Journal on Satisfiability, Boolean Modeling and Computation 3 (2007) 141–224

Lazy Satisfiability Modulo Theories

Roberto Sebastiani * roberto.sebastiani@disi.unitn.it
Dipartimento di Ingegneria e Scienza dell'Informazione (DISI)
Università di 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 reference guide for developers of SMT tools, and to stimulate the cross-fertilization of techniques and ideas among different communities.

Keywords: *propositional satisfiability, satisfiability modulo theories, decision procedures*

Submitted April 2007; revised December 2007; published December 2007

* This survey work has benefited from important discussions with A. Armando, C. Barrett, M. Bozzano, R. Bruttomesso, A. Cimatti, A. Franzen, L. de Moura, S. Ghilardi, A. Griggio, S. Krstic, R. Nieuwenhuis, A. Oliveras, S. Ranise, M. Roveri, O. Strichman, A. Tacchella, C. Tinelli, 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" (disi.unitn.it/~rseba/DIDATTICA/Tutorials/Slides_tutorial_cade03.pdf), at the course "Efficient Boolean Reasoning" at 2003-2005 Int. ICT school of Trento and at 2006 Int. BIT School in Brixen (disi.unitn.it/~rseba/DIDATTICA/SAT_BASED06/), and at my invited talk at FroCoS'07 [154]. This work has been partly supported by ORCHID, a project sponsored by Provincia Autonoma di Trento, and by a grant from Intel Corporation.

©2007 Delft University of Technology and the authors.

Contents

1 Introduction 144
1.1 Satisfiability Modulo Theories - SMT . . . 144
1.2 Lazy SMT = SAT + $\mathcal{T}$-*solvers* . . . 144
1.3 Motivations and goals of the paper . . . 145
1.4 Content of the paper . . . 147

2 Theoretical background 148
2.1 Background on first-order logic and theories . . . 148
2.1.1 Combination of theories . . . 149
2.2 Truth assignments and propositional satisfiability in $\mathcal{T}$ . . . 150
2.3 Enumerators and $\mathcal{T}$-*solvers* . . . 151

3 Basics on SAT solvers 153
3.1 Modern conflict-driven DPLL . . . 153
3.2 The Abstract-DPLL logical framework . . . 157

4 Basics on theory solvers 159
4.1 Important features of $\mathcal{T}$-*solvers* . . . 159
4.1.1 Model generation . . . 159
4.1.2 Conflict set generation . . . 159
4.1.3 Incrementality and Backtrackability . . . 160
4.1.4 Deduction of unassigned literals . . . 160
4.1.5 Deduction of interface equalities . . . 161
4.2 Some relevant theories and $\mathcal{T}$-*solvers* . . . 161
4.2.1 Equality and Uninterpreted Functions . . . 161
4.2.2 Linear arithmetic . . . 162
4.2.3 Difference logic . . . 163
4.2.4 Unit-Two-Variable-Per-Inequality . . . 165
4.2.5 Bit vectors . . . 166
4.2.6 Other theories of interest . . . 166
4.3 Layered $\mathcal{T}$-*solvers* . . . 167

5 Integrating DPLL and $\mathcal{T}$-*solvers* 169
5.1 A basic integration schema . . . 169
5.2 The offline approach to integration . . . 170
5.3 The online approach to integration . . . 171
5.4 The Abstract-DPLL Modulo Theories logical framework . . . 174

6 Optimizing the integration of DPLL and $\mathcal{T}$-*solvers* 176
6.1 Normalizing $\mathcal{T}$-atoms. . . . 177
6.2 Static learning . . . 177
6.3 Early pruning . . . 178
6.3.1 Selective or intermittent early pruning . . . 179
6.3.2 Weakened early pruning . . . 179

6.3.3 Eager early pruning . . . 180
6.4 $\mathcal{T}$-propagation . . . 180
6.5 $\mathcal{T}$-backjumping . . . 181
6.6 $\mathcal{T}$-learning . . . 183
6.7 Splitting on demand . . . 184
6.8 Clustering . . . 185
6.9 Reduction of assignments to prime implicants . . . 185
6.10 Pure-literal filtering . . . 185
6.11 $\mathcal{T}$-deduced-literal filtering . . . 186

7 Discussion **187**
7.1 Guidelines and tips . . . 187
7.1.1 Some general guidelines . . . 187
7.1.2 Offline vs. online integration . . . 188
7.1.3 To $\mathcal{T}$-propagate or not to $\mathcal{T}$-propagate? . . . 188
7.2 Problems of using modern conflict-driven DPLL in SMT . . . 189
7.2.1 Generating partial assignments . . . 189
7.2.2 Avoiding ghost literals . . . 190
7.2.3 Drawbacks of modern $\mathcal{T}$-backjumping . . . 191
7.2.4 Implementing $\mathcal{T}$-propagation . . . 193
7.2.5 DPLL branching heuristics for SMT . . . 193

8 Lazy SMT for combinations of theories **194**
8.1 The Nelson-Oppen formal framework . . . 194
8.2 The Nelson-Oppen combination procedure . . . 196
8.3 The Delayed Theory Combination procedure . . . 200
8.4 SMT($\mathcal{EUF} \cup \mathcal{T}$) via Ackermann's expansion . . . 204

9 Related approaches for SMT **206**
9.1 Alternative Enumerators for lazy SMT . . . 206
9.1.1 Why DPLL? . . . 206
9.1.2 OBDD-based SMT solvers . . . 206
9.1.3 Circuit-based techniques . . . 207
9.2 The rewrite-based approach for building $\mathcal{T}$-*solvers* . . . 207
9.3 The eager approach to SMT . . . 209
9.4 Mixed eager/lazy approaches . . . 210

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 [185] and temporal reasoning [5] to formal verification, the latter including verification of pipelines and of circuits at Register-Transfer Level (RTL) [49, 141, 36], of proof obligations in software systems [144, 81], of compiler optimizations [26], of real-time embedded systems [12, 66, 11].

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 [158], BarceLogic [133], CVCLite/CVC3 [17], DLSAT [123], haRVey [144], MathSAT [37], Sateen [108], SDSAT [83] Simplify [70], TSAT++ [6], UCLID [117], Yices [71], Verifun [79], Zapato [16]), Z3 [63]. An amount of benchmarks, mostly derived from verification problems, is available at the *SMT-LIB* official page [147, 148]. 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 [163, 28, 129, 95, 75, 74]). 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 ATPG [30, 126, 168]; other application areas include, e.g., the verification of safety-critical systems [167, 34], and AI planning in its classical formulation [106], and in its extensions to non-deterministic do-

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 "$\leq$". See, e.g., [146] for some more discussion on this issue.
2. SMT'07, previously called "PDPAR". See `http://www.lsi.upc.edu/~oliveras/smt07/`.
3. SMT-COMP05/06/07 [18]. See `http://http://www.smtcomp.org/`.

mains [50, 98]. 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 [130, 131, 138, 139] and Shostak [161, 162], 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., [130, 131, 138, 139, 161, 162, 78, 25, 157]). 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., [92, 100, 5, 185, 66, 19, 9, 176, 123, 79, 85, 37, 160]). Most such systems have been implemented on top of SAT techniques based on variants of the DPLL algorithm [60, 59, 163, 28, 129, 95, 75, 74]. [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 to the *eager* approach to $SMT(\mathcal{T})$, consisting on encoding an SMT formula into an equivalently-satisfiable Boolean formula, and on feeding the result to a SAT solver (see, e.g., [181, 47, 171, 170, 156]). All the most extensive empirical evaluations performed in the last years [85, 65, 133, 38, 18, 164, 165] confirm the fact that currently all the most efficient SMT tools are based on the lazy approach.

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

4. Notice that the main pattern for integrating decision procedures and DPLL, and many techniques for optimizing this integration, were conceived and implemented much earlier, mostly in the domain of modal and description logics [7, 92, 100, 89], and have been imported into the SMT domain only lately [5]. See [154] for more details on this fact.

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 contexts. For instance, the technique called "$\mathcal{T}$-backjumping" (see §6) has been invented for description logics [100], and then "reinvented" in both the communities of resource planning [185] and formal verification [172, 67], in different moments and without cross-citations.

We notice also that the code of most SMT tools is not publicly available, and that some techniques which are implemented in these tools are not really described in the papers, so that it is often difficult for an outsider to access to this knowledge. (E.g., a significant amount of the author's knowledge comes from his experience in developing the KSAT and MathSAT tools, from discussions with collaborators and from personal communications from the authors of other tools.)

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 researchers 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 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 $\mathcal{T}$ is the combination of two or more different theories. We present and discuss the main techniques for integrating 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 to 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 [155, 93, 153, 10, 40], 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 [69].

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 built 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 [76]. 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 the identity on the underlying domain. Consequently, $=$ is 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

Here we introduce some concepts which are of primary interest for $SMT(\mathcal{T})$ when $\mathcal{T}$ is a combination of theories (see §8).

A conjunction Γ of $\mathcal{T}$-literals in a theory $\mathcal{T}$ is *convex* 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 $\mathcal{T}$. 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. 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 (see [25]).

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.

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}". (For simplicity and w.l.o.g., we assume that identities like $v_i = v_i$ are simplified into $\top$, and that $v_i = v_j$ and $v_j = v_i$ are uniquely represented as $v_i = v_j$ s.t. $v_i \prec v_j$ for some total order $\prec$.)

---

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 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 $\mathcal{T}2\mathcal{B}$ ("Theory-to-Boolean") and its inverse $\mathcal{B}2\mathcal{T} := \mathcal{T}2\mathcal{B}^{-1}$ ("Boolean-to-Theory"), s.t. $\mathcal{T}2\mathcal{B}$ 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. $\mathcal{T}2\mathcal{B}$ and $\mathcal{B}2\mathcal{T}$ 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 $\mathcal{T}2\mathcal{B}(\mu) \models \mathcal{T}2\mathcal{B}(\varphi)$, that is, for all sub-formulas φ_1, φ_2 of φ:

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

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 formula 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 its Boolean abstraction $\mathcal{T}2\mathcal{B}(\varphi)$:*

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

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.

Following [155], 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,

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

$\mathcal{M} := \{\mu_1, \ldots, \mu_n\}$ is complete in the sense that, for for every total assignment η s.t. $\eta \models_p \varphi$, there exists $\mu_j \in \mathcal{M}$ s.t. $\mu_j \subseteq \eta$. Thus $\mathcal{M}$ can be seen as a compact representation of the whole set of total assignments propositionally satisfying φ. The following fact can be proved straightforwardly (see [155]).

Proposition 2.2. *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 $\mu_j \in \mathcal{M}$.*

Finally, we also notice the following fact (see [153]).

Proposition 2.3. *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 $\mu'_j \in \mathcal{M}'$.

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

By proposition 2.2, 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$(\{(x-y=3),(y-z\leq 4),\neg(x-z\leq 8)\})$ 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$(\{(x-2y=3),(4y-2z<9),\neg(x-z\leq 7)\})$ returns Sat;

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

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

$\mathcal{AR}$: $\mathcal{AR}$-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 [130, 161, 78, 25, 157].

Remark 2.4. *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.*

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

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-Logemann-Loveland (*DPLL*)* procedure [60, 59]. Unlike with "classic" representation of DPLL [60, 59], modern conflict-driven DPLL implementation are non-recursive, and are based on very efficient 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 [193] for an overview.)

Remark 3.1. *Modern* DPLL *engines can be partitioned into two main families: conflict-driven* DPLL *[163], in which the search is driven by the analysis of the conflicts at every failed branch, and look-ahead* DPLL *[122], in which the search is driven by a look-ahead procedure evaluating the reduction effect of the selection of each variable in a group. In lazy SMT, however, it seems that all modern tools we are aware of have adopted the former schema for their embedded SAT engine (see §5.3). Although (to the best of our knowledge) this issue has never been explicitly discussed in the SMT literature, we conjecture that the main reason for this fact is that the conflict-driven schema fits more naturally with the lazy SMT schema, in which the Boolean component of search is driven by the theory conflicts which are found by specialized solvers (see §5.3, §6.5 and §6.6). The possibility of adopting a look-ahead schema within a lazy SMT solver is still an open research issue, as hinted in §7.2.5. For these reasons, hereafter we restrict our discussion to the conflict-driven* DPLL *schema, and in the next sections we often omit the adjective "conflict-driven" when referring to* DPLL*.*

3.1 Modern conflict-driven DPLL

A high-level schema of a modern conflict-driven DPLL engine, adapted from [193], 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.

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* and 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,

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 $\varphi \wedge \mu$.

```
1.   SatValue DPLL (Bool_formula φ, assignment & μ) {
2.      if (preprocess(φ,μ)==Conflict);
3.         return Unsat;
4.      while (1) {
5.         decide_next_branch(φ,μ);
6.         while (1) {
7.            status = deduce(φ,μ);
8.            if (status == Sat)
9.               return Sat;
10.           else if (status == Conflict) {
11.              blevel = analyze_conflict(φ,μ);
12.              if (blevel == 0)
13.                 return Unsat;
14.              else backtrack(blevel,φ,μ);
15.           }
16.           else break;
17.  } } }
```

Figure 1. Schema of a modern conflict-driven DPLL engine.

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., [41, 13, 74]). 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 [102] used to select a new literal at each branching point, picking the literal occurring most often in the minimal-size clauses (see, e.g., [99]). The heuristic implemented in SATZ [121] selects a candidate set of literals, performs 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 first (e.g., action variables in SAT-based planning [91], primary inputs in bounded model checking [169]). Modern conflict-driven DPLL solvers adopt evolutions of the VSIDS heuristic [129,

95, 75], 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* [129]) allow for extremely efficient implementations. Other forms of deductions (and formula simplification) are, e.g., *pure literal rule* (now obsolete), on-line equivalence reasoning [120], and variable and clause elimination [74].

It is important to notice that most modern conflict-driven DPLL solvers do not return Sat when all clauses are satisfied, but only when all variables are assigned truth values. [9.] As a consequence, modern conflict-driven 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 [163, 28, 192, 193]. Each literal is tagged with its *decision level*, that is, the literal corresponding to the nth decision and the literals derived by unit-propagation after that decision are labeled with n; 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 clause C is falsified by the current assignment —in which case we say that a *conflict* occurs and C is the *conflicting clause*— a *conflict clause* C' is computed from C s.t. C' contains only one literal l_u which has been assigned at the last decision level. C' is computed starting from $C' = C$ by iteratively resolving C' with the antecedent clause C_l of some literal l in C' (typically the last-assigned literal in C', see [193]), until some stop criterion is met. E.g., with the *1st UIP strategy* it is always picked the last-assigned literal in C', and the process stops as soon as C' contains only one literal l_u assigned at the last decision level; with the *last UIP strategy*, l_u must be the last decision literal.

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. $l_1 \xmapsto{C_l} l, \ldots, l_n \xmapsto{C_l} l$ 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; given a partition of the graph with all decision literals on one side and the conflict on the other, the set of the source nodes of all arcs intersecting the borderline of the partition represents a conflict set. A node l_u in an implication graph is a *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_u; [10.] the most recent decision node is a UIP (the *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 a partition corresponding to the last [resp. 1st] UIP.

9. This is mostly due to the fact that the two-watched-literal scheme [129] 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.)

