Lecture Notes: Satisfiability Modulo Theories

17-355/17-665/17-819: Program Analysis (Spring 2020)
Claire Le Goues*
clegoues@cs.cmu.edu

1 Motivation and Overview

We have now seen several techniques that generate and solve logical formulas. For example, in Hoare-style verification, we used weakest preconditions and verification conditions to generate formulas of the form $P \Rightarrow Q$. Usually P and Q have free variables x, e.g. P could be $x > 3$ and Q could be $x > 1$. We want to prove, ideally automatically, that $P \Rightarrow Q$ no matter what x we choose, i.e. no matter what the model (an assignment from variables to values) is. This is equivalent to saying $P \Rightarrow Q$ is *valid*. Similarly, symbolic execution generates sets of guards g in terms of program and symbolic expressions. Such guards also typically have free variables, such as symbolic values representing all possible program inputs. We would like to determine if such *path conditions* are feasible, when constructing trees of all possible program executions; we moreover would like to identify values for those free variables, because that can help generate test inputs that cover particular program paths. SMT solving addresses this type of problem. Although the general goal won't be feasible for all formulas, it is feasible for a useful subset of formulas.

Solving this problem begins by reducing general formula validity to another problem, that of *satisfiability*. A formula F with free variable x is valid iff for all x, F is true. That's the same thing as saying there is no x for which F is false. But that's furthermore the same as saying there is no x for which $\neg F$ is true. This last formulation is asking whether $\neg F$ is *satisfiable*. It turns out to be easier to seach for a single satisfying model (or prove there is none), then to show that a formula is valid for all models.

Strictly speaking, satisfiability is for boolean formulas, or those that include boolean variables as well as boolean operators such as $\wedge, \vee$, and $\neg$. They may include quantifiers such as $\forall$ and $\exists$, as well. But if we want to have variables over the integers or reals, and operations over numbers (e.g. $+, >$, the types of relations we've included even in our very simple WHILE language), we need a solver for a *theory*, such as the theory of Presburger arithmetic (which could prove that $2 * x = x + x$), or the theory of arrays (which could prove that assigning $x[y]$ to 3 and then looking up $x[y]$ yields 3). SMT solvers include a basic satisfiability checker, and allow that checker to communicate with specialized solvers for those theories. This is the meaning of the "Modulo Theories" in "Satisfiability Modulo Theories."

2 DPLL for Boolean Satisfiability

In building to satisfiability modulo theories, we begin by discussing the problem of satisfiability.

*These notes were developed together with Jonathan Aldrich

2.1 Boolean satisfiability (SAT)

Satisfiability decides whether a conjunction of literals in a theory is satisfiable. The "easiest" theory is propositional logic. The problem of establishing satisfiability for boolean formulas in propositional logic is referred to as SAT, and a decision procedure for it as a "SAT solver."

We begin by transforming a formula F into *conjuctive normal form (CNF)*—i.e. a conjunction of disjunctions of positive or negative literals. For example $(a \vee \neg b) \wedge (\neg a \vee c) \wedge (b \vee c)$ is a CNF formula. If the formula is not already in CNF, we can put it into CNF by using De Morgan's laws, the double negative law, and the distributive laws:

$$\begin{array}{rcl} \neg(P \vee Q) & \iff & \neg P \wedge \neg Q \\ \neg(P \wedge Q) & \iff & \neg P \vee \neg Q \\ \neg\neg P & \iff & P \\ (P \wedge (Q \vee R)) & \iff & ((P \wedge Q) \vee (P \wedge R)) \\ (P \vee (Q \wedge R)) & \iff & ((P \vee Q) \wedge (P \vee R)) \end{array}$$

The goal of the decision procedure is, given a formula, to say it is satisfiable; this can be established by giving an example *satisfying assignment*. A satisfying assignment maps variables to boolean values. So, $X \vee Y$ is satisfiable, and one satisfying assignment for it is $X \mapsto \texttt{true}, Y \mapsto \texttt{false}$ (there are other satisfying assignments as well). $X \wedge \neg X$, by contrast, is not satisfiable.

The Cook-Levin theorem established that boolean satisfiability is NP-complete. In the worst case, one can decide SAT for a given formula by simply trying all possible assignments. For example, given:

$$\exists E.E \vDash (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y) \wedge (z)$$

We can brute-force by conducting a backtracking search that tries all possible assignments of `true` and `false` to x, y, z. There are 2^n possible combinations in the worst case, where n is the number of variables.

