Explaining Unsatisfiability Proofs through Examples

Bachelor Thesis Project Description

Pascal Strebel
supervised by Prof. Dr. Peter Müller, Alexandra Bugariu,
Dr. Malte Schwerhoff

ETH Zürich, Department of Computer Science

March 4, 2021

1 Introduction

A common approach to program verification is to use a verifier that encodes a program and its specification into a set of universally quantified Satisfiability Modulo Theories (SMT) formulas. These formulas can be given to a prover, which decides their satisfiability. Based on the answer of the prover, the verifier then decides if the program is correct with respect to its specification.

There are two main classes of provers that are used most in practice:

> **SMT Solvers** such as *Z3* [1] and *CVC4* [2] invoke algorithms as for example *E-Matching* [3] or *Model-Based Quantifier Instantiation (MBQI)* [4] that differ in how they handle quantifiers. While *MBQI* is essentially a counter-example based refinement loop that can produce unsatisfiability proofs as a side effect, the typically more efficient *E-Matching* algorithm selects partial instantiations based on patterns that are provided for each quantifier.
>
> **Theorem Provers** such as *Vampire* [5] use superposition calculi to identify contradictions in the input formulas. They are designed to generate unsatisfiability proofs (refutations).

We use "solver" to specifically talk about SMT solvers and we use "prover" when we talk about either theorem provers or both of these.

Generally, a prover may return one of the following results:

> **SAT**: The prover concluded that the input formula is satisfiable. Solvers can produce models (valid assignments), but these are not relevant for this project.

UNSAT: The prover concluded that the input formula is unsatisfiable. Depending on the specific algorithm used, it can give us a proof containing the reasoning steps performed to derive the unsatisfiability or a so-called UNSAT core, which is a subset of the input assertions that lead to a contradiction.

UNKNOWN: The prover was not able to decide the satisfiability of the input formula due to incompleteness.

Figure 1 summarizes the tools that will be considered in this project.

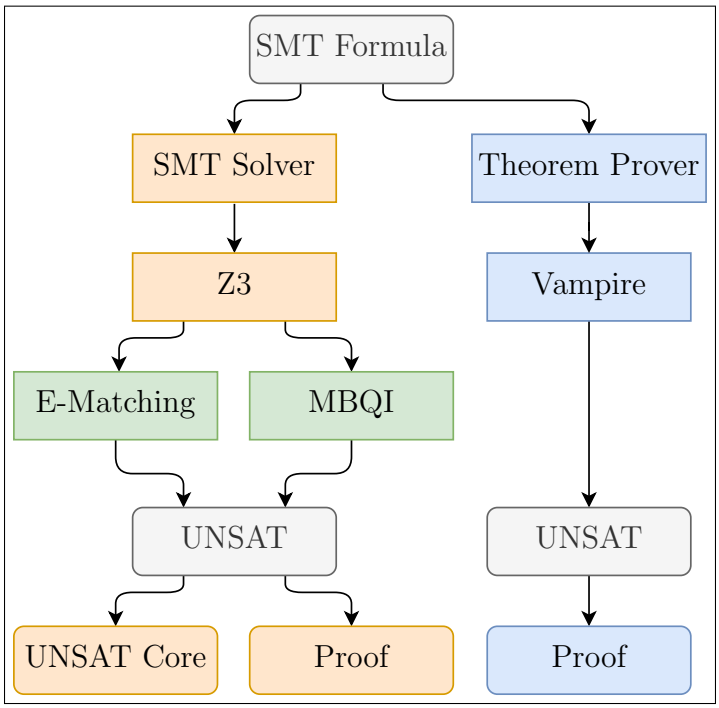

Figure 1: Overview of the tools considered here for deciding unsatisfiability of an SMT formula. The squared boxes represent different types of provers and some of their algorithms, while the rounded boxes represent inputs/outputs.

Program verifiers usually check the negation of a program and its specification. If the negation is satisfiable, then we found a verification error. If it is instead unsatisfiable, we successfully verified our program with respect to the specification.

