TU WIEN Informatics

Unsatisfiability Proofs in the Yices 2 SMT Solver

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Technische Informatik

eingereicht von

Martina Bertalanic, BSc
Matrikelnummer 12230007

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof.in Dr.in techn. Laura Kovacs, MSc
Mitwirkung: Projektass.in Dipl.-Ing.in Dr.in techn. Daniela Kaufmann, MSc
Univ.Ass. Dipl.-Ing. Thomas Hader, MSc

Wien, 30. April 2025

Martina Bertalanic	Laura Kovacs

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Unsatisfiability Proofs in the Yices 2 SMT Solver

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Computer Engineering

by

Martina Bertalanic, BSc
Registration Number 12230007

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof.in Dr.in techn. Laura Kovacs, MSc
Assistance: Projektass.in Dipl.-Ing.in Dr.in techn. Daniela Kaufmann, MSc
Univ.Ass. Dipl.-Ing. Thomas Hader, MSc

Vienna, April 30, 2025

Martina Bertalanic | Laura Kovacs

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Martina Bertalanic, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 30. April 2025

Martina Bertalanic

Acknowledgements

Kurzfassung

Beweise der Unerfüllbarkeit für SMT-Löser zeigen nicht nur die Unerfüllbarkeit der Eingabe durch einen detaillierten Prozess, sondern auch die Gültigkeit der zugrunde liegenden Beweissuche. Diese Arbeit befasst sich mit den potenziellen Herausforderungen, die bei der Implementierung von Unerfüllbarkeitsbeweisen auftreten können, und stellt eine Pipeline zur Erstellung und Validierung des Beweiskonstruktionsrahmens vor.

Konkret wird der mcSAT-Kalkül innerhalb des Yices 2 SMT-Lösers erweitert, um die Protokollierung von Beweisen zu unterstützen. Die protokollierten Beweise werden weiterverarbeitet und in das Auflösungsbeweisformat konvertiert. Um ihre Korrektheit zu beweisen, werden die Beweise mit einem externen Resolution Proof Checker validiert.

Die Grenzen der gegebenen Implementierung werden diskutiert und als Möglichkeiten für Verbesserungen und weitere Arbeiten vorgestellt. Darüber hinaus werden die Ergebnisse der Implementierung anhand mehrerer Beispiele gezeigt.

Abstract

Proofs of unsatisfiability for satisfiability modulo theory (SMT) solvers, apart from demonstrating input unsatisfiability through a detailed process, also demonstrate the validity of the underlying proof search. This thesis addresses some potential challenges that may arise in implementing unsatisfiability proofs and presents a pipeline for producing and validating the proof construction framework.

Specifically, the mcSAT calculus within the Yices 2 SMT solver is expanded to support proof logging. The logged proofs are further processed and converted into the resolution proof format. To prove their correctness, the proofs are validated using an external resolution proof checker.

The limitations of the given implementation are discussed and presented as opportunities for improvement and further work. In addition, the results of the implementation are shown through several examples.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Proofs of unsatisfiability, apart from providing the user with a step-by-step proof showing why an input formula is unsatisfiable, help ensuring the correctness of provers/solvers. Both SAT and satisfiability modulo theory (SMT) solvers are expansive and complex pieces of software, thus unintentional mistakes in implementation are inevitable. The introduction of proofs that can be verified by an external proof checker helps mitigate implementation errors and bugs.

In terms of proof logging support for different SAT and SMT solvers, one can use the SAT competition [1] and the SMT competition [2] as a reference. The SAT and SMT competitions are held yearly and they rose to prominence as means to showcase different solvers and provers. The SAT competition made unsatisfiability proofs mandatory for all participants in 2013 [3]. This is in stark contrast to the SMT competition, where according to the results in 2023 [4], only three solvers were submitted for the proof exhibition track of the competition. Additionally, while binary DRAT proof output is the widely accepted norm in SAT solving, there is no singular proof format for SMT solvers. Considering the fact that the proof exhibition track was added recently in 2022, the proof logging feature of SMT solvers is still in its early stages.

In theory, support for resolution proofs in SMT solvers can be achieved by tracking resolution steps in the conflict resolution state and combining these clauses with the initial clause set in case of input unsatisfiability [5]. In practice, multiple challenges may arise.

One such challenge is *preprocessing*. SMT solvers often rely on preprocessing to simplify formulas. Changes to the original input can range from simple rewrites to intricate simplifications that drastically change the input [6]. These preprocessing changes must also be reflected in the proof.

Another challenge in producing proofs occurs with solvers utilizing conflict-driven clause learning (CDCL)-based decision procedures where *variable assignments* made at decision level $n+1$ may depend on assignments made at decision level n or below.

This thesis aims to mitigate these challenges and produces unsatisfiability resolution proofs obtained using the mcSAT calculus [7]. This calculus is implemented in the Yices 2 SMT solver [8] and we report on our extensions to Yices 2 and mcSAT.

1.2 Problem Statement

A clause set $\mathcal{C} = \{\mathcal{C}_1, ..., \mathcal{C}_n\}$ is given as input. The mcSAT framework, standing for model-constructing SAT solving [7], tries to establish the satisfiability of the input. In case of input unsatisfiability, the mcSAT algorithm terminates with the unsatisfiable status of $\mathcal{C}$.

We define our problem statement as follows: *given an unsatisfiable clause set $\mathcal{C}$, construct an unsatisfiability proof for the mcSAT framework.* The unsatisfiability proof uses the resolution rule performed on the input clauses and learned clauses added by the matching theory solver or the propagations obtained by the Boolean plugin in the conflict analysis phase of the algorithm.

1.3 Related Work

There are multiple SMT solvers and first-order provers with support for unsatisfiability proofs: SMTInterpol [9], cvc5 [10] and Vampire [11].

SMTInterpol [9] is based on the Davis-Putnam-Logemann-Loveland theory (DPLL(T)) [12] and conflict-driven clause learning (CDCL) framework [13] [14]. The execution of the solver can be divided into three phases: the role of the first phase is to simplify the input formula using rewrite rules. In the second phase, the input is converted into CNF. The third phase runs the DPLL(T) algorithm. In case of input unsatisfiability, an unsatisfiability proof is given. SMTInterpol supports proofs for the combination of uninterpreted functions, linear arithmetic over integer and real numbers without quantifiers, arrays and datatypes [15]. The unsatisfiability proof only uses the resolution rule performed on the input formula and axioms of each supported theory. Proof output is optimized by binding proofs which are often reused to variables using the `let` command [16].

cvc5 [10] utilizes the DPLL(T) framework for satisfiability solving. It supports proofs for linear and non-linear arithmetic, arrays, bit-vectors, datatypes, floating-point arithmetic and uninterpreted functions. Two rules are central to the proof procedure: `assume` and `scope`. The assumption rule opens a new leaf in the proof tree and the scope rule is the inverse of the assumption - it is used to close assumptions in a proof. The resolution rule and theory axioms are used in the proof construction as well [17].

Vampire [11] is an automatic theorem prover for first-order logic. It supports proofs for uninterpreted functions, arithmetic and datatypes and is currently under development to include proof logging support for first-order linear arithmetic [18]. Vampire proves conjectures by performing a proof by refutation, i.e. adding the negation of the conjecture to the existing assumptions and checking if the resulting formula is unsatisfiable, using resolution-style calculi. Therefore, the unsatisfiability proof is actually a proof by refutation. The proof consists of steps called inferences; each inference is obtained using inference rules. Examples of inference rules used by Vampire include resolution, superposition and inequality splitting. A notable feature is the inclusion of statistics about the proof, including runtime, memory usage and termination message.

The listed solvers do not use the mcSAT framework. Currently, there is no known unsatisfiability proof logging technique for the mcSAT framework.

1.4 Outline

The outline for the rest of the thesis is as follows. We display the mcSAT decision procedure, formalize the unsatisfiability proof generation in the form of a proof calculus, implement the calculus in the Yices 2 (abbreviated: Yices) SMT solver [8] and evaluate proof generation.

Chapter 2 presents a summary of the necessary theory required for a sufficient understand of the topics that follow. Chapter 3 provides an overview of the mcSAT decision procedure along with a simple example. Chapter 4 covers the unsatisfiability proof generation calculus and its implementation within the Yices SMT solver. The implementation of the proof logger is then evaluated in Chapter 5 using a small benchmark set, followed by a discussion of the results and opportunities for future work. Chapter 6 concludes the thesis by outlining the construction of the implementation for each phase of proof logging and the subsequent evaluation.

CHAPTER 2

Preliminaries

This section provides a brief overview of the following concepts related to this thesis: first-order logic (FOL) [19] and conjunctive normal form (CNF) [20], satisfiability [21], first-order theories and the SMT problem [22], CDCL reasoning [23] and DPLL(T) calculus [24].