10. An UIP is also called an *articulation point* in graph theory (see, e.g., [53]).

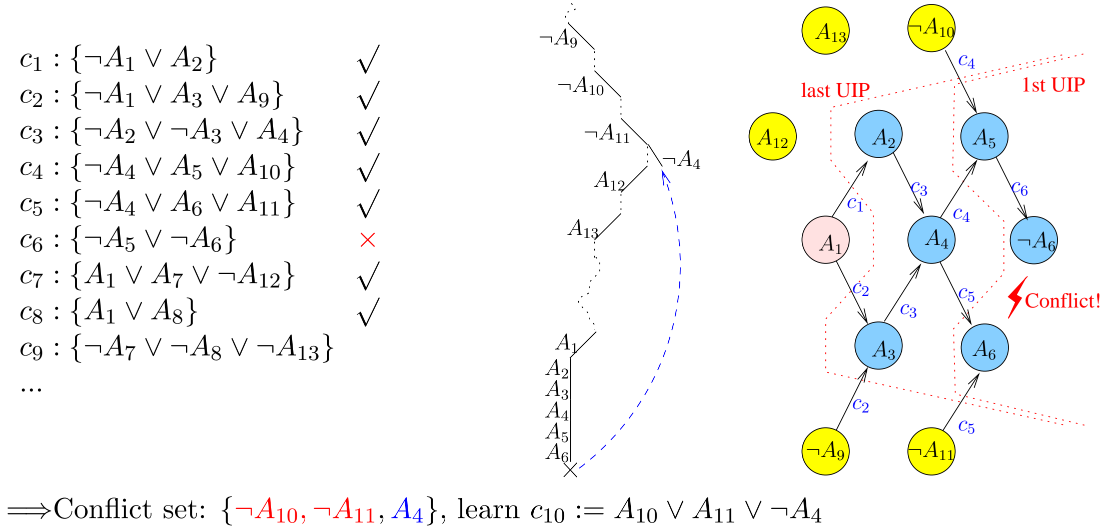

Figure 2. Example of learning and backjumping based on the 1st UIP stategy.

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 modern conflict-driven implementations, DPLL backtracks to the highest point in the stack where the literal l_u in the learned clause C' is not assigned, and unit-propagates l_u. We refer the reader to [192] for an overview on backjumping and learning strategies.

Example 3.2. *Consider a Boolean formula containing the clauses $c_1 \ldots c_9$ in Figure 2, and assume at some point $\mu := \{\ldots, \neg A_9, \ldots, \neg A_{10}, \ldots, \neg A_{11}, \ldots, A_{12}, \ldots, A_{13}, \ldots, A_1\}$.* [11.] *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' each time with the antecedent clause of the last-assigned literal l in C', until it contains only one literal assigned at the current decision level (1st UIP):*

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

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.*

Learning must be used with some care, because it may cause an explosion in the size of φ. To avoid this problem, modern conflict-driven DPLL tools implement techniques for

11. Here and in other examples "..." mean that there may be possibly many other literals in the assignment, which play no direct role in the discourse.

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

Figure 3. The Abstract-DPLL logical framework from [136]. In the Backjump rule, C and $C' \vee l'$ represent the conflicting and the conflict clause respectively.

discharging learned clauses when necessary [163, 28]. 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 [96]; 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 conflict-driven 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

[176, 135, 133, 136] proposed an abstract rule-based formulation of DPLL (*Abstract* DPLL) and of DPLL-based lazy SMT systems (*Abstract* DPLL *Modulo Theories*), 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. [12.]

12. The formalization of the rules in [135, 133, 136] changes slightly from paper to paper. Here we report the most-recent one from [136]. We have adapted the notation to that used in this paper.

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 [28] and [96].

The only non-obvious rule is Backjump, which deserves some more explanation: if a branch $\mu \cup \{l\} \cup \mu'$ falsifies one clause C (the conflicting clause), and a conflict clause $C' \vee l'$ [13.] can be computed from C s.t. $(C' \vee l')$ is entailed by $\varphi \wedge C$, $\neg C' \subseteq \mu$, $l' \notin \mu$, and l' or $\neg l'$ occur in φ or in $\mu \cup \{l\} \cup \mu'$, then it is possible to backjump up to μ, and hence unit-propagate l' on the conflict clause $(C' \vee l')$.

Example 3.3. *Consider the problem in Example 3.2 and Figure 2. The execution can be represented in Abstract-*DPLL *as follows:*

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

$c_1, ..., c_{10}$ *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. [135, 136] provide a group of results about termination, correctness and completeness of various configurations. Importantly, notice only the second, third and fourth rules are strictly necessary for correctness and completeness [135]. We refer the reader to [135, 133, 136] for further details. The Abstract-DPLL Modulo Theories framework will be described in §5.4.

13. Also called the *backjump clause* in [136].

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 [130, 131, 161, 162], 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 an $SMT(\mathcal{T})$ 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. [14.] 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 return a (minimal) conflict set causing the conflict:*

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

14. Theory conflict sets are also called *reasons*, *justifications*, *proofs* or *infeasible sets* by some authors; minimal theory conflict sets are also called *irreducible infeasible sets* in [188].

For instance, there exist conflict-set-producing variants for the Bellman-Ford algorithm for $\mathcal{DL}$, [52], for the Simplex LP procedures for $\mathcal{LA}(\mathbb{Q})$ [15] and for the congruence closure algorithm for $\mathcal{EUF}$ [132]. (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) [37]:

$\mathcal{T}$*-solver* (μ_1)	$\Longrightarrow$ Sat	Undo μ_4, μ_3, μ_2	
$\mathcal{T}$*-solver* $(\mu_1 \cup \mu_2)$	$\Longrightarrow$ Sat	$\mathcal{T}$*-solver* $(\mu_1 \cup \mu_2')$	$\Longrightarrow$ Sat
$\mathcal{T}$*-solver* $(\mu_1 \cup \mu_2 \cup \mu_3)$	$\Longrightarrow$ Sat	$\mathcal{T}$*-solver* $(\mu_1 \cup \mu_2' \cup \mu_3')$	$\Longrightarrow$ Sat
$\mathcal{T}$*-solver* $(\mu_1 \cup \mu_2 \cup \mu_3 \cup \mu_4)$	$\Longrightarrow$ Unsat	...	

Thus, a key efficiency issue of $\mathcal{T}$*-solver* is that of being *incremental* and *backtrackable*. [15] *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 $\mathcal{EUF}$ [132], of the Bellman-Ford algorithm for $\mathcal{DL}$ [52, 133], and of the Simplex LP procedure for $\mathcal{LA}(\mathbb{Q})$ [15, 71]. (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* $\{\neg(2x_2 - x_3 > 2), \neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4)\}$ *(1st, 4th and 7th rows in* φ*); then* $\mathcal{LA}(\mathbb{Q})$*-solver returns* Sat *and may perform and return the deduction*

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

For instance, for $\mathcal{EUF}$, the computation of congruence closure allows for efficiently deducing positive equalities [132]; for $\mathcal{DL}$, a very efficient implementation of a deduction-complete $\mathcal{T}$*-solver* has been presented by [133, 54]; for $\mathcal{LA}$ the task is much harder, and only $\mathcal{T}$*-solvers* capable of incomplete forms of deduction have been presented [71]. (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* [70]. In practice, apart from some application [5], plunging is very inefficient.

15. The latter feature is also called *resettable* in other contests (e.g., in [130]).

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 $\{\dots, (v_1 \geq 0), (v_1 \leq 1), (v_3 = 0), (v_4 = 1), \dots\}$, 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* [162]. Intuitively, a *canonizer* $\texttt{canon}_{\mathcal{T}}$ for a theory $\mathcal{T}$ is a function which maps a term t into another term $\texttt{canon}_{\mathcal{T}}(t)$ in *canonical* form, that is, $\texttt{canon}_{\mathcal{T}}$ 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 $\texttt{canon}_{\mathcal{T}}(t_1)$ and $\texttt{canon}_{\mathcal{T}}(t_2)$ 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 [124].) We assume the notions of stably-infinite and convex theories described in §2.1.1.

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 the identity on the underlying domain. In particular, "=" represents 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:

$$\forall x.\ (x = x), \qquad \forall x, y.\ (x = y \rightarrow y = x), \qquad \forall x, y, z.\ ((x = y \wedge y = z) \rightarrow x = z) \quad (6)$$

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

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 ($\mathcal{EUF}$)* [16] is the quantifier-free F.O. theory with equality with no restrictions on Σ. Semantically, no axioms other than (6)

16. Simply called "the theory of equality" by some authors (e.g., [124]).

and (7) are provided. If Σ contains no uninterpreted functions or predicates, then (7) are 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 [1].

An $\mathcal{E}$-solver can be simply implemented on top of the standard Union-Find algorithm (see, e.g., [134]). 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., [70, 132, 134]). The algorithm in [132] 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) constant symbols, each labeling one value in $\mathbb{Q}$ and $\mathbb{Z}$ respectively. [17.] The atomic expressions are interpreted according to the standard semantics of linear arithmetic on $\mathbb{Q}$ and $\mathbb{Z}$ respectively. (See, e.g., [124] 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 [107]. 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., [35, 68, 70, 151, 71]).

$\mathcal{LA}(\mathbb{Z})$ is stably-infinite and non-convex. [18.] The $\mathcal{LA}(\mathbb{Z})$-satisfiability of sets of quantifier-free literals is decidable and NP-complete [140]. Many algorithms have been conceived, involving techniques like Euler's reduction, Gomory-cuts application, Fourier-Motzkin algorithm, branch-and-bound [118]. Notably, the Omega library [137] 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 [143]. 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., [29, 71]).

Remark 4.5. *It is very important to remark that, in order to avoid incorrect results due to numerical errors and to overflows, all $\mathcal{T}$-solvers for $\mathcal{LA}(\mathbb{Q})$, $\mathcal{LA}(\mathbb{Z})$ and their subtheories which are based on numerical algorithms must be implemented on top of infinite-precision-arithmetic software packages.*

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

17. Notice that in $\mathcal{LA}(\mathbb{Q})$ we can assume w.l.o.g. that all constant symbols a_i are in $\mathbb{Z}$ because, if this is not so, then we can multiply all coefficients a_is in every atom by their greatest common denominator.

18. 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 $\mu \not\models_{\mathcal{LA}(\mathbb{Z})} (x = x_1)$.

4.2.3 Difference logic

The *theory of differences* ($\mathcal{DL}$) [19.] 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 $(x_1 - x_2 \bowtie a)$, s.t. $\bowtie \in \{\leq, <, \neq, =, \geq, >\}$, and the a is an (interpreted) constant symbol labeling one value in $\mathbb{Q}$ and $\mathbb{Z}$ respectively. [20.]

All $\mathcal{DL}$ literals can be rewritten in terms of *positive difference inequalities* $(x - y \leq a)$ only. First, all literals are rewritten into Boolean combinations of difference inequalities, by applying rules like, e.g., $(x - y = a) \Longrightarrow ((x - y \leq a) \wedge (y - x \leq -a))$, $(x - y > a) \Longrightarrow \neg(x - y \leq a)$, $(x - y \neq a) \Longrightarrow (\neg(x - y \leq a) \vee \neg(y - x \leq -a))$. 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 [6]. Notice that disequalities may require being split into the disjunction of two difference inequalities. (With $\mathcal{DL}(\mathbb{Q})$, this step can be avoided, as described below.)

$\mathcal{DL}(\mathbb{Q})$ is stably-infinite and convex. The $\mathcal{DL}(\mathbb{Q})$-satisfiability problem of sets of quantifier-free difference inequalities is decidable and polynomial; thanks to the convexity of $\mathcal{DL}(\mathbb{Q})$, the $\mathcal{DL}(\mathbb{Q})$-satisfiability of sets of quantifier-free difference inequalities, equalities and disequalities —and hence the problem of deducing interface equalities— is also polynomial. Equalities can be split into a conjunction of inequalities, as above. As with $\mathcal{LA}(\mathbb{Q})$, in $\mathcal{DL}(\mathbb{Q})$ disequalities can instead be treated separately and handled one-by-one. In fact, thanks to the convexity of $\mathcal{DL}(\mathbb{Q})$, $\mu \cup \bigcup_i \{(t_i \neq s_i)\}$ is $\mathcal{DL}(\mathbb{Q})$-consistent iff $\mu \cup \{(t_i \neq s_i)\}$ —and hence $\mu \cup \{(t_i < s_i) \vee (t_i > s_i)\}$— is $\mathcal{DL}(\mathbb{Q})$-consistent for every i.

The main algorithms encode the $\mathcal{DL}(\mathbb{Q})$-satisfiability of sets of difference inequalities into the problem of finding negative cycles into a weighted oriented graph, called *constraint graph*, by using variants of the Bellman-Ford shortest-path algorithm (see, e.g., [52]). Intuitively, a node of the graph represents univocally one variable x_i, and a labeled arc $x_1 \stackrel{a}{\longmapsto} x_2$ 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 and only if it contains a negative cycle $x_0 \stackrel{a_0}{\longmapsto} x_1 \stackrel{a_1}{\longmapsto} ... \stackrel{a_{n-1}}{\longmapsto} x_n \stackrel{a_n}{\longmapsto} x_0$ s.t. $\sum_{i=0}^n a_i < 0$, corresponding to a $\mathcal{DL}$-inconsistent subset $\eta =_{def} \bigcup_{i=0}^{n-1} \{x_{i+1} - x_i \leq a_i\} \cup \{x_0 - x_n \leq a_n\}$. Is so, then η is the conflict set which is returned by the $\mathcal{DL}(\mathbb{Q})$-solver.

Let μ be a set of difference inequalities on the variables $x_1, ..., x_N$. First, μ is augmented with all the constraints $(x_i - x_0 \leq 0)$ s.t. $i \in 1, ..., N$ and x_0 is a fresh variable, representing a "fake" source node. [21.] In a nutshell, the algorithm tries to find the tree of the paths from x_0 to all the other nodes x_i which minimize the distance d_{0i} from x_0 to x_i (hereafter denoted as "$x_0 \stackrel{d_{0i}}{\rightsquigarrow} x_i$"). This is done by maintaining the current value of d_{0i} and the antecedent node $p(x_i)$ in the current version of the tree, and iteratively updating them as follows: if for some arc $x_i \stackrel{a}{\longmapsto} x_j$ we have that $d_{0j} > a + d_{0i}$, then x_i becomes the new antecedent of x_j

19. Also called *difference logic* or *separation logic*. Notice the overlapping with the notion of "separation logic" defined in [149], where this term indicates an extension of Hoare logic for reasoning about programs that use shared mutable data structures.

20. Notice that also in $\mathcal{DL}(\mathbb{Q})$ we can assume w.l.o.g. that all constant symbols a occurring in the formula are in $\mathbb{Z}$ because, if this is not so, then we can rewrite the whole formula into an equivalently-satisfiable one by multiplying all constant symbols occurring in the formula by their greatest common denominator.

