Satisfiability Modulo Theories

Clark Barrett and Cesare Tinelli

Abstract Satisfiability Modulo Theories (SMT) refers to the problem of determining whether a first-order formula is satisfiable with respect to some logical theory. Solvers based on SMT are used as back-end engines in model checking applications such as bounded, interpolation-based, and predicate abstraction-based model checking. After a brief illustration of these uses, we survey the predominant techniques for solving SMT problems with an emphasis on the *lazy* approach, in which a propositional satisfiability (SAT) solver is combined with one or more *theory solvers*. We discuss the architecture of a lazy SMT solver, give examples of theory solvers, show how to combine such solvers modularly, and mention several extensions of the lazy approach. We also briefly describe the *eager* approach in which the SMT problem is reduced to a SAT problem. Finally, we discuss how the basic framework for determining satisfiability can be extended with additional functionality such as producing models, proofs, unsatisfiable cores, and interpolants.

1 Introduction

In several areas of computer science, including formal verification of hardware and software, many important problems can be reduced to checking the satisfiability of a formula in some logic. Several of these problems can be naturally formulated as satisfiability problems in propositional logic and solved very efficiently by modern SAT solvers, as described in Chap. 5 of this book. Other problems are formulated more naturally and compactly in classical logics, such as first-order or higher-order

Clark Barrett
Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, e-mail: barrett@cs.nyu.edu

Cesare Tinelli
Department of Computer Science, The University of Iowa, 14 MacLean Hall, Iowa City, IA 52241, e-mail: cesare-tinelli@uiowa.edu

logics, with a more expressive language that includes non-Boolean variables, function and predicate symbols (with positive arity) and quantifiers. There is, of course, a trade-off between the expressiveness of a logic and the ability to automatically check the satisfiability of its formulas.

A practical compromise can be achieved with fragments of first-order logic that are restricted either syntactically, for instance by allowing only certain classes of formulas, or semantically, by constraining the interpretation of certain function and predicate symbols, or both. Such restrictions can make the satisfiability problem decidable and, more importantly, allow the development of specialized satisfiability procedures that exploit properties of the fragment to great advantage for practical efficiency, even in cases with high worst-case computational complexity. When semantic restrictions are involved, they can be understood as limiting the interpretations of certain symbols to models of some logical *background theory* (e.g., the theory of equality, of integer numbers, of real numbers, of arrays, of lists, and so on). In those cases, we speak of *Satisfiability Modulo Theories* (*SMT*).[1]

Building on classical results on decision procedures for first-order reasoning, and on the tremendous advances in SAT solving technology in the last two decades, SMT has grown in recent years into a very active research field whose defining feature is the use of reasoning methods specific to logical theories of interest in target applications. Thanks to advances in SMT research and technology, there are now several powerful and sophisticated SMT solvers (e.g., Alt-Ergo [25], Beaver [98], Boolector [35], CVC4 [10], MathSAT5 [51], openSMT [38], SMTInterpol [49], SONOLAR [132], STP [79], veriT [28], Yices [72], and Z3 [62]) which are being used in a rapidly expanding set of applications. Application areas currently include processor verification, equivalence checking, bounded and unbounded model checking, predicate abstraction, static analysis, automated test case generation, extended static checking, type checking, planning, scheduling, and optimization.

The recent progress in SMT has been driven by several factors, including: a focus on background theories and classes of problems that occur in practice; liftings and adaptations of SAT technology to the SMT case; innovations in core algorithms and data structures; development of abstract constraint solving frameworks and general solver architectures to guide efficient implementations; novel search heuristics; and attention to implementation details.[2] A major enabler of this progress has been SMT-LIB [14], a standardization and benchmark collection initiative collectively developed and supported by the SMT community, together with its derivative activities: the SMT workshop, an international forum for SMT researchers and users of SMT applications or techniques; SMT-COMP [18], an international competition for SMT solvers supporting the SMT-LIB input/output format [15]; and SMT-EXEC a public execution service allowing researchers to run experimental evaluations on SMT solvers.[3]

[1] This terminology originated in [153] and was popularized by the SMT-LIB initiative [14].

[2] It is worth noting that many of the same factors are driving improvements in modern SAT research (see [22] as well as Chap. 5 of this book).

[3] http://smtlib.org , http://smt-workshop.org , http://www.smtexec.org .

This chapter provides a fairly high-level overview of SMT and its main results and techniques, together with references to the relevant literature for a deeper study. It concentrates mostly on the predominant approach for implementing SMT solvers, known as the "lazy approach." Its essence consists of combining an efficient and properly instrumented SAT solver with one or more *theory solvers*, highly specialized solvers for problems consisting just of conjunctions of *theory literals*—atomic and negated atomic formulas in the language of some particular theory T.

The chapter is structured as follows. The rest of this section provides technical background information, defining basic notions and terminology used throughout the chapter. Section 3 describes the lazy approach to SMT in which a SAT solver and a theory solver cooperate to solve an SMT problem. Section 4 discusses theory solvers for a number of background theories used in SMT applications, and specifically in model checking. Section 5 focuses on techniques for combining theory solvers for different theories into a solver for a combination of those theories. Section 6 discusses a few extensions and enhancements to the lazy approach. Section 7 describes an alternative to the lazy approach for SMT, aptly named the "eager approach," which takes advantage of SAT solvers more directly. Finally, Section 8 presents a number of important functionalities provided by modern SMT solvers that go beyond mere satisfiability checking, and that have been crucial to the success of SMT as an enabling technology in applications like model checking.

1.1 Technical Preliminaries

SMT problems are formulated within first-order logic with equality. Since many applications of SMT involve different data types, it is more convenient to work with a sorted (i.e., typed) version of that logic, as opposed to the classical unsorted version. In this chapter we use a basic version of *many-sorted* logic [74, 117], which is adequate for our purposes. More sophisticated typed logics are sometimes used in the literature. For instance, the SMT-LIB 2 standard is based on a sorted logic with non-nullary sort symbols and *let* binders [16]. Other work adopts, and advocates for, a first-order logic with parametric (universal) types [25, 111, 112].

Syntax We fix an infinite set $\mathbf{S}$ of *sort symbols* and consider an infinite set $\mathbf{X}$ of *(sorted) variables*, each uniquely associated with a sort in $\mathbf{S}$. A many-sorted *signature* Σ consists of a set $\Sigma^{\mathrm{S}} \subseteq S$ of sort symbols; a set Σ^{P} of *predicate symbols*; a set Σ^{F} of *function symbols*; a total mapping from Σ^{P} to the set $(\Sigma^{\mathrm{S}})^*$ of strings over Σ^{S}; and a total mapping from Σ^{F} to the set $(\Sigma^{\mathrm{S}})^+$ of non-empty strings over Σ^{S}—where * and $^+$ are the usual regular expression operators. For $n \geq 0$, a function symbol f (resp., predicate symbol p) has a unique[4] *arity* n and *rank* $\sigma_1 \cdots \sigma_n \sigma$ (resp., $\sigma_1 \cdots \sigma_n$) in Σ if it is mapped to the sort sequence $\sigma_1 \cdots \sigma_n \sigma$ (resp., $\sigma_1 \cdots \sigma_n$). When n above is 0, f is also called a *constant symbol (of sort σ)* and p a *propositional symbol*. A signature Σ is a *subsignature* of a signature Ω, written $\Sigma \subseteq \Omega$, and

[4] For simplicity, we do not allow any form of symbol overloading here.

Ω is a *supersignature* of Σ, if $\Sigma^{\mathrm{S}} \subseteq \Omega^{\mathrm{S}}$, $\Sigma^{\mathrm{F}} \subseteq \Omega^{\mathrm{F}}$, $\Sigma^{\mathrm{P}} \subseteq \Omega^{\mathrm{P}}$, and every function or predicate symbol of Σ has the same rank in Σ as in Ω.

A *(Σ-)term of sort σ* is either a sorted variable x of sort $\sigma \in \Sigma^{\mathrm{S}}$ or an expression of the form $f(t_1,\ldots,t_n)$ with $n \geq 0$ where $f \in \Sigma^{\mathrm{F}}$ with rank $\sigma_1 \cdots \sigma_n \sigma$ and t_i is a term of sort σ_i for $i = 1,\ldots,n$. An *atomic (Σ-)formula* is either the symbol $\bot$, for falsity; an expression of the form $t_1 = t_2$ with t_1, t_2 terms of the same sort;[5] or an expression of the form $p(t_1,\ldots,t_n)$ with $n \geq 0$ where $p \in \Sigma^{\mathrm{P}}$ with rank $\sigma_1 \cdots \sigma_n$ and t_i is a Σ-term of sort σ_i for $i = 1,\ldots,n$. A *(Σ-)literal* is an atomic Σ-formula or an expression $\neg\varphi$ where φ is an atomic Σ-formula. A *(Σ-)formula* is an atomic Σ-formula or an expression of the form $\neg\varphi$, $\varphi \vee \psi$, or $\exists x\, \varphi$ where x is a variable with sort in Σ^{S} and φ and ψ are Σ-formulas. We will write $\exists x{:}\sigma\, \varphi$ instead of $\exists x\, \varphi$ to indicate that x has sort σ. The other logical connectives as well as the universal quantifier can be formally defined in terms of the logical symbols above as usual (e.g., $\varphi \Rightarrow \psi$ as a shorthand for $\neg\varphi \vee \psi$; $\forall x\, \varphi$ as a shorthand for $\neg\exists x \neg\varphi$; and so on). Examples of signatures and formulas used in SMT are provided in Section 4.

Free occurrences of a variable in a formula are defined as usual: all variable occurrences in atomic formulas are free; a variable x distinct from a variable y occurs free in a formula $\neg\varphi$, $\varphi_1 \vee \varphi_2$, or $\exists y.\psi$ iff it occurs free respectively in φ, in φ_1 or φ_2, or in ψ. A *(Σ-)sentence* is a Σ-formula with no free variables. If φ is a Σ-formula and $\mathbf{x} = (x_1,\ldots,x_n)$ a tuple of distinct variables, we will write $\varphi[\mathbf{x}]$ or $\varphi[x_1,\ldots,x_n]$ to express that the free variables of φ are in $\mathbf{x}$; furthermore, if $t_1,\ldots,t_n$ are terms with each t_i of the same sort as x_i, we will write $\varphi[t_1,\ldots,t_n]$ to denote the formula obtained from $\varphi[x_1,\ldots,x_n]$ by simultaneously replacing each occurrence of x_i in φ by t_i, for $i = 1,\ldots,k$.

Semantics For each signature Σ and set $X \subseteq \mathbf{X}$ of variables whose sorts are in Σ^{S}, a *Σ-interpretation $\mathscr{A}$ over X* maps

- each sort $\sigma \in \Sigma^{\mathrm{S}}$ to a non-empty set A_σ, the *domain* of σ in $\mathscr{A}$;
- each variable $x \in X$ of sort σ to an element $x^{\mathscr{A}} \in A_\sigma$;
- each function symbol $f \in \Sigma^{\mathrm{F}}$ of rank $\sigma_1 \cdots \sigma_n \sigma$ to a total function $f^{\mathscr{A}} : A_{\sigma_1} \times \cdots \times A_{\sigma_n} \to A_\sigma$ (and in particular each constant c of sort σ to a $c^{\mathscr{A}} \in A_\sigma$),
- each predicate symbol $p \in \Sigma^{\mathrm{P}}$ of rank $\sigma_1 \cdots \sigma_n$ to a relation $p^{\mathscr{A}} \subseteq A_{\sigma_1} \times \cdots \times A_{\sigma_n}$.

A *Σ-model* is a Σ-interpretation over an empty set of variables. Let $\mathscr{A}$ be an Ω-interpretation over some set Y of variables. When $\Sigma \subseteq \Omega$ and $X \subseteq Y$, we denote by $\mathscr{A}^{\Sigma,X}$ the *reduct* of $\mathscr{A}$ to (Σ, X), i.e., the Σ-interpretation over X obtained from $\mathscr{A}$ by restricting it to interpret only the symbols in Σ and the variables in X. $\mathscr{A}$ is an *expansion* of a Σ-interpretation $\mathscr{B}$ over X if $\mathscr{B} = \mathscr{A}^{\Sigma,X}$.

Every Σ-interpretation $\mathscr{A}$ over some $X \subseteq \mathbf{X}$ induces a unique mapping $(\_)^{\mathscr{A}}$ from Σ-terms $f(t_1,\ldots,t_n)$ with variables in X to elements of sort domains such that $(f(t_1,\ldots,t_n))^{\mathscr{A}} = f^{\mathscr{A}}(t_1^{\mathscr{A}},\ldots,t_n^{\mathscr{A}})$. We define a satisfiability relation $\models$ between such interpretations and Σ-formulas with variables in X inductively as follows:

[5] We will use $=$ also to denote equality at the meta-level, relying on the context for disambiguation.

$$\begin{array}{lll}
\mathscr{A} \not\models \bot & & \\
\mathscr{A} \models t_1 = t_2 & \text{iff} & t_1^{\mathscr{A}} = t_n^{\mathscr{A}} \\
\mathscr{A} \models p(t_1,\dots,t_n) & \text{iff} & (t_1^{\mathscr{A}},\dots,t_n^{\mathscr{A}}) \in p^{\mathscr{A}} \\
\mathscr{A} \models \neg\varphi & \text{iff} & \mathscr{A} \not\models \varphi \\
\mathscr{A} \models \varphi \vee \psi & \text{iff} & \mathscr{A} \models \varphi \text{ or } \mathscr{A} \models \psi \\
\mathscr{A} \models \exists x{:}\sigma\, \varphi & \text{iff} & \mathscr{A}[x \mapsto a] \models \varphi \text{ for some } a \in \mathscr{A}_\sigma
\end{array}$$

where $\mathscr{A}[x \mapsto a]$ denotes the Σ-interpretation that maps x to a and is otherwise identical to $\mathscr{A}$. A Σ-interpretation $\mathscr{A}$ *satisfies* a Σ-formula φ if $\mathscr{A} \models \varphi$. A set Φ of Σ-formulas *entails* a Σ-formula φ, written $\Phi \models \varphi$, iff every Σ-interpretation that satisfies all formulas in Φ satisfies φ as well. The set Φ is *satisfiable* iff $\Phi \not\models \bot$, and φ is *valid* iff it is entailed by the empty set.

