Satisfiability Modulo Theories: A Beginner's Tutorial

Clark Barrett[1(✉)], Cesare Tinelli[2], Haniel Barbosa[3], Aina Niemetz[1], Mathias Preiner[1], Andrew Reynolds[2], and Yoni Zohar[4]

[1] Stanford University, Stanford, USA
barrett@cs.stanford.edu
[2] The University of Iowa, Iowa City, USA
[3] Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
[4] Bar-Ilan University, Ramat Gan, Israel

Abstract. Great minds have long dreamed of creating machines that can function as general-purpose problem solvers. Satisfiability modulo theories (SMT) has emerged as one pragmatic realization of this dream, providing significant expressive power and automation. This tutorial is a beginner's guide to SMT. It includes an overview of SMT and its formal foundations, a catalog of the main theories used in SMT solvers, and illustrations of how to obtain models and proofs. Throughout the tutorial, examples and exercises are provided as hands-on activities for the reader. They can be run using either Python or the SMT-LIB language, using either the CVC5 or the Z3 SMT solver.

1 Introduction

Great minds have long dreamed of creating machines that can reason deductively, that is, from a set of assumptions, determine whether a particular conclusion logically follows. The question of whether such a machine is possible was posed formally as a grand challenge by the famous mathematician David Hilbert in 1928, who called it the "Entscheidungsproblem" (decision problem) [24]. In 1936, both Church and Turing showed that, in general, this is impossible—the problem is undecidable [13,42]. Undeterred, researchers in automated reasoning have searched for ways to solve either special cases of the problem that are decidable or to find heuristics that work well in practice. Satisfiability modulo theories (SMT) has emerged as an approach that seems to fill a sweet spot in this search space. SMT leverages a rich collection of decidable theories to provide considerable expressive power without sacrificing decidability. SMT also permits some queries over problems that are undecidable or whose decidability is unknown. For these, it employs powerful heuristics that often work well in practice.

This tutorial is an introduction to SMT for new users. We explain what kinds of problems are suitable for SMT solvers, describe the capabilities of modern solvers, and provide guidance on how to encode problems as SMT queries.

A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 571–596, 2025.
https://doi.org/10.1007/978-3-031-71177-0_31

Throughout the tutorial, we provide examples and exercises to illustrate the concepts being explained. Unless otherwise stated, the exercises can be completed using either the CVC5 [3] or the Z3 SMT solver [32], through either their Python interface or their textual interface based on the SMT-LIB 2 format [8]. The CVC5 website at cvc5.github.io contains documentation that can be used as a reference to supplement the material in this tutorial. An online version of the tutorial is also available on that site by clicking on Tutorials. To work through the examples and exercises, we recommend one of the following options.

A) To use a Python API for SMT, first create a virtual environment.

```
python3 -m venv smt-tutorial
source smt-tutorial/bin/activate
```

Next, install CVC5's Python API or Z3's Python API, or both.

```
python3 -m pip install cvc5
python3 -m pip install z3-solver
```

CVC5 is distributed under the BSD 3-clause license. Some features, however, such as its finite field solver (see Sect. 4.9), are only available in an extended version of cvc5 distributed under the GNU General Public License (GPL).[1] Since GPL is a problem for some users, the GPL version is not built or distributed by default. To install the GPL version of CVC5, use:

```
python3 -m pip install cvc5-gpl
```

Once a solver API is installed, you can copy example Python code into a script file, e.g., `Example.py`, and then type:

```
python3 Example.py
```

Note that, for the examples below, if you are using Z3 instead of CVC5, you must replace the first line of each Python code snippet with:

```
from z3 import *
```

B) Executables for CVC5 and Z3 are available for download. For CVC5, go to the CVC5 website, click on Downloads, and follow the link to the release page on GitHub. Alternatively, for Z3, go to the Z3 releases page at github.com/Z3Prover/z3/releases. From either release page, download the latest release compatible with your machine (for CVC5, choose a GPL download if you want support for finite fields). Once you unzip the downloaded archive, the executable will be in the `bin` directory. Thus, if the unzipped directory is called `release-dir`, and you have downloaded CVC5, you can run an SMT-LIB example called `Example.smt2` by typing:

```
release-dir/bin/cvc5 Example.smt2
```

from your shell's command line. If you downloaded Z3, type instead:

[1] The finite field solver uses the CoCoA library [1], which has a GPL license.

```
release-dir/bin/z3 Example.smt2
```

C) From the CVC5 website, click on Try cvc5 online. This links to a page that provides a web interface for running CVC5 on scripts in the SMT-LIB format.

This tutorial has been tested with CVC5 1.2.0, Z3 4.13.0, and Python 3.12.3, but later releases should work as well. Solver outputs shown below are based on CVC5 version 1.2.0. Other versions or solvers should produce conceptually similar results, but the outputs may not be exactly the same. The SMT-LIB examples are based on version 2.6 of the format [5]. CVC5's Python API was designed to be a drop-in replacement for Z3's Python API. The credit for the design of the Python API thus goes to the Z3 authors.

2 Overview

At an intuitive level, SMT solvers are general-purpose problem solving tools. They are somewhat similar to calculators, in that the user provides the problem of interest, and the tool does some calculation to produce an answer. However, they are much more powerful than a simple calculator.

SMT solvers reason *symbolically*, as is done in grade school algebra. The user provides a set of *assertions* that describe constraints to be satisfied, and the solver produces a *solution* satisfying *all* of the constraints, if there is one. Consider the following simple example, mimicking a typical algebra word problem.

Example 1. In 10 years, Alice will be twice as old as Bob is now, but in 22 years, Bob will be twice as old as Alice is now. How old are Alice and Bob?

First, let's see how to solve this using Python.

```
from cvc5.pythonic import *
a, b = Ints('a b')
solve(a + 10 == 2 * b, b + 22 == 2 * a)
```

The Pythonic API is designed to be as simple and intuitive as possible. We introduce the symbols we are using (SMT solvers always require that symbols be introduced before they are used), and then we call `solve`, passing in the two equations in much the same way we would write them naturally. The output is a simple representation of the solution as a Python list.

```
[a = 18, b = 14]
```

Alternatively, SMT solvers can take as input a script written in the SMT-LIB language [5], a standard developed by the SMT community whose syntax is similar to that of LISP. Below is the same example written in SMT-LIB.

```
(set-logic QF_LIA)
(set-option :produce-models true)

(declare-const a Int)
(declare-const b Int)
```

```
(assert (= (+ a 10) (* 2 b)))
(assert (= (+ b 22) (* 2 a)))

(check-sat)
(get-model)
```

The result is:

```
sat
(
(define-fun a () Int 18)
(define-fun b () Int 14)
)
```

Notice that the solver replies `sat` before giving the solution. This is short for "satisfiable," a word meaning that there is at least one solution. SMT solvers can also identify when a set of assertions has no solution. In this case, the solver replies `unsat`, which is short for "unsatisfiable."

