Explaining Unsatisfiability Proofs through Examples

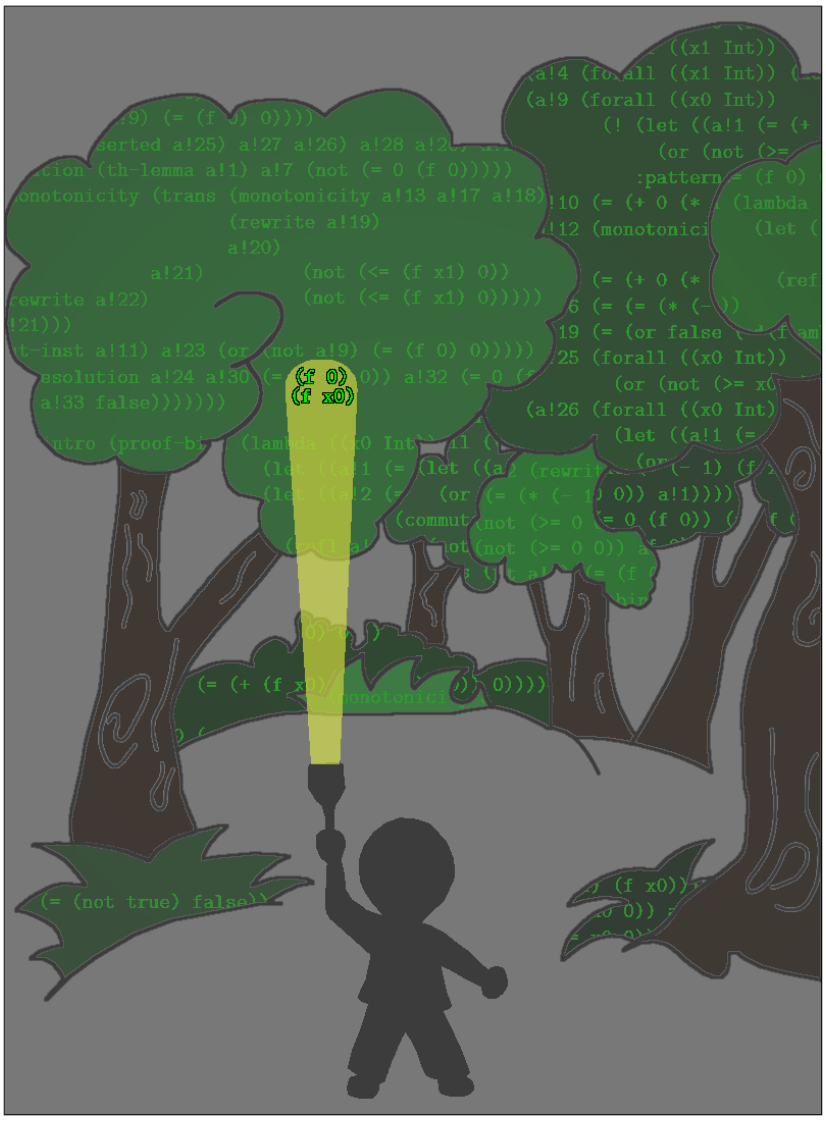
Bachelor Thesis

Pascal Strebel

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

ETH Zürich, Department of Computer Science

September 1, 2021



Abstract

SMT solvers such as *Z3* and theorem provers such as *Vampire* are widely used in program verification. Verifiers usually rely on quantifiers to encode the specification and the semantics of a programming language, which is why universally quantified variables are frequently responsible for the unsatisfiability of a set of SMT formulas. Provers can generate an unsat-proof for unsatisfiable inputs, which contains the reasoning steps performed to derive $false$. They often use explicit or implicit instantiation for quantified variables. However, such unsat-proofs are usually long and complex, and require expert knowledge to be understood.

In this thesis, we present a technique for using unsat-proof information to generate simple EXAMPLES that explain the contradiction derived in these. We implemented our technique as a reusable tool, which is highly modular and can be easily extended to other provers.

Acknowledgements

I would like to express my deepest gratitude to Alexandra Bugariu for her extraordinary support, her many explanations, her numerous ideas, and her valuable feedback.

I would also like to thank Prof. Dr. Peter Müller and the Programming Methodology Group for the opportunity to work on this exciting and simultaneously challenging project.

Contents

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. Most of the universal quantifiers in SMT formulas generated by program verifiers appear in axioms over uninterpreted functions, which are functions that are not defined in a built-in theory. These formulas can then be given to a prover to decide their satisfiability. Based on the answer of the prover, the verifier determines if the program is correct with respect to its specification.

In general, a prover can return one of the following results:

Sat. The prover infers that the input formulas are satisfiable together. Some solvers can generate models (valid assignments), but these are not relevant for this thesis.

Unsat. The prover infers that the input formulas are unsatisfiable together. Some provers can generate an **unsat-proof**, which contains the reasoning steps performed to derive $false$, or an **unsat-core**, which is a subset of the input formulas that contains contradictory constraints.

Unknown. The prover is not able to determine the satisfiability of the input formulas due to incompleteness or insufficient resources.

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

SMT Solvers such as ***Z3*** [1] invoke algorithms such as *Model-Based Quantifier Instantiation (MBQI)* [2] or *E-Matching* [3] to examine the input for satisfiability. While *MBQI* is essentially a counter-example based refinement loop that can generate unsat-proofs as a side effect, the typically more efficient *E-Matching* algorithm performs the instantiations based on patterns that are provided for each quantifier.

Theorem Provers such as ***Vampire*** [4] use superposition calculi to identify a contradiction in the input formulas and are therefore designed to generate unsat-proofs.

Thus, *Z3* and *Vampire* differ in the underlying technique and their approach to generate the unsat-proofs. They are similar because we can use them for the same problem category and because they complement each other (i.e., there are inputs for which one of them is successful and the other is not), presumably just thanks to the different approaches they pursue according to [5].

Figure 1 summarizes the tools that are considered in this thesis.

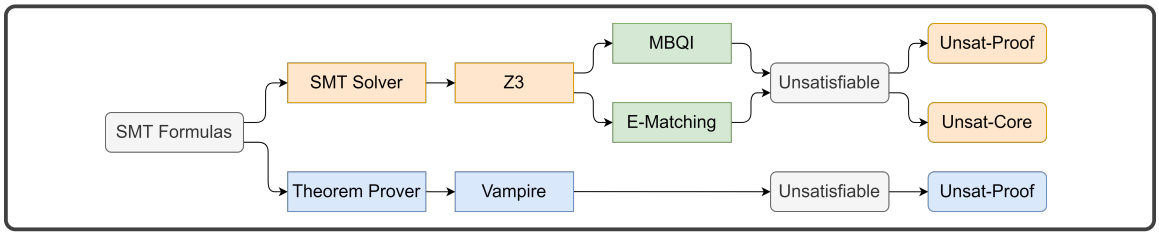


Figure 1: Overview of the tools considered in this thesis for determining the unsatisfiability of a set of SMT formulas. The square boxes represent different types of provers and some of their algorithms, while the rounded boxes represent the inputs and outputs.

1.1 Problem Description

Program verifiers usually check the negation of a program and its specification. If the negation is satisfiable, then there is a verification error. If instead it is unsatisfiable, then the program has been successfully verified with respect to the specification. In some situations, however, the users not only want to know **that** their input formulas are unsatisfiable, but also **why** this is the case, as the following two scenarios show:

Alice *wrote a specification and a program that, by construction, does not fulfill the specification. Therefore, she is very surprised that the prover of her choice infers unsat when used with the encoding of her program. Instead, she expected the formulas to be satisfiable, since the program is not intended to successfully verify. Since Alice assumes that the prover is correct, there must be an error either in her specification or in the way the verifier encoded her program. To find this error, she wants to look at the unsat-proof that the prover can generate.*

Bob *also wrote a program that he expects to successfully verify. However, Z3 with E-Matching returns unknown when used with the encoding of his program. This forces him to use a different algorithm to the same solver or another prover, which indeed infers the unsatisfiability he expects.*[1] *He wonders why E-Matching failed and hopes to get some insight by looking at unsat-proofs generated by the alternative algorithms.*

To reconstruct the experience of *Alice* and *Bob*, we use such unsat-proofs generated by *Z3* with *MBQI* and by *Vampire*. Let us first consider a very simple version of the *Bob* scenario. To this end, consider the following two formulas that might occur in the negated encoding of *Bob*'s program:

$$\forall x_0 \in \mathbb{Z} : x_0 \geq 0 \Rightarrow f(x_0) = x_0$$

$$\forall x_1 \in \mathbb{Z} : f(x_1) > 0$$

These formulas set contradictory constraints on the uninterpreted function f, as they simultaneously require $f(0) = 0$ and $f(0) > 0$. Thus, no interpretation can be found for f, so the two formulas are unsatisfiable together. *Z3* with *E-Matching* returns *unknown* when given these formulas as input in SMT-LIB syntax [6] as shown in Figure 2, while *Vampire* and *Z3* with *MBQI* return *unsat* and generate a proof.

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

(assert (forall ((x0 Int)) (! (=> (>= x0 0) (= (f x0) x0)) :pattern (f x0))))
(assert (forall ((x1 Int)) (! (> (f x1) 0) :pattern (f x1))))
```

Figure 2: Encoding of *Bob*'s input in SMT-LIB syntax. The patterns are only required for *E-Matching*; *MBQI* and *Vampire* ignore them. For simplicity, we therefore omit them in the further inputs that we show.

[1] Note that we use the word *solver* to talk specifically about SMT solvers, and we use the word *prover* when talking about either theorem provers or both.

Same as *Bob*, we now manually inspect the unsat-proofs generated by *Vampire* and by *Z3* with *MBQI*, which can be seen in Figures 3 and 4, respectively.

The unsat-proof generated by *Vampire* in Figure 3 is very structured in the sense that the steps of several applied stages are grouped together within the proof. After stating the inputs and simplifying them into a more general form, *Vampire* usually lists several axioms that are needed in later steps. For our simple input, only the non-reflexivity of strict inequality $\neg(X_0 < X_0)$ is needed, as stated in line 10. In the preprocessing stage, the input formulas are converted into clauses. Then, it uses superposition and resolution calculi to derive a contradiction.

```
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: Unsat-proof for *Bob*'s input generated by *Vampire*. The missing lines were removed by *Vampire* and never appeared in the output.

Each proof step states a formula that is derived by an inference rule, usually using one or more premises. E.g., `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 line implicitly instantiates the quantified variable $X0$ (which, as we explain in Section 3.3.2.3, corresponds to the two quantified variables x_0 and x_1 from the input) with 0 and actually suffices to expose the contradiction in the input formulas.

The unsat-proof generated by *Z3* with *MBQI* in Figure 4 is typically even longer and more complicated. As opposed to the *Vampire* proof, the *Z3* proof includes explicit quantifier instantiations in lines 83 and 106 that it marks as such (`quant-inst`).

By now, it is clear that various challenges arise when working with such unsat-proofs. On the one hand, they are long, complex, and difficult to read for humans. In particular, the preprocessing and simplification steps often only serve the machine-checkability, but do not provide a qualitative contribution to the reasoning. Moreover, quantifier instantiations can occur both explicitly and implicitly within possibly complex, nested expressions. And last but not least, there is no common standard in which these unsat-proofs are presented. Both provers studied in this thesis (i.e., *Z3* and *Vampire*) differ in their semantics, the way they handle quantified variables and references, and their general proof structures.

```
unsat
((proof
(let ((?x127 (f 0)))
 (let (($x172 (= 0 ?x127)))
 (let ((@x178 (symm (commutativity (= $x172 (= ?x127 0))) (= (= ?x127 0) $x172))))
 (let (($x150 (= ?x127 0)))
 (let (($x104 (forall ((x0 Int) )(! (let (($x26 (>= x0 0)))
 (let (($x32 (not $x26)))
 (or $x32 (= (+ x0 (* (- 1) (f x0))) 0)))) :pattern ( (f x0) ) :qid k!9))
 ))
 (let (($x66 (forall ((x0 Int) )(! (let (($x26 (>= x0 0)))
 (let (($x32 (not $x26)))
 (or $x32 (= (+ x0 (* (- 1) (f x0))) 0)))) :qid k!9))
 ))
 (let ((?x107 (lambda ((x0 Int) )(let (($x26 (>= x0 0)))
 (let (($x32 (not $x26)))
 (let (($x87 (or $x32 (= (+ x0 (* (- 1) (f x0))) 0))))
 (refl (= $x87 $x87))))))
 ))
 (let ((?x91 (lambda ((x0 Int) )(let (($x26 (>= x0 0)))
 (let (($x32 (not $x26)))
 (let (($x87 (or $x32 (= (+ x0 (* (- 1) (f x0))) 0))))
 (refl (~ $x87 $x87))))))
 ))
 (let (($x69 (forall ((x0 Int) )(! (let (($x26 (>= x0 0)))
 (let (($x32 (not $x26)))
 (or $x32 (= (+ (f x0) (* (- 1) x0)) 0)))) :qid k!9))
 ))
 (let ((?x75 (lambda ((x0 Int) )(let (($x26 (>= x0 0)))
 (let (($x32 (not $x26)))
 (let (($x87 (or $x32 (= (+ x0 (* (- 1) (f x0))) 0))))
 (let (($x67 (or $x32 (= (+ (f x0) (* (- 1) x0)) 0))))
 (rewrite (= $x67 $x87)))))))
 ))
 (let (($x36 (forall ((x0 Int) )(! (let ((?x27 (f x0)))
 (let (($x28 (= ?x27 x0)))
 (let (($x26 (>= x0 0)))
 (let (($x32 (not $x26)))
 (or $x32 $x28))))) :qid k!9))
 ))
 (let ((?x71 (lambda ((x0 Int) )(let (($x26 (>= x0 0)))
 (let (($x32 (not $x26)))
 (let (($x67 (or $x32 (= (+ (f x0) (* (- 1) x0)) 0))))
 (let ((?x27 (f x0)))
 (let (($x28 (= ?x27 x0)))
 (let (($x33 (or $x32 $x28)))
 (rewrite (= $x33 $x67)))))))))
 ))
 (let (($x30 (forall ((x0 Int) )(! (let ((?x27 (f x0)))
 (let (($x28 (= ?x27 x0)))
 (let (($x26 (>= x0 0)))
 (=> $x26 $x28)))) :qid k!9))
 ))
 (let ((?x37 (lambda ((x0 Int) )(let ((?x27 (f x0)))
 (let (($x28 (= ?x27 x0)))
 (let (($x26 (>= x0 0)))
 (let (($x32 (not $x26)))
 (let (($x33 (or $x32 $x28)))
 (let (($x29 (=> $x26 $x28)))
 (rewrite (= $x29 $x33)))))))))
 ))
 (let ((@x41 (mp (asserted $x30) (quant-intro (proof-bind ?x37) (= $x30 $x36)) $x36)))
 (let ((@x65 (mp (mp @x41 (rewrite (= $x36 $x36)) $x36) (quant-intro (proof-bind ?x71)
    (= $x36 $x69)) $x69)))
 (let ((@x95 (mp~ (mp @x65 (quant-intro (proof-bind ?x75) (= $x69 $x66)) $x66)
    (nnf-pos (proof-bind ?x91) (~ $x66 $x66)) $x66)))
 (let (($x162 (not $x104)))
 (let (($x163 (or $x162 $x150)))
 (let ((?x129 (* (- 1) ?x127)))
 (let ((?x124 (+ 0 ?x129)))
 (let (($x130 (= ?x124 0)))
 (let (($x136 (>= 0 0)))
 (let (($x137 (not $x136)))
 (let (($x131 (or $x137 $x130)))
 (let (($x164 (or $x162 $x131)))
 (let (($x166 (= $x164 $x163)))
 (let ((@x154 (trans (monotonicity (rewrite (= ?x124 ?x129)) (= $x130 (= ?x129 0)))
    (rewrite (= (= ?x129 0) $x150)) (= $x130 $x150))))
 (let ((@x144 (trans (monotonicity (rewrite (= $x136 true)) (= $x137 (not true)))
    (rewrite (= (not true) false)) (= $x137 false))))
 (let ((@x161 (trans (monotonicity @x144 @x154 (= $x131 (or false $x150)))
    (rewrite (= (or false $x150) $x150)) (= $x131 $x150))))
 (let ((@x171 (mp ((_ quant-inst 0) $x164) (trans (monotonicity @x161 $x166)
    (rewrite (= $x163 $x163)) $x166) $x163)))
 (let ((@x174 (unit-resolution @x171 (mp @x95 (quant-intro (proof-bind ?x107)
    (= $x66 $x104)) $x104) $x150)))
 (let (($x180 (not $x172)))
 (let (($x134 (<= ?x127 0)))
 (let (($x135 (not $x134)))
 (let (($x112 (forall ((x1 Int) )(! (not (<= (f x1) 0)) :pattern ( (f x1) ) :qid k!10))
 ))
 (let (($x49 (forall ((x1 Int) )(! (not (<= (f x1) 0)) :qid k!10))
 ))
 (let ((?x115 (lambda ((x1 Int) )(refl (= (not (<= (f x1) 0)) (not (<= (f x1) 0)))))
 ))
 (let ((?x98 (lambda ((x1 Int) )(refl (~ (not (<= (f x1) 0)) (not (<= (f x1) 0)))))
 ))
 (let (($x43 (forall ((x1 Int) )(! (let ((?x27 (f x1)))
 (> ?x27 0)) :qid k!10))
 ))
 (let ((?x50 (lambda ((x1 Int) )(rewrite (= (> (f x1) 0) (not (<= (f x1) 0)))))
 ))
 (let ((@x54 (mp (asserted $x43) (quant-intro (proof-bind ?x50) (= $x43 $x49)) $x49)))
 (let ((@x102 (mp~ (mp @x54 (rewrite (= $x49 $x49)) $x49)
    (nnf-pos (proof-bind ?x98) (~ $x49 $x49)) $x49)))
 (let ((@x183 (unit-resolution ((_ quant-inst 0) (or (not $x112) $x135))
    (mp @x102 (quant-intro (proof-bind ?x115) (= $x49 $x112)) $x112) $x135)))
 (unit-resolution (unit-resolution ((_ th-lemma arith triangle-eq) (or $x180 $x134))
    @x183 $x180) (mp @x174 @x178 $x172) false))))))))))))))))))))))))))))))))))))))))))))))
```