Theories In SMT, one is not interested in arbitrary models but in models belonging to a given *theory* T constraining the interpretation of the symbols in some signature Σ. We define theories most generally as classes of models with the same signature. More precisely, a *Σ-theory* T is a pair $(\Sigma, \mathbf{A})$ where Σ is a signature and $\mathbf{A}$ is a class (in the sense of set theory) of Σ-models. Section 4 discusses several examples of theories commonly used in SMT.

Let $T = (\Sigma, \mathbf{A})$ be a Σ-theory. A *T-interpretation* is any Ω-interpretation $\mathscr{A}$ for some $\Omega \supseteq \Sigma$ such that $\mathscr{A}^{\Sigma,\emptyset} \in \mathbf{A}$. A formula φ is *satisfiable in T*, or *T-satisfiable*, if it is satisfied by some T-interpretation $\mathscr{A}$.[6] A set Φ of Ω-formulas *T-entails* an Ω-formula φ, written $\Phi \models_T \varphi$, iff every T-interpretation that satisfies all formulas in Φ satisfies φ as well. The set Φ is *T-satisfiable* iff $\Phi \not\models_T \bot$, and φ is *T-valid* iff φ is T-entailed by the empty set, written as $\models_T \varphi$. *T-unsatisfiable* abbreviates not T-satisfiable. These notions reduce to the corresponding ones given earlier when T is the class of all Σ-models.

Note that, as defined here, T-interpretations allow us to consider the satisfiability in a Σ-theory T of formulas that contain sort, predicate or function symbols not in Σ. These symbols are traditionally called *uninterpreted*. In SMT applications, it is convenient to use formulas with uninterpreted constant symbols, which for satisfiability purposes are analogous to free variables, or with uninterpreted predicate/function symbols, which can be used as abstractions of other formulas/terms or of operators not in the theory.

Also note that the notions of theory and T-validity presented here are more general than those used traditionally in first-order theorem proving, where a theory is defined as a recursive set of sentences, the *axioms* of the theory, and T-validity as entailment by those axioms. The reason is that every set A of Σ-sentences is characterized by (i.e., has the same set of valid sentences as) a class of Σ-models, namely the Σ-models of A. In contrast, not every class of Σ-models is characterized by a recursive, or even non-recursive, set of (first-order) axioms.[7]

[6] Observe that the class of all T-interpretations includes all possible expansions of models in $\mathbf{A}$. This essentially means that variables and sort, function and predicate symbols not in Σ can be interpreted arbitrarily.

[7] A well-known example of the latter would the SMT theory consisting of a single Σ-interpretation for the integers with the usual operations.

In the next sections, we will often consider the T-satisfiability of conjunctions (or, equivalently, sets) of literals. We will refer to these conjunctions as *constraints* and talk about *constraint satisfiability* in T.

Abstractions SMT techniques often use propositional abstractions of first-order sentences. Since our logic properly embeds propositional logic, such abstractions can be defined as follows. Let us fix a signature Π consisting exclusively of an infinite set of propositional symbols not contained in any theory signature. Every quantifier-free formula (*qff*) of signature Π is in effect a propositional formula, satisfiable in our sense iff it is satisfiable in propositional logic. For every theory signature Σ, we define an injective mapping $(\_)^{\mathrm{a}}$ from the set of all atomic Σ-formulas into Π. This mapping extends homomorphically to a (injective) mapping, also denoted as $(\_)^{\mathrm{a}}$, from quantifier-free Σ-formulas to quantifier-free Π-formulas (i.e., propositional formulas) such that $(\neg\varphi)^{\mathrm{a}} = \neg(\varphi^{\mathrm{a}})$ and $(\varphi \vee \psi)^{\mathrm{a}} = \varphi^{\mathrm{a}} \vee \psi^{\mathrm{a}}$ for all qffs φ and ψ. We denote by $(\_)^{\mathrm{c}}$ the inverse homomorphism of $(\_)^{\mathrm{a}}$, which is such that $(\varphi^{\mathrm{a}})^{\mathrm{c}} = \varphi$, $(\neg\varphi)^{\mathrm{c}} = \neg(\varphi^{\mathrm{c}})$, and $(\varphi \vee \psi)^{\mathrm{c}} = \varphi^{\mathrm{c}} \vee \psi^{\mathrm{c}}$ for all qffs φ and ψ.

We call an *SMT solver* any program that tries to determine the satisfiability of some class $\mathbf{C}$ of formulas in some theory T. What distinguishes SMT as a field is the development and use of efficient reasoning techniques specific to the selected theory and class of formulas.

2 SMT in Model Checking

Model checking has leveraged SMT extensively in the last decade thanks to the impressive growth in the performance and scope of SMT solvers. The use of SMT solving in support of software model checking and, more generally, model checking for infinite-state transition systems is now widespread, as can be seen in the rest of this book. In this section, we give a general—and necessarily incomplete—sampling of that by focusing on a few major model checking methods.

Roughly speaking, we could say that all of these methods rely on some encoding of a software or hardware system under analysis as a transition system S whose state space is represented by the Cartesian product $D_1 \times \cdots \times D_n$ of finite or infinite domains (Booleans, fixed-size bit vectors, integers, and so on) modeled by some Σ-theory T. The system itself is (implicitly or explicitly) described by a pair $(I[\mathbf{x}], Tr[\mathbf{x}, \mathbf{x}'])$ of typically quantifier-free Σ-formulas[8] where

- $\mathbf{x}$ and $\mathbf{x}'$ are n-tuples of variables semantically ranging over $D_1 \times \cdots \times D_n$;
- $I[\mathbf{x}]$ is satisfied exactly by the initial states of S;
- $Tr[\mathbf{x}, \mathbf{x}']$ is satisfied by all pairs $\mathbf{s}, \mathbf{s}'$ of reachable states of S where $\mathbf{s}'$ is a successor of $\mathbf{s}$ in S.

[8] This is an oversimplification because, for instance, several software model checking methods also rely for efficiency on a separate representation of a program's control structure as a control flow graph. See Chap. 16 for more details.

This representation is analogous to that used in SAT-based model checking (see Chap. 10) except that the system is formulated in a more powerful logic than propositional logic, though still endowed with efficient satisfiability checkers: SMT solvers. The SMT setting has a number of advantages. To start, the first-order language allows natural and more or less direct formulations of the system under analysis—regardless of whether the system has finitely or infinitely many states. In the finite-state case, these formulations can also be exponentially more compact than propositional ones because they do not need to encode non-Boolean data types and their operations at the propositional level, which allows better scalability. Moreover, several SAT-based model checking techniques lift naturally, although not necessarily immediately, to the SMT case.

BMC and k-induction-based methods The most obvious example of such lifting is bounded model checking (BMC) [55]. As in the original propositional setting, one tries to disprove that a given state property $P[\mathbf{x}]$ is *invariant* for the system, i.e., true in all reachable states, by looking for a value $i \geq 0$ such that the formula

$$I[\mathbf{x}_0] \wedge Tr[\mathbf{x}_0, \mathbf{x}_1] \wedge \cdots \wedge Tr[\mathbf{x}_{i-1}, \mathbf{x}_i] \wedge \neg P[\mathbf{x}_i] \tag{1}$$

is satisfiable [66, 4, 97]. Another example is k-induction [145], where one tries to prove that a given state property $P[\mathbf{x}]$ is invariant by looking for a $k \geq 0$ such that (1) is unsatisfiable for all $i = 0, \ldots, k$ and the formula

$$Tr[\mathbf{x}_0, \mathbf{x}_1] \wedge \cdots \wedge Tr[\mathbf{x}_k, \mathbf{x}_{k+1}] \wedge P[\mathbf{x}_0] \wedge \cdots \wedge P[\mathbf{x}_k] \wedge \neg P[\mathbf{x}_{k+1}] \tag{2}$$

is also unsatisfiable. In both examples, the only differences with the original formulations are that the formulas (1) and (2) are first-order qffs; propositional satisfiability is replaced by T-satisfiability; and an SMT solver is used to perform the satisfiability check. Again, as in the propositional case, any variable assignment that satisfies (1) can be used to construct a counter-example trace for P. Several enhancements to BMC and k-induction (such as lemma learning, abstraction and refinement, path compression, termination checks, ...) lift to the SMT case as well [66, 97].

Interpolation-based methods Interpolation-based model checking proves a property $P[\mathbf{x}]$ invariant by constructing a formula $R[\mathbf{x}]$ that holds in all reachable states and entails $P[\mathbf{x}]$. This is done incrementally, for $i = 0, 1, \ldots$, by checking the satisfiability of formulas of the form

$$R_i[\mathbf{x}_0] \wedge Tr[\mathbf{x}_0, \mathbf{x}_1] \wedge \cdots \wedge Tr[\mathbf{x}_{k-1}, \mathbf{x}_k] \wedge (\neg P[\mathbf{x}_0] \vee \cdots \vee \neg P[\mathbf{x}_k]) \tag{3}$$

for some $k > 0$, where R_i is a formula satisfied by all states reachable in up to i-steps, starting with $R_0 = I$. When (3) is unsatisfiable, $R_i[\mathbf{x}]$ is generalized to $R_{i+1} := R_i[\mathbf{x}] \vee Int[\mathbf{x}]$ where $Int[\mathbf{x}_1]$ is a formula entailed by $R_i[\mathbf{x}_0] \wedge Tr[\mathbf{x}_0, \mathbf{x}_1]$ and jointly unsatisfiable with $Tr[\mathbf{x}_1, \mathbf{x}_2] \wedge \cdots \wedge Tr[\mathbf{x}_{k-1}, \mathbf{x}_k] \wedge (\neg P[\mathbf{x}_0] \vee \cdots \vee \neg P[\mathbf{x}_k])$, an *interpolant* of those two formulas. The property is proved if at some point R_{i+1} is equivalent to R_i, something that can be checked by verifying the unsatisfiability of $R_{i+1}[\mathbf{x}] \wedge \neg R_i[\mathbf{x}]$. This method was developed originally in the SAT setting [118].

However, it immediately lifts to the SMT setting with theories T and language fragments for which T-entailment is decidable and interpolants exist and are computable (e.g., [119]). Note that in this case a plain SMT solver is not enough, since procedures for computing theory interpolants are also needed. These procedures, however, can often be built within existing SMT solvers (see Section 8). See also Chap. 14A for a comprehensive treatment of interpolation-based methods.

Predicate abstraction-based methods Perhaps the most successful approach to software model checking so far, described in more detail in Chap. 16, is predicate abstraction. In a predicate abstraction method popularized by the SLAM model checker and further improved in other tools [8, 99], a program written in a high-level programming language (such as C or Java) and a safety property P to be checked are modeled as a system $(I[\mathbf{x}], Tr[\mathbf{x}, \mathbf{x}'])$ with a distinguished error state directly reachable from any state that violates the property.

The system (I, Tr) is abstracted to a finite-state system $\overline{S} = (\overline{I}, \overline{Tr})$, obtained, roughly speaking, by replacing *predicates* (i.e, atoms or other sub-formulas) of I and Tr by propositional variables. Then, using traditional symbolic model checking techniques (see Chap. 9), an exhaustive analysis of all the paths of $\overline{S}$ is performed to determine if the abstract error state is reachable. If a trace t to that state is found, it is converted to a formula φ that is T-satisfiable exactly when t corresponds to an execution of the original program that leads to the concrete error state. If φ is not T-satisfiable, the abstraction $\overline{S}$ is refined using techniques like those described in Chap. 16 to remove that spurious error trace t. As in the other methods above, all T-satisfiability checks involved in this process are performed by an SMT solver.

3 The Lazy Approach to SMT

The majority of the work in SMT has focused on the T-satisfiability of quantifier-free formulas and on theories T for which this problem is decidable. We discuss that major case here. Let us start by observing that to decide the quantifier-free T-satisfiability problem it is enough in principle to have a procedure for deciding the T-satisfiability of constraints (conjunctions of literals): one can first convert any quantifier-free formula to Disjunctive Normal Form, and then check each disjunct individually. This solution, however, is impractical because of the frequent exponential blow-up in the size of the resulting DNF formula. Except for degenerate (and uninteresting) examples of theories, this blow-up cannot be eliminated in general because the T-satisfiability of qffs is NP-hard, even if the constraint T-satisfiability problem is polynomial, as one can easily show by simple reductions from SAT.

To avoid the inefficiencies inherent in DNF conversions, most current SMT solvers follow a general approach that essentially amounts to constructing and checking a DNF for the input formula incrementally and as needed. The main characteristic of this approach, referred to as the *lazy approach* in the literature (e.g., [144]), is the combination of one or more specialized constraint satisfiability

```
Require: $\varphi$ is a qff in the signature $\Sigma$ of $T$
Ensure: output is sat if $\varphi$ is $T$-satisfiable, and unsat otherwise
  $F := \varphi^{\mathrm{a}}$
  loop
    $A := \mathsf{get\_model}(F)$
    if $A = \mathsf{none}$ then
      return unsat
    else
      $\mu := \mathsf{check\_sat}_T(A^{\mathrm{c}})$
      if $\mu = \mathsf{sat}$ then
        return sat
      else
        $F := F \wedge \neg\mu^{\mathrm{a}}$
```

Fig. 1 A basic SMT solver based on the lazy approach. The function get_model implements the SAT engine. It takes a propositional formula F and returns either none, if F is unsatisfiable, or a satisfiable conjunction A of propositional literals such that $A \models F$. The function $\mathsf{check\_sat}_T$ implements the theory solver. It takes a conjunction ψ of Σ-literals and returns either sat or a T-unsatisfiable conjunction μ of literals from ψ.

procedures, or *theory solvers*, with a conflict-driven clause learning (CDCL) SAT solver, the *SAT engine*, used to reason efficiently about the propositional connectives. The approach has several variants, differing in the sophistication of the interaction between the SAT engine and the theory solvers. We discuss some of them in the following.