2.1 First-Order Logic, Conjunctive Normal Forms

The syntax of first-order logic can be thought of as an extension of propositional logic. First-order logic uses the following terms to build formulas:

- variables $x, y, z, w, ...$
- constants $a, b, c, d, ...$
- functions $f, g, h, ...$, where an n-ary function f takes n terms as arguments

Definition 1. *An* atom *is* $\top$, $\bot$ *or an* n*-ary predicate applied to* n *terms.*

Definition 2. *A* literal *is an atom or its negation.*

Every literal is a first-order formulas. In general, we have:

Definition 3. *A first-order logic (FOL) formula is a formula constructed using logical connectives* $\neg$, $\wedge$, $\vee$, $\rightarrow$ *or* $\leftrightarrow$ *over subformulas, including subformulas with quantifiers: the existential quantifier* $\exists$ *and the universal quantifier* $\forall$.

When we talk about SAT and SMT solving, it is desirable for formulas to be in conjunctive normal form (CNF). A CNF formula is a conjunction of disjunction of literals:

$$\bigwedge_i \bigvee_j \ell_{i,j} \quad \text{for literals } \ell_{i,j}$$

where each block of disjunctions is called a *clause*. In other words, a CNF formula is a conjunction of clauses, where each clause is a disjunction of literals.

Example 1. Given is a simple CNF formula in propositional logic

$$F = (A \vee \neg B) \wedge (C \vee D).$$

In this formula, $A, \neg B, C$ and D are literals and $(A \vee \neg B)$, $(C \vee D)$ are clauses.

2.2 Satisfiability, First-Order Theories, SMT

A clause is *satisfied* if one or more of its literals are satisfied. A clause is *conflicting* if not all literals are not satisfied a truth assignment. A clause is *unit resolved* if all except one literal are assigned and not satisfied. A clause is *unresolved* otherwise.

Definition 4. *A formula F is* satisfiable *iff there exists an interpretation I such that $I \vDash F$, i.e. F evaluates to* true *under I.*

The above definition applies to both propositional and first-order logic formulas.

Definition 5. *A* first-order theory $\mathcal{T}$ *is defined by the following components.*

1. *Its* signature Σ *is a set of constant, function and predicate symbols.*
2. *Its set of* axioms $\mathcal{A}$ *is a set of closed first-order logic (FOL) formulae in which only constant, function and predicate symbols of Σ appear.*

A Σ-formula is constructed using the syntax defined in first-order logic, such as constants, variables, functions, predicate symbols, logical connectives and quantifiers. The role of the set of axioms $\mathcal{A}$ is to provide meaning to the symbols of Σ. Examples of first-order theories include the theory of equality, theory of integers, theory of arrays, etc.

To define satisfiability modulo theory (SMT), we first need to define a $\mathcal{T}$-interpretation.

Definition 6. *An interpretation I is a* $\mathcal{T}$-interpretation *if every interpretation I that satisfies the axioms $\mathcal{A}$ of first-order theory $\mathcal{T}$,*

$$I \vDash A \text{ for every } A \in \mathcal{A}, \tag{2.1}$$

also satisfies a Σ-formula F: $I \vDash F$.

This means that a valid $\mathcal{T}$-interpretation has to satisfy the axioms of the corresponding first-order theory $\mathcal{T}$.

Definition 7. *A Σ-formula F is* $\mathcal{T}$-satisfiable *or* SMT *if there is a $\mathcal{T}$-interpretation I that satisfies F.*

SMT solving [25], [26], [27], [28] answers satisfiability of a first-order formula F. If a satisfying interpretation I is found, the formula F is satisfiable. Otherwise, the formula is unsatisfiable.

2.3 CDCL

CDCL is a satisfiability solving framework [13], [14]. At its core, CDCL *decides* variable values, *propagates* implications of the decisions and *backtracks* when a conflict is detected. Each decision has a distinct *decision level*, starting from level 1. Propagations implied by a decision do not change the decision level. If the input formula contains unary clauses, the assignments have decision level 0.

A clause under a specific assignment can have one of the four states: satisfied, conflicting, unit resolved or unresolved.

CDCL employs *unit propagation*, where the only literal left unassigned in the clause has to be satisfied in order for the whole clause to be satisfied.
Such clauses have an *antecedent*, which is defined as the unit clause implying the literal. Antecedents do not exist for decision variables or unassigned variables.

CDCL often uses an *implication graph* to easily demonstrate the execution of the algorithm, which is especially useful for representing how a conflict was reached. In short, an implication graph is a directed acyclic graph where the vertices are variables and the edges are antecedents. Based on the definition of the antecedent above, decision variables have degree 0 and variable assignments arising from unit propagation have degree 1 or higher.

Once a conflict has been reached, the procedure enters a state of conflict analysis. In this state, the procedure learns which clauses need to be added to the initial clause set to prevent the conflict from occurring again. New clauses are learned by performing the resolution operation. The conflict analysis state ends by performing a backtrack to a lower decision level.

Definition 8. Decision level δ *of variable* x_i *is the depth of the decision tree at which the variable is assigned a value. The decision level for an unassigned variable is -1,* $\delta(x_i) = -1$.

The described algorithm repeats until either satisfiability is reached or the formula is decided to be unsatisfiable.

Below is an example of the CDCL procedure from Examples 4.2.4 and 4.4.1 in [29].

Example 2. Consider the input formula

$$F = (C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6)$$

denoting $(x_1 \vee x_{31} \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_{21} \vee \neg x_4 \vee \neg x_6) \wedge (x_5 \vee x_6)$.

Since there are no unit clauses, we need to make decisions on variable values. Suppose that $x_{21} = 0@2$, which means that variable x_{21} has been given value 0 at decision level 2. Furthermore, $x_{31} = 0@3$ and $x_1 = 0@5$. Given these decisions, we arrive at a conflict because clause $(x_5 \vee x_6)$ is unsatisfied under the current assignments. This can be

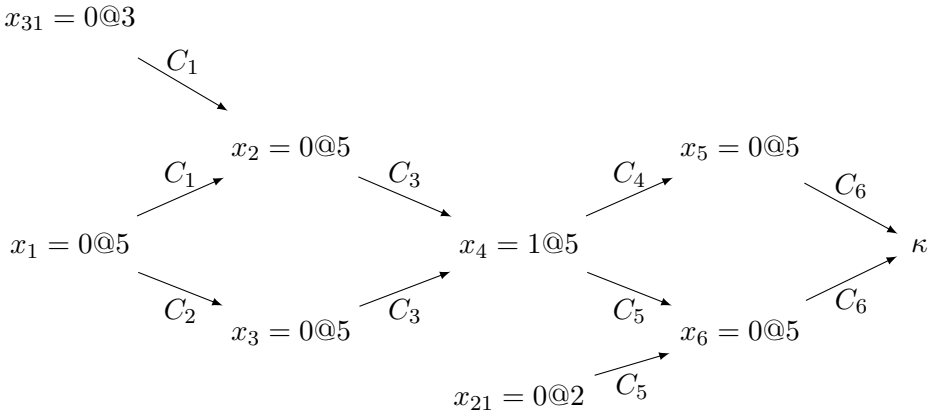

Figure 2.1: Implication graph for example 2

visualized with the implication graph in Figure 2.1 The CDCL procedure then enters into conflict analysis state.
The learnt clause can be obtained in two ways: the first method is achieved by looking at the implication graph and selecting the literals which have been assigned at decision levels less than the current one (in this case less than decision level 5) and at the current decision level.
The second method is to start with the unsatisfied clause and perform binary resolution over the input clause set until there are no more resolution operations left.
In both cases, we obtain the learned clause $x_1 \vee x_{21} \vee x_{31}$. This clause is added to our input clause set, the procedure backtracks to the highest decision level, i.e. level 5 and all decisions at this level are deleted.

Remark. *CDCL has multiple improvements in regard to clause learning, backtracking and use of different data structures, restarts, heuristics and clause deletion [29].*

2.4 DPLL(T)

The Davis-Putnam-Logemann-Loveland DPLL(T) framework [12] is a generalization of the DPLL framework for decidable quantifier-free first-order theories. DPLL(T) combines a SAT solver with a decision procedure for the respective theory T, and can be extended to CDCL(T).

We need to use some form of encoding, as a CDCL-based SAT solver is not applicable for non-propositional formulas F. We proceed as follows: each atom a in the theory signature Σ is associated with a unique Boolean variable $e(a)$ - the *Boolean encoder*. The formula resulting from this encoding is the *propositional skeleton* of F.

Now, the propositional skeleton of F can be passed to the SAT solver. If the SAT solver returns unsatisfiability, then DPLL(T) terminates in an unsatisfiable state. Otherwise, the SAT solver ends in a T-satisfiable state with a model of the Boolean encoders.

The theory solver is then invoked to check if the given model is $\mathcal{T}$-satisfiable, i.e. if the model is satisfiable under the theory's axioms. If the theory solver returns satisfiability, then a satisfying model has been found and DPLL(T) terminates in a satisfiable state. The obtained model is otherwise unsatisfiable. To prevent the unsatisfiable model from being rediscovered in the search, its negation is added to the clause set. This clause is called the *blocking clause*.
After adding the blocking clause, DPLL(T) repeats by invoking the SAT solver until a satisfiable assignment has been found.

