Handbook of Satisfiability

Clark Barrett[1] Roberto Sebastiani[2] Sanjit A. Seshia[3] Cesare Tinelli[4]
[1]New York University, barrett@cs.nyu.edu
[2]Università di Trento, rseba@disi.unitn.it
[3]University of California, Berkeley, sseshia@eecs.berkeley.edu
[4]The University of Iowa, tinelli@cs.uiowa.edu

IOS Press

Contents

Part I. Extensions

Chapter 1. Satisfiability Modulo Theories **1**
1.1 Introduction 1
1.2 Background 3
1.2.1 Formal preliminaries 3
1.2.1.1 Syntax 3
1.2.1.2 Semantics 4
1.2.1.3 Combined Theories 5
1.2.1.4 Abstraction 6
1.2.2 Some theories of interest 6
1.2.2.1 Equality 6
1.2.2.2 Arithmetic 7
1.2.2.3 Arrays 8
1.2.2.4 Fixed-width bit-vectors 8
1.2.2.5 Inductive data types 8
1.3 Eager Encodings to SAT 9
1.3.1 Overview 9
1.3.1.1 Operation 10
1.3.1.2 Eliminating Lambdas 10
1.3.1.3 Eliminating Function Applications 11
1.3.1.4 Summary 12
1.3.2 Small-domain encodings 13
1.3.2.1 Equalities 14
1.3.2.2 Difference Logic 14
1.3.2.3 UTVPI Constraints 15
1.3.2.4 Sparse, Mostly-Difference Constraints 16
1.3.2.5 Summary 16
1.3.3 Direct encoding of theory axioms 17
1.3.4 Hybrid eager approaches 18
1.4 Integrating Theory Solvers into SAT Engines 18
1.4.1 Theory Solvers and their desirable features 19
1.4.2 A generalized DPLL schema 19
1.4.3 Enhancements to the schema 22
1.4.3.1 Normalizing $\mathcal{T}$-atoms. 22

1.4.3.2 Static learning 23
1.4.3.3 Early pruning 23
1.4.3.4 $\mathcal{T}$-propagation 24
1.4.3.5 $\mathcal{T}$-backjumping and $\mathcal{T}$-learning 24
1.4.3.6 Generating partial assignments 25
1.4.3.7 Pure-literal filtering 26
1.4.4 An abstract framework 26
1.5 Theory Solvers 30
1.5.1 Shostak's method 30
1.5.1.1 Combining Shostak theories 32
1.5.2 Splitting on demand 33
1.5.3 Layered theory solvers 35
1.5.4 Rewriting-based theory solvers 35
1.6 Combining Theories 36
1.6.1 A Logical Framework for Nelson-Oppen Combination 37
1.6.2 The Nelson-Oppen Procedure 40
1.6.3 Delayed Theory Combination 40
1.6.4 Ackermann's expansion 42
1.7 Extensions and Enhancements 43
1.7.1 Combining eager and lazy approaches 43
1.7.2 Handling quantifiers 44
1.7.3 Producing models 46
1.7.4 Producing proofs 46
1.7.5 Identifying unsatisfiable cores 46
1.7.6 Computing interpolants 47

Bibliography **49**

Part I

Extensions

Chapter 1

Satisfiability Modulo Theories

1.1. Introduction

Applications in artificial intelligence and formal methods for hardware and software development have greatly benefited from the recent advances in SAT. Often, however, applications in these fields require determining the satisfiability of formulas in more expressive logics such as first-order logic. Despite the great progress made in the last twenty years, general-purpose first-order theorem provers (such as provers based on the resolution calculus) are typically not able to solve such formulas directly. The main reason for this is that many applications require not general first-order satisfiability, but rather satisfiability with respect to some *background theory*, which fixes the interpretations of certain predicate and function symbols. For instance, applications using integer arithmetic are not interested in whether there exists a nonstandard interpretation of the symbols $<$, $+$, and 0 that makes the formula

$$x < y \ \wedge \ \neg(x < y + 0)$$

satisfiable. Instead, they are interested in whether the formula is satisfiable in an interpretation in which $<$ is the usual ordering over the integers, $+$ is the integer addition function, and 0 is the additive identity. General-purpose reasoning methods can be forced to consider only interpretations consistent with a background theory $\mathcal{T}$, but only by explicitly incorporating the axioms for $\mathcal{T}$ into their input formulas. Even when this is possible,[1] the performance of such provers is often unacceptable. For some background theories, a more viable alternative is to use reasoning methods tailored to the theory in question. This is particularly the case for *quantifier-free* formulas, first-order formulas with no quantifiers but possibly with variables, such as the formula above.

For many theories, specialized methods actually yield *decision procedures* for the satisfiability of quantifier-free formulas or some subclass thereof. This is the case, thanks to classical results in mathematics, for the theory of real numbers and the theory of integer arithmetic (without multiplication). In the last two decades, however, specialized decision procedures have also been discovered for

[1] Some background theories such as the theory of real numbers or the theory of finite trees, cannot be captured by a finite set of first-order formulas, or, as in the case of the theory of integer arithmetic (with multiplication), by *any* decidable set of first-order formulas.

a long and still growing list of other theories with practical applications. These include certain theories of arrays and of strings, several variants of the theory of finite sets or multisets, the theories of several classes of lattices, the theories of finite, regular and infinite trees, of lists, tuples, records, queues, hash tables, and bit-vectors of a fixed or arbitrary finite size.

The research field concerned with the satisfiability of formulas with respect to some background theory is called *Satisfiability Modulo Theories*, or SMT, for short. In analogy with SAT, SMT procedures (whether they are decision procedures or not) are usually referred to as *SMT solvers*. The roots of SMT can be traced back to early work in the late 1970s and early 1980s on using decision procedures in formal methods by such pioneers as Nelson and Oppen [108, 107], Shostak [139, 140, 138], and Boyer and Moore [33, 32].[2] Modern SMT research started in the late 1990s with various independent attempts [6, 79, 4, 121, 41] to build more scalable SMT solvers by exploiting advances in SAT technology. The last few years have seen a great deal of interest and research on the foundational and practical aspects of SMT. SMT solvers have been developed in academia and industry with increasing scope and performance. SMT solvers or techniques have been integrated into: interactive theorem provers for high-order logic (such as HOL, Isabelle, and PVS); extended static checkers (such as Boogie and ESC/Java 2); verification systems (such as ACL2, Caduceus, SAL, UCLID, and Why); formal CASE environments (such as KeY); model checkers (such as BLAST, Eureka, MAGIC and SLAM); certifying compilers (such as Touchstone and TVOC); unit test generators (such as DART, EXE, CUTE and PEX).

This chapter provides a brief overview of SMT and its main approaches, together with references to the relevant literature for a deeper study. In particular, it focuses on the two most successful major approaches so far for implementing SMT solvers, usually referred to as the “eager” and the “lazy” approach.

The *eager* approach is based on devising efficient, specialized translations to convert an input formula into an equisatisfiable propositional formula using enough relevant consequences of the theory $\mathcal{T}$. The approach applies in principle to any theory with a decidable ground satisfiability problem, possibly however at the cost of a significant blow-up in the translation. Its main allure is that the translation imposes upfront all theory-specific constraints on the SAT solver's search space, potentially solving the input formula quickly; in addition, the translated formula can be given to any off-the-shelf SAT solver. Its viability depends on the ability of modern SAT solvers to quickly process relevant theory-specific information encoded into large SAT formulas.

The *lazy* approach consists in building ad-hoc procedures implementing, in essence, an inference system specialized on a background theory $\mathcal{T}$. The main advantage of theory-specific solvers is that one can use whatever specialized algorithms and data structures are best for the theory in question, which typically leads to better performance. The common practice is to write *theory solvers* just for conjunctions of literals—i.e., atomic formulas and their negations. These pared down solvers are then embedded as separate submodules into an efficient SAT solver, allowing the joint system to accept quantifier-free formulas with an

[2] Notable early systems building on this work are the Boyer-Moore prover, PVS, Simplify, SteP, and SVC.

$$
\begin{array}{lll}
t & ::= c & \text{where } c \in \Sigma^{\mathrm{F}} \text{ with arity } 0 \\
 & \mid f(t_1, \ldots, t_n) & \text{where } f \in \Sigma^{\mathrm{F}} \text{ with arity } n > 0 \\
 & \mid ite(\varphi, t_1, t_2) & \\
\varphi & ::= A & \text{where } A \in \Sigma^{\mathrm{P}} \text{ with arity } 0 \\
 & \mid p(t_1, \ldots, t_n) & \text{where } p \in \Sigma^{\mathrm{P}} \text{ with arity } n > 0 \\
 & \mid t_1 = t_2 \mid \bot \mid \top \mid \neg\varphi_1 & \\
 & \mid \varphi_1 \rightarrow \varphi_2 \mid \varphi_1 \leftrightarrow \varphi_2 & \\
 & \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 &
\end{array}
$$

Figure 1.1. Ground terms and formulas

arbitrary Boolean structure.

The rest of this chapter is structured as follows. Section 1.2 provides background information, with formal preliminaries and a brief description of a few theories popular in SMT applications. The next two sections respectively describe the eager and the lazy approach in some detail. Section 1.5 describes some general methods for building theory solvers for the lazy approach. Section 1.6 focuses on techniques for combining solvers for different theories into a solvers for a combination of these theories. Finally, Section 1.7, describes some important extension and enhancements on the methods and techniques described in the previous sections.

1.2. Background

1.2.1. Formal preliminaries

In this chapter we will work in the context of (classical) first-order logic with equality (see, e.g., [63, 65]). To make the chapter more self-contained, however, we introduce here all the relevant basic concepts and notation.

1.2.1.1. Syntax

A *signature* Σ is set of *predicate* and *function* symbols, each with an associated *arity*, a non-negative number. For any signature Σ, we denote by Σ^{F} and Σ^{P} respectively the set of function and of predicate symbols in Σ. We call the 0-arity symbols of Σ^{F} *constant* symbols, and usually denote them by the letters a, b possibly with subscripts. We call the 0-arity symbols of Σ^{P} *propositional* symbols, and usually denote them by the letters A, B, possibly with subscripts. Also, we use f, g and p, q, possibly with subscripts, to denote respectively the non-constant symbols of Σ^{F} and the non-propositional symbols of Σ^{P}.

In this chapter, we are mostly interested in quantifier-free terms and formulas built with the symbols of a given signature Σ. As a technical convenience, we treat the (free) variables of a quantifier-formula as constants in a suitable expansion of Σ. For example, if Σ is the signature of integer arithmetic we consider the formula $x < y + 1$ as a *ground* (i.e., variable-free) formula in which x and y are additional constant symbols. Formally, a *ground (Σ-)term* t and a *ground (Σ-)formula* φ are expressions in the language defined by the abstract grammar in Figure 1.1. As

usual, we call *atomic formula* (or *atom*) a formula of the form A, $p(t_1, \ldots, t_n)$, $t_1 = t_2$, $\bot$, or $\top$.[3] A *(Σ-)literal* is an atomic Σ-formula or the negation of one. We will use the letter l possibly with subscripts, to denote literals. The *complement* of a literal l, written $\neg l$ for simplicity, is $\neg\alpha$ if l is an atomic formula α, and is α if l is $\neg\alpha$. A *(Σ-)clause* is a disjunction $l_1 \vee \cdots \vee l_n$ of literals. We will 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 $\bigvee_i \neg l_i$ where each l_i and l_j is a positive literal. We denote clauses with the letter c, possibly with subscripts, and identify the empty clause, i.e., the empty disjunction of literals, with the formula $\bot$. A *unit clause* is a clause consisting of a single literal (not containing $\bot$ or $\top$). When μ is a finite set of literals $l_1, \ldots, l_n$, we may denote by $\neg\mu$ the clause $\neg l_1 \vee \cdots \vee \neg l_n$. Correspondingly, if c is a clause $l_1 \vee \cdots \vee l_n$, we denote by $\neg c$ the set $\{\neg l_1, \ldots, \neg l_n\}$. A *CNF formula* is a conjunction $c_1 \wedge \cdots \wedge c_n$ of zero or more clauses. When it leads to no ambiguities, we will sometimes also write CNF formulas in set notation $\{c_1, \ldots, c_n\}$, or simply replace the $\wedge$ connectives by commas.

1.2.1.2. Semantics

Formulas are given a meaning, that is, a truth value from the set $\{\textbf{true}, \textbf{false}\}$, by means of *(first-order) models*. A model $\mathcal{A}$ for a signature Σ, or Σ-model, is a pair consisting of a non-empty set A, the *universe* of the model, and a mapping $(_)^{\mathcal{A}}$ assigning to each constant symbol $a \in \Sigma^{\mathrm{F}}$ an element $a^{\mathcal{A}} \in A$, to each function symbol $f \in \Sigma^{\mathrm{F}}$ of arity $n > 0$ a total function $f^{\mathcal{A}} : A^n \rightarrow A$, to each propositional symbol $B \in \Sigma^{\mathrm{P}}$ an element $B^{\mathcal{A}} \in \{\textbf{true}, \textbf{false}\}$, and to each $p \in \Sigma^{\mathrm{P}}$ of arity $n > 0$ a total function $p^{\mathcal{A}} : A^n \rightarrow \{\textbf{true}, \textbf{false}\}$. This mapping uniquely determines a homomorphic extension, also denoted as $(_)^{\mathcal{A}}$, that maps each Σ-term t to an element $t^{\mathcal{A}} \in A$, and each Σ-formula φ to an element $\varphi^{\mathcal{A}} \in \{\textbf{true}, \textbf{false}\}$. The extension, which we call an *interpretation* of the terms and the formulas, is defined as expected. In particular, for any $\mathcal{A}$: $f(t_1, \ldots, t_n)^{\mathcal{A}} = f^{\mathcal{A}}(t_1{}^{\mathcal{A}}, \ldots, t_n{}^{\mathcal{A}})$; $ite(\varphi, t_1, t_2)^{\mathcal{A}}$ equals $t_1{}^{\mathcal{A}}$ if $\varphi^{\mathcal{A}} = \textbf{true}$ and $t_2{}^{\mathcal{A}}$ otherwise; $p(t_1, \ldots, t_n)^{\mathcal{A}} = p^{\mathcal{A}}(t_1{}^{\mathcal{A}}, \ldots, t_n{}^{\mathcal{A}})$; $\bot^{\mathcal{A}} = \textbf{false}$; $\top^{\mathcal{A}} = \textbf{true}$; and $(t_1 = t_2)^{\mathcal{A}} = \textbf{true}$ iff $t_1^{\mathcal{A}} = t_2^{\mathcal{A}}$.[4]

We say that a Σ-model $\mathcal{A}$ *satisfies* (resp. *falsifies*) a Σ-formula φ iff $\varphi^{\mathcal{A}}$ is **true** (resp. **false**). In SMT, one is not interested in arbitrary models but in models belonging to a given *theory* $\mathcal{T}$ constraining the interpretation of the symbols of Σ. Following the more recent SMT literature, we define Σ-theories most generally as just sets of one or more (possibly infinitely many) Σ-models. Then, we say that a ground Σ-formula is satisfiable in a Σ-theory $\mathcal{T}$, or $\mathcal{T}$*-satisfiable*, iff there is an element of the set $\mathcal{T}$ that satisfies φ. Similarly, a set Γ of ground Σ-formulas $\mathcal{T}$*-entails* a ground formula φ, written $\Gamma \models_{\mathcal{T}} \varphi$, iff every model of $\mathcal{T}$ that satisfies all formulas in Γ satisfies φ as well. We say that Γ is $\mathcal{T}$*-consistent* iff $\Gamma \not\models_{\mathcal{T}} \bot$, and that φ is $\mathcal{T}$*-valid* iff $\emptyset \models_{\mathcal{T}} \varphi$. We call a clause c a *theory lemma* if c is $\mathcal{T}$-valid (i.e., $\emptyset \models_{\mathcal{T}} c$). All these notions reduce exactly to the corresponding notions in

[3] Note that, by allowing propositional symbols in signatures this language properly includes the language of propositional logic.

[4] Note that we are using the symbol = both as a symbol of the logic and as the usual meta-symbol for equality. The difference, however, should be always clear from context.

standard first-order logic by choosing as $\mathcal{T}$ the set of all Σ-models; for that case, we drop $\mathcal{T}$ from the notation (and for instance write just $\Gamma \models \varphi$).

Typically, given a Σ-theory $\mathcal{T}$, one is actually interested in the $\mathcal{T}$-satisfiability of ground formulas containing additional, *uninterpreted* symbols, i.e., predicate or function symbols not in Σ. This is particularly the case for uninterpreted constant symbols—which, as we have seen, play the role of free variables—and uninterpreted propositional symbols—which can be used as abstractions of other formulas. Formally, uninterpreted symbols are accommodated in the definitions above by considering instead of $\mathcal{T}$, the theory $\mathcal{T}'$ defined as follows. Let Σ' be any signature including Σ. An *expansion* $\mathcal{A}'$ to Σ' of a Σ-model $\mathcal{A}$ is a Σ'-model that has the same universe as $\mathcal{A}$ and agrees with $\mathcal{A}$ on the interpretation of the symbols in Σ. The theory $\mathcal{T}'$ is the set of *all possible expansions* of the models of $\mathcal{T}$ to Σ'. To keep the terminology and notation simple, we will still talk about $\mathcal{T}$-satisfiability, $\mathcal{T}$-entailment and so on when dealing with formulas containing uninterpreted symbols, but with the understanding that we actually mean the Σ'-theory $\mathcal{T}'$ where Σ' is a suitable expansion of $\mathcal{T}$'s original signature.

Then, the *ground $\mathcal{T}$-satisfiability problem* is the problem of determining, given a Σ-theory $\mathcal{T}$, the $\mathcal{T}$-satisfiability of ground formulas over an arbitrary expansion of Σ with uninterpreted constant symbols. Since a formula φ is $\mathcal{T}$-satisfiable iff $\neg\varphi$ is $\mathcal{T}$-valid, the ground $\mathcal{T}$-satisfiability problem has a dual *ground $\mathcal{T}$-validity problem*. The literature in the field (especially the older literature) sometimes adopts this dual view.[5] Finally, a theory $\mathcal{T}$ is *convex* if for all sets μ of ground Σ'-literals (where Σ' is an expansion of Σ with uninterpreted constant symbols) and all sets E of equalities between uninterpreted constant symbols in Σ', $\mu \models_{\mathcal{T}} \bigvee_{e \in E} e$ iff $\mu \models_{\mathcal{T}} e$ for some $e \in E$.

1.2.1.3. Combined Theories

Several applications of SMT deal with formulas involving two or more theories at once. In that case, satisfiability is understood as being modulo some combination of the various theories. If two theories $\mathcal{T}_1$ and $\mathcal{T}_2$ are both defined axiomatically, their combination can simply be defined as the theory axiomatized by the union of the axioms of the two theories, $\mathcal{T}_1$ and $\mathcal{T}_2$. This is adequate if the signatures of the two theories are disjoint. If, instead, $\mathcal{T}_1$ and $\mathcal{T}_2$ have symbols in common, one has to consider whether a shared function (resp. predicate) symbol is meant to stand for the same function (resp. relation) in each theory or not. In the latter case, a proper signature renaming must be applied to the theories before taking the union of their axioms.

With theories specified as sets of first order models, as done here, a suitable notion of theory combination is defined as follows. Let us say that a Σ-model $\mathcal{A}$ is the Σ-*reduct* of a Σ'-model $\mathcal{B}$ with $\Sigma' \supseteq \Sigma$ if $\mathcal{A}$ has the same universe as $\mathcal{B}$ and interprets the symbols of Σ exactly as $\mathcal{B}$ does. Then, the *combination* $\mathcal{T}_1 \oplus \mathcal{T}_2$ of $\mathcal{T}_1$ and $\mathcal{T}_2$ is the set of all $(\Sigma_1 \cup \Sigma_2)$-models $\mathcal{B}$ whose Σ_1-reduct is isomorphic to a model of $\mathcal{T}_1$ and whose Σ_2-reduct is isomorphic to a model of

[5] In $\mathcal{T}$-validity problems, any uninterpreted constant symbols behave like *universally* quantified variables, so it is also common to see ground $\mathcal{T}$-validity being described in the literature as $\mathcal{T}$-validity of *universal* Σ-formulas.

$\mathcal{T}_2$.[6] The correspondence with the axiomatic case is given by the following fact (see, e.g., [148]): when each $\mathcal{T}_i$ is the set of all Σ_i-models that satisfy some set Γ_i of first-order axioms, $\mathcal{T}_1 \oplus \mathcal{T}_2$ is precisely the set of all $(\Sigma_1 \cup \Sigma_2)$-models that satisfy $\Gamma_1 \cup \Gamma_2$.

1.2.1.4. Abstraction

For abstraction purposes, we associate with every signature Σ (possibly containing uninterpreted symbols in the sense above) a signature Ω consisting of the propositional symbols of Σ plus a set of new propositional symbols having the same cardinality as the set of ground Σ-atoms. We then fix a bijection $\mathcal{T}2\mathcal{B}$, called *propositional abstraction*, between the set of ground Σ-formulas without *ite* expressions and the propositional formulas over Ω. This bijection maps each propositional symbol of Σ to itself and each non-propositional Σ-atom to one of the additional propositional symbols of Ω, and is homomorphic with respect to the logical operators.[7] The restriction to formulas with no *ite*'s is without loss of generality because *ite* constructs can be eliminated in advance by a satisfiability-preserving transformation that repeatedly applies the following rule to completion: let $ite(\psi, t_1, t_2)$ be a subterm appearing in a formula φ; we replace this term in φ by some new uninterpreted constant a and return the conjunction of the result with the formula $ite(\psi, a = t_1, a = t_2)$. [8] We denote by $\mathcal{B}2\mathcal{T}$ the inverse of $\mathcal{T}2\mathcal{B}$, and call it *refinement*.