For the rest of the section, we fix a generic Σ-theory T and assume the existence of a theory solver, or T-*solver* for short, that can decide the T-satisfiability of conjunctions of Σ-literals. We will discuss only a few general desirable features of T-solvers here. Details on algorithms and techniques for implementing theory solvers for specific theories of interest in model checking are provided in Section 4. We will assume that the reader has some familiarity with the inner workings of modern SAT solvers (see Chapter 5 for a general overview).

3.1 A Basic Lazy SMT Solver

In the most basic version of the lazy approach, with a single Σ-theory T, one abstracts each atom in the input formula by a new propositional variable (as detailed at the end of Section 1.1), uses the SAT engine to find a *model* of the formula, a satisfying assignment given as a set A of literals, and then asks the T-solver to verify that the Σ-literals abstracted by this model are jointly T-satisfiable [20, 65]. If the latter check succeeds, one can conclude that the input formula is T-satisfiable. Otherwise, one asks the SAT engine for another model—something achievable in the simplest way by adding a proper *blocking lemma*, the negation of a subset of the assignment A, to the original formula and restarting the engine. This process is repeated until a model consistent with the theory is found, or all possible propositional models have been explored with no success—in which case one can conclude that the input formula is T-unsatisfiable.

A pseudo-code description of this algorithm is provided in Figure 1, with the concretization and abstraction functions $(\_)^{\mathrm{c}}$ and $(\_)^{\mathrm{a}}$ defined as in Section 1.1. Current implementations are based on more sophisticated variations on this basic approach that exploit advanced features of modern SAT engines and theory solvers to achieve a tighter integration between them [3, 5, 75, 81, 30]. The most important ones are described next.

3.2 SAT Engine and Theory Solver Features

For efficient integration, in addition to having all the features usually found in modern SAT solvers, it is important for the SAT engine to be *on-line*, i.e., able to take and process its input progressively, maintaining at all times a set Γ of input formulas and a satisfying assignment for it. Initially, Γ is empty (and so satisfied by the empty assignment). When a new formula is fed as input, the engine attempts to modify the current assignment to satisfy the new formula as well, terminating if that is not possible or waiting for more input formulas otherwise.

T-solvers usually maintain internally at all times a set Λ of literals to be checked for T-satisfiability. The salient advanced features for these solvers are listed below.

Incrementality Intuitively, a T-solver is *incremental* if it can be given a set of literals one at a time and determine each time the T-satisfiability of the newly expanded internal set Λ with a cost proportional to the size of the addition—as opposed to the size of Λ. With an incremental T-solver, the model produced by the SAT engine can be checked for T-satisfiability as it is being constructed, and so discarded as soon as it becomes T-unsatisfiable. Decision procedures used for most theories were either already incremental in their original formulation or have been adapted to be so by SMT researchers.

Backtrackability Incremental solvers are naturally state-based. A state-based T-solver is *backtrackable* if, for any of the literals in its current input set L, it is able to restore inexpensively the state it had right before it was fed that literal. This feature is crucial to keep an incremental T-solver in sync with the SAT engine, which itself relies on backtracking to recover from propositional conflicts generated while attempting to construct a model for its input formula.

Conflict set generation A *conflict set* for a T-unsatisfiable input set Λ to a T-solver is a (ideally minimal) subset $\{l_1, \ldots, l_n\}$ of Λ that is already T-unsatisfiable. The T-valid formula $\neg l_1 \vee \cdots \vee \neg l_n$ constructed from this set is called a *justification* or *explanation* (of Λ's unsatisfiability). An explanation can be abstracted and passed to the SAT engine, to be treated as a learned clause. Its immediate effect is to create a conflict in the engine and force a backtrack. If it is kept afterwards, its later effect will be the same as that of learned lemmas in CDCL SAT solvers: to drive the search away from other parts of the search space that would generate the same conflict.

Literal deduction A T-solver with this feature is able to identify consequences of its current set Λ among a predetermined set L of literals—i.e., identify literals $l \in L$ such that $\Lambda \models_T l$. This information is useful to the SAT engine which then does not have to guess the value of these literals. Typically, but not exclusively, L consists of all atoms occurring in the original input formula (the formula φ in Figure 1), as well as their negation. *Theory propagation*, the process of communicating entailed literals to the SAT engine, can be partial or exhaustive, depending on the cost of determining all entailed literals of L. For some theories, such as for instance difference logic (cf. Section 4.5), exhaustive theory propagation is extremely cheap and proves highly effective. For others, it pays off performance-wise to propagate only literals of L that happened to be deduced in the process of checking the satisfiability of the input set Λ. This is for instance the case for the (positive) equalities computed by congruence closure in solvers for the theory of equality (cf. Section 4.1).

Explanation generation With theory propagation, the SAT engine may generate a conflict involving a theory propagated literal l. For the engine to perform its conflict analysis and determine how far to backtrack, it is necessary to have an *explanation* for l, a formula of the form $l_1 \wedge \cdots \wedge l_n \rightarrow l$ where $\{l_1, \ldots, l_n\}$ is a subset of the literals Λ in the T-solver such that $l_1, \ldots, l_n \models_T l$. Typically, the same mechanisms and infrastructure used to compute conflict sets can be used to compute these explanations too.[9] Explanations need not be minimal, as computing those can be unacceptably expensive, but should be relatively small since shorter explanations usually lead to better conflict analysis than longer ones. A complication and main difference with conflict sets is that literal explanations are (best) computed *a posteriori* and as needed, whereas conflict sets are usually computed as soon as the input set becomes unsatisfiable (see [129] for a discussion).

3.3 A General Framework and Architecture

SMT solvers implementing the many variants of the lazy approach can be described abstractly and declaratively in terms of a transition system between states of the form $M \parallel F$, where M is a sequence of Σ-literals and *decision points*, and F is a quantifier-free Σ-formula in conjunctive normal form, or, equivalently, a set of clauses; an additional distinguished state *Fail* is used to model the discovery by the SMT solver that its input formula is T-unsatisfiable [128, 129]. Identifying for simplicity every Σ-literal l with its propositional abstraction l^{a}, the sequence M represents the propositional assignment being built by the SAT engine, together with the engine's non-deterministic guesses; the formula F models the current set of clauses being processed by the SMT solver. Slightly more concrete versions of this

[9] In fact, one can look at a conflict set as an explanation for the literal $\perp$.

framework also model conflict analysis and lemma construction by adding states of the form $M \parallel F \parallel C$ where M and F are as before and C is a *conflict clause* for M and F, a clause T-entailed by F and propositionally falsified by M [111, 139].

This declarative framework has been used to provide a clean formulation of the different lazy variants and a basis for comparison and formal analysis at an abstraction level free of inessential implementation and control details. Its description, however, is beyond the scope of this chapter. We refer the reader to the original work [129, 111] or previous survey work [12] for more information and formal correctness results. Here, we informally describe instead a general modular architecture for SMT solvers based on the lazy approach, known in the literature as DPLL(T).

The DPLL(T) Architecture The architecture relies on a generic CDCL-style SAT engine, called DPLL(X), which is parametric in the theory and theory solver used. Instantiating the parameter X with a theory solver for some theory T produces a *DPLL(T) system* that can be seen as a concrete implementation of the abstract framework mentioned above.[10] In particular, the engine maintains the partial assignment M and the current formula F. The T-solver maintains a set Λ of literals—which at any time is a subset of those in M. The T-solver can be arbitrary as long as it conforms to a specific, simple interface. The precise details of the interface are not needed here (the interested reader is referred to [81, 129]). It suffices to know that the T-solver provides operations that the DPLL(X) engine can invoke to do the following.

- Assert a literal l, that is, ask for l to be added to Λ. This operation is to be invoked when the DPLL(X) engine adds l to its partial assignment M.
- Ask whether the current set Λ of asserted literals is T-unsatisfiable. This request can be made by the DPLL(X) engine with different degrees of *strength*: for theories where deciding unsatisfiability is expensive, it can be more effective for the engine to rely on a cheap, if incomplete, T-unsatisfiability check while it is building the partial assignment M, and request a complete one only when M propositionally satisfies F (and Λ contains all the literals in M).
  In response, when it determines the T-unsatisfiability of Λ, the T-solver returns an explanation of that.
- Request a set of input literals not in Λ that are T-entailed by Λ. The returned set, which is used for theory propagation, need not include all T-entailed literals. Note that for this operation the T-solver must know the set of all input literals.
- Request an explanation for a previously theory propagated literal l. Explanations are used by the DPLL(X) engine during the analysis of conflicts that involve theory propagated literals.
- Request the T-solver to *undo* the n most recent assertions, that is, to remove from Λ its n most recent literals, for some $n > 0$. This operation is to be invoked

[10] The motivation for the abbreviation DPLL in DPLL(T) is historical. At the time the architecture was proposed [81], CDCL solvers were still commonly referred to as DPLL solvers—in reference to the work of Davis, Putnam, Logemann and Loveland [60, 61].

after the engine backtracks to some previous decision level and shrinks its partial assignment M correspondingly.

DPLL(T) is currently the most popular general architecture for SMT solvers based on the lazy approach. However, its black-box treatment of the SAT engine and the theory solvers (originally an asset because it allowed the use of minimally modified off-the-shelf SAT solvers) is becoming a limitation as research in SMT advances. A number of alternative architectures have been proposed quite recently that aim at overcoming these limitations by integrating propositional-level and theory-level reasoning more tightly [33, 101, 121].

4 Theory Solvers for Specific Theories

In this section, we consider solvers for constraint satisfiability problems in several specific theories. For each theory, we first describe the signature and semantics of the theory and then discuss how to solve its constraint satisfiability problem.

4.1 Uninterpreted Function Symbols

We start with the simplest possible theory consisting of a given signature Σ and the class of all Σ-models. This theory, or rather family of theories parametrized by the signature, is known as the theory of *Equality with Uninterpreted Functions* (*EUF*) or the *empty theory*—since it imposes no restrictions on its models.

Conjunctions of literals in this theory can be decided in polynomial time by congruence closure algorithms. For simplicity, we describe a version of the algorithm assuming no predicate symbols. This assumption loses no generality, because predicate symbols can be handled using a simple encoding: introduce a new sort symbol B and a new function symbol f_p of rank $\sigma_1 \cdots \sigma_n \mathsf{B}$ for each predicate symbol p of rank $\sigma_1 \cdots \sigma_n$, plus a new constant symbol $\mathtt{tt}$ of sort B; then, replace each literal $p(t_1, \ldots, t_n)$ with $f_p(t_1, \ldots, t_n) = \mathtt{tt}$ and each literal $\neg p(t_1, \ldots, t_n)$ with $f_p(t_1, \ldots, t_n) \neq \mathtt{tt}$.

Let Φ be a set of literals to be checked for satisfiability. Since there are no predicate symbols, Φ can be partitioned into a set E of equalities and a set D of disequalities. Let E^* be the *congruence closure* of E, defined as the smallest equivalence relation (over the terms in Φ) that includes E and also satisfies the congruence property: for every pair of terms $f(s_1, \ldots, s_n)$ and $f(t_1, \ldots, t_n)$, $(f(s_1, \ldots, s_n), f(t_1, \ldots, t_n)) \in E^*$ whenever $(s_i, t_i) \in E^*$ for $i = 1, \ldots n$.[11] Then, Φ is satisfiable iff for each $t_1 \neq t_2 \in D$, $(t_1, t_2) \notin E^*$.

Example 1. Let $\Phi = \{f(f(a)) = a, f(f(f(a))) = a, g(a) \neq g(f(a))\}$. The equivalence classes induced by E are $\{a, f(f(a)), f(f(f(a)))\}$, $\{f(a)\}$, $\{g(a)\}$, $\{g(f(a))\}$.

[11] Observe that two terms may be related by E^* only if they have the same sort.

Congruence closure requires merging the first two classes and the last two. As a result, $(g(a), g(f(a))) \in E^*$ and Φ is not satisfiable.

Standard algorithms use directed acyclic graphs (DAGs) to represent terms, and a *union-find* data structure [152] to represent equivalence classes of terms. The main work is in the congruence closure step. A simple $O(n^2)$ algorithm for congruence closure is as follows [124]: seed a work-list with the equalities in E; then, while the work-list is non-empty, remove an equality, perform a *union* operation on the two equivalence classes containing the terms on either side of the equality, and then examine all pairs of parents (in the DAG) of these terms to see if any of them newly satisfy the congruence property; if they do, add this new pair to the work-list. Once the work-list is empty, Φ is satisfiable iff for each disequality $t_1 \neq t_2 \in D$, the *find* of t_1 is different from the *find* of t_2. More efficient algorithms ($O(n \log n)$) only require traversing one set of parents after each *union* operation and include efficient mechanisms for computing, in the case when Φ is unsatisfiable, a small unsatisfiable subset of Φ [69, 127].

4.2 Real arithmetic

Next, consider the signature Σ containing a single sort, R, all rational number constants, function symbols $\{+, -, *\}$ and the predicate symbol $\leq$, all with the expected rank. The theory of real arithmetic consists of this signature paired with the standard model of the real numbers, that is the Σ-model that interprets R as the set $\mathbb{R}$ of the real numbers and the constants and operators in the usual way. Satisfiability of Σ-formulas in this theory, even with quantifiers, is decidable [115]. Traditionally, decision procedures for the full theory have not been efficient enough to be practical. It is worth noting, however, that this is an area of active research and several promising new approaches are being investigated [84, 102]. Efficient decision procedures do exist for the satisfiability of appropriately restricted classes of quantifier-free Σ-formulas in this theory.

