Lecture notes for: Introduction to SMT solving

Lindsey Kuper

October 11, 2019

These are lecture notes to accompany sections 3.1-3.4 (pp. 59-72) of Kroening and Strichman's *Decision Procedures: An Algorithmic Point of View*, second edition. See http://composition.al/CSE290Q-2019-09/readings.html for the full collection of notes.

Agenda

- Eager and lazy solvers
- The DPLL(T) framework
  - Boolean abstractions and overapproximations of satisfiability
  - The role of the theory solver
- Offline and online solving

Eager and lazy solvers

We've been discussing the architecture of CDCL SAT solvers. Now we're going to talk about how we can solve formulas written using more interesting theories. In particular, we want to look at solvers for quantifier-free fragments of first-order theories, as we talked about a few lectures ago. Instead of a plain old SAT formula, we might have, say, a linear real arithmetic formula, such as, for instance,

$$((x_1 > 1) \wedge (2x_1 < 5)) \vee \neg(x_2 = 0).$$

So, our formulas still have *propositional structure*, but the atoms of the formulas depend on the theory we're using.

There are traditionally two approaches to SMT solving: the **eager** approach and the **lazy** approach. Eager SMT solving involves compiling the entire SMT formula to a plain old SAT formula and then solving the SAT formula with a SAT solver. This is possible to do for any decidable theory, and is similar in spirit to to compiling a program in a high-level language to assembly language. But the formula can grow in size exponentially when it is encoded as SAT, and moreover, the eager approach means we lose the opportunity to reason at the (higher) level of the theory we're using, instead of at the (lower) level of SAT formulas.

That said, there are some state-of-the-art eager SMT solvers. One of them, for instance, is the *STP* solver, which is what the first paper on our reading list is about. STP solves formulas in the theory of bit-vectors and arrays by eagerly compiling them to SAT (after some preprocessing).

Most modern SMT solvers, though, are *lazy*. In the lazy approach, a SAT solver is still involved, but we also have specialized *theory solvers* that interact with the underlying SAT solver during the solving process. That is, the solving process is an interplay between the SAT solver and the theory solver.

An SMT solver like Z3 has multiple built-in theory solvers, each for a specific theory, although for a given problem you might only use one of the built-in theories.

Today we're going to restrict our attention to discussing lazy SMT solvers, which are the topic of chapter 3 of Kroening and Strichman. It's a sign of how much lazy SMT solving has become the norm in the last ten years that the second edition of Kroening and Strichman, which came out in 2016,

made chapter 3 only about lazy solving. There used to be stuff about eager solving in chapter 3 in the first edition of the book, which came out in 2008, but in the new edition they demoted it to chapter 11. (So if you want to read about eager solvers, look at chapter 11 of the book.)

The DPLL(T) framework

The lazy SMT solver framework we'll discuss is commonly known as DPLL(T). The "(T)" part is to indicate that it is parameterized by some theory T. That is, you can plug in a theory solver for your theory of choice into the DPLL(T) framework, and that theory solver will interact with the underlying SAT solver, which, generally speaking, is a CDCL SAT solver. (As I've mentioned, DPLL is the predecessor to CDCL, and doesn't involve clause learning.) So DPLL(T) is a misnomer; it should really be called CDCL(T), but the name DPLL(T) seems to have stuck.

In the DPLL(T) framework, the underlying CDCL SAT solver and the theory solver for the theory T have distinct roles to play.

What does the theory solver have to be able to do? The theory solver that you plug in to the DPLL(T) framework needs to be able to handle *conjunctions of literals* from the theory T. (Recall that a literal is either an atom or its negation.)

Now, building a theory solver that can handle conjunctions of literals for a given theory is not necessarily easy to do. In fact, the design and implementation of these theory solvers is the topic of most of the rest of the Kroening and Strichman book after this chapter!

But let's assume for a moment that we do have such a theory solver for a particular theory. That is, we have a solver that can take a formula that

consists of a conjunction of literals from T and tell us either "satisfiable" or "unsatisfiable".

And we'll follow Kroening and Strichman and pick a really simple theory to start with, which is the theory of equality.

Boolean abstractions and overapproximations of satisfiability

To do lazy SMT solving, we need a **Boolean abstraction** of the original formula. These are also known as **propositional skeletons**, and that's what the book calls them. But I'll just say "Boolean abstraction" because it's shorter.

To get the Boolean abstraction, or propositional skeleton if you like, we take each atom of the linear arithmetic formula and replace it with a new Boolean variable. Recall that atoms are formulas with no propositional structure.