To streamline the notation we will often write φ^p to denote $\mathcal{T}2\mathcal{B}(\varphi)$. Also, if μ is a set of Σ-formulas, we will write μ^p to denote the set $\{\varphi^p \mid \varphi \in \mu\}$; if μ^p is a set of Boolean literals, then μ will denote $\mathcal{B}2\mathcal{T}(\mu^p)$. A Σ-formula φ is *propositionally unsatisfiable* if $\varphi^p \models \bot$. We will often write $\mu \models_p \varphi$ to mean $\mu^p \models \varphi^p$. We point out that for any theory $\mathcal{T}$, $\mu \models_p \varphi$ implies $\mu \models_{\mathcal{T}} \varphi$, but not vice versa.

1.2.2. Some theories of interest

In order to provide some motivation and connection to applications, we here give several examples of theories of interest and how they are used in applications.[9]

1.2.2.1. Equality

As described above, a theory usually imposes some restrictions on how function or predicate symbols may be interpreted. However, the most general case is a theory which imposes no such restrictions, in other words, a theory that includes all possible models for a given signature.

Given any signature, we denote the theory that includes all possible models of that theory as $\mathcal{T}_{\mathcal{E}}$. It is also sometimes called the *empty* theory because its finite

[6] We refer the reader to, e.g., [63] for a definition of isomorphic models. Intuitively, two models $\mathcal{A}$ and $\mathcal{B}$ are isomorphic if they are identical with the possible exception that the universe of $\mathcal{B}$ is a renaming of the universe of $\mathcal{A}$.

[7] That is, $\mathcal{T}2\mathcal{B}(\bot) = \bot$, $\mathcal{T}2\mathcal{B}(\varphi_1 \wedge \varphi_2) = \mathcal{T}2\mathcal{B}(\varphi_1) \wedge \mathcal{T}2\mathcal{B}(\varphi_2)$, and so on.

[8] The newly-introduced *ite* can be thought of as syntactic sugar for $\psi \rightarrow a = t_1 \wedge \neg\psi \rightarrow a = t_2$. Alternatively, to avoid potential blowup of the formula, a formula-level if-then-else operator can be introduced into the language syntax.

[9] See [96] for an earlier related survey.

axiomatization is just $\emptyset$ (the empty set). Because no constraints are imposed on the way the symbols in the signature may be interpreted, it is also sometimes called the theory of equality with uninterpreted functions (EUF). The satisfiability problem for conjunctions of ground formulas modulo $\mathcal{T}_{\mathcal{E}}$ is decidable in polynomial time using a procedure known as *congruence closure* [10, 60, 112].

Some of the first applications that combined Boolean reasoning with theory reasoning used this simple theory [108]. Uninterpreted functions are often used as an abstraction technique to remove unnecessarily complex or irrelevant details of a system being modeled. For example, suppose we want to prove that the following set of literals is unsatisfiable: $\{a*(f(b)+f(c)) = d, b*(f(a)+f(c)) \neq d, a = b\}$. At first, it may appear that this requires reasoning in the theory of arithmetic. However, if we abstract $+$ and $*$ by replacing them with uninterpreted functions g and h respectively, we get a new set of literals: $\{h(a, g(f(b), f(c))) = d, h(b, g(f(a), f(c))) \neq d, a = b\}$. This set of literals can be proved unsatisfiable using only congruence closure.

1.2.2.2. Arithmetic

Let $\Sigma_{\mathcal{Z}}$ be the signature $(0, 1, +, -, \leq)$. Let the theory $\mathcal{T}_{\mathcal{Z}}$ consist of the model that interprets these symbols in the usual way over the integers.[10] This theory is also known as *Presburger arithmetic*.[11] We can define the theory $\mathcal{T}_{\mathcal{R}}$ to consist of the model that interprets these same symbols in the usual way over the reals.

Let $\mathcal{T}'_{\mathcal{Z}}$ be the extension of $\mathcal{T}_{\mathcal{Z}}$ with an arbitrary number of uninterpreted constants (and similarly for $\mathcal{T}'_{\mathcal{R}}$). The question of satisfiability for conjunctions of ground formulas in either of these theories is decidable. Ground satisfiability in $\mathcal{T}'_{\mathcal{R}}$ is actually decidable in polynomial time [86], though exponential methods such as those based on simplex often perform best in practice (see, e.g. [61]). On the other hand, ground $\mathcal{T}'_{\mathcal{Z}}$-satisfiability is NP-complete [119].

Two important related problems have to do with restricting the syntax of arithmetic formulas in these theories. *Difference logic* formulas require that every atom be of the form $a - b \bowtie t$ where a and b are uninterpreted constants, $\bowtie$ is either $=$ or $\leq$, and t is an integer (i.e. either a sum of 1's or the negation of a sum of 1's). Fast algorithms for difference logic formulas have been studied in [111]. A slight variation on difference logic is UTVPI ("unit two variable per inequality") formulas which in addition to the above pattern also allow $a + b \bowtie t$. Algorithms for UTVPI formulas have been explored in [93, 136].

An obvious extension of the basic arithmetic theories discussed so far is to add multiplication. Unfortunately, this dramatically increases the complexity of the problem, and so is often avoided in practice. In fact, the integer case becomes undecidable even for conjunctions of ground formulas [99]. The real case is decidable but is doubly-exponential [54].

There are obviously many practical uses of decision procedures for arithmetic and solvers for these or closely related theories have been around for a long time.

[10] Of course, additional symbols can be included for numerals besides 0 and 1 or for $<$, $>$, and $\geq$, but these add no expressive power to the theory, so we omit them for the sake of simplicity.

[11] In Presburger arithmetic, the domain is typically taken to be the natural numbers rather than the integers, but it is straightforward to translate a formula with integer variables to one where variables are interpreted over $\mathcal{N}$ and vice-versa by adding (linearly many) additional variables or constraints.

In particular, when modeling and reasoning about systems, arithmetic is useful for modeling finite sets, program arithmetic, manipulation of pointers and memory, real-time constraints, physical properties of the environment, etc.

1.2.2.3. Arrays

Let $\Sigma_{\mathcal{A}}$ be the signature $(read, write)$. Let $\Lambda_{\mathcal{A}}$ be the following axioms:

$$\forall a\, \forall i\, \forall v\, (read(write(a,i,v),i) = v)$$
$$\forall a\, \forall i\, \forall j\, \forall v\, (i \neq j \rightarrow read(write(a,i,v),j) = read(a,j))$$

Then the theory $\mathcal{T}_{\mathcal{A}}$ of *arrays* is the set of all models of these axioms. It is common also to include the following *axiom of extensionality*:

$$\forall a\, \forall b\, ((\forall i\, (read(a,i) = read(b,i))) \rightarrow a = b$$

We will denote the resulting theory $\mathcal{T}_{\mathcal{A}ex}$. The satisfiability of ground formulas over $\mathcal{T}'_{\mathcal{A}}$ or $\mathcal{T}'_{\mathcal{A}ex}$ is NP-complete [145]. Theories of arrays are commonly used to model actual array data structures in programs. They are also often used as an abstraction for memory. The advantage of modeling memory with arrays is that the size of the model depends on the number of accesses to memory rather than the size of the memory being modeled. In many cases, this leads to significant efficiency gains.

1.2.2.4. Fixed-width bit-vectors

A natural theory for high-level reasoning about circuits and programs is a theory of bit-vectors. Various theories of bit-vectors have been proposed and studied [53, 104, 19, 24, 64, 16]. Typically, constant symbols are used to represent vectors of bits, and each constant symbol has an associated bit-width that is fixed for that symbol. The function and predicate symbols in these theories may include extraction, concatenation, bit-wise Boolean operations, and arithmetic operations. For non-trivial theories of bit-vectors, it is easy to see that the satisfiability problem is NP-complete by a simple reduction to SAT. Bit-vectors provide a more compact representation and often allow problems to be solved more efficiently than if they were represented at the bit level [43, 38, 70].

1.2.2.5. Inductive data types

An *inductive data type* (IDT) defines one or more *constructors*, and possibly also *selectors* and *testers*. A simple example is the IDT *list*, with constructors *cons* and *null*, selectors *car* and *cdr*, and testers *is_cons* and *is_null*. The *first order signature* of an IDT associates a function symbol with each constructor and selector and a predicate symbol with each tester. The standard model for such a signature is a term model built using only the constructors. For IDTs with a single constructor, a conjunction of literals is decidable in polynomial time using an algorithm by Oppen [118]. For more general IDTs, the problem is NP complete, but reasonbly efficient algorithms exist in practice [17]. IDTs are very general and can be used to model a variety of things, e.g., enumerations, records, tuples, program data types, and type systems.

$$
\begin{array}{lll}
t & ::= c & \text{where } c \in \Sigma^{\mathrm{F}} \text{ with arity } 0 \\
 & \mid \textit{int-var} & \\
 & \mid \textit{function-expr}(t_1, \ldots, t_n) & \\
 & \mid \textit{ite}(\varphi, t_1, t_2) & \\
\varphi & ::= A & \text{where } A \in \Sigma^{\mathrm{P}} \text{ with arity } 0 \\
 & \mid \textit{predicate-expr}(t_1, \ldots, t_n) & \\
 & \mid t_1 = t_2 \mid \bot \mid \top \mid \neg\varphi_1 & \\
 & \mid \varphi_1 \rightarrow \varphi_2 \mid \varphi_1 \leftrightarrow \varphi_2 & \\
 & \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 & \\
\textit{int-var} & ::= v & \text{where } v \in \mathcal{V} \\
\textit{function-expr} & ::= f & \text{where } f \in \Sigma^{\mathrm{F}} \text{ with arity } n > 0 \\
 & \mid \lambda\, \textit{int-var}, \ldots, \textit{int-var} \,.\, t & \\
\textit{predicate-expr} & ::= p & \text{where } p \in \Sigma^{\mathrm{P}} \text{ with arity } n > 0 \\
 & \mid \lambda\, \textit{int-var}, \ldots, \textit{int-var} \,.\, \varphi &
\end{array}
$$

Figure 1.2. Extended syntax with restricted lambda expressions.

1.3. Eager Encodings to SAT

The *eager approach* to SMT solving involves translating the original formula to an equisatisfiable Boolean formula in a single step. In order to generate a small SAT problem, several optimizations are performed in this translation. As one might expect, some optimizations are computationally expensive and thus there is a trade-off between the degree of optimization performed and the amount of time spent therein. In fact, the translation procedure is much like an optimizing compiler, with the "high-level program" being the original SMT problem and the "low-level object code" being the generated SAT problem. This section describes the main ideas in the eager approach and surveys the state of the art.

1.3.1. Overview

The translations used in the eager approach are, by their very nature, theory-specific. We survey here the transformations for a combination of two theories: $\mathcal{T}_{\mathcal{E}}$ and $\mathcal{T}_{\mathcal{Z}}$. These theories, plus an extension of the basic syntax to include restricted lambda expressions (see below), form the core logic of the original UCLID decision procedure [94, 131], which is the main tool implementing the eager approach. The UCLID logic has sufficed for a range of applications, from microprocessor design verification [91] to analyzing software for security vulnerabilities [131].

As mentioned, the theories of interest for this section are $\mathcal{T}_{\mathcal{E}}$ and $\mathcal{T}_{\mathcal{Z}}$, with signatures as given in §1.2.2. In particular, we assume a signature Σ containing 0, 1, $+$, $-$, $\leq$, and any number of uninterpreted functions and predicates.

Lambda expressions. For additional expressiveness, in this section we consider an extension of the core logical syntax (as given in Figure 1.1) to include lambda expressions. Assuming an infinite set $\mathcal{V}$ of *integer variables*, the extended syntax is given in Figure 1.2.

Notice that the use of lambda expressions is quite restricted. In particular, there is no way in the logic to express any form of iteration or recursion. An

integer variable x is said to be *bound* in expression E when it occurs inside a lambda expression for which x is one of the argument variables. We say that an expression is *well-formed* when it contains no unbound variables.

Satisfiability and entailment are defined for well-formed formulas just as in §1.2.1 for formulas without lambdas. Well-formed formulas containing lambdas are considered to be semantically equivalent to their beta-reduced forms (see §1.3.1.2, below), which do not contain lambdas.

Lambda notation allows us to model the effect of a sequence of *read* and *write* operations on a memory (the *select* and *update* operations on an array) without the added complexity of the theory of arrays ($\mathcal{T}_{\mathcal{A}}$). At any point of system operation, a memory can be represented by a function expression M denoting a mapping from addresses to values (for an array, the mapping is from indices to values). The initial state of the memory is given by an uninterpreted function symbol m_0 indicating an arbitrary memory state. The effect of a write operation with terms A and D denoting the address and data values yields a function expression M':

$$M' = \lambda\, addr\ .\ ITE(addr = A,\ D,\ M(addr))$$

Reading from array M at address A simply yields the function application $M(A)$.

Multi-dimensional memories or arrays are easily expressed in the same way. Moreover, lambda expressions can express *parallel-update* operations, which update multiple memory locations in a single step. The details are outside the scope of this chapter, and can be found elsewhere [131].

1.3.1.1. Operation

Suppose that we are given a formula F_{orig} in the syntax of Figure 1.2 We decide the satisfiability of F_{orig} by performing a three-stage satisfiability-preserving translation to a Boolean formula F_{bool}, and then invoking a SAT solver on F_{bool}.

The three stages of translation are as follows:

1. All lambda expressions are eliminated, resulting in a formula F_{norm}. This stage is described in §1.3.1.2.
2. Function and predicate applications of non-zero arity are eliminated to get a formula F_{arith}. This stage is described in §1.3.1.3.
3. Formula F_{arith} is a quantifier-free linear integer arithmetic ($\Sigma_{\mathcal{Z}}$-) formula. There is more than one way to translate F_{arith} to an equisatisfiable Boolean formula F_{bool}. We describe these techniques in §1.3.2–§1.3.4.

In addition to preserving satisfiability, the mapping between the eliminated symbols of a theory and the new symbols that replace them is maintained. For satisfiable problems, this facilitates model generation from a satisfying assignment generated by the SAT solver. For unsatisfiable problems, it permits the generation of higher level proof information from a Boolean proof of unsatisfiability.

1.3.1.2. Eliminating Lambdas

Recall that the syntax of lambda expressions does not permit recursion or iteration. Therefore, each lambda application in F_{orig} can be expanded by *beta-substitution*, i.e., by replacing each argument variable with the corresponding argument term. Denote the resulting formula by F_{norm}.

This step can result in an exponential blow-up in formula size. Suppose that all expressions in our logic are represented as directed acyclic graphs (DAGs) so as to share common sub-expressions. Then, the following example shows how we can get an exponential-sized DAG representation of F_{norm} starting from a linear-sized DAG representation of F_{orig}.

Example 1.3.1. Let F_{orig} be defined recursively by the following set of expressions:

$$
\begin{aligned}
F_{orig} &\doteq P(L_1(b)) \\
L_1 &\doteq \lambda\, x \,.\, f_1(L_2(x), L_2(g_1(x))) \\
L_2 &\doteq \lambda\, x \,.\, f_2(L_3(x), L_3(g_2(x))) \\
\vdots \;\; & \;\; \vdots \\
L_{n-1} &\doteq \lambda\, x \,.\, f_{n-1}(L_n(x), L_n(g_{n-1}(x))) \\
L_n &\doteq g_n
\end{aligned}
$$

Notice that the representation of F_{orig} is linear in n. Suppose we perform beta-substitution on L_1. As a result, the sub-expression $L_1(b)$ gets transformed to $f_1(L_2(b), L_2(g_1(b)))$. Next, if we expand L_2, we get four applications of L_3, viz., $L_3(b)$, $L_3(g_1(b))$, $L_3(g_2(b))$, and $L_3(g_2(g_1(b)))$. Notice that there were originally only two applications of L_3.

Continuing the elimination process, after $k-1$ elimination steps, we will get 2^{k-1} distinct applications of L_k. This can be formalized by observing that after $k-1$ steps each argument to L_k is comprised of applications of functions from a distinct subset of $\mathcal{P}(\{g_1, g_2, \ldots, g_{k-1}\})$. Thus, after all lambda elimination steps, F_{norm} will contain 2^{n-1} distinct applications of g_n, and hence is exponential in the size of F_{orig}. □

In practice, however, this exponential blow-up is rarely encountered. This is because the recursive structure in most lambda expressions, including those for memory (array) operations, tends to be linear. For example, here is the lambda expression corresponding to the result of the memory write (store) operation:

$$\lambda\, addr \,.\, ITE(addr = A,\ D,\ M(addr))$$

Notice that the "recursive" use of M occurs only in one branch of the *ITE* expression.

1.3.1.3. Eliminating Function Applications

The second step in the transformation to a Boolean formula is to eliminate applications of function and predicate symbols of non-zero arity. These applications are replaced by symbolic constants, but only after encoding enough information to maintain functional consistency (the congruence property).

There are two different techniques of eliminating function (and predicate) applications. The first is a classic method due to Ackermann [1] that involves creating sufficient instances of the congruence axiom to preserve satisfiability. The second is a technique introduced by Bryant et al. [42] that exploits the polarity

of equations and is based on the use of *ITE* expressions. We briefly review each of these methods.

Ackermann's method. We illustrate Ackermann's method using an example. Suppose that function symbol f has three occurrences: $f(a_1)$, $f(a_2)$, and $f(a_3)$. First, we generate three fresh symbolic constants xf_1, xf_2, and xf_3 to replace all instances of these applications in F_{norm}.

Then, the following set of functional consistency constraints for f is generated:

$$\left\{a_1 = a_2 \implies xf_1 = xf_2,\;\; a_1 = a_3 \implies xf_1 = xf_3,\;\; a_2 = a_3 \implies xf_2 = xf_3\right\}$$

In a similar fashion, functional consistency constraints are generated for each function and predicate symbol in F_{norm}. Denote the conjunction of all these constraints by F_{cong}. Then, F_{arith} is the formula $F_{cong} \wedge F_{norm}$.

The Bryant-German-Velev method. The function elimination method proposed by Bryant, German, and Velev exploits a property of function applications called *positive equality* [42]. (This discussion assumes that we are working with the concept of validity rather than satisfiability, but the ideas remain unchanged except for a flipping of polarities.) The general idea is to determine the polarity of each equation in the formula, i.e., whether it appears under an even (positive) or odd (negative) number of negations. Applications of uninterpreted functions can then be classified as either p-function applications, i.e., used only under positive equalities, or g-function applications, i.e., general function applications that appear under other equalities or under inequalities. The p-function applications can be encoded in propositional logic with fewer Boolean variables than the g-function applications, thus greatly simplifying the resulting SAT problem. We omit the details.

In order to exploit positive equality, Bryant et al. eliminate function applications using a nested series of *ITE* expressions. As an example, if function symbol f has three occurrences: $f(a_1)$, $f(a_2)$, and $f(a_3)$, then we would generate three new symbolic constants xf_1, xf_2, and xf_3. We would then replace all instances of $f(a_1)$ by xf_1, all instances of $f(a_2)$ by $ITE(a_2 = a_1,\, xf_1,\, xf_2)$, and all instances of $f(a_3)$ by $ITE(a_3 = a_1,\, xf_1,\, ITE(a_3 = a_2,\, xf_2,\, xf_3))$. It is easy to see that this preserves functional consistency.

Predicate applications can be removed by a similar process. In eliminating applications of some predicate p, we introduce symbolic Boolean constants $xp_1, xp_2, \ldots$. Function and predicate applications in the resulting formula F_{arith} are all of zero arity.

Lahiri et al. [92] have generalized the notion of positive equality to apply to both polarities and demonstrate that further optimization is possible, albeit at the cost of incurring a time overhead.

1.3.1.4. Summary

We conclude this section with observations on the worst-case blow-up in formula size in going from the starting formula F_{orig} to the quantifier-free arithmetic formula F_{arith}. The lambda elimination step can result in a worst-case exponential blow-up, but is typically only linear. In going from the lambda-free formula F_{norm}

to F_{arith}, the worst-case blow-up is only quadratic. Thus, if the result of lambda expansion is linear in the size of F_{orig}, F_{arith} is at most quadratic in the size of F_{orig}.

We next consider the two main classes of methods for transforming F_{arith} to an equisatisfiable Boolean formula F_{bool}: the *small-domain encoding* method and the *direct encoding* method.

1.3.2. Small-domain encodings

The formula F_{arith} is a quantifier-free $\Sigma_{\mathcal{Z}}$-formula, and so, we are concerned with $\mathcal{T}_{\mathcal{Z}}$-satisfiability of quantifier-free $\Sigma_{\mathcal{Z}}$-formulas. Recall that this problem is NP-complete (as discussed in 1.2.2). In the remainder of this section, we assume satisfiability is with respect to $\mathcal{T}_{\mathcal{Z}}$ and that all formulas are quantifier-free $\Sigma_{\mathcal{Z}}$-formulas.