Example 3. The formula

$$F : (a = d \vee b = c) \wedge a = b \wedge b = c \wedge c \neq d$$

is given. The propositional skeleton of the formula is

$$F' = (x_1 \vee x_2) \wedge x_3 \wedge x_2 \wedge \neg x_4$$

with Boolean encoders $x_1 = (a = d)$, $x_2 = (b = c)$, $x_3 = (a = b)$ and $x_4 = (c = d)$. The SAT solver returns satisfiable with the following model

$$I = \{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 1, x_4 \mapsto 0\}.$$

The theory solver returns unsatisfiability of clauses corresponding to I. We arrive at a conflict because $a = b$, $b = c$ implies $a = c$, combined with $a = d$ further implies $c = d$, but $c \neq d$ is in the clause set. Therefore a blocking clause

$$\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4$$

is added. The SAT solver is invoked again and returns satisfiability with the following model

$$I = \{x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 1, x_4 \mapsto 0\}.$$

The theory solver now returns satisfiability since there are no conflicts. The satisfying model is then

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

CHAPTER 3

The mcSAT Calculus

Model-constructing satisfiability calculus (mcSAT) is a satisfiability solving framework initially published by Leonardo de Moura and Dejan Jovanović [7] and implemented in the Yices SMT solver. The goal of this thesis is to expand the mcSAT solver of Yices to return a proof in case of unsatisfiability. Therefore background knowledge of the way this calculus operates is required before moving on to the implementation details.

This chapter gives a summary of the mcSAT search procedure, emphasizing the improvements made to the basic DPLL(T) framework. An overview of the rules utilized by mcSAT is included and a proof of correctness is mentioned. The chapter concludes with a practical example to provide a better understanding of the topic.

3.1 mcSAT and DPLL(T)

DPLL(T) is either used directly or provides the basis for numerous decision procedures used by SMT solvers: Z3[30], SMTInterpol[9], cvc5[10] and OpenSMT[31]. The simplicity of the interaction between a SAT solver and a theory solver proved effective as the modularity of the structure allowed adding existing theory solvers to enable the support of a new theory in the SMT solver. This comes at a cost as the theory solver needs to check the plausibility of the solution given by the SAT solver with respect to the theory axioms, thereby increasing the number of function calls to the theory solver [32]. Another drawback of DPLL(T) was that the emergence of the idea of a decision procedure that constructs a model directly and solves conflicts by resolution was not compatible with DPLL(T) due to exclusively Boolean reasoning.

At its core, mcSAT keeps the principal idea of DPLL(T) while setting out to extend its capabilities.

The theory solver in basic DPLL(T) can only provide information on variable propagations specific to the theory and why a specific assignment is satisfiable/unsatisfiable. Such

a function is called the *explanation function* explain. Whereas DPLL(T) limits the explanations of conflicts to be expressed in terms of existing literals, mcSAT gives flexibility to the explanation function so it includes additional literals obtained by the theory solver, thus increasing expressiveness and exploring a wider search space. Another improvement in mcSAT is that it enables the theory solver to assign specific variable assignments independently.

3.2 mcSAT Search Procedure

The mcSAT procedure can be thought of as a transition system. The system has a set of states with the following form

$$\langle M, \mathcal{C} \rangle$$

where M is called a *trail* and $\mathcal{C}$ is a set of clauses.

The trail M serves as the primary data structure in the search procedure [32].

The procedure starts with the initial state $\langle [\![]\!], \mathcal{C}_0 \rangle$. In this state, the trail M is empty and the set of clauses $\mathcal{C}$ is the set of input clauses $\mathcal{C}_0$. The goal of the procedure is to enter either a satisfiable state which returns a model or an unsatisfiable state. In each step, the procedure utilizes specific rules and enters search states $\langle M, \mathcal{C} \rangle$ or conflict resolution states $\langle M, \mathcal{C} \rangle \vdash C$ in case an assignment falsifies a clause from the clause set. These states and rules are discussed in more detail in the next section.

In the initial step, trail M is empty and new elements are added to the trail as a result of the application of the aforementioned rules. Trail elements can be either

- *a decided literal*: the assumption that the literal is assigned as true.
- *a propagated literal*: literal is true because the rest of the literals in clause C are false, denoted by $C \rightarrow L$ and
- *a model assignment*: used in first-order logic to assign a value α to an uninterpreted symbol x. Model assignments are denoted by $x \mapsto \alpha$.

Decided literals and model assignments are both referred to as decisions.
The model assignments on the trail can be used to construct a first-order interpretation

$$v[M] = [x_{i1} \mapsto \alpha_1, \ldots, x_{ik} \mapsto \alpha_k]$$

where i is the clause index, and k is the index of the literal inside clause i.
It is important to differentiate between an interpretation of a literal L according to the Boolean assignment and an interpretation of a literal L according to the first-order model assignment.
Trails have two important properties: *consistency* and *completeness*. A trail M is *consistent* if the Boolean assignment and the first-order interpretation do not contradict

each other. A trail M is *complete* when each first-order literal evaluates to true. The mcSAT search procedure satisfies the conditions that M is a consistent trail and $\mathcal{C}_0 \subseteq \mathcal{C}$ in every state.

3.3 mcSAT Search Rules

As mentioned previously, the transition system changes from state to state by use of different rules. These rules are divided into one of three categories:

- clausal search rules,
- conflict analysis rules and
- theory-specific rules.

Since mcSAT is based on DPLL(T), the clausal rules and the conflict analysis rules are very similar to the DPLL(T) algorithm described in the preliminaries.

3.3.1 Clausal Search Rules

Clausal search rules are the main part of the search procedure. They are shown in Table 3.1.

DECIDE			
$\langle M, \mathcal{C} \rangle$	$\longrightarrow \langle [\![M, L]\!], \mathcal{C} \rangle$	**if**	$L \in \mathbb{B}, \mathsf{value}(L, M) = \mathsf{undef}$
PROPAGATE			
$\langle M, \mathcal{C} \rangle$	$\longrightarrow \langle [\![M, C{\rightarrow}L]\!], \mathcal{C} \rangle$	**if**	$C = (L_1 \vee \cdots \vee L_m \vee L) \in \mathcal{C}$ $\forall i : \mathsf{value}(L_i, M) = \mathsf{false}$ $\mathsf{value}(L, M) = \mathsf{undef}$
CONFLICT			
$\langle M, \mathcal{C} \rangle$	$\longrightarrow \langle M, \mathcal{C} \rangle \vdash C$	**if**	$C \in \mathcal{C}, \mathsf{value}(C) = \mathsf{false}$
SAT			
$\langle M, \mathcal{C} \rangle$	$\longrightarrow \mathsf{sat}$	**if**	$\mathsf{satisfied}(\mathcal{C}, M)$
FORGET			
$\langle M, \mathcal{C} \rangle$	$\longrightarrow \langle M, \mathcal{C} \backslash \{C\} \rangle$	**if**	C $\in \mathcal{C}$ is a learned clause.

Table 3.1: Clausal search rules.

The DECIDE rule takes an unassigned literal L and assigns a value. The PROPAGATE rule finds a unit clause (i.e. a clause where all literals are set to false except one literal

which is undefined) and assigns it the value true. The CONFLICT rule is used to enter conflict resolution and it is triggered when a clause C is not satisfied. The SAT rule is used to terminate the procedure once a satisfying assignment has been found. The FORGET rule removes clauses that have been learned during the conflict resolution phase.

3.3.2 Conflict Analysis Rules

A conflict has been found if a clause C is falsified by the trail. This clause - the *conflicting clause* - is always implied by C_0, i.e. $\mathsf{C}_0 \models C$ The procedure then enters the conflict analysis state. During this process, new clauses are added to the trail and they place additional restrictions on the domain of variable values. Conflict analysis rules are shown in Table 3.2.

RESOLVE			
$\langle [\![M, D{\rightarrow}L]\!], \mathcal{C}\rangle \vdash C$	$\longrightarrow \langle M, \mathcal{C}\rangle \vdash R$	**if**	$\neg L \in C$, $R = \mathsf{resolve}(C, D, L)$
CONSUME			
$\langle [\![M, D{\rightarrow}L]\!], \mathcal{C}\rangle \vdash C$	$\longrightarrow \langle M, \mathcal{C}\rangle \vdash C$	**if**	$\neg L \notin C$
$\langle [\![M, L]\!], \mathcal{C}\rangle \vdash C$	$\longrightarrow \langle M, \mathcal{C}\rangle \vdash C$	**if**	$\neg L \notin C$
BACKJUMP			
$\langle [\![M, N]\!], \mathcal{C}\rangle \vdash C$	$\longrightarrow \langle [\![M, C{\rightarrow}L]\!], \mathcal{C}\rangle$	**if**	$C = L_1 \vee \cdots \vee L_m \vee L$ $\forall i : \mathsf{value}(L_i, M) = \mathsf{false}$ $\mathsf{value}(L, M) = \mathsf{undef}$ N starts with a decision
UNSAT			
$\langle M, \mathcal{C}\rangle \vdash \mathsf{false}$	$\longrightarrow \mathsf{unsat}$		
LEARN			
$\langle M, \mathcal{C}\rangle \vdash C$	$\longrightarrow \langle M, \mathcal{C} \cup \{C\}\rangle \vdash C$	**if**	$C \notin \mathcal{C}$

Table 3.2: Clausal conflict analysis rules.

