in procedures for many SMT problems.

In its simplest form, EP is based on the empirical observation that most assignments which are enumerated by $\mathcal{T}$-DPLL, and which are found Unsat by $\mathcal{T}$*-solver*, are such that their $\mathcal{T}$-unsatisfiability is caused by much smaller subsets. Thus, if the $\mathcal{T}$-unsatisfiability of an assignment μ is detected during its construction, then this prevents checking the $\mathcal{T}$-satisfiability of all the up to $2^{|Atoms(\varphi)|-|\mu|}$ total truth assignments which extend μ.

This suggests to introduce an intermediate call to $\mathcal{T}$*-solver* on intermediate assignment μ, (at least) before each decision. (I.e., in the $\mathcal{T}$-DPLL of Figure 7, 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; therefore $\mathcal{T}$-DPLL returns Unsat and backtracks, avoiding a possibly big amount of useless search.

Example 6.2. *Consider the formula φ of Example 5.2. Suppose that, after four decisions, $\mathcal{T}$-DPLL builds the intermediate assignment:*

$$\mu \quad = \quad \{\neg(2x_2 - x_3 > 2), \neg A_2, (3x_1 - 2x_2 \leq 3), \neg(3x_1 - x_3 \leq 6)\}, \tag{14}$$

(rows 1, 2, 3 and 5, 4 of φ respectively). If $\mathcal{T}$-solver is invoked on μ, it returns Unsat*, and $\mathcal{T}$-DPLL backtracks without exploring any further extension of μ.*

[30.] Also called *intermediate assignment checking* in [GS96a] and *eager notification* in [BDS02a].

In general, early pruning may introduce a very relevant reduction of the boolean search space, and consequently 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. Anyway, we notice that all EP calls to $\mathcal{T}$*-solver* are *incremental*, as described in §4.1.3; thus, if we use an incremental $\mathcal{T}$*-solver*, the overhead of the extra calls is much mitigated.

Many variants and improvements of early pruning have been proposed in the literature. We recall the most important ones.

6.3.1 Selective or intermittent early pruning

Some heuristic criteria can be introduced in order to reduce the number of redundant calls to $\mathcal{T}$*-solver* in early pruning steps. One way is avoid invoking $\mathcal{T}$*-solver* when it is very unlikely that, since the last call, the new literals added to μ can cause inconsistency. For instance, this is the case when they are added only literals which either are purely-propositional [GS96a] or contain new variables [ABC+02a]. Another way is to call $\mathcal{T}$*-solver* every k branching steps, k being an user-defined integer parameter [ACGM04].

6.3.2 Weakened early pruning

In order to further reduce the overhead due to early pruning, another idea is to use, for intermediate checks only, *weaker* but *faster* versions of $\mathcal{T}$*-solver* [BBC+05b]. This is possible because intermediate checks are not necessary to the correctness and completeness of the procedure. The notion of "weaker $\mathcal{T}$*-solver*" depends on the theory $\mathcal{T}$ we are dealing with. Some general ideas are:

- in case of Sat response, avoid reconstructing the satisfying assignments and models (§4.1.1), which are not used in intermediate checks;

- check only easier-to-check subsets of μ. E.g., as $\mathcal{DL}$ is much easier than $\mathcal{LA}$, if μ is $\{(x - y \leq 4), (z - x \leq -6)(z - y = 0), (x - y = z - w)\}$, then we may test only the sub-assignment dealing with $\mathcal{DL}$-literals (e.g., the first three literals in μ, which are $\mathcal{DL}$-inconsistent) and backtrack if this is inconsistent; [31.]

- check μ only on some easier-to-check sub-theory $\mathcal{T}' \subset \mathcal{T}$ (i.e., s.t., if φ is inconsistent in $\mathcal{T}$' then φ is inconsistent in $\mathcal{T}$). For example, as $\mathcal{LA}(\mathbb{Z})$-satisfiability is way harder than $\mathcal{LA}(\mathbb{Q})$-satisfiability (see [BW01]), we may want to check the consistency of an assignment on $\mathcal{LA}(\mathbb{Q})$ rather than on $\mathcal{LA}(\mathbb{Z})$, and backtrack if the $\mathcal{LA}(\mathbb{Q})$*-solver* return Unsat.

To these extends, notice that weakened EP fits naturally with layered theory solvers (§4.3) because, during intermediate checks, it is possible w.l.o.g. to involve only some of the layers.

On the whole, there is a tradeoff between the benefits of reducing the overhead and the drawbacks of reducing the pruning effect.

[31.] This situation is the frequent in the domain of bounded model checking for timed systems, where we have a big majority of $\mathcal{DL}(\mathbb{Q})$-literals, and only very few $\mathcal{LA}(\mathbb{Q})$-literals (see, e.g., [ACKS02]).

6.3.3 Eager early pruning

Some DPLL-based SMT procedures for various theories (e.g., [WW99, SBD02, GHN+04, NO05a]) perform a more *eager* form of early pruning, in which the theory solver is invoked every time a new $\mathcal{T}$-atom is added to the assignment (including those added by unit propagation). If so, we may avoid performing unit propagations at the cost of extra calls to $\mathcal{T}$*-solver*. In some sense, the eager approach privileges theory reasoning wrt. (deterministic) boolean reasoning.

In some cases (e.g., [SBD02, GHN+04]) $\mathcal{T}$*-solver* works as a fully-incremental deduction process, so that an eager interaction with the SAT solver comes natural. However, if this is not the case, then eager early pruning can be extremely expensive due to the possibly big amount of calls to $\mathcal{T}$*-solver*.

6.4 $\mathcal{T}$-propagation

$\mathcal{T}$-propagation [32] was introduced in its simplest form (plunging, see §4.1.4) by [ACG99] for $\mathcal{DL}$; [ABC+02a] proposed an improved technique for $\mathcal{LA}$; however, $\mathcal{T}$-propagation showed its full potential in [Tin02, GHN+04, NO05a], where it was applied aggressively.

As discussed in §4.1.4, 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 a not-yet-assigned atom in φ. If this is the case, then $\mathcal{T}$*-solver* can return l to $\mathcal{T}$-DPLL, so that $\mathcal{T}2\mathcal{B}(l)$ is unit-propagated. This may induce new literals to be assigned, new calls to $\mathcal{T}$*-solver*, new assignments deduced, and so on, possibly causing a beneficial loop between $\mathcal{T}$-propagation and unit propagation.

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 $\mathcal{T}2\mathcal{B}(\eta \rightarrow l)$ 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 §6.6).

Example 6.3. *Look at Figure 10. Consider the scenario at the end of Example 5.2: the current branch is* $\{\neg B_5, B_8, \neg B_2, \neg A_2, B_3\}$*, corresponding to the set of* $\mathcal{LA}(\mathbb{Q})$*-literals*

$$\{\neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), \neg(x_1 - x_5 \leq 1), (3x_1 - 2x_2 \leq 3)\}.$$

Then the $\mathcal{T}$*solver may perform another* $\mathcal{T}$*-propagation step:*

$$\{\neg(3x_1 - x_3 \leq 6), (3x_1 - 2x_2 \leq 3)\} \models_{\mathcal{T}} (2x_2 - x_3 > 2)$$

from which B_1 *is propagated, and hence* A_1 *is unit propagated on* c_1*.*

Instead, suppose that, after the first $\mathcal{T}$*-propagation of Example 5.2,* $\mathcal{T}$-DPLL *had added to* φ^p *the corresponding deduction clause*

$$c_9: \quad B_5 \vee B_1 \vee \neg B_3.$$

If so, then B_1 *and* A_1 *could have been obtained directly by unit propagation on* c_9 *and* c_1 *respectively, saving one call to* $\mathcal{T}$*-solver and one* $\mathcal{T}$*-propagation step.*

[32] Also called *forward reasoning* in [ACG99], *enhanced early pruning* in [ABC+02a], *theory propagation* in [NOT05, NO05a] *theory-driven deduction* or $\mathcal{T}$*-deduction* in [BBC+05a, BBC+05b].

$c_1 : \{\neg B_1 \vee A_1\}$
$c_2 : \{\neg A_2 \vee B_2\}$
$c_3 : \{B_3 \vee A_2\}$
$c_4 : \{\neg B_4 \vee \neg B_5 \vee \neg A_1\}$
$c_5 : \{A_1 \vee B_3\}$
$c_6 : \{B_6 \vee B_7 \vee \neg A_1\}$
$c_7 : \{A_1 \vee B_8 \vee A_2\}$

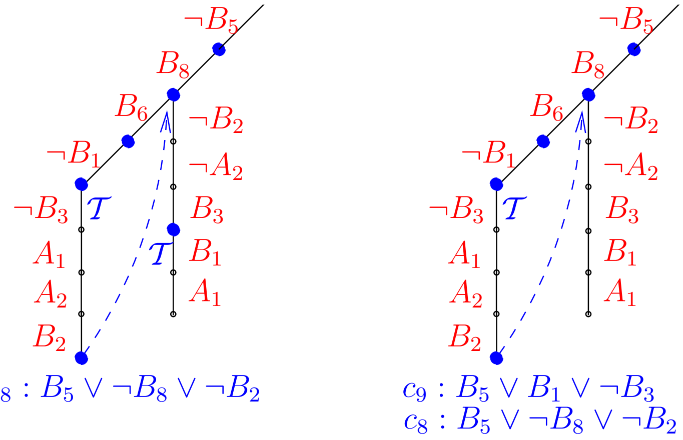


Figure 10. Left: Boolean search tree in the scenario of Example 6.3. Right: same situation, with learning of deduction clause c_9.

As $\mathcal{T}$-propagation is performed during the intermediate calls to $\mathcal{T}$-*solver*, it is always related to early pruning. Like with early pruning, $\mathcal{T}$-propagation can be applied either in a lazy way, before any new branching [ABC+02a, BBC+05a], or, more eagerly, every time a new $\mathcal{T}$-atom is added to the assignment (including those added by unit propagation) [ACG99, Tin02, BT03, GHN+04]. As with early pruning, the eager approach benefits of a more aggressive pruning, but pays for extra overhead.

More generally, there are different strategies by which $\mathcal{T}$-propagation can be applied. We will further discuss this point in §7.1.3.

6.5 $\mathcal{T}$-backjumping

This technique, which generalizes that of backjumping in standard DPLL, was introduced by [Hor98, PS98] for description logics; [WW99, dMRS02b, SBD02, ABC+02a, GHN+04, BBC+05a] adopted variants or improvements of the same idea for many other theories.

As hinted in §5.3, $\mathcal{T}$-backjumping is based on the assumption that, when $\mathcal{T}$-*solver* is invoked on a $\mathcal{T}$-inconsistent assignment μ, it is able to return also the conflict set $\eta \subseteq \mu$ causing the $\mathcal{T}$-unsatisfiability of μ. If so, $\mathcal{T}$-DPLL can use $\eta^p =_{def} \mathcal{T}2\mathcal{B}(\eta)$ as if it were a boolean conflict set to drive the backjumping mechanism of DPLL: the conflict clause $\neg\eta^p$ is added to φ^p (either temporarily or permanently, see §6.6) and the procedure backtracks to the branching point suggested by η^p.

Different backtracking strategies are possible. Older tools [Hor98, PS98, WW99] used to jump up to the most recent branching point s.t. at least one literal $l^p \in \eta^p$ is not assigned. Intuitively, all open subbranches departing from the current branch at a lower decision point contain η, so that there is no need to explore them; this allows for pruning all these subbranches from the search tree. (Notice that these strategies do not explicitly require adding the learned clause $\neg\eta^p$ to φ^p.) Most modern implementations [GHN+04, BBC+05a] inherit the backjumping mechanism of current DPLL tools described in §3.1: $\mathcal{T}$-DPLL

$$\begin{array}{ll}
c_1: & \{\neg B_1 \vee A_1\} \\
c_2: & \{\neg A_2 \vee B_2\} \\
c_3: & \{B_3 \vee A_2\} \\
c_4: & \{\neg B_4 \vee \neg B_5 \vee \neg A_1\} \\
c_5: & \{A_1 \vee B_3\} \\
c_6: & \{B_6 \vee B_7 \vee \neg A_1\} \\
c_7: & \{A_1 \vee B_8 \vee A_2\}
\end{array}$$

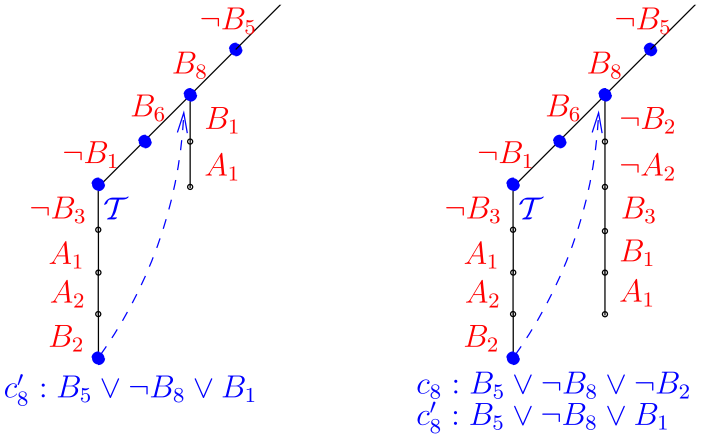


Figure 11. Left: Boolean search tree in the scenario of Example 6.4. Right: same situation, learning both c_8 and c'_8, as in Example 6.5.

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 $\neg l^d$ on $\neg\eta^p$. Intuitively, DPLL backtracks to the highest point where it would have done something different if it had known the conflict clause $\neg\eta^p$ in advance.

In substance, $\mathcal{T}$-backjumping differs from standard boolean backjumping only for the notion of conflict set used: whilst a boolean conflict set μ is an assignment which causes a propositional inconsistency if conjoined to φ (i.e, s.t. $\mu \wedge \varphi \models_p \bot$), a theory conflict set is a set of $\mathcal{T}$-literals which in *intrinsically inconsistent in* $\mathcal{T}$ (i.e, s.t. $\mu \models_{\mathcal{T}} \bot$), no matter φ.

As hinted in §5.3, it is also possible to have *mixed boolean+theory conflicts sets*, i.e., assignments η s.t. an inconsistency can be inferred from $\eta \wedge \varphi$ by means of a combination of boolean and theory reasoning (i.e., s.t. $\eta \wedge \varphi \models_{\mathcal{T}} \bot$) [NOT05]. Such conflict sets/clauses can be obtained starting from the theory conflicting clause $\mathcal{T}2\mathcal{B}(\neg\eta)$ by applying the backward-traversal of the implication graph described in §3.1, until one of the standard conditions (e.g., 1UIP) is achieved. In this process, a $\mathcal{T}$-propagation can be considered as a unit-propagation on the corresponding deduction clause (see §7.2.4).

Example 6.4. *The scenario depicted in Examples 5.2 and 6.3 represents a form of modern $\mathcal{T}$-backjumping, in which the conflict clause c_8 used is a pure $\mathcal{LA}(\mathbb{Q})$ conflict clause (i.e., $\mathcal{B}2\mathcal{T}(c_8)$ 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, $\mathcal{B}2\mathcal{T}(c'_8) = (3x_1 - x_3 \leq 6) \vee \neg(x_3 = 3x_5 + 4) \vee (2x_2 - x_3 > 2)$ is not $\mathcal{LA}(\mathbb{Q})$-valid.)

If so (Figure 11, left.), 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. Then, starting from $\{\neg B_5, B_8, B_1\}$, also A_1 is unit-propagated on c_1.

6.6 $\mathcal{T}$-learning

This technique, which generalizes that of learning in standard DPLL procedures, was introduced by [WW99] for linear arithmetic; [dMRS02b, SBD02, ABC$^+$02a, FJOS03] proposed the same idea for many other theories. Although $\mathcal{T}$-backjumping and $\mathcal{T}$-learning were originally conceived in different moments, in modern tools they are always related.

The rationale of $\mathcal{T}$-learning is the same as with standard boolean learning: when a conflict set η is found, the clause $\mathcal{T}2\mathcal{B}(\neg\eta)$ is added in conjunction to φ^p. Since then, $\mathcal{T}$-DPLL will never again generate any branch containing η. In fact, as soon as $|\eta| - 1$ literals in η are assigned to true, the remaining literal will be immediately assigned to false by unit propagation on $\mathcal{T}2\mathcal{B}(\neg\eta)$.