2.2 The DPLL Algorithm

The DPLL algorithm, named for its developers Davis, Putnam, Logemann, and Loveland, is an efficient approach to deciding SAT. The DPLL algorithm improves on the backtracking search with two innovations: *unit propagation*, and *pure literal elimination*.

Let's illustrate by example. Consider the following formula:

$$(b \vee c) \wedge (\mathbf{a}) \wedge (\neg a \vee c \vee d) \wedge (\neg c \vee d) \wedge (\neg c \vee \neg d \vee \neg a) \wedge (b \vee d)$$

There is one clause with just a in it. This clause, like all other clauses, has to be true for the whole formula to be true, so we must make a true for the formula to be satisfiable. We can do this whenever we have a clause with just one literal in it, i.e. a unit clause. (Of course, if a clause has just $\neg b$, that tells us b must be false in any satisfying assignment). In this example, we use the *unit propagation* rule to replace all occurrences of a with `true`. After simplifying, this gives us:

$$(\mathbf{b} \vee c) \wedge (c \vee d) \wedge (\neg c \vee d) \wedge (\neg c \vee \neg d) \wedge (\mathbf{b} \vee d)$$

Now here we can see that b always occurs positively (i.e. without a $\neg$ in front of it). If we set b to be `true`, that eliminates all occurrences of b from our formula, thereby making it simpler—but it doesn't change the satisfiability of the underlying formula. An analogous approach applies when a variable always occurs negatively. A literal that occurs only positively, or only negatively,

in a formula is called *pure*. Therefore, this simplification is called the *pure literal elimination* rule, and applying it to the example above gives us:

$$(c \vee d) \wedge (\neg c \vee d) \wedge (\neg c \vee \neg d)$$

Now for this formula, neither of the above rules applies. We just have to pick a literal and guess its value. Let's pick c and set it to `true`. Simplifying, we get:

$$(d) \wedge (\neg d)$$

After applying the unit propagation rule (setting d to true) we get:

$$(\texttt{true}) \wedge (\texttt{false})$$

This didn't work out! But remember, we guessed about the value of c. Let's backtrack to the formula where we made that choice:

$$(c \vee d) \wedge (\neg c \vee d) \wedge (\neg c \vee \neg d)$$

and now we'll try things the other way, i.e. with $c = \texttt{false}$. Then we get the formula

$$(d)$$

because the last two clauses simplified to true once we know c is false. Now unit propagation sets $d = \texttt{true}$ and then we have shown the formula is satisfiable. A real DPLL algorithm would keep track of all the choices in the satisfying assignment, and would report back that a is true, b is true, c is false, and d is true in the satisfying assignment.

This procedure—applying unit propagation and pure literal elimination eagerly, then guessing a literal and backtracking if the guess goes wrong—is the essence of DPLL. Here's an algorithmic statement of DPLL, adapted slightly from a version on Wikipedia:

function DPLL(ϕ)
  if $\phi = \texttt{true}$ **then**
    return `true`
  end if
  if ϕ contains a `false` clause **then**
    return `false`
  end if
  for all unit clauses l in ϕ **do**
    $\phi \leftarrow$ UNIT-PROPAGATE(l, ϕ)
  end for
  for all literals l occurring pure in ϕ **do**
    $\phi \leftarrow$ PURE-LITERAL-ASSIGN(l, ϕ)
  end for
  $l \leftarrow$ CHOOSE-LITERAL(ϕ)
  return DPLL($\phi \wedge l$) $\vee$ DPLL($\phi \wedge \neg l$)
end function

Mostly the algorithm above is straightforward, but there are a couple of notes. First, why does the algorithm do unit propagation before pure literal assignment? It's good to do unit propagation first because it can create additional opportunities to apply further unit propagation as well as pure literal assignment. On the other hand, pure literal assignment will never create unit literals

that didn't exist before: pure assignment can eliminate entire clauses, but never makes an existing clause shorter.

Secondly, the last line implements backtracking. We assume a short-cutting $\vee$ operation at the level of the algorithm. So if the first recursive call to DPLL returns true, so does the current call–but if it returns fall, we invoke DPLL with the chosen literal negated, which effectively backtracks.

