Satisfiability Modulo Theories: An Appetizer

Leonardo de Moura and Nikolaj Bjørner
{leonardo, nbjorner}@microsoft.com
Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA.

Abstract

Satisfiability Modulo Theories (SMT) is about checking the satisfiability of logical formulas over one or more theories. The problem draws on a combination of some of the most fundamental areas in computer science. It combines the problem of Boolean satisfiability with domains, such as, those studied in convex optimization and term-manipulating symbolic systems. It also draws on the most prolific problems in the past century of symbolic logic: the decision problem, completeness and incompleteness of logical theories, and finally complexity theory. The problem of modularly combining special purpose algorithms for each domain is as deep and intriguing as finding new algorithms that work particularly well in the context of a combination. SMT also enjoys a very useful role in software engineering. Modern software, hardware analysis and model-based tools are increasingly complex and multi-faceted software systems. However, at their core is invariably a component using symbolic logic for describing states and transformations between them. A well tuned SMT solver that takes into account the state-of-the-art breakthroughs usually scales orders of magnitude beyond custom ad-hoc solvers.

1 Introduction

Satisfiability is one of the most fundamental problems in theoretical computer science, namely the problem of determining whether a formula expressing a constraint has a solution. Constraint satisfaction problems arise in many diverse areas including software and hardware verification, type inference, extended static checking, test-case generation, scheduling, planning, graph problems, among others [1]. The most well-known constraint satisfaction problem is *propositional satisfiability* SAT, where the goal is to decide whether a formula over Boolean variables, formed using logical connectives, can be made true by choosing true/false values for its variables.

Some problems require or are more naturally described in more expressive logics such as first-order logic. A first-order formula is formed using logical connectives, variables, quantifiers, function and predicate symbols. A solution, also known as a *model*, is an interpretation for the variable, function and predicate symbols that makes the formula true. Of particular recent interest is *satisfiability modulo theories* (SMT), where the interpretation of some symbols is constrained by a *background theory*. For example, the theory of *arithmetic* restricts the interpretation of symbols such as: $+$, $\leq$, 0, and 1.

SMT draws on the most prolific problems in the past century of symbolic logic: the decision problem, completeness and incompleteness of logical theories, and finally complexity theory. The computational complexity of most SMT problems is very high. The problem of modularly *combining special purpose algorithms* for each domain is as deep and intriguing as finding new algorithms that work particularly well in the context of a combination. The theory of linear arithmetic, which is the basis of linear programming, is one prominent theory that is useful in many applications. Linear programming algorithms can be used to check satisfiability of conjunctions of linear arithmetic inequalities, but they do not directly apply for Boolean combinations. SMT solvers distinguish themselves by handling such combinations.

It is well-known that SAT is NP-complete and first-order logic is undecidable. Due to this high computational complexity, it is infeasible to build a procedure that can solve arbitrary SMT problems. Therefore, most procedures focus on the more realistic goal of efficiently solving problems that occur in practice. They rely on the assumption that, although potentially big, most formulas produced by verification and analysis tools are *shallow*. That is, only a small fraction of a formula is really critical for establishing satisfiability. The rest consists of irrelevant noise.

In recent years, there has been an enormous progress in the scale of problems that can be solved, thanks to innovations in core algorithms, data structures, heuristics, and paying attention to implementation details. Modern SAT procedures can check formulas with hundreds of thousands variables and millions of clauses. A similar progress has being observed for SMT procedures for the more commonly occurring theories. The annual competition for SAT and SMT procedures is a key ingredient in driving progress [2]. In this paper, we provide a brief overview of SMT and the main technical ideas.

1.1 An Example

We will introduce three theories used in SMT solvers using the following example:

$$b + 2 = c \wedge f(\mathit{read}(\mathit{write}(a, b, 3), c - 2)) \neq f(c - b + 1).$$

The formula uses the theory of arrays. It was introduced by McCarthy in [3] as part of forming a broader agenda for a calculus of computation. In the theory of arrays, there are two functions *read* and *write*. The term $\mathit{read}(a, i)$ produces the value of array a at index i, while the term $\mathit{write}(a, i, v)$ produces an array, which is equal to a except for possibly index i which maps to v. These properties can be summarized using the equations:

$$\begin{array}{rcl} \mathit{read}(\mathit{write}(a, i, v), i) & = & v \\ \mathit{read}(\mathit{write}(a, i, v), j) & = & \mathit{read}(a, j) \;\; \text{for } i \neq j. \end{array}$$

They state that the result of reading $\mathit{write}(a, i, v)$ at index j is v for $i = j$. Reading the array at any other index produces the same value as $\mathit{read}(a, j)$. The formula also uses the function f, therefore for all t and s, if $t = s$, then $f(t) = f(s)$ (congruence rule). In other words, the only assumption about function f is that it always produce the same result when applied to the same arguments. The congruence rule implies that formulas remain equivalent when replacing equal terms. The example formula is unsatisfiable. That is, there is no assignment to the integers b and c and the array a such that the first equality $b + 2 = c$ holds and at the same time the second disequality also is satisfied. One way of establishing the unsatisfiability is by replacing c by $b + 2$ in the disequality, to obtain the equivalent

$$b + 2 = c \wedge f(\mathit{read}(\mathit{write}(a, b, 3), b + 2 - 2)) \neq f(b + 2 - b + 1),$$

which after reduction using facts about arithmetic becomes

$$b + 2 = c \wedge f(\mathit{read}(\mathit{write}(a, b, 3), b)) \neq f(3).$$

The theory of arrays implies that the nested array read/write functions reduce to 3 and the formula becomes:

$$b + 2 = c \wedge f(3) \neq f(3).$$

The congruence property of f entails that the disequality is false.

