15-414: Bug Catching: Automated Program Verification

Lecture Notes on SAT & SMT Certificates

Matt Fredrikson

Carnegie Mellon University
Lecture 20
Tuesday, March 5, 2022

1 Introduction

Given the importance of SAT and SMT solvers in verification and bug-finding, it is vitally important that their results are trustworthy, and that they are not a potential source of bugs. One approach to gaining this trust is to verify the solvers themselves, and this has been done in a few specialized cases [FBL18]. However, this is usually not practical for several reasons. Solvers tend to be implemented in languages like C or C++, and are heavily optimized, which makes them difficult to verify. The strategies that state-of-the-art solvers deploy are constantly changing, and each such update could potentially invalidate the proofs for previously-verified functionality. The time and labor cost of verifying each update is simply too great.

An alternative is to have the solver produce a *certificate* that demonstrates the correctness of its output. On receiving an answer from the solver, a checker can verify that the solver's work on a given single input was correct. As long as the certificate satisfies a few pragmatic concerns, this approach largely addresses the matter of trust cited above, because it isn't necessary to conclude that the solver will *always* produce correct results, but rather that any input we actually give it yields a correct result.

- The certificate should be *efficient* to verify: it should not require as much work to check the solver's work as it would have to decide the original formula, and ideally it should require much less.
- The correctness of the verifier is crucial, so the verification procedure should be simple (ideally, verifiable), and verifier implementations should remain stable over time.

- Producing a certificate should not overburden the solver by imposing heavy computation and memory requirements.
- Certificates should be expressive enough to account for new optimization techniques and heuristics that may be implemented in future updates to the solver.

However, when contrasted with the approach of verifying the solver's implementation directly, certificates pose one drawback: if the certificate checker claims that the solver produced an invalid certificate, then the status of the corresponding input remains unknown.

Finally, it is worth noting that this paradigm of using certificates to establish correctness is not limited to SAT and SMT solvers. It has been applied to a range of numeric algorithms in the past [BK95], and is currently finding applications in trustworthy artificial intelligence [WK18].

Learning goals. After this lecture, you will:

- Learn how producing certificates that demonstrate the correctness of a solver's result is a practical alternative to verifying the solver itself.
- Understand how the resolution principle from Lecture 13 can be applied to generate certificates for SAT solvers.
- Understand how different certificate encodings lead to tradeoffs in solver efficiency and verifier complexity, efficiency, and verifiability.

2 Review: Resolution and SAT solvers

For today's lecture we will mainly return to propositional formulas. Most SAT solvers accept propositional formulas in conjunctive normal form (CNF). A formula P is in conjunctive normal form if it is a conjunction of disjunctions of literals, i.e., it has the form:

$$(l_{00} \vee l_{01} \vee \cdots \vee l_{0n_0}) \wedge \cdots \wedge (l_{m0} \vee l_{m1} \vee \cdots \vee l_{mn_m})$$

where l_{ij} is the jth literal (i.e., variable or negated variable) in the ith *clause* of F. Note that if there are n variables appearing in total in P, then each clause will have at most n literals because any clause with both p and $\neg p$ can be trivially removed. A clause may have fewer than n variables, as those that do not appear in a clause are treated as "don't care". For most algorithms, it is convenient to think of a clause as a *set of literals* $\{l_1, \ldots, l_n\}$ where we write $\perp$ for the empty set. We will still write this as $l_1 \vee \ldots \vee l_n$.

Resolution is both the name of a rule of inference and an algorithm searching for a *refutation* of a theory T. Such a *refutation* is evidence that the theory is unsatisfiable. The rule of inference can be written as follows:

$$\frac{p \vee C \quad \neg p \vee D}{C \vee D}\ \text{resolution}$$

```
1 let rec dpll (f: formula) : bool =
2   let fp = bcp f in
3   match fp with
4   | Some True -> true
5   | Some False -> false
6   | None ->
7     begin
8     let p = choose_var f in
9     let ft = (subst_var f p true) in
10    let ff = (subst_var f p false) in
11    dpll ft || dpll ff
12    end
13  end
```

Figure 1: Basic DPLL algorithm for deciding propositional satisfiability.

The two premises of the rule are clauses (really: sets of literals), even if we write them using disjunction, so C has no copy of p and D has no copy of $\neg p$. When it's convenient, we will denote the application of resolution to clauses C and D on literal p as $C \bowtie_p D$.