Exercise 1. Apply DPLL to the following formula, describing each step (unit propagation, pure literal elimination, choosing a literal, or backtracking) and showing now it affects the formula until you prove that the formula is satisfiable or not:

$$(a \vee b) \wedge (a \vee c) \wedge (\neg a \vee c) \wedge (a \vee \neg c) \wedge (\neg a \vee \neg c) \wedge (\neg d)$$

There is a lot more to learn about DPLL, including hueristics for how to choose the literal l to be guessed and smarter approaches to backtracking (e.g. non-chronological backtracking), but in this class, let's move on to consider SMT.

3 Solving SMT Problems

The approach above targets formulas in the theory of propositional logic. However, there are many other possibly useful theories, and useful formulas may mix them. For example, consider a conjunction of the following formulas:[1]

$$\begin{array}{ll} f(f(x) - f(y)) = a & \wedge \\ f(0) = a + 2 & \wedge \\ x = y & \end{array}$$

This problem mixes linear arithmetic with the theory of uninterpreted functions (here, f is some unknown function). We may have a satisfiability procedure for each theory involved in the formula, but how can we deal with their combination? Note that we *can't* in general just separate out the terms from each theory in a formula to see if they are separately satisfiable, because multiple satisfying assignments might not be compatible. Instead, we handle each domain separately (as a theory), and then combine them all together using DPLL and SAT as the "glue".

3.1 Definitions

A *satisfiability modulo theories (SMT)* solver operates on propositions involving both logical terms and terms from theories (defined below). Effectively, such solvers replace all the theory clauses in a mixed-theory formula with special propositional variables, and then use a pure SAT solver to solve the result. If the solution involves any of the theory clauses, the solver asks the theory if they can all be true. If not, new constraints are added to the formula, and the process repeats.

In general, a *theory* is a set of sentences (syntax) with a deductive system that can determine satisfiability (semantics). Usually, the set of sentences is formally defined by a grammar of terms over atoms. The satisfying assignment (or model, or interpretation) maps literals (terms or negated terms) to booleans. Useful theories include linear and non-linear arithmetic, bitvectors, arrays, quantifiers, or strings, as well as uninterpreted functions (like f in our example above).

An important feature of the kinds of theories we are discussing is that they all understand *equality*. We do not delve deeply into how/why in these notes, but the fact is important to how SMT can solve formulas that mix theories, as we will see below.

[1] This example is due to Oliveras and Rodriguez-Carbonell

3.2 Basic SMT idea, illustrated

We will work through our example above to demonstrate the ideas behind SMT. The first step is to separate the multiple theories. We can do this by replacing expressions with fresh variables, in a procedure named Nelson-Oppen after its two inventors. For example, in the first formula, we'd like to factor out the subtraction, so we generate a fresh variable and divide the formula into two:

$$\begin{array}{ll} f(e1) = a & \textit{// in the theory of uninterpreted functions now} \\ e1 = f(x) - f(y) & \textit{// still a mixed formula} \end{array}$$

Now we want to separate out $f(x)$ and $f(y)$ as variables $e2$ and $e3$, so we get:

$$\begin{array}{ll} e1 = e2 - e3 & \textit{// in the theory of arithmetic now} \\ e2 = f(x) & \textit{// in the theory of uninterpreted functions} \\ e3 = f(y) & \textit{// in the theory of uninterpreted functions} \end{array}$$

We can do the same for $f(0) = a + 2$, yielding:

$$\begin{array}{l} f(e4) = e5 \\ e4 = 0 \\ e5 = a + 2 \end{array}$$

We now have formulas in two theories. First, formulas in the theory of uninterpreted functions:

$$\begin{array}{l} f(e1) = a \\ e2 = f(x) \\ e3 = f(y) \\ f(e4) = e5 \\ x = y \end{array}$$

And second, formulas in the theory of arithmetic:

$$\begin{array}{l} e1 = e2 - e3 \\ e4 = 0 \\ e5 = a + 2 \\ x = y \end{array}$$

Notice that $x = y$ is in both sets of formulas (remember, all theories understand equality). First, however, let's run a solver. The solver for uninterpreted functions has a congruence closure rule that states, for all f, x, and y, if $x = y$ then $f(x) = f(y)$. Applying this rule (since $x = y$ is something we know), we discover that $f(x) = f(y)$. Since $f(x) = e2$ and $f(y) = e3$, by transitivity we know that $e2 = e3$.