2 Preliminaries

A *propositional formula* φ can be a propositional variable p or a negation $\neg\varphi_0$, a conjunction $\varphi_0 \wedge \varphi_1$, a disjunction $\varphi_0 \vee \varphi_1$, an implication $\varphi_0 \Rightarrow \varphi_1$, or a bi-implication $\varphi_0 \Leftrightarrow \varphi_1$ of smaller formulas φ_0, φ_1. A *truth assignment* M for a formula φ maps the propositional variables in φ to $\{\mathsf{true}, \mathsf{false}\}$. We say a truth assignment M *satisfies* φ ($M \models \varphi$), if M makes φ evaluate to true under the usual truth table interpretation of the connectives. For instance, let φ be the formula $p \vee (\neg q \wedge r)$, then the truth assignment $M = \{p \mapsto \mathsf{false}, q \mapsto \mathsf{false}, r \mapsto \mathsf{true}\}$ satisfies φ. A formula φ is *satisfiable* if there is an M s.t. $M \models \varphi$, and φ is *valid* if for all M, $M \models \varphi$. We say φ_1 and φ_2 are *equisatisfiable* if φ_1 is satisfiable iff φ_2 is satisfiable. A literal is either a propositional variable p or its negation $\neg p$. A clause is a disjunction of literals $l_1 \vee \ldots \vee l_n$. A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses $C_1 \wedge \ldots \wedge C_m$. We will write CNF formulas as set of clauses. Any propositional formula can be converted to CNF, in linear time, by introducing fresh variables for each compound subformula and adding suitable clauses. For example, let φ be the formula $\neg p \vee (q \wedge \neg r)$, in converting φ into CNF, we label $q \wedge \neg r$ as k_1 and encode $k_1 \Leftrightarrow (q \wedge \neg r)$ using the set of auxiliary clauses $\Delta_1 = \{\neg k_1 \vee q,\ \neg k_1 \vee \neg r,\ \neg q \vee r \vee k_1\}$, similarly, we label $\neg p \vee k_1$ as k_2 and encode $k_2 \Leftrightarrow (\neg p \vee k_1)$ using the clauses $\Delta_2 = \{p \vee k_2,\ \neg k_1 \vee k_2,\ \neg k_2 \vee \neg p \vee k_1\}$, hence, the formula φ is equisatisfiable to the set of clauses $\{k_2\} \cup \Delta_1 \cup \Delta_2$.

Many-sorted (first-order) logic is a commonly used formalism and framework for formulating SMT problems. A many-sorted *signature* is composed of a set of *sorts*, a set of *function symbols*, and a set of *predicate symbols*. Each function symbol f has associated with it an arity of the form $\sigma_1 \times \ldots \times \sigma_n \to \sigma$, where $\sigma_1, \ldots, \sigma_n, \sigma$ are sorts. If $n = 0$, we say f is a constant symbol. Similarly, each predicate symbol p has associated with it an arity of the form $\sigma_1 \times \ldots \times \sigma_n$. If $n = 0$, we say p is a propositional symbol. We assume a set of *variables* X, where each variable is associated with a sort. A *term* t with sort σ has the form x or $f(t_1, \ldots, t_n)$, where x is a variable with sort σ, and f is a function symbol with arity $\sigma_1 \times \ldots \times \sigma_n \to \sigma$, where for each $i \in \{1, \ldots, n\}$, t_i has sort σ_i. An *atom* is of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol with arity $\sigma_1 \times \ldots \times \sigma_n$, and for each $i \in \{1, \ldots, n\}$, t_i is a term with sort σ_i. A *formula* φ is an atom, or has the form $\neg\varphi_0$, $\varphi_0 \wedge \varphi_1$, $\varphi_0 \vee \varphi_1$, $\varphi_0 \Rightarrow \varphi_1$, $\varphi_0 \Leftrightarrow \varphi_1$, $(\forall x\colon \sigma. \varphi_0)$, or $(\exists x\colon \sigma. \varphi_0)$, where φ_0, φ_1 are smaller formulas. A Σ-*formula* φ is a formula where each symbol in φ is in Σ. We say a variable x is *free* in formula φ if it is not bound by any quantifier $\exists, \forall$. For example, x is free in $(\forall y\colon \sigma. p(x, y))$, but y is not.

A *sentence* is a formula without free variables. We use $vars(\varphi)$ to denote the set of free variables in φ. A *quantifier-free formula* is a formula not containing $\exists$ or $\forall$.