To use this rule to find a refutation, we repeatedly apply it to clauses that are already in the formula, each time adding a new clause as a consequence of the rule. If we eventually add the empty clause, we conclude that the original theory was in fact *unsatisfiable* because clauses added via resolution preserve satisfiability, and no assignment can satisfy the empty clause.

While this approach was used in the past to decide satisfiability, the algorithms that are based on it are not terribly efficient, and have trouble scaling to the types of formulas that are relevant to many applications today. In the 1960's, Martin Davis, Hilary Putnam, George Logemann, and Donald Loveland developed the DPLL algorithm for deciding satisfiability, as shown in Figure 1

DPLL is a relatively straightforward approach based on recursive case-splitting on each variable in the formula, with one important exception that is apparent on line 2. Recall that a clause is *unit* under a partial assignment to the variables if exactly one literal in the clause is unassigned. *Boolean constraint propagation* (BCP), or sometimes *unit propagation* for short, takes advantage of the fact that any satisfying assignment to the formula must satisfy the unassigned literal in a unit clause, and adds these assignments at each step until no unit clauses remain. In practice, this optimization is very helpful, and when implemented intelligently leads to solvers that can handle impressively large formula, despite the inherent complexity of deciding satisfiability.

In the 1990's, another significant improvement was introduced based on learning additional *conflict clauses* during search. Whenever the solver generates a partial assignment that conflicts with its clause set (i.e., under which P evaluates to false), a procedure called *conflict-driven clause learning* analyzes the trace leading to the conflict, and derives a new clause that explains the "reason" for the conflict. Adding this clause to the formula will prevent the solver from exploring assignments that lead to similar conflicts in the future. The procedure for finding a conflict clause under partial assign-

ment M, on finding an unsatisfied clause C, is as follows.

1. While C contains implied literals, do:
2. Let l be the most recent implied literal in C
3. Let C' be the clause that implied l by unit propagation
4. Update $C := C \bowtie_l C'$

Because resolution preserves satisfiability, adding the resulting clause to the formula is safe. While there are other methods for deriving a conflict clause, they are all based on resolution, so this basic procedure will suffice for our purposes today.

3 Certificates for DPLL

A certificate A for a SAT solver should contain all of the information necessary to determine that its final result—either a satisfying assignment or an unsat decision—is correct for the original formula P that it was given. In the first case, the satisfying assignment itself serves as a viable certificate: it is straightforward to evaluate a set of clauses given an assignment, and doing so clearly requires less work than finding one. In the second case, it is not as obvious what the certificate should look like. Naively, one might imagine enumerating each conflicting assignment that the solver encountered. But what about those that it skipped over because of unit propagation? More importantly, will checking this really be less work than what the solver has already done?

The resolution inference rule that we studied in Lecture 13 is refutation-complete, which means that for any unsatisfiable propositional formula, there exists a resolution refutation for it. This suggests that proofs using only the resolution rule could work well as certificates for SAT solvers: whenever the verifier needs to check an unsat decision, it could check that a proof given by the solver only contains correct applications of resolution, and that all of the clauses in the leaves of the proof appeared in the original CNF formula.

A few questions remain about the viability of resolution for this purpose. DPLL does not seem to use resolution during its search, so can we efficiently generate a resolution proof from the work that it does? If that is the case, then will the proof be small enough that checking it is still a reasonable amount of work?

3.1 Basic Approach: DPLL Without BCP

To answer the first question, we'll start by simplifying DPLL by removing the unit propagation (BCP) step. Looking at Figure 1, this means that the procedure will just enumerate all possible assignments, and return unsat if it fails to find a satisfying one. However, because it eagerly evaluates the formula under each extension to an assignment, it might stop early on some recursive calls if it encounters a conflict.

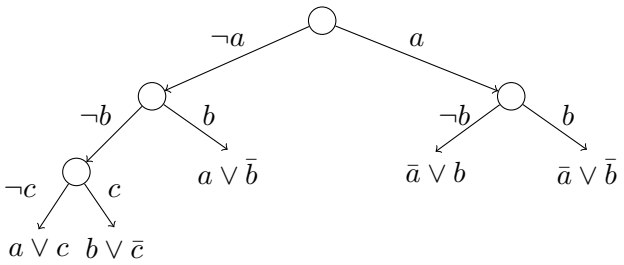
Without BCP, it is not difficult to see that we can extract a resolution proof from the work that this solver does. Consider a case where the solver has decided all but one

variable l, so it branches, assigning l and $\neg l$ and encountering conflicts in both cases. Let C_+ be the clause that conflicted with the assignment l, and C_- be the one that conflicted with $\neg l$. Then $C_l = C_+ \bowtie_l C_-$ is a clause containing literals other than l, which must be covered by any satisfying assignment if both C_+ and C_- are to be satisfied as well. This can be passed along when backtracking, and resolved with a corresponding clause if the other branch of the decision immediately before l is also unsatisfiable.

