Comparison of Proof Producing Systems in SMT Solvers

Arjun Viswanathan

Abstract

Satisfiability Modulo Theories (SMT) solvers input typically large formulas that contain both Boolean logic and logic in different theories - such as arithmetic and strings - and decide whether the formulas are satisfiable or unsatisfiable. Verification tools use these solvers to prove system properties. As a result, solver output must be trustable. However, SMT solvers are really complicated tools that have tens of thousands of lines of code. One way to make SMT solver ouput more reliable is to have them produce proofs of their results. For satisfied formulas, this can be a model of satisfaction, that is, values for all the variables in the formula. For unsatisfied formulas, it is a transformation of the formula into a simple contradiction using a small set of inference rules, checkable by an external tool such as a proof checker. This report describes the proof producing mechanisms commonly used in SMT solvers and compares the proof systems of CVC4, VeriT, and Z3, three state-of-the-art SMT solvers.

1 Introduction

Boolean satisfiability, often called the SAT problem, is the problem of satisfying a Boolean formula, that is, consistently assigning values of $True$ or $False$ to the variables of the formula so that the entire formula evaluates to $True$. For example,

$$(x \lor y) \land z$$

can be satisfied by the assignment $\{x = True,\ y = False,\ z = True\}$. On the other hand,

$$x \land \neg x$$

is unsatisfiable, no matter what value is assigned to x.

Satisfiability Modulo Theories [6] or SMT lifts SAT to a level that includes theories. For example,

$$(a = b) \land (b = c) \land \neg(a = c)$$

is a formula that is unsatisfiable in the theory of equality over uninterpreted functions. This is because, by transitivity of $a = b$ and $b = c$, we have $a = c$. SMT allows us to be more expressive with our formulas, but this comes at the cost of more complicated decision procedures.

SMT solvers have plenty of applications in formal methods and software verification. For instance, SMT solvers are used in the back-end of model checkers [2], which input mathematical models of a software system, and verify whether they satisfy a particular property or not. Another area of application is symbolic execution [3], which is to analyze a program to figure out what set of inputs work for each part of the program. Other uses of SMT solvers include program synthesis [8], static analysis, and interpolant generation [13].

Given their rise in popularity and usage in the software verification world, it is really important that we are able to trust the outputs of SMT solvers. SMT solvers are typically very complex systems with tens of thousands of lines of code, likely to contain bugs. Verifying such a large codebase can be a cumbersome, if not impossible task. An alternative is to rely on tools called proof checkers [20], that have a much more trusted kernel containing a small set of axioms and inference rules. These proof checkers sacrifice in automative capability what they gain in terms of trustability over SMT solvers. To exploit these proof checkers, SMT solvers produce proof certificates of their outputs, that can be checked by the proof checker.

In this report, I introduce the workings of an SMT solver and their proof producing capabilities; I also compare the proof systems of CVC4, VeriT, and Z3 - three state-of-the-art SMT solvers - as respectively described in the research papers titled "Lazy Proofs for DPLL(T)-Based SMT Solvers" [16], "Expressiveness + Automation + Soundness : Towards Combining SMT Solvers and Interactive Proof Assistants" [12], and "Proofs and Refutations, and Z3" [9]. I begin by explaining the algorithm that is at the core of most SMT solvers. I then explain the logical basis for proof production in SMT solvers, and then get into comparing the research work as I explain the proof production systems of the SMT solvers.

2 Formal Preliminaries

The following grammar specifies the syntax of the formulas we will use:

$$
\begin{array}{ll}
Variable & \rightarrow x \mid y \mid ... \\
Constant & \rightarrow c_1 \mid c_2 \mid ... \\
Term & \rightarrow Variable \mid Constant \mid f(t_1, ..., t_n) \\
Literal & \rightarrow Term \mid \neg Term \\
Formula & \rightarrow True \mid False \mid p(t_1, ..., t_n) \mid Literal \mid Literal \vee Literal \\
 & \quad \mid Literal \wedge Literal \mid Literal \Rightarrow Literal \mid Literal \iff Literal
\end{array}
$$

where f is a function symbol and p is a predicate symbol.

Functions can be understood as they are in basic mathematics, and predicates are functions to the Boolean type. Negation ($\neg$), disjunction ($\vee$), conjunction ($\wedge$), implication ($\Rightarrow$), and double implication ($\iff$) have semantics as in classical logic [21].

Our setting consists of a background theory T consisting of m theories $T_1, ..., T_m$ with respective many-sorted signatures $\Sigma_1, ..., \Sigma_m$. All signatures share a set of sort (type) symbols, and equality is the only predicate. The theories also share a set of uninterpreted constants which are used for reasoning about terms that belong to multiple theories.

The preceding part of this section formalizes the fact that the framework here is described for a single theory, but that theory can be considered a combination of multiple theories [18]. This means that formulas are composed of Boolean components, each of which will evaluate to $True$ or $False$. However, since we have theories, these components could be just Boolean variables, or they could be (dis)equalities over our multiple theories, which gives us expressivity.

For example, the following formula is over the theory of linear integer arithmetic (LIA) and the theory of equality over uninterpreted functions (EUF) [17].

$$(x \wedge y) \Rightarrow (a = 0 \wedge a + b = c \wedge \neg(f(b) = f(c)))$$

x and y are terms in the formula, and hence can only evaluate to $True$ or $False$. a, b, and c, on the other hand, are integer variables that must evaluate to an integer constant. To generalize this, we assume our theory T to be a combination of m theories $T_1, ..., T_m$. Each theory T_i is formally described by means of a signature Σ_i, but we will stick to intuitive theories such as the ones mentioned before, so we don't have to deal with such formalisms. A formal definition of theories and SMT can be found at [6].

When it is used as a set, ϕ refers to the empty set. We also allow for representation of formulas that are conjunctions, as sets of the corresponding conjuncts. So we represent $c_1 \wedge ... \wedge c_n$ as $\{c_1, ..., c_n\}$.