Let's take a closer look at the SMT-LIB input file, which is a sequence of *commands*. The command in the first line tells the solver which *logic* we are working in. In this case, we are using `QF_LIA` which stands for quantifier-free linear integer arithmetic. We explain more about logics in Sect. 4 below. The second line tells the solver to produce *models*. A model assigns a concrete meaning to every user-declared symbol. Without turning this option on, a solver will still respond with `sat` or `unsat`, but it may not be able to provide a model. The next two lines declare two *uninterpreted constants* called `a` and `b`. Informally, we often refer to these as variables, because they play the same role that variables do in math. However, in the automated reasoning literature, a *variable* typically refers to a symbol that is bound by a quantifier, whereas an uninterpreted constant is a symbol whose value is determined by a model. SMT-LIB follows the the latter terminology. The next two lines create *assertions*. An assertion is a way of telling the solver about a formula that we would like to be true in the model that is produced. Note that the formulas too are specified in a LISP-like prefix syntax. Finally, the command `(check-sat)` tells the solver to check whether the set of assertions made so far is satisfiable, and the command `(get-model)` (which is only legal if the solver returns `sat`) prints values for each uninterpreted constant, with the guarantee that assigning these values to the constants makes all the assertions true. The values are printed using legal SMT-LIB syntax in case the user wants to copy and paste them into a new SMT-LIB script.

Exercise 1. Consider a modification of Example 1. The first assertion will stay the same, but for the second, let's assert that Bob will be twice as old as Alice in only 20 years. Modify the Python program or SMT-LIB script to reflect the new set of constraints. What output does the SMT solver give?

So far, we have seen the most basic use of an SMT solver. Given a set of assertions, determine whether there is a solution for them. We now show that this basic capability can be used to answer several similar questions.

Suppose we have a set X of assumptions about the world, and we want to know whether some hypothetical Y is possible under those assumptions. If we

can express X and Y as SMT formulas, then an SMT solver can answer the question. In fact, we simply assert each assumption in X as well as the formula representing Y and check whether this set of assertions is satisfiable.

Example 2. Let x and y be 32-bit integers, with x a multiple of 2. Is it possible for the machine arithmetic product of x and y to be 1?

For this problem, we'll use *bit-vectors*. SMT solvers use bit-vectors to model machine arithmetic and other operations on fixed-size vectors of bits. The SMT-LIB encoding is as follows.

```
(set-logic QF_BV)

(declare-const x (_ BitVec 32))
(declare-const y (_ BitVec 32))
(declare-const z (_ BitVec 32))

(assert (= x (bvmul z (_ bv2 32))))
(assert (= (bvmul x y) (_ bv1 32)))

(check-sat)
```

This time, we use the logic `QF_BV` which stands for quantifier-free bit-vectors. The underscore symbol `_` is used in SMT-LIB to indicate that the next symbol is indexed by the following argument. It is used to specify the bit-vector size in this example. The `bvmul` symbol represents bit-vector multiplication, and the notation `bvX` is the bit-vector constant whose value, in decimal notation, is `X`. Constant `z` names the value we must multiply by 2 to get x. Here's how to solve it using the Pythonic API. This time, we'll use the API in a way that more closely resembles the SMT-LIB script.

```
from cvc5.pythonic import *

x, y, z = BitVecs('x y z', 32)
s = SolverFor('QF_BV')

s.add(x == z * 2)
s.add(x * y == 1)

result = s.check()
print("result: ", result)
```

There is no solution because an even number does not have a multiplicative inverse in machine arithmetic (i.e., when doing arithmetic modulo a power of 2).

Exercise 2. Find the multiplicative inverse of 5 (mod 2^8).

Another common situation is when we have a set X of assumptions, and we want to know whether some Y *must* hold as a consequence. If so, we say that Y is *implied* or *entailed* by X. Again, assuming we can represent X and Y using formulas, we can start by asserting the formulas representing X. At this point, however, we do not assert the formula for Y. Instead, we assert its *negation*. If the result is `unsat`, then Y must follow from X. The reasoning is that if it is not possible for the negation of Y to be true when X is true, then Y itself must be true. Let's look at a version of the well-known syllogism about Socrates.

Example 3. If all humans are mortal, and Socrates is a human, then must Socrates be mortal?

The Python code is as follows.

```
from cvc5.pythonic import *
S = DeclareSort("S")
Bool = BoolSort()
Human = Function("Human", S, Bool)
Mortal = Function("Mortal", S, Bool)
Socrates = Const("Socrates", S)

s = SolverFor('UF')

x = Const("x", S)
s.add(ForAll([x], Implies(Human(x), Mortal(x))))
s.add(Human(Socrates))
s.add(Not(Mortal(Socrates)))

print(s.check())
```

The SMT-LIB version of the same problem looks like this.

```
(set-logic UF)

(declare-sort S 0)
(declare-fun Human (S) Bool)
(declare-fun Mortal (S) Bool)
(declare-const Socrates S)

(assert (forall ((x S)) (=> (Human x) (Mortal x))))
(assert (Human Socrates))
(assert (not (Mortal Socrates)))

(check-sat)
```

This problem illustrates a few new encoding tools. First, we use the logic `UF` which stands for "uninterpreted functions." This logic allows us to declare new function symbols. Note that it is also missing the `QF` prefix we've used above, which means that quantifiers are also allowed. We declare a new uninterpreted *sort* `S`. A sort is like a type in programming languages. We use an *uninterpreted* sort to represent a class of individual objects that cannot be modeled with the predefined sorts provided by SMT-LIB, (so far, we've seen the predefined sorts for integers and bit-vectors). Next, we declare two functions, `Human` and `Mortal`, each of which takes a single argument of sort `S` and returns a `Bool`, the SMT-LIB Boolean sort. A function returning a Boolean is also called a *predicate*. We then declare an uninterpreted constant called `Socrates` of sort `S`. Now, we are ready to encode the first fact, namely that all humans are mortal. To do so, we use the *universal quantifier*, `ForAll`. The assertion states that for every individual `x` of sort `S`, if the predicate `Human` holds for that individual, then the predicate `Mortal` also holds. The next assertion states that the `Human` predicate holds for `Socrates`. Finally, we want to see whether the fact that Socrates is mortal necessarily follows from the assumptions. To do this, we assert the negation of the statement and check for satisfiability. Running the example confirms that the result is unsatisfiable and thus, indeed, this statement is entailed.

What we have presented so far should provide a good high-level idea of what is possible with SMT solvers.[2] We cover these ideas in more detail in the following. In Sect. 3, we briefly describe the formal foundations for SMT. Next, in Sect. 4, we catalog the different *theories* supported by SMT solvers and provide examples of how to use them. We cover the different outputs produced by SMT solvers, including models and proofs, in Sect. 5, and conclude in Sect. 6 with pointers to additional resources.

3 Formal Foundations

The satisfiability modulo theories problem can be formalized in many-sorted first-order logic with equality. We briefly outline the necessary concepts here. Due to space constraints, we assume some familiarity with basic concepts and notation from mathematical logic. More details can be found in [21,25].

3.1 Syntax

In first-order logic, one constructs formulas that are statements about individuals in some domain of discourse and their relationships. Many-sorted logic adds the possibility of talking about multiple, separate domains.

Signatures. The language of formulas is determined by a vocabulary of symbols, called a *signature*, which has three main components: *sort symbols* (such as Int, Real, Person, etc.) which name, or *denote*, domains of interest; *function symbols* (such as, $+$, $*$, log, mother, father) which denote total functions over the domains; and *relation symbols* (such as, $=$, $<$, even, married) which denote total relations over the domains. A signature also specifies the *arity* of each function symbol f, which is the number of inputs f takes, as well as its *rank*, which consists of the sort of f's inputs and of f's output.[3] We say that f has arity n and rank $\sigma_1 \cdots \sigma_n \sigma$ in a signature Σ if f takes n inputs of respective sorts $\sigma_1, \ldots, \sigma_n$ and returns an output of sort σ. A function symbol of arity 0 and rank σ (such as 0, 1, true, ...) is also called a *constant symbol* of sort σ. It is convenient to consider only signatures that have a distinguished sort Bool, for the Booleans, and treat relation symbols as function symbols whose return type is Bool. In addition, we assume that every signature contains a distinguished function symbol $\approx_\sigma$ of rank $\sigma\sigma$Bool, denoting the identity relation, for each sort σ of Σ.