To see this in action, consider the formula P:

$$(\neg b \vee c) \wedge (a \vee c) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg b) \wedge (b \vee \neg c) \tag{1}$$

The search performed by DPLL without unit propagation is shown below, where the leaves are the clauses from P that conflicted with each partial (or complete) assignment.



Looking at the only branch that occured on c on the bottom-left, observe what we get from resolving the two conflicting clauses $R_- = a \vee c$, $R_+ = b \vee \neg c$:

$$R_c = a \vee c \bowtie_c b \vee \neg c = a \vee b$$

No satisfying assignment can cover both R_+ and R_- by its value for c, so it must satisfy $a \vee b$. But for the previous branch, the partial assignment with b conflicted on $a \vee \neg b$. Resolving this with the result passed along for the $\neg b$ branch,

$$R_{b1} = a \vee b \bowtie_b a \vee \neg b = a$$

On the other side of the tree, where the assignments have a set, both branches b and $\neg b$ conflicted as well, yielding,

$$R_{b2} = \neg a \vee b \bowtie_b \neg a \vee \neg b = \neg a$$

Then at the topmost branch on the variable a, it is clear that resolving R_{b1} and R_{b2} will yield the empty clause $\bot$. All of this can be combined to produce a refutation of the original formula. Below, let $C_0, C_1, \ldots, C_5$ correspond to the clauses as they are shown

in the example from left to right.

$a \vee c$	C_1
$\neg a \vee b$	C_2
$\neg a \vee \neg b$	C_3
$a \vee \neg b$	C_4
$b \vee \neg c$	C_5
$a \vee b$	$C_6 = C_1 \bowtie_c C_5$
a	$C_7 = C_6 \bowtie_b C_4$
$\neg a$	$C_8 = C_2 \bowtie_b C_3$
$\bot$	$C_9 = C_7 \bowtie_a C_8$

The approach that we have just outlined is sufficient to produce a correct refutation for any run of DPLL without unit propagation, with one minor catch. On some branches, the resolvent passed along from one of the children may not contain an occurrence of the variable that was branched on. For example, there are no satisfying assignments containing $\neg a$:

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

The extension $\neg a, b$ conflicts with the clause $a \vee \neg b$. The extension $\neg a, \neg b$ conflicts with the remaining four clauses, and their resolvent (given by $(C_1 \bowtie_d C_2) \bowtie_c (C_3 \bowtie_e C_4)$) is the unit clause a, because these four clauses contain no occurrences of the variable b. It is not possible to resolve $\neg a, b$ and a on b, so the correct thing to do is to pass along the clause that does *not* contain the branching variable (in this case, a). The correctness of this choice is apparent in the current example: the fact that all extensions of $\neg a$ led to conflicts is reflected by the resolvent a, which "explains" these conflicts by asserting that any satisfying assignment must have a.

The correctness of this procedure for generating refutations follows from the fact that each resolvent generated along the way is composed of literals that are negations of the partial assignment that led to the current conflict. For instance, in the example from earlier, the resolvent obtained after branching on c, under the partial assignment $[\neg a, \neg b]$, was $a \vee b$. Carrying this property to the beginning of the procedure's search, branching on the first variable l will on one side yield a clause containing $\neg l$, and on the other one containing l, which will thus provide a refutation.

This property can be proved by induction on the tree implicit in any DPLL-like search. The base case arises when the current partial assignment conflicts with some clause in the formula; it is immediate that the "resolvent" (a leaf in the refutation), given by the conflicting clause, will only contain negated literals from the partial assignment. In the inductive case, resolving on the clauses obtained on either branch will yield a new clause with all literals for the most recently-assigned variable eliminated. The inductive hypothesis establishes that the remaining literals are all negations of those in the assignment that immediately prefixes the most recent one.