Figure 4: Unsat-proof for *Bob*'s input generated by *Z3* with *MBQI*.

In summary, the unsat-proofs generated by *Z3* with *MBQI* and by *Vampire* contain the reasoning steps performed by the prover to derive *false*, but they are long and complex, and require expert knowledge to be understood. If *Alice* and *Bob* are not so experienced in the field of formal reasoning, they will most likely not understand what is provided to them by their respective prover.

1.2 Solution Approach

Given a set of SMT formulas for which either *Z3* with *MBQI* or *Vampire* can produce an unsat-proof, our technique generates a simple EXAMPLE that explains the contradiction derived in the unsat-proof. This benefits the users' understandability by showing in a simplified way why the input formulas are unsatisfiable, and therefore *Alice* and *Bob* do no longer have to manually reconstruct the unsat-proofs.

We have designed and implemented a technique that generates such EXAMPLES based on instantiations of quantified variables we identify in the unsat-proofs. We consider both **explicit** quantifier instantiations, which are explicitly marked as such, and **implicit** quantifier instantiations, which appear in terms where quantified variables are substituted by or related to concrete values, but the corresponding proof step does not mention this.

An EXAMPLE involves assignments of concrete values to quantified variables and the contradictory constraints for them. Thus, we modify the quantified parts of the input by instantiating the universally quantified variables with concrete values we extracted from the unsat-proof, and expose the contradiction that is derived in it.

1.3 Contributions

With this thesis we make the following contributions:

1. We present a technique that generates simple EXAMPLES to explain unsat-proofs produced by *Z3* with *MBQI* and by *Vampire*, which include all the details that a human user requires to understand why the input formulas are unsatisfiable.

2. We extract explicit and implicit quantifier instantiations from unsat-proofs generated by *Z3* with *MBQI* and by *Vampire* and use them to generate EXAMPLES that expose the contradiction derived in the unsat-proofs. Our technique is successful if the unsat-proof performs all the quantifier instantiations necessary for the contradiction.

3. We implemented our technique as a reusable tool, which is highly modular and can be easily extended to other provers.

4. We evaluated our implementation on manually and automatically generated benchmarks of varying complexity. Our experimental results show that it successfully generated EXAMPLES in 63% of the cases. Our EXAMPLES can significantly reduce the human effort in identifying the reason for the unsatisfiablity.

5. With this written report, we present a detailed study of unsat-proofs generated by *Z3* with *MBQI* and by *Vampire*, and provide a variety of inputs causing specific behaviour.

1.4 Outline

The rest of this thesis is structured as follows: Section 2 presents a high-level overview of our technique; the details follow in Section 3. In Section 4 we explain how we included user needs in the design of our presentation format. Section 5 discusses the implementation details of our technique and in Section 6 we present our experimental results. Section 7 demonstrates another possible application of our EXAMPLES by using them to trigger quantifier instantiations in *E-Matching*. We then discuss related work in Section 8 and possible future work in Section 9. Finally, we draw conclusions in Section 10.

2 Overview

In this section, we give a high-level overview of our approach, which is presented in Figure 5. It takes as input a set of SMT formulas and uses a prover to produce the corresponding unsat-proof, and it generates an EXAMPLE that explains why the input is unsatisfiable. This EXAMPLE is primarily based on quantified variables and their instantiations in the unsat-proof. To find them we rely on explicit markers and heuristics for their implicit occurrence. In the following, we describe the main steps of our technique (Section 2.1) and illustrate them on an example (Section 2.2).

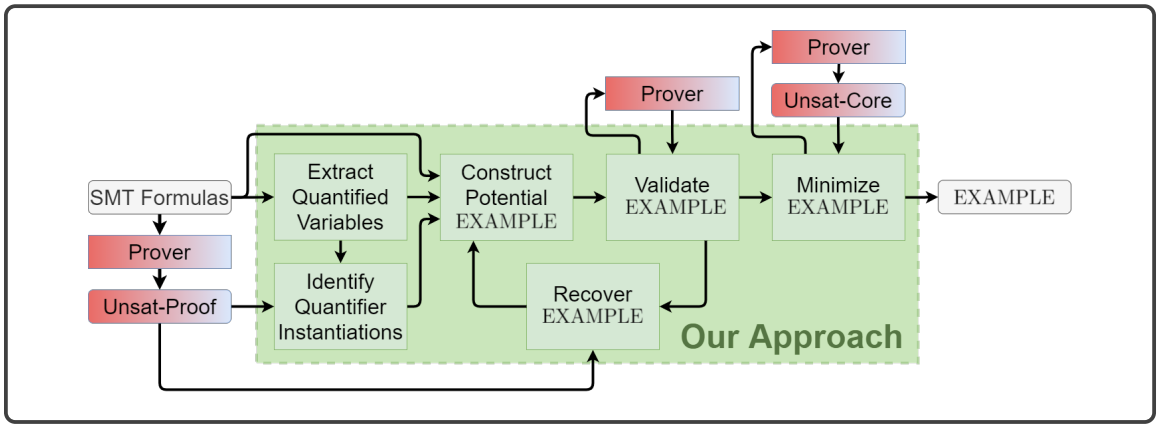


Figure 5: Overview of the approach pursued in this thesis. The square boxes represent the main steps of our technique and indicate where we use the prover, while the rounded boxes represent the inputs and outputs.

2.1 High-Level Idea

Our approach consists of the following steps:

Extract Quantified Variables. Before we consider the unsat-proof, we examine the input and gather information about all quantified variables that appear in it. We memorize the name, which we assume to be unique, and the type of each quantified variable, as well as uninterpreted functions that use it as an argument. To determine if the input is satisfiable, the prover has to find, among others, an interpretation for each uninterpreted function. This is possible if exactly one element of the codomain can be assigned to each element of the domain. If the input imposes contradictory constraints on an uninterpreted function, e.g., through its quantified arguments, then it is not possible to satisfy this property. In that case the input is unsatisfiable.

Identify Quantifier Instantiations. Based on the information about all quantified variables we previously collected, we now identify explicit or implicit quantifier instantiations in the unsat-proof generated by one of the provers considered in this thesis (i.e., *Z3* with *MBQI* or *Vampire*). While explicit quantifier instantiations are marked as such, we rely on heuristics to identify implicit quantifier instantiations. Among others, we consider uninterpreted functions that use quantified variables as arguments as implicit quantifier instantiations. Concretely, we assume that the presence of a quantified variable as an argument of an uninterpreted function (e.g., $f(x_0)$) in the input combined with the presence of the same uninterpreted function with a

concrete value as an argument (e.g., $f(0)$) in the unsat-proof indicates that the quantified variable is implicitly instantiated with that concrete value (e.g., $x_0 = 0$).

Construct Potential EXAMPLE. Once we extracted the quantifier instantiations from the unsat-proof, we use them to generate a quantifier-free version of the input. That is,we instantiate some of the input formulas by substituting each occurrence of a quantified variable that has been instantiated with its concrete value(s). We further keep the necessary definitions that are needed to successfully parse this EXAMPLE as well as quantifier-free constraints.

Validate EXAMPLE. To validate a potential EXAMPLE, i.e., to investigate whether the quantifier-free version of the input is indeed unsatisfiable, we again use the prover. If it is unsatisfiable, then the EXAMPLE is sufficient to expose the contradiction from the input, and we proceed with the minimization step. This is because a contradiction that arises from formulas where universally quantified variables were replaced with concrete values of the same type is inevitably also present between the original quantified formulas. If it is satisfiable instead, then we did not find the necessary quantifier instantiations in the unsat-proof (we discuss potential reasons for this in Section 3.3.4), so we continue with the recovery step to eventually still succeed.

Recover EXAMPLE. After an unsuccessful validation, we again traverse the unsat-proof and apply further heuristics to generate additional concrete values for instantiating the quantified variables. We include them in our potential EXAMPLE and retry the validation.

Minimize EXAMPLE. The heuristics that we use for identifying implicit quantifier instantiations and during the recovery can cause us to instantiate quantified variables with concrete values that are not needed to expose the contradiction, all of which are included in our EXAMPLE so far. We want to remove these unnecessary quantifier instantiations as well as quantifier-free parts that are not needed, so that the EXAMPLE only contains the formulas that are minimally necessary to reveal the contradiction. This corresponds to the unsat-core of the EXAMPLE; we use *Z3* to generate it.

2.2 Walkthrough

We now illustrate the main steps of our technique using the input in Figure 6.

```
(declare-fun f (Int Int) Int)
```
(*in math notation:* $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$)

```
(assert (forall ((x0 Int) (x1 Int)) (=> (and (> x0 (- 0 1)) (> x1 2)) (> (f x0 x1) 7))))
```
(*in math notation:* $\forall x_0, x_1 \in \mathbb{Z} : (x_0 > -1) \wedge (x_1 > 2) \Rightarrow (f(x_0, x_1) > 7)$)

```
(assert (forall ((x2 Int) (x3 Int)) (=> (and (< x2 1) (< x3 4)) (= (f x2 x3) 6))))
```
(*in math notation:* $\forall x_2, x_3 \in \mathbb{Z} : (x_2 < 1) \wedge (x_3 < 4) \Rightarrow (f(x_2, x_3) = 6)$)

```
(assert (forall ((x4 Int) (x5 Int)) (= (f x4 x5) (f x5 x4))))
```
(*in math notation:* $\forall x_4, x_5 \in \mathbb{Z} : f(x_4, x_5) = f(x_5, x_4)$)

Figure 6: SMT formulas we use as input for illustrating the steps from the high-level overview of our approach in Figure 5.

Extract Quantified Variables. In the input we find a total of six quantified variables called $x_0, x_1, ..., x_5$, all of type `Int`. Furthermore, there is an uninterpreted function f that uses each of these quantified variables as its arguments.

Identify Quantifier Instantiations. The unsat-proof generated by *Z3* with *MBQI* explicitly instantiates the quantified variables x_0 and x_2 with 0 as well as x_1 and x_3 with 3, while x_4 and x_5 are not instantiated. Figure 7 shows an extract of the unsat-proof that instantiates x_0 and x_1. Although it is not directly visible, we explain in Section 3.3.2.2 why this is indeed an explicit quantifier instantiation in the form how *Z3* with *MBQI* presents them.

```
...
(let ((a!1 (forall ((x0 Int) (x1 Int)) (or (<= x0 (- 1)) (<= x1 2) (not (<= (f x0 x1) 7))))))
       (a!2 (<= (f (+ 1 (* (- 1) 1)) 3) 7)))
(let ((a!3 (or (not a!1) (<= (+ 1 (* (- 1) 1)) (- 1)) (<= 3 2) (not a!2))))
    (quant-inst a!3)))
...
```

Figure 7: Extract from the unsat-proof generated by *Z3* with *MBQI* for the input in Figure 6. It explicitly instantiates the quantified variables x_0 and x_1 with the concrete values 0 and 3, respectively (the explanation follows in Figure 20).

The unsat-proof generated by *Vampire*, on the other hand, contains no explicit quantifier instantiations. Instead, we use the heuristic described in Section 2.1 to identify the term $f(0, 3)$ that occurs syntactically in the extract of the unsat-proof shown in Figure 8 and implicitly instantiates some of the quantified variables. Namely, because the uninterpreted function f uses the quantified variables x_0 and x_2 as its first argument in the input, and now in the unsat-proof the concrete value 0 is at this position, we instantiate both with 0. For exactly the same reason, we instantiate the quantified variables x_1 and x_3 with 3. Although they are also used as arguments by f, we do not instantiate x_4 and x_5 because they do not occur in the unsat-proof.

```
1. ! [X1 : $int,X0 : $int] : (($greater(X0,$difference(0,1)) & $greater(X1,2))
       => $greater(f(X0,X1),7)) [input]
2. ! [X0 : $int,X1 : $int] : (($less(X0,1) & $less(X1,4)) => f(X0,X1) = 6) [input]
...
258947. f(0,3) = 6 [evaluation 258687]
...
```

Figure 8: Extract from the unsat-proof generated by *Vampire* for the input in Figure 6. It implicitly instantiates the quantified variables x_0 and x_2 with 0 and x_1 and x_3 with 3.