As with $\mathcal{T}$-backjumping, the only difference wrt. standard DPLL learning is in the notion of conflict set/clause used: both theory and mixed boolean+theory clauses can be used. Notice also that theory-conflict clauses and mixed boolean+theory conflict clauses can be computed and used contemporarily.

Example 6.5. *Consider the scenario of Example 6.4 and of Figure 11. If both c_8 and c'_8 were added to φ^p, then $\neg B_2$, $\neg A_2$ and B_3 would be unit-propagated on c_8, c_2 and c_3, and B_1 and A_1 would be unit-propagated on c'_8 and c_1, obtaining the same result as in Example 6.3, although here we do not need assuming that the deduction clause c_9 is learned.*

Notice that, whilst for $\mathcal{T}$-backjumping the best conflict set is that which forces the highest jump in the stack, for $\mathcal{T}$-learning the best conflict set is the one which causes the pruning of most future branches. In practice, these are the shortest conflict sets and those containing most atoms occurring in future branches (relevant atoms). To this extent, techniques to force $\mathcal{T}$-*solver* to return shorter conflict sets have been proposed in [WW00, ACGM04, dRS04] for different theories.

Like all learning techniques, $\mathcal{T}$-learning must be used with some care, because it may cause an explosion in size of φ. To avoid this, one has to introduce techniques for discarding learned clauses when necessary [BS97]. Luckily, by using one modern DPLL implementation from the shelf, one gets this feature for free.

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. [FJOS03] proposed an interesting generalization of $\mathcal{T}$-learning, in which at each consistency check more than one clause may be added, which may contain also new atoms. To overcome the consequent enlargement of the search space, they proposed to restrict splitting to the original atoms. [BBC$^+$05c, BBC$^+$06b] used a similar idea to improve the efficiency of Delayed Theory Combination (see §8.3); [BNOT06] used the same idea in the context of the splitting-on-demand approach (see §6.7); [WGG06] proposed similar ideas for a SMT($\mathcal{DL}$) tool, in which new atoms can be generated selectively according to an ad-hoc heuristic.

6.7 Splitting on demand

A noteworthy case of $\mathcal{T}$-learning in which clauses may contain new atoms is that performed in the *Splitting on demand* technique proposed in [BNOT06]. [33] 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.

Example 6.6. *[BNOT06]. Suppose an $\mathcal{AR}$-solver is given in input a set of literals containing a certain number of equalities in the form*

$$(read(write(a, i, e), j) = read(a, j)), \tag{15}$$

meaning that storing the value e in the i-th cell of array a does not affect the value of the j-th cell (see §4.2.6). (15) holds in two possible situations: when the indexes differ: $(i \neq j)$; when the value of the i-th location in a is already e: $read(a, i) = e$. Thus, for every literal like (15), the $\mathcal{AR}$-solver must consider the two alternatives separately. With splitting on demand, instead, it may return (the boolean abstraction of) the clause

$$(read(write(a, i, e), j) = read(a, j)) \rightarrow (\neg(i = j) \vee (read(a, i) = e)) \tag{16}$$

to the DPLL *solver, forcing it to split on one of the last two literals. Notice that the latter literals do not necessarily occur in the original formula.*

Notice that, to this extent, a $\mathcal{T}$*-solver* invoked on an assignment μ can produce also clauses in the form $\mathcal{T}2\mathcal{B}(\mu^* \rightarrow \bigvee_i e_i)$ s.t. $\mu^* \subseteq \mu$ and the e_i's are interface equalities.

[33] A preliminary form of this technique was briefly described in §3.5.1 of [Bar03] and is implemented in CVC and CVCLite [CVCa, CVCb].

6.8 Clustering

This technique was proposed in [BBC+05a] for of $\mathcal{EUF}$ and $\mathcal{LA}$, and was implicit in DPLL-based procedures for modal and description logics (see, e.g., [GS00]).

At the beginning of the search, the set of $\mathcal{T}$-atoms of φ is partitioned into a set of disjoint *clusters* $C_1 \ldots C_k$, s.t. atoms which do not interfere to each-other's $\mathcal{T}$-satisfiability belong to different clusters. (I.e., two atoms belong to the same cluster if they share a variable.) Consequently, every assignment μ can be partitioned into k disjoint sub-assignments μ_i, one for each cluster, so that μ is $\mathcal{T}$-satisfiable iff each μ_i is. Based on this idea, instead of having a single, monolithic solver, $\mathcal{T}$*-solver* is instantiated (up to) k different times: each is responsible for the handling of the reasoning within a single cluster.

The advantage of this "divide-and-conquer" approach is manifold. First, k solvers running on k disjoint problems are typically faster then running one solver monolithically on the union of the problems. Second, the solvers are activated in a lazy way: if one returns Unsat, there is no need to call the others. Third, the construction of smaller conflict sets becomes easier, and this may result in significant gain in the overall search.

6.9 Reduction of assignments to prime implicants

The following technique was proposed for $\mathcal{DL}$ in [ACGM04].

Let μ be an assignment propositionally satisfying the input formula φ. Sometimes μ may not be a *prime implicant* for φ, that is, some of the literals in μ may be unnecessary to propositionally satisfy φ (i.e., $\mu \setminus \{l\} \models_p \varphi$ for some $l \in \mu$). The typical case is when more than two literals in the same clause are satisfied by μ. Thus $\mathcal{T}$*-solver* may eliminate such literals l's from μ.

There are a couple of 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, $\mathcal{T}$-DPLL can stop. Second, if both μ' and μ are $\mathcal{T}$-unsatisfiable, 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.

Example 6.7. *Consider the following scenario with the $\mathcal{T}$-formula φ in Example 5.2: $\mathcal{T}$-*DPLL *generates the following assignment μ, which propositionally satisfies φ:*

$$\{\neg(2x_2 - x_3 > 2), \neg A_2, (3x_1 - 2x_2 \leq 3), \neg(3x_1 - x_3 \leq 6), \neg A_1, (x_3 = 3x_5 + 4)\}.$$

If $\mathcal{T}$-solver is invoked on μ without reduction, then it will return Unsat *due to the conflict set $\{\neg(2x_2 - x_3 > 2), (3x_1 - 2x_2 \leq 3), \neg(3x_1 - x_3 \leq 6)\}$. We notice that the literal $\neg(3x_1 - x_3 \leq 6)$ is unnecessary for satisfying φ, because the 4th clause is satisfied also by $\neg A_1$. Thus, if we drop it from μ, we obtain a $\mathcal{T}$-satisfiable assignment μ' s.t. $\mu' \models_p \varphi$, so that $\mathcal{T}$-*DPLL *can return* Sat *without further backtracking.*

6.10 Pure-literal filtering

This technique, which we call *pure-literal filtering*,[34.] was implicitly proposed by [WW99] and then generalized by [GGT01, ABC+02a, BBC+05b].

[34.] Also called *triggering* in [ABC+02a].

The idea is that, 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*. (The correctness and completeness of this process is a consequence of Proposition 2.4 in §2.2.) Moreover, 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 by $\mathcal{T}$-*solver*, so that to avoid the $\mathcal{T}$-propagation of literals which have been filtered away.

We notice first that pure-literal filtering has the same two benefits described for reduction to prime implicants in §6.9. Moreover, this technique is particularly useful in some situations. For instance, in $\mathcal{DL}(\mathbb{Z})$ and $\mathcal{LA}(\mathbb{Z})$ many solvers 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 is done either off-line, by rewriting all equalities [resp. disequalities] into a conjunction of inequalities [resp. a disjunction of strict inequalities], or on-line, at each call to $\mathcal{T}$-*solver*.) This causes an enlargement of the search, because the two disjuncts must be investigated separately.

However, in many problems it is very frequent that many equalities $(t_1 = t_2)$ occur with positive polarity only. If so, pure-literal filtering avoids adding $(t_1 \neq t_2)$ to μ when $\mathcal{T}2\mathcal{B}((t_1 = t_2))$ is assigned to false by $\mathcal{T}$-DPLL, so that no split is needed [ABC+02a].

6.11 $\mathcal{T}$-deduced-literal filtering

This technique has been proposed by [BBC+05b, CM06b] to further reduce the amount of $\mathcal{T}$-literals given to $\mathcal{T}$-*solver*.

If the literal l is $\mathcal{T}$-propagated by $\mathcal{T}$-*solver*, or if l is unit-propagated on a learned $\mathcal{T}$-valid clause $C =_{def} (l_1 \wedge \dots \wedge l_n) \rightarrow l$ [35.] s.t. $\{l_1, ..., l_n\} \subseteq \mu$, then there is no need to pass l to $\mathcal{T}$-*solver*, because μ is $\mathcal{T}$-equisatisfiable to $\mu \cup \{l_i\}$. (In order to detect these cases, $\mathcal{T}$-valid clauses can be marked with a flag when they are learned.) As with pure-literal filtering, if $\mathcal{T}$-propagation is implemented, then the filtered literal l must be dropped also from the list of literals which can be $\mathcal{T}$-deduced by $\mathcal{T}$-*solver*.

Notice that combining different filtering methods requires some care because, in order to safely apply $\mathcal{T}$-deduced-literal filtering to l, all literals $l_1, \dots, l_n$ must have been explicitly passed to $\mathcal{T}$-*solver* (i.e., they must not have been filtered).

[35.] I.e., a $\mathcal{T}$-valid clause C s.t. $\mathcal{T}2\mathcal{B}(C)$ has been added to φ^p via static learning (§6.2), $\mathcal{T}$-propagation (§6.4) or $\mathcal{T}$-learning (§6.6).

7. Discussion

For most of the techniques described in §6, the applicability and the benefits of their application depend on many factors.

First, some techniques require as necessary conditions some of the specific features of $\mathcal{T}$*-solver* described in §4.1. E.g., you cannot use $\mathcal{T}$-backjumping and $\mathcal{T}$-learning if your solver is not capable of producing good-enough conflict sets; the benefits of $\mathcal{T}$-propagation depend only on the deduction capabilities of $\mathcal{T}$*-solver*, and on their efficiency.

Second, the effects of the different integration techniques are not mutually independent, and are thus difficult to evaluate as stand-alone ones. Some techniques can share part of their benefits. (E.g., in Examples 6.3 and 6.5, the enhanced versions of $\mathcal{T}$-propagation and $\mathcal{T}$-learning may produce similar effects.) Some other can interact negatively. (E.g., pure-literal filtering can reduce the pruning power of early pruning, because it may drop literals causing $\mathcal{T}$-inconsistencies, and thus some pruning of the boolean search.) Some other techniques are pairwise related. (E.g., $\mathcal{T}$-propagation is associated with early pruning; both $\mathcal{T}$-backjumping and $\mathcal{T}$-learning benefit of the conflict sets generated by $\mathcal{T}$*-solver*, and are thus implemented together in most solvers.)

Third, and most important, the benefits of many integration techniques, like early pruning and $\mathcal{T}$-propagation, depend on the theory $\mathcal{T}$ addressed and, in particular, on the tradeoff between the cost of $\mathcal{T}$-solving (and $\mathcal{T}$-propagation) and the benefits of reducing boolean search space. Notice that "reducing the boolean search" means not only reducing the time spent on boolean reasoning but also, and much more importantly, reducing the size of the boolean search tree, and consequently the number of calls to $\mathcal{T}$*-solver*. As discussed in §4, for some theories (e.g., $\mathcal{EUF}$ and $\mathcal{DL}$) $\mathcal{T}$-solving is relatively cheap, so that it is typically worth performing extra calls to $\mathcal{T}$*-solver* if this allows for pruning the boolean search; for some other theories instead (e.g., $\mathcal{LA}(\mathbb{Z})$ and $\mathcal{BV}$) $\mathcal{T}$-solving may be very expensive, so that trading $\mathcal{T}$*-solver* calls for boolean-search reduction is not always a good deal.

7.1 Guidelines and tips

With very few exceptions, there is no universal, theory-independent recipe for choosing the right integration techniques to apply. In this section we provide some guidelines and tips, based on both theoretical analysis and practical experience.

7.1.1 Some general guidelines

There are very few suggestions which can be given for most or even all situations, no matter which theory $\mathcal{T}$ or $\mathcal{T}$*-solver* we are dealing with.

- *Normalize* $\mathcal{T}$*-atoms in the input formulas* according to the lines of §6.1. This is a very cheap process and may avoid lots of branches and useless calls to $\mathcal{T}$*-solver*.
- *Use some form of early pruning.* In fact, $\mathcal{T}$-unsatisfiable branches are typically much bigger than the conflict sets causing their unsatisfiability, so that using no EP techniques may result into a huge number of branches and useless calls to $\mathcal{T}$*-solver*.
- *Use* $\mathcal{T}$*-backjumping and* $\mathcal{T}$*-learning.* In fact, if these techniques are not used, many branches containing the same conflict sets can be enumerated by $\mathcal{T}$-DPLL, causing

up to huge amounts of useless calls to $\mathcal{T}$*-solver*. Despite potential problems of memory blowup, it is a common experience of the authors of state-of-the-art solvers that both techniques drastically improve the performance when applied.

All state-of-the-art lazy SMT solvers we are aware of comply to the suggestions above.

7.1.2 Offline vs. online integration

As described in §5.2, in the offline integration schema the SAT solver is restarted from scratch each time on augmented boolean formulas. On the one hand, this allows for using a SAT solver from the shelf with minimal or no modification to its source code, whilst the online approach requires a tighter integration of the source codes of the SAT solver and $\mathcal{T}$*-solver*. On the other hand, as remarked in [FJOS03], in the offline approach each call to the SAT solver may repeat some or much of the search already performed by previous calls. In the online approach, instead, after each call to $\mathcal{T}$*-solver* the boolean search recovers from the point it was interrupted, without redoing any work. Moreover, notice that in the offline approach (see Figure 6) it is strictly necessary to keep the theory-conflict clauses learned in order to guarantee the completeness of the approach, whilst in the online approach (see Figure 7) $\mathcal{T}$-learning is just a technique for improving efficiency, so that theory-conflict clauses can be learned and discharged at will. To this extent, [FJOS03] showed in an empirical test a relevant performance superiority of the online approach wrt. the offline one. [36.]

The effect of passing from the "naive" offline SAT solving of §5.2 to its more effective version exploiting conflict sets is analogous to that of using $\mathcal{T}$-backjumping and $\mathcal{T}$-learning with online SAT solving, and the effect of the "eager notification" version of the offline approach described in by [BDS02a] is that of (eager) early pruning. The last improvement, however, requires a tighter integration of the source code of the SAT solver and $\mathcal{T}$*-solver*.

In general, the choice between the offline and online schemata relies on a tradeoff between efficiency and the effort of implementation, in particular that of modifying and integrating with the source code of a SAT solver. Offline integration is suitable for prototyping, whilst online integration is recommendable for building more efficient and stable tools.

7.1.3 To $\mathcal{T}$-propagate or not to $\mathcal{T}$-propagate?

As hinted in §4.1, for some theories (e.g., $\mathcal{EUF}$, $\mathcal{DL}$) $\mathcal{T}$*-solvers* with powerful deduction capabilities are available, so that $\mathcal{T}$-propagation can be implemented in very efficient ways [GHN$^+$04, NO05a, DdM06]. In other cases, however, things are not so nice and we have to seriously take into account the tradeoff between the actual benefits of reducing the boolean search space and the overhead costs introduced.

Many different application strategies for $\mathcal{T}$-propagation are possible. The main choices one has to deal with are the following:

[36.] Notice that one can pass from the offline to the online schema by storing the status of the SAT solver (e.g., stack, implication graph) and retrieve it at the next restart. We consider this as a variant implementation of the online schema, as it shares the same advantages (no repeated boolean search, no need of keeping all the theory-conflict clauses learned) and drawbacks (requires significant modifications to the source code of the SAT solver).