In some situations, however, users of a verifier are not satisfied with a simple UNSAT answer, as the following two scenarios suggest:

> ***Alice*** *wrote a specification and a program that by construction does not fulfill the specification. This is why she is very surprised that the prover of her choice returns UNSAT when run with the encoding of her program. Instead, she expected the formula to be satisfiable, as the program is not supposed to verify. Since Alice assumes the prover to be correct, there must be an error in either her specification or in the way the verifier encoded her program. To find this error, she wants to look at the unsatisfiability proof that the prover provides.*

> ***Bob*** *also wrote a program, which he expects to successfully verify. However, Z3 using E-Matching returns UNKNOWN when run on the encoding of his program. This forces him to use another algorithm to the same solver or another prover, which indeed returns the UNSAT he expects. He now wonders why E-Matching failed and hopes to get some insights when looking at unsatisfiability proofs produced by alternative algorithms.*

The unsatisfiability proofs produced by *Z3* using *MBQI* and by *Vampire* contain the reasoning steps performed by the prover to derive false, but are long and complex, and require expert knowledge to be understood. If *Alice* and *Bob* are not that experienced in the area of formal reasoning, they will most probably not understand what they are provided by their respective provers.

This project aims to help the users understand why their input formulas to provers are unsatisfiable, by generating simple examples based on the information the unsatisfiability proofs provide.

2 Problem Description

To reconstruct the experience that *Alice* and *Bob* are about to have, we will take a look at such unsatisfiability proofs produced by *Z3* using *MBQI* and by *Vampire*.

Instead of trying to verify extensive programs, we will simplify the *Bob* scenario until only essential details remain. To this end, consider the following two assertions (1) and (2) that might occur in the negated encoding of *Bob*'s program:

(1) $\forall x \in \mathbb{Z} : x \geq 0 \Rightarrow f(x) = x$

(2) $\forall x \in \mathbb{Z} : f(x) > 0$

These assertions clearly contradict each other in the case where $x = 0$. *Z3* using *E-Matching* however returns UNKNOWN when given them in SMT-LIB syntax as shown in Figure 2. Same as *Bob*, we now look at the unsatisfiability proofs produced by *Vampire* and by *Z3* using *MBQI*, which can be seen in Figures 3 and 4, respectively.

```
(declare-fun f (Int) Int)

(assert (forall ((x Int)) (=> (>= x 0) (= (f x) x))))
(assert (forall ((x Int)) (> (f x) 0)))

(check-sat)
(get-proof)
```

Figure 2: Encoding $P := (1) \wedge (2)$ of *Bob*'s simplified program in SMT-LIB syntax.

The unsatisfiability proof provided by *Vampire* in Figure 3 is very structured in the sense that the steps of several stages that were applied are grouped together within the proof. After stating the inputs and simplifying them into a more general form,

Vampire usually lists several axioms that will be needed in later steps. For our simple input it only requires the non-reflexivity of strict inequality $\neg(X_0 < X_0)$, as stated in line 10. The preprocessing step then transforms the input formula into clauses, before superposition and resolution calculi are used to arrive at a contradiction.

```
% Refutation found. Thanks to Tanya!
1.  ! [X0 : $int] : ($greatereq(X0,0) => f(X0) = X0) [input]
2.  ! [X0 : $int] : $greater(f(X0),0) [input]
3.  ! [X0 : $int] : $less(0,f(X0)) [evaluation 2]
4.  ! [X0 : $int] : (~$less(X0,0) => f(X0) = X0) [evaluation 1]
10. ~$less(X0,X0) [theory axiom]
17. ! [X0 : $int] : (f(X0) = X0 | $less(X0,0)) [ennf transformation 4]
18. $less(0,f(X0)) [cnf transformation 3]
19. $less(X0,0) | f(X0) = X0 [cnf transformation 17]
21. f(0) = 0 [resolution 19,10]
22. $less(0,0) [superposition 18,21]
23. $false [evaluation 22]
```

Figure 3: Unsatisfiability proof produced by *Vampire* for P. The missing lines have been removed by *Vampire* itself and never appeared in the output.