The RESOLVE rule takes the conflicting clause C and clause D and performs a resolution over clause L. The result of the RESOLVE rule is the clause R. In case the resulting clause R is an empty clause, the input is unsatisfiable. The CONSUME rule is used to ignore literals (either decided or propagated) not relevant for the conflict analysis. This rule is used for simplification of the search space. The BACKJUMP rule exits the conflict analysis phase. In case the input is unsatisfiable, the UNSAT rule is used to terminate. The LEARN rule adds a learned clause C to the set of clauses C.

3.3.3 Theory-specific Rules

Theory-specific rules are the novel part of the mcSAT calculus. The previously presented rules are now extended to allow reasoning about specific theories and are shown in Table 3.3.

T-PROPAGATE			
$\langle M, \mathcal{C} \rangle$	$\longrightarrow \langle [\![M, E \to L]\!], \mathcal{C} \rangle$	**if**	$L \in B, \mathsf{value}(L, M) = \mathsf{undef}$ $\mathsf{infeasible}([\![M, \neg L]\!])$ $E = \mathsf{explain}([\![M, \neg L]\!])$
T-DECIDE			
$\langle M, \mathcal{C} \rangle$	$\longrightarrow \langle [\![M, x \mapsto \alpha]\!], \mathcal{C} \rangle$	**if**	$x \in \mathsf{vars}_T(\mathcal{C})$ $v\,[M]\,(x) = \mathsf{undef}$ $\mathsf{consistent}([\![M, x \mapsto \alpha]\!])$
T-CONFLICT			
$\langle M, \mathcal{C} \rangle$	$\longrightarrow \langle M, \mathcal{C} \rangle \vdash E$	**if**	$\mathsf{infeasible}(M)$ $E = \mathsf{explain}(\mathsf{false}, M)$
T-CONSUME			
$\langle [\![M, x \mapsto \alpha]\!], \mathcal{C} \rangle \vdash C$	$\longrightarrow \langle M, \mathcal{C} \rangle \vdash C$	**if**	$\mathsf{value}(C, M) = \mathsf{false}$
T-BACKJUMP-DECIDE			
$\langle [\![M, x \mapsto \alpha, N]\!], \mathcal{C} \rangle \vdash C$	$\longrightarrow \langle [\![M, L]\!], \mathcal{C} \rangle$	**if**	$C = L_1 \vee \cdots \vee L_m \vee L$ $\exists i : \mathsf{value}(L_i, M) = \mathsf{undef}$ $\mathsf{value}(L, M) = \mathsf{undef}$

Table 3.3: Theory search and conflict rules.

The T-PROPAGATE rule propagates the value of the literal L under the conditions that it has the value undefined and if adding its negation makes the trail unsatisfiable (which is proven using the explanation function). The T-DECIDE rule is used for variable assignment. It has the same conditions as T-PROPAGATE, i.e. the variable value has to be previously undefined and the assignment has to preserve the trail consistency. The T-CONFLICT rule enters the state of conflict resolution. The T-CONSUME rule allows for the removal of variable assignments that do not impact the truth value of the conflicting clause C. The T-BACKJUMP-DECIDE rule is similar to the backjump rule. However, the BACKJUMP rule cannot be applied when clause C has more than one literal L_i that evaluates to undefined once the assignment $x \mapsto \alpha$ that caused the trail inconsistency is removed. The T-BACKJUMP-DECIDE rule allows for the removal of the assignment from the trail if there are more than two literals that evaluate to undefined.

Theorem 1 (Theorem 1 in [7]). *Given a set of clauses $\mathcal{C}$, and assuming a finite basis explanation function* explain*, any derivation starting from the initial state $\langle M, \mathcal{C} \rangle$ will*

terminate either in a state sat*, when* $\mathcal{C}$ *is satisfiable, or in the* unsat *state. In the later case, the set of clauses* $\mathcal{C}$ *is unsatisfiable.*

In order to ensure termination of the procedure, we impose a rule that the literal basis is finite.

3.4 Example of mcSAT Reasoning

For the purpose of better understanding of the presented mcSAT calculus, a simple practical example is given. The following example is from [7].

Consider the set of clauses

$$\mathcal{C} = \{x < 1,\ x < y,\ 1 < z,\ z < x\}.$$

Note that a simplification is used in the notation where $\hookrightarrow L$ denotes that a literal L is implied by the unit clause $L \to L$ when the PROPAGATE rule is used.

$\langle [\![]\!], \mathcal{C}_0 \rangle$
PROPAGATEx4 (propagate all unit clauses)
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x]\!], \mathcal{C} \rangle$
T-DECIDE
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x, x \mapsto 0]\!], \mathcal{C} \rangle$
T-DECIDE
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x, x \mapsto 0, y \mapsto 1]\!], \mathcal{C} \rangle$
T-CONFLICT
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x, x \mapsto 0, y \mapsto 1]\!], \mathcal{C} \rangle \vdash C$
T-CONSUME
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x, x \mapsto 0]\!], \mathcal{C} \rangle \vdash C$
BACKJUMP
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x, C \to 1 < x]\!], \mathcal{C} \rangle$
T-CONFLICT
$\langle [\![\hookrightarrow x < 1, \hookrightarrow x < y, \hookrightarrow 1 < z, \hookrightarrow z < x, C \to 1 < x]\!], \mathcal{C} \rangle \vdash \neg(1 < x) \vee \neg(x < 1)$
RESOLVEx3, CONSUME, RESOLVE, UNSAT
unsat

The algorithm starts with the initial state. All unit clauses are propagated and added to the trail. Using T-DECIDE, we choose $x \mapsto 0$. The mapping $y \mapsto 1$ is obtained by choosing a value for y such that the clause $x < y$ is still satisfied. The conflict arises because the elements on the trail $1 < z$, $z < x$ imply that $1 < x$, but $x \mapsto 0$. The explanation

clause $C \equiv \neg(1 < z) \vee \neg(z < x) \vee 1 < x$ is generated, which we can see is just the result of applying the implication elimination rule on the explanation falsifying the trail $C \equiv (1 < z) \wedge (z < x) \rightarrow (1 < x)$.

In the conflict resolution state, the T-CONSUME rule is used to eliminate the y variable assignment since it does not impact the truth value of the conflicting clause C. BACKJUMP exits conflict resolution by removing the assignment to variable x from the trail and adding $C \rightarrow 1 < x$.

The newly asserted literal $1 < x$ is immediately in conflict with $x < 1$ with the explanation $\neg(1 < x \wedge x < 1) \equiv \neg(1 < x) \vee \neg(x < 1)$. The conflict resolution performs resolution, consumes the redundant clause and terminates with unsat.

CHAPTER 4

Unsatisfiability Proof Generation

This chapter describes the general mcSAT proof production scheme and expands on it by outlining each step in the generation process and the file formats used. A simple example is provided to illustrate the procedure more clearly.

4.1 mcSAT Proof Production Scheme

The fundamental scheme of the proof construction is as follows: the proof needs to result in a final resolution step such that the resolvent is an empty clause. The input set of clauses is unsatisfiable, thereby validating the proof.

The starting point for the proof logging is the conflict clauses, which may consist of either previously learned clauses or newly derived clauses. In the context of the resolution rule, these clauses are either the premise or the resolvent.

The former is a negation of the trail clause. Since a conflict arises only when the trail is inconsistent, some element on the trail must be false to restore consistency.

In the latter case, new clauses do not require additional logging. They may appear in another conflict analysis phase as an already existing clause on the trail. They are handled in the same manner as described above.

An important aspect and the main challenge in the proof construction is distinguishing between clauses obtained from the internal solver and clauses obtained by Boolean propagation. Clauses derived by the internal solver are considered correct and do not need further processing. However, clauses obtained by Boolean propagation cannot be treated in the same manner and it is necessary to exhibit the reasoning behind the propagation. In other words, the proof logger needs to justify the final clause by track all the clauses contributing to it.

4.2 Pipeline of Proof Logging

The unsatisfiability proof generation is comprised of the following steps of our proof logger:

1. Yices receives an input file in SMT-LIB format
2. Yices tracks the relevant clauses during conflict analysis
3. A Python script converts the Yices output into a resolution proof
4. The proof is verified using a resolution proof checker TraceCheck [33].