Consider, for instance, *linear real arithmetic* (LRA). Here, formulas are restricted in that the symbol $*$ can only appear if at least one of its two operands is a rational constant. For illustration purposes, we describe here a simple algorithm based on Fourier–Motzkin elimination [143]. For convenience, let $t_1 < t_2$ abbreviate $\neg(t_2 \leq t_1)$ and assume that in all constraints, like terms are combined.

Now, suppose we are given a set Φ of LRA literals. We first eliminate disequalities by replacing $t_1 \neq t_2$ by $t_1 < t_2 \vee t_2 < t_1$. We also eliminate weak inequalities by replacing $t_1 \leq t_2$ with $t_1 < t_2 \vee t_1 = t_2$. These steps introduce disjunctions, but case-splitting or conversion to DNF can be used to reduce the new problem to several instances of simple conjunctions of strict inequalities. Next, we eliminate equalities. If $t_1 = t_2$ cannot be solved for some variable x, it must either be trivially true or trivially false. If the former, we remove it; if the latter, Φ is unsatisfiable and we are done. Otherwise, the equality is equivalent to $x = t_3$ for some term t_3. In this case, we replace x everywhere by t_3 and then remove the equality.

We are left with only conjunctions of strict inequalities, to which we apply Fourier–Motzkin elimination. We pick a variable x occurring in Φ to eliminate, and rewrite all constraints containing x as either (i) $t_1 < x$ or (ii) $x < t_2$. For every possible pair of constraints in Φ consisting of a constraint of the form (i) and one of the form (ii), we introduce the new constraint $t_1 < t_2$. We then remove all constraints containing x, eliminating x from Φ. We repeat with another variable until no more variables appear in Φ. The result is a set of inequalities over rational constants that can easily be simplified to $\perp$, indicating that Φ is unsatisfiable, or $\neg\perp$, indicating that Φ is satisfiable.

Example 2. Let $\Phi = \Phi_1 \cup \{w \leq y\}$ with $\Phi_1 = \{x < y+z, x-y = z-w, y < 0\}$. Eliminating $\leq$ yields two sets of constraints: $\Phi_1 \cup \{w < y\}$ and $\Phi_1 \cup \{w = y\}$. In the first set, solve the equality for x to get $x = y+z-w$. After substituting and combining like terms, we have $\{0 < w, y < 0, w < y\}$. Applying Fourier–Motzkin elimination to y results in $\{0 < 1, w < 0\}$. Then eliminating w yields $0 < 0$, which simplifies to $\perp$. For the second set of constraints, first eliminate $w = y$ by substituting y for w everywhere to get $\{x < y+z, x = z, y < 0\}$. Next, eliminate x which gives $\{0 < y, y < 0\}$. Fourier–Motzkin elimination on y then again yields $0 < 0$. Thus Φ is unsatisfiable.

Each elimination step in the procedure above introduces in the worst case a quadratic number of new constraints, making the procedure doubly exponential. For this reason, Fourier–Motzkin elimination is usually not practical for large sets of constraints, Though more efficient procedures based on Fourier–Motzkin have been developed [101, 109], procedures based on the Simplex method are currently preferred because of their superior overall performance. A Simplex-based algorithm specialized for use in SMT solvers is given in [70], and further improvements on it are described in [71, 95, 107].

4.3 Integer arithmetic

Consider now a signature Σ containing a single sort Z, for the integers, all integer number constants, function symbols $\{+,-,*\}$ and the predicate symbol $\leq$, all with the expected rank. The theory of integer arithmetic consists of this signature paired with the standard model of the integers, the Σ-model that interprets Z as the set $\mathbb{Z}$ of the integers, and the constants and operators in the usual way. The satisfiability of Σ-formulas in this theory, even without quantifiers, is undecidable [115].

The *linear integer arithmetic* (LIA) fragment is the analog of the LRA fragment described above: the symbol $*$ can only appear if at least one of its operands is an integer constant. The fully quantified LIA fragment is also known as *Presburger arithmetic* and is decidable using Presburger's algorithm [135]. More efficient methods exist for the quantifier-free fragment. Again, for illustration purposes, we describe here a relatively simple procedure for quantifier-free LIA based on the Omega test [21, 110, 137]. This is essentially an integer adaptation of the Fourier–Motzkin elimination procedure described for the reals above.

Let Φ be a set of LIA literals. As before, we assume that all constraints are normalized by combining like terms. We divide coefficients in each constraint by any common factors and check for any contradictions in constraints involving only constants. Then, we eliminate disequalities by replacing $t_1 \neq t_2$ with $t_1 < t_2 \vee t_2 < t_1$, where again $s < t$ abbreviates $\neg(s \leq t)$. We similarly eliminate weak inequalities by replacing $(t_1 \leq t_2)$ with $t_1 < t_2 + 1$.

The next step is the elimination of equalities. If it is possible to solve some equation for some variable x, we do this and either: (i) halt the procedure and report Φ is unsatisfiable if the right-hand side of the solved equation reduces to a non-integer constant; or else (ii) substitute the right-hand side for x in Φ as before. If it is not possible to solve any equation for a variable while maintaining integer coefficients, we proceed as follows: let a be the minimum coefficient of any variable and write the equation it appears in as $ax + \sum_i a_i x_i + c = 0$. Let $m = |a| + 1$ and define $k \mathbin{\widehat{mod}} m = k - m \left\lfloor \frac{k}{m} + \frac{1}{2} \right\rfloor$. Note that $\widehat{mod}$ distributes (modulo m) over both multiplication and addition. Next, apply this operator to both sides of the original equation to get: $\pm(x \mathbin{\widehat{mod}} m) + \sum_i (a_i \mathbin{\widehat{mod}} m)(x_i \mathbin{\widehat{mod}} m) + (c \mathbin{\widehat{mod}} m) = 0$ (modulo m). Expanding the definition of $\widehat{mod}$, this can be rewritten as: $\pm x + \sum_i (a_i \mathbin{\widehat{mod}} m) x_i + (c \mathbin{\widehat{mod}} m) = m \cdot y$ where y is a fresh variable. This equation can now be used to eliminate x from the original equation (and indeed from Φ). The new equation still has the same number of variables, since y was introduced, but in the new equation, the absolute values of the coefficients of all variables other than y are reduced by a factor of at least $2/3$, while the absolute value of the coefficient of y is in fact $|a|$. By repeating this process a logarithmic number of times, we eventually obtain an equation in which some variable has coefficient ± 1 and can thus be eliminated without introducing any new variables. This process can be used to eliminate all equality constraints from Φ.

The final step again involves only conjunctions of strict inequalities and is similar to Fourier–Motzkin elimination. We pick a variable x occurring in Φ to eliminate, and write all constraints containing x as either (i) $ax < t_1$ or (ii) $t_2 < bx$ where a and b are positive integers. We remove these constraints from Φ and then for every possible pair consisting of a constraint of the form (i) and a constraint of the form (ii), we add a new constraint, choosing from the following three alternatives: the *real shadow* $at_2 < bt_1$; the *dark shadow* $bt_1 - at_2 > ab$; and the *gray shadow* $\bigvee_{i=1}^{i=b-1} bx = t_2 + i$. The first two are approximations, with the first preserving the soundness and the second the completeness of the procedure. The gray shadow is exact and can be used to eliminate x via additional case splitting and equation solving. However, this can be prohibitively expensive, so one possible strategy is: check whether the real shadow constraints are sufficient for unsatisfiability; failing that, check whether the dark shadow constraints are satisfiable; and finally, failing that, check the gray shadow constraints.

Example 3. Let $\Phi = \{2x = 3w + 4z,\ w < x,\ 2x + 4z < w\}$. When solving for x, the minimal coefficient is 2, so $m = 3$, and we can derive the new equation $-x - z = 3y$, or $x = -3y - z$. Substituting into the first equation, we get $-6y - 2z = 3w + 4z$, or $w = -2y - 2z$. Substituting for x and w in the second constraint, we get $-2y - 2z < -3y - z$ or $y < z$. Substituting for x and w in the third constraint, we get $2(-3y - z) + 4z < -2y - 2z$ or $z < y$. The real shadow is now unsatisfiable.

As with real arithmetic, better performance is possible by using Simplex-based algorithms, in this case expanded with additional techniques for obtaining integer solutions [68, 70, 71, 95, 96].

4.4 Mixed Integer and Real Arithmetic

Sometimes it is desirable to mix integer and real reasoning. A simple solution is to use two different sorts, Z and R and then create two copies of the arithmetic symbols, one set for integers and one set for reals. Mapping operators, such as toInt of rank RZ (returning the integer part of a real) and toReal of rank ZR (returning the corresponding real) can be used to mix real and integer terms.

Alternatively, mixing can be done by reasoning within the theory of reals with the addition of a unary predicate symbol Int whose interpretation is exactly the set of all whole (real) numbers. Often, constraints of interest limit the use of the Int predicate to variables (as opposed to more complicated terms). In such cases, an algorithm can be obtained by mixing approaches for LRA and LIA [21, 71].

For example, suppose Φ is a set of literals. Let V_{Z} be the set of variables in Φ that are constrained by Int, and let V_{R} be the set of remaining variables of Φ. We can eliminate disequalities and weak inequalities as before. Then, all equations that contain at least one variable in V_{R} can be eliminated by solving for the variable and then substituting for that variable in Φ. Next, the remaining variables in V_{R} can be eliminated by performing Fourier–Motzkin elimination. The result of this step is a system of equalities and inequalities over only the variables in V_{Z}. Furthermore, each constraint can be made to have integer coefficients by multiplying through by the least common multiple of the denominators appearing in its rational coefficients. The resulting set of constraints can be solved using any algorithm for LIA, such as the one described above.

4.5 Difference Logic

Difference logic refers to a quantifier-free arithmetic fragment in which all atoms are of the form $x - y \bowtie c$, where $\bowtie \in \{=, \leq, \geq\}$, c is a constant, and x and y are variables. The background theory may be the theory of real arithmetic, in which case c can be any rational constant and the fragment is called *real difference logic* (RDL). Alternatively, it may be the theory of integer arithmetic, in which case c is required to be an integer constant and the fragment is called *integer difference logic*.

Conjunctions of literals in either IDL or RDL can be solved in polynomial time. A simple algorithm is as follows [125].

Let $t_1 < t_2$ abbreviate $\neg(t_2 \leq t_1)$, and eliminate disequalities by replacing $\neg(x - y = c)$ with $x - y < c \vee x - y > c$. We similarly eliminate equalities by replacing $(x - y = c)$ with $x - y \leq c \wedge x - y \geq c$. Finally, we write all constraints in terms of $\leq$ by applying the following rewriting steps: (i) $x - y \geq c \longrightarrow y - x \leq -c$; (ii) $x - y > c \longrightarrow y - x < -c$; and (iii) $x - y < c \longrightarrow x - y \leq c - 1$. Step (iii) is only valid in IDL. For RDL, a slight variation is possible: $x - y < c \longrightarrow x - y \leq c - \delta$ where δ is a rational positive constant chosen to be sufficiently small [143].

Now, we form a weighted directed graph with a vertex for each variable and an edge from x to y with weight c for each constraint $x - y \leq c$. The set of constraints is satisfiable iff there is no cycle for which the sum of the weights on the edges is negative, which can be determined using standard graph algorithms [48].

Example 4. Let $\Phi = \{x - y = 5, z - y \geq 2, z - x > 2, w - x = 2, z - w < 0\}$. After eliminating equality and rewriting, we have $\{x - y \leq 5, y - x \leq -5, y - z \leq -2, x - z \leq -3, w - x \leq 2, x - w \leq -2, z - w \leq -1\}$. In the associated graph, the cycle from x to z to w to x has total weight -2. Therefore, Φ is unsatisfiable.

The algorithm described here is elaborated and extended in [56, 158]. An efficient alternative algorithm based on a reduction to propositional logic is described in [106].

4.6 Bit Vectors

The theory of *fixed-size bit vectors* is useful for modeling hardware or low-level software. The theory signature consists of one sort BV_n for each bit width $n \geq 1$; 2^n binary constants for each such sort, each representing a constant bit vector of width n; and a large set of operators corresponding to standard hardware and software operations on bit vectors. For example, $t_1 \circ t_2$ represents the *concatenation* of bit vectors t_1 and t_2 and $t[i : j]$ represents the *extraction* of bits i through j of t, where $n > i \geq j \geq 0$ and n is the bit width of t.

A conjunction of equations containing only concatenation and extraction operators can be checked for satisfiability in polynomial time as follows [40, 59]. In step (i), simple rewrites are used to distribute extraction over concatenation, other extractions, or constants, until the only arguments of extractions are variables. In step (ii), whenever an equation contains a concatenation on one side, $r \circ s = t$, it is replaced by two equations: $r = t[n-1 : m]$ and $s = t[m-1 : 0]$, where n is the bit width of t and m is the bit width of s. In step (iii), if a variable x appears as an argument to two different extractions, $x[i : j]$ and $x[k : l]$, with $i \geq k \geq j$, then $x[i : j]$ is replaced by $x[i : k+1] \circ x[k : j]$. Similarly, if $i \geq l \geq j$, then $x[i : j]$ is replaced by $x[i : l+1] \circ x[l : j]$. These three steps are repeated until they can no longer be applied. Let $\sim$ be the equivalence relation over terms induced by the resulting set of equations. The original equations are unsatisfiable iff there exist two distinct binary constants, c_1 and c_2, such that $c_1 \sim c_2$.