We need the notion of entailment on two levels. Propositional entailment $p \models_P q$ read as "formula p propositionally entails formula q" says that q is a logical consequence of p. For example, $a \wedge b \models_P a$, where a and b are Boolean variables. Entailment also occurs in a theory $x \models_i y$ read as "formula x i-entails formula y", where i is the concerned theory. For example, $x > 3 \models_{LIA} x > 0$, where x is an integer variable. If we abstracted these formulas to the propositional level, $m = (x > 3)$, and $n = (x > 0)$, the entailment $m \models_P n$ cannot be realized at the propositional level. We need reasoning at the level of the theory of arithmetic for this entailment to hold.

A formula is $satisfiable$ if we can consistently assign values to all its variables (Boolean and theory), so that the formula evaluates to $True$. A formula is $unsatisfiable$ if there is no consistent assignment that we can give its variables so that the formula evaluates to $True$. $(x + y = 0) \wedge (m \vee n)$ is satisfiable, and a satisfying assignment is $\{x = 0,\ y = 0,\ m = True,\ n = False\}$. $p \wedge \neg p$ is unsatisfiable. Two formulas are $equisatisfiable$ if they are both satisfiable, or if they are both unsatisfiable. Two formulas are *equivalent* if every model that satisfies one of them satisfies the other as well. Consider $a \wedge b$ and $a \wedge b \wedge n$. These are equisatisfiable since both of them are satisfiable - $\{a = True,\ b = True\}$ satisfies the first one and $\{a = True,\ b = True,\ n = True\}$ satisfies the second one. However, they are not equivalent since model $\{a = True,\ b = True,\ n = False\}$ satisfies the first one, but not the second one.

A formula is *valid*, if no matter what values we assign its variables, it evaluates to $True$. In other words, it is entailed from nothing. For example, $\models_P p \vee \neg p$, and $\models_{LIA} (x = 0) \vee (x < 0) \vee (x > 0)$.

Finally, the universal quantifier ($\forall$) and the existential quantifier ($\exists$) help us quantify variables in formulas. For example, $\forall x, x > 0 \Rightarrow \neg(x < 0)$ states the obvious fact that positive numbers aren't negative, and $\exists x, y = 2x$ is the predicate that is true if y is even. Our discussion in this report only involves quantifier-free logic of SMT solvers, but these are useful tools to have at the meta-level.

3 The DPLL(T) Algorithm

Davis-Putnam-Logemann-Loveland or DPLL - named after its developers - is an algorithm developed in the 1960s for deciding the satisfiability of propositional formulas. About half a century later, most SAT solvers and SMT solvers are still based on some form of DPLL. All the solvers discussed in this work are DPLL(T) solvers, that is an extension of DPLL to accommodate theories. Section 3.1 introduces the normal form required by these solvers, and Section 3.2 explains the DPLL(T) algorithm as a transition system as presented in [16] and originally in [19].

3.1 CNF Conversion

SMT solvers require that the input formula is converted to a normal form called conjunctive normal form (CNF), before they start solving them. The CNF of a formula represents the formula as a conjunction of disjunctions. A clause is a disjunction (Boolean OR) of variables. So CNF is a conjunction (Boolean AND) of clauses.

Using Tseitin clausification [1], any formula can be converted into an equisatisfiable CNF formula.

The following is an example where a set of rewrite rules are applied to F to convert it into CNF.

F	$a \Rightarrow (b \wedge c)$
Step 1	$a \Rightarrow x_1 \wedge (x_1 \iff (b \wedge c))$
Step 2	$x_2 \wedge (x_2 \iff a \Rightarrow x_1) \wedge (x_1 \iff (b \wedge c))$
Step 3	$x_2 \ \wedge \ (x_2 \Rightarrow (a \Rightarrow x_1) \ \wedge \ (a \Rightarrow x_1) \Rightarrow x_2) \ \wedge$
	$(x_1 \Rightarrow (b \wedge c) \wedge (b \wedge c) \Rightarrow x_1)$
Step 4	$x_2 \ \wedge \ (\neg x_2 \vee \neg a \vee x_1) \ \wedge \ (a \vee x_2) \wedge (\neg x_1 \vee x_2) \ \wedge$
	$(\neg x_1 \vee b) \wedge (\neg x_1 \vee c) \wedge (\neg b \vee \neg c \vee x_1)$

Steps 1 and 2 involve introducing fresh variables x_1 and x_2 for all subterms of F by means of equivalence between the subterm and the fresh variable. Steps 3 and 4 reduce these subterms to CNF by common rewrite rules such as distribution and De Morgan's law.

This example is to illustrate that the conversion can be done. There are many optimizations that are applied in addition, that keep the normal form's size in check.

3.2 Abstract DPLL(T) Framework

At a high level, DPLL(T) takes a formula in CNF, and tries to satisfy every conjunct. There are two levels of reasoning - propositional and theory. The algorithm can be understood as always working in the propositional level (thoery literals are abstracted to propositional ones), trying to satisfy the formula. If the formula is propositionally unsatisfiable, then the algorithm concludes, without any theory reasoning, that the formula is unsatisfiable. However, if the formula is propositionally satisfiable by a model, then the SAT solver sends the satisfying model to the theory solver(s) concerned to check whether it is satisfiable at the thoery level as well. If it is satisfiable in the theory level, then the solver returns to the user that the formula is satisfiable. If the theory solver finds an inconsistency, it sends back information to the propositional solver, restricting the current model, and the solver tries again. If the propositional solver is unable to please all the theory solvers for any model that it finds at the propositional level, then we conclude by exhaustion that the formula is unsatisfiable.

Checking whether the formula is unsatisfiable at the propositional level is nontrivial. It involves incrementally assigning literals by propagation, and when that isn't an option, by guessing. When a guess takes us down a path that results in unsatisfiability, we must backtrack and check the whether the formula is indeed unsatisfiable if we flip the guess.

Section 3.2.1 explains the preliminaries for the system, Sections 3.2.2 and 3.2.3 discuss solving at the SAT level, and Sections 3.2.4 and 3.2.5 discuss solving at the theory level.