A signature Σ is a *subsignature* of a signature Ω, and Ω is a *supersignature* of Σ, if all the sort and function symbols of Σ are also in Ω and the function symbols have the same rank in Ω as they do in Σ.

[2] More sophisticated features and use cases are beyond the scope of this tutorial, but we plan to provide additional tutorials on more advanced topics in the future.

[3] For simplicity, we do not consider the more general case where function symbols can be overloaded by being assigned more than one arity and/or rank.

Variables, Terms and Formulas. To build formulas, in addition to fixing a signature Σ, we also fix a set $\mathbf{X}$ of *sorted* variables, each associated with a sort σ and standing for some element from (the set denoted by) σ. We can then build terms out of variables and function symbols from Σ. Given a signature Σ, a *well-sorted Σ-term*, or just *term* for short, is defined inductively as follows: (i) a variable or constant symbol of sort σ is a term of sort σ; (ii) if f is a function symbol of rank $\sigma_1 \cdots \sigma_n\sigma$, with $n > 0$, and $t_1, \ldots, t_n$ are terms of sort $\sigma_1 \cdots \sigma_n$, respectively, then the expression $f(t_1, \ldots, t_n)$ is a term of sort σ; (iii) if φ is a term of sort Bool and x is a variable of sort σ, then the expressions $\exists\, x{:}\sigma.\, \varphi$ and $\forall\, x{:}\sigma.\, \varphi$ are terms of sort Bool. We then *identify formulas with terms of sort* Bool. The distinguished symbols $\forall$ and $\exists$ are *quantifier symbols*. We say that a variable x *occurs free* in a formula φ if x occurs in φ and either φ contains no quantifier symbols or it has the form $\exists\, y{:}\sigma.\, \varphi'$ or $\forall\, y{:}\sigma.\, \varphi'$, for some variable y, where x occurs free in φ'.

3.2 Semantics

For each signature Σ, the meaning of Σ-terms is provided by mathematical structures called interpretations. A *Σ-interpretation* $\mathcal{I}$ maps:

- each sort σ of Σ to a *non-empty* set $\sigma^{\mathcal{I}}$, the *domain* of σ in $\mathcal{I}$, with $\mathsf{Bool}^{\mathcal{I}}$ being the binary set $\{\mathsf{true}, \mathsf{false}\}$;
- each variable $x \in \mathbf{X}$ of sort σ to an element $x^{\mathcal{I}} \in \sigma^{\mathcal{I}}$;
- each function symbol f of rank $\sigma_1 \cdots \sigma_n\sigma$ to a *total* function $f^{\mathcal{I}}$ of type $\sigma_1^{\mathcal{I}} \times \cdots \times \sigma_n^{\mathcal{I}} \to \sigma^{\mathcal{I}}$ (and, in particular, each constant symbol c of sort σ to an element $c^{\mathcal{I}} \in \sigma^{\mathcal{I}}$).

We say that σ (resp. x, f) *denotes* the set $\sigma^{\mathcal{I}}$ (element $x^{\mathcal{I}}$, function $f^{\mathcal{I}}$) in $\mathcal{I}$. Every Σ-interpretation $\mathcal{I}$ extends from variables and function symbols to Σ-terms t as follows: (i) a term $f(t_1, \ldots, t_n)$ *evaluates* in $\mathcal{I}$ to $f^{\mathcal{I}}(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}})$, the value returned by function $f^{\mathcal{I}}$ when applied to the elements denoted by $t_1, \ldots, t_n$; (ii) an *existentially quantified* formula $\exists\, x{:}\sigma.\, \varphi$ evaluates to true in $\mathcal{I}$ if and only if φ evaluates to true in an interpretation $\mathcal{I}[x \mapsto a]$ that maps x to *some* suitable $a \in \sigma^{\mathcal{I}}$ and is otherwise identical to $\mathcal{I}$; (iii) a *universally quantified* formula $\forall\, x{:}\sigma.\, \varphi$ evaluates to true in $\mathcal{I}$ if and only if φ evaluates to true in $\mathcal{I}[x \mapsto a]$ for *all* possible choices of values for x in $\sigma^{\mathcal{I}}$.

An interpretation $\mathcal{I}$ *satisfies* a formula φ if $\varphi^{\mathcal{I}} = \mathsf{true}$ and *falsifies* it if $\varphi^{\mathcal{I}} = \mathsf{false}$. In the former case, we also say that $\mathcal{I}$ is a *model* of φ.

The *reduct* of an Ω-interpretation $\mathcal{I}$ to a subsignature Σ of Ω is the (unique) Σ-interpretation that interprets the symbols of Σ exactly as $\mathcal{I}$. Intuitively, the reduct is obtained by *forgetting* the symbols of Ω that are not in Σ.

In the definition of interpretation above, we have not provided a meaning for the usual Boolean connectives such as $\neg, \wedge, \vee, \Rightarrow$ and so on. In SMT, specific interpretations of function symbols are provided by a theory, as explained next.

3.3 Theories

In general, we are not interested in arbitrary interpretations of terms and formulas in a signature Σ but in interpretations belonging to a specific *theory* T that *constrain* the meaning of the symbols in Σ; for instance, that interpret $\neg$ and $\wedge$ as logical negation and conjunction, $0, 1, 2, \ldots$ as the natural numbers, and so on. Traditionally in logic, a theory is defined by a set of formulas, called *axioms*: one considers only Σ-interpretations that satisfy all the axioms. In SMT, a theory is, more generally, a class of interpretations that can be specified axiomatically or in other ways. More precisely, a *Σ-theory* T is a pair $(\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations, however specified. We describe and discuss several examples of theories commonly used in SMT in the next section.

Given a theory $T = (\Sigma, \mathbf{I})$, we consider not just Σ-formulas but Ω-formulas for some supersignature Ω of Σ. In the context of T, we refer to the symbols of Σ as *theory* symbols and to the additional symbols in Ω as *uninterpreted* symbols. For instance, in the theory of reals, we may write a formula of the form $a + 1 > b$ where a and b are uninterpreted, or *symbolic*, constants of sort Real. Intuitively, while the meaning of $+$ and 1 is fixed by the theory, the meaning of a and b is not. Hence, we consider the formula satisfiable if there are real values for a and b which make the formula evaluate to true. This idea is formalized in the notion of *satisfiability in* T.

Satisfiability Modulo a Theory. If T is a Σ-theory, a *T-interpretation* is *any* Ω-interpretation $\mathcal{I}$ for some supersignature Ω of Σ whose restriction to Σ differs from an interpretation of T at most in the way it interprets the variables.

An Ω-formula φ is *satisfiable in* T if it is satisfied by *some* T-interpretation $\mathcal{I}$—which may interpret the variables of φ and the sort, function, and predicate symbols not in Σ arbitrarily. The formula is *valid in* T if it is satisfied by *all* T-interpretations. A set Φ of Ω-formulas *entails* φ *in* T, written $\Phi \models_T \varphi$, if every T-interpretation that satisfies all formulas in Φ satisfies φ as well. The set Φ is *satisfiable in* T if there is a T-interpretation that satisfies all of its formulas.

4 SMT Theories

A key feature of SMT is that the entire problem is parameterized by the choice of a theory T. This is important because it means that SMT is an algorithmic *framework*, rather than a fixed algorithm. Thus, if a particular problem cannot easily be encoded in any existing theory supported by SMT solvers, one option is to add support for a new theory which is better suited to the problem. In fact, this is exactly the process by which many of the theories supported by modern SMT solvers were added.