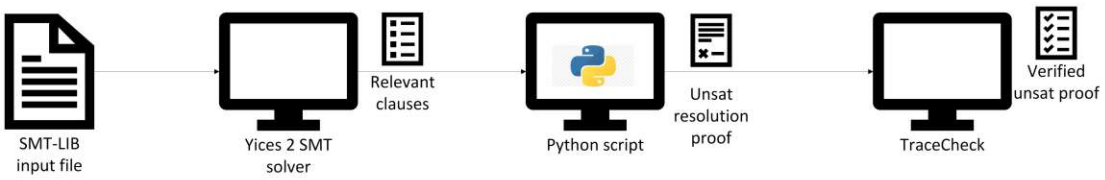

Figure 4.1: Diagram showing the unsatisfiability proof generation pipeline

The first step in the unsatisfiability proof generation involves passing the input to the Yices SMT solver, formatted in SMT-LIB 2.x [34].
The solver attempts to determine satisfiability of the input using the mcSAT rules described in Chapter 3. During the solving process, the conflict clause is tracked, along with trail clauses appearing in the conflict clause and the resolution step. In addition, the output contains text which helps with format conversion in the next step.

Since the output needs additional processing to convert it into a resolution proof format, a Python script is employed. The script removes redundant information, such as text denoting the beginning and end of the conflict analysis phase, and adds the line numbers for the resolution proof.

The script produces a valid resolution proof, which can then be verified. For the purpose of this thesis, TraceCheck was chosen as the proof verification tool.

Given the complexity of proof production, a simple example is used throughout this chapter. The input consists of an unsatisfiable formula consisting of three linear equations in linear integer arithmetic (LIA):

$$\begin{aligned} -2x - y &= 1 \\ 3x - y &= 1 \\ x - y &= -3. \end{aligned}$$

Converted into the SMT-LIB input format, the example input has the following form

```
1 (set-logic QF_LIA)
2 (declare-fun x () Int)
3 (declare-fun y () Int)
4 (assert (= (- (* 2 (- x)) y) 1))
5 (assert (= (- (* 3 x) y) 1))
6 (assert (= (- x y) (- 1)))
7 (check-sat)
```

4.3 Yices Clause Tracking

During the search process, new clauses are added and removed from the trail. Simply outputting the contents on the trail once the solver establishes unsatisfiability is insufficient. Instead, the proof logger must track the clauses on the trail whenever the solver enters a conflict analysis phase. The structure of the output is as follows:

1. Beginning of conflict analysis
2. SMT-LIB output of the conflict clause
3. Propagated clauses
4. Beginning of output relevant for the resolution proof
   a) Conflict clause
   b) Clauses obtained by propagation (optional)
   c) Clauses from the conflict already on the trail
   d) Resolution step
5. End of conflict analysis

The first three steps of output for the first conflict clause will have the following form

```
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ -2 (* -3 x)) 0) v
(/= (+ -1 (* -2 x) (* -1 y)) 0) v
(/= (+ 1 x (* -1 y)) 0)
Conflict analysis phase finished, building output
```

The next step involves tracking the elements required for resolution.
In the first case, the conflict clause is returned by Yices' internal theory solver. The

details on the implementation of the theory solver is out of scope of this thesis. Therefore, it is assumed that the clauses returned by the theory solver are valid and sound and no additional tracking is needed. The only element that is required for logging the resolution proof step are the elements from the trail used to obtain the conflict clause. A check is performed if each clause in the conflict has a negation on the trail. The clauses satisfying this condition are tracked so they can be used in the resolution step.
For the given example, the output for the fourth and fifth step would be

```
Output for conflict clause:
6 -1 -5 0 0
Output for trail input:
1 0 0
5 0 0
Output for resolution:
6 0 1 5 0
Conflict analysis phase finished
```

The literal and clause numbers in the output are derived with the help of Yices' internal variable database, which stores the SMT-LIB literals and clauses and their corresponding indices.
On the above example, number 6 denotes the clause `(>= (+ -2 (* -3 x)) 0)`, number -5 corresponds to `(/= (+ -1 (* -2 x) (* -1 y)) 0)` and -1 represents the clause `(/= (+ 1 x (* -1 y)) 0)`.
Otherwise, the conflict clause is a result of Boolean propagation. In this case, the conflict clause will we written down as the resolvent of the resolution rule.
To illustrate this more clearly, a simple example conflict clause in propositional logic is chosen

$$B \vee X \vee Y.$$

In the variable database, this clause is stored as number 9. Logging the steps performed by the plugin in charge of Boolean reasoning, the following clauses are returned

$$\begin{aligned} \neg C \\ \neg A \vee X \vee Y \\ A \vee B \vee C \end{aligned}$$

In the variable database, $\neg C$ is represented by number 1, $\neg A \vee X \vee Y$ is represented by number 2 and $A \vee B \vee C$ is represented by number 7. Therefore, the output from Yices

will be

```
Output for trail input:
1 0 0
2 0 0
7 0 0
Output for resolution:
9 0 1 2 7 0
Conflict analysis phase finished
```

4.4 Proof Conversion Script

The Python script converts Yices' output into a valid resolution proof format. The output given by Yices is similar to the resolution proof format, however the resolution proof format requires line numbers of clauses when referencing them in the resolution steps (marked in bold).
For the example Yices output

```
Output for conflict clause:
6 -1 -5 0 0
Output for trail input:
1 0 0
5 0 0
Output for resolution:
6 0 1 5 0
```

the resolution proof format converted by the script will be the following

$$\begin{array}{llllll} \mathbf{1} & 1 & 0 & 0 & & \\ \mathbf{2} & 5 & 0 & 0 & & \\ \mathbf{3} & 6 & -1 & -5 & 0 & 0 \\ \mathbf{4} & 6 & 0 & 3 & 1 & 2 & 0 \end{array}$$

where lines 1 and 2 represent the clauses from the trail, line 3 represents the conflict clause and line 4 represents the resolution step. The proof logging process repeats for each conflict analysis phase until it reaches the `unsat` output.

If a clause derived in one conflict analysis phase appears in a later conflict clause, the correct line numbers need to be tracked as well. Continuing with the example, the conflict clause

$$x \geq 0 \quad \vee \quad -2x - y \neq -1 \quad \vee \quad -1 + 3x - y \neq 0$$

is obtained. Performing the translation of clauses into their respectable indices using the variable database once again, with 7 representing $x \geq 0$, number -1 denoting $-2x - y \neq 1$ and number -4 being represented by $-1 + 3x - y \neq 0$. This results in the logging of

$$\begin{array}{cccccccc} \mathbf{5} & 4 & 0 & 0 & & & & \\ \mathbf{6} & 7 & -1 & -4 & 0 & 0 & & \\ \mathbf{7} & 6 & 0 & 6 & 1 & 5 & 0. \end{array}$$

The final conflict clause is

$$-2 - 3x < 0 \quad \vee \quad x < 0$$

and the corresponding logging step is

$$\begin{array}{cccccccc} \mathbf{8} & -6 & -7 & 0 & 0 & & & \\ \mathbf{9} & 0 & 0 & 6 & 7 & 8 & 0 & \end{array}$$

since -6 denotes $-2 - 3x < 0$ and -7 represents $x < 0$.
The proof terminates with the empty clause, which is represented in line 9 starting with 0. Since the proof was able to obtain the empty clause, it is a proof of unsatisfiability.

In order to execute the Python script for format conversion to resolution proofs, the command

```
python processProof.py input.txt output
```

is used, where `input` represents the input file name in .txt format, and output represents the output file name.

4.5 Proof Verification

The final step in proof logging is proof verification. For this purpose, TraceCheck was used. TraceCheck is a verification tool for resolution proofs. It processes a resolution proof and checks the validity of the resolution steps. The proof is verified using the command

```
./tracecheck input
```

where input represents the input file name. For the given example in this chapter, the corresponding TraceCheck output is

```
resolved 1 root and 1 empty clause
```

meaning that an empty clause was obtained during the resolution proof; therefore the LIA input formula is unsatisfiable.

4.6 Limits on Implementation and Future Challenges

The current implementation has two types of limits on the input complexity which can be addressed in future work.

The first limitation concerns the input formulas. Currently, the proof logging procedure supports clauses in the input formula as long as the whole clause does not appear in the conflict analysis phase. For the input example

$$-2x - y = 1 \wedge 3x - y = 1 \wedge (x - y = -3 \vee x - y = -1)$$

if the solver encounters the conflict clause

$$-2x - y \neq 1 \vee x - y \neq -3$$

the proof logging will be correct. However if the solver encounters the conflict clause

$$x - y \neq 3 \vee x - y \neq -1$$

the proof will not be logged or processed correctly.
The internal variable database in Yices does log the index of the clauses, so support for clauses can be implemented fully by further modifying Yices to return the index of the clause from the variable database instead of the underlying literals.