3.2.1 Transition System

DPLL(T) solvers can be formalized abstractly as state transition systems defined by a set of transition rules. The states of a system are either $fail$ or $\langle M, F, C \rangle$ where M is the current context, that is, it is the current assignment of literals in the formula; a literal in M is preceded by a $\bullet$ if the literal was a decision, that is, it was guessed. If $M = M_0 \bullet M_1 \bullet ... \bullet M_n$, each M_i is the decision level, and $M^{[i]}$ denotes $M_0 \bullet ... \bullet M_i$.
F is a set of clauses representing some form of the input formula in CNF. C represents the conflict clause, the clause from F that is falsified by the assignment.
Initial state : $\langle \phi, F_0, \phi \rangle$, where F_0 is the input formula converted to CNF.
Final state:

- $fail$, when F_0 is unsatisfiable in T.
- $\langle M, F, \Phi \rangle$ where M is satisfiable in T, F is equisatisfiable with F_0 in T, and $M \models_P F$.

The transition system goes from the initial state from one of the final states by application of the rules in Sections 3.2.2 and 3.2.4 nondeterministically. In practice and for proof production, they are applied with a particular strategy but for completeness, a nondeterministic order works. A rule is applicable when all its premises hold, and it is applied to make the conclusion true.

The following technicalities are necessary for solving of clauses that are a combination of multiple theories. They are only mentioned here for completeness. For a higher level understanding, consider the term $(f(a) = 1 + x)$ that combines the theory of equality over uninterpreted functions (EUF), and that of arithmetic. Since each theory has its own solver, this term that uses functions from both theories, must be purified for each theory solver, and this is done by means of a shared constant. If the term is replaced by the terms $f(a) = s_1$ and $s_1 = 1 + x$, we now have one term that is entirely over EUF, and one over arithmetic ones.

Int_M is the set of all *interface literals* of M: the (dis)equalities between shared constants. *Shared constants* are the set represented by
$\{c \mid constant\ c\ occurs\ in\ Lit_{M|i}\ and\ Lit_{M|j},\ for\ some\ 1 \leq i < j \leq m\}$. $Lit_{M|i}$ consists of the Σ_i- literals of Lit_M.

The paper gives a guarantee of *refutation soundness*: if an execution starting with $\langle \phi, F_i, \phi \rangle$ ends with $fail$, then F_0 is unsatisfiable in T.

3.2.2 Propositional Rules

Figure 1 enumerates the propositional rules. These rules constitute the DPLL(T) algorithm at the propositional level, or just DPLL. They model the behavior of the SAT engine, which treats atoms as Boolean variables. Propagations (Prop) allow us to assign literals that we are forced to assign by the logic. For example, if the input formula is $\neg a \wedge (a \vee b)$, then a must be assigned to *False*, so that the $\neg a$ conjunct evaluates to *True*. As a consequence, of this, b must be *True*

$$\frac{l_1 \vee ... \vee l_n \vee l \in F \quad \neg l_1, ..., \neg l_n \in M \quad l, \neg l \notin M}{M := Ml} \; Prop$$

$$\frac{l \in Lit_F \cup Int_M \quad l, \neg l \notin M}{M := M \bullet l} \; Dec$$

$$\frac{C = \phi \quad l_1 \vee ... \vee l_n \in F \quad \neg l_1, ..., \neg l_n \in M}{C := \{l_1 \vee ... \vee l_n\}} \; Confl$$

$$\frac{C = \{\neg l \vee D\} \quad l_1 \vee ... \vee l_n \vee l \in F \quad \neg l_1, ..., \neg l_n \prec_M l}{C := \{l_1 \vee ... \vee l_n \vee D\}} \; Expl$$

$$\frac{C = \{l_1 \vee ... \vee l_n \vee l\} \quad lev\ \neg l_1, ..., lev\ \neg l_n \leq i < lev\ \neg l}{C := \phi \quad M := M^{[i]} l} \; Backj$$

$$\frac{C \neq \phi}{F := F \cup C} \; Learn$$

$$\frac{C \neq \phi \quad \bullet \notin M}{fail} \; Fail$$

Figure 1: Transition rules at Propositional level

so that the next conjunct is $True$. Both of these assignments are propagations that the logic forces us to assign, given that we are trying to satisfy the formula. We might not always have this luxury; sometimes, we may have to make a guess on an assignment, and decisions (Dec) let us do this. If we are trying to satisfy $(\neg a \vee c) \wedge (\neg b \vee d) \wedge (a \vee b)$, each conjunct is a non-unit formula, so we need to guess the value of one of the literals. Propogations and/or decisions could lead us to a point where our current assignment conflicts with our goal of trying to satisfy the input formula. The conflict rule (Confl) recognizes this. In the previous example, assume we came up with the assignment $\{a = False,\ b = False\}$ by guessing. This assignment satisfies the first two clauses, but falsifies the third, so the third clause $a \vee b$ is recognized as a conflict clause. If we encounter a conflict, and there were no previous decisions made, then we were forced by the logic to arrive at that conflict, so we conclude that the input formula is unsatisfiable. The Fail rule ensures this. If, there are previous decisions at the time of conflict, then the explain rule (Expl) and the backjump rule (Backj) work together to rewind the state to a point where the decision is flipped. Decisions may cause propagations, and once we suspect that a decision was wrongly made, we want to undo the propagations that were caused by it. This is intuitively what the

M	**F**	**C**	**Rule**	**Step**
	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b$	ϕ	Dec	1
$\bullet a$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b$	ϕ	Prop $(a \vee \neg b)$	2
$\bullet a \neg b$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b$	ϕ	Prop $(b \vee c)$	3
$\bullet a \neg b\ c$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b$	ϕ	Confl $(\neg c \vee b)$	4
$\bullet a \neg b\ c$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b$	$\neg c \vee b$	Expl $(b \vee c)$	5
$\bullet a \neg b\ c$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b$	b	Expl $(\neg a \vee \neg b)$	6
$\bullet a \neg b\ c$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b$	$\neg a$	Learn $(\neg a)$	7
$\bullet a \neg b\ c$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b, \neg a$	$\neg a$	Backj $(\neg a)$	8
$\neg a$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b, \neg a$	ϕ	Prop $(a \vee \neg b)$	9
$\neg a \neg b$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b, \neg a$	ϕ	Prop $(b \vee c)$	10
$\neg a \neg b\ c$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b, \neg a$	ϕ	Confl $(\neg c \vee b)$	11
$\neg a \neg b\ c$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b, \neg a$	$\neg c \vee b$	Expl $(b \vee c)$	12
$\neg a \neg b\ c$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b, \neg a$	b	Expl $(a \vee \neg b)$	13
$\neg a \neg b\ c$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b, \neg a$	a	Expl $(\neg a)$	14
$\neg a \neg b\ c$	$a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b, \neg a$	$\perp$	Fail	15

Figure 2: Example for Propositional level DPLL(T)

explain rule (Expl) does. Once this is done, the backjump rule (Backj) is able to flip the decision. A conflict clause always consists of a formula that is entailed by the input formula. Explanations could transform the conflict clause into a clause that we didn't have to begin with, and this could be useful information for future propagations. The Learn rule allows us to add a non-empty conflict clause to the input clauses to be satisfied.