A *structure* M for a signature Σ and variables X consists of non-empty domains $|M|_\sigma$ for each sort in Σ, for each $x \in X$ with sort σ, $M(x) \in |M|_\sigma$, for each function symbol f with arity $\sigma_1 \times \ldots \times \sigma_n \to \sigma$, $M(f)$ is a total map from $|M|_{\sigma_1} \times \ldots \times |M|_{\sigma_n}$ to $|M|_\sigma$, and for each predicate symbol p with arity $\sigma_1 \times \ldots \times \sigma_n$, $M(p)$ is a subset of $|M|_{\sigma_1} \times \ldots \times |M|_{\sigma_n}$. The interpretation of a term t is given by $M[\![x]\!] = M(x)$ and $M[\![f(t_1, \ldots, t_n)]\!] = M(f)(M[\![t_1]\!], \ldots, M[\![t_n]\!])$. We assume that, for each sort σ, the equality $=_\sigma$ is a *builtin* predicate symbol with arity $\sigma \times \sigma$ that does not occur in any signature and for every structure M, $M(=_\sigma)$ is the identity relation over $|M|_\sigma \times |M|_\sigma$. As a notational convention, we will always omit the subscript. We use $M\{x \mapsto \nu\}$ to denote a structure where the variable symbol x with sort σ is interpreted as ν, $\nu \in |M|_\sigma$, and all other variables, function and predicate symbols have the same interpretation as in M. Given a formula φ and a structure M, satisfaction $M \models \varphi$ is defined as:

$$\begin{array}{lcl}
M \models p(t_1, \ldots, t_n) & \iff & \langle M[\![t_1]\!], \ldots, M[\![t_n]\!]\rangle \in M(p) \\
M \models \neg\varphi & \iff & M \not\models \varphi \\
M \models \varphi_0 \vee \varphi_1 & \iff & M \models \varphi_0 \text{ or } M \models \varphi_1 \\
M \models \varphi_0 \wedge \varphi_1 & \iff & M \models \varphi_0 \text{ and } M \models \varphi_1 \\
M \models (\exists x\colon \sigma.\, \varphi) & \iff & M\{x \mapsto \nu\} \models \varphi \text{ for some } \nu \in |M|_\sigma \\
M \models (\forall x\colon \sigma.\, \varphi) & \iff & M\{x \mapsto \nu\} \models \varphi \text{ for all } \nu \in |M|_\sigma
\end{array}$$

Note that an implication $\varphi_0 \Rightarrow \varphi_1$ is equivalent to $\neg\varphi_0 \vee \varphi_1$, and a bi-implication $\varphi_0 \Leftrightarrow \varphi_1$ is equivalent to $(\neg\varphi_0 \vee \varphi_1) \wedge (\varphi_0 \vee \neg\varphi_1)$. A formula φ is *satisfiable* if there is a structure M s.t. $M \models \varphi$, and is *valid* if for all structures M, $M \models \varphi$. A structure M satisfies a set of formulas S ($M \models S$) if $M \models \varphi$ for every $\varphi \in S$. A formula is in negation normal form (NNF) if the negation only occurs in literals of the form $\neg p(t_1, \ldots, t_n)$. A formula can be converted to NNF by using the equivalences such as: $\neg\neg\varphi \equiv \varphi$, $\neg(\varphi_0 \wedge \varphi_1) \equiv \neg\varphi_0 \vee \neg\varphi_1$, $\neg(\varphi_0 \vee \varphi_1) \equiv \neg\varphi_0 \wedge \neg\varphi_1$, $\neg(\exists x\colon \sigma.\, \varphi) \equiv (\forall x\colon \sigma.\, \neg\varphi)$, and $\neg(\forall x\colon \sigma.\, \varphi) \equiv (\exists x\colon \sigma.\, \neg\varphi)$. We use $t[s/x]$ to denote a term t' where the free variable x is replaced by the term s. *Skolemization* converts an NNF formula φ into an equisatisfiable formula φ' not containing $\exists$. It it is based on the observation that if φ is NNF, then any subformula $(\exists x\colon \sigma.\, \varphi_0)$ can be replaced by $\varphi_0[f(x_1, \ldots, x_n)/x]$, where $vars(\exists x\colon \sigma.\, \varphi_0) = \{x_1, \ldots, x_n\}$, and f is a new fresh function symbol. The resulting formula can then be converted in linear time into CNF using an approach similar to the one used for propositional formulas. The only difference is that if a subformula contains free variables in the context of universal quantifiers $\forall$, then the auxiliary clauses are universally quantified. For example, let φ be the formula $(\forall x\colon \sigma.\, (\forall y\colon \sigma.\, (q(y) \wedge p(y)) \vee \neg r(x, y)))$, the variable y is bound by an universal quantifier $\forall$, now suppose we want to label the subformula $q(y) \wedge p(y)$, then we create a new fresh predicate symbol s, and encode $\forall y\colon \sigma.\, s(y) \Leftrightarrow (q(y) \wedge p(y))$ using the auxiliary clauses $\{(\forall y\colon \sigma.\, \neg s(y) \vee q(y)),\ (\forall y\colon \sigma.\, \neg s(y) \vee p(y)),\ (\forall y\colon \sigma.\, s(y) \vee \neg q(y) \vee \neg p(y))\}$. In practice, solvers try to minimize the number of auxiliary clauses by using, when feasible, the distributivity rule: $\varphi_0 \vee (\varphi_1 \wedge \varphi_2) \equiv (\varphi_0 \vee \varphi_1) \wedge (\varphi_0 \vee \varphi_2)$. Note that, in the worst case, the repeatedly application of the distributivity rule may exponentially increase the size of the resulting formula. From now on, without loss of generality, we assume every formula that is being checked for satisfiability is in CNF. We also use $(\forall x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n.\, \varphi)$ to denote $(\forall x_1\colon \sigma_1.\, \ldots (\forall x_n\colon \sigma_n.\, \varphi) \ldots)$, and $\forall^* \varphi$ to denote a formula with zero or more $\forall$.