The second limitation is the tracking of Boolean propagations mentioned previously. Basic logging of conflict clauses obtained by Boolean propagation in SMT-LIB format is implemented. However, to obtain full support for these types of input, the output for clauses from the conflict already on the trail and the resolution step would need to be modified accordingly.

CHAPTER 5

Evaluation

In this section, the proof logging and validation is demonstrated.
Given the limitations of the current implementation, the use of pre-existing benchmarks wasn't possible due to their complexity. Therefore, custom examples were developed.
Even though the scope of input problems is limited, the examples provided in this section prove that a proof logging procedure is possible.
First, the choice of benchmark examples is discussed, including the theories covered and the complexity. It is followed by a description of the evaluation setup, covering the specification of the machine the evaluation was executed on and the degree of automation in the testing process.
The chapter is concluded by an experimental analysis, highlighting the results and briefly examining the encountered challenges, the effectiveness of the approach and potential rooms for improvement.

5.1 Benchmark Selection

Initially, the intention was to use the SMT-LIB benchmarks [35] for proof logging validation. These benchmarks include complex examples and cover a wide range of theories, making them the de-facto standard for verifying solver correctness and efficiency. However, as discussed in the conclusion of the previous chapter, the current implementation of proof logging within Yices is limited in terms of possible inputs. Therefore, custom examples needed to be generated in order to validate the functionality of the proof generation pipeline.

Given the early stage of the implementation, our benchmarks were focused on the linear integer arithmetic (LIA). These theories were selected as they are sufficiently expressive to demonstrate the feasibility of generating unsatisfiability proof within the current framework. The benchmarks consist of 5 examples all of which are in the LIA theory.

These examples consist of a system of linear equations with constraints on the domain of each variable.

Since the purpose of this thesis is to develop a proof generation pipeline for unsatisfiable input formulas, the evaluation examples consist exclusively of unsatisfiable inputs.

The size of the clauses differ between different examples, with the clause size being within the range of 6-15 clauses. Each example contains a clause with at most 2 literals. This was a deliberate choice, in order to avoid the possibility of whole clauses with all literals appearing in the conflict clause and therefore being unsupported by the implementation.

The set of benchmark examples is summarized in the table below.

Formula	Sat/unsat	Theory	No. clauses	Maximum no. literals	Correct?
$x + y = 10 \vee z > 5$ $x - y = 2 \vee w < 3$ $2x > z + w \vee y = 4$ $z + w = 7 \vee x > 15$ $x \geq 0$ $x \leq 10$ $y \geq 0$ $y \leq 10$ $z \geq 0$ $z \leq 10$ $w \geq 0$ $w \leq 10$ $x + z = 12$ $w - z = 4$ $2y > x + w$	unsat	LIA	15	2	yes
$x + y = 8$ $2x = z$ $3y = z$ $x \geq 0 \vee x \leq 21$ $y \geq -20 \vee y \leq 100$ $z \geq 0 \vee z \leq 15$	unsat	LIA	6	2	yes
$x + y = 5$ $y + z = 10$ $2x \geq z$ $17y \leq z$ $x \leq 2$ $x \geq 0$ $y \geq 0 \vee y \leq 10$ $z \geq 0 \vee z \leq 10$	unsat	LIA	8	2	yes

Formula	Sat/unsat	Theory	No. clauses	Maximum no. literals	Correct?
$x + y = 10 \vee z > 5$ $x - y = 2 \vee w < 3$ $2x > z + w \vee y = 4$ $z + w = 7 \vee x > 15$ $x \geq 0$ $x \leq 10$ $y \geq 0$ $y \leq 10$ $z \geq 0$ $z \leq 10$ $w \geq 0$ $w \leq 10$ $x + z = 12$ $w - z = 4$ $2y > x + w$	unsat	LIA	15	2	yes
$x + y = 5$ $x - y = 2$ $y + z = 3$ $\neg(x = 12)$ $\neg(y = -23)$ $\neg(z = 100)$ $x \geq -3 \vee x \leq 4$ $y \geq 1 \vee y \leq 14$ $z \geq 7 \vee z \leq 42$	unsat	LIA	9	2	yes
$x + y = 5$ $y + z = 10$ $x + z = 12$ $\neg(x - y = 13)$ $\neg(x + y = 2)$ $\neg(z + y = 8)$ $x \geq -12 \vee x \leq 5$ $y \geq 0 \vee y \leq 5$ $z \geq 0 \vee z \leq 73$ $y < 3 \vee x > 12$	unsat	LIA	10	2	yes

In addition, examples represented in SMT-LIB format and the outputs during each stage of the proof generation pipeline are included in the Appendix: Benchmark Evaluation Output.

5.2 Evaluation Setup

The evaluation was performed on a machine with an Intel Core i7-13620H CPU with 16 GB RAM running Windows 11 and Linux.

The evaluation was not fully automated. Due to the relatively small size of the benchmark set, automation through scripting (i.e. bash script) was considered unnecessary.

Yices and TraceCheck were installed on Ubuntu version 20.04.6 LTS via Windows Subsystem for Linux (WSL), while the Python script was executed within the native Windows environment.

For each example, Yices produced an output in .txt format, which was manually transferred to the directory containing the Python script. The Python script then returned the resolution proofs, which were copied to the directory of the TraceCheck installation. The final verification output from TraceCheck was displayed in the console.

Manual guidance was primarily required to perform the migration of output files between operating systems and to initiate the execution of tools with the relevant input file names.

Overall, all benchmarks were executed successfully, with TraceCheck obtaining the empty clause for each example.

5.3 Experimental Analysis

The experimental analysis aimed to evaluate the feasibility of the proof generation pipeline and to assess its current limitations. The primary objective was to verify that unsatisfiability proofs could be successfully generated for the selected examples and validated externally using TraceCheck.

The incremental logging during conflict analysis in Yices was consistent with expectations. The Python script correctly processed the output into a valid resolution proof format suitable for input to TraceCheck, demonstrating that the essential components of the pipeline that were developed for this thesis operated as intended.

The only segment that proved a challenge was the design of the input examples. The examples needed to be complex enough so that they aren't immediately recognized as unsatisfiable by Yices' preprocessing, while also not overly complex so that they aren't outside of the current implementation capabilities.

In summary, the experiment demonstrated that proof generation and external verification are feasible. The core functionality is operational, however further work is required.

Addressing the observed gaps in the implementation. The primary goal of future implementation work should be to expand the support for clauses and tracking Boolean propagations. This would represent the first step toward full support of higher complexity input formulas.

Benchmarking with other theories. In addition to LIA, mcSAT in Yices also support quantifier-free real arithmetic, as well as uninterpreted functions and bitvectors. Designing and benchmarking example in these theories could either provide proof of robustness of the implemented proof generation or show further gaps in the implementation.

Full automation. As the implementation continues to develop and support examples of higher complexity, full automation will become increasingly necessary. Manual handling of output files, proof processing and verification was sufficient for evaluation with a small benchmark set, however it will not scale efficiently to larger or more intricate benchmarks. Automating these stages would not only streamline the process, but also reduce the risk of human error that can occur between each of the steps. In addition, the reproducibility and reliability of the pipeline would be increased.

Preforming the resolution proof generation within Yices. In the early stages of thesis work, the plan was for Yices to directly output the resolution proof, thus eliminating the need for a Python script and reducing the required tools to Yices and TraceCheck as tools. Such an approach would have significantly improved the overall efficiency, since Python tends to be one of the slowest programming languages compared to lower-level languages. Yices is implemented in C, thus a direct integration of proof generation would have resulted in substantial performance gains, especially when scaling to larger and more complex benchmarks.
However, this strategy would have required a significantly higher level of expertise in development in C, given the language's limitations for advanced string and array manipulations compared to Python. As a result, future implementation efforts need to consider the balance between development complexity and immediate functionality.

Regardless of the specific direction of future work - whether it involves one of the observed opportunities for improvement or something new - it has been observed that an incremental development approach yields the best results.
In particular, designing a simple benchmark, making the necessary changes to the pipeline in order to support the designed benchmark and iteratively testing and refining based on the output has proven to be highly effective. This method helped with early detection of issues in the implementation and their revisions, in addition to creating a manageable progression toward more complex examples.

CHAPTER 6

Conclusion

This thesis introduced an expansion for unsatisfiability proof logging in the mcSAT implementation within the Yices SMT solver.

Following a review of the necessary theoretical background and a brief overview of the mcSAT calculus, the design of the proof generation pipeline was introduced.

The proof pipeline started with the incremental logging of clauses during conflict analysis within Yices. Utilizing the internal variable database, a proof format resembling the resolution proof format was constructed. Additionally, the output included a textual description of the steps and clauses, providing capabilities for incremental analysis of the proof process.

Subsequently, a Python script was developed to process the Yices output, removing the redundant text and extracting the relevant segments of the output into a valid resolution proof format suitable for external proof verification.

Finally, the resulting proofs were successfully verified using the TraceCheck resolution proof checker.