3.2.3 DPLL Example

Consider the following example from [16]. The formula F expressed as a set in CNF is:

$$\{a \vee \neg b, \neg a \vee \neg b, b \vee c, \neg c \vee b\}$$

Figure 2 shows how DPLL solves this formula. Notice that at Step 11, the *Fail* rule can be applied to conclude that the formula is unsatisfiable, and this is what the solver would typically do. However, for producing a proof of unsatisfiability, it is necessary for the conflict clause to be the empty clause $\perp$, and hence the solver explains on the conflict instead (explained in section 4). A few steps later, the solver does conclude that the formula is unsatisfiable, while deriving the most basic conflict, the empty clause.

3.2.4 Theory Rules

The rules in Figure 3 model the interaction between the SAT solver and the theory solvers. These rules maintain the invariant that every conflict clause and learned clause is entailed in T by the initial clause set. The rules are analogous

$$\frac{l \in Lit_F \cup Int_M \quad \models_i l_1 \vee ... \vee l_n \vee l \quad \neg l_1, ..., \neg l_n \in M \quad l, \neg l \notin M}{M := Ml} \; Prop_i$$

$$\frac{C = \phi \quad \models_i l_1 \vee ... \vee l_n \quad \neg l_1, ..., \neg l_n \in M}{C := \{l_1 \vee ... \vee l_n\}} \; Confl_i$$

$$\frac{C = \{\neg l \vee D\} \quad \models_i l_1 \vee ... \vee l_n \vee l \quad \neg\, l_1, ..., \neg\, l_n \prec_M l}{C := \{l_1 \vee ... \vee l_n \vee D\}} \; Expl_i$$

$$\frac{l_1, ..., l_n \in Lit_{M|i} \cup Int_M \cup L_i \quad \models_i \exists \mathbf{x}(l_1[\mathbf{x}] \vee ... \vee l_n[\mathbf{x}])}{F := F \cup \{l_1[\mathbf{c}] \vee ... \vee l_n[\mathbf{c}]\}} \; Learn_i$$

Figure 3: Transition rules at Theory level. $\mathbf{x}$ is a possibly empty tuple of variables, and $\mathbf{c}$ is a tuple of fresh constants from C - the set of shared constants between the sorts - of the same sort as $\mathbf{x}$; L_i is a finite set consisting of literals not present in the original formula F.

to their propositional rules, but have reasoning powers at the level of theories. For example, $x \wedge y$ is true at the propositional level, if both x and y are true. However, if the variables store these arithmetic theory literals:
$x = (a > 3)$ and $y = (a < 0)$, then $x \wedge y$ is unsatisfiable in the theory of arithmetic. So even though the $Conflict$ rule wont recognize this inconsistency, the $Conflict_{LIA}$ rule can recognize this, as long as the arithmetic solver is able to generate this fact as a *theory lemma*, which is a fact that is true in the theory, generated in CNF form by the theory solvers. This example as a lemma would look like this: $\neg(a > 3) \vee \neg(a < 0)$ which would trigger the $Conflict_i$ clause given that x and y above are assigned to true, to satisfy $x \wedge y$. Thus, the $Propagate_i$, $Conflict_i$, $Explain_i$, and $Learn_i$ rule are similar to their propositional versions except that each rule is associated to a theory i and operates on lemmas provided by that theory.

3.2.5 DPLL(T) Example

Consider the following example that consists of terms in the theory of EUF. Since this is the only theory we consider T the to be the combination of $T_1 = EUF$. The input formula for the solver in CNF is:

$$(a = b) \wedge (f(a) = f(b) \vee f(g(a) = h(b)) \wedge (f(a) = c) \wedge (c \neq d) \wedge (f(b) = d)$$

Abstracting the terms and using the set notation, we have:

$$\{1, 2 \vee 3, 4, \neg 5, 6\}$$

M	**F**	**C**	**Rule**	**Step**
	$1, 2 \vee 3, 4, \neg 5, 6$	ϕ	$Prop^+$	1
1 4¬5 6	$1, 2 \vee 3, 4, \neg 5, 6$	ϕ	$Prop_1(\neg 1 \vee 2)$	2
1 4¬5 6 2	$1, 2 \vee 3, 4, \neg 5, 6$	ϕ	$Confl_1(\neg 2 \vee 5)$	3
1 4¬5 6 2	$1, 2 \vee 3, 4, \neg 5, 6$	$\neg 2 \vee 5$	$Expl_1(\neg 1 \vee 2)$	4
1 4¬5 6 2	$1, 2 \vee 3, 4, \neg 5, 6$	$\neg 1 \vee 5$	$Expl(1)$	5
1 4¬5 6 2	$1, 2 \vee 3, 4, \neg 5, 6$	5	$Expl(\neg 5)$	6
1 4¬5 6 2	$1, 2 \vee 3, 4, \neg 5, 6$	$\perp$	$Fail$	7

Figure 4: Example for Propositional level DPLL(T)

DPLL(T)'s execution on the example is shown in Figure 4. The interesting steps here are steps 2 to 4 that use theory lemmas. The clauses $\neg 1 \vee 2$ and $\neg 2 \vee 5$ are theory lemmas. Intuitively, they are saying "if 1 is true, then 2 is true" (by congruence), and "if 2 is true, then 5 is true" (from the information gained from 2,4, and 6), respectively. Essentially, these clauses are valid in the theory of EUF, and the solver conveniently gives us these lemmas to help us solve the problem at the propositional level. As in the propositional example, we could've used the Fail rule after the conflict to conclude that the formula is unsatisfiable (Step 4), but we want to reduce the conflict clause to the empty disjunction $\perp$ since it will help us with our proof.