3 Efficient Case-Analysis

Case-analysis is in the core of most automated deduction tools. Most SMT solvers rely on SAT procedures for performing case-analysis efficiently. In this section, we describe the basic techniques used in state-of-the-art SAT solvers. Later, we describe how SMT specific solvers are combined with SAT solvers.

Most successful SAT solvers are based on an approach called *systematic search*. The search space is a tree with each vertex representing a propositional variable and the out edges representing the two choices (i.e., true and false) for this variable. For a formula containing n variable, there are 2^n leaves in this tree. Each path from the root to a leaf corresponds to a truth assignment. Given a formula φ, a procedure, based on systematic search, searches the tree for a truth assignment M that satisfies φ. Most search based SAT solvers are based on the DPLL approach [4]. Given a CNF formula, the DPLL algorithm tries to build a satisfying truth assignment using three main operations: decide, propagate and backtrack. The operation decide heuristically chooses an unassigned propositional variable and assigns it to true or false. This operation is also called *branching* or *case-splitting*. The operation propagate deduces the consequences of a partial truth assignment using deduction rules. The most widely used deduction rule is the *unit-clause rule*, which states that if a clause has all but one literal assigned to false and the remaining literal l is unassigned, then the only way for this clause to evaluate to true is to assign l to true. Let C be the clause $p \vee \neg q \vee \neg r$, and M the partial truth assignment $\{p \mapsto \mathsf{false}, r \mapsto \mathsf{true}\}$, then the only way for C to evaluate to true is by assigning q to false. Given a partial truth assignment M and a clause C in the CNF formula φ such that all literals of C are assigned to false in M, then there is no way to extend M to a complete truth assignment M' that satisfies φ. We say this is a *conflict*, and C is a *conflicting clause*. A conflict indicates that some of the earlier decisions cannot lead to a truth assignment that satisfies φ, and the DPLL procedure must *backtrack* and try a different branch value. If a *conflict* is detected and there are no decisions to backtrack, then the formula φ is unsatisfiable. Many significant improvements of this basic procedure have been proposed over the years. The main improvements are: *lemma learning* [5], *non-chronological backtracking* [5], efficient *indexing techniques* for applying the unit-clause rule [6], and *preprocessing* techniques.

4 What is a Theory?

A theory is essentially a set of sentences. More formally, a Σ-theory is a collection of sentences over a signature Σ. Given a theory T, we say φ is *satisfiable modulo* T if $T \cup \{\varphi\}$ is satisfiable. We use $M \models_T \varphi$ to denote $M \models \{\varphi\} \cup T$. For example, let Σ be the signature containing the symbols 0, 1, $+$, $-$ and $<$, and $\mathbb{Z}$ be the structure that interprets these symbols in the usual way over the integers, then the theory of linear arithmetic is the set of first-order sentences that are true in $\mathbb{Z}$. Let Ω be a class of structures over a signature Σ, then we use $\mathit{Th}(\Omega)$ to denote the set of all sentences ϕ over Σ such that $M \models \phi$ for every M in Ω. In the literature, sometimes a theory T is defined as a class of structures, and φ is satisfiable modulo T if there is a structure M in T such that $M \models \varphi$. Note that these two definitions are not equivalent when checking the satisfiability of a formula φ over an expanded signature (see discussion at [7]).

We say the satisfiability problem for theory T is *decidable* if there is a procedure $\mathfrak{S}$ that checks whether any quantifier-free formula is satisfiable or not. In this case, we say $\mathfrak{S}$ is a *decision procedure* for T.

4.1 Theories

So which theories are integrated with SMT solves? The answer depends on the SMT solver, yet some theories have gained more attention than others. We summarize some of these here.

$$\begin{array}{lcl} y - x & \leq & 1 \\ z - y & \leq & 0 \\ x - z & \leq & -2 \\ x - w & \leq & 0 \\ z - w & \leq & 1 \end{array}$$

$$\begin{tikzcd}
 & \text{y} \arrow[dr, dashed, "0"] & \\
\text{x} \arrow[ur, dashed, "1"] & & \text{z} \arrow[ll, dashed, "-2"] \\
 & \text{w} \arrow[ul, "0"] \arrow[ur, "1"'] &
\end{tikzcd}$$