Theories can be used alone or in arbitrary combinations. Besides the theory, other parameters related to the syntax of formulas include whether or not to enable quantifiers and whether to disallow or limit the use of certain theory operations. In the SMT-LIB standard, and in solvers that support it, these

parameters are configured by specifying a *logic*. A logic identifies the theory (or theories) being used and optionally imposes syntactic restrictions on the allowed formulas. Users can provide the SMT solver with a predefined *logic name* (like `QF_LIA`, `QF_BV`, and `UF` seen earlier) to specify which logic is to be used. By default (i.e., if no logic name is provided), SMT solvers typically enable all the theories they support and allow all operations. This is equivalent to using the special logic name `ALL`. However, solvers are often tuned with specific heuristics for specific logics. Thus, it is advisable to provide the solver with the most specific logic name possible. In this section, we discuss the most common theories and logics supported by SMT solvers, with examples of each.

4.1 Core Theory and Uninterpreted Symbols

The SMT-LIB standard defines a *core theory* which consists of a core signature with a fixed interpretation that is always present, regardless of which other theories are being used. The core theory defines the Boolean sort `Bool` (`BoolSort()` in Python), the Boolean theory constants `true` and `false` (`BoolVal(True)` and `BoolVal(False)` in Python), and the operators `=`, `not`, `and`, `or`, `xor`, and `=>` (`==`, `Not`, `And`, `Or`, `Xor`, and `Implies` in Python), all with the usual meanings. The equality symbol `=` is *polymorphic*: it can be applied to two terms of the same sort, for any predefined or user-declared sort. There are also two more polymorphic operators that require a bit more explanation. The `distinct` (`Distinct`) operator takes two or more arguments of the same sort and returns `true` exactly when all the arguments have pairwise distinct values. The `ite` (`If`) operator takes three arguments, the first of which must be of Boolean sort. The other two arguments can have any sort as long as it is the same for both. The meaning of the `ite` operator is the second argument when the first argument is true, and the third argument otherwise.

The simplest logic that builds on the core theory is `QF_UF`, short for "quantifier-free uninterpreted functions." This logic disallows quantifiers and does not define any new symbols beyond those in the core theory. However, it allows the user to extend the signature with new sorts and symbols. The SMT solver is allowed to interpret these symbols in any way it chooses. This is why they are referred to as uninterpreted: the solver does not impose any restrictions on the interpretation (besides the declared arity and rank). The following example illustrates the use of uninterpreted symbols as well as the `And` and `Distinct` operators.

Example 4. Let f be a unary function from U to U, for some set U. Check that, whatever the meaning of f, if $f(f(f(x))) = x$ and $f(f(f(f(f(x))))) = x$, then $f(x) = x$.

We show a solution in Python followed by one using SMT-LIB.

```
from cvc5.pythonic import *
U = DeclareSort("U")
f = Function("f", U, U)
x = Const("x", U)
```

```
s = SolverFor('QF_UF')
s.add(And((f(f(f(x))) == x), (f(f(f(f(f(x))))) == x)))
s.add(Distinct(f(x), x))  # negation of f(x) = x
print(s.check())
```

```
(set-logic QF_UF)
(declare-sort U 0)

(declare-fun f (U) U)
(declare-const x U)

(assert (and (= (f (f (f x))) x) (= (f (f (f (f (f x))))) x)))
(assert (distinct (f x) x))

(check-sat)
```

We can derive $f(x) = x$ from the first assertion by performing a series of substitutions, and thus the problem is unsatisfiable. Now, we present a simple example that illustrates the `ite` operator. It also shows that in Python, we can use `!=` instead of `Distinct` to assert that two terms are distinct.

Example 5. Suppose we know that x is either equal to y or z, depending on the value of the Boolean b. Suppose we further know that w is equal to one of y or z. Does it follow that $x = w$?

The Python solution is shown below.[4]

```
from cvc5.pythonic import *
U = DeclareSort("U")
b = Const("b", BoolSort())
x, y, z, w = Consts("x y z w", U)

s = SolverFor('QF_UF')
s.add(x == (If(b, y, z)))
s.add(Or((w == y), (w == z)))
s.add(x != w)

if s.check() == sat:
  m = s.model()
  print("\n".join([str(k) + " : " + str(m[k]) for k in m]))
```

CVC5 outputs the following for this example.

```
b : True
w : (as @U_0 U)
x : (as @U_1 U)
y : (as @U_1 U)
z : (as @U_0 U)
```

The result tells us that it does *not* follow that $x = w$. The model gives us a *counterexample* to that claim. Because the sort `U` is uninterpreted, the model returned by CVC5 must choose an interpretation for it. Here, CVC5 tells us that it is interpreting `U` as a set with two elements, named `@U_0` and `@U_1`. The model

[4] Due to space constraints, the SMT-LIB versions of the remaining examples do not appear in the text. They are available in the online version of the tutorial available from the Tutorials link on the CVC5 website.

then specifies that `x` and `y` have one value and `z` and `w` have the other, so `x` is not equal to `w`.

Exercise 3. Modify Example 4 to make it satisfiable and Example 5 to make it unsatisfiable.

4.2 Arithmetic

Though there are many tools available for arithmetic reasoning, SMT solvers are unique in their ability to reason efficiently about arbitrary Boolean combinations of arithmetic constraints, as well as to combine arithmetic reasoning with reasoning about other theories. It is important to note that SMT solvers reason *precisely* about both integer and real arithmetic. That is, they use arbitrary-precision arithmetic as opposed to machine integer or floating-point approximations. This means that SMT solvers are not susceptible to the numerical errors that can arise, for instance, when using floating-point arithmetic to approximate real arithmetic. It also means that for problems whose complexity lies mainly in the arithmetic reasoning, as opposed to Boolean reasoning, SMT solvers are typically slower than tools that use floating-point approximations. The underlying algorithms for arithmetic reasoning in SMT solvers are based on standard techniques that have been adapted to the SMT context, such as the Simplex algorithm [20] and Cylindrical Algebraic Decomposition [2].

There are a large number of logics to choose from within the arithmetic umbrella, with reasoning over reals generally more efficient than reasoning over integers, and reasoning over less expressive formulas generally more efficient than reasoning over more expressive ones. We briefly discuss the various logics here.

Difference Logic. In *difference logic*, every arithmetic constraint must be of the form $x - y \bowtie c$ or $x \bowtie c$, where $\bowtie \in \{=, <, >, \leq, \geq\}$, and c is a numeric theory constant. If x and y range over integers, we call it *integer difference logic*, and if they range over reals, we call it *real difference logic*. Efficient algorithms exist for both [17,36]. The names of these logics are `QF_IDL` and `QF_RDL`, respectively. One application for difference logic is *job shop scheduling* [41].

Example 6. Suppose we have 3 jobs to complete on 2 machines. Job 1 requires machine 1 for 10 min and then machine 2 for 5 min. Job 2 requires machine 2 for 20 min and then machine 1 for 5 min. And Job 3 requires machine 1 for 5 min and then machine 2 for 5 min. Can all jobs be completed in 30 min?