Each proof step states a formula that is derived by an inference rule, which most often uses one or multiple premises. For example "$f(0) = 0$ *[resolution 19,10]*" in line 21 can be translated to "$f(0) = 0$ *is derived from* $(X_0 < 0) \vee (f(X_0) = X_0)$ *(line 19) and* $\neg(X_0 < X_0)$ *(line 10) using a resolution calculus*". Note that this very line implicitly instantiates the quantified variable and actually suffices to generate an example that exposes the contradiction in the input formula.

The unsatisfiability proof provided by *Z3* using *MBQI* in Figure 4 works in a similar fashion and is typically even longer. It also implicitly instantiates the quantified variable in lines 3 and 4, but this proof further contains explicit quantifier instantiations in lines 33 and 46.

```
1.  unsat
2.  ((proof
3.  (let ((?x125 (f 0)))
4.  (let (($x198 (= 0 ?x125)))
5.  (let ((@x204 (symm (commutativity (= $x198 (= ?x125 0))) (= (= ?x125 0) $x198))))
6.  (let (($x176 (= ?x125 0)))
7.  (let (($x104 (forall ((x Int) )(! (let (($x26 (>= x 0))) (let (($x32 (not $x26)))
    (or $x32 (= (+ x (* (- 1) (f x))) 0)))) :pattern ( (f x) ) :qid k!6) ))
8.  (let (($x66 (forall ((x Int) )(! (let (($x26 (>= x 0))) (let (($x32 (not $x26)))
    (or $x32 (= (+ x (* (- 1) (f x))) 0)))) :qid k!6) ))
9.  (let ((?x107 (lambda ((x Int) )(let (($x26 (>= x 0))) (let (($x32 (not $x26)))
    (let (($x87 (or $x32 (= (+ x (* (- 1) (f x))) 0)))) (refl (= $x87 $x87)))))) ))
10. (let ((?x91 (lambda ((x Int) )(let (($x26 (>= x 0))) (let (($x32 (not $x26)))
    (let (($x87 (or $x32 (= (+ x (* (- 1) (f x))) 0)))) (refl (~ $x87 $x87)))))) ))
11. (let (($x69 (forall ((x Int) )(! (let (($x26 (>= x 0))) (let (($x32 (not $x26)))
    (or $x32 (= (+ (f x) (* (- 1) x)) 0)))) :qid k!6) ))
12. (let ((?x75 (lambda ((x Int) )(let (($x26 (>= x 0))) (let (($x32 (not $x26)))
    (let (($x87 (or $x32 (= (+ x (* (- 1) (f x))) 0))))
    (let (($x67 (or $x32 (= (+ (f x) (* (- 1) x)) 0)))) (rewrite (= $x67 $x87)))))))))
13.   (let (($x36 (forall ((x Int) )(! (let ((?x27 (f x))) (let (($x28 (= ?x27 x)))
      (let (($x26 (>= x 0))) (let (($x32 (not $x26))) (or $x32 $x28)))) :qid k!6) ))
14.   (let ((?x71 (lambda ((x Int) )(let (($x26 (>= x 0))) (let (($x32 (not $x26)))
      (let (($x67 (or $x32 (= (+ (f x) (* (- 1) x)) 0)))) (let ((?x27 (f x)))
      (let (($x28 (= ?x27 x))) (let (($x33 (or $x32 $x28))) (rewrite (= $x33 $x67))))))))))
15.   (let (($x30 (forall ((x Int) )(! (let ((?x27 (f x))) (let (($x28 (= ?x27 x)))
      (let (($x26 (>= x 0))) (=> $x26 $x28)))) :qid k!6) ))
16.   (let ((?x37 (lambda ((x Int) )(let ((?x27 (f x))) (let (($x28 (= ?x27 x)))
      (let (($x26 (>= x 0))) (let (($x32 (not $x26))) (let (($x33 (or $x32 $x28)))
      (let (($x29 (=> $x26 $x28))) (rewrite (= $x29 $x33)))))))))) ))
17.   (let (((@x41 (mp (asserted $x30) (quant-intro (proof-bind ?x37) (= $x30 $x36)) $x36)))
18.   (let (((@x65 (mp (mp @x41 (rewrite (= $x36 $x36)) $x36)
      (quant-intro (proof-bind ?x71) (= $x36 $x69)) $x69)))
19.   (let (((@x95 (mp~ (mp @x65 (quant-intro (proof-bind ?x75) (= $x69 $x66)) $x66)
      (nnf-pos (proof-bind ?x91) (~ $x66 $x66)) $x66)))
20.   (let (($x188 (not $x104)))
21.   (let (($x189 (or $x188 $x176)))
22.   (let ((?x158 (* (- 1) ?x125)))
23.   (let ((?x159 (+ 0 ?x158)))
24.   (let (($x160 (= ?x159 0)))
25.   (let (($x153 (>= 0 0)))
26.   (let (($x154 (not $x153)))
27.   (let (($x161 (or $x154 $x160)))
28.   (let (($x190 (or $x188 $x161)))
29.   (let (($x192 (= $x190 $x189)))
30.   (let (((@x180 (trans (monotonicity (rewrite (= ?x159 ?x158)) (= $x160 (= ?x158 0)))
      (rewrite (= (= ?x158 0) $x176)) (= $x160 $x176))))