The unsat-proofs generated by *Z3* with *MBQI* and by *Vampire* therefore use different approaches to instantiate the same quantified variables with the same values.

Construct Potential Example. Based on the quantifier instantiations we identified in the previous step, we generate a quantifier-free version of the input, as shown in Figure 9. For each of the quantified variables we declare a free variable, to which we assign the corresponding concrete value. We then use these free variables to substitute the quantified variables in the input formulas. This detour via variables offers several advantages, one of which is the preservation of the information whether a concrete value in a formula from the EXAMPLE is actually the instantiation of a quantified variable, or whether it is already a concrete value in the input. Other advantages are more involved, which is why we postpone their discussion to Section 3.3.3.

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

(declare-const x0 Int)
(declare-const x1 Int)
(declare-const x2 Int)
(declare-const x3 Int)

(assert (= x0 0))
(assert (= x1 3))
(assert (= x2 0))
(assert (= x3 3))

(assert (=> (and (> x0 (- 1)) (> x1 2)) (> (f x0 x1) 7)))
```
(*in math notation:* $(x_0 > -1) \wedge (x_1 > 2) \Rightarrow (f(x_0, x_1) > 7)$)

```
(assert (=> (and (< x2 1) (< x3 4)) (= (f x2 x3) 6)))
```
(*in math notation:* $(x_2 < 1) \wedge (x_3 < 4) \Rightarrow (f(x_2, x_3) = 6)$)

Figure 9: Potential EXAMPLE for the input in Figure 6.

Validate Example. We again use the prover to verify that our EXAMPLE is unsatisfiable. This is the case, and since the corresponding formulas with concrete values for x_0, x_1, x_2 and x_3 impose contradictory constraints on the uninterpreted function f, this must also be the case for the original quantified formulas from the input, which is why the EXAMPLE indeed reveals the contradiction from the input.

Recover. This step is not needed, as we successfully validated the EXAMPLE.

Minimize Example. We generate an unsat-core for the EXAMPLE in Figure 9, as shown in Figure 10, which confirms that the EXAMPLE is already minimal.

```
(= x0 0)
(= x1 3)
(= x2 0)
(= x3 3)

(=> (and (> x0 (- 1)) (> x1 2)) (> (f x0 x1) 7))
(=> (and (< x2 1) (< x3 4)) (= (f x2 x3) 6))
```

Figure 10: Unsat-core generated by *Z3* with *MBQI* for the EXAMPLE in Figure 9. It includes all the formulas from the EXAMPLE, which is why this is already minimal.

3 Methodology

We start this section with a brief argument of why our technique is based on quantifiers (Section 3.1), and discuss our assumptions about the input formulas (Section 3.2). Then, we revisit the steps from Figure 5, but this time we go into depth and explain our technique in detail (Section 3.3). Finally, we discuss why a brute force approach cannot be applied for constructing the EXAMPLES (Section 3.4).

3.1 The Importance of Quantified Variables

The presence of quantifiers in our application domain of program verification is inevitable, since verifiers usually rely on them to encode the specification and the semantics of a programming language. This is also the reason why universally quantified variables are frequently responsible for the unsatisfiability of a set of SMT formulas. At the same time, the introduction of quantifiers and the resulting potentially infinite possible values for quantified variables is a main reason for the difficulty of solving the satisfiability problem for universally quantified SMT formulas. The most common approach of how solvers such as *Z3* deal with quantified variables is instantiation [7]. Although provers like *Vampire* operate in a fundamentally different way, they too resort to techniques of instantiation [5]. That is, the unsat-proofs generated by both provers considered in this thesis can identify concrete values for quantified variables to expose the contradiction, which is what our technique is based on.

3.2 Assumptions

Our technique makes the following assumptions about quantifiers in the input:
Unique Names. All quantified variables have unique names.
Universal Quantifiers. There are no existential quantifiers.

Making these assumptions does not compromise the generality of our technique, since any set of SMT formulas can always be preprocessed in such a way that they are fulfilled while keeping the satisfiability unchanged.

Figure 11 shows how to preprocess an input to satisfy our assumptions.

```
(assert (forall ((x Int) (y Int) (z Int)) (< (- (- x y) z) 13)))
(assert (exists ((x Int)) (= x 42)))
```

Step 1: Ensure unique names. ↓

```
(assert (forall (( x0 Int) ( x1 Int) ( x2 Int)) (< (- (- x0 x1 ) x2 ) 13)))
(assert (exists (( x4 Int)) (= x4 42)))
```

Step 2: Use Skolemization to remove the existential quantifier. ↓

```
(declare-const a Int)
(assert (forall ((x0 Int) (x1 Int) (x2 Int)) (< (- (- x0 x1) x2) 13)))
(assert (= a 42))
```

Figure 11: How to preprocess a set of SMT formulas to satisfy our assumptions. The math notation is omitted, since the meaning of the formulas is not relevant for preprocessing.

The first assumption about unique names allows us to identify the quantified variables in the unsat-proof directly by name. Otherwise, we would always have to trace back the references to find out from which input formula a quantified variable mentioned in a proof step originates. The second assumption about the non-existence of existential quantifiers allows us to treat all quantified variables equally, i.e., with instantiation. Thus, the comprehensibility and directness of our approach benefits from these assumptions.

3.3 Our Approach

We now explain our technique in detail. We first consider what information we extract from the input (Section 3.3.1) and how we identify quantifier instantiations in the unsat-proofs (Section 3.3.2). Next, we explain how we construct potential EXAMPLES (Section 3.3.3). We then present our validation mechanism (Section 3.3.4) and how we can possibly recover from an unsuccessful validation (Section 3.3.5). Afterwards, we discuss the minimization step (Section 3.3.6).

Throughout our explanations we use various inputs that often use quantified variables of the built-in types for integers and Booleans. However, our technique is not restricted to built-in types, but applies to arbitrary formulas containing also user-defined types.

3.3.1 Extract Quantified Variables

Due to our assumption of unique names (Section 3.2), syntactically extracting all quantified variables from the input is directly feasible without further complications. For collecting each uninterpreted function that uses quantified variables in its arguments, however, we must decide what to consider in case of nested function applications. We use the input in Figure 12 to illustrate the different scenarios.

```
(declare-fun f (Int) Int)
(declare-fun g (Int) Int)
(declare-fun h (Int Int) Int)
```
(*in math notation:* $f : \mathbb{Z} \to \mathbb{Z}, g : \mathbb{Z} \to \mathbb{Z}, h : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$)

```
(assert (forall ((x0 Int)) (> (f (g x0)) 0)))
```
(*in math notation:* $\forall x_0 \in \mathbb{Z} : f(g(x_0)) > 0$)

```
(assert (forall ((x1 Int)) (< (f (+ x1 1)) 0)))
```
(*in math notation:* $\forall x_1 \in \mathbb{Z} : f(x_1 + 1) < 0$)

```
(assert (forall ((x2 Int) (x3 Int)) (= (h x2 (+ x3 1)) 0)))
```
(*in math notation:* $\forall x_2, x_3 \in \mathbb{Z} : h(x_2, x_3 + 1) = 0$)

Figure 12: SMT formulas we use to illustrate what information we extract from the input.

The first assertion in Figure 12 contains the term $f(g(x_0))$. We associate the uninterpreted function g with the quantified variable x_0, since we take an occurrence of $g(c)$ with some concrete value c in the unsat-proof as an indication of an implicit quantifier instantiation of x_0 with c, as we already mentioned in Section 2.1 and we further elaborate on in Section 3.3.2. However, we do not associate the uninterpreted

function f with x_0, since according to our approach, an occurrence of $f(c)$ in the unsat-proof only suggests that $g(x_0)$ is equal to c, but this provides no information about x_0. Thus, for each occurrence of a quantified variable, it is sufficient to memorize the innermost uninterpreted function in whose arguments it occurs.

The second assertion in Figure 12 contains the term $f(x_1 + 1)$, which involves the built-in function for addition, i.e., $+$. In this case, an occurrence of $f(c)$ in the unsat-proof does not provide a possible concrete value for just the quantified variable x_1, but for $x_1 + 1$. Therefore, we not only associate f with x_1, but also memorize the term $f(x_1 + 1)$, so that we can syntactically extract the concrete value c for x_1 if a function application $f(c + 1)$ occurs in the unsat-proof.

The third assertion in Figure 12 contains the term $h(x_2, x_3 + 1)$ and therefore involves two quantified variables x_2 and x_3 in the arguments of the higher-arity uninterpreted function h. We associate h with x_2 and x_3 and remember the term $h(x_2, x_3+1)$ for both, so that we can syntactically extract the concrete values c and c' for x_2 and x_3, respectively, if a function application $h(c, c' + 1)$ occurs in the unsat-proof.

In summary, we syntactically identify all the uninterpreted functions that use as arguments expressions over quantified variables, where for an expression we consider the definition from Figure 13.

```
<expression>     =   <quantified variable>
                     | '('<unop> <expression>')'
                     | '('<binop> <expression> <constant>')'
                     | '('<binop> <constant> <expression>')'
                     | '('<binop> <expression> <expression>')'
<constant>       =   <built-in constant> | <free variable>
<unop>           =   '-' | 'not'
<binop>          =   '+' | '-' | '*' | 'and' | 'or' | '=' | '!=' | '<' | '<=' | '>' | '>='
```

Figure 13: Syntactic definition of an expression in SMT-LIB syntax, as we use it to find arguments of uninterpreted functions in the input, which is neither complete nor semantically correct. We use `|` to separate different options.

3.3.2 Identify Quantifier Instantiations

As explained in Section 3.1, provers may instantiate quantified variables with concrete values. We therefore first consider what a concrete value can be (Section 3.3.2.1), and afterwards we show how we can syntactically detect quantifier instantiations in unsat-proofs. Because our approach for this is prover-specific, we first address the unsat-proofs generated by *Z3* with *MBQI* (Section 3.3.2.2) and then those by *Vampire* (Section 3.3.2.3). Ideally, we could simply search for expressions of the form `quant-inst(x = c)`, where x is a quantified variable and c is a concrete value. Unfortunately, however, this is not the case for either of the unsat-proofs examined here.

Furthermore, we note that the identification of quantifier instantiations and the recovery of an unsuccessful validation (follows in Section 3.3.5) are the only prover-dependent parts of our technique. All other steps are generic.

3.3.2.1 Concrete Values

Intuitively, we would mainly consider built-in constants of the respective type as concrete values, e.g., 42 for integers and $false$ for Booleans. In fact, there are several terms that qualify as concrete values, which is why we define them as in Figure 14.

```
<concrete value> =   <built-in constant> | <free variable>  | '('<function> <concrete value>+')'
<function>       =   <built-in function> | <uninterpreted function>
```

Figure 14: Syntactic definition of a concrete value SMT-LIB syntax. We use `|` to separate different options and `+` to indicate a 1-or-more instance of the preceding item.

We now illustrate this definition with with concrete inputs and the corresponding unsat-proofs.

Figure 15 provides a scenario which shows that quantified variables can be instantiated with free variables from the input. The unsat-proof concludes that the *Barber of Seville* cannot exist by instantiating the quantified variables x_0 and x_1 with the free variable `barber` defined in the input.

Figure 16 goes a step further by providing an incorrect partial specifiaction of a linked list, for which the unsat-proof instantiates the quantified variable x_3 with the term `(next last)`, i.e., with an uninterpreted function that uses as argument a free variable from the input.

Finally, in the world described by Figure 17, the unsat-proof instantiates both quantified variables x_0 and x_1 with the fresh variable `elem!0` of the user-defined type `cat`.

```
(declare-sort man)
(there is a data type called man)
(declare-fun shave (man) man)
(there is a function shave that maps mans to each other, which denotes which man is shaved by whom)
(declare-const barber man)
(there is a dedicated instance of man called barber)
(assert (forall ((x0 man)) (=> (not (= (shave x0) x0)) (= (shave x0) barber))))
(in math notation: ∀x0 ∈ man : (shave(x0) ≠ x0) ⇒ (shave(x0) = barber))
(assert (forall ((x1 man)) (=> (= (shave x1) barber) (not (= (shave x1) x1)))))
(in math notation: ∀x1 ∈ man : (shave(x1) = barber) ⇒ (shave(x1) ≠ x1))
```

```
...
(let ((a!1 (forall ((x0 man)) (or (= (shave x0) x0) (= (shave x0) barber)) ))))
    (quant-inst (or (not a!1) (= (shave barber) barber) (= (shave barber) barber))))
...
(let ((a!1 (forall ((x1 man)) (or (not (= (shave x1) barber)) (not (= (shave x1) x1)))))))
(let ((a!2 (or (not a!1) (not (= (shave barber) barber)) (not (= (shave barber) barber)))))
    (quant-inst a!2)))
...
```

Figure 15: Encoding of the barber paradox in SMT-LIB syntax (above). The corresponding unsat-proof generated by *Z3* with *MBQI* instantiates the quantified variables x_0 and x_1 with the free variable `barber`, which is marked with color (below).

```
(declare-sort list_element)
(there is a data type called list_element)

(declare-fun next (list_element) list_element)
(declare-fun prev (list_element) list_element)
(there are two functions next and prev that map list_elements to each other)

(declare-const first list_element)
(declare-const last list_element)
(there are some dedicated instances of list_element called first and last)

(assert (forall ((x0 list_element)) (= (next (prev x0)) x0)))
(in math notation: ∀x0 ∈ list_element : next(prev(x0)) = x0)

(assert (forall ((x1 list_element) (x2 list_element))
(=> (= (next x1) x2)(= (prev x2) x1))))
(in math notation: ∀x1, x2 ∈ list_element: (next(x1) = x2) ⇒ (prev(x2) = x1))

(assert (forall ((x3 list_element)) (not (= (next last) x3))))
(in math notation: ∀x3 ∈ list_element : next(last) ! = x3))
```

```
...
(let ((a!1 (forall ((x3 list_element)) (not (= (next last) x3 )))))
(let ((a!2 (or (not a!1) (not (= (next last) (next last) )))))
    (quant-inst a!2)))
...
```

Figure 16: Incorrect partial specification of a linked list in SMT-LIB syntax (above). The last formula requires `last` to have no next `list_element`. But this implies that there is a `list_element` for which the uninterpreted function `next` cannot be interpreted, which is a contradiction. The corresponding unsat-proof generated by *Z3* with *MBQI* instantiates the quantified variable x_3 with `(next last)`, which is marked with color (below).