A formula is constructed by combining linear constraints with Boolean operators (such as $\wedge$, $\vee$, $\neg$). Formally, the ith constraint is of the form

$$\sum_{j=1}^{n} a_{i,j} x_j \geq b_i$$

where the coefficients and the constant terms are integer constants and the variables[12] $x_1, x_2, \ldots, x_n$ are integer-valued.

If there is a satisfying solution to a formula, there is one whose size, measured in bits, is polynomially bounded in the problem size [31, 151, 84, 119]. Problem size is traditionally measured in terms of the parameters m, n, $\log a_{\max}$, and $\log b_{\max}$, where m is the total number of constraints in the formula, n is the number of variables (integer-valued symbolic constants), and $a_{\max} = \max_{(i,j)} |a_{i,j}|$ and $b_{\max} = \max_i |b_i|$ are the maximums of the absolute values of coefficients and constant terms respectively.

The above result implies that we can use an enumerative approach to deciding the satisfiability of a formula, where we restrict our search for satisfying solutions to within the bound on the problem size mentioned above. This approach is referred to as the *small-domain encoding* (SD) method.

In this method, given a formula $F_{\mathcal{Z}}$, we first compute the polynomial bound S on solution size, and then search for a satisfying solution to $F_{\mathcal{Z}}$ in the bounded space $\{0, 1, \ldots, 2^S - 1\}^n$. However, a naïve implementation of a SD-based decision procedure fails for formulas encountered in practice. The problem is that the bound on solution size, S, is $\mathcal{O}(\log m + \log b_{\max} + m[\log m + \log a_{\max}])$. In particular, the presence of the $m \log m$ term means that, for problems involving thousands of constraints and variables, as often arises in practice, the Boolean formulas generated are beyond the capacity of even the best current SAT solvers.

In this section, we describe how the small-domain encoding method can be made practical by considering special cases of formulas. Since we consider arbitrary Boolean combinations of these constraints, the decision problems are NP-complete no matter how simple the form of the linear constraint.

[12] The word "variable" is used in this section instead of "symbolic constant" as it is the more common term used in literature on solving integer linear arithmetic constraints.

1.3.2.1. Equalities

When all linear constraints are equalities (or disequalities) over integer variables, the fragment of $\mathcal{T}_{\mathcal{Z}}$ is called *equality logic*. For this fragment, we have the following "folk theorem" that is easily obtained:

Theorem 1.3.2. For an equality logic formula with n variables, $S = \log n$.

The key proof argument is that any satisfying assignment can be translated to the range $\{0, 1, 2, \ldots, n-1\}$, since we can only tell *whether* variable values differ, not by how much.

This bound yields a search space of size $\mathcal{O}(n^n)$, which can be far too large in many cases. Several optimizations are possible. The most obvious is to divide the set of variables into equivalence classes so that two variables that appear in the same equality fall into the same class. Then a separate bound on solution size can be derived for each equivalence class. This optimization is clearly applicable for arbitrary linear constraints, not just equalities.

The *range allocation* method [122] is another effective technique for reducing the size of the search space. This method operates by first building a constraint graph representing equalities and disequalities between variables in the formula, with separate types of edges representing equalities and disequalities in the negation normal form. Connected components of this graph correspond to equivalence classes mentioned above. Furthermore, the only major restriction on satisfying solutions comes from cycles that contain exactly one disequality edge. Pnueli et al. [122] give a graph-based algorithm that assigns, to each variable that participates in both equality and disequality constraints, a set of values large enough to consider all possible legal truth assignments to the constraints containing it. The resulting set of values assigned to each variable can be of cardinality smaller than n, thus often resulting in a more compact search space than n^n. However, the worst case solution size is still n, and the search space can still be $\Theta(n^n)$.

1.3.2.2. Difference Logic

Recall that *difference logic* requires every atom to be of the form $x_i - x_j \bowtie b_t$ where x_i and x_j are variables, $\bowtie$ is either $=$ or $\leq$, and b_t is an integer. Note that a constraint of the form $x_i \bowtie b_t$ can be written as $x_i - x_0 \bowtie b_t$ where x_0 is a special "variable" denoting zero. Also note that an equality can be written as a conjunction of two inequalities, while a strict inequality $<$ can be rewritten as a *le* inequality by rounding the constant term down.

We will refer to such atoms as *difference constraints*. They are also referred to in the literature as *difference-bound constraints* or *separation predicates*, and difference logic has also been termed as *separation logic*. We will use DL as an acronym for difference logic.

A fundamental construct used in the SAT-encoding of difference logic is the *constraint graph*. This graph is a weighted, directed multigraph built from the set of m difference constraints involving n variables as follows:

1. A vertex v_i is introduced for each variable x_i, including for x_0.
2. For each difference constraint of the form $x_i - x_j \geq b_t$, we add a directed edge from v_i to v_j of weight b_t.

The resulting structure has m edges and $n + 1$ vertices. It is, in general, a multigraph since there can be multiple constant (right-hand side) terms for a given left-hand side expression $x_i - x_j$.

The following theorem states the bound on solution size for difference logic.

Theorem 1.3.3. Let F_{diff} be a DL formula with n variables, excluding x_0. Let $b_{\max}$ be the maximum over the absolute values of all difference constraints in F_{diff}. Then, F_{diff} is satisfiable if and only if it has a solution in $\{0, 1, 2, \ldots, d\}^n$ where $d = n \cdot (b_{\max} + 1)$.

The proof can be obtained by an analysis of the constraint graph $\mathcal{G}$. The main insight is that any satisfying assignment for a formula with constraints represented by $\mathcal{G}$ can have a spread in values that is at most the weight of the longest path in $\mathcal{G}$. This path weight is at most $n \cdot (b_{\max} + 1)$. The bound is tight, the "+1" in the second term arising from a "rounding" of inequalities from strict to non-strict.

The above bound can also be further optimized. Equivalence classes can be computed just as before. The range allocation approach has also been extended to apply to difference logic, although the analysis involved in computing ranges can take worst-case exponential time [146]. On the plus side, the range allocation approach can, in some cases, exponentially reduce the size of the search space.

1.3.2.3. UTVPI Constraints

UTVPI constraints include difference constraints as well as *sum constraints* of the form $x_i + x_j \bowtie b_t$. UTVPI constraints over integer variables are also called *generalized 2SAT constraints*. Useful optimization problems, such as the minimum vertex cover and the maximum independent set problems, can be modeled using UTVPI constraints [80], and some applications of constraint logic programming and automated theorem proving also generate UTVPI constraints (e.g., see [83, 11]).

The bound on solution size for UTVPI constraints is only a slight increase over that for difference logic.

Theorem 1.3.4. Let F_{utvpi} be a UTVPI formula with n variables. Let $b_{\max}$ be the maximum over the absolute values of all constraints in F_{utvpi}. Then, F_{utvpi} is satisfiable if and only if it has a solution in $\{0, 1, 2, \ldots, d\}^n$ where $d = 2 \cdot n \cdot (b_{\max} + 1)$.

The theorem can be proved using results from polyhedral theory [131, 134]. A UTVPI formula can be viewed as a union of polyhedra defined by UTVPI hyperplanes. The vertices of these polyhedra are half-integral. The main step in the proof is to show that if a satisfying solution exists (i.e., there is an integer point inside some polyhedron in the union), then there is a solution that can be obtained by rounding some vertex of a polyhedron in the union. Since the above bound works for the vertices of the UTVPI polyhedra, it suffices for searching for integer solutions also.

1.3.2.4. Sparse, Mostly-Difference Constraints

The most general case is when no restricting assumptions can be made on the structure of linear constraints. In this case, we can still improve the "typical-case" complexity of SAT-encoding by exploiting the structure of constraints that appear in practical problems of interest.

It has been observed [131, 123, 59] that formulas arising in software verification have:

1. *Mainly Difference Constraints:* Of the m constraints, $m - k$ are *difference* constraints, where $k \ll m$.
2. *Sparse Structure:* The k non-difference constraints are sparse, with at most w variables per constraint, where w is "small". The parameter w is termed the *width* of the constraint.

Seshia and Bryant [132] exploited above special structure to obtain a bound on solution size that is parameterized in terms of k and w in addition to m, n, $a_{\max}$ and $b_{\max}$. Their main result is stated in the following theorem.

Theorem 1.3.5. Let $F_{\mathcal{Z}}$ be a quantifier-free $\Sigma_{\mathcal{Z}}$-formula. If $F_{\mathcal{Z}}$ is satisfiable, there is a solution to $F_{\mathcal{Z}}$ whose l_∞ norm is bounded by $d = (n+2)\Delta$, where $\Delta = s\,(b_{\max}+1)\,(a_{\max}\,w)^k$ and $s = \min(n+1, m)$.

The proof of the above theorem is based on a theorem given by Borosh, Treybig, and Flahive [30] bounding integer solutions of linear systems of the form $A\mathbf{x} \geq \mathbf{b}$, where $\mathbf{x}$ is of length n. Their result is as follows. Consider the augmented matrix $[A|\mathbf{b}]$. Let Δ be the maximum of the absolute values of all minors of this augmented matrix. Then, the linear system has a satisfying solution if and only if it has one with all entries bounded by $(n+2)\Delta$. Seshia and Bryant used the special structure of sparse, mostly-difference constraints to obtain a bound on the value of Δ.

Several optimizations are possible to reduce the size of the bound given in Theorem 1.3.5, including computing equivalence classes of variables, rewriting constraints to reduce the size and number of non-zero coefficients, a "shift-of-origin" transformation to deal with large constant terms, etc. For brevity, these are omitted here and can be found elsewhere [131].

1.3.2.5. Summary

Table 1.1 summarizes the value of d for all the classes of linear constraints explored in this section. We can clearly see that the solution bound for arbitrary formulas is conservative. For example, if all constraints are difference constraints, the expression for d simplifies to $(n+2)\cdot\min(n+1,m)\cdot(b_{\max}+1)$. This is $n+2$ times as big as the bound obtainable for difference logic in isolation; however, the slack in the bound is a carry-over from the result of Borosh, Treybig, and Flahive [30]. For UTVPI constraints too, the bound derived for arbitrary formulas is much looser. In the worst case, it is looser by an exponential factor: if k is $\mathcal{O}(m)$, $a_{\max}$ is 1, and w is 2, then the bound is $\mathcal{O}((n+2)\cdot\min(n+1,m)\cdot(b_{\max}+1)\cdot 2^m)$, whereas the results of §1.3.2.3 tell us that the solution bound $d = 2\cdot n\cdot(b_{\max}+1)$ suffices.

Table 1.1. Solution bounds for classes of linear constraints. The classes are listed top to bottom in increasing order of expressiveness.

Class of Linear Constraints	Solution Bound d
Equality constraints	n
Difference constraints	$n \cdot (b_{\max} + 1)$
UTVPI constraints	$2 \cdot n \cdot (b_{\max} + 1)$
Arbitrary linear constraints	$(n+2) \cdot \min(n+1, m) \cdot (b_{\max} + 1) \cdot (w \cdot a_{\max})^k$

1.3.3. Direct encoding of theory axioms

A decision procedure based on the direct encoding method operates in three steps:

1. Replace each unique constraint in the linear arithmetic formula F_{arith} with a fresh Boolean variable to get a Boolean formula F_{bvar}.
2. Generate a Boolean formula F_{cons} that constrains values of the introduced Boolean variables so as to preserve the arithmetic information in F_{arith}.
3. Invoke a SAT solver on the Boolean formula $F_{bvar} \wedge F_{cons}$.

The direct encoding approach has also been termed as *per-constraint encoding*.

For integer linear arithmetic, the formula F_{cons} expresses so-called *transitivity constraints*. For equality logic, Bryant and Velev [44] showed how to generate transitivity constraints efficiently so that the size of F_{cons} is in the worst-case only cubic in the size of the number of equalities in F_{arith}. Their transitivity constraint generation algorithm operates on the constraint graph (as introduced in §1.3.2.1). It avoids enumeration of cycles by introducing *chordal edges*, thereby avoiding an exponential blow-up in the number of transitivity constraints.

Strichman et al. [143] generalized the above graph-based approach to difference logic. In this case, the constraint graph is just as in §1.3.2.2. Again, cycle enumeration is avoided by the introduction of new edges. However, in the case of difference logic, the number of added edges can be exponential in the original constraint graph, in the worst case [131]. Even so, in many practical instances, the number of transitivity constraints is small and the resulting SAT problem is easily solved. Various heuristic optimizations are possible based on the Boolean structure of the formula [142].

Strichman [141] also extended the above scheme to operate on an arbitrary linear arithmetic formula (over the integers or the rationals). The "transitivity constraints" are generated using the Fourier-Motzkin variable elimination procedure (the Omega test variant [125], in the case of integers). It is well-known that Fourier-Motzkin can generate doubly-exponentially many new constraints in the worst case [45]. Thus, the worst-case size of F_{cons} is doubly-exponential in the size of F_{arith}. This worst-case behavior does seem to occur in practice, as evidenced by the results in Strichman's paper [141] and subsequent work.

We summarize the complexity of the direct encoding method for the three classes of linear arithmetic formulas in Table 1.2.

Table 1.2. Worst-case size of direct encoding. The classes are listed top to bottom in increasing order of expressiveness.

Class of Linear Constraints	Worst-case size of F_{cons}
Equality constraints	Cubic
Difference constraints	Exponential
Arbitrary linear constraints	Doubly exponential

1.3.4. Hybrid eager approaches

In §1.3.2 and §1.3.3, we have introduced two very distinct methods of deciding a linear arithmetic formula via translation to SAT. This naturally gives rise to the following question: Given a formula, which encoding technique should one use to decide *that* formula the fastest? This question is an instance of the automated algorithm selection problem.

On first glance, it might seem that the small-domain encoding would be best, since it avoids the potential exponential or doubly-exponential blowup in SAT problem size that the direct encoding can suffer in the worst case. However, this blowup is not always a problem because of the special structure of the generated SAT instance. The form of a transitivity constraint is $b_1 \wedge b_2 \implies b_3$ where b_1, b_2, b_3 are Boolean variables encoding linear constraints. If the polarities of these variables are chosen appropriately, the resulting constraint is either a Horn constraint or can be transformed into one by variable renaming. Thus, the overall SAT encoding is a "mostly-HornSAT" problem: i.e., the vast majority of clauses are Horn clauses. It has been observed for difference logic that the generated SAT problems are solved quickly in practice in spite of their large size [131].

The question on the choice of encoding has been studied in the context of difference logic [131]. It has been found that this question cannot be resolved entirely in favor of either method. One can select an encoding method based on formula characteristics using a rule generated by machine learning from past examples (formulas) [133, 131]. Moreover, parts of a single formula corresponding to different variable classes can be encoded using different encoding methods. The resulting hybrid encoding algorithm has been empirically shown to be more robust to variation in formula characteristics than either of the two techniques in isolation [131]. This hybrid algorithm is the first successful use of automated algorithm selection based on machine learning in SMT solvers.

1.4. Integrating Theory Solvers into SAT Engines

The alternative to the eager approach, as described above, is the *lazy* approach in which efficient SAT solvers are integrated with decision procedures for first-order theories (also called *Theory Solvers* or $\mathcal{T}$*-solvers*) [6, 153, 4, 57, 8, 20, 72, 111, 155]). Systems based on this approach are called *lazy SMT solvers*, and include ArgoLib [98], Ario [136], Barcelogic [110], CVC3 [18], Fx7 [105], ICS [66], MathSAT [34], Simplify [59], TSAT++ [5], Verifun [67], Yices [62], and Z3 [56].

1.4.1. Theory Solvers and their desirable features

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

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

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

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

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

Deduction of unassigned literals: when the $\mathcal{T}$*-solver* is invoked on a $\mathcal{T}$-consistent set μ, it can also perform deductions of the form $\eta \models_{\mathcal{T}} l$, where $\eta \subseteq \mu$ and l is a literal on a not-yet-assigned atom in φ.[13]

Deduction of interface equalities: when returning Sat, the $\mathcal{T}$*-solver* can also perform deductions of the form $\mu \models_{\mathcal{T}} e$ (if $\mathcal{T}$ is convex) or $\mu \models_{\mathcal{T}} \bigvee_j e_j$ (if $\mathcal{T}$ is not convex) where e, $e_1, ..., e_n$ are equalities between variables or terms occurring in atoms in μ. We call such equalities *interface equalities* and denote the interface equality $(v_i = v_j)$ by e_{ij}. Deductions of interface equalities are also called e_{ij}*-deductions*. Notice that here the deduced equalities need not occur in the input formula φ.

$\mathcal{T}$*-solvers* will be discussed in more detail in §1.5.

1.4.2. A generalized DPLL schema

Different variants of lazy SMT procedures have been presented. Here we consider variants in which the Boolean abstraction φ^p of the input formula φ is fed into a DPLL-based SAT solver (henceforth referred to as a DPLL solver). Note that since most DPLL solvers accept only CNF, if φ^p has arbitrary Boolean structure, it is first converted into an equisatisfiable CNF formula using standard techniques (see §**??**).

In its simplest integration schema [20, 57], called "*offline*" [67], [14] the Boolean abstraction φ^p of the input formula is fed to a DPLL 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, the set of literals μ corresponding to μ^p is given as input to the $\mathcal{T}$*-solver*. If μ is found to be $\mathcal{T}$-consistent, then φ is $\mathcal{T}$-consistent.

[13] 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 [4]. We call this technique *plunging* [59]. In practice, plunging is very inefficient.