To solve the problem, we create integer variables for the start times of each task within each job. We assert that the start times are non-negative, each task within each job doesn't start until the previous task finishes, and tasks on each machine don't overlap. Finally, we check that each task finishes on time.

```
from cvc5.pythonic import *

j11,j12,j21,j22,j31,j32 = Ints("j11 j12 j21 j22 j31 j32")
```

```
s = SolverFor('QF_IDL')
s.add(And([x >= 0 for x in [j11, j12, j21, j22, j31, j32]]))
s.add(And(j12 - j11 >= 10, j22 - j21 >= 20, j32 - j31 >= 5))
s.add(And(Or(j22 - j11 >= 10, j11 - j22 >= 5),
          Or(j31 - j11 >= 10, j11 - j31 >= 5),
          Or(j31 - j22 >= 5, j22 - j31 >= 5)))
s.add(And(Or(j21 - j12 >= 5, j12 - j21 >= 5),
          Or(j32 - j12 >= 5, j12 - j32 >= 5),
          Or(j32 - j21 >= 5, j21 - j32 >= 5)))
s.add(And(j12 <= 25, j22 <= 25, j32 <= 25))

print(s.model() if s.check() == sat else "unsat")
```

Exercise 4. What is the minimum amount of time that it will take to complete all of the jobs in Example 6?

Linear Arithmetic. The logic of linear arithmetic allows arithmetic constraints to have any form that is equivalent to $\sum c_i x_i + b \bowtie 0$, where b, c_i are numeric theory constants and $\bowtie \in \{=, <, >, \leq, \geq\}$. As before, there are both integer and real variants, `QF_LIA` and `QF_LRA`, respectively. One can also mix the two with `QF_LIRA`. Note that, according to the SMT-LIB standard, when using `QF_LIRA`, integers and reals should not be mixed in the same linear sum, but most solvers (including CVC5 and Z3) are more permissive and do allow mixed terms. Example 1 is a good example of a simple `QF_LIA` problem.

Exercise 5. Repeat Exercise 1, but change the logic to `QF_LRA`, change the types of the variables from `Int` to `Real`, and append `.0` to each numeric constant. Now, what output does the solver give?

Nonlinear Arithmetic. Moving up the expressiveness hierarchy, we next have logics for quantifier-free *nonlinear arithmetic*. In these logics, arbitrary polynomials are allowed in constraints. The logic `QF_NRA` is for nonlinear arithmetic over the reals, which is decidable but with doubly exponential complexity [2]. On the other hand, the same logic over integers, `QF_NIA`, is undecidable. CVC5 implements a decision procedure for `QF_NRA` based on a combination of heuristic pruning and cylindrical algebraic coverings [29]. CVC5 and other tools implement incomplete heuristic procedures for `QF_NIA`.

Example 7. Find a solution for $x^2y + yz + 2xyz + 4xy + 8xz + 16 = 0$.

```
from cvc5.pythonic import *
x, y, z = Reals("x y z")
s = SolverFor('QF_NRA')
s.add(x*x*y + y*z + 2*x*y*z + 4*x*y + 8*x*z + 16 == 0)
print(s.model() if s.check() == sat else "unsat")
```

4.3 Arrays

Consider the following Python function which swaps two elements in a dictionary.

```
def swap(a,i,j):
  tmp = a[i]
  a[i] = a[j]
  a[j] = tmp
```

If `a[i]` and `a[j]` happen to be equal, the dictionary `a` is unchanged by the function. To prove this fact, we could try modeling dictionaries as uninterpreted functions. However, asserting that two functions are equal is not allowed in first-order logic. Alternatively, we could use a quantifier to assert that two functions return the same output when given the same input, for any input. However, we would like to avoid quantifiers when possible, as their use puts us in an undecidable logic.

Fortunately, the SMT-LIB standard includes a theory of arrays [30], which can help in this situation. The theory is perhaps more accurately viewed as a theory of mutable maps and is parameterized by two sorts, one for the index (corresponding to the key type of the dictionary) and one for the elements (values in the dictionary). For example, the SMT-LIB sort `(Array Int Real)` represents arrays indexed by integers and containing reals. Note that SMT arrays are always total, in the sense that they have an element for every value in the index sort. In particular, an array indexed by `Int` is conceptually infinite.

The theory has two operators: `select`, which takes an array and an index and returns the element at that index, and `store`, which takes an array a, an index i, and an element e, and returns a new array that is the result of updating a with the element e at index i.

Typically, the theory of arrays is used in combination with other theories that make sense for the index and element sorts. For example, the logic `QF_ALIA` allows quantifier-free formulas with variables that range over integers and arrays of integers. The simplest logic with arrays is `QF_AX`, in which all the sorts must be uninterpreted.

In the example below, we encode the above problem using the array theory.

Example 8. For the Python program above, show that, for arbitrary index and element sorts, if `a[i]` and `a[j]` are equal, then so are `a` and `swap(a,i,j)`.

```
from cvc5.pythonic import *
I = DeclareSort("I")
E = DeclareSort("E")
i, j = Consts("i j", I)
tmp = Const("tmp", E)
array = ArraySort(I, E)
a_in, a_out = Consts("a_in, a_out", array)

s = SolverFor('QF_AX')

s.add(tmp == (Select(a_in, i)))
s.add(a_out == (Store(Store(a_in, i, Select(a_in, j)),
    j, tmp)))
s.add((Select(a_in, i)) == (Select(a_in, j)))
s.add(a_in != a_out)

print(s.check())
```

Exercise 6. Another property of `swap` that we can prove is that if `a[i]` and `a[j]` are distinct, then `swap` would change `a`. Modify the solution for Example 8 to prove this property.

4.4 Bit-Vectors

Consider a simple implementation (written in a C-like syntax) for computing the absolute value of a 32-bit integer: $abs(x) \coloneqq x < 0 \ ? \ -x : x$. Instead of branching on $x < 0$, it is possible to compute the absolute value of x with three or four branch-free operations [28] as follows. Let xrs be an abbreviation for the arithmetic right shift ($\gg_s$) of x by 31 bits. Note that the result of this operation is either 0 or -1 (all bits set to 1), depending on the most significant bit (MSB) of x: if the MSB of x is 0, xrs is 0; otherwise, xrs is -1. Three branchless alternatives for computing the absolute value of x are as follows.

1. $abs_1(x) \coloneqq (x \oplus xrs) - xrs$
2. $abs_2(x) \coloneqq (x + xrs) \oplus xrs$
3. $abs_3(x) \coloneqq x - ((x \ll 1) \ \& \ xrs)$

These branchless versions of $abs(x)$ make use of the 32-bit versions of the bit-wise operations exclusive or ($\oplus$), bit-wise and (&), logical shift left ($\ll$), and arithmetic shift right ($\gg_s$).

We can use an SMT solver to prove whether the branchless versions are equivalent to the original implementation. Note that integers, as discussed in Sect. 4.2, are not a good fit, as it is difficult to model the bitwise operators using the arithmetic operators. However, the SMT-LIB standard includes a theory of fixed-size bit-vectors, which defines the bit-precise semantics of fixed-size machine integers. The name for the quantifier-free logic containing just this theory is `QF_BV`. Using this logic, we can easily check the equivalence of the absolute value computations.

Example 9. Show that the first branchless alternative abs_1 is equivalent to abs.

```
from cvc5.pythonic import *
x = Const("x", BitVecSort(32))
xrs = x >> 31
s = SolverFor('QF_BV')
s.add(If(x < 0, -x, x) != (x ^ xrs) - xrs)      # prove abs() == abs1()
print(s.model() if s.check() == sat else "unsat")
```

Exercise 7. Show that the second and third branchless alternatives abs_2 and abs_3 are equivalent to abs.

4.5 Datatypes

Built into the SMT-LIB language is a mechanism for defining *(algebraic) datatypes*. Datatypes are highly useful in applications for reasoning about data structures like records, lists, and trees [7]. The quantifier-free logic name is `QF_DT`.