Step	**Partial valuation**
Start with an empty partial valuation.	$\{\}$
Decide a.	$\{a \mapsto \texttt{true}\}$
Propagate b from unit clause C_2.	$\{a \mapsto \texttt{true}, b \mapsto \texttt{true}\}$
Clause C_3 is conflicting. Backtrack.	$\{\}$
Decide $\neg a$.	$\{a \mapsto \texttt{false}\}$
Propagate c from unit clause C_1.	$\{a \mapsto \texttt{false}, c \mapsto \texttt{true}\}$
Propagate b from unit clause C_5.	$\{a \mapsto \texttt{false}, c \mapsto \texttt{true}, b \mapsto \texttt{true}\}$
Clause C_4 is conflicting.	**Unsat**

Figure 2: DPLL trace of example from Equation 1.

3.2 Now with BCP

Now we will see how to extend this procedure to account for unit propagation. We begin by listing out a potential trace of DPLL on the same example from earlier in Figure 2. Because this is a short trace, it is readily apparent that we cannot just ignore the unit propagation steps and apply the same method from before. If we were to do that, then we see that there are just two conflicting clauses that arose during the search: $C_3 = \neg a \vee \neg b$ and $C_4 = a \vee \neg b$. Resolving these on the only branch taken (a) yields $\neg b$, which does not amount to a refutation.

Observe that unit propagation can be viewed as a special case of resolution, where the "remainder" literals for one of the clauses is empty.

$$\frac{p \vee C \quad \neg p}{C}\ \text{unit}$$

Each time that DPLL applies unit propagation, it is effectively performing this special case of resolution by replacing the occurrence of the unit literal in every other clause with $true$. When the unit clause is not *literally* a unit clause, it is still considered as such under a given partial assignment that replaced all of the other literals with $false$. So, we can account for unit propagation in the certificate by passing along the resolvent of the conflicting clause and the unit clause, on the unit literal.

In the current example, after encountering a conflict on $C_3 = \neg a \vee \neg b$, the procedure would generate the following for the branch deciding a:

$$C_{b1} = C_3 \bowtie C_2 = \neg a$$

Likewise, for the branch deciding $\neg a$, it would first resolve the conflict clause C_4 with the most recent unit clause C_5 on its unit literal b, and then resolve with the first unit clause C_1 on its unit literal c:

$$\begin{array}{lcll} C_{b2} & = & C_4 \bowtie_b C_5 & = & a \vee \neg c \\ C_c & = & C_{b2} \bowtie_c C_1 & = & a \end{array}$$

As before, the resulting clauses that are resolved at the top level are $\neg a$ and a, giving a refutation.

Step	Partial valuation
Start with an empty partial valuation.	$\{\}$
Decide a.	$\{a \mapsto \texttt{true}\}$
Propagate b from unit clause C_2.	$\{a \mapsto \texttt{true}, b \mapsto \texttt{true}\}$
Clause C_3 is conflicting.	
Learn $C_6 = \neg a$. Backtrack.	$\{\}$
Propagate $\neg a$ from C_6.	$\{a \mapsto \texttt{false}\}$
Propagate c from unit clause C_1.	$\{a \mapsto \texttt{false}, c \mapsto \texttt{true}\}$
Propagate b from unit clause C_5.	$\{a \mapsto \texttt{false}, c \mapsto \texttt{true}, b \mapsto \texttt{true}\}$
Clause C_4 is conflicting.	**Unsat**

Figure 3: DPLL with clause learning, example from Equation 1.

This procedure maintains the property discussed earlier, that the resolvent at each step only contains negations of literals appearing in the current partial assignment. Observe that when unit propagation occurs, all literals in the unit clause C_u except the unit literal l must be negated from the current partial assignment. The clause C that is resolved with the unit clause by this procedure must also contain only negations of the current assignment: either the partial assignment extended with l directly conflicted with C, or via the inductive hypothesis. Then as before, resolving on the unit literal leaves the union of literals in C_u and C (minus l).

3.3 Incorporating Clause Learning

The last point that we'll cover regarding certificates for SAT is how to incorporate clause learning into certificate generation. Recall that clauses are learned by the application of resolution, starting with the formula clause that was found to be conflicting, and continuing on through resolution with unit clauses in reverse chronological order until there aren't any implied literals left. Because this process is already a direct application of the resolution rule, these steps can be incorporated directly into the certificate: each time a conflict clause is learned, add the corresponding resolution step to the certificate.