[14] The offline approach is also called the "lemmas on demand" approach in [57].

```
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 1.3. An online schema for $\mathcal{T}$-DPLL based on modern DPLL.

If not, $\neg\mu^p$ is added as a clause to φ^p, and the SAT solver is restarted from scratch on the resulting formula. Notice that here DPLL is used as a black-box.

In a more sophisticated schema [6, 79, 4, 153, 8, 67, 72, 37], called "*online*" [67], DPLL is modified to work directly as an enumerator of truth assignments, whose $\mathcal{T}$-satisfiability is checked by a $\mathcal{T}$*-solver*. This schema evolved from that of the DPLL-based procedures for modal logics (see §**??** and §**??**). Figure 1.3 shows an online $\mathcal{T}$-DPLL procedure based on a modern DPLL engine [157, 158]. The inputs 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.[15]

$\mathcal{T}$-`preprocess` simplifies φ into a simpler formula, and updates μ if necessary, so as 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 of the Boolean preprocessing steps of DPLL with some theory-dependent rewriting steps on the $\mathcal{T}$-literals of φ. (The latter are described in §1.4.3.1. and §1.4.3.2.)

$\mathcal{T}$-`decide_next_branch` selects the next literal to split on as in standard DPLL (but it may also take into consideration the semantics in $\mathcal{T}$ of the literals being selected.)

$\mathcal{T}$-`deduce`, in its simplest version, behaves similarly to standard BCP in DPLL: it iteratively deduces Boolean literals l^p implied by the current assignment (i.e., s.t. $\varphi^p \wedge \mu^p \models_p l^p$, "$\models_p$" being propositional entailment) and updates φ^p and μ^p accordingly, until one of the following conditions occur:

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

[15] 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.

`deduce` in DPLL, returning Conflict.

(ii) μ^p propositionally satisfies φ^p ($\mu^p \models_p \varphi^p$). If so, $\mathcal{T}$`-deduce` invokes the $\mathcal{T}$*-solver* on μ: if the latter returns Sat, then $\mathcal{T}$`-deduce` returns Sat; otherwise, $\mathcal{T}$`-deduce` returns Conflict.

(iii) no more literals can be deduced. If so, $\mathcal{T}$`-deduce` returns Unknown. A slightly more elaborate version of $\mathcal{T}$`-deduce` can invoke the $\mathcal{T}$*-solver* on μ also at this intermediate stage: if the $\mathcal{T}$*-solver* returns Unsat, then $\mathcal{T}$`-deduce` returns Conflict. (This enhancement, called *early pruning*, is discussed in §1.4.3.3.)

A much more elaborate version of $\mathcal{T}$`-deduce` can be implemented if the $\mathcal{T}$*-solver* is able to perform deductions of unassigned literals. (This enhancement, called $\mathcal{T}$*-propagation*, is discussed in §1.4.3.4.)

$\mathcal{T}$`-analyze_conflict` is an extension of `analyze_conflict` of DPLL [157, 158]: 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` [157, 158]; if instead the conflict is caused by a $\mathcal{T}$-inconsistency revealed by $\mathcal{T}$*-solver* (case (ii) or (iii) above), then the result of $\mathcal{T}$`-analyze_conflict` is the Boolean abstraction η^p of the theory conflict set $\eta \subseteq \mu$ produced by the $\mathcal{T}$*-solver*, or a mixed Boolean+theory conflict set computed by a backward-traversal of the implication graph starting from the conflicting clause $\neg\eta^p$ (see §1.4.3.5). If the $\mathcal{T}$*-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, either temporarily or permanently, and backtracks up to `blevel`. (These features, called $\mathcal{T}$*-backjumping* and $\mathcal{T}$*-learning*, are discussed in §1.4.3.5.)

$\mathcal{T}$-DPLL differs from the DPLL schema of [157, 158] 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* ($\mu \models_\mathcal{T} l$);
- an extended notion of *conflict*: not only *Boolean conflicts* ($\mu^p \wedge \varphi^p \models_p \bot$), but also *theory conflicts* ($\mu \models_\mathcal{T} \bot$), or even *mixed Boolean+theory conflicts* ($(\mu \wedge \varphi) \models_\mathcal{T} \bot$). See §1.4.3.5.

Example 1.4.1. Consider the formulas φ and φ^p shown in Figure 1.4. Suppose $\mathcal{T}$`-decide_next_branch` selects, in order, $\mu^p := \{\neg B_5, B_8, B_6, \neg B_1\}$ (in c_4, c_7, c_6, and c_1). $\mathcal{T}$`-deduce` cannot unit-propagate any literal. Assuming the enhanced version of step (iii), it invokes the $\mathcal{T}$*-solver* on $\mu := \{\neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2 - x_4 \leq 6), \neg(2x_2 - x_3 > 2)\}$. Suppose the enhanced $\mathcal{T}$-solver not only returns Sat, but also 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 then added to μ^p and propagated ($\mathcal{T}$-propagation). As a result, A_1, A_2 and B_2 are unit-propagated from c_5, c_3 and c_2.

Let μ'^p be the resulting assignment $\{\neg B_5, B_8, B_6, \neg B_1, \neg B_3, A_1, A_2, B_2\}$). By step (iii), $\mathcal{T}$`-deduce` invokes the $\mathcal{T}$*-solver* on μ': $\{\neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2 - x_4 \leq 6), \neg(2x_2 - x_3 > 2), \neg(3x_1 - 2x_2 \leq 3), (x_1 - x_5 \leq 1)\}$ which is inconsistent because of the 1st, 2nd, and 6th literals. As a result, the $\mathcal{T}$*-solver*

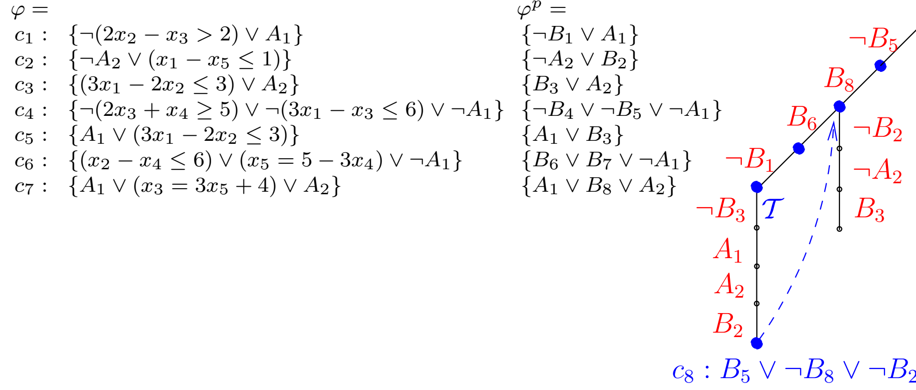

Figure 1.4. Boolean search (sub)tree in the scenario of Example 1.4.1. (A diagonal line, a vertical line and a vertical line tagged with "$\mathcal{T}$" denote literal selection, unit propagation and $\mathcal{T}$-propagation respectively; a bullet "•" denotes a call to the $\mathcal{T}$-*solver*.)

returns Unsat, and hence `T-deduce` returns Conflict. Next, `T-analyze_conflict` and `T-backtrack` learn the corresponding Boolean conflict clause

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

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

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

1.4.3. Enhancements to the schema

We describe some of the most effective techniques which have been proposed in order to optimize the interaction between the DPLL solver and the $\mathcal{T}$-*solver*. (We refer the reader to [130] for a much more extensive and detailed survey.) Some of them derive from those developed in the context of DPLL-based procedures for modal logics (see §**??**).

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

As discussed in §**??**, in order to avoid the generation of many trivially-unsatisfiable assignments, it is wise to preprocess $\mathcal{T}$-atoms so as 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*: $(x_1 + x_2 - x_3 \leq 1), (x_2 + x_1 - 1 \leq x_3) \Longrightarrow (x_1 + x_2 - x_3 \leq 1))$.
- *Exploit $\mathcal{T}$-specific properties*: $(x_1 \leq 3), (x_1 < 4) \Longrightarrow (x_1 \leq 3)$ if x_1 represents an integer.

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

1.4.3.2. Static learning

On some specific kinds of problems, it is possible to quickly detect *a priori* small and "obviously $\mathcal{T}$-inconsistent" sets of $\mathcal{T}$-atoms in φ (typically pairs or triplets). Some examples are:

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

If so, the clauses obtained by negating the literals in such sets (e.g., $\neg(x = 0) \vee \neg(x = 1))$ can be added to the formula before the search starts. Then, whenever all but one of the literals in the set are assigned to true, the negation of the remaining literal is assigned deterministically by unit propagation. This prevents the solver from generating any assignment which include the inconsistent set. This technique may significantly reduce the Boolean search space, and hence the number of calls to the $\mathcal{T}$*-solver*, producing significant speed-ups [4, 9, 34, 155].

Intuitively, one can think of static learning as suggesting 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 [143, 133] (see §1.3), 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. Notice also that these clauses are not needed for correctness or completeness: rather, they are used only for pruning the Boolean search space.

1.4.3.3. Early pruning

Another optimization, here generically called *early pruning – EP*,[16] is to introduce intermediate calls to the $\mathcal{T}$*-solver* while an assignment μ is still under construction (in the $\mathcal{T}$-DPLL scheme of §1.4.2, this corresponds to the "slightly more elaborate version" of step (iii) of $\mathcal{T}$`-deduce`). If $\mathcal{T}$*-solver*(μ) returns `Unsat`, then all possible extensions of μ are unsatisfiable, so $\mathcal{T}$-DPLL can immediately return `Unsat` and backtrack, possibly avoiding a very large amount of useless search.

In general, EP may dramatically reduce the Boolean search space, and hence of the number of calls to the $\mathcal{T}$*-solver*. Unfortunately, as EP may cause useless

[16] Also called *intermediate assignment checking* in [79] and *eager notification* in [20].

calls to the $\mathcal{T}$-*solver*, the benefits of the pruning effect may be partly counterbalanced by the overhead introduced by the extra calls. Many different improvements to EP and strategies for interleaving calls to the $\mathcal{T}$-*solver* with Boolean reasoning steps have been proposed [153, 144, 72, 8, 5, 111, 35, 49].

1.4.3.4. $\mathcal{T}$-propagation

As discussed in §1.4.1, for some theories it is possible to implement the $\mathcal{T}$-*solver* so that a call to $\mathcal{T}$-*solver*(μ) returning Sat can also perform one or more deduction(s) of the form $\eta \models_{\mathcal{T}} l$, where $\eta \subseteq \mu$ and l is a literal on an unassigned atom in φ. If this is the case, then the $\mathcal{T}$-*solver* can return l to $\mathcal{T}$-DPLL, so that l^p is added to μ^p and unit-propagated [4, 8, 72, 111]. This process, which is called $\mathcal{T}$-propagation,[17] may result in new literals being assigned, leading to new calls to the $\mathcal{T}$-*solver*, followed by additional new assignments being deduced, and so on, so that together, $\mathcal{T}$-propagation and unit propagation may produce a much larger benefit than either of them alone. As with early-pruning, there are different strategies by which $\mathcal{T}$-propagation can be interleaved with unit-propagation [4, 8, 72, 34, 111, 49, 114].

Notice that when the $\mathcal{T}$-*solver* deduces $\eta \models_{\mathcal{T}} l$, it can return this deduction to $\mathcal{T}$-DPLL, which can then add the $\mathcal{T}$-deduction clause $(\eta^p \rightarrow l^p)$ to φ^p, either temporarily or permanently. The $\mathcal{T}$-deduction clause can be used during the rest of the search, with benefits analogous to those of $\mathcal{T}$-learning (see §1.4.3.5).

1.4.3.5. $\mathcal{T}$-backjumping and $\mathcal{T}$-learning

As hinted in §1.4.2, we assume that, when the $\mathcal{T}$-*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 (see §1.4.1). If so, $\mathcal{T}$-DPLL can use η^p as if it were a Boolean conflict set to drive the backjumping and learning mechanism of DPLL: the conflict clause $\neg\eta^p$ is added to φ^p either temporarily or permanently ($\mathcal{T}$-learning) and the procedure backtracks to the branching point suggested by η^p ($\mathcal{T}$-backjumping) [81, 120, 153, 57, 144, 8, 72, 34]. Modern implementations inherit the backjumping mechanism of current DPLL tools: $\mathcal{T}$-DPLL learns the conflict clause $\neg\eta^p$ and backtracks to the highest point in the stack where one $l^p \in \eta^p$ is not assigned, and unit propagates $\neg l^p$ on $\neg\eta^p$. Intuitively, DPLL backtracks to the highest point where it would have done something different if it had known in advance the conflict clause $\neg\eta^p$ from the $\mathcal{T}$-*solver*.

As hinted in §1.4.2, it is possible to use either a theory conflict set η (i.e., $\neg\eta$ is a $\mathcal{T}$-valid clause) or a *mixed Boolean+theory conflict set* η', i.e., a set η' s.t. an inconsistency can be derived from $\eta' \wedge \varphi$ by means of a combination of Boolean and theory reasoning ($\eta' \wedge \varphi \models_{\mathcal{T}} \perp$). Such conflict sets/clauses can be obtained starting from the theory conflict clause $\neg\eta^p$ by backward-traversal of the implication graph, until one of the standard conditions (e.g., 1UIP) is achieved. Notice that it is possible to learn *both* clauses $\neg\eta$ and $\neg\eta'$.

Example 1.4.2. The scenario depicted in Example 1.4.1 represents a form of $\mathcal{T}$-backjumping and $\mathcal{T}$-learning, in which the conflict clause c_8 is a $\mathcal{T}_{\mathcal{R}}$-conflict

[17] Also called *forward reasoning* in [4], *enhanced early pruning* in [8], *theory propagation* in [113, 111], and *theory-driven deduction* or $\mathcal{T}$-*deduction* in [34].

clause (i.e., $\mathcal{B}2\mathcal{T}(c_8)$ is $\mathcal{T}_{\mathcal{R}}$-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. This is done by starting with c_8 and performing resolution with each of the clauses that triggered the assignments leading to the conflict, but in the reverse order that the assignments occurred (in this case, clauses c_2 and c_3, the antecedent clauses of B_2 and A_2 respectively, and the $\mathcal{T}$-deduction clause c_9 which "caused" the propagation of $\neg B_3$):

$$\dfrac{\dfrac{\dfrac{\overbrace{B_5 \vee \neg B_8 \vee \neg B_2}^{c_8:\ \mathit{theory\ conflicting\ clause}} \qquad \overbrace{\neg A_2 \vee B_2}^{c_2}}{B_5 \vee \neg B_8 \vee \neg A_2} \qquad \overbrace{B_3 \vee A_2}^{c_3}}{B_5 \vee \neg B_8 \vee B_3} \qquad \overbrace{B_5 \vee B_1 \vee \neg B_3}^{c_9}}{\underbrace{B_5 \vee \neg B_8 \vee B_1}_{c_8':\ \mathit{mixed\ Boolean+theory\ conflict\ clause}}}$$

The result is 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 $\mathcal{T}_{\mathcal{R}}$-valid.) If c_8' is chosen, then $\mathcal{T}$-`backtrack` pops from μ^p all literals up to $\{\neg B_5, B_8\}$, and then unit-propagates B_1 on c_8', and hence A_1 on c_1.

As with static learning, the clauses added by $\mathcal{T}$-learning refer only to atoms which already occur in the original formula, so that no new atom is added. [67] proposed an interesting generalization of $\mathcal{T}$-learning, in which learned clause may contain also new atoms. [36, 37] used a similar idea to improve the efficiency of Delayed Theory Combination (see §1.6.3). [152] proposed similar ideas for a SMT tool for difference logic, in which new atoms can be generated selectively according to an ad-hoc heuristic.

1.4.3.6. Generating partial assignments

Due to the two-watched-literal scheme [106], in modern implementations, DPLL returns Sat only when all variables are assigned truth values, thus returning *total* assignments. 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 propositionally satisfies φ, so the overhead introduced is negligible. In SMT, however, 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.

In order to overcome these problems, it is sufficient to implement some device monitoring the satisfaction of all original clauses in φ. Although this may cause some overhead in handling the Boolean component of reasoning, it may also reduce the overall Boolean search space and hence the number of subsequent calls to the $\mathcal{T}$-*solver*.

1.4.3.7. Pure-literal filtering

If we have non-Boolean $\mathcal{T}$-atoms occurring only positively [resp. negatively] in the input formula, we can safely drop every negative [resp. positive] occurrence of them from an assignment μ whenever μ is to be checked by the $\mathcal{T}$*-solver* [153, 78, 8, 35, 130].[18] We call this technique *pure-literal filtering*.[19]

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

Moreover, this technique is particularly useful in some situations. For instance, many $\mathcal{T}$*-solvers* for $\mathcal{T}_{\mathcal{Z}}$ and its difference logic fragment cannot efficiently handle disequalities, so that they are forced to split them into a disjunction of strict inequalities. For example, the disequality $(x_1 - x_2 \neq 3)$ would be replaced by $(x_1-x_2 > 3)\vee(x_1-x_2 < 3)$. This causes an enlargement of the search, because the two disjuncts must be investigated separately. In many problems, however, it is common for most equalities to $(t_1 = t_2)$ occur with positive polarity only. For such equalities, pure-literal filtering avoids adding $(t_1 \neq t_2)$ to μ when $(t_1 = t_2)^p$ is assigned to false by $\mathcal{T}$-DPLL, so that no split is needed [8].

1.4.4. An abstract framework

The DPLL procedure and its variants and extensions, including $\mathcal{T}$-DPLL, can also be described more abstractly as transition systems. This allows one to ignore unimportant control and implementation details and provide the essence of each variant in terms of a set of state transition rules and a rule application strategy.

Following the *Abstract DPLL Modulo Theories* framework first introduced in [113], the variants of $\mathcal{T}$-DPLL discussed in the previous subsection can be described abstractly as a transition relation over states of the form Fail or $\mu \parallel \varphi$, where φ is a (ground) CNF formula, or, equivalently, a finite set of clauses, and μ is a *sequence* of (ground) literals, each marked as a *decision* or a non-decision literal. As in §1.4.1, the set μ represents a partial assignment of truth values to the atoms of φ. The transition relation is specified by a set of *transition rules*, given below. In the rules, we denote the concatenation of sequences of literals by simple juxtaposition (e.g., $\mu\ \mu'\ \mu''$), treating single literals as one element sequences and denoting the empty sequence with $\emptyset$. To denote that a literal l is annotated as a decision literal in a sequence we write it as $l^\bullet$. A literal l is *undefined in* μ if neither l nor $\neg l$ occurs in μ. We write $S \Longrightarrow S'$ as usual to mean that two states S and S' are related by the transition relation $\Longrightarrow$ and say that there is a *transition* from S to S'. We call any sequence of transitions of the form $S_0 \Longrightarrow S_1 \Longrightarrow S_2 \Longrightarrow \ldots$ a *derivation*.