21. The fake source node x_0 is added so that to guarantee that all nodes are reachable from the source node.

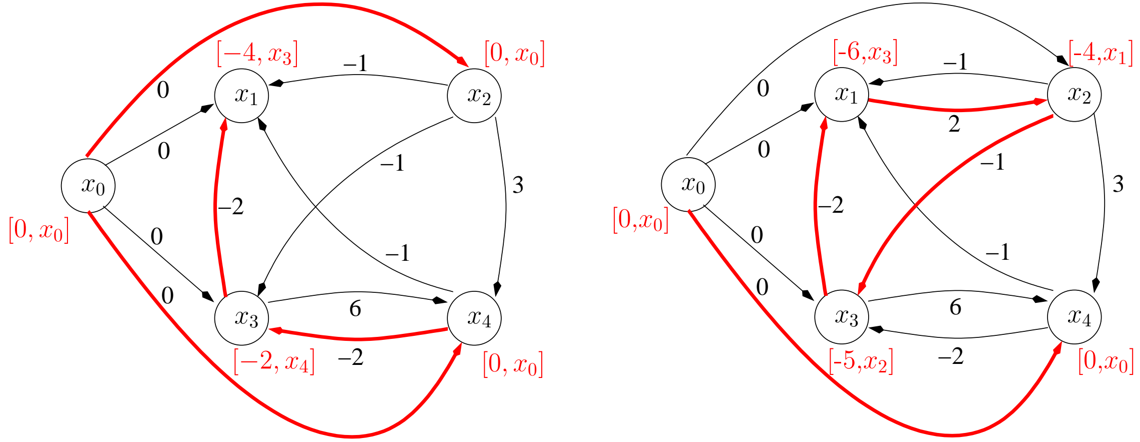

Figure 4. Constraint graphs for the problems of Example 4.6. For each x_i, values in square brackets represent $[d_{0i}, p(x_i)]$, and thick arcs incoming into x_i represents antecedence arcs $p(x_i) \xmapsto{a} x_i$.

and $d_{0j} = a + d_{0i}$ because $x_0 \overset{d_{0i}}{\rightsquigarrow} x_i \xmapsto{a} x_j$ is a shorter path fro x_0 to x_j than the previous shortest one. This process, called *relaxation* of the arc $x_i \xmapsto{a} x_j$, is repeated according to some strategy until some sufficient termination condition is met. A very basic strategy is to repeat N times the process of relaxing all arcs in the graph; hence the graph has a negative cycle if and only if, for some arc $x_i \xmapsto{a} x_j$, $d_{0j} > a + d_{0i}$ (see e.g. [53]). If this is the case, then a negative cycle can be found starting from x_i and following backward the antecedents, until a loop is found. Otherwise, $\mathcal{I} =_{def} \{d_{0i} \mid i \in 1, ..., N\}$ is a model for the set of constraints. (Much more efficient strategies are presented in [52].)

Example 4.6. *Consider the set of difference inequalities*

$$\begin{aligned}\mu = \{ \quad & (x_4 - x_2 \leq 3), (x_4 - x_3 \leq 6), (x_1 - x_2 \leq -1), (x_1 - x_3 \leq -2), (x_1 - x_4 \leq -1), \\ & (x_3 - x_2 \leq -1), (x_3 - x_4 \leq -2) \quad \}.\end{aligned}$$

After adding the fake source node x_0 and the arcs $x_0 \xmapsto{0} x_i$, the corresponding constraint graph is represented in Figure 4 (left). After running Bellman-Ford algorithm with the basic strategy described above, no arc $x_i \xmapsto{a} x_j$ is s.t. $d_{0j} > a+d_{0i}$, so that μ is $\mathcal{DL}(\mathbb{Q})$-consistent, and $\mathcal{I} =_{def} \{x_1 = -4, x_2 = 0, x_3 = -2, x_4 = 0\}$ is a model for μ.

Consider now the set

$$\begin{aligned}\mu' = \{ \quad & (x_4 - x_2 \leq 3), (x_4 - x_3 \leq 6), (x_1 - x_2 \leq -1), (x_1 - x_3 \leq -2), (x_1 - x_4 \leq -1), \\ & (x_2 - x_1 \leq 2), (x_3 - x_2 \leq -1), (x_3 - x_4 \leq -2) \quad \}\end{aligned}$$

(that is, $\mu' = \mu \cup \{(x_2 - x_1 \leq 2)\}$) and the corresponding constraint graph represented in Figure 4 (right), which contains the negative cycle $x_1 \xmapsto{2} x_2 \xmapsto{-1} x_3 \xmapsto{-2} x_1$. After running Bellman-Ford algorithm with the basic strategy described above (each loop analyzing the edges in the order they are listed in μ'), the two arcs $x_3 \xmapsto{-2} x_1$ and $x_4 \xmapsto{3} x_2$ are such that $d_{01} > -2 + d_{03}$ and $d_{04} > 3 + d_{02}$ respectively. Thus, starting from either x_3 or x_2 and following backwards the antecedents, the algorithm tracks the negative cycle, corresponding to the conflict set $\{(x_3 - x_2 \leq -1), (x_2 - x_1 \leq 2), (x_1 - x_3 \leq -2)\}$.

Efficient incremental algorithms for $\mathcal{DL}(\mathbb{Q})$-solvers have been conceived, which can efficiently perform minimal conflict-set generation [52], $\mathcal{T}$-deduction of unassigned literals [133, 54] and of interface equalities (see, e.g., [116]). In a nutshell, an efficient and incremental algorithm able to $\mathcal{T}$-deduce unassigned literals works as follows [133]: each time a new arc $x_i \overset{a}{\longmapsto} x_j$ is added so that the resulting graph is still consistent, then two instances of (an optimized version of) Bellman-Ford algorithm are run to find the minimal paths from x_j to all other nodes and from all other nodes to x_i; then, for every path in the form $x_l \overset{d_{li}}{\rightsquigarrow} x_i \overset{a}{\longmapsto} x_j \overset{d_{jm}}{\rightsquigarrow} x_m$, it is possible to deduce every unassigned literal $(x_m - x_l \leq b)$ s.t. $b \geq d_{li} + a + d_{jm}$ and every unassigned literal $\neg(x_l - x_m \leq c)$ s.t. $-c > d_{li} + a + d_{jm}$.

$\mathcal{DL}(\mathbb{Z})$ is stably-infinite and non-convex. [22.] As with $\mathcal{DL}(\mathbb{Q})$, the $\mathcal{DL}(\mathbb{Z})$-satisfiability of sets of quantifier-free difference inequalities is decidable and polynomial; as before, adding equalities does not affect the complexity of the problem. Instead, and due to the non-convexity of $\mathcal{DL}(\mathbb{Z})$, the $\mathcal{DL}(\mathbb{Z})$-satisfiability of sets of quantifier-free difference inequalities, equalities and disequalities —and hence that of deducing (disjunctions of) interface equalities— is NP-complete [115]. Once the problem is rewritten as a set of difference inequalities, the algorithms used for $\mathcal{DL}(\mathbb{Z})$-solvers are the same as for $\mathcal{DL}(\mathbb{Q})$ [52, 133, 54]. Interesting results have also been obtained by implementing "mixed" techniques combining a lazy approach with "eager" encodings of $\mathcal{DL}(\mathbb{Z})$ into SAT (§9.3) [184, 83, 108].

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. [23.] 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 [97]).

Many $\mathcal{UTVPI}$-solvers are based on the iterative transitive closure [97]: 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. [115] proposed instead one non-incremental though asymptotically faster algorithm based on negative cycle detection on an extended constraint graph. [24.] This algorithm can efficiently perform conflict-set generation and deduction of unassigned equalities, and some form of 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.

22. Same example as in Footnote 18..
23. Same example as in Footnote 18..
24. Intuitively, the set of $\mathcal{UTVPI}$ constraints is encoded into an equi-satisfiable 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 $(x + y \leq a)$).

4.2.5 Bit vectors

The *theory of fixed-width bit vectors ($\mathcal{BV}$)* [25.] 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-formulas ("*bit blasting*"). $\mathcal{BV}$ can also be used to encode software verification problems (see e.g. [84]).

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. [104, 125, 72, 84])) apply word-level preprocessing algorithms, and then encode the result into pure SAT ("*bit blasting*"); some authors (e.g, [77, 191, 42, 141, 36]) encode bits, bit vectors and (most of) their operators into $\mathcal{LA}(\mathbb{Z})$; [43] 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 [49, 58, 128, 24] 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 field of software verification. Here we provide only a very high-level description, and point the reader to the reported bibliography (e.g., to [124]) for a more detailed description.

The *theory of arrays ($\mathcal{AR}$)* [26.] 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 [124]):

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

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

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

(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

25. We should better say the *theories* of bit vectors, because many variants of $\mathcal{BV}$ have been proposed. Like in [124], 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.

26. See Footnote 25..

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. Although many practical problems do not require extensionality, the latter is explicitly required, e.g., in some software verification problems, in order to represent assignments or comparisons between arrays (e.g., C++ vector variables).

The $\mathcal{AR}$-satisfiability of sets of quantifier-free literals is decidable and NP-complete [173]. $\mathcal{AR}$ 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 [130, 70, 173]. We refer the reader to [173] 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 [124]):

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

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

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

(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 [139]. $\mathcal{LI}$ 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 [139, 21, 32] 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, subtheories. Thus, $\mathcal{T}$-*solver* may be organized in a *layered hierarchy* of solvers of increasing solving capabilities [9, 37, 158, 55, 43]. 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 L_n considers only an abstraction of the problem, that is, a subset μ_n of the input set of constraints μ s.t. $\mu_0 \subseteq \mu_1 \subseteq \ldots \subseteq \mu_{N-1} = \mu$ which is analyzed for consistency in a subtheory $\mathcal{T}_n$ of signature Σ_n s.t $\Sigma_0 \subseteq \Sigma_1 \subseteq \ldots \subseteq \Sigma_{N-1} = \Sigma$ and $\mathcal{T}_0 \subset \mathcal{T}_1 \subset \ldots \subset \mathcal{T}_{N-1} = \mathcal{T}$. Since L_n refines L_{n-1}, if the problem does not admit a solution at level L_n (i.e., $\mu_n \models_{\mathcal{T}_n} \bot$) then it does not admit a solution at all (i.e., $\mu \models_{\mathcal{T}} \bot$), and the whole $\mathcal{T}$-*solver* can return Unsat. 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.

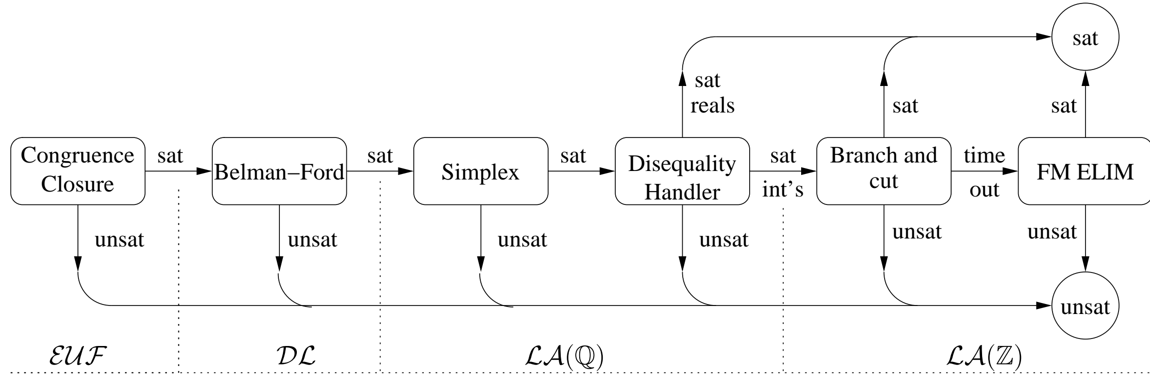

Figure 5. Example of a layered $\mathcal{T}$*-solver*: the $\mathcal{LA}(\mathbb{Z})$-solver in MathSAT [38].

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 twice". The schema can be further enhanced by allowing each layer L_n infer novel equalities and inequalities and to pass them down to the next layer L_{i+n}, so that to better drive its search [158, 159, 55].

For example, Figure 5 describes the control flow of the $\mathcal{LA}(\mathbb{Z})$-solver in MathSAT [38]. 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 [132]. 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 rationals 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. [27.] 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 [143, 137], based on Fourier-Motzkin elimination, is called on the current assignment.

27. Notice that we can assume w.l.o.g. that $\Sigma_{\mathcal{LA}(\mathbb{Q})} \subseteq \Sigma_{\mathcal{LA}(\mathbb{Z})}$ and $\Sigma_{\mathcal{DL}(\mathbb{Q})} \subseteq \Sigma_{\mathcal{DL}(\mathbb{Z})}$ because we can also assume w.l.o.g. that with $\mathcal{LA}(\mathbb{Q})$ and $\mathcal{DL}(\mathbb{Q})$ all formulas contain only integer coefficients, as explained in Footnotes 17. and 20..

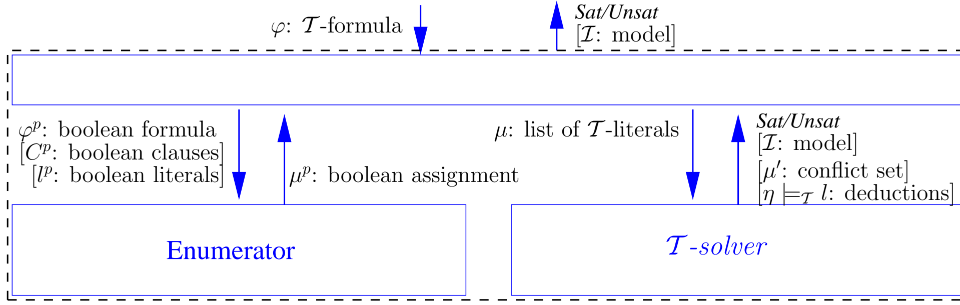

Figure 6. Basic architectural schema of a lazy $SMT(\mathcal{T})$ procedure.

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

Different representations and variants of SMT procedures integrating DPLL and $\mathcal{T}$-*solvers* have been presented. Following [79], we partition all such procedures into two main categories: the *offline* procedures, in which DPLL is used as a SAT solver which is re-invoked from scratch each time an assignment is found $\mathcal{T}$-unsatisfiable (e.g., [19, 67]), and the *on-line* procedures, in which DPLL is modified to be used directly as an enumerator (e.g., [92, 185, 5, 9, 79, 85, 40]). We also recall from [135, 133] the Abstract-DPLL Modulo Theories 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 6. $\mathcal{T}$-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. [28.]

φ^p is given as 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

28. 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 T-DPLL (T-formula φ) {
2.          φ^p = T2B(φ);
3.          while (DPLL(φ^p, μ^p) == Sat) {
4.              if (T-solver(B2T(μ^p)) == Sat)
5.                  return Sat;
6.              φ^p = φ^p ∧ ¬μ^p;
7.          };
8.          return Unsat;
9.      };
```

Figure 7. A simplified offline integration schema for lazy $SMT(\mathcal{T})$ procedures.

until 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 [19] and by [67]. [29.] In its naivest form, the idea works as described in Figure 7.

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, then φ is $\mathcal{T}$-consistent. If not, $\neg\mu^p$ is added as a clause to φ^p [30.], 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.

---

29. The offline approach is also called *lemmas on demand* approach in [67].
30. $\neg\mu^p$ is called *blocking clause*, because it blocks the future generation of every assignment containing μ^p, or *cube*, borrowing a term used in the domain of digital circuit synthesys (see e.g. [61]), because it represents a (hyper)cube of counter-assignments.