Aside from recording all of the resolution steps used to derive the learned conflict clauses, the solver must deduce an order to apply resolution to the learned clauses to produce a refutation. This can be accomplished by a process similar to learning new conflict clauses, applied to the last conflict that occurred immediately prior to returning unsat. Learned clauses have the property discussed in the previous two sections, that their literals are negations of the current partial assignment. This means that backtracking to the level above the most recently-decided literal will always force a unit propagation after adding the new clause, and when the solver reaches its final conflict prior to returning unsat, there will be no decided literals (i.e., the conflict will have arisen purely due to unit propagation). This means that applying resolution to unit clauses in reverse chronological order until only the (empty) set of decided variables remains will yield the empty clause, thus giving a refutation.

Let's see this in action on the same example from the previous two sections. Figure 3 shows a trace of DPLL on the formula from Eq. 1 with clause learning. After deciding a, the conflict on C_3 arises, and resolving C_2 and C_3 yields the clause $C_6 = \neg a$ to add to the formula. This clause is of course unit, which triggers the chain of propagations from C_1 and C_5 leading to the conflict on C_4. As this conflict arose when there were the solver's assignment was empty, the formula must be unsat. Now observe that chaining resolution on the clauses involved in the last conflict leads to a refutation:

$$\begin{aligned} C_4 \bowtie_b C_5 \bowtie_c C_1 \bowtie_a C_6 &= a \vee \neg b \bowtie_b b \vee \neg c \bowtie_c a \vee c \bowtie_a \neg a \\ &= a \vee \neg c \bowtie_c a \vee c \bowtie_a \neg a \\ &= a \bowtie_a \neg a \\ &= \bot \end{aligned}$$

The certificate thus needs to (1) show the resolution steps leading to any learned clause involved in the final conflict, and (2) retrace the final conflict in reverse chronological order to obtain a refutation.

4 Certificate Encodings

There are two principle formats for encoding resolution proofs that have been adopted by state-of-the-art SAT solvers. The first, known (perhaps unsurprisingly) as *resolution proofs*, more closely resembles the example resolution proofs seen in this lecture and Lecture 13. The second, *clausal proofs*, is more concise and is especially common in solvers that use clause learning, as these certificates can be easily extracted from the work needed to learn conflict clauses. Each has its advantages and disadvantages, so the "right" encoding depends on the context in which certificates will be used.

4.1 Resolution Proofs

The most intuitively straightforward way to represent a resolution proof is to begin by listing all of the relevant clauses from the input formula, followed by each application of the resolution rule with references to both antecedent clauses. This is essentially what a resolution proof consists of, but there are a few opportunities to optimize the encoding for smaller proofs.

First, as became apparent in Section 3.2 and 3.3, resolution steps are often chained sequentially. Rather than encoding each step of the chain as a distinct step in the certificate, the entire chain can be represented more compactly as one step in the proof by listing the ordered sequence of clauses appearing in the chain. In fact, it is not strictly necessary to insist that the sequence be given in the correct order, as it is not expensive for the certificate checker to deduce the correct ordering, as long as the resolvent is given in the certificate as well.

Second, as long as the antecedent clauses in chains is given in the correct order, then resolvent clauses do not need to be given explicitly in the certificate, as the checker can easily work the resolvent out from its antecedent clauses. Likewise, the variable that

```
1  1   1   3   0   0
2  2  -1   2   0   0
3  3  -1  -2   0   0
4  4   1  -2   0   0
5  5   2  -3   0   0
6  6   *   0   2   3   0
7  7   0   4   5   1   6   0
```

```
1 -1 0
2  0
```

Figure 4: Representative encodings of the example proof from Section 3.3. On the left is a resolution proof in TraceCheck format, and on the right is a clausal proof in RUP format.

resolution is being applied on, i.e. the p in $\bowtie_p$, does not need to be stated explicitly. The checker can determine which variables are opposing in a given pair of antecedent clauses, and if there is more than one such pair, then it assumes that resolution is applied to all of them. This is not problematic when it comes to expressiveness, because the ultimate goal is always to derive an empty clause, so steps that remove more than one variable at a time are economical.

A representative format that uses this encoding is modeled by the TraceCheck tool, developed as part of the PicoSAT effort [Bie08]. The left trace in Figure 4 is the example from the previous section in this format. Variables are represented by integers starting from 1, signed - to denote negation. Each line of this format represents a clause, along with any antecedents needed to derive it. The first number in each line is an index, used by later steps to refer to the clause represented by that line. After the index, the literals in the clause are given, terminated by a `0`. Then, the list of antecedents needed to derive that clause in a resolution chain are listed, and terminated by a `0`.