Figure 1: Difference inequalities example

Linear Arithmetic: Linear arithmetic, also known as additive arithmetic, is the theory where the only arithmetical functions are $+$ and $-$. The functions may be applied to either numerical constants or variables. Multiplication of a numerical constant with a variable is also allowed, so $5 \cdot x$ is a legal term, and for arithmetic over the reals, $\frac{2}{3} \cdot x$ is allowed. The relations for equality and inequalities $(=, \leq, <)$ are used for forming atomic predicates. A conjunction of $=$ and $\leq$ atoms can be decided using a procedure based on the *dual* simplex algorithm [8]. A method for extending the procedure to strict inequalities is by working with non-standard reals that contain *infinitesimals*. This is achieved by adding a symbolic infinitesimal constant ϵ to strict inequalities to make them non-strict.
Difference arithmetic: is a fragment of linear arithmetic where predicates are restricted to be of the form $x - y \leq c$, for x, y variables and c a numeric constant. Conjunctions of difference arithmetic inequalities can be checked very efficiently for satisfiability by searching for negative cycles in weighted directed graphs. In the graph representation, each variable corresponds to a node, and an inequality of the form $x - y \leq c$ corresponds to an edge from y to x with weight c. Figure 1 shows a conjunction of difference inequalities and the corresponding graph, the negative cycle, with weight -1, is shown by dashed lines.
Non-linear arithmetic: The theory of quantifier-free non-linear arithmetic over the reals is decidable. Tarski established a stronger result, that the full first-order theory of reals with addition and multiplication is decidable [9]. Modern methods for non-linear arithmetic over the reals use algorithms from computer algebra, such as computing a Gröbner basis from equalities [10]. The situation is completely different for integers. Hilbert's famous 10th problem was to develop an algorithm for solving non-linear equalities over the integers. Matiyasevich established that this problem was undecidable That is, there is no algorithm for solving such equalities. It is

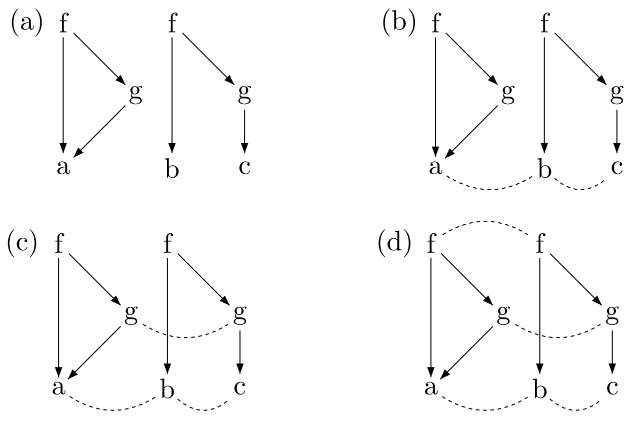

Figure 2: Congruence closure example: $a = b,\ b = c,\ f(a, g(a)) \neq f(b, g(c))$. (a) A DAG for all terms in the example. (b) Equivalences $a = b$ and $b = c$ are shown by dashed lines. (c) Nodes $g(a)$ and $g(c)$ are congruent because $a = c$ is implied by first two equalities. (d) Nodes $f(a, g(a))$ and $f(b, g(c))$ are also congruent, hence the example is unsatisfiable because $f(a, g(a)) \neq f(b, g(c))$.

much worse with quantifiers, which is also known as Peano arithmetic: Gödel established there is not even a computable set of axioms for characterizing Peano arithmetic.

Free functions: The free theory over a signature Σ is the first-order theory with an empty set of sentences. The free theory was used in Section 1.1. It is also known as the theory of uninterpreted functions. Decision procedures for this theory are particularly important because the decision problem for many theories (e.g., arrays) can be reduced to this one. Given a conjunction of equalities between terms using free functions, a *congruence closure* can be used for representing the smallest set of implied equalities. This representation can be used to check if a mixture of equalities and disequalities are satisfiable. Simply check that the terms on both sides of each disequality are in different equivalence classes. Efficient algorithms for computing congruence closure has been the subject of long-running research [11]. In these algorithms, terms are represented as directed acyclic graphs (DAGS). Figure 2 shows the operation of a congruence closure algorithm in a small example.

Bit-vectors: The arithmetic of machines is not the same as arithmetic on mathematical integers. In machine arithmetic, integers fit in fixed size registers. A more suitable domain for machine arithmetic is to represent every number as a fixed-size sequences of bits. On a 64 bit CPU, for instance, an integer is represented as a bit-vector with 64 bits. The theory of bit-vector arithmetic also allows mixing bit-wise operations. For example, when x is a 64-bit integer, then x is a power of two, if and only if $0 = ((x - 1)\&x)$. The theory of bit-vectors can be reduced to Boolean satisfiability by simply *blasting* bit-vector formulas to Boolean formulas. For example, assume x and y are bit-vectors of size 2, then the formula $x + y = 0$ can be blasted into:

$$\begin{array}{lcl} (x_0 \ \mathsf{xor}\ y_0) & \Leftrightarrow & \mathsf{false} \\ (x_1 \ \mathsf{xor}\ y_1) \ \mathsf{xor}\ (x_0 \wedge y_0) & \Leftrightarrow & \mathsf{false} \end{array}$$

where x_0, x_1, y_0, y_1 are propositions corresponding to the *bits* of x and y, xor is the exclusive-or operator, $a \ \mathsf{xor}\ b$ is defined as $a \Leftrightarrow \neg b$. In this example, we are essentially encoding a carry look-ahead adder as a Boolean formula. Current research into efficient decision procedures for bit-vectors seek taking advantage of methods for modular arithmetic, methods for lazy bit-blasting, and approximating long bit-vectors by short bit-vectors.

Arrays: We used the theory of (applicative) arrays in Section 1.1. The theory is useful for encoding state changes to programs with arrays. When a program updates an array a by setting the value of a field i to v it induces a state change. The side-effect can be encoded by referring to the updated array as $\mathit{write}(a, i, v)$. The problem of checking whether a quantifier-free formula is satisfiable modulo the theory of arrays is decidable, and it allows various extensions which have been pursued in recent literature [12, 13].