- apply $\mathcal{T}$-propagation exhaustively [GHN+04, NO05a] (i.e., always try to deduce as many literals as possible) or more selectively [ABC+02a, WGG06] (e.g., deduce only easy-to-deduce literals). The former can prune the boolean search space more aggressively at the expense of a bigger computational effort for $\mathcal{T}$*-solver*, and vice versa;

- privilege $\mathcal{T}$-propagation wrt. unit propagation [NO05a] (e.g., apply $\mathcal{T}$-propagation until possible, and then apply unit propagation), or vice-versa [BBC+05a]. The former trades more $\mathcal{T}$*-solver* work for less BCP effort, and vice versa;

- learn deduction clauses lazily [NO05a] (e.g., only when strictly necessary to backward-traversing the implication graph) or eagerly [BBC+05a] (e.g., learn either temporarily or permanently the deduction clauses at every $\mathcal{T}$-propagation performed). Again, the former trades more $\mathcal{T}$*-solver* work for less boolean reasoning effort, and vice versa;

- in case of layered $\mathcal{T}$*-solvers* (§4.3), how to interleave the hierarchical calls to the different layers and the $\mathcal{T}$-propagation of the literals $\mathcal{T}$-deduced by each layer. E.g., when an unassigned literal l is $\mathcal{T}$-deduced at level L_i, it may either be returned to the SAT solver, so that to be unit-propagated, or be passed down to layer L_{i+1}, so that to produce more information for the lower layers. These issues, combined with the role of $\mathcal{T}$-deduced-literal filtering (§6.11), have been investigated in detail in [CM06b].

In the Abstract DPLL Modulo Theories framework of §3.2 and §5.4, the different alternatives described above may correspond to different strategies by which to apply the Theory Propagate rule wrt. other rules like Decide, Unit Propagate, ($\mathcal{T}$)-Learn and ($\mathcal{T}$-)Discharge.

7.2 Problems of using modern DPLL in SMT

As pointed out in §3, a SAT solver is very different from an Enumerator, so that what makes a DPLL solver efficient is not enough for making an SMT tool efficient, and what causes only an irrelevant overhead for SAT may be a major source of inefficiency in SMT. Thus, we overview a list of problems one may encounter while implementing a lazy SMT tool on top of a modern DPLL implementation, and propose some solutions.

7.2.1 Generating partial assignments

All SMT schemata of §5 are based on the statement (Property 2.3) that φ is $\mathcal{T}$-satisfiable iff a $\mathcal{T}$-satisfiable *partial* assignment μ propositionally satisfies φ. (Notice that it is enough to assume that μ propositionally satisfies the *original* formula φ, i.e., there is no need that μ satisfies also the learned clauses. [37.])

In §3.1 we remarked that, due to the two-watched-literal scheme [MMZ+01], in modern implementations DPLL returns Sat only when all variables are assigned truth values, thus returning *total* assignments, even though the formulas can be satisfied by *partial* ones. Thus, when a partial assignment μ is found which satisfies φ, this causes an unnecessary sequence of decisions and unit-propagations for assigning the remaining variables.

[37.] Every clause C which has been learned is such that $\varphi \models_{\mathcal{T}} C$, so that, if μ is $\mathcal{T}$-satisfiable and $\mu \models_p \varphi$, then φ is $\mathcal{T}$-satisfiable by Prop. 2.3, and hence $\varphi \wedge C$ is $\mathcal{T}$-satisfiable. Thus, there is no need to check also that $\mu \models_p C$.

In SAT, this scenario causes no extra boolean search because every extension of μ satisfies propositionally φ, so that the overhead introduced is negligible.

In SMT, instead, many total assignments extending μ may be $\mathcal{T}$-inconsistent even though μ is $\mathcal{T}$-consistent, so that many useless boolean branches and calls to $\mathcal{T}$-solvers may be required (up to $2^{|Atoms(\varphi)|-|\mu|}-1$). If early pruning (§6.3) is implemented, then the problem happens at most once in the search (when a $\mathcal{T}$-satisfiable satisfying partial assignment is found); however, if weakened early pruning (§6.3.2) is implemented, then the problem arises also when a partial assignment is generated which is unsatisfiable in $\mathcal{T}$ (e.g., $\mathcal{LA}(\mathbb{Z})$) but satisfiable in the "weaker" theory $\mathcal{T}$' (e.g., $\mathcal{LA}(\mathbb{Q})$). The problem is relevant also in Delayed Theory Combination (DTC) [BBC$^+$05c], as it will be made clear in §8.3.

Example 7.1. *Consider the simple $\mathcal{LA}(\mathbb{Z})$-formula*

$$\begin{array}{lll} c_1: & \varphi = & \{(x_1<1)\vee(x_2>1)\} \\ c_2: & & \{(x_2<2)\vee(x_1>0)\} \end{array} \qquad \begin{array}{ll} \varphi^p = \mathcal{T}2\mathcal{B}(\varphi) = & \{B_1\vee B_2\} \\ & \{B_3\vee B_4\}. \end{array} \tag{17}$$

*Suppose that $\mathcal{T}$-*DPLL* has generated the (intermediate) partial assignment $\mu^p =_{def} \{B_1, B_3\}$. As $\mu =_{def} \mathcal{B}2\mathcal{T}(\mu^p)$ is $\mathcal{LA}(\mathbb{Z})$-consistent, this should be enough to state that φ is $\mathcal{LA}(\mathbb{Z})$-satisfiable. However, if $\mathcal{T}$-*DPLL* enumerates total assignments only, then it may have to generate and check up to four total assignments extending μ before finding the only $\mathcal{LA}(\mathbb{Z})$-satisfiable one, i.e., $\mathcal{B}2\mathcal{T}(\{B_1, B_3, \neg B_2, \neg B_4\})$.*

*Suppose instead that $\mathcal{T}$-*DPLL* has generated the partial assignment $\mu^p =_{def} \{B_1, B_4\}$. As $\mu =_{def} \mathcal{B}2\mathcal{T}(\mu^p)$ is $\mathcal{LA}(\mathbb{Z})$-inconsistent, early-pruning would cause a $\mathcal{LA}(\mathbb{Z})$-solver call on μ, forcing $\mathcal{T}$-*DPLL* to backtrack. With weakened early pruning s.t. $\mathcal{T}$' is $\mathcal{LA}(\mathbb{Q})$, instead, the weaker $\mathcal{LA}(\mathbb{Q})$-solver would be invoked on μ. As μ is $\mathcal{LA}(\mathbb{Q})$-consistent, this would cause enumerating up to the four total assignments extending μ, which would be found $\mathcal{LA}(\mathbb{Z})$-inconsistent by the $\mathcal{LA}(\mathbb{Z})$-solver in the complete calls.*

In order to overcome these problems, it is sufficient to implement some device monitoring the satisfaction of all original clauses in φ. Although this may cause some overhead in handling the boolean component of reasoning, this may reduce the overall boolean search space and the number of calls to $\mathcal{T}$-*solver* consequently, in particular when weakened early pruning or DTC are used.

7.2.2 Avoiding ghost literals

As stated in §3.1, in modern DPLL tools `decide_next_branch` selects a new literal according to a score which is updated only at the end of every branch, and is never changed until the end of the next branch. Consequently, `decide_next_branch` may select also literals which occur only in clauses which have already been satisfied (which we call *ghost literals*).

In SAT, the selection of ghost literals in the assignment μ causes no extra boolean search, because it interferes neither with BCP nor with the detection of (un)satisfiability and construction of the implication graph, so that the overhead introduced is negligible.

In SMT, instead, the presence of ghost $\mathcal{T}$-literals in μ may affect the $\mathcal{T}$-satisfiability of μ, forcing unnecessary backtrackings.

Example 7.2. *Suppose that $\mathcal{T}$-*DPLL *implements early pruning and does not implements $\mathcal{T}$-propagation, and consider again the simple $\mathcal{LA}(\mathbb{Z})$-formulas of Example 7.1:*

$$\begin{array}{lrll} c_1: & \varphi = & \{(x_1 < 1) \vee (x_2 > 1)\} & \varphi^p = \mathcal{T}2\mathcal{B}(\varphi) = & \{B_1 \vee B_2\} \\ c_2: & & \{(x_2 < 2) \vee (x_1 > 0)\} & & \{B_3 \vee B_4\}, \end{array} \quad (18)$$

and the intermediate assignment $\mu^p =_{def} \{B_1\}$. Suppose `decide_next_branch` *selects the literals in lexicographic order $B_1 ... B_4$ without detecting ghost literals: it selects the ghost literal B_2, causing the useless call to $\mathcal{T}$-solver on $\mathcal{B}2\mathcal{T}(\{B_1, B_2\})$, and hence B_3, causing the call of $\mathcal{T}$-solver on $\mathcal{B}2\mathcal{T}(\{B_1, B_2, B_3\})$, which returns* Unsat *and forces backtracking on the conflict clause $\neg B_2 \vee \neg B_3$. If* `decide_next_branch` *realizes that B_2 is a ghost literal and skips it, it will select B_3 instead, and find the $\mathcal{T}$-satisfiable assignment $\mathcal{B}2\mathcal{T}(\{B_1, B_3\})$, saving one call to $\mathcal{T}$-solver and one T-backjumping and $\mathcal{T}$-learning step.*

In order to overcome these problems, it is sufficient to implement some device monitoring the satisfaction of the (original) clauses in φ in which the selected literal occurs. Again, although this may cause some overhead in handling the boolean component of reasoning, this may significantly reduce the overall boolean search space and the number of calls to $\mathcal{T}$-*solver* consequently.

7.2.3 Drawbacks of modern $\mathcal{T}$-backjumping

As stated in §6.5, older forms of $\mathcal{T}$-backjumping [Hor98, PS98, WW99] used to jump up to the most recent branching point s.t. at least one literal $l^p \in \eta^p$ is not assigned, η being the theory conflict set; most modern implementations instead (e.g., [GHN$^+$04, BBC$^+$05a]) adopt or inherit the backjumping mechanism of modern DPLL tools, so that $\mathcal{T}$-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 $\neg l^d$ on $\neg\eta^p$. In the latter case, the backtrack mechanism jumps higher.

We notice that, whilst in SAT this is typically a good feature, in SMT this is not necessarily the case: it may often happen that, after $\mathcal{T}$-backjumping and unit-propagating on the learned clause, $\mathcal{T}$-DPLL redoes the same decisions and unit propagations as in the previous branch, until it reaches the same status he would have reached with a "lower" jump (like, e.g., with the "old" backjumping approach). In SAT, the overhead introduced by this extra work is negligible. In SMT, instead, it may cause extra useless calls to $\mathcal{T}$-*solver* and $\mathcal{T}$-propagations, which may be very expensive.

Example 7.3. *Consider the case where the formula φ of Example 5.2 is extended with many other clauses which contain no atoms occurring in $c_1, ..., c_7$. We assume a scenario similar to that of Example 5.2: $\mathcal{T}$-*DPLL *selects $\{\neg B_5, B_8, B_6, l_1, ..., l_n, \neg B_1\}$, it $\mathcal{T}$-propagates $\neg B_3$ and unit-propagates, A_1, A_2, B_2, causing a conflict and learning the conflict clause c_8. (Here "$l_1, ..., l_n$" is a possibly-big sequence of assignments on atoms not occurring in $c_1, ..., c_7$, which we assume do not interact with the other assignments, but may cause a significant amount of calls to $\mathcal{T}$-solver.)*

*If $\mathcal{T}$-*DPLL *adopted an old-style $\mathcal{T}$-backjumping strategy, it would jump up above $\neg B_1$, unit-propagate $\neg B_2$ on c_8 and hence $\neg A_2$ and B_3 on c_2 and c_3. (Figure 12 left.)*

*With a modern $\mathcal{T}$-backjumping strategy, instead, $\mathcal{T}$-*DPLL *would jump up above B_6 and hence unit-propagate $\neg B_2$, $\neg A_2$ and B_3 on c_8, c_2 and c_3. (Figure 12 right.) By construction,*

$$
\begin{array}{rl}
c_1: & \{\neg B_1 \vee A_1\} \\
c_2: & \{\neg A_2 \vee B_2\} \\
c_3: & \{B_3 \vee A_2\} \\
c_4: & \{\neg B_4 \vee \neg B_5 \vee \neg A_1\} \\
c_5: & \{A_1 \vee B_3\} \\
c_6: & \{B_6 \vee B_7 \vee \neg A_1\} \\
c_7: & \{A_1 \vee B_8 \vee A_2\} \\
\ldots: & \langle\ \textit{other clauses}\ \rangle
\end{array}
$$

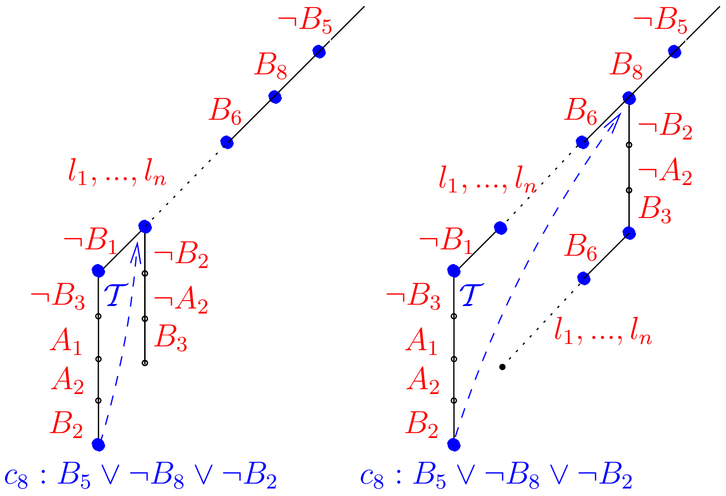


Figure 12. Boolean search trees in the scenarios of Example 7.3. Left: "old-style" $\mathcal{T}$-backjumping. Right: modern $\mathcal{T}$-backjumping.

*$\neg B_2$, $\neg A_2$ and B_3 do not interfere with the $\mathcal{T}$-consistency of B_5, B_8, B_6. If the presence of $\neg B_2$, $\neg A_2$ and B_3 does not influence the literal selection heuristic, then $\mathcal{T}$-*DPLL *may select B_6 and may redo from scratch the assignments of the previous branch, redoing the same decisions, unit-propagations and calls of the $\mathcal{T}$-solver than in the previous branch, until it gets to the same point it would have reached if it would have jumped up to $\neg B_1$, as in Figure 12 left.*

7.2.4 Implementing $\mathcal{T}$-propagation

As remarked in [GHN+04], implementing $\mathcal{T}$-propagation on top of a modern DPLL algorithm can be tricky, because, by construction, the implication graph in DPLL described in §3.1 does not keep track of $\mathcal{T}$-propagations. Therefore, if a $\mathcal{T}$-propagated literal l is encountered during the backward traversal of the implication graph (e.g., when building a boolean conflict set), the information on the $\mathcal{T}$-propagation $\eta \models_{\mathcal{T}} l$ must be recovered in order to complete the process. In [GHN+04], the set of antecedents η of the deduction are computed on demand only in these situations. In [BBC+05a], the deduction clause $\mathcal{T}2\mathcal{B}(\eta \rightarrow l)$ is always computed and added to φ^p, either temporarily or permanently, and the process handles a $\mathcal{T}$-propagated literal as if it were the result of a unit-propagation on the deduction clause.

7.2.5 DPLL Branching heuristics for SMT

In general, good literal-selection heuristics for "pure" DPLL are not necessary good for $\mathcal{T}$-DPLL-like procedures as well. First, a heuristic which is good to search for *one* assignment is not necessary good for enumerating up to a complete collection of them. More importantly, traditional DPLL heuristics are not "*theory-aware*", in the sense that they do not take into account the $\mathcal{T}$-semantics of the literals.

So far there seem to be no really-satisfactory proposal in the direction of building theory-aware heuristics, and most tools simply use standard DPLL heuristics. One of the main reason for this fact may be that the problem is trickier then one would expect: in order to be effective, a theory-aware heuristic should not only take into account the $\mathcal{T}$-semantics of the literal chosen, but also that of *all* the literals that are assigned as a deterministic consequence (unit propagation, $\mathcal{T}$-propagation) of that choice. For instance, with some problems it is often the case that boolean literals are better choices than others, because they cause longer chains of unit propagations [ACKS02].