```
1.     SatValue T-DPLL (T-formula φ, T-assignment & μ) {
2.        if (T-preprocess(φ,μ) == Conflict);
3.           return Unsat;
4.        φ^p = T2B(φ); μ^p = T2B(μ);
5.        while (1) {
6.           T-decide_next_branch(φ^p,μ^p);
7.           while (1) {
8.              status = T-deduce(φ^p,μ^p);
9.              if (status == Sat) {
10.                μ = B2T(μ^p);
11.                return Sat; }
12.             else if (status == Conflict) {
13.                blevel = T-analyze_conflict(φ^p,μ^p);
14.                if (blevel == 0)
15.                   return Unsat;
16.                else T-backtrack(blevel,φ^p,μ^p);
17.             }
18.             else break;
19.    } } }
```

Figure 8. 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., [92, 185, 5, 9, 79, 85, 40]). 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 8 represents 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. [31.]

$\mathcal{T}$-`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*

31. 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 which derive 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 which can be inferred *in* $\mathcal{T}$ (i.e., s.t. $\mathcal{B}2\mathcal{T}(\mu^p) \models_{\mathcal{T}} \mathcal{B}2\mathcal{T}(l)$). (This enhancement, called $\mathcal{T}$*-propagation*, will be discussed with more details in §6.4.)

$\mathcal{T}$-`analyze_conflict` is an extension 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.

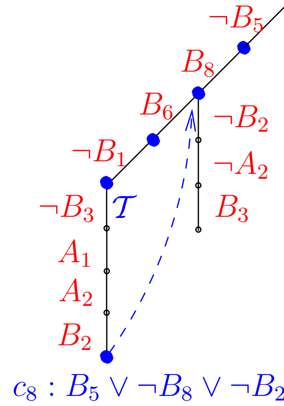

Figure 9. Boolean search (sub)tree in the scenario of Example 5.2. (A diagonal line, a vertical line and a vertical line tagged with "$\mathcal{T}$" denote literal selection, unit propagation and $\mathcal{T}$-propagation respectively; a bullet "•" denotes a call to $\mathcal{T}$-*solver*.)

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 repeat here for convenience):*

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

Consider the Boolean search (sub)tree in Figure 9. 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*

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

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

Figure 10. The Abstract-DPLL Modulo Theories logical framework from [136]. 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

As hinted in §3.2, [176, 135, 133, 136] proposed an abstract rule-based formulation of DPLL-based lazy SMT systems, the *Abstract* DPLL *Modulo Theories*. This allows for expressing and reasoning about most variants of these procedures in a formal way. In particular, [133] 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 10 [135] 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. [32.] The only rule deserving some more explanation is $\mathcal{T}$-Backjump: if a branch $\mu \cup \{l\} \cup \mu'$ falsifies propositionally one clause C (the conflicting clause), and a conflict clause $C' \vee l'$ [33.] can be computed from C s.t. $(C' \vee l')$ is entailed by $\varphi \wedge C$ in $\mathcal{T}$, $\neg C' \subseteq \mu$, $l' \notin \mu$, and l' or $\neg l'$ occur in φ or in $\mu \cup \{l\} \cup \mu'$, then it is possible to backjump up to μ, and hence unit-propagate l' on the conflict clause $(C' \vee l')$.

32. As already remarked in §3.2, the formalization of the rules in [135, 133, 136] changes slightly from paper to paper. Here we report the most-recent one from [136]. We have adapted the notation to that used in this paper.
33. Also called the *backjump clause* in [136].

Example 5.3. *Consider the formula and the scenario of Example 5.2. The execution can be represented in Abstract-*DPLL *Modulo Theories as follows:*

$\langle \quad\quad |\varphi\rangle$

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 $\neg A_2,\ B_1 \ \Rightarrow$

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

...

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. [135, 136] provide a group of results about termination, correctness and completeness of various configurations. We refer the reader to [135, 133, 136] for further details. A very recent improvement of the Abstract-DPLL Modulo Theories framework has been presented in [112].

Strictly related to the Abstract-DPLL Modulo Theories framework is the notion of DPLL($\mathcal{T}$) and DPLL(X) [85]. 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. [85] 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 [85] 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 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 a $\mathcal{T}$-equivalent or $\mathcal{T}$-equisatisfiable 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 [5],* enhanced early pruning *in [9],* theory-propagation *in [135, 133],* theory-driven deduction or $\mathcal{T}$-deduction *in [37].) 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 [85].) Finally, the description of some technique may differ significantly from that given by some authors. (E.g., [135, 133] 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 [92], 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. $(2x_1 - 6x_2 \leq 4)$ 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 $\{(2x_1 - 6x_2 \leq 4), (3x_2 + 2 < x_1), ...\}$).

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*: $(x_1 < x_2), (x_1 \geq x_2) \Longrightarrow \neg(x_1 \geq x_2), (x_1 \geq x_2)$.
- *Exploit associativity*: $(x_1 + (x_2 + x_3) = 1), ((x_1 + x_2) + x_3) = 1) \Longrightarrow (x_1 + x_2 + x_3 = 1)$.
- *Sort*: $(x_1 + x_2 - x_3 \leq 1), (x_2 + x_1 - 1 \leq x_3) \Longrightarrow (x_1 + x_2 - x_3 \leq 1))$.
- *Exploiting specific properties of* $\mathcal{T}$: $(x_1 \leq 3), (x_1 < 4) \Longrightarrow (x_1 \leq 3)$ if $\mathcal{T}$ is $\mathcal{LA}(\mathbb{Z})$.

The applicability and effectiveness of these mappings depends on the theory addressed. Notice that here normalization is not used to make atom representations simpler to handle (which can be done inside the $\mathcal{T}$*-solvers*); rather, its goal is to "recognize" as many as possible equivalent literals, so that Boolean abstraction can map them into the same Boolean literal.

Although rather straightforward, normalizing atoms is an essential step which may drastically reduce the global amount of search [92]. 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 [5] for a lazy SMT procedure for $\mathcal{DL}$. Similar such techniques were generalized and used in [12, 38, 188].

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 literal 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 [5, 12, 38, 188].

Intuitively, one can think of 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 [171, 156] (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*,[34.] was introduced by [92] in procedures for modal logics; [185, 9, 19, 85] 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 8, 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:*

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

34. Also called *intermediate assignment checking* in [92] and *eager notification* in [19].

(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.*

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 counter-balanced 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 reduced.

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 [92] or contain new variables [9]. Another way is to call $\mathcal{T}$-*solver* every k branching steps, k being an user-defined integer parameter [6].

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* [38]. This is possible because intermediate checks are not necessary for 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; [35]
- 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 [31]), 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 from reducing the overhead and the drawbacks of reducing the pruning effect.

35. This situation is 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., [12]).

6.3.3 Eager early pruning

Some DPLL-based SMT procedures for various theories (e.g., [185, 172, 85, 133]) 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., [172, 85]) $\mathcal{T}$-*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 [36.] was introduced in its simplest form (plunging, see §4.1.4) by [5] for $\mathcal{DL}$; [9] proposed an improved technique for $\mathcal{LA}$; however, $\mathcal{T}$-propagation showed its full potential in [176, 85, 133], 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. *Consider Figure 11 and 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*

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

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 (possibly expensive)* $\mathcal{T}$*-propagation step.*

36. Also called *forward reasoning* in [5], *enhanced early pruning* in [9], *theory-propagation* in [135, 133] *theory-driven deduction* or $\mathcal{T}$*-deduction* in [37, 38].

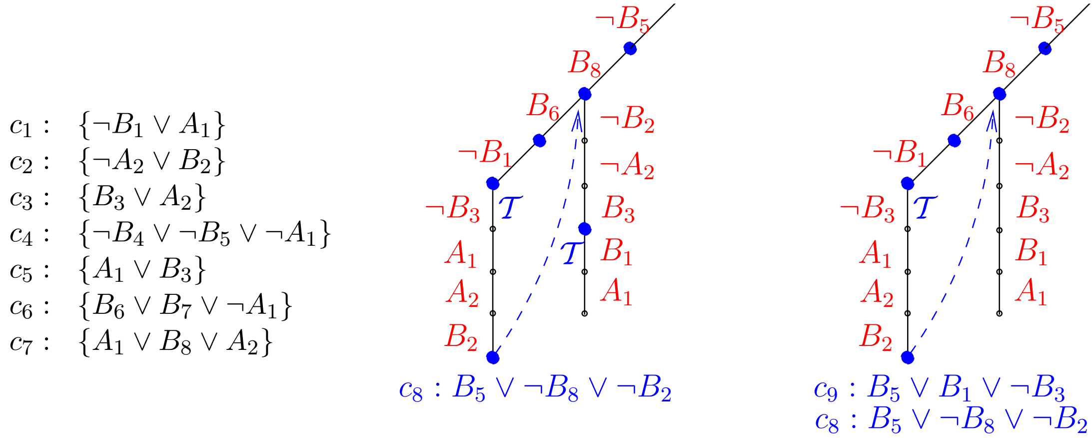

Figure 11. Left: Boolean search tree in the scenario of Example 6.3. Right: same situation, with learning of deduction clause c_9. A bullet "•" denotes a call to $\mathcal{T}$-*solver*.

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 [9, 37], or, more eagerly, every time a new $\mathcal{T}$-atom is added to the assignment (including those added by unit-propagation) [5, 176, 27, 85]. 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 [100] for description logics; [185, 67, 172, 9, 85, 37] adopted variants or improvements of the same idea for many other theories.

$\mathcal{T}$-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 [100, 185] 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 [85, 37] inherit the backjumping mechanism of current DPLL tools described in §3.1: $\mathcal{T}$-DPLL learns the conflict clause

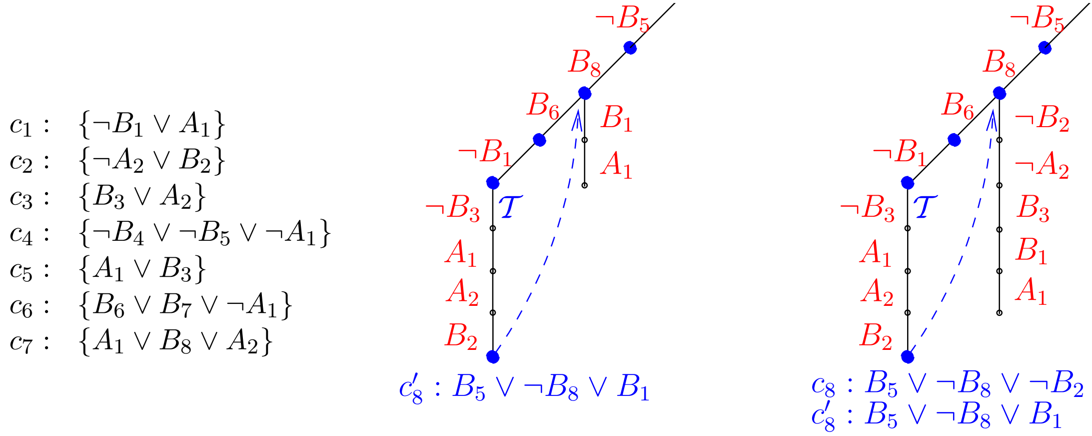

Figure 12. Left: Boolean search tree in the scenario of Example 6.4. Right: same situation, learning both c_8 and c'_8, as in Example 6.5. A bullet "•" denotes a call to $\mathcal{T}$-*solver*.

$\neg\eta^p$ 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. $\eta \wedge \varphi \models_{\mathcal{T}} \bot$) [135]. 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 12, 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 [185] for linear arithmetic; [67, 172, 9, 79] 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 12. 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 [186, 6, 68] 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 [28]. 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. [79] 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. [39, 40] used a similar idea to improve the efficiency of Delayed Theory Combination (see §8.3); [20] used the same idea in the context of the splitting-on-demand approach (see §6.7); [183] 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 [20]. [37.] 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. *[20]. 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.

37. A preliminary form of this technique was briefly described in §3.5.1 of [23] and is implemented in CVC and CVCLite [56, 57].

6.8 Clustering

This technique was proposed in [37] for of $\mathcal{EUF}$ and $\mathcal{LA}$, and was implicit in DPLL-based procedures for modal and description logics (see, e.g., [93]).

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 [6].

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*,[38] was implicitly proposed by [185] and then generalized by [90, 9, 38].

38. Also called *triggering* in [9].

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 2.3 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 [9].

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

This technique has been proposed by [38, 55] 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 $C =_{def} (l_1 \wedge \cdots \wedge l_n) \rightarrow l$ [39.] 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 $l_1, \ldots, l_n$ must have been explicitly passed to $\mathcal{T}$*-solver* (i.e., they must not have been filtered).

39. 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 from 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 [79], 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 7) 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 8) $\mathcal{T}$-learning is just a technique for improving efficiency, so that theory-conflict clauses can be learned and discharged at will. To this extent, [79] showed in an empirical test a relevant performance superiority of the online approach wrt. the offline one. [40.]

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 [19] 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 [85, 133, 71]. 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:

- apply $\mathcal{T}$-propagation exhaustively [85, 133] (i.e., always try to deduce as many literals as possible) or more selectively [9, 183] (e.g., deduce only easy-to-deduce literals).

40. 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).

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 [133] (e.g., apply $\mathcal{T}$-propagation until possible, and then apply unit-propagation), or vice-versa [37]. The former trades more $\mathcal{T}$*-solver* work for less BCP effort, and vice versa;

- learn deduction clauses lazily [133] (e.g., only when strictly necessary to backward-traversing the implication graph) or eagerly [37] (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 [55].

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 conflict-driven 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 2.2) 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. [41.])

In §3.1 we remarked that, due to the two-watched-literal scheme [129], 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.

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

41. 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. 2.2, and hence $\varphi \wedge C$ is $\mathcal{T}$-satisfiable. Thus, there is no need to check also that $\mu \models_p C$.

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 effects of this problem are drastically reduced, because the procedure backtracks as soon as a partial assignment is generated which is unsatisfiable in $\mathcal{T}$; if weakened only early pruning (§6.3.2) is implemented, however, 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) [39], as it will be made clear in §8.3.

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

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