```
(declare-sort cat)
(there is a data type called cat)

(declare-fun alive (cat) Bool)
(there is a function alive that maps a cat to a boolean value, which denotes whether the cat is alive)

(assert (forall ((x0 cat)) (alive x0)))
(in math notation: ∀x0 ∈ cat : alive(x0))

(assert (forall ((x1 cat)) (not (alive x1))))
(in math notation: ∀x1 ∈ cat : ¬(alive(x1)))
```

```
...
(let ((a!1 (or (not (forall ((x0 cat)) (alive x0 ))) (alive elem!0 ))))
    (quant-inst a!1))
...
(let ((a!1 (not (forall ((x1 cat)) (not (alive x1 )))))))
    (quant-inst (or a!1 (not (alive elem!0 )))))
...
```

Figure 17: Encoding of a fact about cats in SMT-LIB syntax, which assumes that they are alive and not alive at the same time (above). The corresponding unsat-proof generated by *Z3* with *MBQI* instantiates both quantified variables x_0 and x_1 with a fresh variable `elem!0` of the user-defined type `cat`, which is marked with color and corresponds to *Schrödingers cat* that cannot exist (below).

3.3.2.2 Quantifier Instantiations in Z3

The unsat-proofs generated by *Z3* explicitly instantiate universally quantified variables with expressions that follow a particular structure [8], which we refer to as **Or-Not** form (named after its format in SMT-LIB syntax). We first explain this form for the simple case of a single quantified variable being instantiated, and then we generalize it to the case with multiple quantified variables.

If x_0 is a quantified variable that occurs in some formula φ, which we denote by $\varphi[x_0]$, then its instantiaton with the concrete value c is presented in the unsat-proof generated by *Z3* with the expression shown in Figure 18, where $\varphi[x_0 \mapsto c]$ denotes that every occurrence of x_0 in $\varphi[x_0]$ is replaced by c without further simplifying the formula φ.

```
(or (not (forall (x0) (φ[x0]))) (φ[x0↦c]))
(in math notation: ¬(∀x0 : φ[x0]) ∨ φ[x0 ↦ c])
```

Figure 18: The Or-Not form, according to the proof format as specified by *Z3* [9] and inspired by SMT-LIB, with which *Z3* instantiates a single quantified variable x_0, occurring in a formula φ, with the concrete value c.

Z3 may also simultaneously instantiate multiple quantified variables $x_0, x_1, ..., x_k$ that occur together in some formula φ, which we denote by $\varphi[x_0, x_1, ..., x_k]$. Their instantiation with the concrete values $c_0, c_1, ..., c_k$ presented in the unsat-proof generated by *Z3* as shown in Figure 19 then follows the same structure.

```
(or (not (forall (x0 x1 ... xk) (φ[x0, x1, ..., xk]))) (φ[x0↦c0, x1↦c1, ..., xk↦ck]))
(in math notation: ¬(∀x0, x1, ..., xk : φ[x0, x1, ..., xk]) ∨ φ[x0 ↦ c0, x1 ↦ c1, ..., xk ↦ ck])
```

Figure 19: The Or-Not form, according to the proof format as specified by *Z3* [9] and inspired by SMT-LIB, with which *Z3* instantiates multiple quantified variables $x_0, x_1, ..., x_k$, occurring in a formula φ, with the concrete values $c_0, c_1, ..., c_k$.

In practice, the formula φ can contain any number of nested function applications. We thus find the concrete value c_i with which x_i is instantiated in two steps. First, we search for an occurrence of x_i in $\varphi[x_i]$. Then we find c_i at the exact same position in $\varphi[x_i \mapsto c_i]$. This approach succeeds because *Z3* does not further simplify $\varphi[x_i \mapsto c_i]$ after replacing each occurrence of x_i by c_i, which is why the positions within the formula stay the same. And even if x_i occurs multiple times in $\varphi[x_i]$, it is sufficient for us to find it once, since the quantified variable is replaced everywhere with the same concrete value c_i in $\varphi[x_i \mapsto c_i]$. Figure 20 shows this procedure for the input in Figure 6 that we used in Section 2.2 for illustrating the main steps of our technique.

```
(let ((a!1 (forall ((x0 Int) (x1 Int)) (or (<= x0 (- 1)) (<= x1 2) (not (<= (f x0 x1) 7))))))
       (a!2 (<= (f (+ 1 (* (- 1) 1)) 3) 7)))
(let ((a!3 (or (not a!1) (<= (+ 1 (* (- 1) 1)) (- 1)) (<= 3 2) (not a!2))))
    (quant-inst a!3)))
```

Figure 20: Extract from the unsat-proof generated by *Z3* with *MBQI* for the input in Figure 6, which is split up into multiple parts but explicitly instantiates the quantified variables x_0 and x_1 according to the Or-Not form. To extract them, we locate an occurrence of x_i in the first part of the formula, and then we find the corresponding concrete value at the same position in the second part, which is marked with colors.

3.3.2.3 Quantifier Instantiations in Vampire

Although the unsat-proofs generated by *Vampire* can instantiate universally quantified variables with concrete values, there is no common method or structure for doing so; instead, the instantiations are implicitly embedded in superposition and resolution steps without appropriate marking. Due to this unspecified behaviour of *Vampire*, it is not possible to guarantee that we identify all quantifier instantiations in every case (as we can according to Section 3.3.2.2 for *Z3*). Our technique instead follows a best-effort approach based on heuristics.

Unsat-Proof Structure. Before presenting our heuristics for identifying implicit quantifier instantiations, we must elaborate on some special characteristics of unsat-proofs generated by *Vampire* that are relevant for us. First of all, they are structured as a derivation tree with $false$ at the root. Thus, the order in the printed unsat-proof we use is not from top to bottom, but is given by the premises at the end of each line. Furthermore, as we have already seen in Section 1.1, unsat-proofs generated by *Vampire* use different names for quantified variables than the input does, and if later steps treat multiple quantified variables as identical, they are unified earlier in the proof. In Figure 3, this resulted in the name $X0$ being used for both quantified variables x_0 and x_1, and with $X0$ we thus instantiated x_0 and x_1 simultaneously. Moreover, *Vampire* does not preserve the uniqueness of names from the input. Instead, the branches of the derivation tree can use the same names for different quantified variables. We therefore must maintain a dynamic mapping of names of quantified variables in the input to the corresponding name in the unsat-proof. We then use the names given by *Vampire* only for extracting quantifier instantiations from the unsat-proof, and in all other steps of our technique we use the names from the input.

Uninterpreted Functions. As introduced in Section 2.1, our main heuristic for identifying implicit quantifier instantiations is based on uninterpreted functions. We make use of the fact mentioned in Section 3.1 that the presence of quantified variables as arguments of an uninterpreted function can lead to contradictory constraints on the interpretation of that function, which is then often the cause for the unsatisfiability. In order to reveal these contradictory constraints, *Vampire* may apply the uninterpreted function to concrete values. In concrete terms, if we identify in the input a function application of some uninterpreted function f that uses a quantified variable x_i in its arguments, and then in the unsat-proof we identify a function application of the same function f that uses some concrete value c in its arguments instead of x_i, then we consider this as an implicit quantifier instantiation of x_i with c.

Inequalities. An unsat-proof generated by *Vampire* may prevent the instantiation of a quantified variable with a certain concrete value, so to speak, using the built-in function for inequality (i.e., $!=$). Since, according to our assumptions from Section 3.2, all quantified variables are universal, a contradiction arises directly if a certain concrete value cannot be used for the instantiation, which provides us another heuristic. In concrete terms, if we identify in the unsat-proof a formula of the shape $x_i != c$ or $c != x_i$, where x_i is a quantified variable and c a concrete value, then we consider this as an implicit quantifier instantiation of x_i with c.

Figure 21 provides an input and the corresponding unsat-proof generated by *Vampire*, which contains the formula $7 \; ! = X0$ and therefore implicitly instantiates the quantified variable $X0$ with the concrete value 7 according to this heuristic.

```
(declare-fun f (Int) Int)
(declare-fun g (Int) Int)
(in math notation: f : ℤ → ℤ, g : ℤ → ℤ)
(assert (forall ((x0 Int)) (not (= (f x0) 7))))
(in math notation: ∀x0 ∈ ℤ : (f(x0) ≠ 7))
(assert (forall ((x1 Int)) (= (f (g x1)) x1)))
(in math notation: ∀x1 ∈ ℤ : f(g(x1)) = x1)
```

```
1.  !  [X0 :  $int] :  ~f(X0) = 7 [input]
2.  !  [X0 :  $int] :  f(g(X0)) = X0 [input]
3.  !  [X0 :  $int] :  f(X0) != 7 [flattening 1]
4.  f(g(X0)) = X0 [cnf transformation 2]
5.  f(X0) != 7 [cnf transformation 3]
6.  7 != X0 [superposition 5,4]
7.  $false [equality resolution 6]
```

Figure 21: SMT formulas that set contradictory constraints if the quantified variables x_0 and x_1 are both instantiated with the concrete value 7 (above), and the corresponding unsat-proof generated by *Vampire* that, according to our heuristics, implicitly instantiates them by using an inequality (below).

Of course, it depends on the exact context in which the inequality occurs, so that it really can be an implicit quantifier instantiation, i.e., inequalities can also occur in the unsat-proof for other reasons than implicitly instantiating quantified variables.

Comparisons. Finally, we consider cases where the unsat-proof relates different quantified variables to each other and generate concrete values for one of them based on the concrete values we already extracted for the other. In concrete terms, if we identify in the unsat-proof a formula of the shape $x_i \circ \varphi[x_j]$ or $\varphi[x_j] \circ x_i$, where x_i and x_j are distinct quantified variables, φ is a formula containing x_j (and no other quantified variable), and $\circ$ is a relational operator such as $=, <, \leq, >$ or $\geq$, then we can use this constraint to generate concrete values for x_i based on the concrete values we have already extracted for x_j.

Figure 22 provides an input and the corresponding unsat-proof generated by *Vampire*, which relates the quantified variables $X1$ and $X0$ with the formula $X1 < X0$. First, we can use the previously presented heuristic based on inequalities to instantiate the quantified variable $X0$ with 42. Then, we could replace $X0$ with 42 in the formula $X1 < X0$ and use this constraint to generate a concrete value for $X1$.

However, the generation of such concrete values requires semantic reasoning, as replacing x_j with some concrete value c_j in the formula only implicitly generates a concrete value for x_i (i.e., $x_i \circ \varphi[x_j \mapsto c_j]$), which does not suffice for our technique. While technically our approach only uses syntactic reasoning, we nevertheless implemented parts of this heuristic. As we describe in Section 5.3.2.3, we support equalities of the form $x_i = x_j \pm c$, where c is a concrete value. We leave the further potential of this heuristic open for future work (Section 9).

```
(declare-fun f (Int) Int)
(in math notation: f : ℤ → ℤ)
(assert (forall ((x0 Int) (x1 Int)) (=> (= x0 42) (> x0 x1))))
(in math notation: ∀x0, x1 ∈ ℤ : (x0 = 42) ⇒ (x0 > x1))
```

```
1.  !  [X0 :  $int,X1 :  $int] :  (42 = X0 => $greater(X0,X1)) [input]
2.  !  [X0 :  $int,X1 :  $int] :  (42 = X0 => $less(X1,X0)) [theory normalization 1]
8.  ~$less(X0,X0) [theory axiom]
15.  !  [X0 :  $int,X1 :  $int] :  ($less(X1,X0) | 42 != X0 ) [ennf transformation 2]
16.  $less(X1,X0)  | 42 != X0  [cnf transformation 15]
18.  $false [equality resolution 8,16]
```

Figure 22: SMT formulas that set contradictory constraints if the quantified variable x_0 is instantiated with 42 and x_1 with some integer ≥ 42 (above), and the corresponding unsat-proof generated by *Vampire* that, according to our heuristics, implicitly instantiates them both with 42 by using an inequality and a comparison (below).

3.3.3 Construct Potential Example

Once we extracted the instantiations of quantified variables from the unsat-proof into some intermediate data structure, we can use them to construct a potential EXAMPLE. In principle, we want to replace all quantified variables that have been instantiated with their corresponding concrete values, directly in the formulas where they occur in the input, thus creating a quantifier-free version of the input. However, with help of several concrete inputs and special cases, we explain the difficulties in doing so and gradually arrive at the approach that finally works. As we assume that the quantifier instantiations have been extracted at this point, we omit showing the unsat-proofs for most of the following inputs, but directly provide the possible concrete values.

Detour via Variables. For the input in Figure 2 used in Section 1.1, we generate the potential EXAMPLE in Figure 23 according to the procedure described above.

```
(declare-fun f (Int) Int)
(assert (=> (>= 0 0) (= (f 0) 0)))
(in math notation: 0 ≥ 0 ⇒ f(0) = 0)
(assert (> (f 0) 0))
(in math notation: f(0) > 0)
```

Figure 23: First approach for constructing a potential EXAMPLE for the input in Figure 2. We directly replace the quantified variables in the input formulas with concrete values.

The fact that we do not show the input in Figure 2 again here presents the first problem, namely that we can no longer tell from the EXAMPLE in Figure 23 which parts are actually instantiated quantified variables, and which have always been concrete values. While this is not a problem for the validation, it is one for the user presentation that we address in Section 4.

Moreover, this approach becomes a problem for the validation as well once we encounter an input formula for which some of the quantified variables are instantiated and others are not. Then, we have no concrete values to replace the quantified variables that are not instantiated, so we cannot generate quantifier-free formulas.

The EXAMPLE in Figure 24 solves these problems by taking the detour via variables already mentioned in Section 2.2, i.e., for each quantified variable we declare a free variable, with which we replace it in the input formula. We then assign the concrete values for instantiating the quantified variables to the corresponding free variables.

```
(declare-fun f (Int) Int)
(declare-const x0 Int)
(declare-const x1 Int)
(assert (= x0 0))
(assert (= x1 0))
(assert (=> (>= x0 0) (= (f x0 ) x0 )))
(assert (> (f x1 ) 0))
```

Figure 24: Second approach for constructing a potential EXAMPLE for the input in Figure 2. We replace the quantified variables in the input formulas with free variables, to which we assign the concrete values instead of replacing them directly with the concrete values.

This procedure also simplifies the solution of the problems described next.

Quantifier-Free Parts. Figure 25 shows that we should include quantifier-free parts of the input in the EXAMPLE, as they may contribute to the contradiction as well.

```
(declare-fun f (Int) Int)
(assert (forall ((x0 Int)) (> (f x0) 0)))
```
(*in math notation:* $\forall x_0 \in \mathbb{Z} : f(x_0) > 0$)

```
(assert (= (f 0) 0))
```
(*in math notation:* $f(0) = 0$)

```
(declare-fun f (Int) Int)
(declare-const x0 Int)
(assert (= x0 0))
(assert (> (f x0) 0))
(assert (= (f 0) 0))
```

Figure 25: SMT formulas that generate a contradiction which includes quantifier-free parts (above), and the potential EXAMPLE that must include them as well (below).

No Quantified Variables. If the input contains no quantifiers, then we do not find any quantifier instantiations in the corresponding unsat-proofs. In this case, by keeping the quantifier-free parts, we trivially generate a potential EXAMPLE that is equal to the input and therefore also successfully validates.

Figure 26 shows such an input, together with the corresponding unsat-proofs generated by *Z3* with *MBQI* and by *Vampire*, and the EXAMPLE we generate for it.

Multiple Concrete Values. So far, we have only considered cases where for each quantified variable we find at most one instantiation in the unsat-proof. The heuristics described in Section 3.3.2.3 may however provide multiple concrete values for a quantified variable, and the input in Figure 27 even requires the instantiation of a quantified variable with multiple concrete values to expose the contradiction.