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

[19] Also called *triggering* in [153, 8].

Definition 1.4.3 (Transition Rules)**.** The following is a set of rules for *Abstract* $\mathcal{T}$-DPLL. Except for Decide, all the rules that introduce new literals annotate them as non-decision literals.

$$\textsf{Propagate}: \quad \mu \parallel \varphi, c \vee l \implies \mu\, l \parallel \varphi, c \vee l \;\; \textbf{if} \begin{cases} \mu \models_p \neg c \\ l \text{ is undefined in } \mu \end{cases}$$

$$\textsf{Decide}: \quad \mu \parallel \varphi \implies \mu\, l^{\bullet} \parallel \varphi \;\; \textbf{if} \begin{cases} l \text{ or } \neg l \text{ occurs in } \varphi \\ l \text{ is undefined in } \mu \end{cases}$$

$$\textsf{Fail}: \quad \mu \parallel \varphi, c \implies \mathit{Fail} \;\; \textbf{if} \begin{cases} \mu \models_p \neg c \\ \mu \text{ contains no decision literals} \end{cases}$$

$$\textsf{Restart}: \quad \mu \parallel \varphi \implies \emptyset \parallel \varphi$$

$$\mathcal{T}\textsf{-Propagate}: \quad \mu \parallel \varphi \implies \mu\, l \parallel \varphi \;\; \textbf{if} \begin{cases} \mu \models_{\mathcal{T}} l \\ l \text{ or } \neg l \text{ occurs in } \varphi \\ l \text{ is undefined in } \mu \end{cases}$$

$$\mathcal{T}\textsf{-Learn}: \quad \mu \parallel \varphi \implies \mu \parallel \varphi, c \;\; \textbf{if} \begin{cases} \text{each atom of } c \text{ occurs in } \mu \parallel \varphi \\ \varphi \models_{\mathcal{T}} c \end{cases}$$

$$\mathcal{T}\textsf{-Forget}: \quad \mu \parallel \varphi, c \implies \mu \parallel \varphi \;\; \textbf{if} \; \{ \varphi \models_{\mathcal{T}} c$$

$\mathcal{T}$-Backjump :

$$\mu\, l^{\bullet}\, \mu' \parallel \varphi, c \implies \mu\, k \parallel \varphi, c \;\; \textbf{if} \begin{cases} \mu\, l^{\bullet}\, \mu' \models_p \neg c, \text{ and there is} \\ \text{some clause } c' \vee l' \text{ such that:} \\ \quad \varphi, c \models_{\mathcal{T}} c' \vee l' \text{ and } \mu \models_p \neg c', \\ \quad l' \text{ is undefined in } \mu, \text{ and} \\ \quad l \text{ or } \neg l \text{ occurs in } \mu\, l^{\bullet}\, \mu' \parallel \varphi \end{cases}$$

The clauses c and $c' \vee l'$ in the $\mathcal{T}$-Backjump rule are respectively the *conflicting* clause and the *backjump* clause of the rule.

The rules Propagate, Decide, Fail and Restart, operate at the propositional level. The other rules involve the theory $\mathcal{T}$ and have rather general preconditions. While all of these preconditions are decidable whenever the $\mathcal{T}$-satisfiability of sets of ground literals is decidable, they might be expensive to check in their full generality.[20] However, there exist restricted applications of these rules that are both efficient and enough for completeness [114]. Given a ground CNF formula φ, the purpose of the rules above is to extend and modify an originally empty sequence until it determines a total, satisfying assignment for φ or the Fail rule becomes applicable.

Example 1.4.4. The computation discussed in Example 1.4.2 can be described by the following derivation in Abstract $\mathcal{T}$-DPLL, where φ is again the formula consisting of the $\mathcal{T}_{\mathcal{R}}$-clauses $c_1, \ldots, c_7$ in Figure 1.4 and c_8 is the clause abstracted by $B_5 \vee \neg B_8 \vee \neg B_2$. For space constraints, instead of φ's literals we use their

[20] In particular, the precondition of $\mathcal{T}$-Backjump seems problematic on a first look because it relies on the computability of the backjump clause $c' \vee l'$.

propositional abstraction here, and use $\Longrightarrow^+$ to denote multiple transitions.

1–4. $\emptyset \parallel \varphi$	$\Longrightarrow^+$	$\neg B_5{}^\bullet\, B_8{}^\bullet\, B_6{}^\bullet\, \neg B_1{}^\bullet \parallel \varphi$	(by Decide)
5.	$\Longrightarrow$	$\neg B_5{}^\bullet\, B_8{}^\bullet\, B_6{}^\bullet\, \neg B_1{}^\bullet\, \neg B_3 \parallel \varphi$	(by $\mathcal{T}$-Propagate)
6–8.	$\Longrightarrow^+$	$\neg B_5{}^\bullet\, B_8{}^\bullet\, B_6{}^\bullet\, \neg B_1{}^\bullet\, \neg B_3\, A_1\, A_2\, B_2 \parallel \varphi$	(by Propagate)
9.	$\Longrightarrow$	$\neg B_5{}^\bullet\, B_8{}^\bullet\, B_6{}^\bullet\, \neg B_1{}^\bullet\, \neg B_3\, A_1\, A_2\, B_2 \parallel \varphi,\, c_8$	(by $\mathcal{T}$-Learn)
10.	$\Longrightarrow$	$\neg B_5{}^\bullet\, B_8{}^\bullet\, B_1 \parallel \varphi,\, c_8$	(by $\mathcal{T}$-Backjump)
11.	$\Longrightarrow$	$\neg B_5{}^\bullet\, B_8{}^\bullet\, B_1\, A_1 \parallel \varphi,\, c_8$	(by Propagate)

Recall that clause c_8 in Step 9 is a theory lemma added in response to the $\mathcal{T}$-inconsistency of the assignment produced by Step 8. In step 10, $\mathcal{T}$-Backjump is applied with conflicting clause $c = c_8$ and backjump clause $c' \vee l' = B_5 \vee \neg B_8 \vee B_1$ with $l' = B_1$. The backjump clause is derived as explained in Example 1.4.2.

Let us say that a state is *$\mathcal{T}$-compatible* if it is *Fail* or has the form $\mu \parallel \varphi$ where μ is $\mathcal{T}$-consistent. The transition rules can be used to decide the $\mathcal{T}$-satisfiability of an input formula φ_0 by generating a derivation

$$\emptyset \parallel \varphi_0 \Longrightarrow_{\mathrm{B}} S_1 \Longrightarrow_{\mathrm{B}} \cdots \Longrightarrow_{\mathrm{B}} S_n, \tag{1.1}$$

where S_n is a $\mathcal{T}$-compatible state to which none of the rules Propagate, Decide, Fail, or $\mathcal{T}$-Backjump applies. We call such a derivation *exhausted*.

To generate from $\emptyset \parallel \varphi_0$ only derivations like (1.1) above it is enough to impose a few, rather weak, restrictions on a rule application strategy. Roughly speaking, these restrictions rule out only derivations with subderivations consisting exclusively of $\mathcal{T}$-Learn and $\mathcal{T}$-Forget steps, and derivations that do not apply Restart with increased periodicity.[21]

A rule application strategy is *fair* it it conforms to these restrictions and stops (extending) a derivation only when the derivation is exhausted. Every fair strategy is *terminating*, *sound*, and *complete* in the following sense (see [114] for details).

Termination: Starting from a state $\emptyset \parallel \varphi_0$, the strategy generates only finite derivations.

Soundness: If φ_0 is $\mathcal{T}$-satisfiable, every exhausted derivation of $\emptyset \parallel \varphi_0$ generated by the strategy ends with a state of the form $\mu \parallel \varphi$ where μ is a ($\mathcal{T}$-consistent) total, satisfying assignment for φ.

Completeness: If φ_0 is not $\mathcal{T}$-satisfiable, every exhausted derivation of $\emptyset \parallel \varphi_0$ generated by the strategy ends with *Fail*.

In the setting above, fair strategies stop a derivation for a $\mathcal{T}$-satisfiable φ_0 only once they compute a *total* assignment for φ_0's atoms. A more general setting can be obtained, with similar results, by defining a finite derivation to be exhausted if its last state is $\mathcal{T}$-compatible and, when the state is $\mu \parallel \varphi$, the assignment μ propositionally satisfies φ. We refer back to the discussion in §1.4.3.6 for why this can be more convenient computationally.

The rule set in Definition 1.4.3 is not minimal. For instance, there are fair strategies that use only $\mathcal{T}$-Propagate or only $\mathcal{T}$-Learn in addition to Decide, Fail,

[21] In fact, the actual restrictions are even weaker. See Theorem 3.7 of [114] for details.

Propagate and $\mathcal{T}$-Backjump. The rule set is not exhaustive either because it models only the most common (and experimentally most useful) operations found in lazy SMT solvers. We explain how below, focusing on the less trivial rules.

Decide. This rule represents a case split by adding an undefined literal of φ, or its negation, to μ. The added literal l is annotated as a *decision literal*, to denote that if $\mu\, l$ cannot be extended to a model of φ then an alternative extension of μ must be considered—something done with the $\mathcal{T}$-Backjump rule.[22]

$\mathcal{T}$-Propagate. This rule performs the kind of theory propagation described in §1.4.3.4 by adding to the current assignment μ any undefined literal l that is $\mathcal{T}$-entailed by μ. The rule's precondition maintains the invariant that every atom in an assignment occurs in the initial formula of the derivation. This invariant is crucial for termination.

$\mathcal{T}$-Backjump. This rule models the kind of *conflict-driven* backjumping described in §1.4.3.5. As defined, the rule is only triggered by a propositional conflict with one of the current clauses ($\mu \models_p \neg c$). This is for simplicity and uniformity, and without loss of generality thanks to the $\mathcal{T}$-Propagate and $\mathcal{T}$-Learn rules. Using those rules, any conflict involving the theory $\mathcal{T}$ is reducible to a propositional conflict. For instance, if the current assignment μ has a $\mathcal{T}$-inconsistent subset η, then $\emptyset \models_{\mathcal{T}} \neg\eta$. The theory lemma $\neg\eta$ can then be added to the current clause set φ by one application of $\mathcal{T}$-Learn and then play the role of clause c in $\mathcal{T}$-Backjump.

$\mathcal{T}$-Backjump is triggered by the existence of a conflicting clause, but in order to undo (the effect of) an earlier decision step it needs a backjump clause $c' \vee l'$, which synthesizes the reason of the conflict, indicating the level to backjump to and the literal to put in place of the decision literal for that level. This clause is the dual of the conflict set η discussed in §1.4.3.5; that is, $c' \vee l' = \neg\eta$.

$\mathcal{T}$-Learn. This rule allows the addition to the current formula φ of an arbitrary clause c that is $\mathcal{T}$-entailed by φ and consists of atoms in the current state. It can be used to model the static learning techniques described in §1.4.3.2 as well as the usual conflict-driven lemma learning that adds backjump clauses, or any other technique that takes advantage of theory consequences of φ. In particular, it can be used to model uniformly all the early pruning techniques described in §1.4.3.3. This is because any backjumping motivated by the $\mathcal{T}$-inconsistency of the current assignment μ can be modeled as the discovery and learning of a theory lemma c that is propositionally falsified by μ, followed by an application of $\mathcal{T}$-Backjump with c as the conflicting clause. The rule does not model the learning of clauses containing *new* literals, which is done by some SMT solvers, as seen in §1.4.3.5. An extension of Abstract $\mathcal{T}$-DPLL that allows that is dicussed later in §1.5.2.

We can now describe some of the approaches for implementing $\mathcal{T}$-DPLL solvers discussed in §1.4.2 in terms of Abstract $\mathcal{T}$-DPLL. (See [114] for more details.)

[22] Note that the precondition of Decide does not include a check on whether Propagate, say, applies to l or its negation. This is intentional since such control considerations are best left to a rule application strategy.

Offline integrations of a theory solver and a DPLL solver are modeled by a class of rule application strategies that do not use the $\mathcal{T}$-Propagate rule. Whenever a strategy in this class produces a state $\mu \parallel \varphi$ irreducible by Decide, Fail, Propagate, or $\mathcal{T}$-Backjump, the sequence μ is a satisfying assignment for the formula φ_0 in the initial state $\emptyset \parallel \varphi_0$, but it may not be $\mathcal{T}$-consistent. If it is not, there exists a $\eta \subseteq \mu$ such that $\emptyset \models_{\mathcal{T}} \neg\eta$. The strategy then adds the theory lemma $\neg\eta$ (the blocking clause) with one $\mathcal{T}$-Learn step and applies Restart, repeating the same cycle above until a $\mathcal{T}$-compatible state is generated. With an incremental *$\mathcal{T}$-solver* it might be more convenient to check the $\mathcal{T}$-consistency of μ in a state $\mu \parallel \varphi$ even if the state is reducible by one of the four rules above. In that case, it is possible to learn a blocking clause and restart earlier. Strategies like these are sound and complete. To be fair, and so terminating, they must not remove (with $\mathcal{T}$-Forget) any blocking clause.

A strategy can take advantage of an online SAT-solver by preferring backjumping to systematic restarting after μ is found $\mathcal{T}$-inconsistent. This is done by first learning a lemma $\neg\eta$ for some $\eta \subseteq \mu$, and then *repairing* μ using that lemma. In fact, since $\neg\eta$ is a conflicting clause by construction, either $\mathcal{T}$-Backjump or Fail applies—depending respectively on whether μ contains decision literals or not. To be fair, strategies that apply $\mathcal{T}$-Backjump/Fail instead of Restart do not need to keep the blocking clause once they use it as the conflicting clause for backjumping.

Any fair strategy above remains fair if it is modified to interleave arbitrary applications of $\mathcal{T}$-Propagate. Stronger results are possible if $\mathcal{T}$-Propagate is applied eagerly (in concrete, if it has higher priority than Decide). In that case, it is impossible to generate $\mathcal{T}$-inconsistent assignments, and so it is unnecessary for correctness to learn any blocking clauses, or any theory lemmas at all.

1.5. Theory Solvers

The lazy approach to SMT as described above relies on combining a SAT solver with a *theory solver* for some theory $\mathcal{T}$. The role of the theory solver is to accept a set of literals and report whether the set is $\mathcal{T}$-satisfiable or not. Typically, theory solvers are *ad hoc*, with specialized decision procedure tailored to the theory in question. For details on such procedures for some common theories, we refer the reader to the references given in §1.2.2. Because different theories often share some charactaristics, a natural question is whether there exist general or parameterized methods that have broader applicability. This section discusses several such approaches that have been developed: Shostak's method (§1.5.1), splitting on demand (§1.5.2), layered theory solvers (§1.5.3), and rewriting-based theory solvers (§1.5.4).

1.5.1. Shostak's method

Shostak's method was introduced in a 1984 paper [138] as a general method for combining the theory of equality with one or more additional theories that satisfy certain specific criteria. The original paper lacks rigor and contains serious errors, but work since then has shed considerable light on the original claims [52, 128, 135, 89, 21, 71]. We summarize some of the most signifiant results here. We start with some definitions.

Definition 1.5.1. A set $\mathcal{S}$ of equations is said to be in *solved form* iff the left-hand side of each equation in $\mathcal{S}$ is a ground constant which appears only once in $\mathcal{S}$. We will refer to these constants appearing only on the left-hand sides as *solitary* constants.

A set $\mathcal{S}$ of equations in solved form defines an idempotent substitution: the one which replaces each solitary constant with its corresponding right-hand side. If S is a term or set of terms, we denote the result of applying this substitution to S by $\mathcal{S}(S)$.

Definition 1.5.2. Given a formula F and a formula G, we define $\gamma_F(G)$ as follows:

1. Let G' be the formula obtained by replacing each free constant symbol in G that does not appear in F with a fresh variable.
2. Let $\overline{v}$ be the set of all fresh variables introduced in the previous step.
3. Then, $\gamma_F(G) = \exists\, \overline{v}.\, G'$.

Definition 1.5.3. A consistent theory $\mathcal{T}$ with signature Σ is a *Shostak* theory if the following conditions hold.

1. Σ does not contain any predicate symbols.
2. There exists a *canonizer canon*, a computable function from Σ-terms to Σ-terms, with the property that $\models_{\mathcal{T}} s = t$ iff $canon(s) \equiv canon(t)$.
3. There exists a *solver solve*, a computable function from Σ-equations to sets of formulas defined as follows:
   (a) If $\models_{\mathcal{T}} s \neq t$, then $solve(s = t) \equiv \{\bot\}$.
   (b) Otherwise, $solve(s = t)$ returns a set $\mathcal{S}$ of equations in solved form such that $\models_{\mathcal{T}} (s = t) \leftrightarrow \gamma_{s=t}(\mathcal{S})$.

The main result in Shostak's paper is that given a Shostak theory $\mathcal{T}$, a simple algorithm can be used to determine the satisfiability of conjunctions of Σ-literals. Algorithm *S1* (shown in Figure 1.5) makes use of the properties of a Shostak theory to check the joint satisfiability of an arbitrary set of equalities, Γ, and an arbitrary set of disequalities, Δ, in a Shostak theory with canonizer *canon* and solver *solve*.

Algorithm *S1* is sound and complete whenever Δ contains at most one disequality or if $\mathcal{T}$ satisfies the additional requirement of being convex (as defined in §1.2.1.2).

Example 1.5.4. Perhaps the most obvious example of a Shostak theory is $T'_{\mathcal{R}}$. A simple canonizer for this theory can be obtained by imposing an order on all ground constants and combining like terms. For example, $canon(c+3b-a-5c) \equiv -a + 3b + (-4c)$. Similarly, a solver can be obtained simply by solving for one of the constants in an equation (returning $\bot$ if no solution exists). For this theory, Algorithm *S1* corresponds to Gaussian elimination with back-substitution. Consider the following set of literals: $\{a+3b-2c = 1,\, a-b-6c = 1,\, b+a \neq a-c\}$. The following table shows the values of Γ, $\mathcal{S}$, $s^* = t^*$, and $\mathcal{S}^*$ on each iteration of Algorithm *S1* starting with $\Gamma = \{a + 3b - 2c = 1, a - b - 6c = 1\}$:

```
S1(Γ, Δ, canon, solve)
1.  S := ∅;
2.  WHILE Γ ≠ ∅ DO BEGIN
3.    Remove some equality s = t from Γ;
4.    s* := S(s); t* := S(t);
5.    S* := solve(s* = t*);
6.    IF S* = { ⊥ } THEN RETURN FALSE;
7.    S := S*(S) ∪ S*;
8.  END
9.  IF canon(S(s)) ≡ canon(S(t)) for some s ≠ t ∈ Δ THEN RETURN FALSE;
10. RETURN TRUE;
```

Figure 1.5. Algorithm S1: A simple satisfiability checker based on Shostak's algorithm

Γ	$\mathcal{S}$	$s^* = t^*$	$\mathcal{S}^*$
$a + 3b - 2c = 1$ $a - b - 6c = 1$	$\emptyset$	$a + 3b - 2c = 1$	$a = 1 - 3b + 2c$
$a - b - 6c = 1$	$a = 1 - 3b + 2c$	$1 - 3b + 2c - b - 6c = 1$	$b = -c$
$\emptyset$	$a = 1 + 5c$ $b = -c$		

Now, notice that $canon(\mathcal{S}(b+a)) \equiv canon(-c+1+5c) \equiv 1+4c$ and $canon(\mathcal{S}(a-c)) \equiv canon(1 + 5c - c) \equiv 1 + 4c$. Since $b + a \neq a - c \in \Delta$, the algorithm returns `FALSE` indicating that the orginal set of literals is unsatisfiable in $\mathcal{T}$.

1.5.1.1. Combining Shostak theories

Besides providing a satisfiability procedure for a single Shostak theory $\mathcal{T}$, the original paper makes several additional claims. The first is that a variation of Algorithm *S1* can be used to decide the satisfiability of any theory $\mathcal{T}'$, where $\mathcal{T}'$ is the extension of $\mathcal{T}$ after adding an arbitrary number of constant and function symbols. In other words, there is an algorithm for the *combination* of $\mathcal{T}$ with the theory of equality. This claim has been clarified and proved to be correct in later work [128, 71], and we do not elaborate on it here.

The second claim regarding combinations of theories is that given any two Shostak theories, their canonizers and solvers can be combined to obtain a decision procedure for the combined theories. While it is true that canonizers can be combined (see [89, 135]), it was shown in [89] that solvers can almost never be combined, and thus Shostak's method as orignially presented does not provide a way to combine theories (beyond simple combinations of a single Shostak theory with the theory of equality). In [135], a correct method for combining Shostak theories is given. However, the method does not combine theory solvers as proposed by Shostak, but relies, instead, on the Nelson-Oppen framework covered in §1.6.1.

1.5.2. Splitting on demand

Thus far, we have assumed that a theory solver $\mathcal{T}$*-solver* for a theory $\mathcal{T}$ takes as input a set of literals and outputs true if the set is $\mathcal{T}$-consistent and false otherwise. For some important theories, determining the $\mathcal{T}$-consistency of a conjunction of literals requires *internal* case splitting (i.e. case splitting within the $\mathcal{T}$*-solver*).

Example 1.5.5. In the theory $\mathcal{T}_\mathcal{A}$ of arrays introduced in §1.2.2, consider the following set of literals: $\mathit{read}(\mathit{write}(A, i, v), j) = x, \mathit{read}(A, j) = y, x \neq v, x \neq y$. To see that this set is unsatisfiable, notice that if $i = j$, then $x = v$ because the value read should match the value written in the first equation. On the other hand, if $i \neq j$, then $x = \mathit{read}(A, j)$ and thus $x = y$. Deciding the $\mathcal{T}_\mathcal{A}$-consistency of larger sets of literals may require a significant amount of such reasoning by cases.

Because theories like $\mathcal{T}_\mathcal{A}$ require internal case splits, solving problems with general Boolean structure over such theories using the framework developed in §1.4 results in a system where case splitting occurs in two places: in the Boolean DPLL (SAT) engine as well as inside the theory solver. In order to simplify the implementation of theory solvers for such theories, and to centralize the case splitting in a single part of the system, it is desirable to allow a theory solver $\mathcal{T}$*-solver* to *demand* that the DPLL engine do additional case splits before determining the $\mathcal{T}$-consistency of a partial assignment. For flexibility—and because it is needed by actual theories of interest—the theory solver should be able to demand case splits on literals that may be unfamiliar to the DPLL engine and may possibly even contain fresh constant symbols. Here, we give a brief explanation of how this can be done in the context of the abstract framework given in §1.4.4. Details can be found in [14].

Recall that in the abstract framework, the $\mathcal{T}$-Learn rule allows the theory solver to add an arbitrary clause to those being processed by the DPLL engine, so long as all the atoms in that clause are already known to the DPLL engine. Our approach will be to relax this restriction. It is not hard to see, however, that this poses a potential termination problem. We can overcome this difficulty so long as for any input formula ϕ, the set of all literals needed to check the $\mathcal{T}$-consistency of ϕ is finite. We formalize this notion by introducing the following definition.

Definition 1.5.6. $\mathcal{L}$ is a *suitable literal-generating function* if for every finite set of literals L:

1. $\mathcal{L}$ maps L to a new finite set of literals L' such that $L \subseteq L'$.
2. For each atomic formula α, $\alpha \in \mathcal{L}(L)$ iff $\neg\alpha \in \mathcal{L}(L)$.
3. If L' is a set of literals and $L \subseteq L'$, then, $\mathcal{L}(L) \subseteq \mathcal{L}(L')$ (monotonicity).
4. $\mathcal{L}(\mathcal{L}(L)) = \mathcal{L}(L)$ (idempotence).

For convenience, given a formula ϕ, we denote by $\mathcal{L}(\phi)$ the result of applying $\mathcal{L}$ to the set of all literals appearing in ϕ. In order to be able to safely use splitting on demand for a theory solver $\mathcal{T}$*-solver*, we must be able to show the existence of a suitable literal-generating function $\mathcal{L}$ such that: for every input formula ϕ, the set of all literals on which the $\mathcal{T}$*-solver* may demand case splits when starting with a conjunction of literals from ϕ is contained in $\mathcal{L}(\phi)$. For example, for $\mathcal{T}_\mathcal{A}$, $\mathcal{L}(\phi)$

could contain atoms of the form $i = j$, where i and j are array indices occurring in ϕ. Note that there is no need to explicitly construct $\mathcal{L}(\phi)$. It is enough to know that it exists.

As mentioned above, it is sometimes useful to demand case splits on literals containing new constant symbols. The introduction of new constant symbols poses potential problems not only for termination, but also for soundness. This is because the abstract framework relies on the fact that whenever $\emptyset \parallel \phi$ reduces in one or more steps to $\mu \parallel \phi'$, the formulas ϕ and ϕ' are $\mathcal{T}$-equivalent. This is no longer true if we allow the introduction of new constant symbols. Fortunately, it is sufficient to ensure $\mathcal{T}$-equisatisfiability of ϕ and ϕ'. With this in mind, we can give a new transition rule called Extended $\mathcal{T}$-Learn which replaces $\mathcal{T}$-Learn and allows for the desired additional flexibility.

Definition 1.5.7. The *Extended DPLL Modulo Theories system*, consists of the rules of §1.4.4 except that the $\mathcal{T}$-Learn rule is replaced by the following[23] rule:

Extended $\mathcal{T}$-Learn

$$\mu \parallel \phi \quad \Longrightarrow \quad \mu \parallel \phi, c \quad \textbf{if} \begin{cases} \text{each atom of } c \text{ occurs in } \phi \text{ or in } \mathcal{L}(\mu) \\ \phi \models_T \gamma_\phi(c) \end{cases}$$

The key observation is that an implementation using Extended $\mathcal{T}$-Learn has more flexibility when a state $\mu \parallel \phi$ is reached that is final with respect to the basic rules Propagate, Decide, Fail, and $\mathcal{T}$-Backjump. Whereas before it would have been necessary to determine the $\mathcal{T}$-consistency of μ when such a state was reached, the Extended $\mathcal{T}$-Learn rule allows the possibility of *delaying* a response by demanding that additional case splits be done first. As shown in [14], the extended framework retains soundness and completeness. Furthermore, the properties of $\mathcal{L}$ ensure that a delayed response cannot be delayed indefinitely, and thus the framework also ensures termination under similar conditions as the original framework.

Example 1.5.8. A careful examination of the decision procedure for $\mathcal{T}_\mathcal{A}$ given in [145] reveals the following:

1. Each term can be categorized as an *array* term, an *index* term, a *value* term, or a *set* term.
2. No new array terms are ever introduced by the inference rules.
3. At most one new index term for every pair of array terms is introduced.
4. Set terms are made up of some finite number of index terms.
5. The only new value terms introduced are of the form $read(a, i)$ where a is an array term and i is an index term.

It follows that the total number of possible terms that can be generated by the procedure starting with any finite set of literals is finite. Because there are only a finite number of predicates, it then follows that this set of rules is literal-bounded.

In general, a similar analysis must be done for every theory before it can be integrated with the Extended DPLL Modulo Theories framework as described above. It should also be pointed out that the extended framework requires a DPLL engine that is capable of dealing with a dynamically expanding set of literals. However,

[23] The definition of γ was given in §1.5.1.

because the splitting on demand approach can result in drastically simpler theory solvers, it remains an attractive and useful implementation strategy.

The approach can be refined to work with several theory solvers when the background theory $\mathcal{T}$ is a combination of the solver's theories [15, 14]. Then, the lemma generation mechanism is also used to achieve a sound and complete cooperation among the theory solvers, in the spirit of the Nelson-Oppen combination method described in §1.6. In this form, splitting on demand is implemented in the CVC, CVC Lite and CVC3 systems [144, 13, 18]. A major requirement of the refinement is that each theory solver must be aware that it is being combined with others so that it can generate suitable lemmas necessary for the cooperation. An alternative approach to achieving inter-solver cooperation through the DPLL engine that does not have this requirement is described in some detail in §1.6.3.

1.5.3. Layered theory solvers

Sometimes, a theory $\mathcal{T}$ has the property that a fully general solver for $\mathcal{T}$ is not always needed: rather, the unsatisfiability of an assignment μ can often be established in less expressive, but much easier, sub-theories. Thus, the $\mathcal{T}$*-solver* may be organized in a *layered hierarchy* of solvers of increasing solving capabilities [8, 34, 136, 50, 38]. The general idea consists of stratifying the problem over N layers $L_0, L_1, \ldots, L_{N-1}$ of increasing complexity, and searching for a solution "at as simple a level as possible". If one of the simpler solvers finds a conflict, then this conflict is used to prune the search at the Boolean level; if it does not, the next solver in the sequence is activated.

Since L_{n+1} refines L_n, if the set of literals is not satisfiable at level L_n, then it is not at $L_{n+1}, \ldots, L_{N-1}$. If indeed a model 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 for at L_{n+1}. In this way, much of the reasoning can be performed at a high level of abstraction. This results in increased efficiency during the search for a solution, since the later solvers, which are often responsible for most of the complexity, are avoided whenever possible.

The schema can be further enhanced by allowing each layer L_i to infer novel equalities and inequalities and to pass them down to the next layer L_{i+1}, so as to better drive its search [136, 137, 50].

1.5.4. Rewriting-based theory solvers

Another approach for building theory solvers relies on the power and flexibility of modern automated theorem provers, in particular, provers based on the *superposition calculus* [115], a modern version of the resolution calculus for first-order logic with equality. This calculus is based on term rewriting techniques and comes equipped with powerful redundancy criteria that allow one to build very effective control strategies for reducing the search space.

The superposition-based approach to SMT, first proposed in [7] and then further elaborated upon in [3, 27, 25, 26], applies to theories $\mathcal{T}$ that are axiomatizable by a finite (and relatively small) set of first-order clauses, such as for

instance the theory of arrays in §1.2.2. The main idea is to instrument a superposition prover with specialized control strategies which, together with the axioms of $\mathcal{T}$, effectively turn the prover into a decision procedure for ground $\mathcal{T}$-satisfiability.

An advantage of the approach is a simplified proof of correctness, even in the case of combined theories, which reduces to a routine proof of termination of the application of the various superposition rules (see, e.g., [7] for details). Another potential advantage is the reuse of efficient data structures and algorithms for automated deduction implemented in state-of-the-art theorem provers. The main disadvantage is that to get additional features typically required in SMT such as model generation, incrementality, and so on, one may need to modify the theorem prover in ways not foreseen by the original implementors, possibly at the cost of a considerable (re)implementation effort. While this approach has generated an active stream of interesting theoretical work, its practical impact has been limited so far by a scarcity of robust and competitive implementations.

1.6. Combining Theories

We mentioned that in SMT one is often faced with formulas involving several theories at once. This is particularly true in software verification, where proof obligations are formulas talking about several datatypes, each modeled by its own theory. We have seen that for many of these theories the ground satisfiability problem is decidable, and often by efficient theory solvers. Hence, a natural question is whether it is possible to combine theory solvers for several *component* theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$ *modularly* into a theory solver that decides ground satisfiability modulo their combination $\mathcal{T}_1 \oplus \cdots \oplus \mathcal{T}_n$.

In general, the answer is negative, simply because there exist theories with a decidable ground satisfiability problem whose combination has an undecidable ground satisfiability problem (see, e.g., [28]). Positive results are however possible by imposing restrictions on the component theories and their combination. A successful combination method for theory solvers is due to Nelson and Oppen [107]. The success of the method is based on the fact that its restrictions on the theories are not very strong in practice, and lead to efficient implementations. It is fair to say that most work in theory combination in SMT is based on extensions and refinements of Nelson and Oppen's work (see, e.g., [147, 127, 22, 148, 149, 75, 150, 126, 77, 37, 76, 48, 14, 90]).

In this section, we present a declarative non-deterministic combination framework, first presented in [117], that captures the essence of the original combination procedure by Nelson and Oppen in [107], and briefly discuss a few variants and extensions. Then we describe how an efficient implementation of this framework can be incorporated into the lazy approach to SMT.

For simplicity, we consider just the case of combining two theories and their solvers since the case of more theories is analogous. Therefore, let Σ_1, Σ_2 be two signatures and let $\mathcal{T}_i$ be a Σ_i theory for $i = 1, 2$. The combination of $\mathcal{T}_1$ and T_2 will be the theory $\mathcal{T}_1 \oplus \mathcal{T}_2$ as defined in §1.2.1.3.

1.6.1. A Logical Framework for Nelson-Oppen Combination

The original Nelson-Oppen method applies to theories $\mathcal{T}_1$ and $\mathcal{T}_2$ with disjoint signatures and each equipped with a theory solver deciding a ground $\mathcal{T}_i$-satisfiability problem. The gist of the method is to take a ground formula of signature $\Sigma_1 \cup \Sigma_2 \cup C$, where C is a set of constant symbols not in $\Sigma_1 \cup \Sigma_2$, convert it into an equisatisfiable conjunction $\varphi_1 \wedge \varphi_2$ with each φ_i of signature $\Sigma_i \cup C$, and feed each φ_i to $\mathcal{T}_i$'s theory solver. The two solvers cooperate by exchanging entailed constraints about their respective formulas until one of them has enough information to decide the satisfiability in $\mathcal{T}_1 \oplus \mathcal{T}_2$ of the original formula.

Any version of the Nelson-Oppen method is more conveniently described by considering only $(\Sigma_1 \cup \Sigma_2 \cup C)$-formulas φ that are just *conjunctions of literals*. This restriction is without loss of generality both in theory, via a preliminary conversion to disjunctive normal form, and in practice, as we will see in §1.6.3. Now, queries like φ cannot be processed directly by either theory solver unless they have the *pure form* $\varphi_1 \wedge \varphi_2$ where each φ_i is a $(\Sigma_i \cup C)$-formula—possibly $\top$. Even then, however, using the theory solvers in isolation is not enough because it is possible for $\varphi_1 \wedge \varphi_2$ to be $\mathcal{T}_1 \oplus \mathcal{T}_2$-unsatisfiable while each φ_i is $\mathcal{T}_i$-satisfiable. By a simple application of Craig's interpolation lemma [51] this situation happens precisely when there is a first-order formula ψ over the signature C such that $\varphi_1 \models_{\mathcal{T}_1} \psi$ and $\varphi_2 \wedge \psi \models_{\mathcal{T}_2} \bot$. Note that Craig's lemma tells us here that $=$ is the only predicate symbol and the elements of C are the only function symbols occurring in ψ. But it does not provide any indication of whether ψ has quantifiers or not, and which ones. One of the main theoretical contributions of Nelson and Oppen's work was to show that, under the right conditions, ψ is actually ground. Moreover, it is also computable.

The Nelson-Oppen method is not limited to formulas in pure form because any ground formula can be (efficiently) turned into an equisatisfiable pure form by a suitable *purification procedure.* indexpurification The procedure most commonly used (and its correctness proof) is straightforward but to describe it we need some definitions and notation first.

Let $\Sigma = \Sigma_1 \cup \Sigma_2 \cup C$ and fix $i \in \{1, 2\}$. A Σ-term t is an *i-term* if its top function symbol is in $\Sigma_i \cup C$. A Σ-literal α is an *i-literal* if its predicate symbol is in $\Sigma_i \cup C$ or if it is of the form $(\neg)s = t$ and both s and t are i-terms or $i = 1$.[24] A subterm of an i-atom α is an *alien* subterm of α if it is a j-term, with $j \neq i$, and all of its superterms in t are i-terms. An i-term or i-literal is *pure* (or also, *i-pure*) if it only contains symbols from $\Sigma_i \cup C$. Note that the constants in C are the only terms that are both a 1-term and a 2-term, and that an equation is pure whenever one of its members is a constant of C and the other is a pure term. The purification procedure consists of the following steps.

Purification Procedure. Let φ be a conjunction of Σ-literals.

1. **Abstract alien subterms.** Apply to completion the following transformation to φ: replace an alien subterm t of a literal of φ with a fresh constant c from C and add (conjunctively) the equation $c = t$ to φ.

[24] This means that any literal $(\neg)s = t$ where s and t are not both i-terms for some i is considered to be a 1-term. This choice is arbitrary and immaterial.

2. **Separate.** For $i = 1, 2$, let φ_i be the conjunctions of all the i-literals in (the new) φ.[25]

It is not hard to see that this procedure always terminates, runs in time linear in the size of φ, and produces a formula $\varphi_1 \wedge \varphi_2$ that is equisatisfiable with φ. More precisely, every model of $\varphi_1 \wedge \varphi_2$ is also a model of φ, and for every model $\mathcal{A}$ of φ there is a model of $\varphi_1 \wedge \varphi_2$ that differs from $\mathcal{A}$ at most in the interpretation of the new constants introduced in the abstraction step above.

The constraint propagation mechanism of the original Nelson-Oppen procedure can be abstracted by a preliminary non-deterministic guess of a truth assignment for all possible *interface equalities*, that is, equations between the (uninterpreted) constants shared by φ_1 and φ_2. To describe that, it is convenient to introduce the following notion [147].

Let R be any equivalence relation over a finite set S of terms. The *arrangement of S induced by R* is the formula

$$ar_R(S) := \bigwedge_{(s,t)\in R} (s = t) \ \wedge \bigwedge_{(s,t)\notin R} s \neq t$$

containing all equations between R-equivalent terms and all disequations between non-R-equivalent terms of S.

The combination procedure below just guesses an arrangement of the shared constants in a pure form $\varphi_1 \wedge \varphi_2$ of φ, adds it to each φ_i, and then asks the corresponding theory solver to check the satifiability of the extended φ_i.

The Combination Procedure. Let φ be a conjunction of Σ-literals.

1. **Purify input.** For $i = 1, 2$, let φ_i be the i-pure part of φ's pure form.
2. **Guess an arrangement.** Where C_0 is the set of all the constant symbols occurring in both φ_1 and φ_2, choose an arrangement $ar_R(C_0)$.
3. **Check pure formulas.** Return "satisfiable" if $\varphi_i \wedge ar_R(C_0)$ is $\mathcal{T}_i$-satisfiable for $i = 1$ and $i = 2$; return "unsatisfiable" otherwise.

We call any arrangement guessed in Step 2 of the procedure above an *arrangement for φ*. The procedure is trivially terminating, even if all possible arrangements are considered in step 2.[26] It is also refutationally sound for *any* signature-disjoint theories $\mathcal{T}_1$ and $\mathcal{T}_2$: for such theories, φ is $\mathcal{T}_1 \oplus \mathcal{T}_2$-unsatisfiable if the procedure returns "unsatisfiable" for every possible arrangement for φ. The procedure, however, is not complete without further assumptions on $\mathcal{T}_1$ and $\mathcal{T}_2$; that is, in general there could be arrangements for which it returns "satisfiable" even if φ is in fact $\mathcal{T}_1 \oplus \mathcal{T}_2$-unsatisfiable. A sufficient condition for completeness is that both theories be *stably infinite* [117].

Definition 1.6.1. Let Σ be a signature and C an infinite set of constants not in Σ. A Σ-theory T is *stably infinite* if every $\mathcal{T}$-satisfiable ground formula of signature $\Sigma \cup C$ is satisfiable in a model of $\mathcal{T}$ with an infinite universe.

[25] Note that (dis)equations between constants of C end up in both φ_1 and φ_2.

[26] Observe that there is always a finite, albeit possibly very large, set of arrangements for any finite set of terms.

Proposition 1.6.2 (Soundness and completeness)**.** If $\mathcal{T}_1$ and $\mathcal{T}_2$ are signature-disjoint and both stably infinite, then the combination procedure returns "unsatisfiable" for an input φ and every possible arrangement for φ iff φ is $\mathcal{T}_1 \oplus \mathcal{T}_2$-unsatisfiable.

The first (correct) proof of the result above was given in [117]. More recent proofs based on model-theoretic arguments can be found in [147, 97, 75], among others.[27] The completeness result in the original paper by Nelson and Oppen [107] is incorrect as stated because it imposes no restrictions on the component theories other than that they should be signature-disjoint. The completeness proof however is incorrect only because it implicitly assumes there is no loss of generality in considering only infinite models, which is however not the case unless the theories are stably infinite.

We remark that the stable infiniteness requirement is not an artifice of a particular completeness proof. The combination method is really incomplete without it, as shown for instance in [150]. This is a concern because non-stably infinite theories of practical relevance do exist. For instance, all the theories of a single finite model, such as the theory of bit-vectors (or of strings) with some maximum size, are not stably infinite.

Current practice and recent theoretical work have shown that it is possible to lift the stable infiniteness requirement in a number of ways provided that the combination method is modified to include the propagation of certain cardinality constraints, in addition to equality constraints between shared constants [68, 150, 69, 126, 90]. More importantly, recent research has shown (see [69, 126, 90]) that the vexation of the requirement is greatly reduced or disappears in practice if one frames SMT problems within a sorted (i.e., typed) logic, a more natural kind of logic for SMT applications than the classical, unsorted logic traditionally used. The main idea is that, in SMT, combined theories are usually obtained by combining the theory of some parametric data type $T(\alpha_1, \ldots, \alpha_n)$ with the theories of particular instances of the type parameters $\alpha_1, \ldots, \alpha_n$. A simple example would be the combined theory of lists of real numbers, say, where the type $\mathsf{List}(\mathsf{Real})$ is obtained by instantiating with the type Real the parameter α in the parametric type $\mathsf{List}(\alpha)$.

With combined theories obtained by type parameter instantiation, the theories of the instantiated parameters can be arbitrary, as long as entailed cardinality constraints on the parameter types are propagated properly between theory solvers. For instance, in the theory of lists of Booleans, say, a pure formula φ_1 of the form