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. E.g., one can introduce a "watch" for one not-yet-satisfied original clause. [42.] Moreover, one can obtain it as a byproduct of the techniques for avoiding ghost literals described in §7.2.2. Although these devices 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 may cause some overhead but it causes no extra Boolean search, because it does not interfere with the detection of (un)satisfiability and with the construction of the implication graph.

42. That is, one maintains a pointer to one original clause which is not satisfied by the current assignment. When it becomes satisfied, a new such clause is looked for: if none is found, then the formula is satisfied.

In SMT, instead, the presence of ghost $\mathcal{T}$-literals in μ causes useless extra work to the $\mathcal{T}$*-solver* and may affect the $\mathcal{T}$-satisfiability of μ, forcing unnecessary backtracks and causing unnecessary Boolean search and hence useless calls to the $\mathcal{T}$*-solver*.

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:*

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

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 unnecessary EP 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* $\mathcal{B}2\mathcal{T}(\{B_1, B_3\})$*. This saves one call to* $\mathcal{T}$*-solver and one T-backjumping and* $\mathcal{T}$*-learning step, and makes the only one call to* $\mathcal{T}$*-solver easier to solve.*

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. A simple way of doing that is to associate a priori to every literal l a list of the original clauses $C_1, ..., C_k$ in which l occurs. Every time a new decision literal l is assigned, then if all $C_1, ..., C_k$ are already satisfied, then l is ghost. [43.] 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 the $\mathcal{T}$*-solver* consequently.

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

As stated in §6.5, unlike older forms of $\mathcal{T}$-backjumping [100, 185], most modern implementations adopt or inherit the backjumping mechanism of modern DPLL tools, so that $\mathcal{T}$-DPLL learns the conflict clause C^p and backtracks to the highest point in the stack where one $l^p \in \eta^p$ is not assigned, and unit-propagates $\neg l^p$ on C^p. Thus, the backtrack mechanism allows for jumping high in the Boolean search tree, typically allowing for a very relevant reduction in the number of branches.

However, as noticed in [88, 142] for CSP and plain SAT, this form of "far backtracking" may have some drawbacks. In fact, it may often happen that one conflicting branch contains one (or more) set of literals $l_1, ..., l_n$ satisfying a subset of clauses $C_1, ..., C_k$ which do not interfere with those causing the conflict, and that $l_1, ..., l_n$ are skipped by the jump in the backtracking step. If so, after $\mathcal{T}$-backjumping and unit-propagating on the conflict clause, it is possible that $\mathcal{T}$-DPLL redoes all or part of the same decisions, unit-propagations, early-pruning calls and $\mathcal{T}$-propagations regarding $l_1, ..., l_n$ (e.g., reaching the same status he would have reached with a lower jump like, e.g., with the "old" $\mathcal{T}$-backjumping approach

43. Notice that one can restrict this test to $\mathcal{T}$-literals only, which typically have very few occurrences each. Notice also that one can extend the test also to literals which have been $\mathcal{T}$-deduced or unit-propagated on non-original clauses (there is no need to check l if l has been unit-propagated on one original clause, because it is obviously non-ghost). See also the related issue of $\mathcal{T}$-deduced-literal filtering of §6.11.

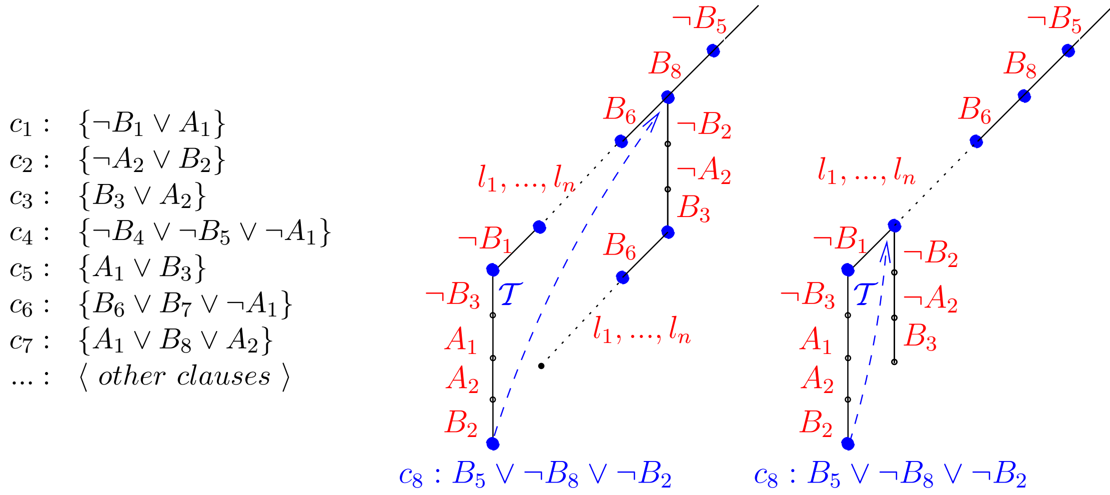

Figure 13. Boolean search trees in the scenarios of Example 7.3. Left: modern $\mathcal{T}$-backjumping. Right: comparison with the "old-style" $\mathcal{T}$-backjumping. A bullet "•" denotes a call to $\mathcal{T}$-*solver*.

of [100, 185]). As a consequence, the effort of finding a solution to (a subpart of) the "subproblem" $C_1, ..., C_k$ may be unnecessarily repeated several times in different branches.

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 interfere with the other assignments, but may independently require an amount of early-pruning calls to $\mathcal{T}$-solver and $\mathcal{T}$-propagations.)*

With a modern $\mathcal{T}$-backjumping strategy $\mathcal{T}$-DPLL jumps up above B_6 and hence unit-propagates $\neg B_2$, $\neg A_2$ and B_3 on c_8, c_2 and c_3. (Figure 13 left.) By construction, $\neg B_2$, $\neg A_2$ 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 find from scratch the assignments $l_1, ..., l_n$ of the previous branch.

As a comparison, notice that, if $\mathcal{T}$-DPLL adopted an old-style $\mathcal{T}$-backjumping strategy, then 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, ending up in the same status as in the previous case. (Figure 13 right.)

In plain SAT (and CSP), this fact can cause some overhead due to useless branching and unit-propagation steps (see [88, 142]). In SMT the effects of this problem may be even worse, because the repeated reasoning on $l_1, ..., l_n$ involves also useless early-pruning calls to the $\mathcal{T}$-*solvers* and $\mathcal{T}$-propagation steps, which may be much more expensive than plain Boolean inference steps.

There seems to be no panacea to this problem so far. As typically we do not want to give up the benefits of modern $\mathcal{T}$-backjumping in terms of number of branches explored, some other solution should be explored. One partial solution is that of adopting a literal-selection

heuristic which promotes the "locality" of search (e.g., those which select first the literals which have had an actual role in the building of the last conflict) so that to avoid analyzing $l_1, ..., l_n$ until (and unless) all relevant atoms have been explored. [44] As possible research directions, one might want to explore the possibility of adapting to lazy SMT techniques like Dynamic Backtracking [88] or the Lightweight Component Caching Scheme of [142].

7.2.4 Implementing $\mathcal{T}$-propagation

As remarked in [85], 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 [85], the set of antecedents η of the deduction are computed on demand only in these situations. In [37], 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 [12].

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 research direction is to adopt look-ahead SAT techniques (see §3) and to perform explicitly all propagations on the candidate literals in turn [121, 122]. 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 [12].

44. E.g., MathSAT adopts one such heuristic from the MiniSAT solver.

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 $\mathcal{T}_1$,...,$\mathcal{T}_n$. For instance, an atom of the form $f(x + 4y) = g(2x - y)$, that combines uninterpreted function symbols (from $\mathcal{EUF}$) with arithmetic functions (from $\mathcal{LA}(\mathbb{Z})$), could be used to naturally model in a uniform setting the abstraction of some functional blocks in an arithmetic circuit (see e.g. [42, 36]). 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 theories.

The work on combining decision procedures (i.e., $\mathcal{T}$*-solvers* in our terminology) for distinct theories was pioneered by Nelson and Oppen [130, 138] and Shostak [162]. [45.] In particular, Nelson and Oppen established the theoretical foundations onto which most current combined procedures are still based on (hereafter *Nelson-Oppen (N.O.) formal framework*, §8.1). They also proposed a general-purpose procedure for integrating $\mathcal{T}_i$*-solvers* into one combined $\mathcal{T}$*-solver* (hereafter *Nelson-Oppen (N.O.) procedure*, §8.2). This procedure is then extended to SMT by integrating the combined $\mathcal{T}$*-solver* with DPLL according to the lazy paradigm of §5. More recently Bozzano et al. [39, 40] proposed *Delayed Theory Combination, DTC* (§8.3), a novel combination procedure which builds a combined SMT solver directly by exploiting DPLL also for theory combination. For the very specific case in which $\mathcal{EUF}$ is combined with one theory $\mathcal{T}$, a viable alternative is that of applying *Ackermann's expansion* [1] (§8.4).

Variants and evolutions of N.O. procedure were implemented in the SVC/CVC/CVCLite [17], ICS [64], Simplify [70], Verifun [79], Zapato [16]) lazy SMT tools. Variants or evolutions of DTC procedure are currently implemented in the MathSAT [40], Yices [73], and Z3 [62] lazy SMT tools; CVC3 [22] combines ideas from Dtc and splitting-on-demand (§6.7). Ackermann's expansion for $\mathcal{EUF} \cup \mathcal{T}$ is implemented in BarceLogic [133] and in MathSAT [40].

In what follows we assume all the notions described in §2.1.1, in particular those of stably-infinite and convex theories, of alien subterms, of pure formulas and of interface variables and equalities. As in §2.1.1, for simplicity we refer to combinations of two theories $\mathcal{T}_1 \cup \mathcal{T}_2$ only; however, all the discourse can be easily generalized to combination of many theories $\mathcal{T}_1 \cup ... \cup \mathcal{T}_n$.

8.1 The Nelson-Oppen formal framework

Consider two decidable stably-infinite theories with equality $\mathcal{T}_1$ and $\mathcal{T}_2$ and disjoint signatures Σ_1 and Σ_2 (often called *Nelson-Oppen theories*) and consider a pure conjunction of $\mathcal{T}_1 \cup \mathcal{T}_2$-literals $\mu =_{def} \mu_{\mathcal{T}_1} \wedge \mu_{\mathcal{T}_2}$ s.t. $\mu_{\mathcal{T}_1}$ is i-pure for each i. Nelson&Oppen's key observation is that μ is $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiable if and only if it is possible to find two satisfying interpretations $\mathcal{I}_1$ and $\mathcal{I}_2$ s.t. $\mathcal{I}_1 \models_{\mathcal{T}_1} \mu_{\mathcal{T}_1}$ and $\mathcal{I}_2 \models_{\mathcal{T}_2} \mu_{\mathcal{T}_2}$ which agree on all equalities on the shared variables. This is stated in the following theorem. [46.]

45. 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 [150, 25, 70].

46. Since [130] many different formulations of N.O. correctness and completeness results have been presented (e.g. [130, 138, 177, 178]). Here we adopt a notational variant of that in [177].

Theorem 8.1. *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two stably-infinite theories with equality and disjoint signatures; let $\mu =_{def} \mu_{\mathcal{T}_1} \wedge \mu_{\mathcal{T}_2}$ be a conjunction of $\mathcal{T}_1 \cup \mathcal{T}_2$-literals s.t. $\mu_{\mathcal{T}_i}$ is i-pure for each i. Then $\mu_{\mathcal{T}_1} \wedge \mu_{\mathcal{T}_2}$ is $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiable if and only if there exists some equivalence relation $e(.,.)$ over $Vars(\mu_{\mathcal{T}_1}) \cap Vars(\mu_{\mathcal{T}_2})$ s.t. $\mu_{\mathcal{T}_i} \wedge \mu_e$ is $\mathcal{T}_i$-satisfiable for every i, where:*

$$\mu_e =_{def} \bigwedge_{(v_i,v_j)\ \in\ e(.,.)} (v_i = v_j)\ \wedge \bigwedge_{(v_i,v_j)\ \notin\ e(.,.)} \neg(v_i = v_j). \tag{19}$$

μ_e is called the arrangement of $e(.,.)$.

Overall, Nelson-Oppen results reduce the $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiability problem of a set of pure literals μ to that of finding (the arrangement of) an equivalence relation on the shared variables which is consistent with both pure parts of μ.

Example 8.2. *[45] Consider the following pure conjunction of $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Z})$-literals $\mu =_{def} \mu_{\mathcal{EUF}} \wedge \mu_{\mathcal{LA}(\mathbb{Z})}$ s.t.*

$$\begin{array}{ll} \mu_{\mathcal{EUF}}: & \neg(f(v_1) = f(v_2)) \wedge \neg(f(v_2) = f(v_4)) \wedge (f(v_3) = v_5) \wedge (f(v_1) = v_6) \\ \mu_{\mathcal{LA}(\mathbb{Z})}: & (v_1 \geq 0) \wedge (v_1 \leq 1) \wedge (v_5 = v_4 - 1) \wedge (v_3 = 0) \wedge (v_4 = 1) \wedge \\ & (v_2 \geq v_6) \wedge (v_2 \leq v_6 + 1). \end{array} \tag{20}$$

Here $v_1, \ldots, v_6$ are interface variables, because they occur in both $\mathcal{EUF}$ and $\mathcal{LA}(\mathbb{Q})$-pure terms. We consider the arrangement

$$\mu_e =_{def} (v_1 = v_4) \wedge (v_3 = v_5) \wedge \bigwedge_{(v_i = v_j) \notin \{(v_1 = v_4), (v_3 = v_5)\}} \neg(v_i = v_j).$$

It is easy to see that $\mu_{\mathcal{EUF}} \wedge \mu_e$ is $\mathcal{EUF}$-consistent (because no equality or congruence constraint is violated) and that $\mu_{\mathcal{LA}(\mathbb{Z})} \wedge \mu_e$ is $\mathcal{LA}(\mathbb{Z})$-consistent (e.g., $v_3 = v_5 = 0$, $v_1 = v_4 = 1$, $v_2 = 4$, $v_6 = 3$ is a $\mathcal{LA}(\mathbb{Z})$-model). Thus, by Theorem 8.1, μ is $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Z})$-consistent.

The condition of having only pure conjunctions as input allows to partition the problem into two independent $\mathcal{T}_i$-satisfiability problems $\mu_{\mathcal{T}_i} \wedge \mu_e$. This condition is easy to address, because every non-pure $\mathcal{T}_1 \cup \mathcal{T}_2$-formula φ can be converted into a $\mathcal{T}_1 \cup \mathcal{T}_2$-equisatisfiable and pure one by recursively replacing each alien subterm t by a new variable v_t and conjoining the equality $v_t = t$ with φ. E.g., if $\mathcal{T}_1$ is $\mathcal{EUF}$ and $\mathcal{T}_2$ is $\mathcal{LA}(\mathbb{Q})$, then:

$$(f(x + 4y) = g(2x - y)) \implies (f(v_{x+4y}) = g(v_{2x-y})) \wedge (v_{x+4y} = x + 4y) \wedge (v_{2x-y} = 2x - y).$$

This process is called *purification*, and the size of the resulting formula is linear in that of φ.

The condition of having stably-infinite theories is sufficient to guarantee enough values in the domain to allow the satisfiability of every possible set of disequalities one may encounter. [47.] This fact is illustrated by the following example (adapted from [177]).

47. More formally, if $\mathcal{I}_1 \models (\mathcal{T}_1 \wedge \mu_{\mathcal{T}_1} \wedge \mu_e)$ and $\mathcal{I}_2 \models (\mathcal{T}_2 \wedge \mu_{\mathcal{T}_2} \wedge \mu_e)$ and both $\mathcal{T}_i$'s are stably infinite, then there exist other two models $\mathcal{I}_1'$ and $\mathcal{I}_2'$ of infinite cardinality s.t. $\mathcal{I}_1' \models (\mathcal{T}_1 \wedge \mu_{\mathcal{T}_1} \wedge \mu_e)$ and $\mathcal{I}_2' \models (\mathcal{T}_2 \wedge \mu_{\mathcal{T}_2} \wedge \mu_e)$. Thus it is possible to find an isomorphism between $\mathcal{I}_1'$ and $\mathcal{I}_2'$ and hence build a common model $\mathcal{I}$ s.t. $\mathcal{I} \models (\mathcal{T}_1 \wedge \mathcal{T}_2 \wedge \mu_{\mathcal{T}_1} \wedge \mu_{\mathcal{T}_2} \wedge \mu_e)$.

Example 8.3. *Let* $\mathcal{T}_1$ *be* $\mathcal{EUF}$ *and* $\mathcal{T}_2$ *be* $\mathcal{BV}_{[1]}$ *(the theory of bit-vectors of length 1). Clearly* $\mathcal{BV}_{[1]}$*, and hence* $\mathcal{EUF} \cup \mathcal{BV}_{[1]}$*, admits only models of cardinality 2 and thus it is not stably-infinite, so that Theorem 8.1 cannot be applied to these theories.*