Example 7.4. *Consider the $\mathcal{T}$-formula φ in Example 5.2. Branching on the boolean literal $\neg A_1$ causes the assignment of $\neg(2x_2 - x_3 > 2)$ and $(3x_1 - 2x_2 \leq 3)$ by unit propagation and hence of $(3x_1 - x_3 \leq 6)$ by $\mathcal{T}$-propagation (rows 1, 5 and hence 3).*

Unfortunately, the whole sets of deterministic consequences of a branch choice are difficult to predict a priori. One possible direction, is to perform all propagations explicitly on all the candidate literals in turn as in [LA97], but this is likely to be extremely expensive. Another direction is, as with the pure boolean case, to provide to $\mathcal{T}$-DPLL solver a list of "privileged" variables on which to branch on first [ACKS02].

8. Lazy SMT for combinations of theories

In many practical applications of SMT, the theory $\mathcal{T}$ is a combination of two (or more) theories $\mathcal{T}_1$ and $\mathcal{T}_2$. (As stated in §2.1.1, we restrict our interest on signature-disjoint stably infinite theories with equality.) For instance, an atom of the form $f(x+3y) = g(2x-y)$, that combines uninterpreted function symbols (from $\mathcal{EUF}$) with arithmetic functions (from $\mathcal{LA}(\mathbb{Q})$), could be used to naturally model in a uniform setting the abstraction of some functional blocks in an arithmetic circuit. In the following, we discuss the main approaches to the development of lazy $SMT(\mathcal{T})$ tools where $\mathcal{T}$ is the combination of two or more different theories. For the sake of simplicity and w.l.o.g., we also assume that the input formulas are pure (see §2.1.1), although this assumption is not strictly necessary [BDS02b].

8.1 Ackermann's expansion

When one of the theories $\mathcal{T}_i$ is $\mathcal{EUF}$, one possible approach to the $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$ problem is to eliminate uninterpreted function symbols by means of Ackermann's expansion [Ack54] so that to obtain an $SMT(\mathcal{T})$ problem with only one theory. The method works by replacing every function application occurring in the input formula φ with a fresh variable and then adding to φ all the needed functional congruence constraints. The new formula φ' obtained is equisatisfiable with φ, and contains no uninterpreted function symbols.

First, each distinct function application $f(x_1, \ldots, x_n)$ is replaced by a fresh variable $v_{f(x_1,\ldots,x_n)}$. Then, for every pair of distinct applications of the same function, $f(x_1, \ldots, x_n)$ and $f(y_1, \ldots, y_n)$, a congruence constraint

$$\bigwedge_{i=1}^{arity(f)} (ack(x_i) = ack(y_i)) \rightarrow (v_{f(x_1,\ldots,x_n)} = v_{f(y_1,\ldots,y_n)}), \qquad (19)$$

is added, where ack is a function that maps each function application $g(z_1, \ldots, z_n)$ into the corresponding variable $v_{g(z_1,\ldots,z_n)}$, each variable into itself and is homomorphic wrt. the interpreted symbols. The atom $(ack(x_i) = ack(y_i))$ is not added if the two sides of the equality are syntactically identical; if so, the corresponding implication in (19) is dropped.

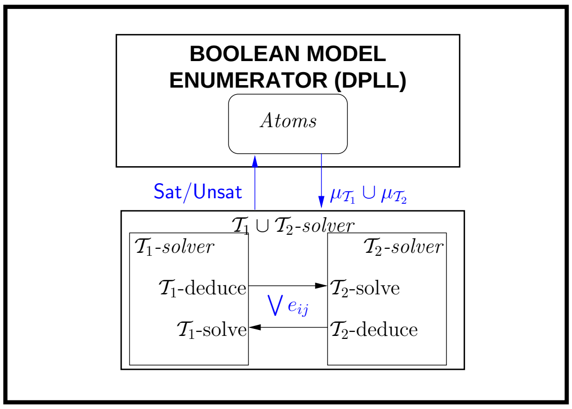
Example 8.1. *Consider the following $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$ pure formula φ:*

$$\begin{array}{ll} \mathcal{EUF}: & (v_3 = h(v_0)) \wedge (v_4 = h(v_1)) \wedge (v_6 = f(v_2)) \wedge (v_7 = f(v_5)) \wedge \\ \mathcal{LA}(\mathbb{Q}): & (v_0 \geq v_1) \wedge (v_0 \leq v_1) \wedge (v_2 = v_3 - v_4) \wedge (RESET_5 \rightarrow (v_5 = 0)) \wedge \\ Both: & (\neg RESET_5 \rightarrow (v_5 = v_8)) \wedge \neg(v_6 = v_7). \end{array} \qquad (20)$$

By replacing every function application with a fresh variable, and adding all the functional consistency constraints, we obtain the SMT($\mathcal{LA}(\mathbb{Q})$) formula:

$$\begin{array}{ll} \mathcal{LA}(\mathbb{Q}): & (v_3 = v_{h(v_0)}) \wedge (v_4 = v_{h(v_1)}) \wedge (v_6 = v_{f(v_2)}) \wedge (v_7 = v_{f(v_5)}) \wedge \\ & (v_0 \geq v_1) \wedge (v_0 \leq v_1) \wedge (v_2 = v_3 - v_4) \wedge (RESET_5 \rightarrow (v_5 = 0)) \wedge \\ & (\neg RESET_5 \rightarrow (v_5 = v_8)) \wedge \neg(v_6 = v_7) \wedge \\ Congruence: & ((v_0 = v_1) \rightarrow (v_{h(v_0)} = v_{h(v_1)})) \wedge ((v_2 = v_5) \rightarrow (v_{f(v_2)} = v_{f(v_5)})), \end{array} \qquad (21)$$

which can be solved by a SMT($\mathcal{LA}(\mathbb{Q})$) solver. Notice that four new equalities and two congruence constraints have been added.



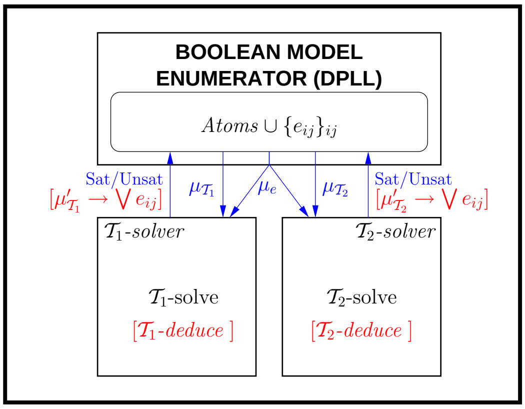


Figure 13. $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$ via NO.

Figure 14. $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$ via DTC.

Notice that in Example 8.1 we have considered a pure formula φ, which might be the result of purifying some non-pure formula φ'. If so, applying Ackermann's expansion directly to φ' might result into a more compact formula than (21).

An interesting variant of Ackermann's expansion has been introduced for eager encodings (see §9.3) in [BGV99]. This method differs from that in [Ack54] because it replaces each term with a nested series of if-then-else constraints. This allows for effectively exploiting the presence of positive equalities. We refer the reader to [BGV99] for details.

8.2 Nelson-Oppen Combination

The first general approach we discuss is based on combining two $\mathcal{T}_i$-*solvers* into one solver $\mathcal{T}_1 \cup \mathcal{T}_2$-*solver* by means of the Nelson-Oppen combination schema [NO79] or of its variant due to Shostak [Sho84] [38] (*NO* hereafter). The $\mathcal{T}_1 \cup \mathcal{T}_2$-*solver* is then integrated with DPLL as described in §5. Here we provide only a high-level description of this approach. The reader may refer, e.g., to [NO79, Sho79, FORS01, BDS02b, SR02, MZ03] for more details.

A basic architectural schema of $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$ via NO is described in Figure 13. Let $\mathcal{T}_1$, $\mathcal{T}_2$ be two signature-disjoint stably-infinite theories and let $\mathcal{T}_1$-*solver*, $\mathcal{T}_2$-*solver* be their respective theory solvers s.t. each of them is e_{ij}-deduction complete (see §4.1.5). The combined decision procedure $\mathcal{T}_1 \cup \mathcal{T}_2$-*solver* is based on a structured interchange of (disjunction of) interface equalities which are inferred by either $\mathcal{T}_i$-*solver* and then propagated to the other, until convergence is reached.

In the case of convex theories, the two $\mathcal{T}_i$-*solvers* exchange single interface equalities. We illustrate first the behavior of the NO schema within a SMT tool in the case of convex theories in the following example. Hereafter, we denote with $\mu_{\mathcal{T}_i}$ the subassignment of μ containing only i-pure literals.

[38] Nowadays there seems to be a general consensus on the fact that Shostak's method should not be considered as an independent combination method, rather as a collection of ideas on how to implement Nelson-Oppen's combination method efficiently [RS01, BDS02b, DNS05].

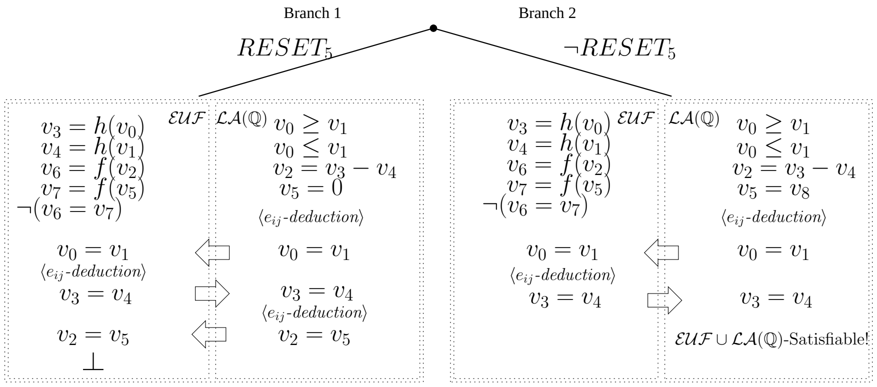


Figure 15. Search tree for the formula of Example 8.2

Example 8.2. *[BCF+06b] Consider the following $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$ formula φ (see Fig. 15)*

$$\begin{array}{ll}\mathcal{EUF}: & (v_3 = h(v_0)) \wedge (v_4 = h(v_1)) \wedge (v_6 = f(v_2)) \wedge (v_7 = f(v_5)) \wedge \\ \mathcal{LA}(\mathbb{Q}): & (v_0 \geq v_1) \wedge (v_0 \leq v_1) \wedge (v_2 = v_3 - v_4) \wedge (RESET_5 \rightarrow (v_5 = 0)) \wedge \\ Both: & (\neg RESET_5 \rightarrow (v_5 = v_8)) \wedge \neg(v_6 = v_7).\end{array} \tag{22}$$

v_0, v_1, v_2, v_3, v_4, v_5 are interface variables, v_6, v_7, v_8 are not. (Thus, e.g., $(v_0 = v_1)$ is an interface equality, whilst $(v_0 = v_6)$ is not.) $RESET_5$ is a boolean variable.
After the first run of unit propagations, assume DPLL *selects the literal $RESET_5$, resulting in the assignment $\mu =_{def} \mu_{\mathcal{EUF}} \cup \mu_{\mathcal{LA}(\mathbb{Q})}$ s.t.*

$$\begin{array}{ll}\mu_{\mathcal{EUF}} = & \{(v_3 = h(v_0)), (v_4 = h(v_1)), (v_6 = f(v_2)), (v_7 = f(v_5)), \neg(v_6 = v_7)\} \\ \mu_{\mathcal{LA}(\mathbb{Q})} = & \{(v_0 \leq v_1), (v_0 \geq v_1), (v_2 = v_3 - v_4), (v_5 = 0)\},\end{array} \tag{23}$$

which propositionally satisfies φ. Now, the set of literals $\mu_{\mathcal{EUF}} \subset \mu$ is given to the $\mathcal{EUF}$-solver, which reports its consistency and deduces no new interface equality. Then the set $\mu_{\mathcal{LA}(\mathbb{Q})} \subset \mu$ is given to the $\mathcal{LA}(\mathbb{Q})$-solver, which reports consistency and deduces the interface equality $v_0 = v_1$, which is passed to the $\mathcal{EUF}$-solver. The new set $\mu_{\mathcal{EUF}} \cup \{(v_0 = v_1)\}$ is still $\mathcal{EUF}$-consistent, but this time the $\mathcal{EUF}$-solver deduces the equality $(v_3 = v_4)$, which is in turn passed to the $\mathcal{LA}(\mathbb{Q})$-solver, which deduces $(v_2 = v_5)$. The $\mathcal{EUF}$-solver is then invoked again to check the $\mathcal{EUF}$-consistency of the assignment $\mu_{\mathcal{EUF}} \cup \{(v_0 = v_1), (v_2 = v_5)\}$: since this check fails, the Nelson-Oppen method reports the $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$-unsatisfiability of φ under the whole assignment μ. At this point, then, DPLL *backtracks and tries assigning false to $RESET_5$, resulting in the new assignment $\mu' =_{def} \mu_{\mathcal{EUF}} \cup \mu'_{\mathcal{LA}(\mathbb{Q})}$ s.t.*

$$\begin{array}{ll}\mu_{\mathcal{EUF}} = & \{(v_3 = h(v_0)), (v_4 = h(v_1)), (v_6 = f(v_2)), (v_7 = f(v_5)), \neg(v_6 = v_7)\} \\ \mu'_{\mathcal{LA}(\mathbb{Q})} = & \{(v_0 \leq v_1), (v_0 \geq v_1), (v_2 = v_3 - v_4)(v_5 = v_8)\},\end{array} \tag{24}$$

which is found $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$-satisfiable (see Fig. 15).

In the case of non-convex theories, the NO schema becomes more complicated, because the two solvers need to exchange arbitrary disjunctions of interface equalities, which have to be managed within the decision procedure by means of case splitting and of backtrack

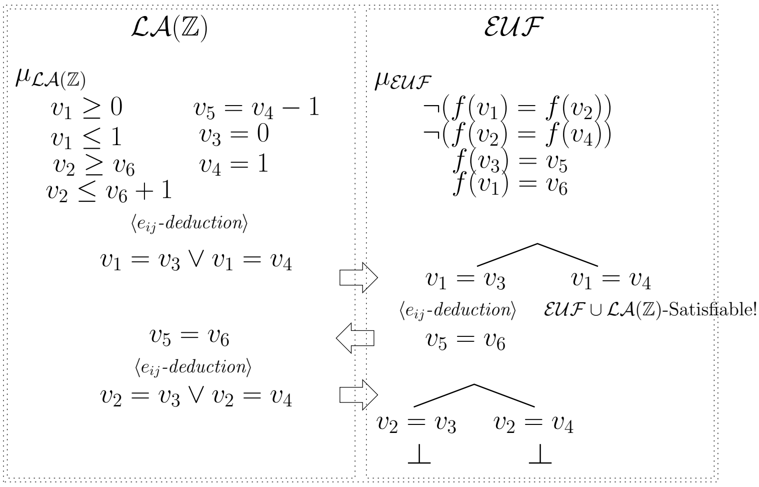


Figure 16. The NO search tree for the formula of Example 8.3

search. In the latter case, the NO schema performs a number of branches to check the consistency of a set of literals which depends on how many disjunctions of equalities are exchanged at each step: if the current set of literals is μ, and one of the $\mathcal{T}_i$-solver sends the disjunction $\bigvee_{k=1}^{n}(e_{ij})_k$ to the other, the latter must further investigate up to n branches to check the consistency of each of the $\mu \cup \{(e_{ij})_k\}$ sets separately.

We illustrate the behavior of the NO schema in the case of non-convex theories in the following example.