```
(declare-const x0 Int)
(declare-const x1 Int)
(assert (and (<= x0 x1) (> x0 x1)))
```
(*in math notation:* $(x_0 \leq x_1) \wedge (x_0 > x_1)$)

```
(let ((a!1 (and-elim (asserted (and (<= x0 x1) (> x0 x1))) (> x0 x1)))
      (a!2 (rewrite (= (> x0 x1) (not (<= x0 x1)))))
      (a!3 (not (<= (+ x0 (* (- 1) x1)) 0)))
      (a!6 (and-elim (asserted (and (<= x0 x1) (> x0 x1))) (<= x0 x1)))
      (a!7 (= (<= x0 x1) (<= (+ x0 (* (- 1) x1)) 0)))
      (a!9 (= (<= (+ x0 (* (- 1) x1)) 0) true)))
(let ((a!4 (rewrite (= (not (<= x0 x1)) a!3)))
      (a!8 (mp a!6 (rewrite a!7) (<= (+ x0 (* (- 1) x1)) 0))))
(let ((a!5 (mp a!1 (trans a!2 a!4 (= (> x0 x1) a!3)) a!3))
      (a!10 (trans (monotonicity (iff-true a!8 a!9) (= a!3 (not true)))
                   (rewrite (= (not true) false))
                   (= a!3 false))))
  (mp a!5 a!10 false))))
```

```
1.  $lesseq(x0,x1) & $greater(x0,x1) [input]
2.  ~$less(x1,x0) & $less(x1,x0) [theory normalization 1]
15.  $less(x1,x0) [cnf transformation 2]
16.  ~$less(x1,x0) [cnf transformation 2]
17.  $false [subsumption resolution 15,16]
```

```
(declare-const x0 Int)
(declare-const x1 Int)
(assert (and (<= x0 x1) (> x0 x1)))
```

Figure 26: SMT formulas that contain no quantifiers (above), together with the corresponding unsat-proofs generated by *Z3* with *MBQI* and by *Vampire* that contain no quantifier instantiations (middle), and the potential EXAMPLE that is equal to the input (below).

```
(declare-fun f (Int) Int)
(assert (forall ((x0 Int)) (> (f x0) (f (- x0 1)))))
```
(*in math notation:* $\forall x_0 \in \mathbb{Z} : f(x_0) > f(x_0 - 1)$)

```
(assert (= (f 0) 1))
(assert (= (f 2) 0))
```
(*in math notation:* $f(0) = 1$ *and* $f(2) = 0$)

Figure 27: SMT formulas that cause the unsat-proof to instantiate the quantified variable x_0 with two different values 1 and 2, which is also necessary to expose the contradiction.

To solve the problem of having multiple concrete values for a single quantified variable, one might first propose the idea of assigning all of them to the same free variable using disjunctions, as is done in Figure 28, where the term $(x_0 = 1) \vee (x_0 = 2)$ is supposed to provide a selection of assignments to the free variable x_0 associated with the quantified variable of the same name.

```
(declare-fun f (Int) Int)
(declare-const x0 Int)
(assert (or (= x0 1) (= x0 2)))
(assert (> (f x0) (f (- x0 1))))
(assert (= (f 0) 1))
(assert (= (f 2) 0))
```

Figure 28: First (erroneous) approach for constructing a potential EXAMPLE for the input in Figure 27, where the term $(x_0 = 1) \vee (x_0 = 2)$ provides a selection of values for the free variable x_0. The EXAMPLE is satisfiable, because $x_0 = 1$ satisfies all the assertions.

However, this erroneous approach leads to the satisfiability of the potential EXAMPLE in Figure 28, i.e., it does not expose the contradiction from the input formulas. As mentioned before, the quantified variable x_0 must be instantiated twice for exposing the contradiction, which is not feasible with only one free variable. Furthermore, as explained in Section 3.3.2.3, we might extract noisy information from the unsat-proof, i.e., our heuristics may identify false positive quantifier instantiations. If we then have a possible concrete value that would not reveal the contradiction, this leads to the EXAMPLE being satisfiable.

To prevent these problems, we use a different free variable for each possible concrete value that was extracted for a quantified variable, as is done in Figure 29, where we declare two free variables x_{00} and x_{01} that we both associate with the quantified variable x_0 and which we assign the possible concrete values to.[2] We also instantiate the corresponding input formula twice, once with each free variable.

```
(declare-fun f (Int) Int)
(declare-const x01 Int)
(declare-const x00 Int)

(assert (= x00 1))
(assert (= x01 2))

(assert (> (f x00 ) (f (- x00 1))))
(assert (> (f x01 ) (f (- x01 1))))
(assert (= (f 0) 1))
(assert (= (f 2) 0))
```

Figure 29: Second approach for constructing a potential EXAMPLE for the input in Figure 27, declaring two free variables x_{00} and x_{01} for the quantified variable x_0 from the input, and using each of them to instantiate it with a different concrete value in the input formula.

Consider Combinations. For the input in Figure 27, it was sufficient to instantiate the input formula twice, once with each of the two free variables. However, if an ipnut formula uses multiple quantified variables and we find multiple concrete values for some of them, as it is the case in Figure 30, it is no longer clear which of the corresponding free variables we should combine to instantiate the input formula.

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

(assert (forall ((x0 Int) (x1 Int)) (= (f x0 x1) (f x1 x0))))
(in math notation: ∀x0, x1 ∈ ℤ : f(x0, x1) = f(x1, x0))

(assert (not (= (f 2 3) (f 3 2))))
(in math notation: ¬(f(2, 3) = f(3, 2)))
```

Figure 30: SMT formulas that include two quantified variables x_0 and x_1 in one formula, and for which our main heuristic for identifying implicit quantifier instantiations in the unsat-proof generated by *Vampire* (as presented in Section 3.3.2.3) extracts the concrete values 2 and 3 for both quantified variables.

[2]To provide a better overview in the figures, we use short names for free variables that are associated with the same quantified variable. In practice we have to be careful that all free variables have unique names. For instance, x_0 and x_{01} are valid names for quantified variables according to our assumptions from Section 3.2, but they cause a problem if we use two free variables x_{00} and x_{01} for the quantified variable x_0 and only one with the same name for the quantified variable x_{01}.

In this case we must find exactly the combination of concrete values that causes the contradiction. This is why we instantiate the input formula with all possible combinations of the free variables associated with the corresponding quantified variables. Figure 31 shows how we do this for the input in Figure 30.

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

(declare-const x00 Int)
(declare-const x01 Int)
(declare-const x10 Int)
(declare-const x11 Int)

(assert (= x00 3))
(assert (= x01 2))
(assert (= x10 2))
(assert (= x11 3))

(assert (= (f x00 x10) (f x10 x00)))
(assert (= (f x00 x11) (f x11 x00)))
(assert (= (f x01 x10) (f x10 x01)))
(assert (= (f x01 x11) (f x11 x01)))

(assert (not (= (f 2 3) (f 3 2))))
```

Figure 31: Potential EXAMPLE for the input in Figure 30 that instantiates the quantified formula with all possible combinations of free variables.

Nested Quantifiers. Finally, we consider the case where not all quantified variables have the same scope. Figure 32 provides such a scenario, where we again have two quantified variables in one formula, but there is an inner and an outer quantifier.

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

(assert (forall ((x0 Int)) (or (= (f x0) 0) (forall ((x1 Int)) (< (f x0) (+ (f x1) 1))))))
(in math notation: ∀x0 ∈ ℤ : (f(x0) = 0) ∨ (∀x1 ∈ ℤ : f(x0) < f(x1) + 1))

(assert (= (f 0) 1))
(assert (= (f 1) 0))
(in math notation: f(0) = 1 and f(1) = 0)
```

Figure 32: SMT formulas that include a nested quantifier, and for which our main heuristic for identifying implicit quantifier instantiations in the unsat-proof generated by *Vampire* (as presented in Section 3.3.2.3) extracts the concrete values 0 and 1 for both quantified variables x_0 and x_1.

If we only consider quantifiers at the beginning of the input formula, we do instantiate the outer quantifier and therefore the quantified variable x_0 in Figure 32, but we miss to instantiate the quantified variable x_1, as is done in Figure 33.

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

(declare-const x00 Int)
(declare-const x01 Int)

(assert (= x00 0))
(assert (= x01 1))

(assert (or (= (f x00) 0) ( forall ((x1 Int)) ) (< (f x00) (+ (f  x1 ) 1)))))
(assert (or (= (f x01) 0) ( forall ((x1 Int)) ) (< (f x01) (+ (f  x1 ) 1)))))

(assert (= (f 0) 1))
(assert (= (f 1) 0))
```

Figure 33: First (erroneous) approach for constructing a potential EXAMPLE for the input in Figure 32, where we do not instantiate the inner quantifier.

While we can use the same approach for instantiating the quantified formula as before, we now must also identify nested quantifiers and instantiate them as well with all possible combinations of concrete values. Figure 34 correctly instantiates the inner quantifier and therefore the quantified variable x_1 as well.

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

(declare-const x00 Int)
(declare-const x01 Int)
(declare-const x10 Int)
(declare-const x11 Int)

(assert (= x00 0))
(assert (= x01 1))
(assert (= x10 1))
(assert (= x11 0))

(assert (or (= (f x00) 0) (< (f x00) (+ (f x10) 1))))
(assert (or (= (f x01) 0) (< (f x01) (+ (f x10) 1))))
(assert (or (= (f x00) 0) (< (f x00) (+ (f x11) 1))))
(assert (or (= (f x01) 0) (< (f x01) (+ (f x11) 1))))

(assert (= (f 0) 1))
(assert (= (f 1) 0))
```

Figure 34: Second approach for constructing a potential EXAMPLE for the input in Figure 32, where we correctly detect the nested quantifier and instantiate it accordingly.

Thus, we arrived at an approach that correctly constructs potential EXAMPLES. Whilst we refined it, our EXAMPLES have also increased in size. To accommodate this, we introduce the minimization step in Section 3.3.6, after which the EXAMPLES include only the formulas necessary to expose the contradiction.

To summarize, we perform the following steps to construct a potential EXAMPLE:

Quantifier-Free Parts. The input can contain declarations, which are used in instantiated formulas, and quantifier-free formulas, which may also contribute to the contradiction. We must include them in our EXAMPLES.

Detour via Variables. For each quantified variable, we create a free variable. If we extracted multiple concrete values for a single quantified variable from the unsat-proof, then we create as many free variables. We assign the corresponding concrete values to these free variables and use them to instantiate the input formulas.

Consider Combinations. If an input formula uses multiple quantified variables and we find multiple concrete values for some of them, then we instantiate the input formula multiple times. We consider all possible combinations of the corresponding free variables.

Nested Quantifiers. We also identify nested quantifiers and instantiate them by again considering all possible combinations of concrete values.

Finally, we want to note that a potential EXAMPLE may be empty if we did not extract any quantifier instantiations from the unsat-proof.

3.3.4 Validate Example

Once we have constructed a potential EXAMPLE, we need to check its correctness, that is, whether it can expose the contradiction from the input.

Due to the nature of our approach for constructing the EXAMPLES presented in Section 3.3.3, it must include the same contradiction that the unsat-proof exposes in the input formulas. As paraphrased in Section 2.1, a contradiction arising from formulas where universally quantified variables are replaced with concrete values of the same type is inevitably also present between the original quantified formulas. That is, a contradiction that is caused by some formula $\varphi[x_0 \mapsto c]$, where every occurrence of the quantified variable x_0 in $\varphi[x_0]$ is replaced by some concrete value c, is inevitably also caused by the formula $\forall x_0 : \varphi[x_0]$.

Checking whether an EXAMPLE contains a contradiction is then reduced to the familiar problem of showing whether a set of SMT formulas is unsatisfiable, for which we can again use the prover. Since our EXAMPLES are free of quantified variables, we are in a simpler problem class, which we expect the prover to solve very quickly.

Unsuccessful Validation. There are two possible reasons why the validation of a syntactically correct EXAMPLE can fail, firstly if the prover returns *unknown*, and secondly if it returns *sat*.

In the first case, the prover is incomplete and our algorithm terminates without generating an EXAMPLE that is guaranteed to expose the contradiction. Nevertheless, it can provide the potential EXAMPLE for the user to examine manually.

In the second case, our potential EXAMPLE does not expose the contradiction.[3] The only difference between the EXAMPLE and the input are the quantified formulas, some of which are instantiated in the EXAMPLE and some of which are omitted, because we did not find instantiations of the corresponding quantified variables in the unsat-proof. So if the input is unsatisfiable and the EXAMPLE is not, the reason must be erroneous or missing quantifier instantiations. In the following, we explain for each of the provers considered in this thesis why it can happen that we did not find the necessary quantifier instantiations in their unsat-proofs.

Z3. As stated in Section 3.3.2.2, we find all quantifier instantiations in an unsat-proof generated by *Z3* if we identify the corresponding explicit markers and then use the structure of the Or-Not form. Thus, the success of our technique using *Z3* with *MBQI* depends only on whether the unsat-proof performs all the necessary instantiations, and an unsuccessful validation therefore implies that this is not the case.

Vampire. If instead we extract implicit quantifier instantiations using heuristics from an unsat-proof generated by VAMPIRE, as explained in Section 3.3.2.3, then there is a gap between the information from the unsat-proof and that which we extract. On the one hand, our heuristics cause us to extract implicit quantifier instantiations that

[3] As mentioned in Section 3.3.3, the potential EXAMPLE may be empty if we did not extract any quantifier instantiations from the unsat-proof. *Z3* with *MBQI* and *Vampire* both return *sat* for empty inputs.

are actually not quantifier instantiations. On the other hand, we may miss actual implicit quantifier instantiations, because *Vampire* does not specify how exactly it instantiates quantifiers. We eliminate false positive quantifier instantiations using the minimization step we introduce in Section 3.3.6, but we cannot prevent missing quantifier instantiations without the presence of explicit quantifier instantiations. Also, as mentioned in Section 3.2, *Vampire* can perform quantifier instantiations, but due to the underlying technique this is not always the case in practice. We can see this in Figure 35, where we have an input and the corresponding unsat-proof generated by *Vampire*, which does not instantiate the quantified variables, because there are infinitely many possible values, all of them exposing the contradiction.