Example 10. Model a binary tree containing integer data. Find trees x and y such that (i) the left subtree of x is the same as the right subtree of y and (ii) the data stored in x is greater than 100.

Note that we need both datatypes and integer arithmetic for this example. CVC5 supports the logic name `QF_DTLIA`, but Z3 does not. Fortunately, we can always use `ALL` for the logic if a more specific logic is not available.

```
from cvc5.pythonic import *
decl = Datatype("tree")
decl.declare("node", ("data", IntSort()), ("left", decl), ("right", decl))
decl.declare("nil")
Tree = decl.create()

x, y = Consts("x y", Tree)

s = SolverFor('ALL')
s.add(Tree.is_node(x))
s.add(Tree.is_node(y))
s.add(Tree.left(x) == Tree.right(y))
s.add(Tree.data(x) > 100)

print(s.model() if s.check() == sat else "unsat")
```

The output gives the values for x and y.

```
[ x = node(101, nil, node(0, nil, nil)),
  y = node(0, node(0, nil, node(0, nil, nil)), nil)]
```

Exercise 8. Show that a tree cannot be equal to its own left subtree.

4.6 Floating-Point Arithmetic

The most common representation of real numbers in hardware and software is the binary floating-point number representation system as defined by the IEEE Standard 754-2019 for Floating-Point Arithmetic [27]. Floating-point numbers are encoded as a triple of bit-vectors: the fractional part (the significand), the exponent (a power of 10 by which the significand is multiplied), and a sign bit. This representation is of limited range and precision, and thus, the domain of floating-point numbers is finite. It also includes special values for representing errors as not-a-number and for plus and minus infinity. In SMT-LIB, the IEEE-754 standard is formalized as the theory of floating-point arithmetic [11]. The quantifier-free logic name is `QF_FP`.

Example 11. The SMT-LIB standard supports a *fused multiplication and addition* operator `fp.fma`. Given three single precision floating-point numbers a, b, and c, show that the floating-point fused multiplication and addition of a, b, and c is different from first multiplying a and b and then adding c.

```
from cvc5.pythonic import *

a, b, c = FPs("a b c", Float32())
rm = Const("rm", RNE().sort())
s = SolverFor('QF_FP')
```

```
s.add(Distinct(fpFMA(rm, a, b, c), fpAdd(rm, fpMul(rm, a, b),c)))
result = s.check()
m = s.model()
print(m)
print(f'fpFMA(rm, a, b, c) = {m.eval(fpFMA(rm, a, b, c))}')
print(f'fpAdd(rm, fpMul(rm, a, b),c) = {m.eval(fpAdd(rm, fpMul(rm, a, b),c))}')
```

The output gives the solution.

```
[a = -1.3333333730697632*(2**-1), b = -1.9999998807907104*(2**-1),
 c = -1.9999998807907104*(2**-1), rm = RTP()]
fpFMA(rm, a, b, c) = -1.333333134651184*(2**-2)
fpAdd(rm, fpMul(rm, a, b),c) = -1.3333330154418945*(2**-2)
```

Exercise 9. Modify the solution to Example 11 to show that floating-point addition is not associative, i.e., $a + (b + c) \neq (a + b) + c$.

4.7 Strings

It is often necessary to reason about string data when reasoning about programs. Reasoning about bit-vector representations of strings has the disadvantage that it requires fixing the string length up front. Also, the theory of bit-vectors does not include many of the utility functions for strings that exist in string libraries in programming languages. The SMT-LIB theory of strings provides support for variable-length strings and a large set of string operations. The quantifier-free logic name is `QF_S`. Typically, though, we use `QF_SLIA` since we need arithmetic to reason about string lengths.

Example 12. Given two strings, `x1` and `x2`, each consisting of no more than two characters, is it possible to build the string `"abbaabb"` using only 3 string concatenations (where each concatenation may use any previous result including `x1` and `x2`)?

We can solve this problem by building a *circuit* of string concatenations and using nondeterministic choice to pick the inputs for each concatenation.

```
from cvc5.pythonic import *
p, x, i = {}, {}, {}
for k in range(1, 13): p[k] = Bool("p" + str(k))
for k in range(1, 6): x[k] = String("x" + str(k))
for k in range(1,7): i[k] = String("i" + str(k))

result = StringVal("abbaabb")

s = SolverFor('QF_SLIA')
s.add(And(Length(x[1]) <= 2, Length(x[2]) <= 2))

s.add(i[1] == If(p[1], x[1], x[2]))
s.add(i[2] == If(p[2], x[1], x[2]))
s.add(x[3] == Concat(i[1], i[2]))

s.add(i[3] == If(p[3], x[1], If(p[4], x[2], x[3])))
s.add(i[4] == If(p[5], x[1], If(p[6], x[2], x[3])))
s.add(x[4] == Concat(i[3], i[4]))

s.add(i[5] == If(p[7], x[1], If(p[8], x[2], If(p[9], x[3], x[4]))))
s.add(i[6] == If(p[10], x[1], If(p[11], x[2], If(p[12], x[3], x[4]))))
```

```
s.add(x[5] == Concat(i[5], i[6]))

s.add(x[5] == result)

print(s.model() if s.check() == sat else "unsat")
```

Exercise 10. Use SMT to determine how many concatenations are needed to get `"abbaabb"` if `x1` and `x2` are both restricted to have a length of 1.

4.8 Quantifiers

We saw an example of quantified formulas in Example 3. Quantifiers can be enabled in SMT solvers by dropping `QF` from the logic name. However, enabling quantifiers typically increases the complexity of the decision problem significantly. In fact, solving `UF` problems is equivalent to solving the decision problem for first-order logic, Hilbert's original Entscheidungsproblem, which is undecidable. And although `LIA`, `LRA`, and `NRA` are decidable, the decision procedures are expensive. For these reasons, SMT solvers mostly handle quantifiers by attempting to find quantifier *instantiations* that, together with the other quantifier-free assertions, are unsatisfiable. For problems that are expected to be unsatisfiable, this approach can be quite effective. Moreover, by using different instantiation techniques and effort levels, a wide variety of problems can be solved.

CVC5 supports several techniques for handling quantified formulas, which can vary based on the logic. By default, CVC5 limits its effort so that it usually returns quickly with an answer of either `unsat` or `unknown`. For logics that include uninterpreted functions, it uses a combination of E-matching [31] and conflict-based instantiation [40]. In case the user wants to invest more effort, these techniques can be supplemented with techniques such as enumerative instantiation [38] (option `enum-inst`). For logics that admit quantifier elimination (e.g., quantified linear arithmetic or bit-vectors), it uses counterexample-guided quantifier instantiation [34,39], which is a complete procedure for these logics.

By default, CVC5 will generally not attempt to determine that an input with quantified formulas is satisfiable. However, more advanced techniques can be used to answer `sat` in the presence of quantified formulas, including finite model finding [37] (option `finite-model-find`), model-based quantifier instantiation [23] (option `mbqi`), and syntax-guided quantifier instantiation [35] (option `sygus-inst`).

In general, to set options that are not on by default, we can use the `setOption` solver method in Python, as shown below.

```
from cvc5.pythonic import *
s = SolverFor('UF')
s.setOption('enum-inst', True)
s.setOption('finite-model-find', True)
s.setOption('mbqi', True)
s.setOption('sygus-inst', True)
```

4.9 Non-standard Theories

CVC5 and Z3 support several theories that are not (yet) part of the SMT-LIB standard. We discuss a few of them briefly here, focusing on those supported by CVC5. More documentation about non-standard theories, including reference tables describing the supported operators can be found on the CVC5 website.