4 DPLL(T) Proofs

This section describes the general framework used for a DPLL(T) solver to produce proofs for unsatisfiable formulas. A proof for a satisfiable formula is just a model that satisfies that formula, that is an assignment of values to the variables in the formula that make the formula evaluate to $True$. For an unsatisfiable formula, the proof involves transforming formula into a simple contradiction by means of proof rules or inference rules.

An important rule is that of resolution, which is introduced in Section 4.1. Proofs of unsatisfiability are then explained, at the propositional level in Section 4.2, and at the theory level in Section 4.3. Finally, Section 4.4 explains how theory solver produce proofs of theory lemmas.

4.1 Propositional Resolution

Logical calculi operate by means of rules of inferences. A rule of inference consists of a number of premises and a conclusion. The rule of inference specifies a schema for a logical calculus where, if the premises are true, then the conclusions are true. For example,

$$\frac{\phi \Rightarrow \psi \quad \phi}{\psi}\ \textit{modus ponens}$$

$$\dfrac{\dfrac{\dfrac{\neg c \vee b\ (4) \qquad b \vee c\ (5)}{b} \qquad \neg a \vee \neg b\ (6)}{\neg a\ (14)} \qquad \dfrac{\dfrac{\neg c \vee b\ (11) \qquad b \vee c\ (12)}{b} \qquad a \vee \neg b\ (13)}{a}}{\bot}$$

Figure 5: Proof tree for example in Figure 2

Modus ponens is a rule in classical logic that says that "if A implies B is true *and* if A is true, then B is true".

The inference rule that is central to the construction of proofs for SMT solvers is propositional resolution. It is stated as follows.

$$\frac{\phi_1 \vee ... \vee \phi_n \vee \chi \quad \neg\chi \vee \psi_1 \vee ... \vee \psi_m}{\phi_1 \vee ... \vee \phi_n \vee \psi_1 \vee ... \vee \psi_m}\ resolution$$

Assuming the usual interpretations of the conjunction ($\wedge$) and disjunction ($\vee$) in classical logic, the intuition behind the resolution rule is explained using the following instance of the resolution rule.

$$\frac{a \vee \neg b \quad b \vee c}{a \vee c}$$

If the first premise is true, and b is true - that is, $\neg b$ is false, then a must be true. If the second premise is true, and b is false, then c must be true. Now, both premises must be true for the conclusion to be true, and b must be either true or false in classical logic. So, either a must be true, or c must be true, and this is the logical representation of our conclusion.

There exist logical calculi - that is sets of inference rules - that, given a set of clauses, can generate all clauses that are logically entailed by this set. In other words, these calculi are *generatively complete*. Resolution is a powerful rule since it alone constitutes a logical calculus. Even though the resolution calculus isn't generatively complete, it has the useful property that if a set of clauses are unsatisfiable, then the calculus will derive the empty clause $\bot$ from this set of clauses.

4.2 Proofs at the Propositional Level

At the propositional level, the proof system works as follows. Given an execution of an DPLL(T) based SMT solver that ends in the $fail$ state, we can prove that the input formula F_0 is unsatisfiable as follows. We use the resolution calculus mentioned above to build a refutation tree - a tree with the input clauses and the learned clauses at the leaves, that lead - through the application of the resolution rule - to the empty clause $\bot$. The learned clauses aren't technically in the leaf position, since they need to be justified as well. A learned clause is obtained, when an input clause that is conflicted by the context M is modified by the application of the *Explain* rule. In fact, the *Explain* rule resolves the

$$\dfrac{\neg 5\ (6) \qquad \dfrac{\dfrac{\dfrac{EUF\ Proof}{\neg 2 \vee 5\ (3)} \quad \dfrac{EUF\ Proof}{\neg 1 \vee 2\ (4)}}{\neg 1 \vee 5} \qquad 1\ (5)}{5}}{\bot}$$

Figure 6: Proof tree for example in Figure 4

current conflict clause with a clause belonging to F to obtain a new conflict clause. So proofs for learned clauses are also resolution proofs that are built by observing the conflict clauses and the explanations applied.

Figure 5 shows the proof tree for the unsatisfiability of F from the example in Figure 2. The nodes of the tree contain the clause being used followed by the step number from algorithm in parentheses. The left subtree is a proof of $\neg a$. The right subtree is the proof of a which is a learned clause.

Steps 1 to 7 constitute the left subtree of the proof tree that adds $\neg a$ as a learned clause to F (Although $\neg a$ is actually used in Step 14, it is derived in Step 7). Steps 8 to 14 constitute the left subtree of the proof tree. Conflict clauses and explanations represent propositional resolution between clauses and the proof tree has a leaf for every application of the Conflict and the Explain rules - except the $\neg a$ node since it is learned - and the node at the root represents the Fail rule. For the learned clause, the tree could be considered as a split of the current tree at the $\neg a$ node. The left subtree of the new tree would then just consist of $\neg a$ as a leaf, and the proof for $\neg a$ would be a satellite proof tree representing the left sub-tree of the current tree. That is how it is depicted in [16].

4.3 Proofs at the Theory Level

At the propositional level, we saw proof trees as being refutation trees that derived $\bot$ at the root using input clauses and learned clauses at the leaves, with learned clauses supported by satellite resolution proofs. At the theory level, we have clauses called *theory lemmas* that the theory solvers present to us to help us apply constraints from the theory level. Since the theory solvers come up with the lemmas, the proof is also expected to be at the theory level. And thus the theory solver also provides us with the proof for the lemma. In our refutation tree, when we arrive at a leaf with a theory clause, we ask the theory solver for its proof and plug it into the tree.

Figure 6 shows the proof tree for the example from Figure 4. The part here that is different from the propositional case here is the theory lemmas in steps 3 and 4. Notice these aren't leaves in the proof tree. A proof from the theory solver is plugged in to support these lemmas. The parenthesized numbers indicate the relevant step from the algorithm.