Example 5. Let x be of width 4 and consider the equation $1 \circ x = x \circ 0$. Step (ii) produces three new equations: $1 = x[3:3]$, $x[0:0] = 0$, and $x[3:1] = x[2:0]$. Then, step (iii) requires that the last equation be replaced with $x[3:3] \circ x[2:1] = x[2:1] \circ x[0:0]$. Repeating step (ii) on this equation gives $x[3:3] = x[2:2]$, $x[2:2] = x[1:1]$, and $x[1:1] = x[0:0]$. The equivalence relation induced by all of these equations equates 0 and 1, so the original equation is unsatisfiable.

Almost any extensions beyond this core fragment of the theory, including just allowing disequalities, make the constraint satisfiability problem NP-hard. Recent results show that, depending on the extension, the problem can be NP-complete, PSPACE-complete, or up to NEXPTIME-complete for the full fragment [77]. Solvers typically handle the general case by first employing a set of rewrite rules to simplify and normalize parts of the input and then encoding the result as a propositional satisfiability problem. This can be done by assigning a propositional variable to each bit in each bit vector variable and then using propositional logic formulas to encode each equation in terms of these variables—a process known as *bit blasting*. In reality, the situation is more nuanced as several bit blasting SMT solvers, including non-DPLL(T) solvers such as Boolector and STP, bit blast some of their internal formulas only as needed, and so combine aspects of both the lazy and the eager approach.

Both the rewrite rules and the method of encoding can dramatically affect performance, as detailed in an extensive set of publications on the subject [7, 19, 24, 35, 36, 42, 80, 98, 116].

4.7 Arrays

Consider a signature Σ with sorts $\mathsf{A}, \mathsf{I}, \mathsf{E}$ (for *arrays*, *indices* and array *elements*) and function symbols: read, of rank $\mathsf{A\,I\,E}$ and write of rank $\mathsf{A\,I\,E\,A}$. Then, consider the theory consisting of all Σ-structures satisfying the axioms:

1. $\forall a{:}\mathsf{A}\,\forall i{:}\mathsf{I}\,\forall v{:}\mathsf{E}\ \mathsf{read}(\mathsf{write}(a,i,v),i) = v$,
2. $\forall a{:}\mathsf{A}\,\forall i,j{:}\mathsf{I}\,\forall v{:}\mathsf{E}\ i \neq j \Rightarrow \mathsf{read}(\mathsf{write}(a,i,v),j) = \mathsf{read}(a,j)$,
3. $\forall a\,\forall b{:}\mathsf{A}\ (\forall i{:}\mathsf{I}\,\mathsf{read}(a,i) = \mathsf{read}(b,i)) \Rightarrow a = b$.

This is the theory of functional arrays with extensionality. (Axiom (3) may be omitted to obtain a theory without extensionality.) This theory is especially useful for modeling memories or array data structures. The full theory is undecidable although it contains a number of decidable fragments [32].

A simple algorithm for constraint satisfiability can be obtained by naive instantiation of the axioms plus the use of congruence closure (e.g., [104]). Let Φ be a set of Σ-literals. With no loss of generality, assume that each element of Φ is a *flat literal*, that is, of the form $a = b$, $a \neq b$, $v = \mathsf{read}(a,i)$, and $b = \mathsf{write}(a,i,v)$, where

a,b,i,v are variables.[12] First, replace any disequality $a \neq b$ between array variables with $\mathsf{read}(a,k) \neq \mathsf{read}(b,k)$, where k is a fresh variable of sort I. Now, let I be the set of all variables in Φ of sort I, and replace each formula of the form $a = \mathsf{write}(b,i,v)$ with $\mathsf{read}(a,i) = v \wedge \bigwedge_{j \in I}(i = j \vee \mathsf{read}(a,j) = \mathsf{read}(b,j))$. Since this step introduces disjunctions, case-splitting or conversion to Disjunctive Normal Form (DNF) can be used to reduce the new problem to several instances of sets of literals. Each such instance can be checked for satisfiability using only congruence closure over read, since write no longer appears in Φ. The set Φ is satisfiable iff one of these instances is satisfiable.

Example 6. Let $\Phi = \{\mathsf{read}(a,i) = v, \mathsf{read}(b,i) \neq v, w \neq v, a = \mathsf{write}(b,j,w)\}$. The reduction replaces the last equation with $\mathsf{read}(a,j) = w \wedge (i = j \vee \mathsf{read}(a,i) = \mathsf{read}(b,i))$. Now, note that if $i = j$, then congruence closure generates $\mathsf{read}(a,i) = \mathsf{read}(a,j)$ and so $v = w$, contradicting $w \neq v$. On the other hand, if $\mathsf{read}(a,i) = \mathsf{read}(b,i)$, then this contradicts $\mathsf{read}(b,i) \neq v$. Thus, Φ is unsatisfiable.

In practice, various heuristics and optimizations are used to avoid many unnecessary axiom instantiations, greatly reducing the number of cases that must be considered [26, 34, 63, 80, 93, 150].

4.8 Other Theories

There are many other theories of general interest with decision procedures that have been or could be integrated into SMT solvers. These include theories of finite sets [46], finite multi-sets [133], inductive data types [13] (lists, records, and tuples can be handled as special cases), character strings [105], pointers [47, 114], and floating point numbers [141]. It is also possible to design special-purpose theories for specific application domains (see, e.g. [126]).

5 Combining Theory Solvers

All constraint satisfiability procedures described in the previous section consider theories of a single data type. In many applications of SMT, however, including model checking, one is often interested in the satisfiability of formulas over several data types (e.g., arrays with integer indices and real values, lists of integers, etc.) and consequently, over some combination of their theories. An important theoretical and practical question then is whether and how constraint satisfiability procedures for different theories can be combined modularly into a single one so as to allow the construction of theory solvers for a combination of these theories. This section gives an overview of notable combination methods and results.

[12] Any set of literals can be converted into an equisatisfiable set of flat literals by introducing new variables.

A general mechanism for combination is available when the desired combination of theories is axiomatized simply by the union of the axioms of the individual theories.[13] More formally, since theories here are defined as classes of models, a modular combination of two theories (the combination of more theories is similar) is defined as follows.

Let $T_1 = (\Sigma_1, \mathbf{A}_1)$ and $T_2 = (\Sigma_2, \mathbf{A}_2)$ be two theories such that Σ_1 and Σ_2 agree on the rank they assign to their shared function and predicate symbols.[14] The *combination* of T_1 and T_2 is the theory $T_1 \oplus T_2 = (\Sigma_1 \oplus \Sigma_2, \mathbf{A})$ where $\Sigma_1 \oplus \Sigma_2$ is the smallest supersignature of both Σ_1 and Σ_2, and $\mathbf{A} = \{\mathscr{A} \mid \mathscr{A}^{\Sigma_1,\emptyset} \in \mathbf{A_1} \text{ and } \mathscr{A}^{\Sigma_2,\emptyset} \in \mathbf{A_2}\}$. These definitions encompass the more traditional view of theories defined by a set of axioms. In particular, if $\mathbf{A}_i$ is the class of all Σ_i models that satisfy a set $\mathbf{Ax}_i$ of Σ_i-sentences for $i = 1,2$, then $\mathbf{A}$ above is the class of all $\Sigma_1 \oplus \Sigma_2$-models that satisfy the set $\mathbf{Ax}_1 \cup \mathbf{Ax}_2$ [138, 155]. Given, for $i = 1,2$, a decision procedure for the satisfiability of sets of Σ_i-literals in a Σ_i-theory T_i, we are interested in constructing a decision procedure for the satisfiability of sets of $\Sigma_1 \oplus \Sigma_2$-literals in $T_1 \oplus T_2$ using those procedures as black boxes.

5.1 A Basic Combination Method

A combination method originally due to Nelson and Oppen [123], and later adapted and extended to sorted logics by others [91, 138, 156], provides a general mechanism for combining decision procedures as above. Variants of the method are implemented in all major SMT solvers. Its essence is captured by the following nondeterministic procedure.

The Nelson–Oppen procedure Let Γ be a set of literals in the combined signature $\Sigma_1 \oplus \Sigma_2$. (i) First, *purify* Γ by constructing an equisatisfiable literal set $\Gamma_1 \cup \Gamma_2$ where each Γ_i consists of Σ_i-literals only. This can be easily done by finding a pure (i.e., Σ_i-for some i) subterm t, replacing it with a new variable v, adding the equation $v = t$ to the set, and then repeating this process until all literals are pure. (ii) Then get the component satisfiability procedures to agree on the values assigned to the *shared variables*[15] of Γ_1 and Γ_2, the variables appearing in both Γ_1 and Γ_2. This is done by guessing an *arrangement of* V, that is, a set $arr(V)$ of equations and disequations encoding an equivalence relation over V (and so expressing which pairs of variables take the same value and which do not). (iii) Finally, check each Γ_i locally for T_i-satisfiability under the chosen arrangement.

If each satisfiability procedure finds its respective input $\Gamma_i \cup arr(V)$ satisfiable, report the original set Γ to be $T_1 \oplus T_2$-satisfiable. Otherwise, repeat steps (ii) and

[13] An example of a theory which is *not* a modular combination in this sense is the theory of finite sets with cardinality. This theory includes the theory of finite sets and the theory of integers, but also additional, *mixed* axioms defining the cardinality operator.

[14] Shared symbols with (same name but) different ranks can always be renamed apart.

[15] Also called *interface variables* in recent literature—see, e.g., [37].

(iii) with another arrangement. If no suitable arrangement exists, report Γ to be $T_1 \oplus T_2$-unsatisfiable. $\square$

The non-deterministic combination procedure above yields a decision procedure for a large class of theories. Its main requirement is that T_1 and T_2 be *disjoint* in the sense of not sharing any function or predicate symbols. The procedure is terminating simply because the purification step is terminating and the number of possible arrangements is finite (although exponential in the number of shared variables). The procedure is *refutationally sound* for any two disjoint theories: every set it declares $T_1 \oplus T_2$-unsatisfiable is indeed so. Without additional restrictions the method is not *refutationally complete*, as it may fail to detect the unsatisfiability of its input for certain pairs of disjoint theories [157]. It becomes complete if both T_1 and T_2 are *stably infinite* over the sorts they share [131, 154, 156]. A Σ-theory T is stably infinite over a sort σ in Σ if every T-satisfiable quantifier-free Σ-formula is satisfiable in a T-interpretation that interprets σ as an infinite set.

Many theories of interest in SMT applications are indeed stably infinite over some or all of their sorts. Examples include the various theories of arithmetic discussed in Section 4 and the theory of arrays, which is stably infinite over its array, index and element sorts. However, some are not—most notably the theory of bit vectors described in Section 4.6.

With disjoint stably infinite theories the combination method has an exponential worst-case time complexity in general. More precisely, if the constraint satisfiability problem for T_i can be decided in time $O(f_i(n))$ for $i = 1,2$, the corresponding problem for $T_1 \oplus T_2$ can be decided in time $O(2^{n^2} \times (f_1(n) + f_2(n)))$, with the exponential factor due to the need to guess the right arrangement [131].

5.2 Combination Variants and Extensions

Actual implementations of the non-deterministic procedure sketched above try to reduce its exponential penalty by reducing the amount of guessing with arrangements. The most common approach is to check the satisfiability of Γ_1 and Γ_2 locally and then *deduce and propagate*, from one component decision procedure to the other, disjunctions of shared equalities entailed by Γ_1 or Γ_2. This is particularly effective when T_1 and T_2 are both *convex* over the sorts they share because then it is enough for completeness to consider only individual entailed equalities. A Σ-theory T is convex over a sort $\sigma \in \sigma^{\mathrm{S}}$ if for all sets Φ of Σ-literals and all sets E of equalities between variables of sort σ, $\Phi \models_T \bigvee_{e \in E} e$ iff $\Phi \models_T e$ for some $e \in E$. With convex theories, worst-case time complexity of the combination goes down to $O(n^4 \times (f_1(n) + f_2(n)))$ [131].

With non-convex theories, or convex theories for which computing entailed equalities is expensive, another approach is to check the T_i-satisfiability of Γ_i alone for some $i = 1,2$ and, once a model $\mathscr{A}_i$ is found, make the optimistic assumption that $\Gamma_j \cup arr(V)$ is T_j-satisfiable, where $j \neq i$ and $arr(V)$ is the arrangement of V induced by $\mathscr{A}_i$. If $\Gamma_j \cup arr(V)$ is unsatisfiable because of some of the literals in $arr(V)$, a new model for Γ_i with different truth values for those literals must be found. For some theory combinations, this heuristic approach is highly effective in practice [64].

The stable infiniteness requirement can be relaxed for one theory if the other satisfies stronger properties [112, 138, 157]. However, the equality sharing mechanism of the original combination procedure needs to be extended to certain cardinality constraints. The most general results so far in the context of many-sorted logic are described in [100]. A case for using a typed logic with parametric types to frame and generalize Nelson–Oppen combination is provided in [112]. A few extensions have been proposed to lift the disjointness restriction, most notably by Ghilardi and his collaborators, although their interest thus far has been mostly theoretical [88, 91, 155]. Recent work, however, uses Ghilardi's results to develop novel SMT-based LTL model checking algorithms [89, 90, 92].

6 SMT Solving Extensions and Enhancements

The scope of SMT solvers, especially those based on the lazy approach, has been extended further in a number of directions. Also, several solvers contain further enhancements aimed at improving their overall performance. We briefly describe a few significant extensions and enhancements next.

Multiple Theories When working with multiple theories $T_1, \dots, T_n$ that can be combined with the Nelson–Oppen method, one can generate a single theory solver for their combination T by combining the constraint satisfiability procedures for various theories, as described in Section 5. With solvers based on the DPLL(T) architecture a better approach is to develop a independent theory solver for each theory and extend the interface of the SAT engine so that it interacts directly with each theory solver and coordinates among them as Nelson–Oppen-style, to maintain soundness and completeness.