The limitations of the current implementation were discussed, particularly regarding completeness. An experimental evaluation on a set of custom examples was conducted to demonstrate the validity. Finally, the completed work was assessed, highlighting both the challenges encountered and the potential avenues for further refinement of the approach.

Appendix: Benchmark Evaluation Output

This chapter shows the output from each stage of the pipeline for each example from the benchmark set. First the input formula is shown in SMT-LIB format. Then, the output from Yices is shown, followed by the output of the Python script. Finally, the output from TraceCheck is displayed.

Example 1

The following input formula is used

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(declare-fun w () Int)

(assert (or (= (+ x y) 10) (> z 5)))
(assert (or (= (- x y) 2) (< w 3)))
(assert (or (> (* 2 x) (+ z w)) (= y 4)))
(assert (or (= (+ z w) 7) (> x 15)))

(assert (and (>= x 0) (<= x 10)))
(assert (and (>= y 0) (<= y 10)))
(assert (and (>= z 0) (<= z 10)))
(assert (and (>= w 0) (<= w 10)))

(assert (= (+ x z) 12))
(assert (= (- w z) 4))
(assert (> (* 2 y) (+ x w)))

(check-sat)
```

The Yices output is

```
Entered conflict analysis phase
Printing output for conflict clause:
( >= (+ 6 (* -1 z)) 0 ) v
( < (+ 10 (* -1 w)) 0 ) v
( /= (+ -4 (* -1 z) w) 0 )
Conflict analysis phase finished, building output
Output for conflict clause:
32 -28 -30 0 0
Output for trail input:
28 0 0
30 0 0
Output for resolution:
32 0 28 30 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
( >= (+ 3 (* -2 z)) 0 ) v
( /= (+ -7 z w) 0 ) v
( /= (+ -4 (* -1 z) w) 0 )
Conflict analysis phase finished, building output
Output for conflict clause:
33 -15 -30 0 0
Output for trail input:
15 0 0
30 0 0
Output for resolution:
33 0 15 30 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
( >= (+ -2 z) 0 ) v
( < (+ 10 (* -1 x)) 0 ) v
( /= (+ -12 x z) 0 )
Conflict analysis phase finished, building output
Output for conflict clause:
34 -19 -29 0 0
Output for trail input:
19 0 0
29 0 0
Output for resolution:
34 0 19 29 0
```

```
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
( < (+ 3 (* -2 z)) 0 ) v
( < (+ -2 z) 0 )
Conflict analysis at base level, unsat
Output for conflict clause:
-33 -34 0 0
Output for resolution:
0 33 34 0
Conflict analysis phase finished
unsat
```

The Python script output is

```
1 28 0 0
2 30 0 0
3 15 0 0
4 19 0 0
5 29 0 0
6 32 -28 -30 0 0
7 33 -15 -30 0 0
8 34 -19 -29 0 0
9 -33 -34 0 0
10 32 0 6 1 2 0
11 33 0 7 3 2 0
12 34 0 8 4 5 0
13 0 9 11 12 0
```

The TraceCheck output returns the empty clause with the following message

```
resolved 2 roots and 1 empty clause
```

Example 2

The following input formula is used

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)

(assert (= (+ x y) 8))
```

```
(assert (= (* 2 x) z))
(assert (= (* 3 y) z))
(assert (or (>= x 0) (<= x 21)))
(assert (or (>= y (- 20)) (<= y 100)))
(assert (or (>= z 0) (<= z 15)))

(check-sat)
```

The Yices output is

```
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ (* 2 x) (* -3 y)) 0) v
(/= (+ (* 2 x) (* -1 z)) 0) v
(/= (+ (* 3 y) (* -1 z)) 0)
Conflict analysis phase finished, building output
Output for conflict clause:
16 -4 -6 0 0
Output for trail input:
4 0 0
6 0 0
Output for resolution:
16 0 4 6 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ -24 (* 5 x)) 0) v
(/= (+ -8 x y) 0) v
(< (+ (* 2 x) (* -3 y)) 0)
Conflict analysis phase finished, building output
Output for conflict clause:
17 -1 -16 0 0
Output for trail input:
1 0 0
Output for resolution:
17 0 1 16 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ (* -2 x) (* 3 y)) 0) v
(/= (+ (* 2 x) (* -1 z)) 0) v
(/= (+ (* 3 y) (* -1 z)) 0)
Conflict analysis phase finished, building output
```

```
Output for conflict clause:
18 -4 -6 0 0
Output for trail input:
4 0 0
6 0 0
Output for resolution:
18 0 4 6 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ 24 (* -5 x)) 0) v
(/= (+ -8 x y) 0) v
(< (+ (* -2 x) (* 3 y)) 0)
Conflict analysis phase finished, building output
Output for conflict clause:
19 -1 -18 0 0
Output for trail input:
1 0 0
Output for resolution:
19 0 1 18 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
(< (+ -24 (* 5 x)) 0) v
(< (+ 24 (* -5 x)) 0)
Conflict analysis at base level, unsat
Output for conflict clause:
-17 -19 0 0
Output for trail input:
Output for resolution:
0 17 19 0
Conflict analysis phase finished
unsat
```

The Python script output is

```
1 4 0 0
2 6 0 0
3 1 0 0
4 16 -4 -6 0 0
5 17 -1 -16 0 0
6 18 -4 -6 0 0
7 19 -1 -18 0 0
```

```
8 -17 -19 0 0
9 16 0 4 1 2 0
10 17 0 5 3 9 0
11 18 0 6 1 2 0
12 19 0 7 3 11 0
13 0 8 10 12 0
```

The TraceCheck output returns the empty clause with the following message

```
resolved 1 root and 1 empty clause
```

Example 3

The following input formula is used

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)

(assert (= (+ x y) 5))
(assert (= (+ y z) 10))
(assert (>= (* 2 x) z))
(assert (<= (* 17 y) z))
(assert (<= x 2))
(assert (>= x 0))
(assert (or (>= y 0) (<= y 10)))
(assert (or (>= z 0) (<= z 10)))

(check-sat)
```

The Yices output is

```
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ -3 y) 0) v
(< (+ 2 (* -1 x)) 0) v
(/= (+ -5 x y) 0)
Conflict analysis phase finished, building output
Output for conflict clause:
16 -8 -1 0 0
Output for trail input:
8 0 0
```

```
1 0 0
Output for resolution:
16 0 8 1 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ 5 (* -9 y)) 0) v
(/= (+ -10 y z) 0) v
(< (+ (* -17 y) z) 0)
Conflict analysis phase finished, building output
Output for conflict clause:
17 -4 -7 0 0
Output for trail input:
4 0 0
7 0 0
Output for resolution:
17 0 4 7 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
(< (+ -3 y) 0) v
(< (+ 5 (* -9 y)) 0)
Conflict analysis at base level, unsat
Output for conflict clause:
-16 -17 0 0
Output for trail input:
Output for resolution:
0 16 17 0
Conflict analysis phase finished
unsat
```

The Python script output is

```
1 8 0 0
2 1 0 0
3 4 0 0
4 7 0 0
5 16 -8 -1 0 0
6 17 -4 -7 0 0
7 -16 -17 0 0
8 16 0 5 1 2 0
9 17 0 6 3 4 0
10 0 7 8 9 0
```

The TraceCheck output returns the empty clause with the following message

```
resolved 1 root and 1 empty clause
```

Example 4

The following input formula is used

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)

(assert (= (+ x y) 5))
(assert (= (- x y) 2))
(assert (= (+ y z) 3))

(assert (not (= x 12)))
(assert (not (= y (- 23))))
(assert (not (= z 100)))

(assert (or (>= x (- 3)) (<= x 4)))
(assert (or (>= y 1) (<= y 14)))
(assert (or (>= z 7) (<= z 42)))

(check-sat)
```

The Yices output is

```
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ -3 (* 2 y)) 0) v
(/= (+ -5 x y) 0) v
(/= (+ -2 x (* -1 y)) 0)
Conflict analysis phase finished, building output
Output for conflict clause:
19 -1 -4 0 0
Output for trail input:
1 0 0
4 0 0
Output for resolution:
19 0 1 4 0
Conflict analysis phase finished
```

```
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ 3 (* -2 y)) 0) v
(/= (+ -5 x y) 0) v
(/= (+ -2 x (* -1 y)) 0)
Conflict analysis phase finished, building output
Output for conflict clause:
20 -1 -4 0 0
Output for trail input:
1 0 0
4 0 0
Output for resolution:
20 0 1 4 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
(< (+ -3 (* 2 y)) 0) v
(< (+ 3 (* -2 y)) 0)
Conflict analysis at base level, unsat
Output for conflict clause:
-19 -20 0 0
Output for trail input:
Output for resolution:
0 19 20 0
Conflict analysis phase finished
unsat
```

The Python script output is

```
1 1 0 0
2 4 0 0
3 19 -1 -4 0 0
4 20 -1 -4 0 0
5 -19 -20 0 0
6 19 0 3 1 2 0
7 20 0 4 1 2 0
8 0 5 6 7 0
```