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

(assert (forall ((x0 Int)) (> (f x0) 0)))
(in math notation: ∀x0 ∈ ℤ : f(x0) > 0)

(assert (forall ((x1 Int)) (<= (f x1) 0)))
(in math notation: ∀x1 ∈ ℤ : f(x1) <= 0)
```

```
1.  !  [X0 :  $int] :  $greater(f(X0),0) [input]
2.  !  [X0 :  $int] :  $lesseq(f(X0),0) [input]
3.  !  [X0 :  $int] :  ~$less(0,f(X0)) [theory normalization 2]
4.  !  [X0 :  $int] :  $less(0,f(X0)) [theory normalization 1]
17.  ~$less(0,f(X0)) [cnf transformation 3]
18.  $less(0,f(X0)) [cnf transformation 4]
19.  $false [subsumption resolution 18,17]
```

Figure 35: SMT formulas for which the contradiction is revealed by instantiating the quantified variables x_0 and x_1 with the same, arbitrary value (above), and the corresponding unsat-proof generated by *Vampire* that, according to our heuristics presented in Section 3.3.2.3, contains no implicit quantifier instantiations (below).

So, there are cases where either the unsat-proof does not perform all the necessary quantifier instantiations, or where we cannot find all of them with the technique presented so far. Section 3.3.5 presents an approach, with which we can nevertheless possibly generate an EXAMPLE that successfully validates.

3.3.5 Recover Example

We saw in Section 3.3.5 that the validation of an EXAMPLE fails if we do not find the necessary quantifier instantiations in the unsat-proof. This may be because the unsat-proof does not use quantifier instantiations at all, but derives the contradiction in a different way. In order to nevertheless generate successful examples for some of these cases, we resort to various recovery heuristics.

We are aware that by doing so, we are moving away from our actual task of explaining unsat-proofs. While the concrete values that we have extracted through explicit and implicit quantifier instantiations appear in any case in proof steps related to the quantified variables, this is no longer the case here. That is, with the heuristics presented below, we can no longer guarantee that the unsat-proof uses the same concrete values for quantifier instantiations. Nevertheless, an EXAMPLE we generate this way can provide insights into the contradiction in the input formulas and is thus valuable. However, we mark accordingly that it was generated with the help of recovery methods and thus does not necessarily correspond to the unsat-proof.

Default Values. A simple solution is to associate different quantified variables with each other by declaring a fresh variable for each interpreted and user-defined type, which we consider as a concrete value for all quantified variables of that same type for which we identified no quantifier instantiation in the unsat-proof. For interpreted types, such as integers or Booleans, we could also use built-in constants as default values, such as 0 and *true*. But this would be more of a guess, so the approach with fresh variables corresponds closer to the behaviour of unsat-proofs. For the input in Figure 35 we thus generate the EXAMPLE in Figure 36, which successfully validates.

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

(declare-const x0 Int)
(declare-const x1 Int)

(declare-const c Int)

(assert (= x0 c))
(assert (= x1 c))

(assert (> (f x0) 0))
(assert (<= (f x1) 0))
```

Figure 36: Potential EXAMPLE for the input in Figure 35, where we use the heuristic with default values to instantiate both quantified variables with the same free variable c.

However, this heuristic does not guarantee success, as we can see in Figure 37, where using the same free variable for instantiating both quantified variables still generates a satisfiable EXAMPLE.

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

(assert (forall ((x0 Int)) (> (f x0) 0)))
```
(*in math notation:* $\forall x_0 \in \mathbb{Z} : f(x_0) > 0$)

```
(assert (forall ((x1 Int)) (=> (> x1 1337) (<= (f x1) 0))))
```
(*in math notation:* $\forall x_1 \in \mathbb{Z} : x_1 > 1337 \Rightarrow f(x_1) <= 0$)

Figure 37: SMT formulas for which the contradiction is revealed by instantiating the quantified variables x_0 and x_1 with the same, arbitrary value greater than 1337. Using a free variable c for instantiating both quantified variables does not expose the contradiction, because the constraint $c > 1337$ is not present.

Concrete Values from the Unsat-Proof. A brute force approach (as we present in Section 3.4) syntactically identifies all concrete values in the unsat-proof, and then it tries all the type-correct combinations of quantifier instantiations. This is not scalable. However, we can build on it by not only considering type-correctness, but by including the information we extracted from the unsat-proof. That is, on the one hand, for each quantified variable where we extracted an explicit quantifier instantiation, we also keep this and do not consider any further concrete values. On the other hand, we use the syntactic information we extracted from the unsat-proof according to Section 3.3.2.3 when identifying implicit quantifier instantiations to prune the search space, and thus we do not have to explore all the combinations.

Boundary Testing. If we have quantified variables of the type integer, then we can use syntactic constraints from the unsat-proof to extend the previous heuristic by possibly considering values in the range ± 1 of the concrete values.

The latter two heuristics lead to the extraction of the necessary concrete values from the unsat-proof in Figure 38 for the input in Figure 37. We consider the syntactically occuring constraint $1337 < X0$ and use the heurisitc inspired from boundary testing to instantiate the quantified variable $X0$ with 1338, which exposes the contradiction. This way we can generate an EXAMPLE that successfully validates.

The recovery step is thus a compromise between considering too many concrete values, which leads to an explosion of the search space, and considering too few values, which may not reveal the contradiction.

```
1.  !  [X0 :  $int] :  $greater(f(X0),0) [input]
2.  !  [X0 :  $int] :  ($greater(X0,1337) => $lesseq(f(X0),0)) [input]
3.  !  [X0 :  $int] :  ($less(1337,X0) => ~$less(0,f(X0))) [theory normalization 2]
4.  !  [X0 :  $int] :  $less(0,f(X0)) [theory normalization 1]
14.  $less(X1,$sum(X0,1)) | $less(X0,X1) [theory axiom]
17.  !  [X0 :  $int] :  (~$less(0,f(X0)) | ~$less(1337,X0)) [ennf transformation 3]
18.  ~$less(0,f(X0)) | ~$less(1337,X0) [cnf transformation 17]
19.  $less(0,f(X0)) [cnf transformation 4]
21.  ~$less(1337,X0) [resolution 18,19]
33.  $less(X1,1337) [resolution 14,21]
38.  $false [resolution 33,21]
```

Figure 38: Unsat-proof generated by *Vampire* for the input in Figure 37, which does not contain an implicit quantifier instantiation according to our heuristics form Section 3.3.2.3 due to the syntactic absence of any concrete value greater than 1337.

3.3.6 Minimize Example

The EXAMPLES we generate following the approach in Section 3.3.3 are not always minimal in the sense that they contain declarations and formulas that do not contribute to the contradiction from the input. One reason for this are false positive quantifier instantiations; another is the fact that provers give no guarantees on the minimality of the unsat-proofs they generate. However, the smaller the size of an EXAMPLE, the more the informative value and readability benefit, which is why we want to keep our EXAMPLES as small as possible.

This is reduced to the familiar problem of finding an unsat-core, for which we can again use *Z3* (*Vampire* cannot generate unsat-cores). The unsat-core contains a subset of the formulas from the EXAMPLE, i.e., assignments of concrete values to free variables and formulas that use these variables, but no declarations, as we can see in Figure 39.

We therefore only use the concrete values appearing in the unsat-core for instantiating the quantified variables, and remove all other possibilities we extracted from the unsat-proof or generated using recovery methods, as shown in Figure 40.

This concludes the detailed discussion of our technique for generating simple EXAMPLES that explain the contradiction derived in unsat-proofs generated by *Z3* with *MBQI* and by *Vampire*.

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

(declare-const x00 Int)
(declare-const x01 Int)
(declare-const x10 Int)
(declare-const x11 Int)

(assert (= x00 3))
(assert (= x01 2))
(assert (= x10 2))
(assert (= x11 3))

(assert (= (f x00 x10) (f x10 x00)))
(assert (= (f x00 x11) (f x11 x00)))
(assert (= (f x01 x10) (f x10 x01)))
(assert (= (f x01 x11) (f x11 x01)))

(assert (not (= (f 2 3) (f 3 2))))
```

```
(assert (= x00 3))
(assert (= x10 2))

(assert (= (f x00 x10) (f x10 x00)))

(assert (not (= (f 2 3) (f 3 2))))
```

Figure 39: Potential EXAMPLE generated by our technique (above) and the corresponding unsat-core generated by *Z3* with *MBQI* (below). To reproduce this result when using *Z3* with *MBQI* with the command line and generate a non-empty unsat-core, the formulas must be named, e.g., `(assert (! (= x00 3) :named A0))`.

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

(declare-const x0 Int)
(declare-const x1 Int)

(assert (= x0 3))
(assert (= x1 2))

(assert (= (f x0 x1) (f x1 x0)))

(assert (not (= (f 2 3) (f 3 2))))
```

Figure 40: Minimized version of the EXAMPLE in Figure 39.

3.4 Brute Force Baseline

A simpler alternative approach for finding possible quantifier instantiations just traverses the unsat-proof and collects every term that is a concrete value according to the definition from Section 3.3.2, which it then uses for instantiating each quantified variable of the corresponding type. So if the instantiated quantified variable occurs syntactically in the unsat-proof, this brute force approach is guaranteed to find it.

The big disadvantage here, however, is the large number of possible concrete values to instantiate the quantified variables, which leads to the generation of exponentially many instantiated formulas and very quickly becomes inefficient. The main advantage of our technique is thus the fact that in many cases we extract as many quantifier instantiations as necessary, but at the same time as few as possible. Moreover, in the cases where no recovery is applied (as explained in Section 3.3.5), we indeed explain the unsat-proof, whereas the brute force approach is mere guesswork and does not take quantifier instantiations performed by the unsat-proof into account at all.

4 User Presentation

We have discussed the contents of our EXAMPLES thoroughly in Section 3, but we have so far omitted how we present them to users. As our solution is meant to improve the user experience when working with unsat-proofs, we should not only consider whether the information provided by our EXAMPLES is sufficient to reveal the contradiction, as we did in Section 3.3.4, but it should also be presented to users in a way that meets their expectations without being too overwhelming or complicated.

To understand these expectations, we asked potential users by conducting a small survey within ETH Zürich among 19 people working with SMT solvers or theorem provers. We presented them the input in Figure 41, which contains a contradiction that is not directly visible because it includes formulas that are not relevant for proving *unsat*. We gave to the participants several possible outputs for our tool (shown in Figures 43 - 46) that are all correct, but differ in form, length, syntax, and content. We then let them decide for each output how useful they perceive it on a scale from 1 (not useful) to 10 (useful).

```
(declare-fun f (Int) Int)
(declare-fun g (Bool) Bool)
(declare-fun h (Int Bool) Int)
```
(*in math notation:* $f : \mathbb{Z} \to \mathbb{Z}, g : \mathbb{B} \to \mathbb{B}, h : \mathbb{Z} \times \mathbb{B} \to \mathbb{Z}$)

```
(assert (forall ((x0 Bool)) (g x0)))
```
(*in math notation:* $\forall x_0 \in \mathbb{B} : g(x_0)$)

```
(assert (forall ((x1 Bool) (x2 Int)) (=> (not (g x1)) (>= (h x2 x1) x2))))
```
(*in math notation:* $\forall x_1 \in \mathbb{B}, x_2 \in \mathbb{Z} : (\neg g(x_1)) \Rightarrow (h(x_2, x_1) \geq x_2)$)

```
(assert (forall ((x3 Int)) (=> (> x3 -1) (> (f x3) 7))))
```
(*in math notation:* $\forall x_3 \in \mathbb{Z} : (x_3 > -1) \Rightarrow (f(x_3) > 7)$)

```
(assert (forall ((x4 Int)) (=> (< x4 1) (= (f x4) 6))))
```
(*in math notation:* $\forall x_4 \in \mathbb{Z} : (x_4 < 1) \Rightarrow (f(x_4) = 6)$)

```
(assert (forall ((x5 Int) (x6 Bool)) (=> (g x6) (< (h x5 x6) x5))))
```
(*in math notation:* $\forall x_5, x_6 \in \mathbb{Z} : g(x_6) \Rightarrow (h(x_5, x_6) < x_5)$)

```
(assert (forall ((x7 Bool) (x8 Int) (x9 Int)) (=> (= (f x8) x9) (g x7))))
```
(*in math notation:* $\forall x_7 \in \mathbb{B}, \forall x_8, x_9 \in \mathbb{Z} : (f(x_8) = x_9) \Rightarrow g(x_7)$)

Figure 41: SMT formulas we presented to the participants of our user study.

The EXAMPLE that our technique generates for the input in Figure 41 is shown in Figure 42, but this was not displayed to the participants.

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

(declare-const x3 Int)
(declare-const x4 Int)

(assert (= x3 0))
(assert (= x4 0))

(assert (=> (> x3 (- 1)) (> (f x3) 7)))
(assert (=> (< x4 1) (= (f x4) 6)))
```

Figure 42: EXAMPLE for the input in Figure 41.

In the following, we discuss the possible outputs and present the results of the survey. We use *avg* to indicate the average number of points that the corresponding output received out of 10.

The outputs that performed worst among all participants, as shown in Figure 43, were those providing the least amount of information, which however would still be sufficient to expose the contradiction. From this we conclude that it is not a good idea to reduce the information to the most essential.

Output 9 ($avg = 2.74$):
```
f(0).
```

Output 10 ($avg = 4.16$):
```
x3 = 0, x4 = 0.
```

Figure 43: Minimal outputs. They received the worst ratings in terms of usefulness.

The outputs in Figure 44 do not mention any quantified variables or input formulas, but only the uninterpreted function f and the fact that no interpretation can be found for it due to contradictory constraints. They performed similarly badly.

Output 3 ($avg = 4.32$):
```
The prover cannot find an interpretation for f(0) due to contradictory constraints.
```

Output 4 ($avg = 4.37$):
```
The prover cannot find an interpretation for the uninterpreted function (f (Int) Int)
due to contradictory constraints for (f 0).
```

Output 5 ($avg = 4.89$):
```
The prover cannot find an interpretation for the uninterpreted function f: Int -> Int
due to contradictory constraints for f(0).
```

Figure 44: Outputs that only mention the uninterpreted function. They also received rather low ratings in terms of usefulness. Outputs 4 and 5 differ only in the syntax used for the declaration and application of f.

Figure 45 presents further outputs, some of which contain less and some more information, but all of them are in their own way somewhat more detailed than the options considered so far. Here, the opinions differ the most in terms of usefulness. Although all outputs from Figure 45 achieved a relatively high average score, there were individual users for each output who did not consider it particularly useful.

Output 1 ($avg = 6.79$):
```
Instantiating the quantified variables x3 = 0, x4 = 0 leads to a contradiction.
```

Output 8 ($avg = 7.16$):
```
The assertions require both f(0) = 6 and f(0) > 7, which is a contradiction.
```

Output 2 ($avg = 7.63$):
```
The following assertions set contradictory constraints for x3 = 0, x4 = 0:
(assert (forall ((x3 Int)) (=> (> x3 -1) (> (f x3) 7))))
  and
(assert (forall ((x4 Int)) (=> (< x4 1) (= (f x4) 6)))).
```

Figure 45: Outputs that are more verbose. They performed better on average, but for all options there were also participants who marked them as rather less useful.

Lastly, however, there are two very detailed outputs that clearly stood out as the most useful, as shown in Figure 46.

Output 6 ($avg = 7.74$):

```
Instantiating the quantified variables x3 = 0, x4 = 0 leads to the following contradiction:
(=> (> 0 -1) (> (f 0) 7))
  and
(=> (< 0 1) (= (f 0) 6)).
```

Output 7 ($avg = 8.58$):

```
Instantiating the quantified variables x3 = 0, x4 = 0 leads to the following contradiction:
(0 > -1 => f(0) > 7)
  and
(0 < 1 => f(0) = 6).
```

Figure 46: Outputs containing all the information from the EXAMPLE in Figure 42. They performed best among all participants, Output 7 even slightly better than Output 6, mainly due to the more human-readable syntax, which is the only difference between them.

From this we conclude that our potential users would rather receive more information than less. In fact, it was even the case that the participants in the study preferred to receive combinations of the options:

"*A combination of Output 8 and Output 6/7 would be perfect in my opinion. Output 8 has the nice, to the point description of the contradiction, but doesn't tell me from which constraints it derived it; if that is added it would be perfect.*"

"*It would be nice to get the quantifier instantiations and the fact that f(0) cannot be given a model together (instead of just one of the two).*"

"*It is important to know which constraints cause the contradiction and how the variables are instantiated. While preserving this information, the formulas should be simplified as much as possible.*"

These statements and the results of the rating suggest that we should provide both the quantifier instantiations and the formulas from the EXAMPLES to increase the overall comprehensibility, where applicable (when the number of formulas involved in the contradiction is not too high), and we should also show where quantified variables have been instantiated in the formulas. This reduces to a problem that we have already solved with the detour via variables, which we introduced in Section 3.3.3.