A general framework for doing this is known as *delayed theory combination (DTC)* [29, 37]. At the level of abstract transition systems described in Section 3.3, the essence of DTC is to work again with states of the form $M \parallel F$ except that now every atom occurring in M or in F is *pure*, i.e., in the signature of one of the theories $T_1, \dots, T_n$. A preprocessing purification step can be applied to the SMT solver's input to guarantee this for the initial formula F_0. The atoms in M come from F_0 or from the set S of all *interface equalities*, equalities between variables that occur in two atoms of F_0 belonging to different theories. The SAT engine is modified so that it also determines, by guessing, the truth value of the atoms in S, in addition to those in F_0. In its more general and advanced form, DTC benefits from changes to the theory solvers as well that enable them to propagate entailed interface equalities or disjunctions of them, thus reducing the SAT engine's guesswork. More details on DTC together with a study of its relative merits with respect to the encapsulation approach mentioned at the beginning can be found in [37]. A general abstract formulation of multi-theory lazy SMT that encompasses DTC is provided in [111].

Quantifiers Checking the satisfiability of quantified formulas is a long-standing challenge in SMT. A typical use of quantifiers in input formulas is to provide axiomatic definitions for function or predicate symbols not in the solver's background theories. In model checking applications, other uses of quantifiers include assertions involving all the elements of a collection datatype (such as arrays and sets) and assertions about concurrent systems (which for instance quantify over all processes). Extending decision procedures to such quantified formulas without losing termination is in general impossible because of basic undecidability results for first-order logic. In fact, even maintaining (refutational) completeness is already difficult, both in theory and in practice.

While the T-satisfiability of quantified formulas is decidable for certain theories T (such as, for instance, the theory of real numbers), their decision procedures use *quantifier elimination* methods, which convert formulas into T-equivalent quantifier-free ones, and are quite heavy computationally. Furthermore, these methods normally break down in the presence of additional symbols, such as uninterpreted ones. As a consequence, SMT solvers use incomplete methods based on instantiating quantified formulas into a set of *ground* ones.[16] Existential quantifiers in formulas of the form $\forall x_1 \cdots \forall x_n \, \exists x \, \varphi$ (with $n \geq 0$) are eliminated by dropping $\exists x$ and replacing all free occurrences of x in φ by the term $f(x_1, \ldots, x_n)$ where f is a fresh (uninterpreted) function symbol or arity n. Then, each universally quantified formula $\forall x \, \varphi$ is conjoined with a number of its *instances*, obtained from the qff φ by replacing its free occurrences of x with some ground term of the same sort. The selection of these instances is driven by incomplete heuristics.

The most common strategy is to select for instantiation ground terms that are *relevant* to $\forall x \, \varphi$, according to some heuristic relevance criterion. The SMT solver tries to find a subterm $t[x]$ of $\forall x \, \varphi$ properly containing x, a ground term g among those in its working memory, and a subterm s of g, such that $t[s]$, the result of replacing x by s in t, is T-equivalent to g. The expectation is that instantiating x with s is more likely to be helpful than instantiating it with an arbitrary ground term. In terms of unification theory [6], checking that $\models_T t[s] = g$ is a special case of *T-matching*. In the context of SMT, because of the richness of the background theory T, it may be very difficult if not impossible to determine whether an arbitrary term T and a ground term g T-match. As a result, most implementations use some form of T-matching only for uninterpreted terms. More details on this and on heuristics that are fairly effective in practice can be found in [23, 67, 86].

More recent work has focused on identifying fragments of first-order logic modulo theories for which is it possible to produce complete, and in some cases also terminating, quantifier-instantiation methods [73, 87, 147]. Some of this work [87] is based on a general *model-based quantifier instantiation* approach where the SMT solver maintains at all times (a finite representation of) a *candidate model*, a T-

[16] A term or formula is ground if it contains no variables (although it may contain uninterpreted constant symbols).

model that satisfies the current set G of ground formulas. The solver uses the candidate model to focus instance generation on only a few ground instances falsified by that model. Unless extending G with these instances makes G unsatisfiable, the solver then constructs a new candidate model for the extended G, and repeats the process until it is able to construct one that satisfies all quantified formulas as well.

A similar idea is used in the Inst-Gen calculus for first-order logic (with no theories) [82]. New ground instances are generated based on a model for the ground ones computed by an off-the-shelf SAT solver. A theorem prover based on this calculus has been shown to be very effective [108]. The Inst-Gen calculus has been extended to built-in theories [83]. However, implementing the extended calculus in an efficient solver has proven elusive so far.

A recent and quite promising line of work on model-based instantiation focuses on SMT formulas, all of whose quantifiers range over uninterpreted sorts [139, 140]. There, the solver tries to prove its input formula T-satisfiable by imposing finite cardinality constraints on those sorts, identifying for each sort σ a set U_σ of ground terms that enumerates the sort's finite domain, and instantiating quantifiers with these terms. The candidate model is used also to avoid exhaustive instantiation over each U_σ by identifying, and ignoring, whole sets of instances that are equisatisfiable with an already generated one.

Layered Theory Solvers Some theories T with a decidable constraint satisfiability problem contain less expressive fragments whose constraint satisfiability problem can be decided by more efficient methods. For example, the theory of real numbers includes a chain of increasingly larger and harder to decide fragments: inequalities between variables, difference constraints, linear constraints, and nonlinear constraints. For these theories, one can design a *layered* T-solver consisting of a sequence of subsolvers of decreasing performance but increasing generality [5, 31, 36, 57, 146]. In principle then, the solver can use the most efficient subsolver that is able to process the conjunction of literals given as input.

In reality, inputs rarely fall neatly in one of the fragments in the sequence. So abstraction and refinement techniques, similar in spirit to those used in model checking, must be used to take advantage of the faster subsolvers. Considering a nonincremental theory solver, for simplicity, the layering mechanism works as follows. The solver abstracts the literals in the input formula ψ as needed to get a formula ψ' T-entailed by ψ and accepted by its most restricted subsolver. If that subsolver determines ψ' to be T-unsatisfiable, the solver reports ψ to be T-unsatisfiable.[17] Otherwise, it refines ψ' just enough for it to fall into the fragment processed by the next more general subsolver, and sends it to that one, repeating the same process until a subsolver finds the refined formula ψ' unsatisfiable or ψ' gets refined to ψ.

Incomplete Theory Solvers For some theories—such as the theories of arrays, linear integer arithmetic, algebraic datatypes, and finite sets—the constraint satisfiability problem alone is NP-hard. To be refutationally complete then, a solver for one of those theories T must perform internal search and case splitting. In a DPLL(T)

[17] When a conflict set is required for ψ, it can be computed from one for ψ'.

setting, it is possible to use much simpler, albeit incomplete, T-solvers by delegating all search and case splitting to the SAT engine, a module already designed to do that efficiently.

The main idea, developed in the *splitting on demand* framework [11], is the following. Any time the T-solver needs to do a case analysis to determine the T-satisfiability of its input, it encodes the needed case split into a T-valid disjunction of literals, a theory lemma in effect. Then, instead of returning a sat/unsat answer, it returns the lemma demanding that the SAT engine process it—doing case splits on it as it would do with any other lemma. When the engine adds a literal from the lemma to its variable assignment, the literal will be asserted back to the T-solver, letting it proceed with that choice. After that, the T-solver either manages to determine the satisfiability of the newly extended input set or repeats the process with a new lemma. For termination, and overall completeness, there must be a finite upper bound on the number of splitting demands a T-solver needs to make for any given input before it is able to reply with sat or unsat. General sufficient conditions for the correctness of splitting on demand are discussed in [11].

Although splitting on demand simplifies the construction of theory solvers, it does not always provide the best performance. A discussion of this issue for real arithmetic solvers can be found in [96].

7 Eager Encodings to SAT

An alternative to the lazy approach to SMT is one usually referred to as the *eager* approach. It encompasses any technique that aims to fully reduce SMT problems to propositional satisfiability (SAT) problems via some kind of encoding. More formally, an eager SMT solver accepts a first-order formula φ in the signature of some theory T, generates a propositional formula ψ that is propositionally satisfiable iff φ is T-satisfiable, and then it feeds ψ to an off-the-shelf SAT solver.

Although the *lazy* approach is now predominant in SMT, mostly because of its flexibility and generality, efficient eager solvers do exist for a number of important theories. To give a sense of how some of them work, let us look at EUF.

Eager solvers for quantifier-free formulas in EUF can be constructed using *Ackermann's reduction* [1]. Suppose f is a unary function symbol (the generalization to n-ary symbols is straightforward) in an input formula φ, and let $\{f(t_1),\ldots,f(t_n)\}$ be the set of occurrences of f in φ. We introduce n new constant symbols $f_1,\ldots,f_n$ and replace each $f(t_i)$ with f_i in φ. Let φ' denote the result of this replacement. Then the formula $\varphi' \wedge \bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} (t_i = t_j \Rightarrow f_i = f_j)$ is satisfiable in EUF iff φ is. By repeating this process, all function symbols can be removed.[18]

To complete the eager translation, we must also remove all equality literals. One way to do this is by introducing propositional variables and transitivity constraints [45]. Suppose ψ is an EUF formula with equalities but no function symbols.

[18] A method due to Bryant can be used as an alternative that can sometimes be more efficient [41].

Let $S = \{s_1, \ldots, s_m\}$ be the set of all terms appearing in equalities. Let ψ' be the result of replacing each equality $s_i = s_j$ or $s_j = s_i$ where $i < j$ with a propositional variable $e_{i,j}$. Let G be an undirected graph on S with an edge between s_i and s_j iff $e_{i,j}$ appears in ψ'. Let G' be a *chordal* graph (no chord-free cycle of size 4 or more) obtained from G by adding arbitrary chords within cycles of size 4 or more. For each *triangle* (s_i, s_j, s_k), i.e., cycle of size 3 in G' with $i < j < k$, we add the following *transitivity constraint* to ψ': $((e_{i,j} \wedge e_{j,k}) \Rightarrow e_{i,k}) \wedge ((e_{i,j} \wedge e_{i,k}) \Rightarrow e_{j,k}) \wedge ((e_{i,k} \wedge e_{j,k}) \Rightarrow e_{i,j})$. The result is a propositional formula that is satisfiable iff ψ is satisfiable in EUF. For additional details on and extensions to this algorithm, see [110].

Example 7. Consider again the EUF example $\Phi = \{f(f(a)) = a, f(f(f(a))) = a, g(a) \neq g(f(a))\}$. After applying the Ackermann reduction, we have: $\{f_2 = a, f_3 = a, g_4 \neq g_5, a = f_1 \Rightarrow f_1 = f_2, a = f_2 \Rightarrow f_1 = f_3, f_1 = f_2 \Rightarrow f_2 = f_3, a = f_1 \Rightarrow g_4 = g_5\}$. The graph G is already chordal and has 4 triangles: (a, f_1, f_2), (a, f_1, f_3), (a, f_2, f_3), (f_1, f_2, f_3). Let $a_0 \equiv a$ and introduce the propositional terms $e_{i,j}$ according to the subscripts. Also, let $B_{i,j,k}$ be the transitivity constraint on $e_{i,j}$, $e_{j,k}$, $e_{i,k}$ introduced above. The set $\{e_{0,2},\ e_{0,3},\ \neg e_{4,5},\ e_{0,1} \Rightarrow e_{1,2},\ e_{0,2} \Rightarrow e_{1,3},\ e_{1,2} \Rightarrow e_{2,3},\ e_{0,1} \Rightarrow e_{4,5},\ B_{0,1,2},\ B_{0,1,3},\ B_{0,2,3},\ B_{1,2,3}\}$ of propositional formulas is satisfiable iff Φ is satisfiable. It is easy to see that this set is unsatisfiable: $e_{0,2}$ and $e_{0,3}$ must be true and $e_{4,5}$ must be false. The fifth constraint then implies that $e_{1,3}$ must also be true. But then $B_{0,1,3}$ entails $e_{0,1}$ which implies that $e_{4,5}$ must be true, a contradiction.

The UCLID solver [43, 44, 113] uses these and other techniques to solve (eagerly) problems specified in the *CLU logic*, a logic of Counter arithmetic with Lambda expressions and Uninterpreted functions. Other eager approaches have looked at small domain instantiations [134] and various fragments of arithmetic [148, 149]. As mentioned in Section 4.6, a common approach to construct solvers for the theory of bit vectors is to apply some rewriting to the input formula followed by bit blasting. This too is an instance of the eager approach.

8 Additional Functionalities of SMT Solvers

Arguably, the success of SMT solvers as embedded deductive reasoning engines is due in large part to the emergence of additional functionalities well beyond the mere checking of a formula's T-satisfiability. These functionalities are used extensively and with great benefit by tools such as model checkers, interactive provers, program verifiers, test case generators and so on. We discuss a selection of them next.

Models In many applications it is useful not only to know that a formula is T-satisfiable but also to obtain a witness of its T-satisfiability in the form of a T-interpretation (in the sense of Section 1.1) satisfying the formula. Fully representing first-order models such as T-interpretations finitarily, however, is challenging, when possible at all. Hence SMT solvers usually return only partial information, in the form of value assignments to selected symbols in the input formula. Furthermore, they restrict consideration only to models that permit a finitary representation of these values.[19] For instance, for the theory of arrays they only consider models that interpret array variables as *almost constant maps*, unary functions mapping all their inputs to the same value except for finitely many inputs. A similar restriction is adopted for EUF in computing the interpretation of function symbols.

Even with these restrictions, returning models may require strictly more work than just determining satisfiability. Depending on the theory, different approaches are possible. One approach, followed for instance by the CVC3 solver [17], is first to compute a partial model sufficient to establish the input formula's satisfiability, and then to do additional work as needed to extend that partial model to include values for symbols of interest (variables and function/predicate symbols) to the user. Another approach, followed for instance by the solvers Yices and Z3, is to instrument the theory solver to maintain some value for every symbol at all times, starting with some default assignment, and modifying the assignment as needed until it becomes a satisfying one. Yet another approach, which is implemented in CVC4 and is beneficial with quantified formulas whose quantifiers range only over uninterpreted sorts, is to explicitly construct models that interpret those sorts as finite sets [139].