Sequences. The theory of *sequences* brings together features of the theories of arrays and strings. Similar to arrays, sequences are parameterized by the sort of their elements. So we can declare a sequence of integers, a sequence of bit-vectors, and so on. Like strings, sequences have a variable but finite length and can be concatenated together. The sequence theory is enabled whenever the string theory is enabled (e.g., by using the logic name `QF_S` or `QF_SLIA`). Note that Z3 also supports a theory of sequences that is mostly (but not entirely) compatible with the CVC5 version.

Example 13. Let `x` be a sequence of integers. Find a value for `x` such that the first and last elements sum to 9, and if we concatenate `x` with itself, then `(3,4,5)` appears as a subsequence.

```
from cvc5.pythonic import *
x, y, z = Consts("x y z", SeqSort(IntSort()))

s = SolverFor('QF_SLIA')

s.add(Length(x) > 0)
s.add(x[0] + x[Length(x) - 1] == 9)
s.add(y == Concat(x,x))
s.add(z == Concat(Unit(IntVal(3)), Unit(IntVal(4)), Unit(IntVal(5))))
s.add(Contains(y, z))

print(s.model() if s.check() == sat else "unsat")
```

Exercise 11. Show that it's not possible to have sequences `x`, `y`, and `z` such that `x` is a proper prefix of `y`, `y` is a proper prefix of `z`, and `z` is a proper prefix of `x`.

Finite Fields. CVC5 can reason about constraints over finite fields of order p, where p is any prime. It relies on the fact that a field of order p is isomorphic to the integers modulo p. The quantifier-free logic name for finite fields is `QF_FF`. At the time of writing, this theory is not supported by other SMT solvers.

Example 14. In a finite field of order 13, find two elements such that their sum and product are both equal to the multiplicative identity in the field.

Running this example requires a GPL build of CVC5, as explained in Sect. 1.

```
from cvc5.pythonic import *
F = FiniteFieldSort(13)
x, y = FiniteFieldElems("x y", F)
s = SolverFor("QF_FF")
s.add(x + y == 1)
s.add(x * y == 1)
print(s.model() if s.check() == sat else "unsat")
```

Exercise 12. In a finite field of order 13, find an element such that if you square it twice you get the multiplicative identity.

Finite Sets. CVC5 has support for the theory of finite sets. This theory supports basic set operations like membership, union, and intersection, as well as constraints on a set's cardinality. The quantifier-free logic name is `QF_FS`. At the time of writing, this theory is not supported by other SMT solvers.

Example 15. Verify that union distributes over intersection.

```
from cvc5.pythonic import *
S = DeclareSort("S")
A, B, C = [Set(i, S) for i in ["A","B","C"]]
s = SolverFor('QF_FS')
s.add(Not((A | (B & C)) == ((A | B) & (A | C))))
print(s.check())
```

Exercise 13. Does set difference distribute over intersection? If not, find a counterexample.

4.10 Combinations of Theories

So far, we have mostly seen examples of how to pose queries that involve a single theory. Part of the appeal of SMT solvers is their ability to mix reasoning about different theories. This can be done in a natural way. Any well-sorted formula is allowed, and all sort constructors can take any other sort as an argument.

One slight complication is the question of how to specify the logic name. It is always safe to use `ALL` as the logic name, though as mentioned above, it may be more efficient to give a more precise logic name. When mixing theories, CVC5 allows any logic name that follows the following rules. First, the logic name must start with the prefix `QF_` if the intent is to limit reasoning to quantifier-free formulas. The rest of the logic name can include any of the following components, in any order: (*i*) `A` for arrays; (*ii*) `UF` for uninterpreted functions; (*iii*) `BV` for bit-vectors; (iv) `FP` for floating-point numbers; (*v*) `DT` for datatypes; (vi) `S` for strings and sequences; (*vii*) either `IDL`, `RDL`, `LIA`, `LRA`, `LIRA`, `NIA`, `NRA`, or `NIRA` for arithmetic; (*viii*) `FF` for finite fields; and (*ix*) `FS` for finite sets. Thus, for example, `QF_AUFDTBVLRA` allows formulas that are quantifier-free and mix arrays, uninterpreted functions, datatypes, bit-vectors, and linear real arithmetic. Examples 10, 12, and 13 illustrate combinations of theories.

5 SMT Solver Outputs

As we have seen, the main result of an SMT query is either `sat` or `unsat`. In some cases, the solver may also output `unknown`. This can happen, for example, if the problem includes quantifiers. In this section, we discuss how to obtain more information from the solver in each case.

Satisfiable Queries. When a solver returns `sat`, we have already seen that one possible way to get more information is to call `get-model`, which returns values for all of the uninterpreted constants in the formula. A more fine-grained approach is to call `get-value` which takes a term as an argument and returns the value of that specific term.

Unsatisfiable Queries. When a solver returns `unsat`, it makes a quite strong statement: there is *no interpretation* of the user-declared symbols that satisfies the formula. SMT solvers can provide more information as to why a formula is unsatisfiable via an *unsat(isfiable) core*, a subset of the assertions that is already unsatisfiable. In SMT-LIB scripts, it can be obtained with the command `get-unsat-core`. The unsat core is not guaranteed to be minimal, but solvers generally make an effort to reduce its size as much as possible without having to solve additional SMT queries.

Some solvers can also produce *proofs* for the unsatisfiability of a formula, i.e., a structured argument showing how an inconsistency can be derived from an unsat core of the formula. A proof can serve as a certificate of the result and be used to independently validate the solver's response [4] . A proof (if supported) can be obtained in an SMT-LIB script with the command `get-proof`. The result is dependent on the proof system and format the solver uses to represent its reasoning. CVC5 has full support for proofs and unsat cores.

Consider again the Socrates example (Example 3). Below, we show how to retrieve an unsat core and a proof of its unsatisfiability.

```
from cvc5.pythonic import *

s = SolverFor('UF')

s.set("produce-proofs", "true")
s.set("proof-granularity", "theory-rewrite")
s.set("produce-unsat-cores", "true")

S = DeclareSort("S")
Human = Function("Human", S, BoolSort())
Mortal = Function("Mortal", S, BoolSort())
Socrates = Const("Socrates", S)

x = Const("x", S)

s.add(ForAll([x], Implies(Human(x), Mortal(x))))
s.add(Human(Socrates))
s.add(Not(Mortal(Socrates)))

print(s.check())
print("The core is: ", s.unsat_core())

p = s.proof()

print("The proof is:\n", p)
```

The first part of the output is the unsat core.

```
The core is:
   - (forall ((x S)) (=> (Human x) (Mortal x)))
   - (Human Socrates)
   - (not (Mortal Socrates))
```

The core contains all three assertions. In this case, the core is minimal, as all three are needed to derive `unsat`. The reasoning is shown in the proof. The result of the proof() method is a proof object which connects the input assertions to the conclusion (`unsat`) via a sequence of steps justified by proof rules. The proof rules used by CVC5 are documented on the CVC5 website.