Based on the results of this small user study and the information we have from the unsat-proofs, we finally designed the presentation format shown in Figure 47. It is scalable to a certain extent, supports multiple instantiations of the same quantified variable and can be generated with purely syntactic information.

```
Instantiating the quantified variables x3 = [0], x4 = [0] leads to the following contradiction:
      (or (not (<= (f x3) 7)) (<= x3 (- 1)))
      (or (<= 1 x4) (= (f x4) 6))
The following input formulas were instantiated:
      (forall ((x3 Int)) (=> (> x3 (- 1)) (> (f x3) 7)))
      (forall ((x4 Int)) (=> (< x4 1) (= (f x4) 6)))
```

Figure 47: User presentation of the EXAMPLE for the input in Figure 41.

We have not yet included a translation of SMT-LIB syntax into a more human-readable format, but this can be implemented on top of our technique and is therefore covered in Section 9, which presents possible future work.

5 Implementation Details

This section discusses the implementation of our technique as a stand-alone tool, some of the challenges we faced, and how we solved them. We start by giving an overview of our implementation in Section 5.1 and present the preliminaries that it relies on in Section 5.2. Then, we once more revisit the steps from Figure 5 and describe how we implemented each of them in Section 5.3.

5.1 Overview

Figure 48 gives an overview of the main classes used in our implementation. It is divided into three parts:

Interactive Component. This is the part that the clients are exposed to.

Proof Analysis. This is the direct implementation of our technique.

Utility. This part is responsible for specifying the setup and I/O interaction, as well as syntactic extraction, syntax manipulation, and parsing.

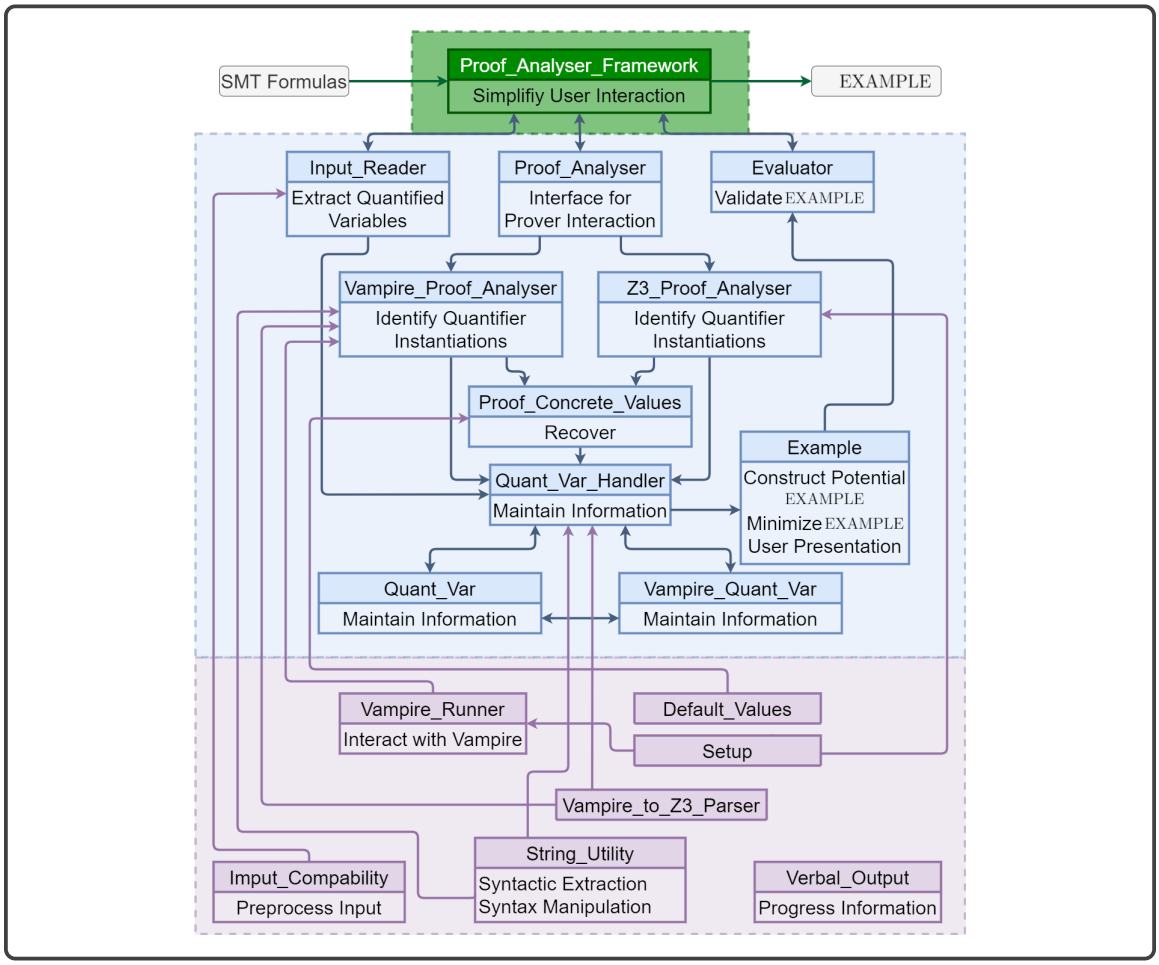


Figure 48: Overview of the main classes used in our implementation. It is divided into an interactive component, the direct implementation of our technique, and further utility methods. The square boxes represent classes that we implemented, while the rounded boxes represent the inputs and outputs. If the purpose of a class is not directly visible by its name, a few keywords follow to describe it.

5.2 Setup and Assumptions

Our implementation [10] is written in *Java*. We use *Z3* version 4.8.10 [11] with *Java* bindings (*Z3* API [9]), and *Vampire* version 4.5.1 with the command-line [12]. We expect the user to have the corresponding version of *Z3* and/or *Vampire* installed, depending on which prover(s) they want to use our tool with.

Based on the user's settings, we use either the *Z3* API or *Vampire* to generate the unsat-proofs. For the validation step, however, we always use the *Z3* API. Based on empirical evidence, there is little difference in the performance between the *Z3* API with *MBQI* and *Vampire* for inputs that use no universal quantifiers. Furthermore, for the minimization step we need an unsat-core, which only *Z3* can generate.

We read the input formulas in a file in SMT-LIBv2 syntax (.smt2), as both *Z3* and *Vampire* support SMT-LIBv2 inputs. Moreover, we assume the input to satisfy our assumptions from Section 3.2, i.e., all quantifiers are universal and quantified variables have unique names. If this is not the case, our implementation throws an exception during the phase that extracts quantified variables from the input.

For reproducibility, we fix the random seeds as follows: If an input file specifies random seeds, we adopt them, otherwise we use the ones we defined in the `Setup` class (`sat.random_seed` to 488, `smt.random_seed` to 599, and `nlsat.seed to 611`).

For the implementation to successfully terminate for a given input, we require the prover to generate an unsat-proof within the available resources, which instantiaties all quantified variables necessary for exposing the contradiction. For this we provide a time limit of 600 s and a memory limit of 6000 MB to the prover.

5.3 Our Approach

The aim of this section is to describe the interaction and tasks of the classes from Figure 48 and to explain how each of the steps of our technique from Figure 5 is implemented by them. Among others, it thus provides the necessary overview so that one could further extend the implementation.

This section is structured in the same way as the detailed explanation of our technique in Section 3.3. We first consider how we extract the information related to quantified variables from the input (Section 5.3.1) and how we identify quantifier instantiations in the unsat-proofs (Section 5.3.2). Next, we explain the approach that our implementation pursues to construct potential EXAMPLES (Section 5.3.3). We then discuss our validiation process (Section 5.3.4) and how we can possibly recover from an unsuccessful validation (Section 5.3.5). Finally, we present the implementation of the minimization step (Section 3.3.6).

5.3.1 Extract Quantified Variables

We use the `Input_Reader` to preprocess and parse the input file with the *Z3* API (as introduced in Section 5.2) and collect all information related to quantified variables and their occurrence within the input that we need for the subsequent steps.

Preprocessing. The *Z3* API differs from the command-line tool in that we cannot directly pass the input to the parser of the *Z3* API, as some of the configurations (e.g., `(set-option :smt.auto-config false)`) trigger an error. Our preprocessing step therefore removes these configurations from the input file, among others. We then set these configuration options directly in the *Z3* API.

If the user wants to use *Vampire* for generating the unsat-proof, we further preprocess the input so that *Vampire* succeeds as well in parsing it, since it does not support the entirety of SMT-LIBv2. We further explain this in Section 5.3.2.3.

Extract and Maintain Information. Once the *Z3* API has parsed the preprocessed input, we search for quantified expressions marked as `Z3_QUANTIFIER_AST`. We create a fresh `Quant_Var` object for each quantified variable we encounter to store its name and type, the input formula it appears in, and applications of uninterpreted functions that use it as an argument (as specified in Section 3.3.1). We use these `Quant_Var` objects in all steps of our approach through a `Quant_Var_Handler` object, which is a wrapper with methods for adding information to and also retrieving information from them.

5.3.2 Identify Quantifier Instantiations

The `Proof_Analyser` interface in Figure 49 captures the generality and modularity of our approach in the sense that any algorithm can be used to generate and analyse an unsat-proof, as long as the methods `prove()` and `collect_concrete_values()` are provided as specified. The `prove()` method implements the part of our main technique that extracts information about quantifier instantiations from the unsat-proof (as explained in Section 3.3.2). The `collect_concrete_values()` method implements the part of our recovery approach that extracts all syntactically appearing concrete values from the unsat-proof (as explained in Section 3.3.5).

```
public interface Proof_Analyser {

        // Uses a prover to generate an unsat-proof for the input.
        // Searches quantifier instantiations in there.
        // Throws a Proof_Exception if the prover cannot prove unsat within the resources
        // defined in the class Setup.
        // Returns a Quant_Var_Handler object that contains all the information extracted from
        // both the input and the unsat-proof.
        public Quant_Var_Handler prove() throws Proof_Exception;

        // Traverses the unsat-proof and collects all syntactically occurring concrete values,
        // which we can then consider as possible values for all quantified variables of the
        // same type.
        // Returns a Proof_Concrete_Values object that contains all these.
        // Assumes that the method prove has been called before.
        public Proof_Concrete_Values collect_concrete_values();

}
```

Figure 49: The `Proof_Analyser` interface, which specifies the requirements for a class to analyse unsat-proofs generated by any prover to be compatible with the other parts of our implementation.

In the following, we explain how we detect concrete values (Section 5.3.2.1). We then present the classes `Z3_Proof_Analyser` (Section 5.3.2.2) and `Vampire_Proof_Analyser` (Section 5.3.2.3), which generate and analyse the unsat-proofs of the respective prover.

5.3.2.1 Concrete Values

The concrete values we extract from the unsat-proofs always satisfy our syntactic definition from Figure 14.

For explicit quantifier instantiations we identify in unsat-proofs generated by the *Z3* API with *MBQI* (as we describe in Section 5.3.2.2), there is no need to check whether the concrete values satisfy our syntactic definition, since this is guaranteed by the correctness of *Z3*. If, independently of quantifier instantiations, we identify further concrete values in the unsat-proof, then we also use the definition of Figure 14, but with a subset of the built-in functions so that the search space does not become extremely large. This subset is defined in the `Z3_Proof_Analyser`.

For concrete values we identify in unsat-proofs generated by *Vampire*, we always use the `Vampire_to_Z3_Parser` (which we describe in Section 5.3.2.3). It is also based on our syntactic definition for concrete values.

5.3.2.2 Quantifier Instantiations in Z3

The `Z3_Proof_Analyser` uses the *Z3* API introduced in Section 5.2 to generate an unsat-proof for the input and identify explicit quantifier instantiations. If this succeeds, we search expressions marked as `Z3_OP_PR_QUANT_INST`. These have the Or-Not form introduced in Section 3.3.2.2 and we use the approach described there to extract the quantifier instantiations. We then give them to our `Quant_Var_Handler`, which maintains the information about quantified variables.

Enforce *MBQI*. We have empirically observed that *Z3*, although configured to use *MBQI*, does not do so in some cases and therefore cannot generate useful unsat-proof due to missing patterns (both with the command-line and with the API). We can force *Z3* to always use *MBQI* by using `(check-sat-using smt)` with the command-line, and in the API with a corresponding tactic.

De Bruijn Indexing. One of the main challenges we solve in the `Z3_Proof_Analyser` is the conversion from *de Bruijn* indexes so that we can find the corresponding quantified variables. Namely, the body of a quantifier instantiation does not directly contain the quantified variables, but local variables that reference the actual quantified variables with *de Bruijn* indexes. They denote the number of quantifiers that are in scope between the occurrence and the declaration of the corresponding quantified variable. Figure 50 shows a concrete example for this scenario.

In particular with nested quantifiers, implementing this translation is nontrivial, since the quantified variables have different scopes and are thus also defined in different expressions marked as `Z3_OP_PR_QUANT_INST`.

De-Brujn indexes become once more a challenge in Section 5.3.3, when we instantiate the input formulas to construct potential EXAMPLES.

```
(forall ((x1 Int) (x2 Int)) (or (<= x1 (- 1)) (<= x2 2) (not (<= (f x1 x2) 7)))))
```

```
(or (<= (:var 1) (- 1)) (<= (:var 0) 2) (not (<= (f (:var 1) (:var 0)) 7)))
```

Figure 50: Explicit quantifier instantiation from the unsat-proof generated by the *Z3* API with *MBQI* for the input in Figure 6 (above), and the body of it (below). Instead of quantified variables, the body uses local variables that reference the actual quantified variables with *de Brujn* indexes, as marked with colors.

5.3.2.3 Quantifier Instantiations in Vampire

The `Vampire_Proof_Analyser` uses *Vampire* with the command-line to generate an unsat-proof for the input. If this succeeds, we use the heuristics described in Section 3.3.2.3 to identify implicit quantifier instantiations. We extract them with the methods in `String_Utility`, which are based on regular expressions, and give them to our `Quant_Var_Handler`, which maintains the information about quantified variables.

Preprocessing. As mentioned in Section 5.3.1, *Vampire* does not support the entirety of SMT-LIBv2, which is why we preprocess the input once more. For instance, the unary minus operator `-x` is not supported by *Vampire*. We rewrite these occurrences to `(- 0 x)`. Other expressions that we rewrite for this reason are contained in the `vampirize` method of `String_Utility`.

Names of Quantified Variables. As explained in Section 3.3.2.3, unsat-proofs generated by *Vampire* use different names for quantified variables than the input does. Thus, we need an appropriate mapping. For this, we again use *Vampire* with the command-line to translate all input formulas individually into *Vampire* proof syntax, so we know which line marked as `[input]` in the unsat-proof corresponds to which input formula. Based on that, we can identify the names that *Vampire* uses for each quantified variable occurring in that line. Figure 51 presents the steps of our approach for this with a concret input.