Proofs For most applications that utilize SMT solvers, it is important to have confidence in their refutational soundness. Since proving the soundness of an SMT solver is unrealistic, due to the complexity of such tools, a reasonable approach is for the solver to accompany its `unsat` answers with a *certificate* that can be checked independently with much simpler and more trustworthy tools. This certificate is in general a proof of the input formula's unsatisfiability, expressed as a proof term in some suitable proof system.

With SMT solvers based on the lazy approach it is possible to produce proofs with a two-tiered structure, consisting of a propositional skeleton filled with several theory-specific subproofs. In these two-tiered proofs, the conclusion is reached by means of propositional inferences applied to a set of input formulas and *theory lemmas*. The latter are disjunctions of theory literals deduced from no assumptions, using proof rules specific to the background theory in question. The propositional skeleton is generated using techniques similar to those used by proof-producing SAT solvers (e.g., [2, 9]). The proofs of the various lemmas used as hypotheses in the propositional skeleton are produced typically using natural deduction inference rules with theory specific axioms [27, 76, 85, 122].

A major challenge for the field is to devise a common proof system for proof-producing SMT solvers. The wide diversity of theories and solving algorithms in SMT makes it difficult to find a single proof system that is universally good. One way to address this difficulty is to use a meta-language for specifying proof systems for SMT [151]. The advantage of a meta-language solution is that one can build an automatic proof checker generator that takes as input a proof system and generates an efficient proof checker for that system [130].

[19] These restrictions cause no a loss of generality with quantifier-free queries.

Unsatisfiable Cores Most SMT solvers allow the user to inquire about the joint T-satisfiability of a set of formulas. For T-unsatisfiable sets Φ, some solvers are able to return a *T-unsatisfiable core*, a possibly minimal subset of Φ that is also T-unsatisfiable. This functionality, which is useful in many applications, is patterned after the analogous one offered at the propositional level by many modern SAT solvers. Research on producing minimal or small T-unsatisfiable cores in SMT is not as extensive as in SAT. Current methods either are inspired by similar ones in the SAT literature or rely directly on propositional technology. Following Barrett *et al.*'s terminology [12], we can identify three main approaches.

In the *proof-based approach*, adopted by proof-producing SMT solvers such as CVC3 and MathSAT, a T-unsatisfiable core is extracted from the produced proof of unsatisfiability simply by collecting all the formulas of Φ that appear as premises in the proof. The size of the returned core depends on the sophistication of the proof-generation mechanism in producing compact proofs. This approach requires only a small additional implementation effort but incurs the (heavy) cost of producing a proof, even when none is requested.

In the *assumption-based approach*, implemented in Yices and applicable to any DPLL(T)-style solver, the input set $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ is internally converted into the equisatisfiable set $\{p_1 \Rightarrow \varphi_1, \ldots, p_n \Rightarrow \varphi_n, p_1, \ldots, p_n\}$ where each p_i is a fresh propositional symbol. Then, the same conflict analysis mechanism used by the DPLL engine can be used to identify a subset of $\{p_1, \ldots, p_n\}$ that caused the last conflict. The returned T-unsatisfiable core consists of the corresponding φ_i's.

In the *lemma-lifting approach* [54], implemented in more recent versions of MathSAT, one uses the fact that a DPLL(T) solver will discover the T-unsatisfiability of Φ by adding theory lemmas until Φ becomes propositionally unsatisfiable. Once the DPLL engine detects that, any external propositional core extractor can be used to produce an unsatisfiable core C for the propositional abstraction $\{\varphi^{\mathrm{a}} \mid \varphi \in \Phi\}$ of the extended Φ. The returned T-unsatisfiable core consists then of $\{\varphi \mid \varphi^{\mathrm{a}} \in C\}$, the formulas of Φ whose abstraction is in C.

Interpolants A fundamental result in model theory due to Craig [58] asserts the existence of an *interpolant* for every pair of first-order formulas A and B such that $A \models B$. This is a formula I written using only logical symbols and symbols occurring in both A and B such that $A \models I$ and $I \models B$. Analogues of this result, expressed in terms of unsatisfiability instead of entailment, hold for a variety of logics and logic fragments. In the SMT case, the result states that for all first-order theories T and formulas A,B such that $A,B \models_T \bot$, there is a formula I using only logical symbols, symbols of T and symbols occurring in both A and B such that $A \models_T I$ and $I,B \models_T \bot$.

Starting with the seminal work by McMillan [118], interpolants have found a number of practical uses in model checking (see Chap. 14A). Applications involve the computation of interpolants in propositional logic or in logics with (combinations of) theories such as the theory of equality, linear rational arithmetic, arrays, and finite sets [52, 103, 120, 159].

In propositional logic, interpolants can be computed from resolution proofs using a simple method due to Pudlák [136]. For theories T with the *quantifier-free inter-*

polation property, which guarantees the existence of quantifier-free interpolants for any T-unsatisfiable pair A, B of quantifier-free formulas, interpolants can be computed using SMT techniques. In many cases, it is possible to produce interpolants efficiently by modifying existing theory solvers in relatively minor ways [52, 78, 142].

Under fairly general conditions, the generation of theory interpolants for sets of literals can be extended modularly to (i) sets of arbitrary quantifier-free formulas and (ii) combinations of theories (each with the quantifier-free interpolation property), thanks to a method by Yorsh and Musuvathi [159]. This allows one to turn an SMT solver into an interpolant generator. The first extension is possible with SMT solvers that produce the sort of two-tiered proofs mentioned earlier in this section, and relies on an adaptation Pudlák's method to deal with the proof's propositional skeleton. The second extension additionally requires each component theory T to be *equality-interpolating*: whenever $A, B \models_T r = t$ where r is a term occurring in A and T a term occurring in B, it is possible to compute a term s in the language shared by A and B such that $A, B \models_T r = s \wedge s = t$. A further, and related, requirement is that the unsatisfiability proof from which to extract the interpolant contain no *AB-mixed* literals, literals with symbols occurring only in A and symbols occurring only in B. Unfortunately, typical SMT solvers do not guarantee the absence of *AB*-mixed literals from their proofs.[20] Initial implementations of the Yorsh-Musuvathi method imposed restrictions on solver search strategies in order to produce proofs of a certain shape from which it is possible to extract interpolants even in the presence of *AB*-mixed literals [52]. In later work, these restrictions, and their potential performance penalty, have been increasingly and considerably reduced by relying on a certain amount of proof post-processing [39, 53, 94] or by considering only certain classes of theories [50].

References

1. W. Ackermann. *Solvable Cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. H. Amjad. A Compressing Translation from Propositional Resolution to Natural Deduction. In *Proceedings of the 6th Symposium on Frontiers of Combining Systems (FroCoS'07)*, volume 4720 of *LNCS*, pages 88–102. Springer, 2007.
3. A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proceedings of the 5th European Conference on Planning (ECP'99)*, volume 1809 of *LNCS*, pages 97–108. Springer, 2000.
4. A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *Proceedings of the 13th International SPIN Workshop on Model Checking of Software (SPIN'06)*, volume 3925 of *LNCS*, pages 146–162. Springer, 2006.
5. 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 *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *LNAI*, pages 195–210. Springer, 2002.

[20] Both Delayed Theory Combination and Splitting on Demand generate new literals during a proof which may be *AB*-mixed.

6. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
7. D. Babić. *Exploiting Structure for Scalable Software Verification*. PhD thesis, University of British Columbia, Vancouver, Canada, 2008.
8. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 203–213, 2001.
9. C. Barrett and S. Berezin. A proof-producing Boolean search engine. In *Proceedings of the* 1^{st} *International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR'03)*, 2003.
10. C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
11. C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'06), Phnom Penh, Cambodia*, volume 4246 of *LNCS*, pages 512–526. Springer, 2006.
12. C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185, chapter 26, pages 825–885. IOS Press, February 2009.
13. 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.
14. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.smtlib.org`, 2010.
15. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (SMT'10)*, 2010.
16. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.smtlib.org`.
17. C. Barrett and C. Tinelli. CVC3. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, LNCS. Springer, 2007.
18. C. W. Barrett, L. de Moura, and A. Stump. Design and results of the first satisfiability modulo theories competition (SMT-COMP 2005). *Journal of Automated Reasoning*, 35(4):373–390, 2005.
19. C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the* 35^{th} *Design Automation Conference (DAC '98)*, pages 522–527. Association for Computing Machinery, 1998. San Francisco, California.
20. C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proceedings of the International Conference on Computer-Aided Verification*, LNCS, 2002.
21. S. Berezin, V. Ganesh, and D. L. Dill. An online proof-producing decision procedure for mixed-integer linear arithmetic. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 521–536. Springer, 2003.
22. A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.
23. N. Bjørner and L. de Moura. Efficient E-matching for SMT solvers. In *Proceedings of the* 21^{st} *International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNAI*, pages 183–198. Springer, 2007.
24. N. Bjørner and M. Pichora. Deciding fixed and non-fixed size bit-vectors. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*, pages 376–392. Springer, 1998.

25. F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing polymorphism in SMT solvers. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, pages 1–5. ACM, 2008.
26. M. Bofill, R. Nieuwenhuis, A. Oliveras, E. R. Carbonell, and A. Rubio. A write-based solver for SAT modulo the theory of arrays. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, FMCAD '08, Piscataway, NJ, USA, 2008. IEEE Press.
27. S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In *Proceedings of the International Conference on Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.
28. T. Bouton, D. C. B. De Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In *Automated Deduction (CADE-22)*, pages 151–156. Springer, 2009.
29. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, R. Sebastiani, and P. van Rossu. Efficient satisfiability modulo theories via delayed theory combination. In *Proceedings of the 17th International Conference on Computer Aided Verification*, LNCS. Springer, 2005.
30. 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):1411–1596, 2006.
31. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 317–333, 2005.
32. A. Bradley, Z. Manna, and H. Sipma. What's Decidable About Arrays? In *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *LNCS*, chapter 28, pages 427–442. Springer, 2006.
33. M. Brain, V. D'Silva, L. Haller, A. Griggio, and D. Kroening. An abstract interpretation of DPLL(T). In *Verification, Model Checking, and Abstract Interpretation*, pages 455–475. Springer, 2013.
34. R. Brummayer and A. Biere. Lemmas on demand for the extensional theory of arrays. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, SMT '08/BPR '08, pages 6–11, New York, NY, USA, 2008. ACM.
35. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *LNCS*, chapter 16, pages 174–177. Springer, 2009.
36. 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 *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification*, volume 4590 of *LNCS*, pages 547–560. Springer, 2007.
37. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. Delayed theory combination vs. Nelson-Oppen for satisfiability modulo theories: a comparative analysis. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):63–99, 2009.
38. R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The openSMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 150–153. Springer, 2010.
39. R. Bruttomesso, S. Rollini, N. Sharygina, and A. Tsitovich. Flexible interpolation with local proof transformations. In *Proceedings of the 2010 International Conference on Computer-Aided Design*, pages 770–777. IEEE, 2010.
40. R. Bruttomesso and N. Sharygina. A scalable decision procedure for fixed-width bit-vectors. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 13–20, New York, NY, USA, 2009. ACM.
41. R. Bryant, S. German, and M. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Computer Aided Verification*, volume 1633 of *LNCS*, pages 470–482. Springer, 1999.

42. R. Bryant, D. Kroening, J. Ouaknine, S. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *LNCS*, pages 358–372. Springer, 2007.
43. R. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer Aided Verification*, volume 2404 of *LNCS*, pages 106–122. Springer, 2002.
44. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Deciding CLU logic formulas via Boolean and pseudo-Boolean encodings. In *In Proc. Intl. Workshop on Constraints in Formal Verification (CFV'02)*, 2002.
45. R. E. Bryant and M. N. Velev. Boolean satisfiability with transitivity constraints. *ACM Trans. Comput. Logic*, 3(4):604–627, Oct. 2002.
46. D. Cantone and C. Zarba. A New Fast Tableau-Based Decision Procedure for an Unquantified Fragment of Set Theory. In *Automated Deduction in Classical and Non-Classical Logics*, volume 1761 of *LNCS*, chapter 8, pages 492–495. Springer, May 2000.
47. S. Chatterjee, S. Lahiri, S. Qadeer, and Z. Rakamarić. A Reachability Predicate for Analyzing Low-Level Software. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *LNCS*, chapter 4, pages 19–33. Springer, 2007.
48. B. V. Cherkassy and A. V. Goldberg. Negative-cycle detection algorithms. In *European Symposium on Algorithms*, pages 349–363, 1996.
49. J. Christ, J. Hoenicke, and A. Nutz. SMTinterpol: An interpolating SMT solver. In *Model Checking Software - 19th International Workshop*, volume 7385 of *LNCS*, pages 248–254. Springer, 2012.
50. J. Christ, J. Hoenicke, and A. Nutz. Proof tree preserving interpolation. In N. Piterman and S. Smolka, editors, *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*, volume 7795 of *LNCS*, pages 124–138. Springer, 2013.
51. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.
52. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS. Springer, 2008.
53. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient generation of Craig interpolants in satisfiability modulo theories. *ACM Transactions on Computational Logic*, 12(1):7, 2010.
54. A. Cimatti, A. Griggio, and R. Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *Journal of Artificial Intelligence Research*, 40(1):701–728, 2011.
55. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
56. S. Cotton and O. Maler. Fast and Flexible Difference Constraint Propagation for DPLL(T). In *Theory and Applications of Satisfiability Testing - SAT 2006*, volume 4121 of *LNCS*, chapter 19, pages 170–183. Springer, 2006.
57. S. Cotton and O. Maler. Satisfiability modulo theory chains with DPLL(T). Research Report TR2006-04, Verimag, 2006.
58. W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, sep 1957.
59. D. Cyrluk, O. Möller, and H. Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Computer Aided Verification*, volume 1254 of *LNCS*, pages 60–71. Springer, 1997.
60. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
61. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
62. L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, pages 337–340. Springer, 2008.