31.   (let (((@x170 (trans (monotonicity (rewrite (= $x153 true)) (= $x154 (not true)))
      (rewrite (= (not true) false)) (= $x154 false))))
32.   (let (((@x187 (trans (monotonicity @x170 @x180 (= $x161 (or false $x176)))
      (rewrite (= (or false $x176) $x176)) (= $x161 $x176))))
33.   (let (((@x197 (mp ((_ quant-inst 0) $x190) (trans (monotonicity @x187 $x192)
      (rewrite (= $x189 $x189)) $x192) $x189)))
34.   (let (((@x200 (unit-resolution @x197 (mp @x95 (quant-intro (proof-bind ?x107)
      (= $x66 $x104)) $x104) $x176)))
35.   (let (($x206 (not $x198)))
36.   (let (($x126 (<= ?x125 0)))
37.   (let (($x141 (not $x126)))
38.   (let (($x112 (forall ((x Int) )(! (not (<= (f x) 0)) :pattern ( (f x) ) :qid k!7))))
39.   (let (($x49 (forall ((x Int) )(! (not (<= (f x) 0)) :qid k!7)) ))
40.   (let ((?x115 (lambda ((x Int) )(refl (= (not (<= (f x) 0)) (not (<= (f x) 0)))))))
41.   (let ((?x98 (lambda ((x Int) )(refl (~ (not (<= (f x) 0)) (not (<= (f x) 0)))))) ))
42.   (let (($x43 (forall ((x Int) )(! (let ((?x27 (f x))) (> ?x27 0)) :qid k!7)) ))
43.   (let ((?x50 (lambda ((x Int) )(rewrite (= (> (f x) 0) (not (<= (f x) 0)))))) ))
44.   (let (((@x54 (mp (asserted $x43) (quant-intro (proof-bind ?x50) (= $x43 $x49)) $x49)))
45.   (let (((@x102 (mp~ (mp @x54 (rewrite (= $x49 $x49)) $x49)
      (nnf-pos (proof-bind ?x98) (~ $x49 $x49)) $x49)))
46.   (let (((@x209 (unit-resolution ((_ quant-inst 0) (or (not $x112) $x141))
      (mp @x102 (quant-intro (proof-bind ?x115) (= $x49 $x112)) $x112) $x141)))
47.   (unit-resolution (unit-resolution ((_ th-lemma arith triangle-eq)
      (or $x206 $x126)) @x209 $x206)
48.   (mp @x200 @x204 $x198) false)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

Figure 4: Unsatisfiability proof produced by *Z3* using *MBQI* for P.

The overall challenges that arise when working with these unsatisfiability proofs are:

- They are long, complex and hard to read for humans. Especially the simplification and preprocessing steps only serve the machine-checkability, but do not provide any qualitative contribution to the reasoning.
- Some quantifier instantiations are only implicit, which is why it does not suffice to just search all the explicit quantifier instantiations.
- There is no common standard in which the proofs are presented. Both provers examined here employ differences in their semantics, the way they handle references and their general proof structures.

3 Solution Approach

Having simple examples that expose the contradictions derived in unsatisfiability proofs prevents *Alice* and *Bob* from the trouble of reconstructing these proofs. We want to design and implement an algorithm that provides such examples.

In a first step, we will determine what exact information the proofs include, by manually analyzing a variety of them. Next, we will shift our focus to the users and their needs and accordingly define the format of our examples. These primary steps will equip us with all the information necessary to formulate the specific task of our algorithm, which we can then realize.

To evaluate our solution approach, we will design a validation mechanism that checks if the examples we generate are indeed sufficient to expose the contradiction.

4 Core Goals

With the high-level overview given in the Solution Approach in mind we set ourselves the following core goals:

- **Analyzing Proofs**
  We analyze various proofs generated by *Z3* using *MBQI* and by *Vampire* to gain the required knowledge of the information they provide. We assume that the contradictions always include universally quantified variables, as the formulas could otherwise be solved by *E-Matching*. Since the explicit quantifier instantiations typically do not suffice to determine the contradiction, we will also consider implicit instantiations of quantified variables and what information they provide with their syntactical appearance.