The TraceCheck output returns the empty clause with the following message

```
resolved 1 root and 1 empty clause
```

Example 5

The following input formula is used

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)

(assert (= (+ x y) 5))
(assert (= (+ y z) 10))
(assert (= (+ x z) 12))

(assert (not (= (- x y) 13)))
(assert (not (= (+ x y) 2)))
(assert (not (= (+ z y) 8)))

(assert (or (>= x (- 12)) (<= x 5)))
(assert (or (>= y 0) (<= y 5)))
(assert (or (>= z 0) (<= z 73)))
(assert (or (< y 3) (> x 12)))

(check-sat)
```

The Yices output is

```
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ 2 (* -1 x) y) 0) v
(/= (+ -10 y z) 0) v
(/= (+ -12 x z) 0)
Conflict analysis phase finished, building output
Output for conflict clause:
22 -4 -6 0 0
Output for trail input:
4 0 0
6 0 0
Output for resolution:
22 0 4 6 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ -3 (* 2 y)) 0) v
```

```
(/= (+ -5 x y) 0) v
(< (+ 2 (* -1 x) y) 0)
Conflict analysis phase finished, building output
Output for conflict clause:
23 -1 -22 0 0
Output for trail input:
1 0 0
Output for resolution:
23 0 1 22 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ -7 (* -1 y)) 0) v
(/= (+ -5 x y) 0) v
(>= (+ 12 (* -1 x)) 0)
Clauses dependent on propagation:
(< (+ -7 (* -1 y)) 0)
(= (+ -5 x y) 0)
(< (+ 12 (* -1 x)) 0)
false
Conflict analysis phase finished, building output
Output for conflict clause:
24 -1 -20 0 0
Output for trail input:
1 0 0
20 0 0
Output for resolution:
24 0 1 20 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
(< (+ -3 (* 2 y)) 0) v
(< (+ -7 (* -1 y)) 0)
Conflict analysis phase finished, building output
Output for conflict clause:
-23 -24 0 0
Output for trail input:
Output for resolution:
-23 0 24 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ -2 x (* -1 y)) 0) v
```

```
(/= (+ -10 y z) 0) v
(/= (+ -12 x z) 0)
Conflict analysis phase finished, building output
Output for conflict clause:
25 -4 -6 0 0
Output for trail input:
4 0 0
6 0 0
Output for resolution:
25 0 4 6 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
(>= (+ 3 (* -2 y)) 0) v
(/= (+ -5 x y) 0) v
(< (+ -2 x (* -1 y)) 0)
Conflict analysis phase finished, building output
Output for conflict clause:
26 -1 -25 0 0
Output for trail input:
1 0 0
Output for resolution:
26 0 1 25 0
Conflict analysis phase finished
Entered conflict analysis phase
Printing output for conflict clause:
(< (+ -3 (* 2 y)) 0) v
(< (+ 3 (* -2 y)) 0)
Conflict analysis at base level, unsat
Output for conflict clause:
-23 -26 0 0
Output for trail input:
Output for resolution:
0 23 26 0
Conflict analysis phase finished
unsat
```

The Python script output is

```
1 4 0 0
2 6 0 0
3 1 0 0
4 20 0 0
```

```
5 22 -4 -6 0 0
6 23 -1 -22 0 0
7 24 -1 -20 0 0
8 -23 -24 0 0
9 25 -4 -6 0 0
10 26 -1 -25 0 0
11 -23 -26 0 0
12 22 0 5 1 2 0
13 23 0 6 3 12 0
14 24 0 7 3 4 0
15 -23 0 8 14 0
16 25 0 9 1 2 0
17 26 0 10 3 16 0
18 0 11 13 17 0
```

The TraceCheck output returns the empty clause with the following message

```
resolved 2 roots and 1 empty clause
```

List of Figures

List of Tables

Bibliography

[1] SAT Association, "The International SAT Competition Web Page." https://satcompetition.github.io/. Last Accessed 15 August 2024.

[2] C. Barrett, L. de Moura, and A. Stump, "SMT-COMP," in *CAV* (K. Etessami and S. K. Rajamani, eds.), pp. 20–23, Springer, Berlin, Heidelberg, 2005.

[3] SAT Association, "SAT Competition 2022: UNSAT Certificates." https://satcompetition.github.io/2022/certificates.html. Last Accessed 15 August 2024.

[4] S. S. Commitee, "SMT-COMP 2023 Results." https://smt-comp.github.io/2023/results.html. Last Accessed 25 May 2024.

[5] A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds., *Handbook of Satisfiability - Second Edition*, vol. 336 of *Frontiers in Artificial Intelligence and Applications*, p. 871. IOS Press, 2021.

[6] C. Barrett, L. de Moura, and P. Fontaine, "Proofs in satisfiability modulo theories," pp. 5–6, 2014.

[7] L. M. de Moura and D. Jovanovic, "A model-constructing satisfiability calculus," in *VMCAI 2013, Rome, Italy, Proceedings*, vol. 7737 of *Lecture Notes in Computer Science*, pp. 1–12, Springer, 2013.

[8] B. Dutertre, "Yices 2.2," in *CAV 2014*, vol. 8559 of *Lecture Notes in Computer Science*, pp. 737–744, Springer, 2014.

[9] J. Christ, J. Hoenicke, and A. Nutz, "SMTInterpol: An Interpolating SMT Solver," in *Model Checking Software*, pp. 248–254, Springer Berlin Heidelberg, 2012.

[10] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *TACAS, 2022, Munich, Germany, Proceedings*, vol. 13243 of *Lecture Notes in Computer Science*, pp. 415–442, Springer, 2022.

[11] L. Kovács and A. Voronkov, "First-Order Theorem Proving and Vampire," in *CAV 2013*, pp. 1–35, Springer Berlin Heidelberg, 2013.

[12] 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*, vol. 53, no. 6, p. 937–977, 2006.

[13] J. Marques Silva and K. Sakallah, "Grasp-a new search algorithm for satisfiability," in *Proceedings of International Conference on Computer Aided Design*, pp. 220–227, 1996.

[14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, p. 530–535, ACM, 2001.

[15] J. Hoenicke and T. Schindler, "SMTInterpol," 2024.

[16] J. Hoenicke and T. Schindler, "A simple proof format for SMT," in *SMT 2022, August 11-12, 2022, Haifa, Israel*, 2022.

[17] A. of cvc5, "cvc5 Proof rules." `https://cvc5.github.io/docs/cvc5-1.0.2/proofs/proof_rules.html`. Last Accessed 4 September 2024.

[18] K. Korovin, L. Kovács, G. Reger, J. Schoisswohl, and A. Voronkov, "Alasca: Reasoning in quantified linear arithmetic," in *TACAS*, pp. 647–665, Springer Nature Switzerland, 2023.

[19] A. R. Bradley and Z. Manna, *The calculus of computation - decision procedures with applications to verification*, pp. 35–36. Springer, 2007.

[20] A. R. Bradley and Z. Manna, *The calculus of computation - decision procedures with applications to verification*, p. 21. Springer, 2007.

[21] A. R. Bradley and Z. Manna, *The calculus of computation - decision procedures with applications to verification*, pp. 8–9. Springer, 2007.

[22] A. R. Bradley and Z. Manna, *The calculus of computation - decision procedures with applications to verification*, pp. 69–70. Springer, 2007.

[23] A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds., *Handbook of Satisfiability - Second Edition*, vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 131–138. IOS Press, 2021.

[24] D. Kroening and O. Strichman, *Decision Procedures - An Algorithmic Point of View, Second Edition*, pp. 60–63. Texts in Theoretical Computer Science. An EATCS Series, Springer, 2016.

[25] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 2, p. 245–257, 1979.

[26] G. Nelson and D. C. Oppen, "Fast decision procedures based on congruence closure," *J. ACM*, vol. 27, no. 2, p. 356–364, 1980.

[27] R. E. Shostak, "An algorithm for reasoning about equality," *Commun. ACM*, vol. 21, no. 7, p. 583–585, 1978.

[28] R. E. Shostak, "A practical decision procedure for arithmetic with function symbols," *J. ACM*, vol. 26, no. 2, p. 351–360, 1979.

[29] A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds., *Handbook of Satisfiability - Second Edition*, vol. 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.

[30] L. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS 2008*, pp. 337–340, Springer, Berlin, Heidelberg, 2008.

[31] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The opensmt solver," pp. 150–153, 2010.

[32] T. Hader, "Non-linear SMT-reasoning over finite fields (Diploma Thesis, Technische Universität Wien)," 2022.

[33] A. Biere, "TraceCheck." `https://fmv.jku.at/tracecheck/index.html`. Last Accessed 24 February 2025.

[34] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," tech. rep., Department of Computer Science, The University of Iowa, 2017.

[35] S. Initiative, "SMT-LIB Benchmarks." `https://smt-lib.org/benchmarks.shtml`. Last Accessed 28 April 2025.