Looking at Figure 4, the first line is indexed `1`, and represents the clause `1 3` (i.e. $a \vee c$). After the first terminating `0`, the second one follows with no intervenining indices to represent resolution antecedents, which means that this line encodes a clause that was assumed from those given in the original formula. This is also true for indices `2`, `3`, `4`, and `5`. The clause at index `6` has literals given by `*`, which means that the checker should compute the resolvent from the list of antecedents that follow the first terminating `0`: `2 3`. In other words, the clause at `6` is the one learned by the first conflict encountered in the example from Section 3.3, by resolving C_2 and C_3 to obtain $\neg a$. It is not necessary to have the checker compute the resolvent, and this line could have been given by:

```
1 6 -1  0  2  3  0
```

Finally, on the last line the first `0` terminator follows the index number without any intervening clause literals, which denotes that this step encodes the derivation of the empty clause. It encodes the chain given in Section 3.3 directly.

4.2 Clausal Proofs

The second predominant type of encoding takes advantage of the fact that the clauses learned by solvers via resolution satisfy the *reverse unit propagation* property.

Definition 1 (Reverse Unit Propagation (RUP) Clause)**.** A clause C is called a reverse unit propagation clause with respect to CNF formula P if and only if the formula $P \wedge \neg C$ simplifies to $\bot$ via unit propagation.

To understand Definition 1, first note that if $C = l_1 \vee l_2 \vee \cdots \vee l_n$ is a clause, then $\neg C = \neg l_1 \wedge \neg l_2 \wedge \cdots \wedge \neg l_n$ is a CNF with only unit clauses. In other words, $\neg C$ can be viewed as an assignment to the variables appearing in C. If P simplifies to $\bot$ after making these assignments and applying BCP, then we say that C is a RUP clause for P. The claim above is that clauses learned by the procedure described in Section 2 are always RUP clauses. First, note that because a learned clause C is entailed by the original formula P, $P \wedge \neg C$ is indeed a contradiction. Because C consists of (negations of) literals that were decided on a trace that ultimately unit-propagated to a conflict, negating C and conjoining with P amounts to making the same partial assignment, so BCP should yield the same conflict.

Now recall the example from Section 3.3: it consisted of the derivation of $C_6 = C_2 \bowtie_b C_3$, and the chain $C_4 \bowtie_b C_5 \bowtie_c C_1 \bowtie_a C_6$ which resulted in the empty clause. Observe that the certificate checker does not need to be given the resolution steps to derive C_6, it just needs to establish that C_6 is a consequence of the original formula P, i.e., to check that $P \wedge \neg C_6 \leftrightarrow \bot$. The RUP property means that the checker can accomplish this by applying BCP to $P \wedge \neg C_6$, and checking to ensure that the result is a conflict.

Likewise, notice that the steps listed out in the chain $C_4 \bowtie_b C_5 \bowtie_c C_1 \bowtie_a C_6$ correspond to a sequence of unit propagations stemming from C_6. This chain resulted in the final conflict that the solver encountered before returning `unsat`, so we may think of it as a chain that results in the "learned" clause $\bot$ (which gives the desired refutation). Just as before, the checker does not need to be given this chain, and can instead check that $\bot$ is RUP for $P \wedge C_6$ by applying BCP to $P \wedge C_6 \wedge \neg\bot$, or simply $P \wedge C_6$.

This reasoning gives rise to clausal proofs. A clausal proof for a CNF P is simply a list of clauses $[C_1, C_2, \ldots, C_n]$ where $C_n = \bot$, with the following property:

$$P \wedge C_1 \wedge \cdots \wedge C_{i-1} \wedge \neg C_i \text{ unit propagates to } \bot, \text{ for all } i \in \{1, \ldots, n\} \tag{2}$$

Note that for solvers which implement clause learning, there is no question about how to generate a clausal proof: just output each learned clause generated by the solver, in the order that they are learned.

Likewise, verifying a clausal proof is conceptually straightforward, as the checker relies exclusively on BCP. Most checkers verify the clauses in the reverse order in which they were learned, i.e. $P \wedge C_1 \wedge \cdots \wedge \neg C_n$ is checked before $P \wedge C_1 \wedge \cdots \wedge \neg C_{n-1}$, because there is no point in doing more work if later steps to not give a valid refutation. This gives rise to the term *Reverse* Unit Propagation in Definition 1.