Consider the following pure conjunction of $\mathcal{EUF} \cup \mathcal{BV}_{[1]}$*-literals* $\mu =_{def} \mu_{\mathcal{EUF}} \wedge \mu_{\mathcal{BV}_{[1]}}$*:*

$$
\begin{array}{ll}
\mu_{\mathcal{EUF}}: & \neg(f(v_2) = f(v_3)) \;\wedge\; \neg(f(v_1) = f(v_3)) \;\wedge \\
\mu_{\mathcal{BV}_{[1]}}: & \neg(\ \textbf{\textit{not}}\ v_1 = \ \textbf{\textit{not}}\ v_2).
\end{array}
$$

Here f *is an uninterpreted function symbol in* $\mathcal{EUF}$ *and* ***not*** *represents the inverter gate in* $\mathcal{BV}_{[1]}$*.* v_1 *and* v_2 *are interface variables, and* $v_1 = v_2$ *is the only interface equality.*

On the one hand, it is easy to see that μ *is* $\mathcal{EUF} \cup \mathcal{BV}_{[1]}$*-unsatisfiable. In fact, we have that* $\mu \models_{\mathcal{EUF}\cup\mathcal{BV}_{[1]}} (\neg(v_1 = v_2) \wedge \neg(v_2 = v_3) \wedge \neg(v_2 = v_3))$*, so that* μ *can be satisfied only by models of cardinality greater or equal than 3, whilst* $\mathcal{EUF} \cup \mathcal{BV}_{[1]}$ *admits only models of cardinality 2.*

On the other hand, if we ignored the stably-infinite precondition and tried to apply Theorem 8.1, we could consider the arrangement $\mu_e =_{def} \{\neg(v_1 = v_2)\}$*, so that* $\mu_{\mathcal{EUF}} \wedge \mu_e$ *is* $\mathcal{EUF}$*-satisfiable and* $\mu_{\mathcal{BV}_{[1]}} \wedge \mu_e$ *is* $\mathcal{BV}_{[1]}$*-satisfiable, from which we could conclude that* μ *is* $\mathcal{EUF} \cup \mathcal{BV}_{[1]}$*-satisfiable.*

A significant research effort has been paid to extend N.O. framework by releasing the conditions it is based on (purity the of inputs, stably-infiniteness and signature-disjointness of the theories.) We briefly overview some of them. [48.]

First, [25] shows that the purity condition is not really necessary in practice. Intuitively, one may consider alien terms as if they were variables, and consider equalities between alien terms as interface equalities. We refer the reader to [25] for details.

Many approaches have been presented in order to release the condition of stably-infiniteness (e.g., [189, 190, 80, 179, 145, 33]). In particular, [190, 179] proposed a method which extends the N.O. framework by reasoning not only on interface equalities, but also on particular cardinality constraints; this method has been extended to many-sorted logics in [145]; the problem has been further explored theoretically in [33], and related to that of combining rewrite-based decision procedures (see §9.2). Finally, the paradigm in [179] has been recently extended in [113] so that to handle also parametric theories.

A few approaches have been proposed also to release the condition of signature-disjointness [178, 86]. A theoretical framework addressing this problem was proposed in [178], which allowed for producing semi-decision procedures. [86] proposed a general theoretical framework based on classical model theory. An even more general framework for combining decision procedures, which captures that in [86] as a subcase, has been recently presented in [87].

All these results, however, involve a level of theoretical analysis which is way out of the scope of this survey, so that we refer the reader to the cited bibliography for further details.

8.2 The Nelson-Oppen combination procedure

In [130] Nelson Oppen proposed also a general-purpose combination procedure for combining two (or more) $\mathcal{T}_i$-*solvers* into one $\mathcal{T}_1 \cup \mathcal{T}_2$-*solver*. (The $\mathcal{T}_1 \cup \mathcal{T}_2$-*solver* is then integrated with DPLL as described in §5.) The combined decision procedure $\mathcal{T}_1 \cup \mathcal{T}_2$-*solver* works

48. The list of references and approaches listed here is by no means intended to be exhaustive.

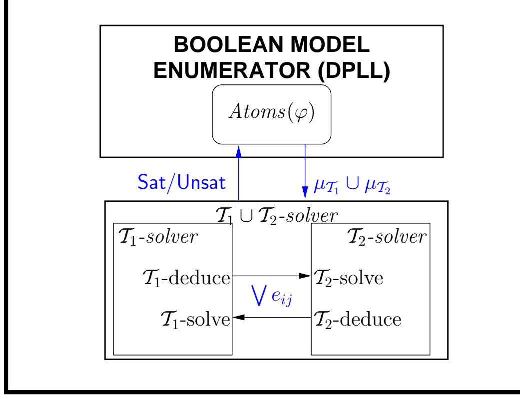

Figure 14. A basic architectural schema of $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$ via the N.O. procedure.

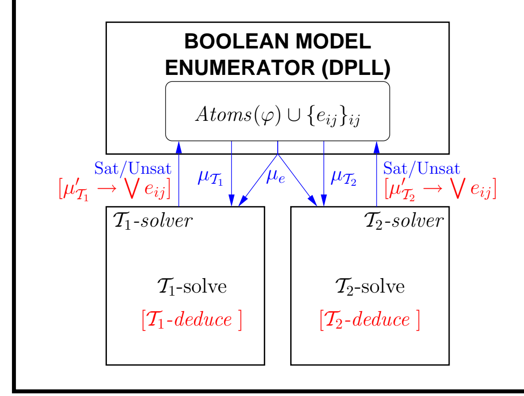

Figure 15. A basic architectural schema of $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$ via the DTC procedure.

by performing a structured interchange of interface equalities (disjunctions of interface equalities if $\mathcal{T}_i$ is non-convex) which are inferred by either $\mathcal{T}_i$*-solver* and then propagated to the other, until convergence is reached. (Here we provide only a high-level description of this procedure. The reader may refer, e.g., to [130, 161, 78, 25, 157, 70] for more details.)

A basic architectural schema of $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$ via N.O. is described in Figure 14. Both $\mathcal{T}_i$'s are N.O. theories s.t. their respective $\mathcal{T}_i$*-solvers* are e_{ij}-deduction complete (see §4.1.5). We consider first the case in which both theories are convex. The combined $\mathcal{T}_1 \cup \mathcal{T}_2$*-solver* receives from DPLL a pure set of literals μ, and partitions it into $\mu_{\mathcal{T}_1} \cup \mu_{\mathcal{T}_2}$, s.t. $\mu_{\mathcal{T}_i}$ is i-pure, and feeds each $\mu_{\mathcal{T}_i}$ to the respective $\mathcal{T}_i$*-solver*. Each $\mathcal{T}_i$*-solver*, in turn:

(i) checks the $\mathcal{T}_i$-satisfiability of $\mu_{\mathcal{T}_i}$,
(ii) deduces all the interface equalities deriving from $\mu_{\mathcal{T}_i}$,
(iii) passes them to the other $\mathcal{T}$-solver, which adds it to his own set of literals.

This process is repeated until either one $\mathcal{T}_i$*-solver* detects inconsistency ($\mu_1 \cup \mu_2$ is $\mathcal{T}_1 \cup \mathcal{T}_2$-unsatisfiable), or no more e_{ij}-deduction is possible ($\mu_1 \cup \mu_2$ is $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiable).

Example 8.4. *[45] Consider the following $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$ pure formula φ*

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

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.
Consider the search tree in Fig. 16. 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.*

$$\begin{array}{ll} \mu_{\mathcal{EUF}} = & \{(v_3 = h(v_0)), (v_4 = h(v_1)), (v_6 = f(v_2)), (v_7 = f(v_5)), \neg(v_6 = v_7)\} \\ \mu_{\mathcal{LA}(\mathbb{Q})} = & \{(v_0 \leq v_1), (v_0 \geq v_1), (v_2 = v_3 - v_4), (v_5 = 0)\}, \end{array} \tag{22}$$

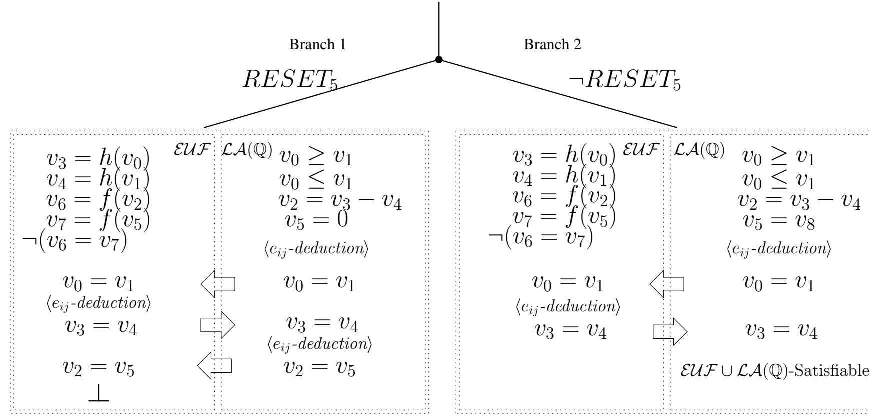

Figure 16. Search tree for the formula of Example 8.4

which propositionally satisfies φ. Now, the set of literals $\mu_{\mathcal{EUF}}$ is given to the $\mathcal{EUF}$-solver, which reports its consistency and deduces no new interface equality. Then the set $\mu_{\mathcal{LA}(\mathbb{Q})}$ 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 can deduce from it 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 procedure reports the $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$-unsatisfiability of 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.*

$$\begin{array}{ll} \mu_{\mathcal{EUF}} = & \{(v_3 = h(v_0)), (v_4 = h(v_1)), (v_6 = f(v_2)), (v_7 = f(v_5)), \neg(v_6 = v_7)\} \\ \mu'_{\mathcal{LA}(\mathbb{Q})} = & \{(v_0 \leq v_1), (v_0 \geq v_1), (v_2 = v_3 - v_4)(v_5 = v_8)\}, \end{array} \quad (23)$$

which is found $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$-satisfiable with a similar process (see Fig. 16).

In the case of non-convex theories, the N.O. procedure becomes more complicated, because the two solvers need to exchange arbitrary disjunctions of interface equalities. As each $\mathcal{T}_i$*-solver* can handle only conjunctions of literals, the disjunctions must be managed by means of case splitting and of backtrack search. Thus, the N.O. procedure must explore 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.

Example 8.5. *[45] Consider the conjunction of literals $\mu =_{def} \mu_{\mathcal{EUF}} \wedge \mu_{\mathcal{LA}(\mathbb{Z})}$ (20) of Example 8.2:*

$$\begin{array}{ll} \mu_{\mathcal{EUF}}: & \neg(f(v_1) = f(v_2)) \wedge \neg(f(v_2) = f(v_4)) \wedge (f(v_3) = v_5) \wedge (f(v_1) = v_6) \wedge \\ \mu_{\mathcal{LA}(\mathbb{Z})}: & (v_1 \geq 0) \wedge (v_1 \leq 1) \wedge (v_5 = v_4 - 1) \wedge (v_3 = 0) \wedge (v_4 = 1) \wedge \\ & (v_2 \geq v_6) \wedge (v_2 \leq v_6 + 1). \end{array} \quad (24)$$

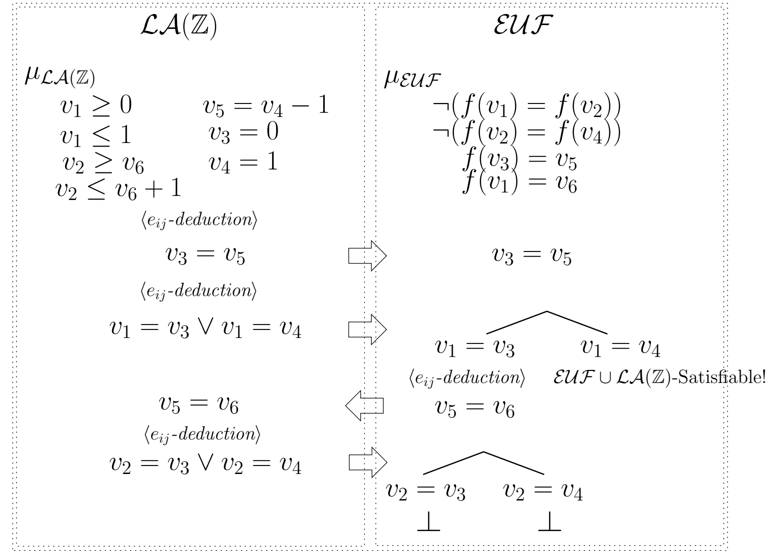

Figure 17. The N.O. search tree for the formula of Example 8.5

Here all the variables ($v_1, \ldots, v_6$) are interface ones. μ 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 μ. One possible run of the N.O. procedure is depicted in Fig. 17.* [49]

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 first $(v_3 = v_5)$ and then the disjunction $(v_1 = v_3) \vee (v_1 = v_4)$, which are both passed to th $\mathcal{EUF}$-solver. Whilst the first produces no effect, the second forces a case-splitting so that the two equalities $(v_1 = v_3)$ and $(v_1 = v_4)$ must be analyzed separately by 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 μ under the whole assignment μ.

Notice that the ability to carry out e_{ij}-deductions (see §4.1.5) 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.

49. Notice that there may be different runs depending on the order in which the e_{ij}-deductions are performed.

We also notice that, in a standard N.O.-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 8.6. *Consider the scenario of the left branch in Example 8.4 and Fig. 16. 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.

We refer the reader to [130, 161, 78, 25, 157, 70] for a more detailed description of N.O. procedure and its implementation.

8.3 The Delayed Theory Combination procedure

*Delayed Theory Combination (*Dtc*)* is a more recent general-purpose procedure for tackling the problem of theory combination directly in the context of lazy SMT [39, 40, 45]. Dtc works by performing Boolean reasoning on interface equalities, possibly combined with $\mathcal{T}$-propagation. As with N.O. procedure, Dtc is based on Theorem 8.1, and thus considers signature-disjoint stably-infinite theories with their respective $\mathcal{T}_i$*-solvers*, and pure input formulas (although most of the considerations on releasing purity and stably-infiniteness in §8.1 hold for Dtc as well). Importantly, no assumption is made about the e_{ij}-deduction capabilities of the $\mathcal{T}_i$*-solvers* (§4.1.5): for each $\mathcal{T}_i$*-solver*, every intermediate situation from complete e_{ij}-deduction to no e_{ij}-deduction capabilities is admitted.

A basic architectural schema of Dtc is described in Figure 15. In Dtc, each of the two $\mathcal{T}_i$*-solvers* interacts directly and only with the Boolean enumerator, so that there is no direct exchange of information between the $\mathcal{T}_i$*-solvers*. The Boolean enumerator is instructed to assign truth values not only to the atoms in $Atoms(\varphi)$, but also to the interface equalities e_{ij}'s. Consequently, each assignment enumerated by DPLL μ^p is partitioned into three components $\mu^p_{\mathcal{T}_1}$, $\mu^p_{\mathcal{T}_2}$ and μ^p_e, s.t. each $\mu_{\mathcal{T}_i}$ is the set of i-pure literals and μ_e is the set of interface (dis)equalities in μ, so that each $\mu_{\mathcal{T}_i} \cup \mu_e$ is passed to the respective $\mathcal{T}_i$*-solver*.

An implementation of Dtc [40, 45] is based on the online schema of Figure 8 in §5.3, exploiting early pruning (§6.3), $\mathcal{T}$-propagation (§6.4), $\mathcal{T}$-backjumping (§6.5) and $\mathcal{T}$-learning (§6.6). Each of the two $\mathcal{T}_i$*-solvers* interacts with the DPLL engine by exchanging literals via the assignment μ in a stack-based manner. The $\mathcal{T}$-DPLL algorithm of Figure 8 in §5.3 is modified to the following extents [40, 45]:

(i) $\mathcal{T}$-DPLL is instructed to assign truth values not only to the atoms in φ, but also to the interface equalities not occurring in φ. $\mathcal{B}2\mathcal{T}$ and $\mathcal{T}2\mathcal{B}$ are modified accordingly.

(ii) $\mathcal{T}$-`decide_next_branch` is modified to select also interface equalities e_{ij}'s not occurring in the formula yet [50.], after the current assignment propositionally satisfies φ.

(iii) $\mathcal{T}$-`deduce` is modified to work as follows: for each $\mathcal{T}_i$, $\mu_{\mathcal{T}_i} \cup \mu_e$, is fed to the respective $\mathcal{T}_i$-*solver*. If both return Sat, then $\mathcal{T}$-`deduce` returns Sat, otherwise it returns Conflict.

(iv) $\mathcal{T}$-`analyze_conflict` and $\mathcal{T}$-`backtrack` are modified so that to use the conflict set returned by one $\mathcal{T}_i$-*solver* for $\mathcal{T}$-backjumping and $\mathcal{T}$-learning. Importantly, such conflict sets may contain interface (dis)equalities.

(v) Early-pruning and $\mathcal{T}$-propagation are performed. If one $\mathcal{T}_i$-solver performs the e_{ij}-deduction $\mu^* \models_{\mathcal{T}_i} \bigvee_{j=1}^{k} e_j$ s.t. $\mu^* \subseteq \mu_{\mathcal{T}_i} \cup \mu_e$, each e_j being an interface equality, then the deduction clause $\mathcal{T}2\mathcal{B}(\mu^* \rightarrow \bigvee_{j=1}^{k} e_j)$ is learned.

(vi) **[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. [51.]

In order to achieve efficiency, other heuristics and strategies have been further suggested in [39, 40, 45], and more recently in [73, 62].

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

First, due to (ii), the Boolean search tree is divided into two parts: the top part, performed on the atoms currently occurring in the formula, in which a (partial) truth assignment μ propositionally satisfying φ is searched, and the bottom part, performed on the e_{ij}'s which do not yet occur in the formula, which checks the $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiability of μ by building a candidate arrangement μ_e. Thus, in every branch the reasoning on e_{ij}'s is not performed until (and unless) it is strictly necessary. [52.] E.g., if in one branch μ is such that one $\mu_{\mathcal{T}_i}$ component is $\mathcal{T}_i$-unsatisfiable, no reasoning on e_{ij}'s is performed.

To this extent, it is important 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 only the new e_{ij}'s are then selected. Importantly, if μ is a partial assignment, then it is sufficient that μ_e assigns only the e_{ij}'s which have an actual interface role in μ. [53.]

Second, thanks to (iv) and (v), the interface equalities e_{ij}'s are included in the learned clauses derived by $\mathcal{T}$-conflicts and $\mathcal{T}$-deduction. Therefore, the reasoning steps on e_{ij}'s which are performed in order to decide the $\mathcal{T}_1 \cup \mathcal{T}_2$-consistency of one branch μ (both