4.4 Theory Solver Proofs

So far, section 4 has presented the proof-producing system of SMT solvers as building refutation trees up to the root containing $\bot$, from leaves that are either inputs or learned clauses with supported proofs. In section 4.3 we added to this picture, the idea of proofs from theory solvers. This section talks about these proofs produced by theory solvers which aren't the resolution proofs we have got used to seeing. Instead, these proofs are in natural deduction [23], which is a calculus that is more expressive than the resolution calculus, and can even be generatively complete. The main rule we are concerned with is the *proof by contradiction* rule, which works as follows. If you assume ψ and derive $False$, then $\neg\psi$ is true. Theory solvers use theory specific rules along with the proof by contradiction rule, to prove their lemmas.

Consider the following example from [16]. The EUF proof for the lemma $L = (x \neq y) \vee (z \neq f(y)) \vee (f(x) = z)$ is as follows.

1. Convert the lemma into its negation:
$\neg L : \neg((x \neq y) \vee (z \neq f(y)) \vee (f(x) = z))$
which is equivalent to
$L_1 : (x = y) \wedge (z = f(y)) \wedge (f(x) \neq z)$.

2. Prove that L_1 is false using rules of EUF as necessary.

$$\dfrac{f(x) \neq z \qquad \dfrac{\dfrac{x = y}{f(x) = f(y)}\,\text{Cong.} \qquad \dfrac{z = f(y)}{f(y) = z}\,\text{Symm.}}{f(x) = z}\,\text{Trans.}}{\bot}$$

The leaves of this contain input formulas, or the literals in L_1. The congruence rule (Cong.) gives $f(a) = f(b)$, given $a = b$. The other two rules refer to symmetry (Symm.) and transitivity (Trans.) of equality.

5 Comparison

This section compares the proof systems of 3 state-of-the-art solvers CVC4, Z3, and VeriT as described in [16], [9], and [12] respectively.

Although each paper introduces proof production in an SMT solver, they are from different times, and aimed at different goals. This section begins by summarizing each paper and then comparing the proof production systems by some particular metrics.

5.1 CVC4

The proof production system of the CVC4 SMT solver is described in [16]. The paper introduces the concept of lazy proof production for SMT solvers. In

DPLL(T), the SAT solver reasons about the input formula using feedback from theory solvers in the form of theory lemmas. The proof of unsatisfiability is a refutation proofs that has input clauses, learned clauses, and theory lemmas at its leaves, and derives the empty clause at the root. The SAT solver is able to build the refutation proofs for the input formula; however, these proofs are completed using proofs for the theory lemmas from the theory solver.

One way to produce these proofs would be to have the theory solver produce proofs eagerly every time a theory lemma is generated. However, in [16], the authors found that doing this *lazily* is efficient - that is, all proofs for theory lemmas are generated only after the final refutation tree has been found. Once the tree is generated, the solver asks the respective theory solver for the proofs for each theory lemma to complete the tree.

This means that each theory lemma occurring in the proof gets processed twice: once when they are generated for solving, and again when they need to be proved for the tree. However, by producing proofs lazily, in most cases the solvers save a lot of computation since many lemmas produced during solving do not end up contributing to the final refutation.

[16] discusses lazy proof production for three different theories - equality over uninterpreted functions (EUF), arrays with extensionality (AX), and bitvectors (BV). For EUF, they propose a completely lazy approach. For AX, they do lazy proof productions with some book-keeping. Briefly, arrays are representable as terms in the AX theory - $a[i]$ is the result of reading the value at index i in array a and $a[i] := b$ is the result of writing value b to array a at index i. An axiom that the array solver uses to produce proofs is the extensionality axiom: for any two arrays a and b, if $a \neq b$ then there exists a k such that $a[k] \neq b[k]$. Since this axiom requires that the disequality of two arrays is witnessed by an index (k), the value of k needs to be stored so that when the proof is being produced for this unsatisfiability lazily, the solver can use this information to produce the proof. Finally, proof production for BV, elaborated in [14], is done semi-lazily. Bit-vectors are arrays of binary bits, so they are solved by a technique called $bit - blasting$ where bitvector formulas are converted into an equisatisfiable propositonal formula which is solved by an internal SAT solver. Since redoing the bit-blasting technique can be expensive, the SAT solver eagerly records a trace of the bit-blasting refutations. In contrast, the proof for the bit-blasting process - converting a formula to a propositional one, is done lazily for lemmas that occur in the final proof.

The paper concludes by showing comparisons between lazy and eager proofs for EUF and AX- they extend CVC4 with lazy and eager proof production systems, and check the proofs for a set of standard benchmarks. The evaluations show that for most of the benchmarks, CVC4 produces lazy proofs faster than eager ones.

5.2 Z3

Z3 is an SMT solver developed at Microsoft Research. [9] elaborates on the proof production system of Z3, highlighting their approach of *implicit quotation* and their natural deduction style proofs for theory lemmas, also explained in Section 4.4. Implicit quotation is an implementation detail that saves the Z3 solver computation when it comes to CNF conversion of input formulas of the SMT solvers. As explained in 3.1, a formula that needs to be checked for satisfiability by an SMT solver is first converted to or CNF. This is done by recursively taking the subterms of the input formula, and creating equivalences between the subterms and fresh variables. Since an equivalence is logically the same as a conjunction of inferences, this method fits well into CNF conversion. For example, $a \iff b$ is converted to CNF as follows.

$$\begin{array}{ll} \mathit{Step}\ 1: & a \iff b \\ \mathit{Step}\ 2: & (a \Rightarrow b) \wedge (b \Rightarrow a) \\ \mathit{Step}\ 3: & (\neg a \vee b) \wedge (\neg b \vee a) \end{array}$$

Introducing these fresh variables increases the number of literals in the formula. Z3 treats the subterm itself as a literal. To distinguish the literal representation from the sub-term, they quote it. If a above is a fresh literal, and b is a sub-term representing theh formula $x \wedge y$, instead of introducing the fresh literal a, Z3 stores a quoted version of the formula - $\lceil x \wedge y \rceil$. By doing this, they avoid all the extra clauses that are added to the formula to maintain this equivalence, as shown in the example above. Instead, the name of the literal gives enough information to tell us what formula it is abstracting. They provide an example of using implicit quotation for slack variables during linear integer arithmetic solving.