For instance, in our linear real arithmetic formula $((x_1 > 1) \wedge (2x_1 < 5)) \vee \neg(x_2 = 0)$, the atoms are $(x_1 > 1)$, $(2x_1 < 5)$, and $(x_2 = 0)$.

So the Boolean abstraction of that formula is $(b_1 \wedge b_2) \vee \neg b_3$ (I'm using b to make it more obvious that the variables are Booleans).

Notice that the Boolean abstraction has three variables (which are Boolean variables) whereas the original formula had two (which are real numbers). So in general, the number of variables doesn't stay the same from the original formula to the Boolean abstraction.

The function that turns a formula into its Boolean abstraction is called, unsurprisingly, a **Boolean abstraction function**.

For our running example, we're using the theory of equality. In this theory,

atoms are equalities, like $x = y$. So here's a formula in the theory of equality:

$$x = y \wedge ((y = z \wedge \neg(x = z)) \vee x = z)$$

And its Boolean abstraction looks like this:

$$b_1 \wedge ((b_2 \wedge \neg b_3) \vee b_3)$$

Since the Boolean abstraction is just a SAT formula, we can pass it to a SAT solver, which will run and come up with an assignment.

Q: Is the Boolean abstraction satisfiable?

A: Sure, of course it is. There are multiple satisfying assignments, even. But let's suppose the SAT solver happens to produce the following satisfying assignment:

$$\{b_1 = \text{true}, b_2 = \text{true}, b_3 = \text{false}\}$$

But that's a satisfying assignment for the Boolean abstraction, not for the original formula.

Q: If the Boolean abstraction was satisfiable, does that mean that the original formula is satisfiable?

A: No, not necessarily! So we still have to determine whether the original formula is satisfiable. We'll come back to that.

But let's say we had a different equality logic formula to begin with. Say we had this.

$$(x = y \vee x = z) \wedge \neg(x = y) \wedge (x = y \vee \neg(x = z))$$

Then the Boolean abstraction would be this:

$(b_1 \vee b_2) \wedge \neg b_1 \wedge (b_1 \vee \neg b_2)$

Q: Is this one satisfiable?

A: No, this one is unsatisfiable.

Q: Does that mean that the original formula was unsatisfiable?

A: It turns out that the answer is yes! In general, *if the Boolean abstraction of a formula is unsatisfiable, then so is the original formula*. However, if the Boolean abstraction is satisfiable, then the original formula *may or may not* be satisfiable.

- Boolean abstraction is unsatisfiable $\implies$ original formula is unsatisfiable modulo the theory, and we're done, yay!
- Boolean abstraction is satisfiable $\implies$ original formula may or may not be satisfiable modulo the theory, and we have more work to do.

We say that the Boolean abstraction **overapproximates** the satisfiability of the original formula.

That "more work to do" part is where the theory solver comes in.

The role of the theory solver

OK, so suppose we have a Boolean abstraction that *is* satisfiable. Then what?

Well, now begins a conversation between the SAT solver and the theory solver.

Going back to our original formula $x = y \wedge ((y = z \wedge \neg(x = z)) \vee x = z)$, the SAT solver determined that its Boolean abstraction $b_1 \wedge ((b_2 \wedge \neg b_3) \vee$

$b_3)$ is satisfiable with the assignment $\{b_1 = \text{true}, b_2 = \text{true}, b_3 = \text{false}\}$.

But we still need to figure out whether the original formula is satisfiable.

Now what we can do is go back to the theory solver. Recall that the theory solver can only solve conjunctive formulas in the theory. But we can construct one of those by just writing down the formula that corresponds to the assignment that the SAT solver just gave us!

In particular, the SAT solver gave us this assignment: $\{b_1 = \text{true}, b_2 = \text{true}, b_3 = \text{false}\}$. b_1 corresponds to the atom $x = y$, b_2 to the atom $y = z$, and b_3 to the atom $x = z$. So, we can write down the following formula:

$$x = y \wedge y = z \wedge \neg(x = z)$$

We hand that to the theory solver (recall we're assuming we have one of those). And the theory solver runs, and what do you know? It returns "unsatisfiable".

So now what? Well, what happens now is similar in spirit to what happened in CDCL, when we ran into a conflict using a particular partial assignment and then we had to learn a clause that would prevent us from trying that partial assignment again.