50. Notice that an interface equality occurs in the formula after a clause containing it is learned, see (iv).
51. This is identical to the $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiability termination condition of N.O. procedure.
52. From which the name "Delayed Theory Combination".
53. 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.

Boolean search on e_{ij}'s and e_{ij}-deduction steps) are saved in the form of clauses and thus they can be reused to check the $\mathcal{T}_1 \cup \mathcal{T}_2$-consistency of all subsequent branches. This allows from pruning search and prevents redoing the same search/deduction steps from scratch.

Third, in case of non-convex theories, the case-splits caused by the deduction of disjunctions of e_{ij}'s by the $\mathcal{T}_i$*-solvers* are handled directly by the DPLL engine. (Notice that, unlike with splitting on demand in §6.7, here we refer to the case-splits which are necessary to *handle* the deductions performed by the other $\mathcal{T}_i$*-solver*, rather to those which may be necessary to *perform* such deductions.)

Fourth, Dtc allows for using $\mathcal{T}_i$*-solvers* with partial or no e_{ij}-deduction capability, because part of or all the e_{ij}-deductions can be substituted by extra Boolean search on the e_{ij}'s performed by the DPLL engine. Thus, by adopting $\mathcal{T}$*-solvers* with different e_{ij}-deduction power, one can trade part or all the (possibly very expensive) e_{ij}-deduction effort for extra Boolean search.

[45] analyzes the enlargement of the Boolean search space in Dtc wrt. N.O. procedure, and proves the following facts.

1. Under the same working hypotheses of N.O. procedure (stably-infinite theories, incremental, backtrackable and e_{ij}-deduction-complete $\mathcal{T}_i$*-solvers*), there is a "N.O.-mimicking" strategy for Dtc s.t. no extra Boolean search on the e_{ij}'s is performed wrt. N.O procedure: in case of convex theories, no extra Boolean search on e_{ij}'s is performed; in case on non-convex theories, the only Boolean search on e_{ij}'s performed is that caused by the case-splits induced by the disjunctions of e_{ij}'s (which N.O. procedure must perform internally to the combined $\mathcal{T}$*-solver*).

2. If some $\mathcal{T}_i$*-solver* is not e_{ij}-deduction complete, then the "N.O.-mimicking" strategy for Dtc mimics the e_{ij}-deductions performed by N.O. procedure via $\mathcal{T}$-backjumping, and the cost in terms of Boolean search can be controlled in terms of the "quality" of the $\mathcal{T}$-conflict sets η returned by the $\mathcal{T}_i$*-solvers*: the more redundant $\neg e_{ij}$'s are removed from η, the more branches are pruned; if the η's contain no redundant $\neg e_{ij}$, then the Boolean search reduces to only one branch for every e_{ij}-deduction mimicked.

Result 1 states that, under the same working hypotheses, the Dtc procedure is "at least as good as the N.O. procedure" in terms of Boolean search space. Result 2 states that, if the $\mathcal{T}_i$*-solver* have partial or no e_{ij}-deduction capabilities, then the amount of extra Boolean search required can be reduced down to a negligible amount by reducing the presence of redundant negated disequalities in the $\mathcal{T}$-conflict sets returned by the $\mathcal{T}$*-solvers*.

We illustrate all these facts with the following examples. [54.] As in §8.2, we consider first the case in which both $\mathcal{T}_i$*-solvers* are e_{ij}-complete.

Example 8.7. *[45] Consider again the $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$ formula φ of Example 8.4. Figure 18 illustrates a* Dtc *execution when both $\mathcal{T}_i$-solvers are e_{ij}-deduction complete.*

54. In all the following examples Dtc adopts the "N.O.-mimicking strategy" described in [45]. Notationally, $\mu'_{\mathcal{T}_i}$, $\mu''_{\mathcal{T}_i}$, $\mu'''_{\mathcal{T}_i}$ denote generic subsets of $\mu_{\mathcal{T}_i}$ s.t. $\mathcal{T}_i \in \{\mathcal{EUF}, \mathcal{LA}(\mathbb{Q}), \mathcal{LA}(\mathbb{Z})\}$, and "$C_{ij}$" denotes either the $\mathcal{T}$-deduction clause causing the $\mathcal{T}$-propagation of $(v_i = v_j)$ or the conflicting clause causing the backjump to $(v_i = v_j)$.

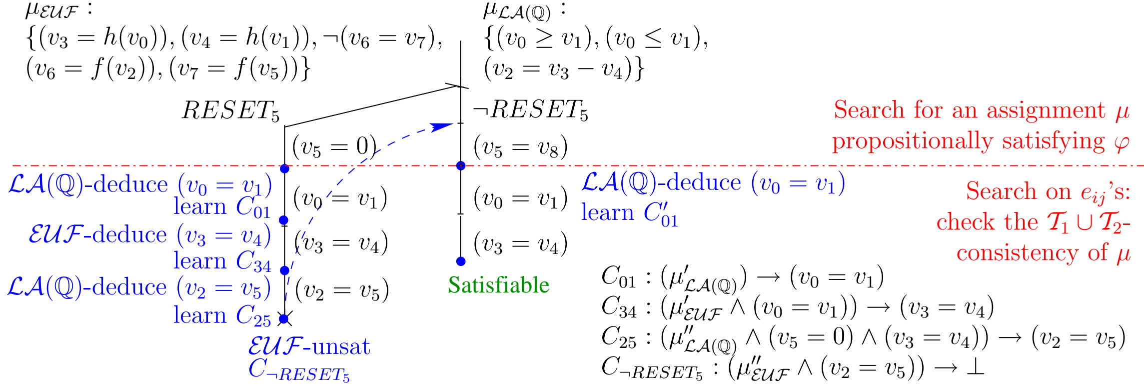

Figure 18. DTC execution of Example 8.7, with e_{ij}-deduction-complete $\mathcal{T}_i$-solvers.

On the left branch (when $RESET_5$ is selected), after the unit-propagation of $(v_5 = 0)$, the $\mathcal{LA}(\mathbb{Q})$-solver deduces $(v_0 = v_1)$, and thus by (v) the clause C_{01} is learned and $(v_0 = v_1)$ is unit-propagated. Consequently, the $\mathcal{EUF}$-solver can deduce $(v_3 = v_4)$, causing the learning of C_{34} and the unit-propagation of $(v_3 = v_4)$, which in turn causes the $\mathcal{LA}(\mathbb{Q})$-deduction of $(v_2 = v_5)$, the learning of C_{25} and the unit-propagation of $(v_2 = v_5)$.

At this point, $\mu''_{\mathcal{EUF}} \cup \{(v_2 = v_5)\}$ is found $\mathcal{EUF}$-inconsistent, so that the $\mathcal{EUF}$-solver returns (the negation of) the clause $C_{\neg RESET_5}$, which is resolved backward with the clauses C_{25}, C_{34}, C_{01}, and $(RESET_5 \rightarrow (v_5 = 0))$ forcing DTC *to backjump up to the last branching point and to unit-propagate $\neg RESET_5$. Hence $(v_5 = v_8)$ is unit-propagated on the clause $\neg RESET_5 \rightarrow (v_5 = v_8)$, which produces another assignment propositionally satisfying φ.*

Then, $(v_0 = v_1)$ and hence $(v_3 = v_4)$ are unit-propagated on C_{01} and C_{34} respectively, with no need to call the $\mathcal{T}$-solvers. [55] *At this point, since neither $\mathcal{T}_i$-solver can deduce any new e_{ij}, by (vi)* DTC *concludes that φ is $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$-satisfiable.*

Notice that the left branch of the DTC search tree of Figure 18 mimics directly that of the N.O. execution of Figure 16. No extra Boolean search is performed by DTC. In the right branch of the DTC search tree, instead, all values are assigned directly by unit-propagation. Thus, DTC "remembers" in form of clauses the e_{ij}-deductions performed in the first branch and reuses them in the subsequent branch so that to avoid redoing them from scratch.

Example 8.8. *[45] Consider again the conjunction of literals $\mu =_{def} \mu_{\mathcal{EUF}} \wedge \mu_{\mathcal{LA}(\mathbb{Z})}$ of Examples 8.2 and 8.5. We assume here that both the $\mathcal{EUF}$- and $\mathcal{LA}(\mathbb{Z})$-solver's have no e_{ij}-deduction capabilities, and that they always return conflict sets which do not contain redundant negated interface equalities. One possible session of* DTC *is depicted in Fig. 19.*

Initially, both $\mu_{\mathcal{LA}(\mathbb{Z})}$ and $\mu_{\mathcal{EUF}}$ are found consistent in each of the theories by the respective $\mathcal{T}_i$-solvers. Then DTC *starts selecting new $\neg e_{ij}$'s, and proceeds without causing conflicts, until it selects $\neg(v_3 = v_5)$, which causes a $\mathcal{LA}(\mathbb{Z})$ conflict on the conflicting clause C_{35}, and forces* DTC *to backjump and to unit-propagate $(v_3 = v_5)$.*

Then, in short, $\mathcal{T}$-DPLL performs a Boolean search on (the negated values of) e_{ij}'s, backjumping also on the $\mathcal{T}$-conflicting clauses C_{13}, C_{56}, C_{23}, C_{24} and C_{14}, which in the

55. Here we assume for simplicity that $\mu'_{\mathcal{LA}(\mathbb{Q})}$ in C_{01} does not contain the redundant literal $(v_5 = 0)$. If this is not the case, one more $\mathcal{T}$-propagation of $(v_0 = v_1)$ is needed.

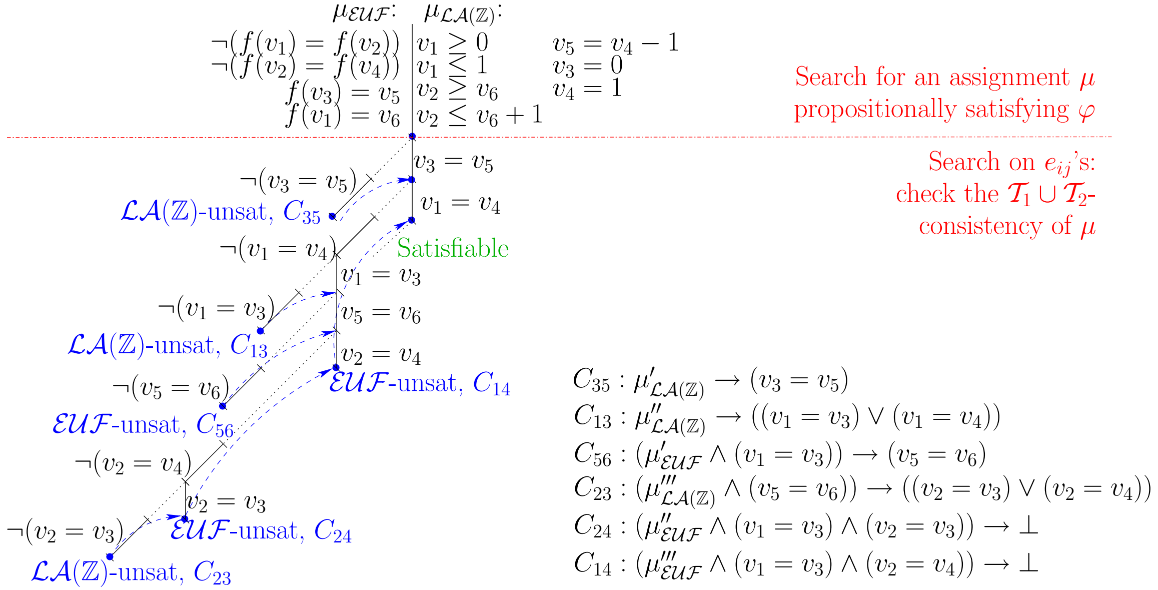

Figure 19. The DTC search tree for Example 8.8 on $\mathcal{LA}(\mathbb{Z}) \cup \mathcal{EUF}$, with no e_{ij}-deduction.

end causes the unit-propagation of $(v_1 = v_4)$*. Then,* $\mathcal{T}$-DPLL *selects a sequence of* $\neg e_{ij}$*'s without generating conflicts, and concludes that the formula is* $\mathcal{T}_1 \cup \mathcal{T}_2$*-satisfiable.*

Comparing Figures 17 and 19, we notice that here DTC mimics the behaviour of N.O. procedure in the sense that the backjumping steps on the clauses C_{35}, C_{13}, C_{56}, and C_{23} mimic the effects of performing the corresponding e_{ij}-deductions in Figure 17. We also notice that the case-splits induced by th e_{ij}-deductions in $\mathcal{LA}(\mathbb{Z})$ are handled within the DPLL search in DTC. Moreover, we notice that in Figure 19 DTC explores only seven branches, four for mimicking the corresponding e_{ij}-deductions and three for mimicking the three case-split branches in Figure 17.

We refer the reader to [39, 40, 45] for a more detailed description of DTC and for an analytical comparison between N.O. and DTC procedures.

8.4 SMT($\mathcal{EUF} \cup \mathcal{T}$) via Ackermann's expansion

When one of the theories $\mathcal{T}_i$ is $\mathcal{EUF}$, one further approach to the $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$ problem is to eliminate uninterpreted function symbols by means of Ackermann's expansion [1] 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 $f(t_1, \ldots, t_n)$ is replaced by a fresh variable $v_{f(t_1,\ldots,t_n)}$. Then, for every pair of distinct applications of the same function, $f(t_1, \ldots, t_n)$

and $f(u_1, \ldots, u_n)$, a congruence constraint

$$\bigwedge_{i=1}^{arity(f)} (ack(t_i) = ack(u_i)) \rightarrow (v_{f(t_1,\ldots,t_n)} = v_{f(u_1,\ldots,u_n)}), \tag{25}$$

is added, where ack is a function that maps each function application $g(w_1, \ldots, w_n)$ into the corresponding variable $v_{g(w_1,\ldots,w_n)}$, each variable into itself and is homomorphic wrt. the interpreted symbols. The atom $(ack(t_i) = ack(u_i))$ is not added if the two sides of the equality are syntactically identical; if so, the corresponding implication in (25) is dropped.

Example 8.9. *Consider the $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$ pure formula φ of examples 8.4 and 8.7:*

$$\begin{array}{ll} \mathcal{EUF}: & (v_3 = h(v_0)) \wedge (v_4 = h(v_1)) \wedge (v_6 = f(v_2)) \wedge (v_7 = f(v_5)) \wedge \\ \mathcal{LA}(\mathbb{Q}): & (v_0 \geq v_1) \wedge (v_0 \leq v_1) \wedge (v_2 = v_3 - v_4) \wedge (RESET_5 \rightarrow (v_5 = 0)) \wedge \\ Both: & (\neg RESET_5 \rightarrow (v_5 = v_8)) \wedge \neg(v_6 = v_7). \end{array} \tag{26}$$

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:

$$\begin{array}{ll} \mathcal{LA}(\mathbb{Q}): & (v_3 = v_{h(v_0)}) \wedge (v_4 = v_{h(v_1)}) \wedge (v_6 = v_{f(v_2)}) \wedge (v_7 = v_{f(v_5)}) \wedge \\ & (v_0 \geq v_1) \wedge (v_0 \leq v_1) \wedge (v_2 = v_3 - v_4) \wedge (RESET_5 \rightarrow (v_5 = 0)) \wedge \\ & (\neg RESET_5 \rightarrow (v_5 = v_8)) \wedge \neg(v_6 = v_7) \wedge \\ Congruence: & ((v_0 = v_1) \rightarrow (v_{h(v_0)} = v_{h(v_1)})) \wedge ((v_2 = v_5) \rightarrow (v_{f(v_2)} = v_{f(v_5)})), \end{array} \tag{27}$$

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

Notice that in Example 8.9 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 (27). [44] presents a comparison between Dtc and Ackermann's expansion on SMT($\mathcal{EUF} \cup \mathcal{T}$).

An interesting variant of Ackermann's expansion has been introduced for eager encodings (see §9.3) in [48]. This method differs from that in [1] 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 [48] for details.)

9. Related approaches for SMT

All the most extensive empirical evaluations performed in the last years [85, 65, 133, 38, 18, 164, 165] 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 [93, 10]. First, DPLL performs a depth-first-search on assignments, so that it requires a polynomial amount of memory. [56.] This is not the case, e.g., of OBDD's [46]. 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 [166]. We refer the reader to [10] 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 [46] with theory-information in order to handle complex verification problems. [51] integrated OBDD's with an (incomplete) quadratic constraint solver to verify transition systems with integer data values; [127] developed Difference Decision Diagrams (DDDs), OBDD-like data structures handling Boolean combinations of temporal constraints, and used them to verify timed systems; [94] developed OBDD-based

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

procedures for $\mathcal{EUF}$; [187] developed a library of procedures combining OBDD's and a solver for Pressburger arithmetic, and used them to verify infinite-state systems. [57.] [144] combine OBDDs with superposition theorem provers (see also §9.2). In a more general perspective, [2] 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., [105, 114, 82, 101, 175, 119, 103]). 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, [82] 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; [175] generalized the two-watched-literals technique to Boolean circuits.

[119, 103] focused on the problem of enumerating complete sets of assignments for Boolean circuit representations: [119] 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; [103] 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 [126], we believe they may find a natural application in lazy SMT. With some noteworthy exception (e.g., [141] extended the circuit-based solver in [101] 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* [58.] approach [8, 144, 4, 109, 33, 110]. The main idea is that of producing $\mathcal{T}_i$-*solvers* for 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* [59.]. This is done by incorporating the axioms of the theory and by introducing ad-hoc control strategies in order to drive the search.

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 [8] a refutationally-complete rewrite-based inference system was shown to generate finitely-many clauses on satisfiability problems in a

57. 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.
58. Also referred as *superposition-based* approach.
59. The superposition calculus [14] 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 [152]).

bunch of theories. [4, 109, 33, 110] 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, [4] performed an empirical comparison, showing performances comparable to those of CVCLite. The empirical evaluation of these techniques, however, is still at a preliminary stage.

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 [109], 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 [109, 110].

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 [8]. Fourth, and probably most important, under some sufficient conditions [4], 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*.