This paper also briefly explains the proof producing calculus, which is similar to the calculus explained in sections 2 to 4, except that they choose a different representation. Their main rules can be summarized as follows. A rule called $unit_resolution$ handles a general form of resolution. The *hypothesis* rule states a statement that needs to be proved. This is done by the *lemma* rule by a proof by contradiction.

An important characteristic of proofs in Z3 are that the proof producer axiomatizes many theory rewrites. These can be considered holes in proofs that must be proven by the proof checker. As argued in [16], this gives the proof checker work that is more nontrivial than just checking proofs - they actually need to fill in parts of the proofs. [9] argues that this is a trade-off that saves the solver time in producing proofs and that proof checkers are fairly well-equipped to prove these axioms.

5.3 VeriT

[12] describes the proof production sytem of the haRVey SMT solver, which was a precursor to the solver VeriT. This paper is more application-oriented than the other two. As mentioned in Section 1, verification tools can be evaluated using three different metrics - soundness or trustability, automation, and expressiveness. There are classes of tools that score highly on one or two of those

qualities, but not all. [12] combines harVey, a highly automated tool with Isabelle/HOL a *proof assistant*, which is a tool that interactively allows a user to prove theorems in higher-order logic. The user benefits from the expressive higher-order logic and the high trustability of Isabelle/HOL while also benefiting from the automation of the SMT solver. The trade-off of course is the work done to have the SMT solver produce proofs of its work to the proof assistant, before its results are trusted, which has been the theme in all the papers that I chose to study. This work is an improvement on previous work that use the SMT solver as an oracle, without requiring it to produce proofs.

HaRVey provides *proof hints* to Isabelle that helps the proof assistant reconstruct the proofs of the theorems proven by the SMT solver. Proof assistants such as Isabelle offer users functions known as *tactics* that help the user in their proof aspirations. HaRVey benefits from converting its proofs into expressions in *sequent calculus*, which is an alternate notation for formulas in natural deduction, since Isabelle is well-equipped to deal with sequents. Modulo this encoding as sequents, proof production is similar to the methods described in this report.

They extended Isabelle with the *sat* and *satx* tactic that can be called in the proof assistant for proving subgoals of the current goal being proven.

This work also suggests optimization techniques for SMT solving such as operating on *partial models* that are models of a formula where not all literals in the formula are assigned a value. This has the benefit that finding a partial model that is unsatisfiable eliminates multiple full models that the partial model can be extended to. For instance, if a formula contains literals $a = b$ and $f(a) \neq f(b)$ and 1000 other literals, all models in which $a = b$ and $f(a) \neq f(b)$ are unsatisfiable, regardless of the values of the 1000 other literals.

Finally, the paper presents a *congruence closure* algorithm to solve formulas in the theory of equality over uninterpreted functions (EUF). In this theory, we only know the function symbols as they appear in terms, without knowing their full behavior. The algorithm propagates the equalities from the formula using the rules of reflexivity, transitivity, and symmetry along with the congruence rule that gives $f(a) = f(b)$ given that $a = b$. Once a disequality is found between terms found to be equal, the algorithm concludes that the formula is unsatisfiable. Section 4.4 shows a small example.

The implementation interface between Isabelle and haRVey takes proofs for lemmas in the theory of EUF and reconstructs them for the proof assistant. In short, this is how the interface works. When the user is trying to prove something in Isabelle, they may have the subgoal of proving a formula F, that is to show that F is valid. This is the same as checking whether $\neg F$ is unsatisfiable. So the user invokes the *sat* or *satx* tactic from Isabelle which has haRVey check $\neg F$ for unsatisfiability. If $\neg F$ is satisfiable, the model provided by haRVey is given to the user as a counterexample for F. If it is unsatisfiable, then haRVey constructs a proof trace of the unsatisfiability along with proofs of any theory lemmas used, encodes them as sequents, and sends it to the interface at Isabelle, where the proof is reconstructed.

5.4 Differences and Similarities

All three papers discuss the proof production systems of DPLL(T)-based SMT-solvers. Sections 2 to 4 explain the concepts that these systems have in common - the DPLL(T) algorithm and resolution-based proofs. Each paper does take a different approach in explaining this concept. This report borrows the formalizations of these concepts from [16] which seemed to do the best job of elaboration with examples. The following discuss some particular points of comparison between the papers.

Eager vs Lazy Proof Production. CVC4 introduces the idea of lazy proof production in the paper, and argues for their use by presenting evaluations on particular fragments of the solver. The authors also suggest that one approach might work better than the other, depending on the theory solver concerned. Z3 takes an eager approach, in which it skip logging the steps taken by the solver. Instead it produces proofs during conflict resolution.

Fine-Grained vs Coarse-Grained Proofs. CVC4 produces fine-grained proofs in which the tinier details of the proofs are also accounted for. Input formulas are converted to CNF; the CNF clauses, the learned clauses, and the theory lemmas constitute the leaves of the proof tree. While the input clauses are axioms, the learned clauses are entailed by the input formula and this entailment is proven by a separate resolution tree for each clause. Theory clauses are produced by theory solvers which also provide the proofs for these clauses. The advantage of fine-grained proofs is that they are complete, and the proof checker only needs to check them. The disadvantage is that the proofs must be well specified by the solver. There may be more rules, and since none of the truths in the theories are assumed, they must be specified as well. Proofs in Z3 are more coarse-grained in that they have holes in them that the proof assistant must fill in during proof checking/reconstruction. These holes might be obvious to fill for certain theories and prove practical in such cases. VeriT, or haRVey in this case, suggests taking a slightly coarse-grained approach as well, since they produce proof hints to the proof assistant to help it reconstruct the proof. However, this was aimed at a particular application, and papers published by the VeriT group seem to have fine-grained proofs [4].