$$l_1 \neq l_2 \wedge l_1 \neq l_3 \wedge l_2 \neq l_3 \wedge \mathit{tail}(l_1) = \mathit{nil} \wedge \mathit{tail}(l_2) = \mathit{nil} \wedge \mathit{tail}(l_3) = \mathit{nil},$$

stating that l_1, l_2, l_3 are distinct lists of length 1, entails the existence of at least 3 distinct values for the element type Bool. If φ_1 above is part of the formula sent to the list solver, its entailed minimal cardinality constraint on the Bool type must be communicated to the theory solver for that type. Otherwise, the unsatisfiability of φ_1 in the combined theory of $\mathsf{List}(\mathsf{Bool})$ will go undetected. A

[27] While those papers consider only theories specified by a set of axioms, their proofs also apply to theories specified as sets of models.

description of how it is possible to propagate cardinality constraints conveniently for theories of typical parametric datatypes, such as lists, tuples, arrays and so on, can be found in [126]. A discussion on why in a typed setting parametricity and not stable infiniteness is the key notion to consider for Nelson-Oppen style combination is provided in [90].

1.6.2. The Nelson-Oppen Procedure

The original Nelson-Oppen procedure can be seen as a concrete, and improved, implementation of the non-deterministic procedure in the previous subsection.

The first concrete improvement concerns the initial purification step on the input formula. Actually purifying the input formula is unnecessary in practice provided that each theory solver accepts literals containing alien subterms, and treats the latter as if they were free constants. In that case, interface equalities are actually equalities between certain alien subterms.[28] A description of the Nelson-Oppen procedure without the purification step is provided in [21].

A more interesting improvement is possible when each component theory $\mathcal{T}_i$ is convex—as defined in §1.2.1.2. Then, it is enough for each theory solver to propagate recursively and to completion all the interface equalities entailed by its current pure half of the input formula. In concrete, this requires each solver to be able to infer entailed interface equalities and to pass them to the other solver (for addition to its pure half) until one of the two detects an unsatisfiability, or neither has any new equalities to propagate. In the latter case, it is safe to conclude that the input formula is satisfiable in the combined theory.

When one of the two theories, or both, are non-convex, exchanging just entailed interface equalities is no longer enough for completeness. The solver for the non-convex theory must be able to infer, and must propagate, also any *disjunction* of interface equality entailed by its pure half of the formula. Correspondingly, the other solver must be able to process such disjunctions, perhaps by case splits, in addition to the conjunction of literals in the original input.

For convex theories—such as, for instance $\mathcal{T}_{\mathcal{E}}$—computing the interface equalities entailed by a conjunction φ of literals can be done very efficiently, typically as a by-product of checking φ's satisfiability (see, e.g., [109, 112]). For non-convex theories, on the other hand, computing entailed disjunctions of equalities can be rather expensive both in practice and in theory. For instance, for the difference logic fragment of $\mathcal{T}_{\mathcal{Z}}$, this entailment problem is NP-complete even if the satisfiability of conjunctions of literals can be decided in polynomial time [93].

1.6.3. Delayed Theory Combination

*Delayed Theory Combination(*DTC*)* is a general method for tackling the problem of theory combination within the context of lazy SMT [36, 37]. As in §1.6.1, we assume that $\mathcal{T}_1$, $\mathcal{T}_2$ are two signature-disjoint stably-infinite theories with their respective $\mathcal{T}_i$*-solvers*. Importantly, no assumption is made about the capability of the $\mathcal{T}_i$*-solvers* of deducing (disjunctions of) interface equalities from the input

[28] While this simple fact was well understood by earlier implementors of the Nelson-Oppen procedure—including its authors—it was (and still is) often overlooked by casual readers.

set of literals (e_{ij}-deduction capabilities, see §1.4.1): for each $\mathcal{T}_i$*-solver*, every intermediate situation from complete e_{ij}-deduction (like in deterministic Nelson-Oppen) to no e_{ij}-deduction capabilities (like in non-deterministic Nelson-Oppen) is admitted.

In a nutshell, in DTC the embedded DPLL engine not only enumerates truth-assignments for the atoms of the input formula, but also "nondeterministically guesses" truth values for the equalities that the $\mathcal{T}$*-solvers* 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 assigned to the $\mathcal{T}_i$*-solvers*.

An implementation of DTC [37, 40] is based on the online integration schema of Figure 1.3, exploiting early pruning, $\mathcal{T}$-propagation, $\mathcal{T}$-backjumping and $\mathcal{T}$-learning. Each of the two $\mathcal{T}_i$*-solvers* interacts only with the DPLL engine by exchanging literals via the truth assignment μ in a stack-based manner, so that there is no direct exchange of information between the $\mathcal{T}_i$*-solvers*. Let $\mathcal{T}$ be $\mathcal{T}_1 \cup \mathcal{T}_2$. The $\mathcal{T}$-DPLL algorithm is modified in the following ways [37, 40]:[29]

- $\mathcal{T}$-DPLL must be instructed to assign truth values not only to the atoms in φ, but also to the interface equalities not occurring in φ. $\mathcal{B}2\mathcal{T}$ and $\mathcal{T}2\mathcal{B}$ are modified accordingly. In particular, $\mathcal{T}$`-decide_next_branch` is modified to select also new interface equalities not occurring in the original formula.
- μ^p is partitioned into three components $\mu^p_{\mathcal{T}_1}$, $\mu^p_{\mathcal{T}_2}$ and μ^p_e, s.t. $\mu_{\mathcal{T}_i}$ is the set of i-pure literals and μ_e is the set of interface (dis)equalities in μ.
- $\mathcal{T}$`-deduce` is modified to work as follows: for each $\mathcal{T}_i$, $\mu^p_{\mathcal{T}_i} \cup \mu^p_e$, is fed to the respective $\mathcal{T}_i$*-solver*. If both return Sat, then $\mathcal{T}$`-deduce` returns Sat, otherwise it returns Conflict.
- Early-pruning is performed; if some $\mathcal{T}_i$*-solver* can deduce atoms or single interface equalities, then $\mathcal{T}$-propagation is performed. If one $\mathcal{T}_i$-solver performs the 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 $\mathcal{T}2\mathcal{B}(\mu^* \to \bigvee_{j=1}^k e_j)$ is learned.
- $\mathcal{T}$`-analyze_conflict` and $\mathcal{T}$`-backtrack` are modified so as to use the conflict set returned by one $\mathcal{T}_i$*-solver* for $\mathcal{T}$-backjumping and $\mathcal{T}$-learning. Importantly, such conflict sets may contain interface equalities.

In order to achieve efficiency, other heuristics and strategies have been further suggested in [36, 37, 40, 62, 55].

Example 1.6.3. [40] Consider the set of $\mathcal{T}_\mathcal{E} \cup \mathcal{T}_\mathcal{Z}$-literals $\mu =_{def} \mu_\mathcal{E} \wedge \mu_\mathcal{Z}$ of Figure 1.6. We assume that both the $\mathcal{T}_\mathcal{E}$ and $\mathcal{T}_\mathcal{Z}$ solvers have no e_{ij}-deduction capabilities. For simplicity, we also assume that both $\mathcal{T}_i$*-solvers* always return conflict sets which do not contain redundant interface disequalities $\neg e_{ij}$. (We adopt here a strategy for DTC which is described in detail in [40].) In short, $\mathcal{T}$-DPLL performs a Boolean search on the e_{ij}'s, backjumping on the conflicting clauses C_{13}, C_{56}, C_{23}, C_{24} and C_{14}, which in the end causes the unit-propagation of $(v_1 = v_4)$. Then, $\mathcal{T}$-DPLL selects a sequence of $\neg e_{ij}$'s without generating conflicts, and concludes that the formula is $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiable. Notice that the

[29] For simplicity, we assume φ is pure, although this condition is not necessary, as in [21].

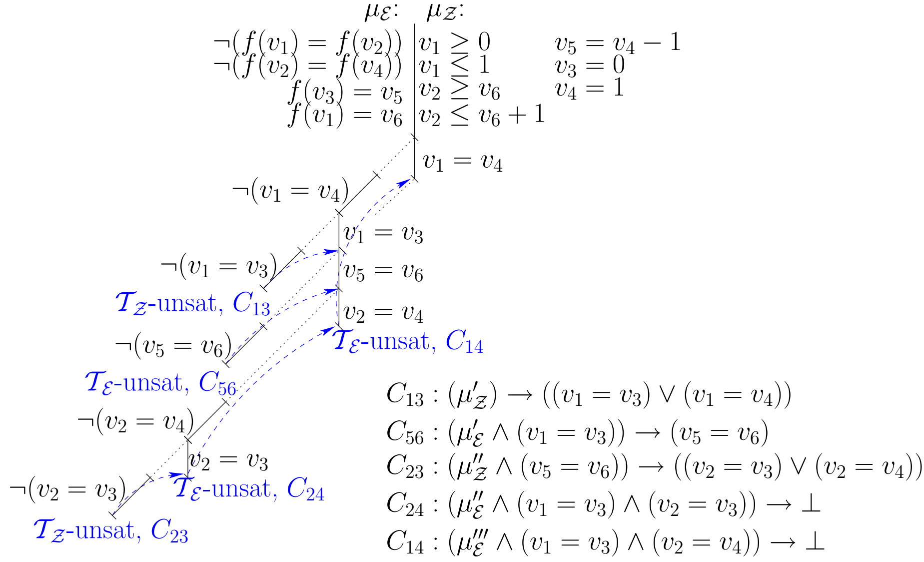

Figure 1.6. The DTC search tree for Example 1.6.3 on $\mathcal{T}_{\mathcal{Z}} \cup \mathcal{T}_{\mathcal{E}}$, with no e_{ij}-deduction. $v_1, \ldots, v_6$ are interface terms. $\mu'_{\mathcal{T}_i}, \mu''_{\mathcal{T}_i}, \mu'''_{\mathcal{T}_i}$ denote appropriate subsets of $\mu_{\mathcal{T}_i}$, $\mathcal{T}_i \in \{\mathcal{T}_{\mathcal{E}}, \mathcal{T}_{\mathcal{Z}}\}$.

backjumping steps on the clauses C_{13}, C_{56}, and C_{23} mimic the effects of performing e_{ij}-deductions.

By adopting $\mathcal{T}$*-solvers* with different e_{ij}-deduction power, one can trade part or all the e_{ij}-deduction effort for extra Boolean search. [40] shows that, if the $\mathcal{T}$*-solvers* have full e_{ij}-deduction capabilities, then no extra Boolean search on the e_{ij}'s is required; otherwise, the Boolean search is controlled by the quality of the conflict sets returned by the $\mathcal{T}$*-solvers*: the more redundant interface disequalities are removed from the conflict sets, the more Boolean branches are pruned. If the conflict sets do not contain redundant interface disequalities, the extra effort is reduced to one branch for each deduction saved, as in Example 1.6.3.

As seen in §1.5.2, the idea from DTC of delegating to the DPLL engine part or most of the, possibly very expensive, reasoning effort normally assigned to the $\mathcal{T}_i$*-solvers* (e_{ij}-deduction, case-splits) is pushed even further in the splitting on demand approach. As far as multiple theories are concerned, the latter approach differs from DTC in the fact that the interaction is controlled by the $\mathcal{T}_i$*-solvers*, not by the DPLL engine. Other improvements of DTC are currently implemented in the MATHSAT [37], YICES [62], and Z3 [55] lazy SMT tools. In particular, [62] introduced the idea of generating e_{ij}'s on-demand, and [55] that of having the Boolean search on e_{ij}'s driven by a model under construction.

1.6.4. Ackermann's expansion

When combining one or more theories with $\mathcal{T}_{\mathcal{E}}$, one possible approach is to eliminate uninterpreted function symbols by means of Ackermann's expansion [1]. The

method works as described in §1.3.1.3 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 formula φ' obtained is equisatisfiable with φ, and contains no uninterpreted function symbols. [39] presents a comparison between DTC and Ackermann's expansion.

Example 1.6.4. Consider the $\mathcal{T}_{\mathcal{E}} \cup \mathcal{T}_{\mathcal{Z}}$ conjunction μ of Example 1.6.3 [40]. Applying Ackermann's expansion we obtain the conjunction of $\mathcal{T}_{\mathcal{Z}}$-literals:

$$\begin{array}{rl}
\mu_{\mathcal{E}}: & \neg(v_{f(v_1)} = v_{f(v_2)}) \wedge \neg(v_{f(v_2)} = v_{f(v_4)}) \wedge (v_{f(v_3)} = v_5) \wedge (v_{f(v_1)} = v_6) \wedge \\
\mu_{\mathcal{Z}}: & (v_1 \geq 0) \wedge (v_1 \leq 1) \wedge (v_5 = v_4 - 1) \wedge (v_3 = 0) \wedge (v_4 = 1) \wedge \\
& (v_2 \geq v_6) \wedge (v_2 \leq v_6 + 1) \wedge \\
Ack: & ((v_1 = v_2) \rightarrow (v_{f(v_1)} = v_{f(v_2)})) \wedge ((v_1 = v_3) \rightarrow (v_{f(v_1)} = v_{f(v_3)})) \wedge \\
& ((v_1 = v_4) \rightarrow (v_{f(v_1)} = v_{f(v_4)})) \wedge ((v_2 = v_3) \rightarrow (v_{f(v_2)} = v_{f(v_3)})) \wedge \\
& ((v_2 = v_4) \rightarrow (v_{f(v_2)} = v_{f(v_4)})) \wedge ((v_3 = v_4) \rightarrow (v_{f(v_3)} = v_{f(v_4)})),
\end{array} \tag{1.2}$$

which every $\mathcal{T}_{\mathcal{Z}}$-solver finds $\mathcal{T}_{\mathcal{Z}}$-satisfiable. (E.g., the $\mathcal{T}_{\mathcal{Z}}$-model $v_2 = 2$, $v_3 = v_5 = v_6 = v_{f(v_1)} = v_{f(v_3)} = 0$, $v_1 = v_4 = v_{f(v_2)} = v_{f(v_4)} = 1$ satisfies it.)

1.7. Extensions and Enhancements

1.7.1. Combining eager and lazy approaches

The Ackermann expansion described above is one way to combine eager and lazy approaches. Other hybrids of eager and lazy encoding methods can also be effective.

For instance, consider the satisfiability problem for integer linear arithmetic and the small-domain encoding technique presented in §1.3.2. Due to the conservative nature of the bound derived, and in spite of the many optimizations possible, the computed solution bound can generate a SAT problem beyond the reach of current solvers. For example, this situation can arise for problem domains that do not generate sparse linear constraints.

One can observe that the derived bounds are dependent only on the "bag of constraints", rather than on their specific Boolean combination in the input formula. Thus, there is hope that a smaller solution bound might suffice. Kroening et al. [87] have presented an approach to compute the solution bound incrementally, starting with a small bound and increasing it "on demand". Figure 1.7 outlines this *lazy* approach to computing the solution bound. Given a $\mathcal{T}_{\mathcal{Z}}$-formula $F_{\mathcal{Z}}$, we start with an encoding size for each integer variable that is smaller than that prescribed by the conservative bound (say, for example, 1 bit per variable).

If the resulting Boolean formula is satisfiable, so is $F_{\mathcal{Z}}$. If not, the proof of unsatisfiability generated by the SAT solver is used to generate a *sound abstraction* $F'_{\mathcal{Z}}$ of $F_{\mathcal{Z}}$. A sound abstraction is a formula, usually much smaller than the original, such that if it is unsatisfiable, so is the original formula. A sound and complete decision procedure for quantifier-free formulas in $\mathcal{T}_{\mathcal{Z}}$ is then used on $F'_{\mathcal{Z}}$. If this decision procedure concludes that $F'_{\mathcal{Z}}$ is unsatisfiable, so is $F_{\mathcal{Z}}$. If not, it provides a counterexample which indicates the necessary increase in the encoding size. A new SAT-encoding is generated, and the procedure repeats.

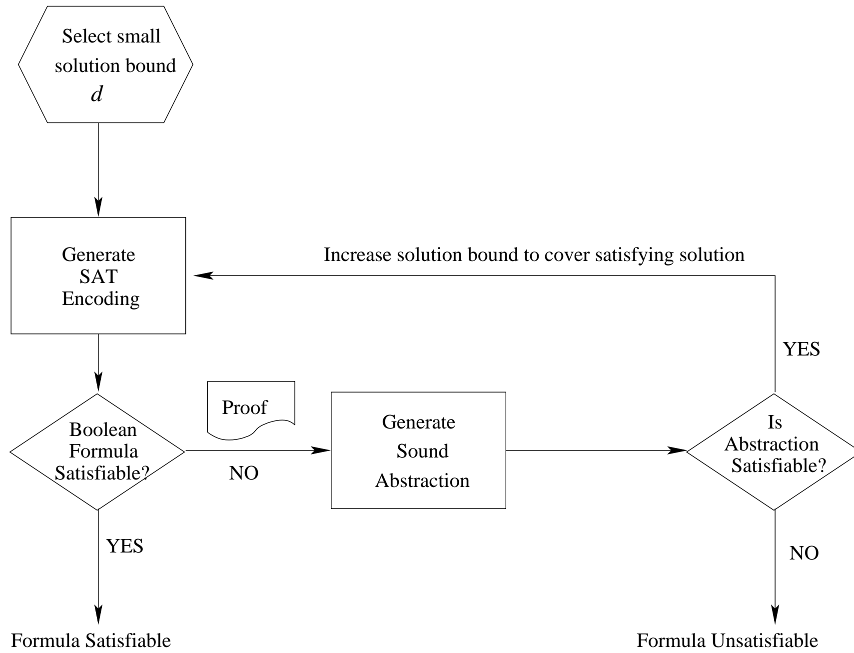

Figure 1.7. Lazy approach to computing solution bound

The bound S on solution size that is derived in §1.3.2 implies an upper bound nS on the number of iterations of this lazy encoding procedure; thus the lazy encoding procedure needs only polynomially many iterations before it terminates with the correct answer. Of course, each iteration involves a call to a SAT solver as well as to the $\mathcal{T}_{\mathcal{Z}}$-solver.

A key component of this lazy approach is the generation of the sound abstraction. While the details are outside the scope of this chapter, we sketch one approach here. (Details can be found in [87].) Assume that $F_{\mathcal{Z}}$ is in conjunctive normal form (CNF); thus, $F_{\mathcal{Z}}$ can be viewed as a set of clauses, each of which is a disjunction of linear constraints and Boolean literals. A subset of this set of clauses is a sound abstraction of $F_{\mathcal{Z}}$. This subset is computed by retaining only those clauses from the original set that contribute to the proof of unsatisfiability of the SAT-encoding.

If the generated abstractions are small, the sound and complete decision procedure used by this approach will run much faster than if it were fed the original formula. Thus, one can view this approach as an "accelerator" that can speed up any SMT solver. The approach is also applicable to other theories; for instance, it has been successfully extended to finite-precision bit-vector arithmetic [43].

1.7.2. Handling quantifiers

Because the Nelson-Oppen framework (§1.6.1) only works for quantifier-free formulas, SMT solvers have traditionally been rather limited in their ability to rea-

son about quantifiers. A notable exception is the prover Simplify [59] which uses *quantifier instantiation* on top of a Nelson-Oppen style prover. Several modern SMT solvers have adopted and extended these techniques [73, 23, 105].

The basic idea can be understood by extending the abstract framework described in §1.4.4 to include rules for quantifier instantiation. The main modification is to also allow closed quantified formulas wherever atomic formulas are allowed. Define a *generalized atomic formula* as either an atomic formula or a closed quantified formula. A *generalized literal* is either a generalized atomic formula or its negation; a generalized clause is a disjunction of generalized literals. The modified abstract framework is obtained simply by allowing literals and clauses to be replaced by their generalized counterparts. For instance, non-fail states become pairs $M \parallel F$, where M is a sequence of generalized literals and F is a conjunction of generalized clauses. With this understanding, the two rules below can be used to model quantifier instantiation. For simplicity and without loss of generality, we assume here that quantified literals in M appear only positively and that the bodies of quantified formulas are themselves in abstract CNF.

$$\exists\text{-}\mathsf{Inst}: \quad M \parallel F \implies M \parallel F, \neg\exists\mathbf{x}.\varphi \vee \varphi[\mathbf{x}/\mathbf{a}] \quad \textbf{if} \begin{cases} \exists\mathbf{x}.\varphi \text{ is in } M \\ \mathbf{a} \text{ are fresh constants} \end{cases}$$

$$\forall\text{-}\mathsf{Inst}: \quad M \parallel F \implies M \parallel F, \neg\forall\mathbf{x}.\varphi \vee \varphi[\mathbf{x}/\mathbf{s}] \quad \textbf{if} \begin{cases} \forall\mathbf{x}.\varphi \text{ is in } M \\ \mathbf{s} \text{ are ground terms} \end{cases}$$

The ∃-Inst rule essentially models skolemization and the ∀-Inst rule models instantiation. It is also clear that termination can only be guaranteed by limiting the number of times the rules are applied. For a given existentially quantified formula, there is no benefit to applying ∃-Inst more than once, but a universally quantified formula may need to be instantiated several times with different ground terms before unsatisfiability is detected. The main challenge then, in applying these rules, is to come up with an effective strategy for applying ∀-Inst. For some background theories (e.g., universal theories) completeness can be shown for exhaustive and fair instantiation strategies. In practice, however, the most effective techniques are incomplete and heuristic.

The most common approach is to select for instantiation only ground terms that are *relevant* to the quantified formula in question, according to some heuristic relevance criteria. The idea is as follows: given a state $M \parallel F$ and an abstract literal $\forall x.\varphi$ in M, try to find a subterm t of $\forall x.\varphi$ properly containing x, a ground term g in M, and a subterm s of g, such that $t[x/s]$ is equivalent to g modulo the background theory $\mathcal{T}$ (written $t[x/s] =_{\mathcal{T}} g$). In this case, we expect that instantiating x with s is more likely to be helpful than instantiating with other candidate terms. Following Simplify's terminology, the term t is called a *trigger* (for $\forall x.\varphi$). In terms of unification theory, the case in which $t[x/s] =_{\mathcal{T}} g$ is a special case of *$\mathcal{T}$-matching* between t and g.

In general, in the context of SMT, given the complexity of the background theory $\mathcal{T}$, it may be very difficult if not impossible to determine whether a trigger and a ground term $\mathcal{T}$-match. As a result, most implementations use some form of $\mathcal{T}_{\mathcal{E}}$-matching. For details on effective implementation strategies, we refer the

reader to [59, 73, 23, 105].

1.7.3. Producing models

For some applications, it is desirable not only to know whether a formula is satisfiable, but if so, what a satisfying model is. In general, it may be challenging to capture all of the structure of an arbitrary first-order model. However, it is often sufficient to know how to assign values from some "standard" model to the ground terms in a formula. Several SMT solvers have implemented support for "models" of this kind.

One approach is to do additional work guessing values for ground terms and then to double-check that the formula is indeed satisfied. This is the approach followed by CVC3 [18].

An alternative approach is to instrument the $\mathcal{T}$*-solver* to continually maintain a value for every ground term associated with the theory. This is the strategy followed by the solvers YICES and Z3 [61, 55].

1.7.4. Producing proofs

In both SAT and SMT communities, the importance of having tools able to produce proofs of the ($\mathcal{T}$-)unsatisfiability of the input formulas has been recently stressed, due to the need for independently verifiable results, and as a starting point for the production of unsatisfiable cores (§1.7.5) and interpolants (§1.7.6).

A DPLL solver can be easily modified to generate a resolution proof of unsatisfiability for an input CNF formula, by keeping track of all resolution steps performed when generating each conflict clause, and by combining these subproofs iteratively into a proof of unsatisfiability whose leaves are original clauses. Techniques for translating a Boolean resolution proof into a more human-readable and verifiable format have been proposed, e.g., in [12, 2].

Similarly, a lazy SMT solver can be modified to generate a resolution proof of unsatisfiability for an input CNF formula, whose leaves are either clauses from the original formula or $\mathcal{T}$-lemmas (i.e. negations of $\mathcal{T}$-conflict sets and $\mathcal{T}$-deduction clauses returned by the $\mathcal{T}$-solver, see §1.4.3), which are $\mathcal{T}$-valid clauses. Such a resolution proof can be further refined by providing a proof for each $\mathcal{T}$-lemma in some theory-specific deductive framework (like, e.g. that in [102] for inequalities in $\mathcal{T}_{\mathcal{R}}$).

1.7.5. Identifying unsatisfiable cores

An *unsatisfiable core(UC)* of an unsatisfiable CNF formula φ is an unsatisfiable subset of the clauses in φ.

In SAT, the problem of finding small unsatisfiable cores has been addressed by many authors in recent years [95, 103, 159, 116, 82, 58, 29, 74, 156] due to its importance in formal verification [100]. In particular, lots of techniques and optimizations have been introduced with the aim of producing small [159, 74], minimal [116, 82, 58], or even minimum unsat cores [95, 103, 156].

In SMT, despite the fact that finding unsatisfiable cores has been addressed explicitly in the literature only recently [46], at least three SMT solvers (i.e.

CVC3, MathSAT, and Yices) support UC generation.[30] We distinguish three main approaches.

In the approach implemented in CVC3 and MathSAT (*proof-based approach* hereafter), the UC is produced as a byproduct of the generation of resolution proofs. As in [159], the idea is to return the set of clauses from the original problem which appear as leaves in the resolution proof of unsatisfiability. ($\mathcal{T}$-lemmas are not included as they are $\mathcal{T}$-valid clauses, so they play no role in the $\mathcal{T}$-unsatisfiability of the core.)

The approach used by Yices (*assumption-based approach* hereafter) is an adaptation of a method used in SAT [95]: for each clause C_i in the problem, a new Boolean "selector" variable S_i is created; then, each C_i is replaced by $(S_i \to C_i)$; finally, before starting the search each S_i is forced to true. In this way, when a conflict at decision level zero is found by the DPLL solver, the conflict clause C contains only selector variables, and the UC returned is the union of the clauses whose selectors appear in C. Neither approach aims at producing minimal or minimum unsatisfiable cores, nor does anything to reduce their size.

In the *lemma-lifting approach* [46] implemented in MathSAT, a lazy SMT solver is combined with an external (and possibly highly-optimized) propositional core extractor. The SMT solver stores and returns the $\mathcal{T}$-lemmas it had to prove in order to refute the input formula; the external core extractor is then called on the Boolean abstraction of the original SMT problem and of the $\mathcal{T}$-lemmas. Clauses corresponding to $\mathcal{T}$-lemmas are removed from the resulting UC, and the remaining abstract clauses are refined back into their original form. The result is an unsatisfiable core of the original SMT problem. This technique is able to benefit from any size-reduction techniques implemented in the Boolean core extractor used.

1.7.6. Computing interpolants

A *Craig interpolant* ("interpolant" hereafter) of a pair of formulas (ψ, ϕ) s.t. $\psi \wedge \phi \models_{\mathcal{T}} \bot$ is a formula ψ' s.t.:

- $\psi \models_{\mathcal{T}} \psi'$,
- $\psi' \wedge \phi \models_{\mathcal{T}} \bot$, and
- $\psi' \preceq \psi$ and $\psi' \preceq \phi$,

where $\alpha \preceq \beta$ denotes that all uninterpreted (in the signature of $\mathcal{T}$) symbols of α appear in β. Note that a quantifier-free interpolant exists if $\mathcal{T}$ admits quantifier elimination [85] (e.g., in $\mathcal{T}_{\mathcal{Z}}$ a quantifier-free interpolant for (ψ, ϕ) may not exist).

The use of interpolation in formal verification was introduced by McMillan in [101] for purely-propositional formulas, and it was subsequently extended to handle $\mathcal{T}_{\mathcal{E}} \cup \mathcal{T}_{\mathcal{R}}$-formulas in [102]. The technique, which is based on earlier work by Pudlák [124], works as follows. (We assume w.l.o.g. that the formulas are in CNF.) An interpolant for (ψ, ϕ) is constructed from a resolution proof $\mathcal{P}$ of the unsatisfiability of $\psi \wedge \phi$, according to the following steps:

[30] Apart from [46], the information reported here on the computation of unsatisfiable cores in these tools comes from personal knowledge of the tools (CVC3 and MathSAT) and from private communications from the authors (Yices).

1. for every clause C occurring as a leaf in $\mathcal{P}$, set $I_C \equiv C \downarrow \phi$ if $C \in \psi$, and $I_C \equiv \top$ if $C \in \phi$;
2. for every $\mathcal{T}$-lemma $\neg\eta$ occurring as a leaf in $\mathcal{P}$, generate an interpolant $I_{\neg\eta}$ for $(\eta \setminus \phi, \eta \downarrow \phi)$;