- **Analyzing Users Needs**
  Our solution should enhance the user experience when working with unsatisfiability proofs. We achieve this by discussing the following components:
  - The **information** provided by our examples should include all the essential details that a human requires to reproduce the contradiction.

  - We want to find a suitable way to **present** the extracted examples that is not too overwhelming or complicated.

- **Designing a Representation Format**
  We introduce a layer of abstraction by designing a generic format that represents the information required to construct our examples. This separates preprocessing from the main algorithm and provides the modularity required to add support for other provers and algorithms that have a different output format.

- **Choosing a Programming Language**
  We choose a suitable programming language for implementing our algorithms based on the knowledge we gain through the theoretical study of the unsatisfiability proofs and on the existence of helpful libraries that possibly exist.

- **Extracting Information**
  We develop and implement an algorithm that automatically extracts the necessary information from the proofs produced by both *Z3* using *MBQI* and *Vampire* and translates it into our representation format. Our examples include pairs of assignments from quantified variables to values and additional information about how the accordingly instantiated formulas relate to each other. Given the proofs of our simple formula from the Problem Description, we would therefore extract the information $x = 0$ and $f(0)$ that leads to the contradicting constraints for f which arise when the quantified variable x is instantiated accordingly.

- **Generating Examples**
  We develop and implement an algorithm that generates the examples by using the information given in our generic format and meets the users needs.

- **Validation**
  We develop and implement a validation mechanism that checks if the generated examples are indeed sufficient to expose the contradiction in the input formulas. Given again the proofs of our simple formula from the Problem Description, we could for example instantiate $x = 0$ and detect the contradicting constraints for $f(0)$ by deriving $f(0) = 0$ based on assertion (1) and then comparing it to the constraint set by assertion (2), which will expose the contradiction.
  We could again rely on a prover to evaluate the assertions. Validation gets more difficult if the contradiction is between quantified and quantifier-free parts of the input formula, which is typically the case.

- **Evaluation**
  We evaluate our approach on benchmarks from *SMT-COMP* [6].

5 Extension Goals

Additionally, we could address some of the following extension goals:

- **Minimization Algorithm**
  Unsatisfiability proofs produced by *Z3* using *MBQI* and by *Vampire* are not guaranteed to be minimal. If a formula is extensive, the extracted example will include assignments of a possibly huge number of quantified variables, which is not human-readable. We could design an algorithm that extracts the subset of assertions that contradict each other, i.e. a minimal example. The information provided by UNSAT core might help us there.

- **Support for additional Provers**
  We could look at other provers (e.g. *CVC4*) that also produce unsatisfiability proofs and translate their output into our generic format.