Example 8.3. *[BCF+06b] Consider the following $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Z})$ assignment $\mu =_{def} \mu_{\mathcal{EUF}} \cup \mu_{\mathcal{LA}(\mathbb{Z})}$ s.t.*

$$\begin{array}{ll} \mu_{\mathcal{EUF}}: & \neg(f(v_1) = f(v_2)) \wedge \neg(f(v_2) = f(v_4)) \wedge (f(v_3) = v_5) \wedge (f(v_1) = v_6) \wedge \\ \mu_{\mathcal{LA}(\mathbb{Z})}: & (v_1 \geq 0) \wedge (v_1 \leq 1) \wedge (v_5 = v_4 - 1) \wedge (v_3 = 0) \wedge (v_4 = 1) \wedge \\ & (v_2 \geq v_6) \wedge (v_2 \leq v_6 + 1). \end{array} \quad (25)$$

Here all the variables $(v_1, \ldots, v_6)$ are interface ones. φ contains only unit clauses, so after the first run of unit propagations, DPLL *generates the assignment μ which is simply the set of literals in φ. The NO combination schema then runs as depicted in Fig. 16.*

First, the sub-assignment $\mu_{\mathcal{EUF}}$ is given to the $\mathcal{EUF}$-solver, which reports its consistency and deduces no interface equality. Then, the sub-assignment $\mu_{\mathcal{LA}(\mathbb{Z})}$ is given to the $\mathcal{LA}(\mathbb{Z})$-solver, which reports its consistency and deduces the disjunction $(v_1 = v_3) \vee (v_1 = v_4)$. Next, there is a case-splitting and the two equalities $(v_1 = v_3)$ and $(v_1 = v_4)$ are passed to the $\mathcal{EUF}$-solver. The first branch, corresponding to selecting $(v_1 = v_3)$, is opened: then the set $\mu_{\mathcal{EUF}} \cup \{(v_1 = v_3)\}$ is $\mathcal{EUF}$-consistent, and the equality $(v_5 = v_6)$ is deduced. After that, the assignment $\mu_{\mathcal{LA}(\mathbb{Z})} \cup \{(v_5 = v_6)\}$ is passed to the $\mathcal{LA}(\mathbb{Z})$-solver, that reports its consistency and deduces another disjunction, $(v_2 = v_3) \vee (v_2 = v_4)$. At this point, another case-splitting is needed in the $\mathcal{EUF}$-solver, resulting in the two branches $\mu_{\mathcal{EUF}} \cup \{(v_1 = v_3), (v_2 = v_3)\}$ and $\mu_{\mathcal{EUF}} \cup \{(v_1 = v_3), (v_2 = v_4)\}$. Both of them are found inconsistent, so the whole branch previously opened by the selection of $(v_1 = v_3)$ is found inconsistent.

At this point, the other case of the branch (i.e. the equality $(v_1 = v_4)$) is selected, and since the assignment $\mu_{\mathcal{EUF}} \cup \{(v_1 = v_4)\}$ is $\mathcal{EUF}$-consistent and no new interface equality is deduced, the Nelson-Oppen method reports the $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Z})$-satisfiability of φ under the whole assignment μ.

Notice that the ability to carry out e_{ij}-deductions is crucial: each solver must be e_{ij}-deduction complete, that is, it must be able to derive the (disjunctions of) interface equalities e_{ij} which are entailed by its current facts φ. (Hereafter we assume that all the $\mathcal{T}_i$-solver's used in a NO schema are e_{ij}-deduction complete.)

We also notice that, in a standard NO-style SMT procedure, the DPLL solver is not aware of the interface equalities e_{ij}, so that the latter cannot occur in conflict clauses. Therefore, in order to construct the $\mathcal{T}_1 \cup \mathcal{T}_2$-conflict clause, it is necessary to resolve backwards the last conflict clause with (the deduction clauses corresponding to) the e_{ij}-deductions performed by each $\mathcal{T}_i$*-solver*.

Example 8.4. *Consider the scenario of the left branch in Example 8.2 and Fig. 15. Starting from the final $\mathcal{EUF}$ conflict, and resolving backwards wrt. the deductions performed, it is possible to obtain a final $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$-conflict clause as follows:*

$$\begin{array}{ll}
\mathcal{EUF} - conflict: & ((v_6 = f(v_2)) \wedge (v_7 = f(v_5)) \wedge \neg(v_6 = v_7) \wedge (v_2 = v_5)) \rightarrow \bot \\
\mathcal{LA}(\mathbb{Q}) - deduction: & ((v_2 = v_3 - v_4) \wedge (v_5 = 0) \wedge (v_3 = v_4)) \rightarrow (v_2 = v_5) \\
\mathcal{EUF} - deduction: & ((v_3 = h(v_0)) \wedge (v_4 = h(v_1)) \wedge (v_0 = v_1)) \rightarrow (v_3 = v_4) \\
\mathcal{LA}(\mathbb{Q}) - deduction: & ((v_0 \geq v_1) \wedge (v_0 \leq v_1)) \rightarrow (v_0 = v_1) \\
\Longrightarrow & \\
\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q}) - conflict: & ((v_6 = f(v_2)) \wedge (v_7 = f(v_5)) \wedge \neg(v_6 = v_7) \wedge (v_2 = v_3 - v_4) \wedge \\
 & (v_5 = 0) \wedge (v_3 = h(v_0)) \wedge (v_4 = h(v_1)) \wedge (v_0 \geq v_1)) \rightarrow \bot.
\end{array}$$

Notice that the novel conflict clause simply causes the propagation of $\neg(v_5 = 0)$ and of $\neg RESET_5$, and provides no help for pruning search in the right branch.

8.3 Delayed Theory Combination

A more SAT-related approach for tackling the $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$ problem, called *Delayed Theory Combination (*Dtc*)*, has been proposed in [BBC+05c, BBC+06b]. A basic architectural schema of Dtc is described in Figure 14.

Again, let $\mathcal{T}_1$, $\mathcal{T}_2$ be two signature-disjoint stably-infinite theories and let $\mathcal{T}_1$*-solver*, $\mathcal{T}_2$*-solver* be their respective theory solvers (unlike with NO, they are not required to be e_{ij}-deduction complete). In Dtc, each of the two $\mathcal{T}_i$*-solvers* interacts only with the boolean enumerator: there is no direct exchange of information between the $\mathcal{T}_i$*-solvers*. Their mutual consistency is ensured by augmenting the input problem with all interface equalities e_{ij}, even if these do not occur in the original problem: the boolean enumerator is instructed to assign truth values not only to the atoms in *Atoms*, but also the interface equalities e_{ij}'s. Consequently, the enumerated assignments include not only the atoms in the formula, but also the e_{ij}'s. Both theory solvers receive, from the boolean level, the same truth assignment μ_e for e_{ij}: under such conditions, the two "partial" models found by each decision procedure can be merged into a model for the input formula.

A simplified view of the algorithm, in accordance with the *offline* integration schema of §5.2, is presented in Fig. 17. Initially (rows 2–4), the formula is purified, the *new* interface

```
1.   SatValue $\mathcal{T}_1 \cup \mathcal{T}_2$-DPLL ($\mathcal{T}_1 \cup \mathcal{T}_2$-formula $\varphi$) {
2.       $\varphi$ = Purify($\varphi$);
3.       $\mathcal{A}^p$ = $\mathcal{T}2\mathcal{B}$($Atoms \cup InterfaceEqualities(\varphi)$);
4.       $\varphi^p$ = $\mathcal{T}2\mathcal{B}$($\varphi$);
5.       while (DPLL($\varphi^p, \mu^p$) == Sat){
6.           $\mu^p_{\mathcal{T}_1} \wedge \mu^p_{\mathcal{T}_2} \wedge \mu^p_e$ = $\mu^p$;
7.           (res$_1$,$\pi_1$) = $\mathcal{T}_1$-solver($\mathcal{B}2\mathcal{T}$($\mu^p_{\mathcal{T}_1} \wedge \mu^p_e$));
8.           (res$_2$,$\pi_2$) = $\mathcal{T}_2$-solver($\mathcal{B}2\mathcal{T}$($\mu^p_{\mathcal{T}_2} \wedge \mu^p_e$));
9.           if (res$_1$ == Sat && res$_2$ == Sat)
10.              return Sat;
11.          else if (res$_1$ == Unsat)
12.              $\varphi^p$ = $\varphi^p \wedge \neg\mathcal{T}2\mathcal{B}(\pi_1)$;
13.          else if (res$_2$ == Unsat)
14.              $\varphi^p$ = $\varphi^p \wedge \neg\mathcal{T}2\mathcal{B}(\pi_2)$;
15.      };
16.      return Unsat;
17.  };
```

Figure 17. An offline schema of DTC for $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$.

equalities e_{ij}'s are created and added to the set of propositional symbols $\mathcal{A}^p$, and the propositional abstraction φ^p of φ is created. Then, the main loop is entered (rows 5–15): while φ^p is found propositionally satisfiable by DPLL (row 5), a satisfying truth assignment μ^p is selected (row 6). It is important to stress that truth values are associated not only to atoms in φ, but also to the e_{ij}'s, even though they do not occur in φ. μ^p is then partitioned into $\mu^p_{\mathcal{T}_1} \wedge \mu^p_e \wedge \mu^p_{\mathcal{T}_2}$, where $\mathcal{B}2\mathcal{T}(\mu^p_i)$ is the subset of i-pure literals and $\mathcal{B}2\mathcal{T}(\mu^p_e)$ is the subset of e_{ij}-literals in μ. For each $\mathcal{T}_i$, the $\mathcal{T}_i$-relevant part of μ^p, $\mu^p_{\mathcal{T}_i} \wedge \mu^p_e$, is checked for $\mathcal{T}_i$-consistency (rows 7–8); each $\mathcal{T}_i$-*solver* returns a pair (res$_i$, π_i), where res$_i$ is Unsat iff $\mathcal{B}2\mathcal{T}(\mu^p_{\mathcal{T}_i} \wedge \mu^p_e)$ is $\mathcal{T}_i$-unsatisfiable, and Sat otherwise. If both calls to $\mathcal{T}_i$-solver return Sat, then the formula is satisfiable. Otherwise, when res$_i$ is Unsat, then π_i is a conflict set. Then, φ^p is strengthened to exclude truth assignments which may fail in the same way (row 11–14), and the loop is resumed. Unsatisfiability is returned (row 16) when the loop is exited without having found a model.

The above schema can be improved to many extents: step 8 can be invoked only if res$_1$ is Sat; more importantly, if DPLL returns a *partial* assignment μ, then it is sufficient that μ_e assigns only the e_{ij}'s which have an actual interface role in μ. [39.]

A more efficient implementation of DTC [BBC$^+$06b] is based on the *online* integration with a modern DPLL engine —exploiting early pruning, $\mathcal{T}$-propagation, $\mathcal{T}$-backjumping and $\mathcal{T}$-learning— described in §5.3. Let $\mathcal{T}$ be $\mathcal{T}_1 \cup \mathcal{T}_2$. In order to guarantee the correctness and completeness of DTC, the $\mathcal{T}$-DPLL algorithm of §5.3 (see Fig. 7) must be modified to the following extents [BBC$^+$06b]:

[39.] E.g., if μ is partial and v is an interface variable in φ but it occurs in no 1-pure literal in μ, then v has no "interface role" for μ, so that every interface equality containing v can be ignored by μ_e. (If μ is total, then this situation cannot occur.)

- $\mathcal{T}$-`preprocess` must perform also the purification of the input formula φ.
- DPLL must be instructed to assign truth values not only to the atoms in $Atoms(\varphi)$, but also to the interface equalities e_{ij}'s. $\mathcal{B}2\mathcal{T}$ and $\mathcal{T}2\mathcal{B}$ are modified accordingly.
- $\mathcal{T}$-`decide_next_branch` is modified to select not only atoms in the original formula, but also new interface equalities.
- $\mathcal{T}$-`deduce`, step (ii), is modified to work like Rows 6–14 of Fig. 17: μ^p is partitioned into $\mu^p_{\mathcal{T}_1} \wedge \mu^p_{\mathcal{T}_2} \wedge \mu^p_e$, and both $\mathcal{T}_i$*-solvers* are invoked on $\mathcal{B}2\mathcal{T}(\mu^p_{\mathcal{T}_2} \wedge \mu^p_e)$: if both return Sat, then $\mathcal{T}$-`deduce` returns Sat, otherwise it returns Conflict.
- $\mathcal{T}$-`analyze_conflict` and $\mathcal{T}$-`backtrack` are modified so that to use the conflict set returned by one $\mathcal{T}_i$*-solver* for backjumping and learning. Importantly, such conflict sets may contain interface equalities.

In order to achieve efficiency, the following modifications have been further suggested in [BBC+05c, BBC+06b, BCF+06b].

$\mathcal{T}$-`decide_next_branch` is instructed to hamper the selection of positive new interface equalities e_{ij}'s, that is,

(i) a new e_{ij} is selected only after all original atoms have been assigned;
(ii) when selected, a new e_{ij} is assigned a negative value first;
(iii) as described above, only the e_{ij}'s which have an actual interface role in μ are selected.

Intuitively, positive interface equalities are considered only when they are strictly necessary to guarantee the mutual consistency of the two sub-assignments. An important improvement of (i) is to exploit the issue of partial assignments described in §7.2.1: when the current partial assignment μ propositionally satisfies the input formula φ, the remaining atoms occurring in φ can be ignored and (the negation of) the new e_{ij}'s are selected. One further variant is to limit (i) only to the new e_{ij}'s which do not occur in learned clauses: the others can be selected by $\mathcal{T}$-`decide_next_branch` with no restriction. Intuitively, the e_{ij}'s which participate in conflicts gain on-the-field a better consideration from the SAT solver.

$\mathcal{T}$-`deduce` can be improved in the following ways.

(i) Early pruning or weakened EP is applied before every selection of a new e_{ij}.
(ii) If some $\mathcal{T}_i$*-solver* has deduction or e_{ij}-deduction capabilities, then $\mathcal{T}$-propagation is performed.
(iii) Each $\mathcal{T}_i$*-solver* is invoked only if at least one literal (which has not been deduced singularly by $\mathcal{T}_i$*-solver* itself) has been added to its input since the last call. [40]
(iv) At every early-pruning call on a branch μ which is found both $\mathcal{T}_1$- and $\mathcal{T}_2$-consistent, if one $\mathcal{T}_i$-solver performs the e_{ij}-deduction $\mu^* \models_{\mathcal{T}_i} \bigvee_{j=1}^{k} e_j$, s.t. $\mu^* \subseteq \mu_{\mathcal{T}_i} \wedge \mu_e$, then:
   (a) the deduction clause $\mathcal{T}2\mathcal{B}(\mu^* \rightarrow \bigvee_{j=1}^{k} e_j)$ is learned;
   (b) if $k > 1$, then $\neg e_1, ..., \neg e_k$ are put on the top of the literal selection list, so that to be the next $\neg e_{ij}$'s selected by the literal selection heuristic.

[40] This avoids invoking one $\mathcal{T}_i$*-solver* twice in sequence on the same input. The restriction "which ... itself" means that, if $\mathcal{T}_i$*-solver* (μ) returns Sat and deduces e_{ij}, then $\mathcal{T}_i$*-solver* is not invoked on $\mu \cup \{e_{ij}\}$.