3. for every node C of $\mathcal{P}$ obtained by resolving $C_1 \equiv p \vee \phi_1$ and $C_2 \equiv \neg p \vee \phi_2$, set $I_C \equiv I_{C_1} \vee I_{C_2}$ if p does not occur in ϕ, and $I_C \equiv I_{C_1} \wedge I_{C_2}$ otherwise;
4. return $I_\perp$ as an interpolant for (ψ, ϕ);

where $C \downarrow \phi$ is the clause obtained by removing all the literals in C whose atoms do not occur in ϕ, and $C \setminus \phi$ that obtained by removing all the literals whose atoms do occur in ϕ. In the purely-Boolean case the algorithm reduces to steps 1., 3. and 4. only. Notice that step 2 of the algorithm is the only part which depends on the theory $\mathcal{T}$, so that the problem reduces to that of finding interpolants for (negated) $\mathcal{T}$-lemmas.

A number of techniques exist for theory-specific interpolation generation. For example, [102] provides a set of rules for constructing interpolants for $\mathcal{T}$-lemmas in $\mathcal{T}_\mathcal{E}$, for weak linear inequalities $(0 \leq t)$ in $\mathcal{T}_\mathcal{R}$, and their combination. [154] uses a variant of the Nelson-Oppen procedure (§1.6.2) for generating interpolants for $\mathcal{T}_1 \cup \mathcal{T}_2$ using the interpolant-generation procedures of $\mathcal{T}_1$ and $\mathcal{T}_2$ as black-boxes. The combination $\mathcal{T}_\mathcal{E} \cup \mathcal{T}_\mathcal{R}$ can also be used to compute interpolants for other theories, such as those of lists, arrays, sets and multisets [85]. [129] computes interpolants for $\mathcal{T}$-lemmas in $\mathcal{T}_\mathcal{E} \cup \mathcal{T}_\mathcal{R}$ by solving a system of Linear Programming (LP) constraints. [88] extends the *eager* SMT approach to the generation of interpolants. (The approach is currently limited to the theory of equality only, without uninterpreted functions).) [47] presents some extensions to the work in [102], including an optimized interpolant generator for the full theory $\mathcal{T}_\mathcal{R}$, an ad hoc interpolant generator for difference logic, and an interpolant combination method based on Delayed Theory Combination (§1.6.3).

Bibliography

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

[2] H. Amjad. A Compressing Translation from Propositional Resolution to Natural Deduction. In *Proc. FroCoS*, volume 4720 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 2007.

[3] 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 B. Gramlich, editor, *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2005.

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

[5] 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.

[6] 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.

[7] A. Armando, S. Ranise, and M. Rusinowitch. A Rewriting Approach to Satisfiability Procedures. *Information and Computation*, 183(2):140–164, June 2003.

[8] 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.

[9] 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.

[10] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003.

[11] T. Ball, B. Cook, S. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Proc. Computer-Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 457–461, 2004.

[12] C. Barrett and S. Berezin. A proof-producing boolean search engine. In *Proceedings of the* 1st *International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '03)*, July 2003.

[13] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *Proceedings of the* 16th *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer, July 2004.

[14] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In M. Hermann and A. Voronkov, editors, *Proceedings of the* 13th *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '06)*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer, Nov. 2006.

[15] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT Modulo Theories. Technical Report 06-05, Department of Computer Science, University of Iowa, Aug. 2006.

[16] C. Barrett, S. Ranise, C. Tinelli, and A. Stump. The SMT-LIB web site, 2008.

[17] C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for a theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:21–46, 2007.

[18] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19th *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, July 2007.

[19] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the* 35th *Design Automation Conference (DAC '98)*, pages 522–527. Association for Computing Machinery, June 1998.

[20] C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the* 14th *International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer-Verlag, July 2002.

[21] C. W. Barrett, D. L. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In A. Armando, editor, *Proceedings of the* 4th *International Workshop on Frontiers of Combining Systems (FroCoS '02)*, volume 2309 of *Lecture Notes in Artificial Intelligence*, pages 132–146. Springer, Apr. 2002.

[22] C. W. Barrett, D. L. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In A. Armando, editor, *Proceedings of the 4th International Workshop on Frontiers of Combining Systems, FroCoS'2002 (Santa Margherita Ligure, Italy)*, volume 2309 of *Lecture Notes in Computer Science*, pages 132–147, apr 2002.

[23] N. Bjørner and L. de Moura. Efficient E-matching for SMT solvers. In F. Pfenning, editor, *Proceedings of the* 21st *International Conference on Automated Deduction (CADE '07)*, volume 4603 of *Lecture Notes in Arti-*

ficial Intelligence, pages 183–198. Springer-Verlag, July 2007.

[24] N. Bjørner and M. C. Pichora. Deciding fixed and non-fixed size bit-vectors. In *TACAS '98: Proceedings of the* 4th *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 376–392. Springer-Verlag, 1998.

[25] M. P. Bonacina and M. Echenim. Rewrite-based decision procedures. In M. Archer, T. B. de la Tour, and C. Munoz, editors, *Proceedings of the Sixth Workshop on Strategies in Automated Deduction (STRATEGIES)*, volume 174(11) of *Electronic Notes in Theoretical Computer Science*, pages 27–45. Elsevier, July 2007.

[26] M. P. Bonacina and M. Echenim. T-decision by decomposition. In F. Pfenning, editor, *Proceedings of the Twenty-first International Conference on Automated Deduction (CADE)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 199–214. Springer, July 2007.

[27] 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 U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 513–527. Springer, 2006.

[28] 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 U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Auto mated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Computer Science*, pages 513–527, Seattle (WA, USA), 2006. Springer.

[29] Booleforce, `http://fmv.jku.at/booleforce/`.

[30] I. Borosh, M. Flahive, and L. B. Treybig. Small solutions of linear Diophantine equations. *Discrete Mathematics*, 58:215–220, 1986.

[31] I. Borosh and L. B. Treybig. Bounds on positive integral solutions of linear Diophantine equations. *Proceedings of the American Mathematical Society*, 55(2):299–304, March 1976.

[32] R. S. Boyer and J. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. *Machine Intelligence*, 11:83–124, 1988.

[33] R. S. Boyer and J. S. Moore. A theorem prover for a computational logic. In M. E. Stickel, editor, *10th International Conference on Automated Deduction (CADE)*, LNAI 449, pages 1–15, Kaiserslautern, FRG, July 24–27, 1990. Springer-Verlag.

[34] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. 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.

[35] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. 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.

[36] 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.

[37] 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.

[38] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19th *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 547–560. Springer-Verlag, July 2007.

[39] 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.

[40] 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.

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

[42] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.

[43] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2007.

[44] R. E. Bryant and M. N. Velev. Boolean satisfiability with transitivity constraints. In A. Emerson and P. Sistla, editors, *Computer-Aided Verification (CAV 2000)*, LNCS 1855. Springer-Verlag, July 2000.

[45] V. Chandru. Variable elimination in linear constraints. *The Computer Journal*, 36(5):463–472, Aug. 1993.

[46] A. Cimatti, A. Griggio, and R. Sebastiani. A Simple and Flexible Way of Computing Small Unsatisfiable Cores in SAT Modulo Theories. In *Proc. SAT'07*, volume 4501 of *LNCS*. Springer, 2007.

[47] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Theories. In *Proc. TACAS'08*, volume 4963 of *LNCS*. Springer, 2008.

[48] Conchon and Krstic. Strategies for combining decision procedures. *Theoretical Computer Science*, 354, 2006.

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

[50] S. Cotton and O. Maler. Satisfiability modulo theory chains with DPLL(T). Unpublished. Available from http://www-verimag.imag.fr/˜maler/, 2006.

[51] W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[52] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. McRobbie and J. Slaney, editors, *13th International Conference on Computer Aided Deduction*, volume 1104 of *Lecture Notes in Computer Science*, pages 463–477. Springer, 1996.

[53] D. Cyrluk, M. O. Möller, and H. Ruess. An efficient decision procedure for the theory of fixed-size bit-vectors. In *Proceedings of the* 9th *International Conference on Computer Aided Verification (CAV '97)*, pages 60–71. Springer, 1997.

[54] J. H. Davenport and J. Heintz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5:29–35, 1988.

[55] L. de Moura and N. Bjorner. Model-based Theory Combination. In *Proc. 5th workshop on Satisfiability Modulo Theories, SMT'07*, 2007.

[56] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, 2008.

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

[58] N. Dershowitz, Z. Hanna, and A. Nadel. A Scalable Algorithm for Minimal Unsatisfiable Core Extraction. In *Proc. SAT'06*, volume 4121 of *LNCS*. Springer, 2006.

[59] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003.

[60] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the Association for Computing Machinery*, 27(4):758–771, Oct. 1980.

[61] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In T. Ball and R. B. Jones, editors, *Proceedings of the* 18th *International Conference on Computer Aided Verification (CAV '06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, Aug. 2006.

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

[63] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical logic*. Undergraduate Texts in Mathematics. Springer, New York, second edition edition, 1994.

[64] J. Elgaard, N. Klarlund, and A. Möller. Mona 1.x: New techniques for WS1S and WS2S. In *Proceedings of the* 10th *International Conference on Computer Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.

[65] H. B. Enderton. *A Mathematical Introduction to Logic*. Undergraduate Texts in Mathematics. Academic Press, second edition edition, 2000.

[66] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated can-

onizer and solver. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, 2001.

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

[68] P. Fontaine and E. P. Gribomont. Combining non-stably infinite, non-first order theories. In C. Tinelli and S. Ranise, editors, *Proceedings of the IJCAR Workshop on Pragmatics of Decision Procedures in Automated Deduction, PDPAR*, July 2004.

[69] P. Fontaine, S. Ranise, and C. G. Zarba. Combining lists with non-stably infinite theories. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 3452 of *Lecture Notes in Computer Science*, pages 51–66. Springer-Verlag, 2005.

[70] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19th *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer-Verlag, July 2007.

[71] H. Ganzinger. Shostak light. In A. Voronkov, editor, *Proceedings of the* 18th *International Conference on Computer-Aided Deduction (CADE '02)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 332–346. Springer, July 2002.

[72] 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.

[73] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In F. Pfenning, editor, *Proceedings of the* 21st *International Conference on Automated Deduction (CADE '07)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 167–182. Springer-Verlag, July 2007.

[74] R. Gershman, M. Koifman, and O. Strichman. Deriving Small Unsatisfiable Cores with Dominators. In *Proc. CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[75] S. Ghilardi. Model theoretic methods in combined constraint satisfiability. *Journal of Automated Reasoning*, 3(3–4):221–249, 2004.

[76] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Deciding extensions of the theory of arrays by integrating decision procedures and instantiation strategies. In M. Fisher, W. van der Hoek, B. Konev, and A. Lisitsa, editors, *Logics in Artificial Intelligence, 10th European Conference, JELIA 2006, Liverpool, UK, September 13-15, 2006, Proceedings*, volume 4160 of *Lecture Notes in Computer Science*, pages 177–189. Springer, 2006.

[77] S. Ghilardi, E. Nicolini, and D. Zucchelli. A comprehensive framework for combined decision procedures. In B. Gramlich, editor, *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2005.

[78] E. Giunchiglia, F. Giunchiglia, and A. Tacchella. SAT Based Decision Procedures for Classical Modal Logics. Journal of Automated Reasoning. Spe-

cial Issue: Satisfiability at the start of the year 2000, 2001.

[79] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. CADE'13*, number 1104 in LNAI. Springer, 1996.

[80] D. Hochbaum, N. Megiddo, J. Naor, and A. Tamir. Tight bounds and 2-approximation algorithms for integer programs with two variables per inequality. *Mathematical Programming*, 62:63–92, 1993.

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

[82] J. Huang. MUP: a minimal unsatisfiability prover. In *Proc. ASP-DAC '05*. ACM Press, 2005.

[83] J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap. Beyond finite domains. In *2nd International Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, volume 874 of *Lecture Notes in Computer Science*, pages 86–94, 1994.

[84] R. Kannan and C. L. Monma. On the computational complexity of integer programming problems. In *Optimisation and Operations Research*, volume 157 of *Lecture Notes in Economics and Mathematical Systems*, pages 161–172. Springer-Verlag, 1978.

[85] D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for data structures. In *Proc. SIGSOFT FSE*. ACM, 2006.

[86] N. Karmakar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[87] D. Kroening, J. Ouaknine, S. A. Seshia, and O. Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In *Proc. 16th International Conference on Computer-Aided Verification (CAV)*, pages 308–320, July 2004.

[88] D. Kroening and G. Weissenbacher. Lifting propositional interpolants to the word-level. In *Proceedings of FMCAD*, pages 85–89. IEEE, 2007.

[89] S. Krstic and S. Conchon. Canonization for disjoint unions of theories. In F. Baader, editor, *Proceedings of the* 19th *International Conference on Computer-Aided Deduction (CADE '03)*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 197–211. Springer, Aug. 2003. Miami Beach, FL.

[90] S. Krstić, A. Goel, J. Grundy, and C. Tinelli. Combined satisfiability modulo parametric theories. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Braga, Portugal)*, volume 4424 of *Lecture Notes in Computer Science*, pages 618–631. Springer, 2007.

[91] S. K. Lahiri and R. E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *Proc. 15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 341–354, 2003.

[92] S. K. Lahiri, R. E. Bryant, A. Goel, and M. Talupur. Revisiting positive equality. In *Proc. Tools and Algoriths for the Construction and Analysis of Systems (TACAS)*, LNCS 2988, pages 1–15, 2004.

[93] S. K. Lahiri and M. Musuvathi. An efficient decision procedure for UTVPI

constraints. In J. G. Carbonell and J. Siekmann, editors, *Proceedings of the 5th International Workshop on Frontiers of Combining Systems (FroCos '05)*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 168–183. Springer, Sept. 2005.

[94] S. K. Lahiri and S. A. Seshia. The UCLID decision procedure. In *Proc. 16th International Conference on Computer-Aided Verification (CAV)*, pages 475–478, July 2004.

[95] I. Lynce and J. P. M. Silva. On computing minimum unsatisfiable cores. In *SAT*, 2004.

[96] Z. Manna and C. Zarba. Combining decision procedures. In *Formal Methods at the Crossroads: from Panacea to Foundational Support*, volume 2787 of *Lecture Notes in Computer Science*, pages 381–422. Springer, Nov. 2003.

[97] Z. Manna and C. G. Zarba. Combining decision procedures. In *Formal Methods at the Cross Roads: From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 381–422. Springer, 2003.

[98] F. Maric and P. Janicic. ARGO-Lib: A generic platform for decision procedures. In *Proceedings of IJCAR '04*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 213–217. Springer, 2004.

[99] Y. V. Matiyasevich. Diophantine representation of recursively enumerable predicates. In J. E. Fenstad, editor, *Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 171–177. North-Holland Publishing Company, 1971.

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

[101] K. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, 2003.

[102] K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1), 2005.

[103] M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. P. M. Silva, and K. A. Sakallah. A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas. In *Proc. SAT'05*, volume 3569 of *LNCS*. Springer, 2005.

[104] M. O. Möller. *Solving Bit-Vector Equations – a Decision Procedure for Hardware Verification.* PhD thesis, University of Ulm, 1997.

[105] M. Moskałand J. Łopuszański. Fast quantifier reasoning with lazy proof explication. Technical report, Institute of Computer Science, University of Wrocław, May 2006.

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

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

[108] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, Apr. 1980.

[109] 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.

[110] R. Nieuwenhuis and A. Oliveras. Decision Procedures for SAT, SAT Modulo Theories and Beyond. The BarcelogicTools. (Invited Paper). In G. Sutcliffe and A. Voronkov, editors, *12h International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'05*, volume 3835 of *Lecture Notes in Computer Science*, pages 23–46. Springer, 2005.

[111] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In K. Etessami and S. K. Rajamani, editors, *Proceedings of the* 17th *International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer, July 2005.

[112] 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.

[113] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04), Montevideo, Uruguay*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

[114] 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, Nov. 2006.

[115] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.

[116] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. Amuse: A Minimally-Unsatisfiable Subformula Extractor. In *Proc. DAC'04*. ACM/IEEE, 2004.

[117] D. C. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, 12:291–302, 1980.

[118] D. C. Oppen. Reasoning about recursively defined data structures. *Journal of the Association for Computing Machinery*, 27(3):403–411, July 1980.

[119] C. H. Papadimitriou. On the complexity of integer programming. *Journal of the Association for Computing Machinery*, 28(4):765–768, 1981.

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

[121] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In *Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 455–469. Springer, 1999.

[122] A. Pnueli, Y. Rodeh, M. Siegel, and O. Strichman. The small model property: How small can it be? *Journal of Information and Computation*, 178(1):279–293, Oct. 2002.

[123] V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977. Cambridge, MA.

[124] P. Pudlák. Lower bounds for resolution and cutting planes proofs and

monotone computations. *J. of Symbolic Logic*, 62(3), 1997.

[125] W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[126] S. Ranise, C. Ringeissen, and C. G. Zarba. Combining data structures with nonstably infinite theories using many-sorted logic. In B. Gramlich, editor, *Proceedings of the Workshop on Frontiers of Combining Systems*, volume 3717 of *Lecture Notes in Computer Science*, pages 48–64. Springer, 2005.

[127] C. Ringeissen. Cooperation of decision procedures for the satisfiability problem. In F. Baader and K. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop, Munich (Germany)*, Applied Logic, pages 121–140. Kluwer Academic Publishers, Mar. 1996.

[128] H. Ruess and N. Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28. IEEE Computer Society, June 2001. Boston, MA.

[129] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. In *VMCAI*, LNCS. Springer, 2007.

[130] R. Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation – JSAT.*, 3, 2007.

[131] S. A. Seshia. *Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification*. PhD thesis, Carnegie Mellon University, 2005.

[132] S. A. Seshia and R. E. Bryant. Deciding quantifier-free Presburger formulas using parameterized solution bounds. *Logical Methods in Computer Science*, 1(2):1–26, December 2005.

[133] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *40th Design Automation Conference (DAC '03)*, pages 425–430, June 2003.

[134] S. A. Seshia, K. Subramani, and R. E. Bryant. On solving boolean combinations of UTVPI constraints. *Journal of Satisfiability, Boolean Modeling, and Computation (JSAT)*, 3:67–90, 2007.

[135] N. Shankar and H. Rueß. Combining Shostak theories. In S. Tison, editor, *Int'l Conf. Rewriting Techniques and Applications (RTA '02)*, volume 2378 of *LNCS*, pages 1–18. Springer, 2002.

[136] H. M. Sheini and K. A. Sakallah. A scalable method for solving satisfiability of integer linear arithmetic logic. In F. Bacchus and T. Walsh, editors, *Proceedings of the* 8th *International Conference on Theory and Applications of Satisfiability Testing (SAT '05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 241–256. Springer, June 2005.

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

[138] R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.

[139] R. E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7), 1978.

[140] R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, Apr. 1979.

[141] O. Strichman. On solving Presburger and linear arithmetic with SAT. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517,

pages 160–170. Springer-Verlag, November 2002.

[142] O. Strichman. Optimizations in decision procedures for propositional linear inequalities. Technical Report CMU-CS-02-133, Carnegie Mellon University, 2002.

[143] O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In E. Brinksma and K. G. Larsen, editors, *Proc. 14th Intl. Conference on Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 209–222. Springer-Verlag, July 2002.

[144] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A cooperating validity checker. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the* 14th *International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer, July 2002.

[145] A. Stump, D. L. Dill, C. W. Barrett, and J. Levitt. A decision procedure for an extensional theory of arrays. In *Proceedings of the* 16th *IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 29–37. IEEE Computer Society, June 2001.

[146] M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range allocation for separation logic. In *Proc. Computer-Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 148–161, 2004.

[147] C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In F. Baader and K. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop (Munich, Germany)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, Mar. 1996.

[148] C. Tinelli and C. Ringeissen. Unions of non-disjoint theories and combinations of satisfiability procedures. *Theoretical Computer Science*, 290(1):291–353, Jan. 2003.

[149] C. Tinelli and C. Zarba. Combining decision procedures for sorted theories. In J. Alferes and J. Leite, editors, *Proceedings of the 9th European Conference on Logic in Artificial Intelligence (JELIA'04), Lisbon, Portugal*, volume 3229 of *Lecture Notes in Artificial Intelligence*, pages 641–653. Springer, 2004.

[150] C. Tinelli and C. Zarba. Combining nonstably infinite theories. *Journal of Automated Reasoning*, 34(3):209–238, Apr. 2005.

[151] J. von zur Gathen and M. Sieveking. A bound on solutions of linear integer equalities and inequalities. *Proceedings of the American Mathematical Society*, 72(1):155–158, October 1978.

[152] 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.

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

[154] G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In *CADE*, volume 3632 of *LNCS*. Springer, 2005.

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

[156] J. Zhang, S. Li, and S. Shen. Extracting Minimum Unsatisfiable Cores with a Greedy Genetic Algorithm. In *Proc. ACAI*, volume 4304 of *LNCS*. Springer, 2006.

[157] 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.

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

[159] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *Proc. of SAT*, 2003.