63. L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *Formal Methods in Computer-Aided Design (FMCAD'09)*, pages 45–52, 2009.
64. L. de Moura and N. S. Bjørner. Model-based theory combination. In *Proc. 5th workshop on Satisfiability Modulo Theories, SMT'07*, 2007.
65. L. de Moura and H. Rueß. Lemmas on demand for satisfiability solvers. In *Proc. of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*, 2002.
66. L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV'03)*, volume 2725 of *LNCS*. Springer, 2003.
67. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of ACM*, 52(3):365–473, 2005.
68. I. Dillig, T. Dillig, and A. Aiken. Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers. In *Computer Aided Verification*, volume 5643 of *LNCS*, chapter 20, pages 233–247. Springer, 2009.
69. P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the Common Subexpression Problem. *J. ACM*, 27(4):758–771, October 1980.
70. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.
71. B. Dutertre and L. de Moura. Integrating simplex with DPLL(T). Technical report, CSL, SRI International, 2006.
72. B. Dutertre and L. de Moura. The YICES SMT solver. Technical report, SRI International, 2006.
73. M. Echenim and N. Peltier. An instantiation scheme for satisfiability modulo theories. *Journal of Automated Reasoning*, 2010.
74. H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2nd edition, 2001.
75. C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In *Proceedings of the 15th International Conference on Computer Aided Verification*, volume 2725 of *LNCS*, pages 355–367. Springer, 2003.
76. P. Fontaine, J.-Y. Marion, S. Merz, L. P. Nieto, and A. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In *Proceedings of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 167–181. Springer, 2006.
77. A. Fröhlich, G. Kovásznai, and A. Biere. More on the complexity of quantifier-free fixed-size bit-vector logics with binary encoding. In *Computer Science–Theory and Applications*, pages 378–390. Springer, 2013.
78. A. Fuchs, A. Goel, J. Grundy, S. Krstić, and C. Tinelli. Ground interpolation for the theory of equality. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 413–427. Springer, 2009.
79. V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, pages 519–531. Springer, 2007.
80. V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, volume 4590 of *LNCS*, pages 519–531. Springer, 2007.
81. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts)*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.
82. H. Ganzinger and K. Korovin. New directions in instantiation-based theorem proving. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science (LICS'03)*, pages 55–64. IEEE Computer Society Press, 2003.
83. H. Ganzinger and K. Korovin. Theory Instantiation. In *Proceedings of the 13th Conference on Logic for Programming Artificial Intelligence Reasoning (LPAR'06)*, volume 4246 of *LNCS*, pages 497–511. Springer, 2006.

84. S. Gao, S. Kong, and E. M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In *Automated Deduction (CADE-24)*, volume 7898 of *LNCS*, pages 208–214. Springer, 2013.
85. Y. Ge and C. Barrett. Proof translation and SMT-LIB benchmark certification: A preliminary report. In *Proceedings of SMT'08*, 2008.
86. Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *Proceedings of the 21st International Conference on Automated Deduction (CADE-21), Bremen, Germany*, LNCS. Springer, 2007.
87. Y. Ge and L. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 306–320. Springer, 2009.
88. S. Ghilardi. Model-theoretic methods in combined constraint satisfiability. *Journal of Automated Reasoning*, 33(3–4):221–249, 2005.
89. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Combination methods for satisfiability and model-checking of infinite-state systems. In *Proceedings of the 21st International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNCS*, pages 362–378. Springer, 2007.
90. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT model checking of array-based systems. In *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR 2008)*, volume 5195 of *LNCS*, pages 67–82, Sydney (Australia), 2008. Springer.
91. S. Ghilardi, E. Nicolini, and D. Zucchelli. A comprehensive combination framework. *ACM Transactions on Computational Logic*, 9(2):1–54, 2008.
92. S. Ghilardi and S. Ranise. MCMT: A model checker modulo theories. In *Proceedings of the 5th International Joint Conference Automated Reasoning (IJCAR'10)*, volume 6173 of *LNCS*, pages 22–29. Springer, 2010.
93. A. Goel, S. Krstić, and A. Fuchs. Deciding array formulas with frugal axiom instantiation. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, SMT '08/BPR '08, pages 12–17, New York, NY, USA, 2008. ACM.
94. A. Goel, S. Krstić, and C. Tinelli. Ground interpolation for combined theories. In *Proceedings of the 22nd International Conference on Automated Deduction (CADE-22)*, volume 5663 of *LNAI*, pages 183–198. Springer, 2009.
95. A. Griggio. *An Effective SMT Engine for Formal Verification*. PhD thesis, DISI, University of Trento, December 2009.
96. A. Griggio. A Practical Approach to SMT(LA(Z)). In *Proceedings of the 2010 SMT Workshop*, 2010.
97. G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAV'08), Portland, Oregon*, pages 109–117. IEEE, 2008.
98. S. K. Jha, R. S. Limaye, and S. A. Seshia. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. Technical Report UCB/EECS-2009-95, EECS Department, University of California, Berkeley, Jun 2009.
99. R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In *Proceedings of 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 39–51. Springer, 2005.
100. D. Jovanović and C. Barrett. Polite theories revisited. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 6397 of *LNCS*, pages 402–416. Springer, 2010.
101. D. Jovanović, C. Barrett, and L. de Moura. The design and implementation of the model constructing satisfiability calculus. In *Formal Methods in Computer-Aided Design (FMCAD'13)*. ACM/IEEE, 2013.
102. D. Jovanović and L. de Moura. Solving non-linear arithmetic. In *Automated Reasoning*, volume 7364 of *LNCS*, pages 339–354. Springer, 2012.

103. D. Kapur, R. Majumdar, and C. Zarba. Interpolation for data structures. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 105–116. ACM, 2006.
104. D. Kapur and C. G. Zarba. A Reduction Approach to Decision Procedures. Technical Report TR-CS-1005-44, University of New Mexico, 2005.
105. A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 105–116, New York, NY, USA, 2009. ACM.
106. H. Kim and F. Somenzi. Finite Instantiations for Integer Difference Logic. In *Formal Methods in Computer Aided Design (FMCAD'06)*, pages 31–38, 2006.
107. T. King, C. Barrett, and B. Dutertre. Sum of infeasibility simplex for SMT. In *Proceedings of the* 13^{th} *International Conference on Formal Methods In Computer-Aided Design (FMCAD'13)*, LNCS, 2013.
108. K. Korovin. iProver – an instantiation-based theorem prover for first-order logic (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR 2008)*, volume 5195 of *LNCS*, pages 292–298. Springer, 2008.
109. K. Korovin, N. Tsiskaridze, and A. Voronkov. Conflict resolution. In I. Gent, editor, *Principles and Practice of Constraint Programming (CP 2009)*, volume 5732 of *LNCS*, pages 509–523. Springer, 2009.
110. D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.
111. S. Krstić and A. Goel. Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In *Proceeding of the Symposium on Frontiers of Combining Systems (FroCoS'07)*, volume 4720 of *LNCS*, pages 1–27. Springer, 2007.
112. S. Krstić, A. Goel, J. Grundy, and C. Tinelli. Combined satisfiability modulo parametric theories. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Braga, Portugal)*, volume 4424 of *LNCS*, pages 618–631. Springer, 2007.
113. S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Formal Methods in Computer-Aided Design*, volume 2517 of *LNCS*, pages 142–159. Springer, 2002.
114. T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating Reachability Using First-Order Logic with Applications to Verification of Linked Data Structures. In *Automated Deduction (CADE-20)*, volume 3632 of *LNCS*, chapter 8, pages 99–115. Springer, 2005.
115. Z. Manna and C. G. Zarba. Combining Decision Procedures. In *Formal Methods at the Crossroads: From Panacea to Foundational Support*, volume 2757 of *LNCS*, chapter 24, pages 381–422. Springer, 2003.
116. P. Manolios, S. Srinivasan, and D. Vroon. BAT: The Bit-Level Analysis Tool. In *Computer Aided Verification*, volume 4590 of *LNCS*, chapter 35, pages 303–306. Springer, 2007.
117. M. Manzano. Introduction to many-sorted logic. In *Many-sorted logic and its applications*, pages 3–86. John Wiley & Sons, Inc., 1993.
118. K. McMillan. Interpolation and SAT-based model checking. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.
119. K. L. McMillan. Applications of Craig interpolants in model checking. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 1–12. Springer, 2005.
120. K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.
121. K. L. McMillan, A. Kuehlmann, and M. Sagiv. Generalizing DPLL to richer logics. In *Computer Aided Verification*, volume 5643 of *LNCS*, pages 462–476, 2009.

122. M. Moskal. Rocket-fast proof checking for SMT solvers. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 486–500. Springer, 2008.
123. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.
124. G. Nelson and D. C. Oppen. Fast Decision Procedures Based on Congruence Closure. *Journal of the ACM*, 27(2):356–364, April 1980.
125. R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic. In *Computer Aided Verification*, volume 3576 of *LNCS*, chapter 33, pages 321–334. Springer, 2005.
126. R. Nieuwenhuis and A. Oliveras. On SAT Modulo Theories and Optimization Problems. In *Theory and Applications of Satisfiability Testing (SAT'06)*, volume 4121 of *LNCS*, chapter 18, pages 156–169. Springer, 2006.
127. R. Nieuwenhuis and A. Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, April 2007.
128. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04)*, volume 3452 of *LNCS*, pages 36–50. Springer, 2005.
129. 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.
130. D. Oe, A. Reynolds, and A. Stump. Fast and flexible proof checking for SMT. In *Proceedings of SMT'09*, 2009.
131. D. C. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, 12:291–302, 1980.
132. J. Peleska, E. Vorobev, and F. Lapschies. Automated test case generation with SMT-solving and abstract interpretation. In *NASA Formal Methods*, pages 298–312. Springer, 2011.
133. R. Piskac and V. Kuncak. Decision procedures for multisets with cardinality constraints. In *Proceedings of the 9th international conference on Verification, model checking, and abstract interpretation*, VMCAI'08, pages 218–232. Springer, 2008.
134. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In *Computer Aided Verification*, volume 1633 of *LNCS*, pages 687–688. Springer, 1999.
135. M. Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929. Warszawa.
136. P. Pudlák. Lower bounds for resolution and cutting planes proofs and monotone computations. *Journal of Symbolic Logic*, 62(3), 1997.
137. W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.
138. S. Ranise, C. Ringeissen, and C. G. Zarba. Combining data structures with nonstably infinite theories using many-sorted logic. In *Proceedings of the Workshop on Frontiers of Combining Systems (FroCoS'05)*, LNCS. Springer, 2005.
139. A. Reynolds, C. Tinelli, A. Goel, and S. Krstić. Finite model finding in SMT. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*, volume 8044 of *LNCS*, pages 640–655. Springer, 2013.
140. A. Reynolds, C. Tinelli, A. Goel, S. Krstić, M. Deters, and C. Barrett. Quantifier instantiation techniques for finite model finding in SMT. In M. P. Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *LNCS*, pages 377–391. Springer, 2013.
141. P. Rümmer and T. Wahl. An SMT-LIB theory of binary floating-point arithmetic. In *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland*, 2010.

142. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. *Journal of Symbolic Computation*, 45:1212–1233, November 2010.
143. A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.
144. R. Sebastiani. Lazy satisability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3(3-4):141–224, 2007.
145. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125, London, UK, 2000. Springer.
146. H. M. Sheini and K. A. Sakallah. A scalable method for solving satisfiability of integer linear arithmetic logic. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, volume 3569 of *LNCS*. Springer, 2005.
147. V. Sofronie-Stokkermans. Hierarchic reasoning in local theory extensions. In *Automated Deduction (CADE-20)*, pages 219–234. Springer, 2005.
148. O. Strichman. On solving Presburger and linear arithmetic with SAT. In *Formal Methods in Computer-Aided Design*, volume 2517 of *LNCS*, pages 160–170. Springer, 2002.
149. O. Strichman, S. Seshia, and R. Bryant. Deciding separation formulas with SAT. In *Computer Aided Verification*, volume 2404 of *LNCS*, pages 113–124. Springer, 2002.
150. A. Stump, C. W. Barrett, D. L. Dill, and J. Levitt. A decision procedure for an extensional theory of arrays. In *Proceedings of the* 16^{th} *IEEE Symposium on Logic in Computer Science (LICS'01)*, pages 29–37. IEEE Computer Society, 2001. Boston, Massachusetts.
151. A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 41(1):91–118, Feb. 2013.
152. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.
153. C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In G. Ianni and S. Flesca, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, volume 2424 of *LNAI*. Springer, 2002.
154. C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In *Frontiers of Combining Systems: Proceedings of the 1st International Workshop (FroCoS'96)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, 1996.
155. C. Tinelli and C. Ringeissen. Unions of non-disjoint theories and combinations of satisfiability procedures. *Theoretical Computer Science*, 290(1):291–353, Jan. 2003.
156. C. Tinelli and C. Zarba. Combining decision procedures for sorted theories. In *Proceedings of the 9th European Conference on Logic in Artificial Intelligence (JELIA'04)*, volume 3229 of *LNAI*, pages 641–653. Springer, 2004.
157. C. Tinelli and C. Zarba. Combining nonstably infinite theories. *Journal of Automated Reasoning*, 34(3):209–238, Apr. 2005.
158. C. Wang, A. Gupta, and M. Ganai. Predicate learning and selective theory deduction for a difference logic solver. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 235–240, New York, NY, USA, 2006. ACM.
159. G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In *Proceedings of the 20th International Conference on Automated Deduction (CADE-20)*, volume 3632 of *LNCS*, pages 353–368. Springer, 2005.

Index