Figure 4 on the right shows an representative encoding of a clausal proof in the RUP format [Gel08]. The format is very simple: each line lists a clause by its literals, terminated by `0`. The same signed integer encoding for literals is used as in the TraceCheck

format discussed earlier. This example certificate is impressively concise: the first line gives the lemma $\neg a$, and the second denotes the empty clause $\bot$.

4.3 Tradeoffs

The relevant tradeoffs for certificate encodings involve the amount of work done by the solver versus the checker, storage considerations, and the degree of complexity involved in checking a certificate. Resolution and clausal proofs clearly represent very different points along these axes.

Resolution proofs are more verbose, and generating them requires more bookkeeping by the solver. The fact that indices of earlier clauses are needed to correctly format later proof steps implies a non-trivial lower bound on the amount of memory that the solver will need to devote to certificate construction, and as the size of the proof grows the amount of work needed to consult this bookkeeping will grow linearly with it. In short, resolution proofs are more costly to construct, and depending on the scale of the formula and its proof, doing so may be prohibitively expensive in some situations. On the other hand, they are simple to verify, as the checker just needs to determine that the result of each resolution chain listed in the certificate gives the expected result.

Clausal proofs are extremely simple for most solvers to generate, as emitting learned clauses requires just a few lines of code and imposes no appreciable time or space overhead. The story is much different for the checker. The most straightforward approach for verifying them is quite costly, as the checker needs to validate Equation 2 for each $i \in \{1, \ldots, n\}$. There are more sophisticated methods and heuristics for checking these proofs more efficiently, and checkers can potentially re-use highly optimized implementations of BCP from state-of-the-art solvers. But the cost is still typically greater than checking resolution proofs, and this additional complexity is not desirable from a correctness standpoint (nor is using the same code to both produce an answer, and check that it is correct).

5 SMT Certificates

In Lecture 19, we learned how SMT solvers extend DPLL by incorporating a conjunctive theory solver, in a procedure referred to as DPLL(T). Recall that at each step, a propositional abstraction of a theory formula P is checked for satisfiability by DPLL. If the abstraction is unsatisfiable, then the procedure returns unsat. Otherwise, a satisfying assignment is used to construct a query to the theory solver, which could either yield a satisfying assignment for P, or a new theory lemma to add to the propositional abstraction.

Summarizing the above, whenever DPLL(T) returns unsat, any certificate that it produces will have a portion that is a refutation for the propositional skeleton. However, this is not sufficient, because the propositional skeleton will have clauses that were not part of the original formula, and correspond to lemmas provided by the theory solver. If the theory solver is able to produce certificates for each theory lemma, then these

```
unit-resolution(
  Not(a == f(f(a, b), b)),
  trans*(a == f(a, b),
         symm(
          monotonicity(
            symm(a == f(a, b),
                 f(a, b) == a),
            f(f(a, b), b) == f(a, b)),
          f(a, b) == f(f(a, b), b)),
         a == f(f(a, b), b)),
  False)
```

```
f 134 (decl a () S) 0
e 5 a 0
f 135 (decl b () S) 0
e 6 b 0
f 133 (decl f (S S) S) 0
e 7 f 5 6 0
e 8 = 7 5 0
b 1 8 0
i 1 0
1 0
e 9 f 7 6 0
e 10 = 5 9 0
b 2 10 0
i -2 0
r euf 2 0
0
```

Figure 5: Two proofs produced by Z3 for the EUF formula $f(a,b) = a \wedge f(f(a,b),b) \neq a$. The left formula is formatted as a proof tree, and the right is a clausal proof. Note that the formula on the left was simplified slightly for presentation by removing redundant rewrite and introduction terms, so is slightly different from the literal output given by Z3.

can be interleaved with the steps of the resolution proof to produce a certificate for the original theory formula.

Two proof certificates generated by Z3 for the EUF formula $f(a,b) = a \wedge f(f(a,b),b) \neq a$ are shown in Figure 5. The first thing to note is that SMT certificates are at an earlier stage of development and adoption than those for SAT, so common encodings between solvers do not exist; in fact, the clausal proof on the right of the figure is the output of an experimental feature in Z3, and it currently cannot produce clausal proofs for most theories.