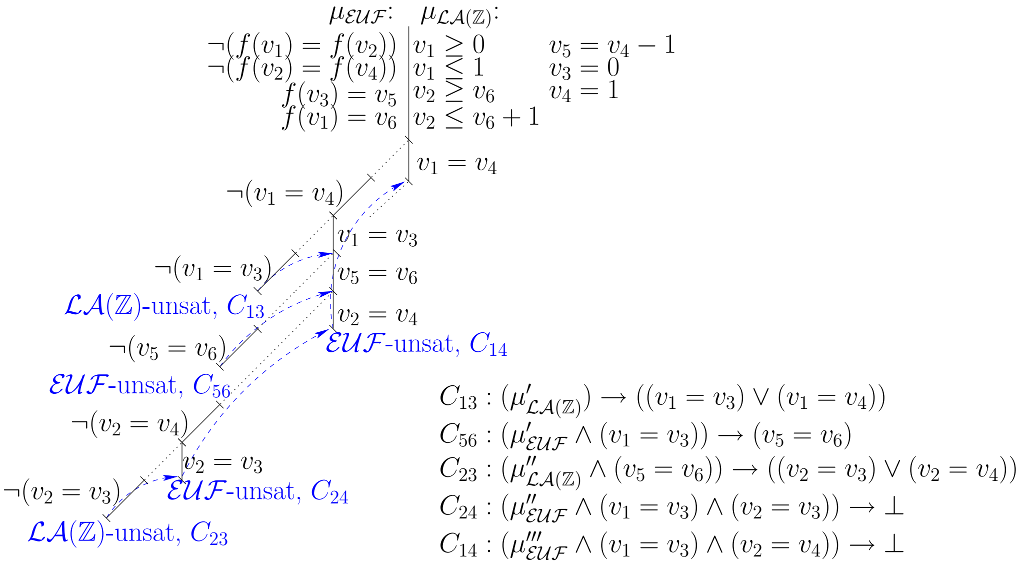

$\mu_{\mathcal{EUF}}$:
$\mu_{\mathcal{LA}(\mathbb{Z})}$:
$\neg(f(v_1) = f(v_2))$
$\neg(f(v_2) = f(v_4))$
$f(v_3) = v_5$
$f(v_1) = v_6$
$v_1 \geq 0$
$v_1 \leq 1$
$v_2 \geq v_6$
$v_2 \leq v_6 + 1$
$v_5 = v_4 - 1$
$v_3 = 0$
$v_4 = 1$
$v_1 = v_4$
$\neg(v_1 = v_4)$
$v_1 = v_3$
$\neg(v_1 = v_3)$
$v_5 = v_6$
$\mathcal{LA}(\mathbb{Z})$-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}
$C_{13} : (\mu'_{\mathcal{LA}(\mathbb{Z})}) \rightarrow ((v_1 = v_3) \vee (v_1 = v_4))$
$C_{56} : (\mu'_{\mathcal{EUF}} \wedge (v_1 = v_3)) \rightarrow (v_5 = v_6)$
$C_{23} : (\mu''_{\mathcal{LA}(\mathbb{Z})} \wedge (v_5 = v_6)) \rightarrow ((v_2 = v_3) \vee (v_2 = v_4))$
$C_{24} : (\mu''_{\mathcal{EUF}} \wedge (v_1 = v_3) \wedge (v_2 = v_3)) \rightarrow \bot$
$C_{14} : (\mu'''_{\mathcal{EUF}} \wedge (v_1 = v_3) \wedge (v_2 = v_4)) \rightarrow \bot$

Figure 18. The DTC search tree for Example 8.5 on $\mathcal{LA}(\mathbb{Z}) \cup \mathcal{EUF}$, with no e_{ij}-deduction. Here "C_{ij}" denotes the conflicting clause causing the backjump to $(v_i = v_j)$.

(v) **[If and only if both $\mathcal{T}_i$*-solvers* are e_{ij}-deduction complete.]** If an assignment μ which propositionally satisfies φ is found $\mathcal{T}_i$-satisfiable for both $\mathcal{T}_i$'s, and neither $\mathcal{T}_i$-solver performs any e_{ij}-deduction from μ, then $\mathcal{T}$-DPLL stops returning Sat. [41.]

$\mathcal{T}$`-analyze_conflict` and $\mathcal{T}$`-backtrack` can be improved as follows.

(i) Each conflict clause is a mixed boolean+theory conflict clause which is built from the conflict set returned by one $\mathcal{T}_i$*-solver* as described in §6.5.

(ii) All the conflict clauses derived by theory conflicts are learned, either temporarily or permanently.

Example 8.5. *[BCF⁺06b] Consider the $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Z})$ formula φ (25) and the assignment μ of Example 8.3. We assume here that both the $\mathcal{EUF}$- and $\mathcal{LA}(\mathbb{Z})$-solver's have no e_{ij}-deduction capabilities, but that they always return $\neg e_{ij}$-minimal conflict sets. A session of* DTC *is depicted in Fig. 18.*

Initially, both $\mu_{\mathcal{LA}(\mathbb{Z})}$ and $\mu_{\mathcal{EUF}}$ are found consistent in each of the theories by the respective solvers. Then DTC *starts selecting new $\neg e_{ij}$'s, and proceeds without causing conflicts, until it selects $\neg(v_1 = v_4)$ and $\neg(v_1 = v_3)$, which cause a $\mathcal{LA}(\mathbb{Z})$ conflict. The branch is in the form $\mu \cup \bigcup_j \neg e_j$, so that, the $\neg e_{ij}$-minimal conflict set η_{13} returned is in the form $\mu'_{\mathcal{LA}(\mathbb{Z})} \cup \{\neg(v_1 = v_3), \neg(v_1 = v_4)\}$.* [42.] DTC *learns the corresponding clause C_{13}, and backjumps up to the highest point which allows for unit-propagating $(v_1 = v_3)$ on C_{13}, and performs such unit propagation. Then* DTC *selects a chain of new $\neg e_{ij}$'s without causing conflicts, until it selects $\neg(v_5 = v_6)$, which causes a $\mathcal{EUF}$ conflict. As $\mathcal{EUF}$ is convex,*

[41.] This is identical to the $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiability termination condition of NO.

[42.] Hereafter, $\mu'_{\mathcal{T}_i}$, $\mu''_{\mathcal{T}_i}$, $\mu'''_{\mathcal{T}_i}$ will denote generic subsets of $\mu_{\mathcal{T}_i}$, $\mathcal{T} \in \{\mathcal{EUF}, \mathcal{LA}(\mathbb{Q}), \mathcal{LA}(\mathbb{Z})\}$.

$\neg(v_5 = v_6)$ is the only $\neg e_{ij}$ occurring in the conflict set, so that Dtc *learns clause C_{56}, backtracks over the last chain of $\neg e_{ij}$'s and unit-propagates $(v_5 = v_6)$.*

Again, Dtc *selects a chain of new $\neg e_{ij}$'s without causing conflicts, until it selects $\neg(v_2 = v_4)$ and $\neg(v_2 = v_3)$, which cause a $\mathcal{LA}(\mathbb{Z})$ conflict. As before, it learns clause C_{23}, and backjumps to the highest point where it can unit-propagate $(v_2 = v_3)$ on C_{23}. Performing the latter unit propagation causes a $\mathcal{EUF}$ conflict, with conflicting clause C_{24}. By resolving on literals $(v_2 = v_3)$, $(v_5 = v_6)$, $(v_1 = v_3)$ the conflicting clause C_{24} with the clauses C_{23}, C_{56} and C_{13}, (which caused the unit-propagation of $(v_2 = v_3)$, $(v_5 = v_6)$ and $(v_1 = v_3)$ respectively),* Dtc *obtains a mixed theory/boolean clause C'_{24} : $(\mu'_{\mathcal{LA}(\mathbb{Z})} \wedge \mu'_{\mathcal{EUF}} \wedge \mu''_{\mathcal{LA}(\mathbb{Z})} \wedge \mu''_{\mathcal{EUF}}) \rightarrow ((v_1 = v_4) \vee (v_2 = v_4))$, which allows it for backjumping over all the remaining $\neg e_{ij}$'s of the current chain and unit-propagating $(v_2 = v_4)$.*

The latter causes a new $\mathcal{EUF}$ conflict represented by the conflicting clause C_{14}. Then C_{14} is resolved with the clauses C'_{24}, C_{13} (which caused the unit-propagation of $(v_2 = v_4)$ and $(v_1 = v_3)$ respectively), obtaining thus the new conflict clause C'_{14} : $(\mu'_{\mathcal{LA}(\mathbb{Z})} \wedge \mu''_{\mathcal{LA}(\mathbb{Z})} \wedge \mu'_{\mathcal{EUF}} \wedge \mu''_{\mathcal{EUF}} \wedge \mu'''_{\mathcal{EUF}}) \rightarrow (v_1 = v_4)$, which allows for backjumping up to μ and for unit-propagating $(v_1 = v_4)$.

Finally, Dtc *selects a sequence of $\neg e_{ij}$'s (possibly unit-propagating some value due to the clauses learned) without generating conflicts, so that to conclude that the formula is $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiable.*

8.4 Discussion

An essential difference between NO and Dtc is that the former requires the *$\mathcal{T}_i$-solvers* to be deduction-complete, whilst the latter allows for using $\mathcal{T}_i$-solver's with partial or no e_{ij}-deduction capability. In fact, in Dtc part of or all the e_{ij}-deductions are substituted by extra boolean search on the e_{ij}'s performed by the SAT solver. [BCF+06b] shows that:

- under the same hypotheses of NO (stably-infinite theories, incremental, backtrackable and e_{ij}-deduction-complete *$\mathcal{T}_i$-solvers*) Dtc can be implemented so that to emulate NO, requiring no extra boolean search;

- if the *$\mathcal{T}_i$-solvers* are not e_{ij}-deduction complete, $\mathcal{T}$-backjumping eliminates the extra boolean search except for some caused by the presence of redundant negated e_{ij}'s in the conflict sets. If the *$\mathcal{T}_i$-solvers* can remove such redundancies, the boolean search reduces down to at most one extra boolean branch for every e_{ij}-deduction avoided.

Another difference is that, in case of non-convex theories, in NO each *$\mathcal{T}$-solver* must handle the case-splits caused by the deduction of disjunctions of e_{ij}'s performed by the other *$\mathcal{T}$-solver*; in Dtc this is handled directly by the SAT solver (e.g., compare Examples 8.3 and 8.5). Notice that, unlike with splitting on demand [BNOT06] in §6.7, here we refer to the case-splits which are necessary to *handle* the deductions performed by the other *$\mathcal{T}$-solver*, rather to those which may be necessary to *perform* such deductions.

Finally, in Dtc the SAT solver is aware a priori of the e_{ij}'s, so that Dtc allows for learning clauses containing e_{ij}'s, which can be used in subsequent branches to prune search and avoid redoing the same search/e_{ij}-deductions from scratch.

We refer the reader to [BBC$^+$05c, BBC$^+$06b] for a more detailed description of Dtc, to [BCF$^+$06a] for a comparison between Dtc and Ackermann's expansion on SMT($\mathcal{EUF} \cup \mathcal{T}$), and to [BCF$^+$06b] for an analytical comparison between Dtc and NO.

9. Related approaches for SMT

All the most extensive empirical evaluations performed in the last years [GHN+04, dMR04, NO05a, BBC+05b, BdMS05, SMT05, SMT06] seem to confirm the fact that currently all the most efficient SMT tools are based on the lazy approach combining state-of-the-art DPLL implementations with specialized $\mathcal{T}_i$*-solvers*. In this section we discuss this fact, and survey the current alternatives.

9.1 Alternative Enumerators for lazy SMT

Most state-of-the-art lazy SMT tools use DPLL as Enumerator. We first try to motivate this fact (§9.1.1), and then we survey some promising alternatives (§9.1.2 and §9.1.3).

9.1.1 Why DPLL?

We believe there are a few facts which contribute to the popularity of DPLL as Enumerator in lazy SMT tools.

From a theoretical viewpoint, DPLL has some important properties which make it very suitable for implementing an Enumerator [GS00, ABC+02b]. First, DPLL performs a depth-first-search on assignments, so that it requires a polynomial amount of memory. [43.] This is not the case, e.g., of OBDD's [Bry86]. Second, DPLL allows for implementing Enumerators which are intrinsically non-redundant, in the sense that they avoid generating partial assignments which cover areas of the boolean search space which are already covered by previously-generated assignments. This is not the case, e.g., of semantic tableaux [Smu68]. We refer the reader to [ABC+02b] for a more detailed explanation on these issues.

From a more practical perspective, a few other facts contribute to the success of DPLL in lazy SMT. First, most efficient (complete) SAT solvers are DPLL implementations, and the source code of extremely efficient implementations of DPLL is available from the shelf. In particular, by integrating a state-of-the-art DPLL solver, one can benefit for free of all the enhancements briefly described in §3.1. Second, there is a wide and detailed literature on efficient DPLL solvers, involving both high-level algorithmical issues and implementation tricks, so that one can achieve all the information needed to successfully implement an efficient DPLL-based Enumerator in-house. Third, DPLL is more a family of algorithms than one single algorithm: a plethora of variants of DPLL can be constructed on, e.g., different literal selection heuristics, different forms of preprocessing and simplification, and different strategies for restarts, conflict-set generation, clause learning and discharging. This gives a wide choice of possibilities for customizing an ad-hoc variant of DPLL.

9.1.2 OBDD-based SMT solvers

In the Model Checking community there has been some work on integrating Ordered Binary Decision Diagrams - OBDD's [Bry86] with theory-information in order to handle complex verification problems. [CABN97] integrated OBDD's with an (incomplete) quadratic constraint solver to verify transition systems with integer data values; [MLAH99] developed Difference Decision Diagrams (DDDs), OBDD-like data structures handling boolean combi-

[43.] Here we assume that a proper strategy of clause learning and discharging is implemented to avoid learning an exponential number of clauses. See §3.

nations of temporal constraints, and used them to verify timed systems; [GSZ$^+$98] developed OBDD-based procedures for $\mathcal{EUF}$; [YKTB00] developed a library of procedures combining OBDD's and a solver for Pressburger arithmetic, and used them to verify infinite-state systems. [44.] [RD03] combine OBDDs with superposition theorem provers (see also §9.2). In a more general perspective, [Arm03] introduced an optimized way to efficiently integrate OBDD's with solvers for decidable theories. Notably, all these approaches adopt a technique similar to early pruning (§6.3) to reduce the size of the final OBDD. Unfortunately, all these approaches inherit from OBDD's the drawback of requiring exponential space in worst case.

9.1.3 Circuit-based techniques

In the recent years, alternative boolean solvers which are specialized for reasoning directly on boolean circuits, rather than on CNF formulas, have been proposed (see, e.g., [JN00, KGP01, GAG$^+$02, IPC03, TBW04, LHS04, JS05]). Unlike DPLL on CNF-ized representation of circuits, these techniques benefit from the structure of the circuit representation, and can perform efficient boolean constraint propagation, including that of *don't care* values. In particular, [GAG$^+$02] proposed a mixed Circuit-based and DPLL-based approach, combining the power of DPLL learning with Circuit-based boolean value propagation and structure-driven search; [TBW04] generalized the two-watched-literals technique to boolean circuits.

[LHS04, JS05] focused on the problem of enumerating complete sets of assignments for boolean circuit representations: [LHS04] developed a group of very efficient techniques, including a mixed conflict/success-driven learning scheme, a form of quantified backjumping and a practical method for storing solutions into OBDD's; [JS05] developed a technique for the enumeration of *prime clauses*, which allow for significantly reducing the overall boolean search space. Although these techniques were mostly conceived in the context of SAT-based unbounded model checking [McM02], we believe they may find a natural application in lazy SMT.

With some noteworthy exception (e.g., [PICW04] extended the circuit-based solver in [IPC03] to the domain of $\mathcal{BV}$), the application of circuit-based techniques to lazy SMT is still ongoing research work.

9.2 The rewrite-based approach for building $\mathcal{T}$-*solvers*

A relatively-recent and promising approach for building $\mathcal{T}$-*solvers* is that referred as *rewrite-based* [45.] approach [ARR03, RD03, ABRS05, KRRT05, BGN$^+$06, KRRT06]. The main idea is that of producing $\mathcal{T}_i$-*solvers* for (typically finitely-axiomatizable) theories $\mathcal{T}_i$ and for their combinations by customizing an equational FOL theorem prover, in particular one based on the *superposition calculus* [46.]. This is done by incorporating the axioms of the theory and by introducing ad-hoc control strategies in order to drive the search.

[44.] The list of references presented here is not intended to be exhaustive; rather, it aims at providing some representative samples of the OBDD-based approach in the literature.

[45.] Also referred as *superposition-based* approach.

[46.] The superposition calculus [BG94] is a calculus for reasoning in equational FOL, which combines first-order resolution with ordering-based equality handling. Most current state-of-the-art theorem provers are based on the superposition calculus (e.g., the superposition-based equational theorem prover E [Sch02]).

The key issue for this approach is proving termination: in order to obtain a $\mathcal{T}$*-solver*, it is necessary to prove that theorem-proving strategy is bound to terminate on satisfiability problem in the theories of interest. In [ARR03] a refutationally-complete rewrite-based inference system was shown to generate finitely-many clauses on satisfiability problems in a bunch of theories. [ABRS05, KRRT05, BGN^{+}06, KRRT06] extended these results to some more theories and, more importantly, showed how to apply the methods also to *combinations* of theories. These termination results show that, at least in principle, rewrite-based theorem provers could be used off-the-shelf as validity checkers. To this extent, [ABRS05] performed an empirical comparison, showing performances comparable or superior to those of CVC and CVCLite.