Other theories: There are several other theories of interest and relevance in applications of SMT solvers. We cannot survey them all here, but mention a just few to give an idea of the scope. These include the theory of *pairs*, or more generally tuples, allows working with pairs and accessing components within pairs after they have been built. The basic theory of acyclic finite *lists* is tailored to the list data-structure found in functional programming languages. A theory of *strings* is closely related to the theory of lists. It is distinguished as *concatenation* is assumed as the basic way of building strings, as opposed to *consing* new elements to the front of a list. Concatenation is found in programs that manipulate strings. Of equal relevance for string-manipulating programs are operations for taking lengths of strings, indexing into strings, and checking membership in regular and context-free languages. Unfortunately not all combinations of these extensions remain decidable. The theory of acyclic finite *recursive data-types* generalizes both the theory of pairs and lists. It can be used for algebraic data-types, known from functional programming.

5 SAT + Theory Solvers

The previous section summarized an array of different theories, and described decision procedures for deciding the satisfiability of conjunction of literals modulo a given theory. From now on, we say these procedures are *theory solvers*. In practice, we are usually interested in deciding the satisfiability of arbitrary quantifier-free formulas. One simple idea is to integrate SAT techniques described in Section 3 with theory solvers [14, 15, 16, 17].

First, we introduce an abstraction function α that maps a quantifier-free formula φ into a propositional formula $\alpha(\varphi)$ by replacing atoms in φ with (fresh) propositional variables. More formally, given a formula φ with atoms $A = \{a_1, \ldots, a_n\}$ and a set of propositional variables $P = \{p_1, \ldots, p_n\}$ not occurring in φ, the mapping α from formulas over A to propositional formulas over P is defined as the homomorphism induced by $\alpha(a_i) = p_i$. The inverse γ of such an abstraction mapping α simply replaces propositional variables p_i with their associated atom a_i. For instance, let φ be the formula $f(x) \not\simeq x \wedge f(f(x)) \simeq x$, $\alpha(f(x) \simeq x) = p_1$ and $\alpha(f(f(x)) \simeq x) = p_2$, then $\alpha(\varphi) = \neg p_1 \wedge p_2$. Moreover, the truth assignment M induces a set of literals

$$\gamma(M) = \{\gamma(p_i) \mid M(p_i) = \mathsf{true}\} \cup \{\neg\gamma(p_i) \mid M(p_i) = \mathsf{false}\}$$

Now, given a truth assignment $M = \{p_1 \mapsto \mathsf{false}, p_2 \mapsto \mathsf{true}\}$, $\gamma(M) = \{f(x) \not\simeq x, f(f(x)) \simeq x\}$.

Given an unsatisfiable set of literals S, we say a *justification* for S is any unsatisfiable subset J of S. Of course, any unsatisfiable set S is a justification for itself. We say a justification J is *non-redundant* if there is no strict subset J' of J that is also unsatisfiable.

The basic integration of a SAT solver with a theory solver is reported in Figure 3. The procedure SAT(φ) (satisfiability solver) returns a tuple $\langle r, M\rangle$ where r is *sat* if φ is satisfiable and *unsat* otherwise, and M is a truth assignment that satisfies φ if r is *sat*. The procedure T-Solver(S) (theory solver) returns a tuple $\langle r, J\rangle$ where r is *sat* if the set of literals S is satisfiable and *unsat* otherwise, and J is a justification for S if r is *unsat*. Note that $\bigvee_{l \in J} \neg\alpha(l)$ is a new clause not in φ', and we say it is a *theory lemma*.

The algorithm described in Figure 3 is also known as the *lazy offline* approach. There are many refinements for this basic algorithm. The basic idea is to have a tighter integration between the two procedures, where the T-solver is used to check partial truth assignments being explored by the SAT solver (*online integration*). In this refinement, additional performance gains can be obtained if the theory solver is incremental and backtrackable.

```
SMT-Solver($\varphi$)
    $\varphi' := \alpha(\varphi)$
    loop
        $\langle r, M\rangle :=$ SAT($\varphi'$)
        if $r =$ unsat then return unsat
        $\langle r, J\rangle :=$ T-Solver($\gamma(M)$)
        if $r =$ sat then return sat
        $C := \bigvee_{l \in J} \neg\alpha(l)$
        $\varphi' := \varphi' \wedge C$
```

Figure 3: Basic SAT + Theory Solver integration

Theory deduction rules can also be used to prune the search space being explored by the DPLL solver (*theory propagation*). More formally, let M be a partial truth assignment, and $\gamma(M)$ implies $\gamma(l_i)$, then l_i is assigned to true by theory propagation. Finally, it is desirable to have a theory solver that produces non-redundant justifications, because they may drastically reduce the search space. This observation follows from the fact that if $J \subset J'$, then the clause $\bigvee_{l \in J} \neg\alpha(l)$ is smaller than $\bigvee_{l \in J'} \neg\alpha(l)$, and consequently the number of truth assignments that satisfy the first clause is smaller than the second.