So, what we want to do is try again with the SAT solver and get it to come up with a *different* assignment that *also* satisfies the Boolean abstraction, and that moreover corresponds to a satisfiable formula in the theory. So what we can do is take the *negation* of the unsatisfiable formula above:

$$\neg(x = y \wedge y = z \wedge \neg(x = z))$$

Because this is a negation of an unsatisfiable formula, that means it's always true. We convert that back into a Boolean abstraction:

$$\neg(b_1 \wedge b_2 \wedge \neg(b_3))$$

which is the equivalent of:

$$\neg b_1 \vee \neg b_2 \vee b_3$$

So that's a clause that we can add to our Boolean abstraction:

$$b_1 \wedge ((b_2 \wedge \neg b_3) \vee b_3) \wedge (\neg b_1 \vee \neg b_2 \vee b_3)$$

This newly added clause is called a **blocking clause**. The purpose of the blocking clause is to prevent the SAT solver from coming up with the same assignment that it came up with last time.

The blocking clause is analogous to the conflict clauses that the SAT solver learns during CDCL. In fact, some sources call it a "theory conflict clause". Kroening and Strichman call it a blocking clause. Either name works.

And now we go back to the SAT solver once again.

The SAT solver runs, and it comes up with another satisfying assignment:

$$\{b_1 = \text{true}, b_2 = \text{true}, b_3 = \text{true}\}.$$

And this new satisfying assignment from the SAT solver corresponds to the following formula in the theory:

$$x = y \wedge y = z \wedge x = z$$

So we run *that* through the theory solver, and the theory solver reports that it is satisfiable.

And this means that the original formula in the theory, which was

$$x = y \wedge ((y = z \wedge \neg(x = z)) \vee x = z),$$

is indeed satisfiable.

To sum up, this is the overall pattern:

- Given a formula in the theory, construct a Boolean abstraction of it that *overapproximates* satisfiability.
- If the SAT solver returns "unsatisfiable", we're done, yay.
- If the SAT solver returns "satisfiable", check to make sure that the satisfying assignment is legitimate, as follows:
  - Run the theory solver on the formula corresponding to the assignment returned by the SAT solver.
    - If the theory solver returns "satisfiable", we're done, yay.
    - If the theory solver returns "unsatisfiable", add a blocking clause to the Boolean abstraction, and try again with the SAT solver.

Q: Do you see any problems with doing things this way?

A: So, one problem that I see is that when you add the blocking clause to the Boolean abstraction, it rules out only *one particular assignment*. It will stop the solver from making *that same exact assignment* again. But that's not very efficient. We might end up having to go through every single satisfying assignment, and then checking the formula that corresponds to that assignment using the theory solver. If there are a lot of satisfying assignments, that could be really slow.

So ruling out one bad assignment at a time will work, eventually, but it will be slow. It would be really nice if we could add blocking clauses that would rule out large chunks of the space of assignments in one fell swoop.

So for example, say you have this formula that's a big conjunction of a thousand literals:

$$x = y \wedge x < y \wedge l_1 \wedge l_2 \wedge ... \wedge l_{998}$$

We can tell by looking at this that it's unsatisfiable, because there's no way for the first two atoms to both be satisfiable. Regardless of what l_1 through l_{998} are, we know that it's unsatisfiable. So a blocking clause that only rules out one of the assignments to the Boolean abstraction of this formula is not that helpful.

We call $x = y \wedge x < y$ the *minimal unsatisfiable core* of the formula. And it turns out you can compute these minimal unsatisfiable cores and add the negation of *them* instead.

Online and offline solving

Lazy SMT solvers can be further subcategorized into **offline** and **online** solvers. I don't think the book uses this terminology, but it's used in other sources.

- Offline lazy SMT solvers are SMT solvers where the theory solver treats the underlying SAT solver as a *black box*. What I mean by "black box" is that the interaction between the theory solver and the SAT solver consists of the theory solver waiting for the SAT solver to generate a complete satisfying assignment to the Boolean abstraction, and then checking whether the corresponding formula is satisfiable in the theory solver. What we just talked about doing above is offline solving. The book calls this the "Lazy-Basic" algorithm.
- Online lazy SMT solvers have a tighter integration between the underlying SAT solver and the theory solver. And this is done by integrating the theory solver right into the CDCL loop. I *think* this is what the book calls this the "Lazy-CDCL" algorithm.

In particular, suppose you're running CDCL and you've made a partial as-

signment in the Decide step and done BCP. Now, if there's no conflict, you can immediately invoke the theory solver on the formula corresponding to the *partial* assignment. If that turns out to be unsatisfiable in the theory solver, you can add its negation (the blocking clause) to the formula, just like you would normally add a conflict clause during CDCL.

(Better yet, you can add the negation of the minimal unsatisfiable core.)