We notice that, for this approach, the desirable features of $\mathcal{T}$*-solvers* described in §4.1 depend on the specific features of the FOL theorem prover used.

- *Model generation* (§4.1.1) derives straightforwardly from the capability of the FOL theorem prover of returning a model when the formula is satisfiable. Unfortunately, superposition-based theorem provers are not always able to do that.

- *Conflict set generation* (§4.1.2) is achievable, as most provers are capable of producing a proof for the unsatisfiability of a set of literals, whose leaves can be grouped into a conflict set.

- *Incrementality* (§4.1.3) can be achieved, in principle, by means of an incremental FOL theorem prover. Unfortunately, incrementality is not a typical item in the agenda of FOL theorem provers developers. This problem is discussed in [KRRT05], where it is proposed a solution by means of an incremental congruence-closure algorithm.

- *Deduction of unassigned literal* (§4.1.4) is still an open problem, as far as we are aware.

- *Deduction of interface equalities* (§4.1.5) can be implemented by means of the techniques described in [KRRT05, KRRT06].

On the whole the rewrite-based approach has, at least in principle, some very nice features. First, it is conceptually simple and elegant. Second, it benefits automatically and nearly for free of the improvements in the technology of superposition-based FOL theorem provers. Third, the task of proving the correctness is reduced to the (typically simpler) task of proving the termination for the rules of superposition calculus [ARR03]. Fourth, and probably most important, under some sufficient conditions [ABRS05], a combined $\mathcal{T}_1 \cup \mathcal{T}_2$*-solver* can be obtained from $\mathcal{T}_1$*-solver* and $\mathcal{T}_2$*-solver* by providing the union of the axiomatizations of $\mathcal{T}_1$ and $\mathcal{T}_2$.

On the negative side, we notice that, in order to get extra functionalities (e.g., incrementality, deduction of unassigned literals) one must put the hands into the code of FOL theorem provers, which are typically very sophisticated and complicated tools. Finally, the approach is not completely mature yet and, although promising, the performances are not yet comparable with those of state-of-the-art lazy tools implementing specialized $\mathcal{T}$*-solvers*.

9.3 The eager approach to SMT

For some theories $\mathcal{T}$, a different approach to the usage of SAT tools for SMT($\mathcal{T}$) is that of reducing $\mathcal{T}$-satisfiability to SAT: the input $\mathcal{T}$-formula is translated into an equi-satisfiable boolean formula, and a SAT solver is used to check its satisfiability. (This approach is often called *eager*, in contra-position to the *lazy* approach described in §5 and §6. [47.])

Effective encodings into SAT have been conceived for a significant amount of first-order theories of interest for formal verification, including $\mathcal{EUF}$, $\mathcal{CLU}$ [48.], $\mathcal{DL}$, $\mathcal{SUF}$ [49.], $\mathcal{LA}$, and $\mathcal{AR}$ [VB99, BLS02, SSB02, Str02, SLB03]. In particular, two main families of encodings are worth-mentioning:

- in the *small-domain encoding, SD* [BLS02, TSSP04], for each variable v in a finite model an appropriate range of values $|v|$ is found. Then each v is encoded into a vector of $\lceil log_2(|v|)\rceil$ boolean variables. Binary arithmetic is then used to transform teh original input formula into a boolean formula;

- in the *per-constraint-encoding* [GSZ+98, SSB02, Str02] a new boolean variable A_ψ is introduced for every atom ψ in the input formula φ (i.e., $A_\psi := \mathcal{T}2\mathcal{B}(\psi)$). Then φ is encoded into a boolean formula $\varphi^p \wedge \varphi^{\mathcal{T}}$, φ^p being $\mathcal{T}2\mathcal{B}(\varphi)$ and $\varphi^{\mathcal{T}}$ being a boolean formula encoding a set of constraints over the variables in φ^p which mimic the constraints induced in $\mathcal{T}$on the corresponding $\mathcal{T}$-atoms. (E.g., transitivity constraints.) Notice that, unlike with static learning in lazy SMT (§6.2), $\varphi^{\mathcal{T}}$ may contain many boolean variables which do not occur in φ^p. E.g., if $\mathcal{T}$ is $\mathcal{DL}$ and φ contains $(x \leq y)$ and $(y \leq z)$ but not $(x \leq z)$, then the new boolean atom $A_{(x\leq z)}$ must be introduced and $\varphi^{\mathcal{EUF}}$ will contain the clause $(A_{(x\leq y)} \wedge A_{(y\leq z)}) \rightarrow A_{(x\leq z)}$.

A mixed method [SLB03] combines the strengths of the two encoding schemata, producing a very significant performance improvements over both of them.

The eager approach, which has been pioneered by the UCLID tool [SLB03, UCL, LS04], leverages on the accuracy of the encodings and on the effectiveness of propositional SAT solvers. In particular, it presents some nice features.

- It is relatively easy to implement: once the encoding has been formally defined, the whole implementation reduces to that of an encoder, as the whole search can be performed by some state-of-the-art SAT solver.

- The proof of soundness and completeness of the whole procedure reduces to proving the fact that the encoding is satisfiability preserving.

[47.] The adjectives "lazy" and "eager" derive from the fact that, in the former approach the theory information is used "lazily" during the search, as the boolean models are abstracted and checked for $\mathcal{T}$-consistency one-by-one, whilst in the latter approach the whole theory information is used "eagerly" from the beginning, as the whole input formula is converted one-shot into a boolean formula, so that its boolean models are abstracted and searched altogether.

[48.] $\mathcal{CLU}$ is the theory of Counter arithmetic with Lambda expressions and Uninterpreted functions. This theory generalizes $\mathcal{EUF}$ with constrained lambda expressions, ordering, and successor and predecessor functions [BLS02].

[49.] $\mathcal{SUF}$ is the logic of *Separation predicates and Uninterpreted Functions*, combining $\mathcal{EUF}$ and $\mathcal{DL}$ [SLB03].

- It benefits automatically and for-free of all improvements in the efficiency of state-of-the-art SAT solvers.

In particular, the UCLID tool has achieved important results in the formal verification of pipelined microprocessors (see, e.g., [BGV99, VB99, VB03]).

However, the eager approach often suffers from a blow-up in the encoding to propositional logic. The bottleneck is even more evident in the case of theories involving arithmetic, such as $\mathcal{DL}$ and $\mathcal{LA}$ [SSB02, Str02]. E.g., although SMT($\mathcal{DL}$) and SMT($\mathcal{LA}$) are NP-complete, the encodings proposed in [SSB02] and [Str02] blow up exponentially and doubly-exponentially respectively, because they mimic the Fourier-Motzkin expansion of $\mathcal{DL}$ constraints and linear inequalities respectively. In fact, in recent papers UCLID has been totally outperformed by DPLL-based lazy tools [GHN$^+$04, dMR04, BBC$^+$05b, NO05a], and no eager SMT tool took part at the SMT competitions [BdMS05, SMT05, SMT06]. As a consequence, there seems to be a general consensus that the eager approach is no more at the state-of-the-art of SMT tools, at least in terms of efficiency.

9.4 Mixed eager/lazy approaches

Recently, some mixed eager/lazy approaches have been proposed.

In [KOSS04] some of the UCLID authors have explored a mixed eager/lazy approach for $\mathcal{LA}$, based on an alternation of SD encoding and lazy SMT. In brief, the technique works as follows. Starting from a small range, the SD encoding φ^* of the input formula φ is produced. If φ^* is satisfiable, then φ is $\mathcal{T}$-satisfiable. Otherwise, the unsatisfiable core ψ^p of φ^* is produced, and the corresponding $\mathcal{T}$-formula ψ is given in input to a lazy SMT solver. (Notice that ψ is a subformula of φ, possibly much smaller than φ.) If ψ is $\mathcal{T}$-unsatisfiable, then φ is $\mathcal{T}$-unsatisfiable. Otherwise, a solution for φ is used to compute a new (bigger) range, and the loop proceeds. The process is guaranteed to terminate because (if φ is $\mathcal{T}$-satisfiable) a sufficiently big range is eventually reached and (if φ is $\mathcal{T}$-unsatisfiable) only a finite number of ψ's can be generated.

[GTG06] presented SDSAT, a mixed eager/lazy tool for $\mathcal{DL}$. SDSAT works in two phases: first (allocation phase) it allocates non-uniform SD ranges for each variable (without performing the SD encoding one-shot); then (solve phase) it uses a lazy refinement approach for searching a model within the allocated ranges. A novel algorithm for range allocation is also presented, which much better performances wrt. previous ones.

[KS06] presented a completely-different mixed eager/lazy approach. Within a standard lazy framework, they proposed a novel $\mathcal{T}$*-solver* for $\mathcal{DL}(\mathbb{Z})$, which is particularly focused on efficiently handling disequalities. The $\mathcal{T}$*-solver* is organized according to a layered schema (§4.3): first, positive equalities are processed, equivalence classes are computed and variable substitutions are performed, so that to eliminate them from the assignment; the remaining inequalities are then entered a certain amount of graph-based procedures (including SCC-partitioning and standard negative-path detection): if a solution is found, it is checked against the remaining disequalities. If also this technique fails, everything is encoded into SAT by means of SD encoding and is is solved by an incremental SAT solver (which can benefit from previous calls on other assignments). This technique showed good performances wrt. state-of-the-art tools when lots of disequalities come into play.

References

[ABC$^+$02a] 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.

[ABC$^+$02b] G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. Integrating Boolean and Mathematical Solving: Foundations, Basic Algorithms and Requirements. In *Proc. AIARSC'2002*, volume 2385 of *LNAI*. Springer, 2002.

[ABCS03] G. Audemard, M. Bozzano, A. Cimatti, and R. Sebastiani. Verifying Industrial Hybrid Systems with MathSAT. In *Proc. BMC'04*, volume 89/4 of *ENTCS*. Elsevier, 2003.

[ABRS05] A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. On a rewriting approach to satisfiability procedures: extension, combination of theories and an experimental appraisal. In *Proc. of FroCoS'2005*, volume 3717 of *LNCS*. Springer, 2005.

[ACG99] A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proc. European Conference on Planning, CP-99*, 1999.

[ACGM04] A. Armando, C. Castellini, E. Giunchiglia, and M. Maratea. A SAT-based Decision Procedure for the Boolean Combination of Difference Constraints. In *Proc. SAT'04*, 2004.

[Ack54] W. Ackermann. *Solvable Cases of the Decision Problem*. North Holland Pub. Co., Amsterdam, 1954.

[ACKS02] G. Audemard, A. Cimatti, A. Korniłowicz, and R. Sebastiani. SAT-Based Bounded Model Checking for Timed Systems. In *Proc. FORTE'02.*, volume 2529 of *LNCS*. Springer, November 2002.

[Arm03] A. Armando. Simplifying OBDDs in Decidable Theories. In *Proc. PDPAR'03.*, 2003.

[ARR03] A. Armando, S. Ranise, and M. Rusinowitch. A Rewriting Approach to Satisfiability Procedures. *Journal of Information and Computation — Special Issue on Rewriting Techniques and Applications (RTA'01)*, 183(2), June 2003.

[Bar03] C. W. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories*. PhD thesis, Stanford University, january 2003.

[BB01] G. J. Badros and A. Borning. The Cassowary Linear Arithmetic Constraint Solving Algorith. *ACM Transactions on Computer Human Interaction*, 8(4):267–306, december 2001.

[BB04] C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *LNCS*. Springer, 2004.

[BBC$^+$05a] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P.van Rossum, S. Schulz, and R. Sebastiani. An incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic. In *Proc. TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.

[BBC$^+$05b] 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.

[BBC$^+$05c] 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.

[BBC$^+$06a] M. Bozzano, R. Bruttomesso, A. Cimatti, A. Franzen, Z. Hanna, Z. Khasidashvili, A Palti, and R. Sebastiani. Encoding RTL Constructs for MathSAT: a Preliminary Report. In *Proc. PDPAR'05*, volume 144 of *ENTCS*. Elsevier, 2006.

[BBC$^+$06b] 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.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. CAV'99*, 1999.

[BCF$^+$06a] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, A. Santuari, and R. Sebastiani. To Ackermann-ize or not to Ackermann-ize? On Efficiently Handling Uninterpreted Function Symbols in $SMT(\mathcal{EUF} \cup \mathcal{T})$. In *Proc. LPAR'06*, volume 4246 of *LNAI*. Springer, 2006.

[BCF$^+$06b] 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.

[BCF$^+$07] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification Problems. In *Proc. CAV'07*, LNCS. Springer, 2007. To appear.

[BCLZ04] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic Theorem Proving for Predicate Abstraction Refinement. In *Proc. CAV'04*, volume 3114 of *LNCS*. Springer, 2004.

[BD94] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *Proc. CAV '94*, volume 818 of *LNCS*. Springer, 1994.

[BD02] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proc. ASP-DAC 2002*, pages 741–746. IEEE, 2002.

[BDL98] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Proc. DAC '98.* ACM Press, 1998.

[BdMS05] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In *Proc. CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

[BDS02a] 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.

[BDS02b] C. W. Barrett, D. L. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In *Frontiers of Combining Systems (FROCOS)*, LNAI. Springer, April 2002. S. Margherita Ligure, Italy.

[BE06] M. P. Bonacina and M. Echenim. Rewrite-Based Satisfiability Procedures for Recursive Data Structures. In *Proc. PDPAR'06*, ENTCS. Elsevier, 2006. To appear. Available at http://www.easychair.org/FLoC-06/floc-workshop-preproceedings.html.

[BFG$^+$05] C. W. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. D. Zuck. TVOC: A Translation Validator for Optimizing Compilers. In *Proc. CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

[BG94] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4), 1994.

[BGD03] S. Berezin, V. Ganesh, and D. L. Dill. An online proof-producing decision procedure for mixed-integer linear arithmetic. In *TACAS'03*, volume 2619 of *LNCS*, pages 521–536. Springer, 2003.

[BGN$^+$06] M. P. Bonacina, S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decidability and Undecidability Results for Nelson-Oppen and Rewrite-Based Decision Procedures. In *Proc. of IJCAR'06*, number 4130 in LNAI, 2006.

[BGV99] R.E. Bryant, S. German, and M.N. Velev. Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions. In *Proc. CAV'99*, volume 1633 of *LNCS*. Springer, 1999.

[BLS02] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Proc. CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[BMSX97] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications. In *Proc. UIST'97*, pages 87–96. ACM, 1997.

[BNOT06] 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.

[Bor97] A. Borälv. A Fully Automated Approach for Proving Safety Properties in Interlocking Software Using Automatic Theorem-Proving. In *Proceedings of the Second International ERCIM Workshop on Formal Methods for Industrial Critical Systems*, 1997.

[Bra01] R. Brafman. A simplifier for propositional formulas with many binary clauses. In *Proc. IJCAI01*, 2001.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BS97] R. J. Bayardo and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT instances. In *Proc. AAAI'97*, pages 203–208. AAAI Press, 1997.

[BST06] C. Barrett, I. Shikanian, and C. Tinelli. An Abstract Decision Procedure for Satisfiability in the Theory of Recursive Data Types. In *Proc. PDPAR'06*, ENTCS. Elsevier, 2006. To appear. Available at http://www.easychair.org/FLoC-06/floc-workshop-preproceedings.html.

[BT03] P. Baumgartner and C. Tinelli. The Model Evolution Calculus. In F. Baader, editor, *Proc. CADE-19*, number 2741 in LNAI, pages 350–364. Springer, 2003.

[BW01] A. Bockmayr and V. Weispfenning. Solving Numerical Constraints. In *Handbook of Automated Reasoning*, pages 751–842. MIT Press, 2001.

[BW03] F. Bacchus and J. Winter. Effective Preprocessing with Hyper-Resolution and Equality Reduction. In *Proc. Sixth International Symposium on Theory and Applications of Satisfiability Testing*, 2003.

[CABN97] W. Chan, R. J. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *Proc. CAV'97*, volume 1254 of *LNCS*, pages 316–327, Haifa, Israel, June 1997. Springer.

[CG99] B. V. Cherkassky and A. V. Goldberg. Negative-cycle detection algorithms. *Mathematical Programming*, 85(2):277–311, 1999.