Rewriting. Formula rewriting is an aspect of SMT solving that wasn't mentioned in this work so far. Apart from conversion to a normal form, theory solvers rewrite simple truths in the formula. For example, $x + 0$ could be rewritten to x. Because of how solvers work to combine multiple theory solvers, these rewrites might complicate things by changing what a formula appears like. In CVC4's lazy solvers, it is even more dangerous because solving and proving are decoupled. So a formula may look one way while solving, and be reduced to a rewritten form before the proof is extracted. The solution for this is to store the information of the rewrite so that the proof producer can prove the formula before it was rewritten, and the formula is then rewritten with some justification provided for the rewrite itself. Note that in the experiments that they conducted, the rewrites were still unimplemented, and thus axiomatized. In Z3, the rewrites seem to be prime candidates for the holes in the proofs -

they are axiomatized and the proof checker is expected to prove them.

Proof Format. Although these papers don't talk about the formats of the solvers' proofs, these solvers have other publications that explain in detail their proof format. CVC4 outputs proofs in the LFSC (Logical Framework with Side Conditions) format [22] that was developed by many of the same authors as [16]. LFSC is based on a simply typed λ-calculus with dependent types. The main idea is that the type theory uses the Curry-Howard isomorphism or the propositionas-as-types/ programs-as-proofs analogy to reduce proof checking to type checking. LFSC takes an input proof term and a signature which declares the constants and functions in the theories, along with the proof rules. In [7], the developers of VeriT proposed a proof format that motivated by SMTLib, a standard syntax for SMT solvers [5], which it currently uses. To the best of my knowledge, Z3 doesn't have a paper describing its proof format, but it does have its own proof format. Even though SMT solvers have been able to converge on a common input format in SMTLib, the same can't be said for a proof format, as explained in [11].

6 Conclusion and Moving Forward

In this report, I have tried to summarize my investigation of the proof production systems of three state-of-the-art SMT solvers - CVC4, Z3, and VeriT. In the process, I have learned how SMT solvers produce refutation proofs for the formulas that they solve as unsatisfiable, by combining the internal SAT solver with its theory solvers. The papers also explain how theory solvers produce their proofs by means of a natural deduction calculus. This report presents these findings after introducing the basic working of an SMT solver. It also compares the the papers of the respective solvers and discusses some points of difference such as granularity of proofs, lazy proof production, and handling of rewrites by the solvers.

The report also mentions the motivation for SMT solvers to produce proofs - these highly automatic tools with thousands of lines of code are ubiquitous in the verification community, and thus would benefit from giving soundness guarantees. One of the papers briefly investigates the connection between an SMT solver and a proof assistant. My motivation is aligned with this kind of work. The Coq proof assistant [15] is a popular proof checker that falls in the category of trustable verification tools. SMTCoq [10] is a Coq plugin that is able to leverage the power of an SMT solver to prove goals in Coq. The support for CVC4 in SMTCoq has been implemented for a few theories by my advisor, Cesare Tinelli's research group at the Computer Science Department at the University of Iowa. My goal is to extend this support to other theories and the literature survey of the proof producing systems of SMT solvers is an important step toward that goal.

References

[1] M. Baaz, U. Egly, and A. Leitsch. Normal form transformations. In A. V. A. Robinson, editor, *Handbook of Automated Reasoning*, pages 275–333. Elsevier, 2001.

[2] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.

[3] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, 2018.

[4] H. Barbosa, J. C. Blanchette, and P. Fontaine. Scalable fine-grained proofs for formula processing. In *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, pages 398–412, 2017.

[5] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[6] C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking.*, pages 305–343. 2018.

[7] F. Besson, P. Fontaine, and L. Théry. A flexible proof format for smt : a proposal . 2011.

[8] J. Bornholt. Program synthesis explained. `https://homes.cs.washington.edu/~bornholt/post/synthesis-explained.html`, 2015. [Online; accessed 28-August-2018].

[9] L. M. de Moura and N. Bjørner. Proofs and refutations, and Z3. In *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008*, 2008.

[10] B. Ekici, A. Mebsout, C. Tinelli, C. Keller, G. Katz, A. Reynolds, and C. W. Barrett. Smtcoq: A plug-in for integrating SMT solvers into coq. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, pages 126–133, 2017.

[11] P. Fontaine, C. Barrett, and L. de Moura. Proofs in satisfiability modulo theories.

[12] P. Fontaine, J. Marion, S. Merz, L. P. Nieto, and A. F. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, pages 167–181, 2006.

[13] A. Fuchs, A. Goel, J. Grundy, S. Krstic, and C. Tinelli. Ground interpolation for the theory of equality. *Logical Methods in Computer Science*, 8(1), 2012.

[14] L. Hadarean, C. W. Barrett, A. Reynolds, C. Tinelli, and M. Deters. Fine grained SMT proofs for the theory of fixed-width bit-vectors. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 340–355, 2015.

[15] INRIA. The coq proof assistant. `https://coq.inria.fr/`, 2018. [Online; accessed 28-August-2018].

[16] G. Katz, C. W. Barrett, C. Tinelli, A. Reynolds, and L. Hadarean. Lazy proofs for dpll(t)-based SMT solvers. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 93–100, 2016.

[17] M. Köhler. The theories of linear arithmetic and equality logic and uninterpreted functions in the context of satisfiablity modulo theories. `https://www.tcs.cs.tu-bs.de/documents/Logics_Seminar_2014/TheorySolvers.pdf`, 2014. [Online; accessed 28-August-2018].

[18] Z. Manna and C. G. Zarba. *Combining Decision Procedures*, chapter 10, pages 381–422. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[19] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *J. ACM*, 53(6):937–977, 2006.

[20] P. Schnider. An introduction to proof assistants. `http://www-oldurls.inf.ethz.ch/personal/fukudak/lect/mssemi/reports/09_rep_PatrickSchnider.pdf`, 2015. [Online; accessed 28-August-2018].

[21] A. N. Sebastian Thrun, Daphne Koller. Propositional logic. `http://intrologic.stanford.edu/notes/chapter_02.html`, 2000. [Online; accessed 28-August-2018].

[22] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.

[23] C. University. Lecture 15: Natural deduction. `http://www.cs.cornell.edu/courses/cs3110/2012fa/supplemental/lectures/lec15-logic-contd/lec15.html`, 2012. [Online; accessed 28-August-2018].