6 Combining Procedures

Section 4.1 summarized an array of different theories. Most of these theories are decidable and their decision procedures use specialized efficient algorithms. As the example in Section 1.1 illustrated, it does not always suffice to use one theory in isolation. A fundamental question arises: is the union of two solvable theories still solvable? If they are, how can procedures be combined? Can the glue for combining two procedures be defined without specific dependencies on the theories?

Given a Σ_1-theory T_1 and a Σ_2-theory T_2, we use $T_1 \oplus T_2$ to denote the $(\Sigma_1 \cup \Sigma_2)$-theory that is the union of the sentences of T_1 and T_2.

6.1 Strongly Disjoint Theories

We say Σ_1-theory T_1 and Σ_2-theory T_2 are *strongly disjoint* if Σ_1 and Σ_2 do not have sort symbols in common, and consequently no function and predicate symbols in common. For example, the theory of arithmetic and bit-vectors are strongly disjoint. Let $\mathfrak{S}_i$ be a decision procedure for theory T_i, then it is very easy to build a procedure $\mathfrak{S}$ for the $(\Sigma_1 \cup \Sigma_2)$-theory $T_1 \oplus T_2$. It is based on the simple observation that any set S of $\Sigma_1 \cup \Sigma_2$-literals is of the form $S_1 \cup S_2$ where S_i is a set of Σ_i-literals for $i = 1, 2$. Hence, S is satisfiable iff S_1 and S_2 are satisfiable.

6.2 Nelson-Oppen Combination

We say Σ_1-theory T_1 and Σ_2-theory T_2 are *disjoint* if Σ_1 and Σ_2 do not have function and predicate symbols in common. Note that Σ_1 and Σ_2 may have sort symbols in common. The Nelson-Oppen procedure [18] gives a method for combining decision procedures for disjoint theories T_1 and T_2 into one for $T_1 \oplus T_2$.

A theory T is *stably infinite* with respect to sort σ if for every formula φ satisfiable in T, there exists a structure M s.t. $M \models_T \varphi$ and $|M|_\sigma$ is infinite. We say a $(\Sigma_1 \cup \Sigma_2)$-formula φ is *pure* if every literal l in φ is a Σ_i-literal for $i = 1, 2$. Every quantifier-free $(\Sigma_1 \cup \Sigma_2)$-formula φ can be *purified* into a pure and equisatisfiable formula φ_p. The basic idea is to use the following satisfiability preserving transformation:

$$F[t] \rightsquigarrow F[u] \wedge u = t, \quad \text{where } u \text{ is a fresh variable.}$$

For example, let φ be the formula $f(x-1) - 1 = x \wedge f(y) + 1 = y$, then after purification we obtain the equisatisfiable formula φ_p:

$$(u_2 - 1 = x \wedge u_3 + 1 = y \wedge u_1 = x - 1) \wedge (u_2 = f(u_1) \wedge u_3 = f(y))$$

A *partition* Π on a set of variables X is a disjoint collection of subsets $X_1, \ldots, X_n$ s.t. $(\bigcup_{i=1}^n X_i) = X$, and for all $x, y \in X_i$, x and y have the same sort. Given a partition Π, an *arrangement* A_Π is a union of the set of equalities $\{x \simeq y \mid \text{for some } i \text{ s.t. } x, y \in X_i\}$ and disequalities $\{x \not\simeq y \mid \text{for some } i, j \text{ s.t. } i \neq j, x \in X_i, y \in X_j\}$.

Given two disjoint theories T_1 and T_2 such that T_i is stably infinite with respect to each sort σ in Σ_1 and Σ_2, for $i = 1, 2$. The Nelson-Oppen combination theorem states that a pure formula $\varphi_1 \wedge \varphi_2$ is satisfiable in $T_1 \oplus T_2$ iff there exists an arrangement A_Π of the shared variables $X = \mathit{vars}(\varphi_1) \cap \mathit{vars}(\varphi_2)$ such that $\varphi_i \cup A_\Pi$ is satisfiable for $i = 1, 2$. The stable-infiniteness requirement in the Nelson-Oppen framework is used to bring the interpretation of the shared sorts to the same infinite cardinality.

A naïve implementation of the Nelson-Oppen combination method simply tries all possible arrangements. There are many refinements for this basic approach: the SAT solver is used to "guess" the arrangement (*delayed theory combination* [19]), candidate models (structures), produced by $\mathfrak{S}_i$, are used to "guess" the right arrangement (*model-based theory combination* [20]).

A theory T is *convex* iff for all finite sets S of literals and for all non-empty disjunctions $\bigvee_{i \in I} u_i \simeq v_i$ of variables, S implies $\bigvee_{i \in I} u_i \simeq v_i$ in T iff S implies $u_i \simeq v_i$ in T for some $i \in I$. Intuitively, a theory is convex if for every satisfiable set of literals there is a model where variables not implied to be equal have a distinct interpretation. The theory of linear rational arithmetic is convex, but the theory of linear integer arithmetic is not (e.g., if x, y and z are integers, then $\{x \simeq 0, y \simeq 1, 0 \leq z \leq 1\}$ implies $x \simeq z \vee y \simeq z$, but does not imply $x \simeq z$ or $y \simeq z$). For convex theories, instead of *guessing* a partition, one can *deduce* the equalities to be shared. The key idea is to propagate $x \simeq y$ to φ_2 whenever $T_1 \cup \varphi_1$ implies $x \simeq y$, and vice-versa. This process is repeated until no further equations can be propagated. Then, the individual procedures are used to decide whether φ_i is satisfiable. Sharing equalities in this case is sufficient, because $\mathfrak{S}_1$ can assume that in the structure M_2 produced by $\mathfrak{S}_2$ to satisfy φ_2, $M_2(x) \neq M_2(y)$ whenever $x \simeq y$ was not propagated and vice versa. So, for convex theories, there is an efficient way to construct a partition of the set of shared variables.