[CGT03] C. Castellini, E. Giunchiglia, and A. Tacchella. SAT-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence*, 147(1-2):85–117, 2003.

[CM06a] S. Cotton and O. Maler. Fast and Flexible Difference Logic Propagation for DPLL(T). In *Proc. SAT'06*, volume 4121 of *LNCS*. Springer, 2006.

[CM06b] S. Cotton and O. Maler. Satisfiability Modulo Theory Chains with DPLL(T). Unpublished. Available from http://www-verimag.imag.fr/˜maler/, 2006.

[CMR97] D. Cyrluk, M. Oliver Möller, and H. Ruess. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Proceedings of CAV'97*, volume 1254 of *LNCS*, pages 60–71. Springer, 1997.

[CVCa] CVC. `http://verify.stanford.edu/CVC`.

[CVCb] CVCLite. `http://verify.stanford.edu/{CVC,CVCL,SVC}`.

[DdM06] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proc. CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier and MIT Press, 1990.

[DLL62] M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.

[dMR04] L. de Moura and H. Ruess. An Experimental Evaluation of Ground Decision Procedures. In *Proc. CAV'04*, volume 3114 of *LNCS*. Springer, 2004.

[dMRS02a] L. de Moura, H. Rueß, and M. Sorea. Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. In *Proc. of the 18th International Conference on Automated Deduction*, volume 2392 of *LNCS*, pages 438–455. Springer, July 2002.

[dMRS02b] L. de Moura, H. Rueß, and M. Sorea. Lemmas on Demand for Satisfiability Solvers. Proc. SAT'02, 2002.

[DNS05] D. Detlefs, G. Nelson, and J. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

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

[dRS04] L. deMoura, H. Ruess, and N. Shankar. Justifying Equality. In *Proc. PDPAR'04*, volume 68 of *ENTCS*. Elsevier, 2004.

[EB05] N. Een and A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *proc. SAT'05*, volume 3569 of *LNCS*. Springer, 2005.

[End72] H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[ES04] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[FDK98] F. Fallah, S. Devadas, and K. Keutzer. Functional Vector Generation for HDL Models Using Linear Programming and 3-Satisfiability. In *Proc. DAC'98*, pages 528–533, 1998.

[FJOS03] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem Proving Using Lazy Proof Explication. In *Proc. CAV 2003*, LNCS. Springer, 2003.

[FORS01] J.C: Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonizer and Solver. Proc. CAV'2001, 2001.

[GAG$^+$02] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proc. DAC'02*. ACM Press, 2002.

[GGST98] E. Giunchiglia, F. Giunchiglia, R. Sebastiani, and A. Tacchella. More evaluation of decision procedures for modal logics. In *Proc. Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, Trento, Italy, 1998.

[GGT01] 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.

[GHN$^+$04] 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.

[GMS98] E. Giunchiglia, A. Massarotto, and R. Sebastiani. Act, and the Rest Will Follow: Exploiting Determinism in Planning as Satisfiability. In *Proc. AAAI'98*, pages 948–953, 1998.

[GN02] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. In *Proc. DATE '02*, page 142, Washington, DC, USA, 2002. IEEE Computer Society.

[GS96a] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *CADE-13*, LNAI, New Brunswick, NJ, USA, August 1996. Springer Verlag.

[GS96b] F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for ALC. In *Proc. of the 5th International Conference on Principles of Knowledge Representation and Reasoning - KR'96*, Cambridge, MA, USA, November 1996.

[GS00] 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.

[GSK98] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 431–437, Madison, Wisconsin, 1998.

[GSZ$^+$98] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD Based Procedures for a Theory of Equality with Uninterpreted Functions. In *Proc. CAV'98*, volume 1427 of *LNCS*. Springer, 1998.

[GTG06] M. K. Ganai, M. Talupur, and A. Gupta. *SDSAT*: Tight integration of *small domain encoding* and *lazy* approaches in a separation logic solver. In *Proc. TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.

[HB05] J. Hoffmann and R. I. Brafman. Contingent Planning via Heuristic Forward Search witn Implicit Belief States. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, pages 71–80. AAAI, 2005.

[Hor98] I. Horrocks. The FaCT system. In H. de Swart, editor, *Proc. TABLEAUX-98*, volume 1397 of *LNAI*, pages 307–312. Springer, 1998.

[HS97] W. Harvey and P. Stuckey. A unit two variable per inequality integer constraint solver for constraint logic programming. In *Australian Computer Science Conference (Australian Computer Science Communications)*, pages 102–111, 1997.

[HV95] John N. Hooker and V. Vinay. Branching Rules for Satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

[IPC03] M. K. Iyer, G. Parthasarathy, and K.-T. Cheng. SATORI - A Fast Sequential SAT Engine for Circuits. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society, 2003.

[JN00] T. A. Junttila and I. Niemelä. Towards an Efficient Tableau Method for Boolean Circuit Satisfiability Checking. In *Proc. CL'00*, volume 1861 of *LNCS*. Springer, 2000.

[JS05] H. Jin and F. Somenzi. Prime clauses for fast enumeration of satisfying assignments to boolean circuits. In *Proc. DAC'05*. ACM Press, 2005.

[JW90] R.G. Jeroslow and J. Wang. Solving Propositional Satisfiability Problems. *Annals of Mathematics and Artificial Intelligence*, 1(1-4):167–187, 1990.

[KGP01] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean Reasoning. In *Proc. DAC '01*. ACM Press, 2001.

[Kha79] L. G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[KMS96] H. Kautz, D. McAllester, and B. Selman. Encoding Plans in Propositional Logic. In *Proc. KR'96*, 1996.

[KOSS04] D. Kroening, J. Ouaknine, S. Seshia, and O. Strichman. Abstraction-Based Satisfiability Solving of Presburger Arithmetic. In *Proc. CAV'04*, volume 3114 of *LNCS*, pages 308–320. Springer, 2004.

[KRRT05] H. Kirchner, S. Ranise, C. Ringeissen, and D.-K. Tran. On Superposition-Based Satisfiability Procedures and their Combination. In *Proc.ICTAC'05*, volume 3722 of *LNCS*. Springer, 2005.

[KRRT06] H. Kirchner, S. Ranise, C. Ringeissen, and D. K. Tran. Automatic Combinability of Rewriting-Based Satisfiability Procedures. In *Proc. LPAR'06*, volume 4246 of *LNAI*. Springer, 2006.

[KS06] H. Kim and F. Somenzi. Finite Instantiations for Integer Difference Logic. In *proc FMCAD'06*. ACM Press, 2006.

[LA97] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371, 1997.

[LD60] H. Land and A.G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[LHS04] B. Li, M. S. Hsiao, and S. Sheng. A Novel SAT All-Solutions Solver for Efficient Preimage Computation. In *Proc. DATE'04*. IEEE Computer Society, 2004.

[Li00] C. M. Li. Integrating equivalency reasoning into davis-putnam procedure. In *AAAI: 17th National Conference on Artificial Intelligence*. AAAI / MIT Press, 2000.

[LM05] 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.

[LM06] S. K. Lahiri and M. Musuvathi. An Efficient Nelson-Oppen Decision Procedure for Difference Constraints over Rationals. In *Proc. Third Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR'05)*, volume 144 of *ENTCS*. Elsevier, 2006.

[LS04] S. K. Lahiri and S. A. Seshia. The UCLID Decision Procedure. In *Proc. CAV'04*, volume 3114 of *LNCS*, 2004.

[McM02] K. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *Proc. CAV '02*, number 2404 in LNCS. Springer, 2002.

[MLAH99] J. Moeller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. In *Proc. Workshop on Symbolic Model Checking (SMC), FLoC'99*, Trento, Italy, July 1999.

[MMZ$^+$01] M. W. Moskewicz, C. F. Madigan, Y. Z., L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, 2001.

[MNAM02] M. Mahfoudh, P. Niebert, E. Asarin, and O. Maler. A Satisfibaility Checker for Difference Logic. In *Proceedings of SAT-02*, pages 222–230, 2002.

[MR98] M. O. Möller and Harald Ruess. Solving bit-vector equations. In *Proceedings of FMCAD'98*, 1998.

[MZ03] Z. Manna and C. Zarba. Combining Decision Procedures. In *Formal Methods at the Crossroads: from Panacea to Foundational Support*, volume 2787 of *LNCS*. Springer, 2003.

[NO79] C. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1(2):245–257, 1979.

[NO80] G. Nelson and D. C. Oppen. Fast Decision Procedures Based on Congruence Closure. *Journal of the ACM*, 27(2):356–364, 1980.

[NO03] R. Nieuwenhuis and A. Oliveras. Congruence closure with integer offsets. In *In 10th Int. Conf. Logic for Programming, Artif. Intell. and Reasoning (LPAR)*, volume 2850 of *LNAI*, pages 78–90. Springer, 2003.

[NO05a] R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In *Proc. CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

[NO05b] 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.

[NOT05] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories. In *Proc. LPAR'04*, volume 3452 of *LNCS*. Springer, 2005.

[NOT06] 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.

[Ome] Omega. `http://www.cs.umd.edu/projects/omega`.

[Opp80] D.C. Oppen. Reasoning about Recursively Defined Data Structures. *Journal of the ACM*, 27(3):403–411, 1980.

[Pap81] C. H. Papadimitriou. On the complexity of integer programming. *JACM*, 28(4):765–768, 1981.

[PICW04] G. Parthasarathy, M. K. Iyer, K.-T. Cheng, and L.-C. Wang. An efficient finite-domain constraint solver for circuits. In *Proc. DAC'04*. ACM Press, 2004.

[PS98] P. F. Patel-Schneider. DLP system description. In *Proc. DL-98*, pages 87–89, 1998.

[Pug91] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, New York, NY, USA, 1991. ACM Press.

[RD03] S. Ranise and D. Deharbe. Light-Weight Theorem Proving for Debugging and Verifying Units of Code-. In *Proc. of the International Conference on Software Engineering and Formal Methods SEFM03*. IEEE Computer Society Press, 2003.

[Rey02] J. Reynolds. Separation logic: a logic for shared mutable data structures, 2002.

[RS01] H. Rueßand N. Shankar. Deconstructing Shostak. In *Proc. LICS '01*. IEEE Computer Society, 2001.

[RS04] Rueß and Natarajan Shankar. Solving linear arithmetic constraints. Technical Report CSL-SRI-04-01, SRI International, Computer Science Laboratory, 333 Ravenswood Ave, Menlo Park, CA, 94025, January 2004. revised, August 2004.

[RT06a] S. Ranise and C. Tinelli. Satisfiability Modulo Theories. *In Trends and Controversies - IEEE Intelligent Systems Magazine*, 21(6):71–81, 2006.

[RT06b] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2006.

[RT06c] S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

[SBD02] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A Cooperating Validity Checker. In *Proc. CAV'02*, number 2404 in LNCS. Springer Verlag, 2002.

[SBSV96] P. Stephan, R. Brayton, , and A. Sangiovanni-Vincentelli. Combinational Test Generation Using Satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15:1167–1176, 1996.

[Sch02] S. Schulz. E - A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3), 2002.

[SDBL01] A. Stump, D. L. Dill, C. W. Barrett, and J. Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *Proc LICS '01*, pages 29–37. IEEE Computer Society, 2001.

[Seb01] R. Sebastiani. Integrating SAT Solvers with Math Reasoners: Foundations and Basic Algorithms. Technical Report 0111-22, ITC-IRST, Trento, Italy, November 2001. http://sra.itc.it/tr/Seb01.pdf.

[Sho79] R. Shostak. A Pratical Decision Procedure for Arithmetic with Function Symbols. *Journal of the ACM*, 26(2):351–360, 1979.

[Sho84] R.E. Shostak. Deciding Combinations of Theories. *Journal of the ACM*, 31:1–12, 1984.

[SLB03] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A Hybrid SAT-Based Decision Procedure for Separation Logic with Uninterpreted Functions. In *Proc. DAC'03*, 2003.

[SMT05] SMT-COMP'05: 1st Satisfiability Modulo Theories Competition, 2005. `http://www.csl.sri.com/users/demoura/smt-comp/2005/`.

[SMT06] SMT-COMP'06: 2nd Satisfiability Modulo Theories Competition, 2006. `http://www.csl.sri.com/users/demoura/smt-comp/`.

[Smu68] R. M. Smullyan. *First-Order Logic*. Springer-Verlag, NY, 1968.

[SR02] N. Shankar and Harald Rueß. Combining shostak theories. Invited paper for Floc'02/RTA'02, 2002.

[SS90] G. Stalmarck and M. Saflund. Modelling and Verifying Systems and Software in Propositional Logic. *Ifac SAFECOMP'90*, 1990.

[SS96] J. P. M. Silva and K. A. Sakallah. GRASP - A new Search Algorithm for Satisfiability. In *Proc. ICCAD'96*, 1996.

[SS05] H. M. Sheini and K. A. Sakallah. A Scalable Method for Solving Satisfiability of Integer Linear Arithmetic Logic. In *Proc. SAT'05*, volume 3569 of *LNCS*. Springer, 2005.

[SS06a] H. M. Sheini and K. A. Sakallah. A Progressive Simplifier for Satisfiability Modulo Theories. In *Proc. SAT'06*, volume 4121 of *LNCS*. Springer, 2006.

[SS06b] H. M. Sheini and K. A. Sakallah. From Propositional Satisfiability to Satisfiability Modulo Theories. Invited lecture. In *Proc. SAT'06*, volume 4121 of *LNCS*. Springer, 2006.

[SSB02] O. Strichman, S. Seshia, and R. Bryant. Deciding separation formulas with SAT. In *Proc. of Computer Aided Verification, (CAV'02)*, LNCS. Springer, 2002.

[Str00] O. Strichman. Tuning SAT checkers for Bounded Model Checking. In *Proc. CAV00*, volume 1855 of *LNCS*, pages 480–494. Springer, 2000.

[Str02] O. Strichman. On Solving Presburger and Linear Arithmetic with SAT. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD 2002)*, LNCS. Springer, 2002.

[SV98] 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.

[TBW04] C. Thiffault, F. Bacchus, and T. Walsh. Solving Non-clausal Formulas with DPLL Search. In *proc. 7th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2004*, LNCS. Springer, 2004.

[Tin02] C. Tinelli. A DPLL-based Calculus for Ground Satisfiability Modulo Theories. In *Proc. JELIA-02*, volume 2424 of *LNAI*, pages 308–319. Springer, 2002.

[TSSP04] M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range Allocation for Separation Logic. In *Proc. CAV'04*, volume 3114 of *LNCS*. Springer, 2004.

[UCL] UCLID. `http://www-2.cs.cmu.edu/∼uclid`.

[VB99] M. N. Velev and R. E. Bryant. Exploiting Positive Equality and Partial Non-Consistency in the Formal Verification of Pipelined Microprocessors. In *Design Automation Conference*, pages 397–401, 1999.

[VB03] M. N. Velev and R. E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, 2003.

[WGG06] C. Wang, A. Gupta, and M. Ganai. Predicate learning and selective theory deduction for a difference logic solver. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*. ACM Press, 2006.

[WIGG05] C. Wang, F. Ivancic, M. K. Ganai, and A. Gupta. Deciding Separation Logic Formulae by SAT and Incremental Negative Cycle Elimination. In *Proc. LPAR'05*, volume 3835 of *LNCS*, pages 322–336. Springer, 2005.

[WW99] S. Wolfman and D. Weld. The LPSAT Engine & its Application to Resource Planning. In *Proc. IJCAI*, 1999.

[WW00] S. Wolfman and D. Weld. Combining linear programming and satisfiability solving for resource planning. *Knowledge Engineering Review*, 2000.

[YKTB00] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A Library for Composite Symbolic Representation. In *Proc. TACAS2001*, volume 2031 of *LNCS*. Springer Verlag, 2000.

[YM06] Y. Yu and S. Malik. Lemma Learning in SMT on Linear Constraints. In *Proc. SAT'06*, volume 4121 of *LNCS*. Springer, 2006.

[ZKC01] Z. Zeng, P. Kalla, and M. Ciesielski. LPSAT: a unified approach to RTL satisfiability. In *Proc. DATE '01*. IEEE Press, 2001.

[ZM02] 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.

[ZMMM01] 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.