But $e2$ and $e3$ are symbols that the arithmetic solver knows about, so we add $e2 = e3$ to the set of formulas we know about arithmetic. Now the arithmetic solver can discover that $e2 - e3 = 0$, and thus $e1 = e4$. We communicate this discovered equality to the uninterpreted functions theory, and then we learn that $a = e5$ (again, using congruence closure and transitivity).

This fact goes back to the arithmetic solver, which evaluates the following constraints:

$$
\begin{aligned}
&e1 = e2 - e3 \\
&e4 = 0 \\
&e5 = a + 2 \\
&x = y \\
&e2 = e3 \\
&a = e5
\end{aligned}
$$

Now there is a contradiction: $a = e5$ but $e5 = a + 2$. That means the original formula is unsatisfiable.

In this case, one theory was able to infer equality relationships that another theory could directly use. But sometimes a theory doesn't figure out an equality relationship, but only certain correlations, e.g., e1 is either equal to e2 or e3. In the more general case, we can simply generate a formula that represents all possible equalities between shared symbols, which would look something like:

$$(e1 = e2 \vee e1 \neq e2) \wedge (e2 = e3 \vee e2 \neq e3) \wedge (e1 = e3 \vee e1 \neq e3) \wedge ...$$

We can now look at all possible combinations of equalities. In fact, we can use DPLL to do this, and DPLL also explains how we can combine expressions in the various theories with boolean operators such as $\wedge$ and $\vee$. If we have a formula such as:[2]

$$x \geq 0 \wedge y = x + 1 \wedge (y > 2 \vee y < 1)$$

We can then convert each arithmetic (or uninterpreted function) formula into a fresh propositional symbol, to get:

$$p1 \wedge p2 \wedge (p3 \vee p4)$$

and then run a SAT solver using DPLL. DPLL will return a satisfying assignment, such as $p1, p2, \neg p3, p4$. We then check this against each of the theories. In this case, the theory of arithmetic finds a contradiction: $p1$, $p2$, and $p4$ can't all be true, because $p1$ and $p2$ together imply that $y \geq 1$. We add a clause saying that these can't all be true and give it back to the SAT solver:

$$p1 \wedge p2 \wedge (p3 \vee p4) \wedge (\neg p1 \vee \neg p2 \vee \neg p3)$$

Running DPLL again gives us $p1, p2, p3, \neg p4$. We check this against the theory of arithmetic, and it all works out.

3.3 DPLL(T)

This use of DPLL parameterized with respect to a set of theories T is called DPLL(T) or (DPLL-T), and it is an SMT algorithm. At a high level, DPLL(T) works as we illustrated above: it converts mixed constraints to boolean constraints and then runs DPLL, and then checks the resulting assignments with the underlying theories to determine if they are valid. The version of DPLL used in DPLL(T) has two key changes compared to the original, however.

First, DPLL(T) does not use the pure variable elimination optimization. This is because, in pure propositional logic, variables are necessarily independent. If some variable x only appears

[2]If we had multiple theories, I am assuming we've already added the equality constraints between them, as described above.

positively, you can set it to `true` and save time. With theories, variables may be dependent. For example, consider

$$(\mathbf{x} > \mathbf{10} \vee x < 3) \wedge (\mathbf{x} > \mathbf{10} \vee x < 9) \wedge (x < 7)$$

In the above, $x > 10$, but if we just set that term to be true as part of the model, the other terms all become false. We cannot simply skip over it.

Second, unit propagation interacts with the theories to add constraints to the formula when available. We saw this example above, when the theory of arithmetic found a contraditiction in the formula

$$p1 \wedge p2 \wedge (p3 \vee p4)$$

And the solving procedure added a clause to the formula before giving it back to the SAT solver.

3.4 Bonus: Arithmetic solvers

We discussed above how the solver for the theory of uninterpreted functions work; how does the arithmetic solver work? In cases like the above example where we assert formulas of the form $y = x + 1$, we can eliminate y by substituting it with $x + 1$ everywhere. In the cases where we only constrain a variable using inequalities, there is a more general approach called Fourier-Motzkin Elimination. In this approach, we take all inequalities that involve a variable x and transform them into one of the following forms:

$$A \leq x$$
$$x \leq B$$

where A and B are linear formulas that don't include x. We can then eliminate x, replacing the above formulas with the equation $A \leq B$. If we have multiple formulas with x on the left and/or right, we just conjoin the cross product. There are various optimizations that are applied in practice, but the basic algorithm is general and provides a broad understanding of how arithmetic solvers work.