- **Integration into *Viper***
  Viper (Verification Infrastructure for Permission-based Reasoning) is a language and suite of tools developed at ETH Zurich, providing an architecture on which new verification tools and prototypes can be developed simply and quickly. [7] We could extend *Viper* by integrating our algorithm.

- **Build the Bridge to *E-Matching***
  E-Matching does a controlled number of quantifier instantiations based on given patterns to stay as time-efficient as possible. A failure of *E-Matching* (i.e., it returning UNKNOWN) is therefore often caused by a lack of triggering terms that would be necessary to discover the contradiction. Maybe we can use the examples our algorithm generates to add further assertions to the input formula that only syntactically match patterns to trigger the necessary quantifier instantiations.

6 Schedule

To meet our core goals, we will roughly adhere to the following schedule:

Week 1: 01.02.2021 - 07.02.2021	Theoretical Study
Week 2: 08.02.2021 - 14.02.2021	Theoretical Study
Week 3: 15.02.2021 - 21.02.2021	Project Description
Week 4: 22.02.2021 - 28.02.2021	Project Description
Week 5: 01.03.2021 - 07.03.2021	Initial Presentation
Week 6: 08.03.2021 - 14.03.2021	Analyzing Proofs
Week 7: 15.03.2021 - 21.03.2021	Analyzing Proofs
Week 8: 22.03.2021 - 28.03.2021	Analyzing User Needs
Week 9: 29.03.2021 - 04.04.2021	Analyzing User Needs
Week 10: 05.04.2021 - 11.04.2021	Designing a Representation Format
Week 11: 12.04.2021 - 18.04.2021	Designing a Representation Format
Week 12: 19.04.2021 - 25.04.2021	Extracting Information
Week 13: 26.04.2021 - 02.05.2021	Extracting Information
Week 14: 03.05.2021 - 09.05.2021	Generating Examples
Week 15: 10.05.2021 - 16.05.2021	Generating Examples
Week 16: 17.05.2021 - 23.05.2021	Generating Examples
Week 17: 24.05.2021 - 30.05.2021	Validation
Week 18: 31.05.2021 - 06.06.2021	Validation
Week 19: 07.06.2021 - 13.06.2021	Validation
Week 20: 14.06.2021 - 20.06.2021	Validation
Week 21: 21.06.2021 - 27.06.2021	Evaluation
Week 22: 28.06.2021 - 04.07.2021	Write Thesis
Week 23: 05.07.2021 - 11.07.2021	Write Thesis
Week 24: 12.07.2021 - 18.07.2021	Write Thesis
Week 25: 19.07.2021 - 25.07.2021	Prepare Presentation
Week 26: 26.07.2021 - 01.08.2021	Prepare Presentation

References

[1] de Moura, L., Bjørner, N.: **Z3: An efficient smt solver**. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

[2] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: **Cvc4**. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 171–177. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[3] de Moura, L., Bjørner, N.: **Efficient e-matching for smt solvers**. In: Pfenning, F. (ed.) Automated Deduction – CADE-21. pp. 183–198. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)

[4] Ge, Y., de Moura, L.: **Complete instantiation for quantified formulas in satisfiabiliby modulo theories**. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. pp. 306–320. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

[5] Kovács, L., Voronkov, A.: **First-order theorem proving and vampire**. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 1–35. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[6] SMT-COMP: The 15th international satisfiability modulo theories competition (2020), https://smt-comp.github.io/2020/

[7] Müller, P., Schwerhoff, M., Summers, A. J.: **Viper: A verification infrastructure for permission-based reasoning**. In: Jobstmann, B., Leino, K.R.M (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI), ser. LNCS, vol. 9583. pp. 41–62. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)