There are many extensions for the Nelson-Oppen combination method. For example, some of them are extensions for non-stably infinite theories [21, 22] and for non-disjoint theories [23].

7 Meta-Procedures

It is infeasible to implement a (semi-) decision procedure for every possible theory that may be useful in practice. Thus, some SMT solvers implement *meta-procedures* for classes of theories that can be described by a finite number of sentences. A *meta-procedure* $\mathfrak{S}$ is a (semi-) decision procedure for a class of theories Ω. Given a theory $T \in \Omega$ and a formula φ, $\mathfrak{S}$ can decide whether φ is satisfiable modulo T or not.

Instantiation Based Meta-Procedures: The effectively propositional class, EPR, also known as the Bernays-Schönfinkel-Ramsey class of first-order formulas, comprises of formulas of the form $\forall^* \varphi$, where φ is a quantifier-free formula with predicate symbols and constant symbols, but without non-constant function symbols. The satisfiability problem for the EPR class can be reduced to Boolean satisfiability problem by instantiating the quantified formulas by all combinations of constants. Several useful theories, such as the theory of partial orders, are in the EPR class. The satisfiability problem for many other classes of formulas can be decided using instantiation methods [13, 7, 22].

Rewriting Based Meta-Procedures: An *equational theory* is a theory containing only sentences of the form $\forall^* t = s$. Given a finite equational theory T, the Knuth-Bendix completion algorithm [24] is an algorithm for transforming the equations in T into a confluent term rewriting system. When the algorithm succeeds, it has effectively solved the satisfiability problem for T. Similarly, a *Superposition-Calculus* procedure [25] is a semi-decision procedure for the satisfiability problem for a finite set of many-sorted sentences. In many cases, superposition-calculus is a decision procedure [26].

8 Conclusion

In the last few years, satisfiability became the core engine underlying a wide range of powerful technologies. SMT is an active and exciting area of research with many practical applications [1]. We have presented some of the basic ideas, but a real implementation requires careful attention to a large number of details and heuristics that we have not covered. SAT and SMT solving technologies are already making a profound impact on a number of application areas. The theoretical challenges include better representations and algorithms, efficient methods for combining procedures, and various extensions to the basic search method.

References

[1] Bjørner, N., de Moura, L.: Z3^{10}: Applications, Enablers, Challenges and Directions. In: CFV. (2009)

[2] Barrett, C., de Moura, L., Stump, A.: Design and Results of the 1st Satisfiability Modulo Theories Competition. JAR (2005)

[3] McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress. (1962) 21–28

[4] Davis, M., Logemann, G., , Loveland, D.: A machine program for theorem proving. Communications of the ACM (1962)

[5] Marques-Silva, J.P., Sakallah, K.A.: GRASP - A New Search Algorithm for Satisfiability. In: ICCAD. (1996)

[6] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC. (2001)

[7] Ge, Y., de Moura, L.: Complete instantiation for quantified SMT formulas. In: CAV. (2009)

[8] Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: CAV. (2006)

[9] Tarski, A.: A decision method for elementary algebra and geometry. Technical report, 2nd edn. University of California Press, Berkeley (1951)

[10] Buchberger, B.: Ein algorithmus zum auffinden der basiselemente des restklassenringes nach einem nulldimensionalen polynomideal. Technical report, Mathematical Institute, University of Innsbruck, Austria (1965)

[11] Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. J. ACM **27** (1980) 758–771

[12] Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for an extensional theory of arrays. In: LICS. (2001) 29–37

[13] Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: VMCAI. (2006) 427–442

[14] de Moura, L., Rueß, H.: Lemmas on Demand for Satisfiability Solvers. In: SAT. (2002)

[15] Flanagan, C., Joshi, R., Ou, X., Saxe, J.B.: Theorem Proving Using Lazy Proof Explication. In: CAV. (2003)

[16] Barrett, C., Dill, D., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to SAT. In: CAV. (2002)

[17] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53** (2006)

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

[19] Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: Delayed Theory Combination vs. Nelson-Oppen for Satisfiability Modulo Theories: A Comparative Analysis. In: LPAR. (2006)

[20] de Moura, L., Bjørner, N.: Model-based theory combination. In: Proc. 5th SMT Workshop, CAV 2007. (2007)

[21] Jovanović, D., Barrett, C.: Polite Theories Revisited (2009) to appear.

[22] de Moura, L., Bjørner, N.: Generalized, Efficient Array Decision Procedures (2009) to appear.

[23] Tinelli, C., Ringeissen, C.: Unions of Non-Disjoint Theories and Combinations of Satisfiability Procedures. Theoretical Computer Science (2003)

[24] Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. Computational Problems in Abstract Algebra (1970)

[25] de Moura, L., Bjørner, N.: Engineering DPLL(T) + saturation. In: Proc. 4th IJCAR. (2008)

[26] Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM TOCL **10** (2009) 129–179