Figure 1 shows a visualization of the proof as a tree. For readability, we use simple names to abbreviate long terms. Each node in the tree shows: (*i*) the formula proved (the conclusion); (*ii*) the name of the proof rule used; (*iii*) a numeric id; and (*iv*) the total number of descendants. Immediate children of each node represent premises required for the node's proof rule. The root of the tree is `let9`, which stands for `(not (and let4 let3 let2))`, where `let4`, `let3`, and `let2` represent the three assertions. This node has a single child containing the conclusion `false`, based on a proof tree whose leaves are the three assertions. The derivation of `false` depends on instantiating the quantified assertion (`let4`) with `x` as `Socrates`. This is done in node 5, only after `(forall ((x S)) (=> (Human x) (Mortal x)))` (i.e., `let4`) is rewritten (node 8) into `(forall ((x S)) (or (not (Human x)) (Mortal x)))` (i.e., `let8`). The instantiation `(or (not (Human Socrates)) (Mortal Socrates))` is named `let6`. Node 9 concludes `(not let6)` from the other assertions. Finally, node 2 concludes `false` from the mutually inconsistent clauses derived by the solver (where `let7` is `(not let6)`, `let2` is `(not let1)`, and `let5` is `(not let3)`).

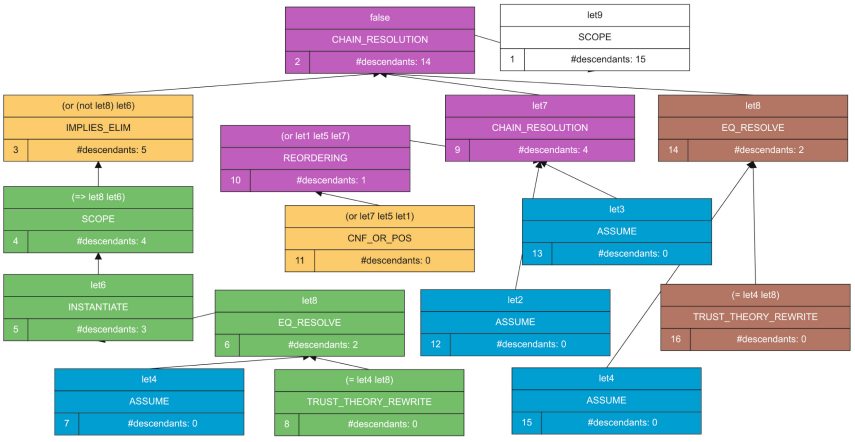

Fig. 1. A proof tree generated by CVC5

Unknown Queries. A solver returns `unknown` when it is unable to solve the input problem. There could be several different reasons for this. One is that the solver's procedure may be incomplete for the class of problems the input

belongs to, which means that it is not always able to determine if the problem is satisfiable or not. Another possible reason is that some resource limit was exceeded, causing the solver to stop before it could find an answer. In SMT-LIB, the command `(get-info :reason-unknown)` can be used to request more information about why a solver returned `unknown`.

6 Conclusion

This tutorial is a basic introduction to using SMT solvers. There are numerous resources available for those who wish to learn more.

The SMT-LIB website smt-lib.org has details about the SMT-LIB standard [5], as well as links to software and an extensive collection of benchmarks. More information on the foundations of SMT and how solvers work under the hood can be found in several overview papers and book chapters [6,9,18]. There are also tool papers describing the most prominent SMT solvers, including: Alt-Ergo [15], Bitwuzla [33], CVC5 [29], MathSAT [14], OpenSMT2 [26], SMTInterpol [12], SMT-RAT [16], STP [22], veriT [10], Yices2 [19], and Z3 [32]. More information about CVC5 is available on its website.

Data Availability Statement. An artifact with all the examples and tools from the paper is available at: https://doi.org/10.5281/zenodo.12763927.

References

1. Abbott, J., Bigatti, A.M., Palezzato, E.: New in CoCoA-5.2.4 and CoCoALib-0.99600 for SC-square. In: Satisfiability Checking and Symbolic Computation. CEUR Workshop Proceedings, vol. 2189, pp. 88–94. CEUR-WS.org (2018). http://ceur-ws.org/Vol-2189/paper4.pdf
2. Ábrahám, E., Davenport, J.H., England, M., Kremer, G.: Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. J. Log. Algebraic Methods Program. **119**, 100633 (2021). https://doi.org/10.1016/J.JLAMP.2020.100633
3. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Barbosa, H., et al.: Generating and exploiting automated reasoning proof certificates. Commun. ACM **66**(10), 86–95 (2023). https://doi.org/10.1145/3587692
5. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
6. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, chap. 33, pp. 825–885. IOS Press (2021)
7. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for a theory of inductive data types. J. Satisfiabil. Boolean Model. Comput. **3**, 21–46 (2007)

8. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theorie, Edinburgh, UK (2010)
9. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
10. Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., Fontaine, P.: veriT: an open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 151–156. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_12
11. Brain, M., Tinelli, C., Rümmer, P., Wahl, T.: An automatable formal semantics for IEEE-754 floating-point arithmetic. In: ARITH, pp. 160–167. IEEE (2015)
12. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: an interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31759-0_19
13. Church, A.: An unsolvable problem of elementary number theory. Am. J. Math. **58**(2), 345–363 (1936)
14. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
15. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories, Oxford, United Kingdom (2018). https://inria.hal.science/hal-01960203
16. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 360–368. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_26
17. Cotton, S., Maler, O.: Fast and flexible difference constraint propagation for DPLL(T). In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 170–183. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_19
18. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM **54**(9), 69–77 (2011). https://doi.org/10.1145/1995376.1995394
19. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
20. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_11
21. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press, Cambridge (1972)
22. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_52
23. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
24. Hilbert, D., Ackermann, W.: Grundzüge der theoretischen Logik, Berlin 1928. Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete **27** (1938)

25. Hodges, W.: A Shorter Model Theory. Cambridge University Press, Cambridge (1997)
26. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: an SMT solver for multi-core and cloud computing. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 547–553. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_35
27. IEEE: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008), pp. 1–84 (2019). https://doi.org/10.1109/IEEESTD.2019.8766229
28. Jr., H.S.W.: Hacker's Delight, 2nd edn. Pearson Education, Boston (2013). http://www.hackersdelight.org/
29. Kremer, G., Reynolds, A., Barrett, C.W., Tinelli, C.: Cooperating techniques for solving nonlinear real arithmetic in the cvc5 SMT solver (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13385, pp. 95–105. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-10769-6_7
30. McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress, North-Holland, pp. 21–28 (1962)
31. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_13
32. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
33. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR arxiv:2006.01621 (2020)
34. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Solving quantified bit-vectors using invertibility conditions. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 236–255. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_16
35. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Syntax-guided quantifier instantiation. In: TACAS 2021. LNCS, vol. 12652, pp. 145–163. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_8
36. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with exhaustive theory propagation and its application to difference logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_33
37. Reynolds, A., Tinelli, C., Goel, A., Krstić, S., Deters, M., Barrett, C.: Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 377–391. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_26
38. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 112–131. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_7
39. Reynolds, A., King, T., Kuncak, V.: Solving quantified linear arithmetic by counterexample-guided instantiation. Formal Methods Syst. Des. **51**(3), 500–532 (2017). https://doi.org/10.1007/s10703-017-0290-y
40. Reynolds, A., Tinelli, C., de Moura, L.M.: Finding conflicting instances of quantified formulas in SMT. In: Formal Methods in Computer-Aided Design, FMCAD

2014, Lausanne, Switzerland, 21–24 October 2014, pp. 195–202. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987613
41. Roselli, S.F., Bengtsson, K., Åkesson, K.: SMT solvers for job-shop scheduling problems: models comparison and performance evaluation. In: 14th IEEE International Conference on Automation Science and Engineering, CASE 2018, Munich, Germany, 20–24 August 2018, pp. 547–552. IEEE (2018). https://doi.org/10.1109/COASE.2018.8560344
42. Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. J. Math. **58**(345–363), 5 (1936)