Currently, the approach has focused on theories of interest for the verification of software programs, including $\mathcal{AR}$ (with and without extensionality) and $\mathcal{LI}$ (§4.2), and other theories not explicitly mentioned in this paper — like those of records, finite sets, integer offsets— and of their combination. We refer the reader to [3] for a more detailed survey.

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. [60.])

Effective encodings into SAT have been conceived for a significant amount of first-order theories of interest for formal verification, including $\mathcal{EUF}$, $\mathcal{CLU}$ [61.], $\mathcal{DL}$, $\mathcal{SUF}$ [62.], $\mathcal{LA}$, and $\mathcal{AR}$ [181, 47, 171, 170, 156]. In particular, two main families of encodings are worth-mentioning:

- in the *small-domain encoding, SD* [47, 174], 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 the original input formula into a Boolean formula;

- in the *per-constraint-encoding* [94, 171, 170] 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 [156] 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 [156, 180, 117], 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.

60. 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.

61. $\mathcal{CLU}$ 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 [47].

62. $\mathcal{SUF}$ is the logic of *Separation predicates and Uninterpreted Functions*, combining $\mathcal{EUF}$ and $\mathcal{DL}$ [156].

- 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., [48, 181, 182]).

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 $\mathcal{LA}$ [171, 170]. E.g., although SMT($\mathcal{DL}$) and SMT($\mathcal{LA}$) are NP-complete, the encodings proposed in [171] and [170] 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 [85, 65, 38, 133], and no eager SMT tool took part at the SMT competitions [18, 164, 165]. 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 [111] 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.

[83] 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.

[108] 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

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

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

[3] A. Armando, M. P. Bonacina, S. Ranise, , and S. Schulz. New results on rewrite-based satisfiability procedures. *ACM Transactions on Computational Logic (TOCL)*, 2008. To appear.

[4] 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.

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

[6] 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.

[7] A. Armando and E. Giunchiglia. Embedding Complex Decision Procedures inside an Interactive Theorem Prover. *Annals of Mathematics and Artificial Intelligence*, **8**(3–4):475–502, 1993.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] C. Barrett, I. Shikanian, and C. Tinelli. An Abstract Decision Procedure for Satisfiability in the Theory of Recursive Data Types. In *Proc. PDPAR'06*, volume **174** of *ENTCS*. Elsevier, 2007.

[22] C. Barrett and C. Tinelli. Cvc3. In *Proc. CAV'07*, volume **4590** of *Lecture Notes in Computer Science*. Springer, 2007.

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

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

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

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

[32] M. P. Bonacina and M. Echenim. Rewrite-Based Satisfiability Procedures for Recursive Data Structures. In *Proc. PDPAR'06*, volume **174** of *ENTCS*. Elsevier, 2007.

[33] 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*, volume **4130** of *LNAI*, 2006.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

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

[43] 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*, volume **4590** of *LNCS*. Springer, 2007.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

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

[50] 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.

[51] 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.

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

[53] . H. Cormen, C. E. Leiserson, and R. R. Rivest. *Introduction to Algorithms*. MIT Press, 1998.

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

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

[56] CVC. http://verify.stanford.edu/CVC.

[57] CVCLite. http://verify.stanford.edu/CVCL.

[58] 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.

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

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

[61] G. de Micheli. *Synthesis of Digital Circuits.* Stanford University, 1993.

[62] L. de Moura and N. Bjørner. Model-based Theory Combination. In *Proc. of the 5th Workshop on Satisfiability Modulo Theories SMT'07*, 2007. http://www.lsi.upc.edu/~oliveras/smt07/.

[63] L. de Moura and N. Bjørner. System Description: Z3 0.1. In *3rd Int. Competition of Satisfiability Modulo Theories tools*, 2007. http://research.microsoft.com/projects/z3/smtcomp07.pdf.

[64] L. de Moura, S. Owre, H. Ruess, J. Rushby, and N. Shankar. The ICS Decision Procedures for Embedded Deduction. In *Proc. IJCAR'04*, volume **3097** of *LNCS*, pages 218–222. Springer, 2004.

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

[66] 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.

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

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

[69] 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.

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

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

[72] B. Dutertre and L. de Moura. System Description: Yices 1.0. In *Proc. SMT-COMP'06*, 2006.

[73] B. Dutertre and L. de Moura. System Description: Yices 1.0. In *Proc. on 2nd SMT competition, SMT-COMP'06*, 2006.
Available at yices.csl.sri.com/yices-smtcomp06.pdf.

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

[75] 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.

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

[77] 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.

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

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

[80] P. Fontaine, S. Ranise, and C. G. Zarba. Combining lists with non-stably infinite theories. In *Proc LPAR'04*, volume **3452** of *LNCS*. Springer, 2004.

[81] Anders Franzén. Using Satisfiability Modulo Theories for Inductive Verification of Lustre Programs. In *Proc. BMC'05*, volume **144** of *ENTCS*. Elsevier, 2006.

[82] 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.

[83] 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.

[84] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proc. Computer Aided Verification, 19th International Conference, CAV 2007*, volume **4590** of *LNCS*, pages 519–531, 2007.

[85] 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.

[86] S. Ghilardi. Model theoretic methods in combined constraint satisfiability. *Journal of Automated Reasoning*, **33**(3), 2005.

[87] S. Ghilardi, E. Nicolini, and D. Zucchelli. A comprehensive framework for combined decision procedures. In *Proc. FroCos'05*, volume **3717** of *LNCS*. Springer, 2005.

[88] M. L. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research JAIR*, **1**, 1993.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

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

[100] I. Horrocks and P. F. Patel-Schneider. Optimising propositional modal satisfiability for description logic subsumption. In *Proc. AISC'98*, volume **1476** of *LNAI*. Springer, 1998.

[101] 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.

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

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

[104] P. Johannsen and R. Drechsler. Speeding Up Verification of RTL Designs by Computing One-to-one Abstractions with Reduced Signal Widths. In *VLSI-SOC*, 2001.

[105] 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.

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

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

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

[109] 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.

[110] 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.

[111] 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.

[112] S. Krstic and A. Goel. Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *Proc. Frontiers of Combining Systems, 6th International Symposium, FroCoS 2007*, volume **4720** of *LNAI*. Springer, 2007.

[113] S. Krstić, A. Goel, J. Grundy, and C. Tinelli. Combined Satisfiability Modulo Parametric Theories. In *TACAS'07*, volume **4424** of *LNCS*. Springer, 2007.

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

[115] 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.

[116] 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.

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

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

[119] 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.

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

[121] 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.

[122] Chu Min Li and Anbulagan. Look-Ahead Versus Look-Back for Satisfiability Problems. In *Principles and Practice of Constraint Programming*, volume **1330** of *LNCS*, 1997.

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

[124] 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.

[125] P. Manolios, S.K. Srinivasan, and D. Vroon. Automatic Memory Reductions for RTL-Level Verification. In *Proc. ICCAD 2006*. ACM Press, 2006.

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

[127] 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.

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

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

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

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

[132] 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.

[133] 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.

[134] 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.

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

[136] 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.

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

[138] D. C. Oppen. Complexity, Convexity and Combinations of Theories. *Theoretical Computer Science*, **12**:291–302, 1980.

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

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

[141] 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.

[142] K. Pipatsrisawat and A. Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *Proc. SAT'07*, volume **4501** of *LNCS*. Springer, 2007.

[143] 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.

[144] 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.

[145] S. Ranise, C. Ringeissen, and C. G. Zarba. Combining Data Structures with Nonstably Infinite Theories Using Many-Sorted Logic. In *Proc FroCos'05*, volume **3717** of *LNCS*. Springer, 2005.

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

[147] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). http://www.SMT-LIB.org, 2006.

[148] 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 http://www.SMT-LIB.org.

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

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

[151] H. Rueß and N. 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.

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

[153] 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.

[154] R. Sebastiani. From KSAT to Delayed Theory Combination: Exploiting DPLL Outside the SAT Domain. In *Proc. Frontiers of Combining Systems, FroCoS'07*, volume **4720** of *LNCS*. Springer, 2007. Invited talk.

[155] 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.

[156] 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.

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

[158] 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.

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

[160] 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.

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

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

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

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

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

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

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

[168] 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.

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

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

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

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

[173] 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.

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

[175] 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.

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

[177] C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In *Proc. Frontiers of Combining Systems, FroCoS'06*, Applied Logic. Kluwer, 1996.

[178] C. Tinelli and C. Ringeissen. Unions of non-disjoint theories and combinations of satisfiability procedures. *Theoretical Computer Science*, **290**(1), 2003.

[179] C. Tinelli and C. Zarba. Combining nonstably infinite theories. *Journal of Automated Reasoning*, **34**(3), 2005.

[180] UCLID. http://www-2.cs.cmu.edu/~uclid.

[181] 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.

[182] 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.

[183] 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.

[184] 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.

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

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

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

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

[189] C. G. Zarba. A Tableau Calculus for Combining Non-disjoint Theories. In *Proc. Tableaux'02*, volume **2381** of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2002.

[190] C. G. Zarba. Combining Sets with Integers. In *FroCos'02*, volume **2309** of *Lecture Notes in Computer Science*, pages 103–116. Springer, 2002.

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

[192] 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.

[193] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proc. CAV'02*, volume **2404** of *LNCS*, pages 17–36. Springer, 2002.