The certificate on the left likely comports with the idea of a first-order theory proof that comes to mind for many of us. It maintains a tree-like structure, where each node is either a formula (typically coming from an assumption given by the original theory formula), or is labeled with a proof rule: `unit-resolution`, `trans*` (transitivity), `symm` (symmetry), and `monotonicity` (congruence). Notice that the topmost reasoning is resolution, its first antecedent is one of the two clauses in the original formula, and its second antecedent derives from an application of transitivity from EUF, which relates $a = f(a,b)$ to $a = f(f(a,b),b)$. This portion was generated by the theory solver, and captures the reasoning needed to conclude that the negation of the second literal is a consequence of the first.

The certificate on the right is a trace of the same solver run given as a clausal proof. As before, the `0` digits at the end of each line are terminators. The lines starting with `f` relate the definition of "auxiliary" functions and constants a, b, and f. Those starting with `e` define terms and theory literals, so `e 5 a 0` on line 2 defines an expression with

index `5` to correspond to the EUF term `a`; on line 7, `e 8 = 7 5 0` defines index `8` to be the application of the predicate `=` to indices `7` ($f(a,b)$) and `5` (a). Lines beginning with `b` define the propositional abstraction, so `b 1 8 0` defines a propositional literal indexed by `1` to be mapped to the theory literal `8` ($f(a,b) = a$). Lines beginning with `i` give input assertions, so `i 1 0` relates that propositional variable `1`, which is the theory literal $f(a,b) = a$, is given as an assumption. Finally, lines starting with `r` denote theory lemmas: `r euf 2 0` says that literal `2` ($f(f(a,b),b) = a$) is a consequence of the theory axioms and asserted theory literals.

The clausal proof ends with a line consisting only of `0` to denote a refutation. It is the checker's job to work out that the theory lemma can be resolved with the line preceeding it to derive the empty clause. Likewise, whereas the tree-like proof on the left provided detailed steps to establish the antecedents of the topmost resolution step, the clausal proof leaves it to the checker to work out that $f(f(a,b),b) = a$ is a consequence of the asserted $f(a,b) = a$. However, whereas lemmas learned by SAT solvers uniformly satisfy the RUP property, there is no such unified approach to take with theory lemmas, and in general the checker may need to retrace many of the same steps taken by the SMT solver, and implement much of the same functionality. A survey article by Barrett et al. [BDMF15] provides a good overview of these and other challenges that arise in practice when generating SMT certificates.

6 Limitations

Finally, returning to the desiderata listed at the beginning of these notes, an important quality for proof certificates to have is conciseness. If a proof of unsatisfiability takes as long (or longer) for the checker to process, or require gigabytes of storage, then their practical utility may be limited in some cases.

Some formulas do not have polynomial-sized resolution proofs. One such famous example are the so-called "pigeon-hole" formulas that you encountered on Assignment 5: whenever the number n of pigeons in such a formula is greater than the number of holes, then any refutation based on resolution will require an exponential number of steps in n [Hak85]. While problems like this may not arise frequently in most practical applications of SAT or SMT, it is important to keep in mind that there is not a polynomial bound on the size of the certificate that applies to general CNF formulas.

Another source of difficulty stems from the fact that the actions of the solver must be efficiently simulated via resolution in order to produce a concise certificate. Some solvers employ strategies that add clauses to a formula either as a preprocessing step, or during search ("inprocessing"). It must be possible to show that these clauses are entailed by the original formula with a short resolution proof, or the resulting certificate cannot be checked. Some forms of clause addition cannot be simulated with resolution, so to address this solvers may need to turn to richer proof systems than resolution for producing refutation certificates. The survey article by Huele and Biere [HB15] provides a good overview of these techniques.

References

[BDMF15] Clark Barrett, Leonardo De Moura, and Pascal Fontaine. Proofs in satisfiability modulo theories. *All about proofs, Proofs for all*, 55(1), 2015.

[Bie08] Armin Biere. PicoSAT essentials. *Journal of Satisfiability, Boolean Modeling, and Computation*, 4(2-4):75–97, 2008.

[BK95] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1), jan 1995.

[FBL18] Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified sat solver with watched literals using imperative hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018.

[Gel08] Allen Van Gelder. Verifying rup proofs of propositional unsatisfiability. In *ISAIM*, 2008.

[Hak85] Armin Haken. The intractability of resolution. *Theoretical computer science*, 39:297–308, 1985.

[HB15] Marijn JH Heule and Armin Biere. Proofs for satisfiability problems. *All about Proofs, Proofs for all*, 55(1), 2015.

[WK18] Eric Wong and Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning*, 2018.