Heuristics. Our implementation uses the heuristics presented in Section 3.3.2.3 only partially. It supports the heuristics with uninterpreted functions and inequalities in that they are considered if they can be successfully identified with help of regular expression and parsed by our `Vampire_to_Z3_Parser`. The heuristic based on comparisons is only partially implemented, namely we support equalities of the form $x_i = x_j \pm c$, where x_i and x_j are quantified variables, and c is a concrete value, as already mentioned in Section 3.3.2.3.

Parsing. We use the data structures from the *Z3* API to maintain the assignment of concrete values to quantified variables in our `Quant_Var_Handler` (regardless of which prover we use to generate the unsat-proof). After syntactically extracting the implicit quantifier instantiations from the unsat-proof, we must therefore parse the concrete values accordingly. For this we use the `Vampire_to_Z3_Parser`. It is by no means complete, but it can handle built-in constants, free variables defined in the input, and a subset of unary and binary operations, which suffices for the treatment of many concrete values. Figure 52 shows the syntactic extraction and parsing step by step using an example.

```
(declare-fun f (Int) Int)
(assert(forall ((x0 Int)) (=> (>= x0 0) (= (f x0) x0))))
(assert (forall ((x1 Int)) (> (f x1) 0)) )
...
```

Step 1: Choose an input formula and turn it into a formula that is always *unsat* (conjunction with `false`). ↓

```
(declare-fun f (Int) Int)
(assert (and (forall ((x1 Int)) (> (f x1) 0)) false))
```

Step 2: Run *Vampire* with this single-formula input. ↓

```
1.  !  [X0 :  $int] :  ($greater(f(X0),0) & false [input]
```

Step 3: Extract the relevant part from the unsat-proof for the single-formula input. ↓

```
($greater(f(X0),0)
```

Step 4: Find the best match in the lines marked as `[input]` from the unsat-proof for the initial input. ↓

```
1.  !  [X0 :  Int] :  ($greatereq(X0,0) => f(X0) = X0) [input]
2.  !  [X1 :  Int] :  ($greater(f(X1),0)  [input]
...
```

Step 5: Associate the corresponding quantified variables with each other. ↓

```
The quantified variable x1 from the input is called X0 in line 1 of the unsat-proof.
```

Figure 51: Steps for mapping the names of the quantified variables from an input formula to the corresponding names in the unsat-proof generated by *Vampire*. Note that not every quantified variable from the input necessarily occurs in the unsat-proof.

```
... f($uminus(1)) ...
```

Step 1: Syntactic extraction based on regular expressions with `String_Utility`. ↓

```
$uminus(1)
```

Step 2: Syntactic rewriting with `String_Utility`. ↓

```
-1
```

Step 3: Parsing to a Z3 expression with `Vampire_to_Z3_Parser`. ↓

```
Context.mkUnaryMinus(Context.mkInt(1))
```

Figure 52: Steps for syntactically extracting concrete values from unsat-proofs generated by *Vampire* and parsing them to an expression accepted by the *Z3* API.

5.3.3 Construct Potential Example

Our `Quant_Var_Handler` manages all quantified variables and the concrete values with which we want to instantiate them. The `Example` accesses the `Quant_Var_Handler` whenever we want to generate a potential EXAMPLE.

In the `Quant_Var_Handler` object, we use the *Z3* API to generate expressions for the declarations of free variables (e.g., `(declare-const x0 Int)`), the assignments of concrete values to free variables (e.g., `(= x0 0)`), and the instantiated input formulas, as explained in detail in Section 3.3.3. Along with these we also include declarations and quantifier-free constraints from the input in our EXAMPLE.

Nested Quantifiers. One of the main challenges we solve here is the instantiation of nested quantifiers. Namely, we must do this from the outside in, not from the inside out, since the inner quantifier may use the quantified variables from the outer one. Figure 53 shows the problem that arises when instantiating them starting form the inner quantifier.

```
(forall ((x0 Int)) (or (= (f x0) 0) (forall ((x1 Int)) (< (f x0) (+ (f x1) 1)))))
```

Step 1: Detect the inner quantifier. ↓

```
(forall ((x0 Int)) (or (= (f x0) 0) (forall ((x1 Int)) (< (f x0) (+ (f x1) 1))) ))
```

Step 2: Extract the body of the inner quantifier. ↓

```
(< (f (:var 1) (+ (f (:var 0) 1)))
```

Step 3: Instantiate the inner quantifier. ↓

```
(< (f (:var 1) (+ (f x1 1))))))
```

Figure 53: Wrong order for the instantiation of a nested quantifier. The *de Bruijn* index 1 of the local variable `(var: 1)` in the inner quantifier is converted to -1, because the inner quantifier declares only one quantified variable, which results in an array out-of-bounds.

5.3.4 Validate Example

The `Evaluator` checks whether a potential EXAMPLE generated by `Example` is unsatisfiable. For this, the `Input_Compability` saves the EXAMPLE as a file and then we use the *Z3* API again to parse and check its satisfiability.

5.3.5 Recover Example

If the EXAMPLE is satisfiable, then we use the heuristics described in Section 3.3.5, one after the other with increasing complexity, i.e., the more concrete values a recovery method is likely to identify or generate, the later we use it. We start with the default values provided by `Default_Values`. The recovery heuristics adjust the concrete values directly in our `Quant_Var_Handler`. Whenever we have applied a recovery heuristic, we generate a new EXAMPLE and try to validate it again.

If all recovery heuristics fail or the timeout or memory limit defined in `Setup` is reached, we return the most recently generated potential EXAMPLE, which is satisfiable due to the failed validation, but which still contains potentially useful information for the user.

5.3.6 Minimize Example

Once we have successfully validated an EXAMPLE, we use the *Z3* API to generate an unsat-core for it. We then keep in the `Quant_Var_Handler` only the quantifier instantiations that are also present in the unsat-core. Then, again using `Example`, we generate a minimized EXAMPLE, which we validate again and return if successful. If the minimized EXAMPLE cannot be successfully validated (which we have never encountered in our experiments), we return the non-minimized EXAMPLE.

6 Evaluation

To evaluate the success rate of our technique, we tested our implementation on different sets of *unsat* benchmarks of varying complexity.

Benchmarks. First, we used a set of manually created benchmarks, from which most of the inputs shown in the previous sections are taken (Test Set 1). While they are rather short and concise, they expose various edge cases and helped us examine different approaches taken by the provers. Next, we used a set of benchmarks from the figures in [13], some of which are already more involved, and which we have used as a basis for studying unsat-proofs since the beginning of the work on the thesis (Test Set 2). And finally, we used some more complex benchmarks from *Viper*'s test suite [14], from various unsound axiomiatations for *Gobra* [15], from *Dafny* [16], from *F** [17], and from the SMT-COMP suite [18]. These latter benchmarks were automatically preprocessed by and also used in the evaluation of [13]. All benchmarks that we used are available on [10].

Experimental Setup. We used our general setup from Section 5.2, with the only difference that we provided an overall timeout of 600 s per input. The experiments with *Z3* were performed on a Windows computer with 32 GB of RAM and an AMD Ryzen 7 3800X 8-core processor. The experiments with *Vampire* were performed on the same machine, but inside an Ubuntu virtual machine with only 8 GB of RAM and 4 cores. For each of the experiments, we removed the benchmarks for which the prover cannot generate an unsat-proof within the given resources on our machine (according to empirical evidence).

Table 1 shows the results of our experiments.

Benchmarks	**#**	**#F**	**#∀**	***Z3 MBQI***				***Vampire***			
				#OK	**#✓**	#MIN	#REC	#OK	**#✓**	#MIN	#REC
Test Set 1	40	2	3	40	**40**	3	0	40	**39**	20	20
Test Set 2	25	4	6	20	**20**	4	0	21	**17**	9	11
Viper	15	118	408	9	**7**	6	0	15	**0**	0	0
Gobra	11	69	242	6	**6**	6	0	11	**0**	0	0
Dafny	4	11	23	1	**1**	1	0	1	**0**	0	0
F*	2	1202	4092	0	**0**	0	0	0	**0**	0	0
SMT-COMP	61	519	673	30	**20**	4	0	52	**6**	0	0
				106	**94**	24	0	140	**62**	29	31

Table 1: Results of our experiments to evaluate our implementation on different sets of benchmarks. The columns, from left to right, show: the source of the benchmarks, the number of benchmarks from each source (**#**) and their average number of input formulas (#F) and quantifiers (#∀), and then, for each prover, the number of benchmarks that we did not remove (**#**OK), the number of successfully validated EXAMPLES generated by our implementation (**#✓**), the number of successful EXAMPLES that required minimization (**#**MIN), and the number of successful EXAMPLES that required recovery (**#**REC).

In the following, we discuss the consequences of these results as well as limitations for our implementation with *Z3* (Section 6.1) and with *Vampire* (Section 6.2).

6.1 Discussion of the Evaluation with Z3

When using *Z3* with *MBQI* to generate the unsat-proofs, we were successful for all the smaller benchmarks and for about $\frac{3}{4}$ of the complex benchmarks. Thus, our implementation clearly performs better with *Z3* than with *Vampire*.

Technique vs. Implementation. Our implementation did not generate a potential EXAMPLE that was satisfiable for any of the benchmarks. Instead, there were only unsuccessful outcomes due to timeouts. This means that we did not encounter any input during the experiments for which our technique using *Z3* with *MBQI* failed. Based on our empirical results, we can conclude that *MBQI* performs all the necessary quantifier instantiations in the unsat-proofs it generates.

Minimization. We were initially surprised to observe that EXAMPLES generated by only considering explicit quantifier instantiations were not always already minimal. However, we then discovered that the EXAMPLES generated for the simple benchmarks could only be further minimized if the contradiction already occurs by assigning concrete values to a subset of the free variables in an instantiated formula. *Z3* with *MBQI* instantiates all quantified variables of an input formula simultaneously, which is why, as for the input in Figure 54, sometimes not all of them are necessary.

```
(declare-fun f (Bool Bool) Bool)
(in math notation: f : 𝔹 × 𝔹 → 𝔹)
(assert (forall ((x0 Bool) (x1 Bool)) (and x0 (f x0 x1))))
(in math notation: ∀x0, y0 ∈ 𝔹 : x0 ∧ f(x0, x1))
```

```
(let ((a!1 (forall ((x0 Bool) (x1 Bool))
        (not (or (not x0 ) (not (f x0 x1 )))))))
        (a!2 (not (or (not false ) (not (f false false ))))))
    (quant-inst (or (not a!1) a!2)))
```

Figure 54: SMT formulas (above) and an extract from the corresponding unsat-proof generated by *Z3* with *MBQI*, for which our implementation requires minimization (below). The unsat-proof explicitly instantiatiates both quantified variables with $false$, which is marked with colors. The potential EXAMPLE can be minimized because the instantiation of the quantified variable x_0 with $false$ already exposes the unsatisfiablity regardless of x_1.

Recovery. Recovery was not needed for any of the benchmarks.

Limitations. As mentioned above, our implementation is not performant enough for long unsat-proofs, as we have reached the time limit for some of the complex benchmarks without generating a potential EXAMPLE. We leave the optimization of our implementation open for future work in Section 9.

Furthermore, we observed an extreme difference in performance between the *Z3* API and *Z3* with the command-line under equal conditions (i.e., with the same random seeds and configurations). For some of the benchmarks, the *Z3* API did not manage to generate an unsat-proof within the time limit specified in Section 5.2, while *Z3* with the command-line succeeded in a very short time.

6.2 Discussion of the Evaluation with Vampire

When using *Vampire* to generate the unsat-proofs, we were successful for many of the smaller benchmarks, but only for very few of the complex benchmarks.

Technique vs. Implementation. As explained in Section 5.3.2.3, the syntactic identification and extraction of implicit quantifier instantiations is based on regular expressions and an incomplete parser, which both do not fully capture the heuristics described in Section 3.3.2.3. That is, not every implicit quantifier instantiation is identified by the regular expressions, and not every expression that is extracted can be parsed. So there are cases where our technique would work, but the implementation does not in its current state.

Recovery. Recovery was used in relatively many cases, which was also to be expected when only using implicit quantifier instantiations presented in Section 3.3.2.3. In fact, for the simple benchmarks, each of the recovery heuristics from Section 3.3.5 was successfully applied at least once, which means that they do theoretically work. Table 1 shows, however, that the recovery approaches were never successfully applied to the complex benchmarks. The reason for this is that the simpler heuristics are not sufficient, and the more involved ones quickly lead to a search space explosion.

Minimization. Minimization was also often needed, mostly in combination with recovery. This underlines how the two steps together can generate a successfully validated EXAMPLE even if we do not find implicit quantifier instantiations.

Limitations. The obvious explanation for the not so good results would be that the heuristics described in Section 3.3.2.3 are not sufficient for extracting all implicit quantifier instantiations. Therefore, the question arises whether using a single additional heuristic would lead to much better results. The problem, however, is rather that for certain inputs, applying the heuristics one after the other is not sufficient to find the required quantifier instantiations. Instead, we would need combinations of different heuristics for identifying implicit quantifier instantiations.

Figure 55 provides such an input and the corresponding unsat-proof generated by *Vampire*. To expose the contradiction, we must instantiate the quantified variable x_0 with some integer c and x_1 with $g(c) - 1$, where g is an uninterpreted function declared in the input. The unsat-proof contains neither function applications of g with concrete values as arguments nor inequalities with the quantified variables, which is why our heuristics for identifying quantifier instantiations from Section 5.3.2 are initially unsuccessful. The recovery heuristic based on default values from Section 3.3.5 then instantiates all quantified variables with the same fresh variable c_fresh of type integer, but this does not expose the contradiction. To actually succeed, we would have to extract $g(X0)$, instantiate $X0$ with the fresh variable c_fresh, and use the (not yet implemented) comparison heuristic for $<$ to generate $g(c\_fresh) - 1$. Even if we fully integrate the corresponding heuristics in our implementation, jumps between the steps for identifying quantifier instantiations and recovery would be necessary to generate a corresponding concrete value. At the same time, these combination of heuristics would also extract noise, which would quickly lead to a state space explosion for bigger unsat-proofs. For the other benchmarks this looks similar.

```
(declare-fun f (Int) Int)
(declare-fun g (Int) Int)
(declare-fun h (Int Int) Int)
```
(*in math notation:* $f : \mathbb{Z} \to \mathbb{Z}, g : \mathbb{Z} \to \mathbb{Z}, h : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$)

```
(assert (forall ((x0 Int)) (> (f (g x0)) 0)))
```
(*in math notation:* $\forall x_0 \in \mathbb{Z} : f(g(x_0)) > 0$)

```
(assert (forall ((x1 Int)) (< (f (+ x1 1)) 0)))
```
(*in math notation:* $\forall x_1 \in \mathbb{Z} : f(x_1 + 1) < 0$)

```
(assert (forall ((x2 Int) (x3 Int)) (= (h x2 (+ x3 1)) 0)))
```
(*in math notation:* $\forall x_2, x_3 \in \mathbb{Z} : h(x_2, x_3 + 1) = 0$)

```
1.  !  [X0 :  $int] :  $greater(f(g(X0)),0) [input]
2.  !  [X0 :  $int] :  $less(f($sum(X0,1)),0) [input]
4.  !  [X0 :  $int] :  $less(0,f(g(X0))) [evaluation 1]
6.  $sum(X0,$sum(X1,X2)) = $sum($sum(X0,X1),X2) [theory axiom]
7.  $sum(X0,0) = X0 [theory axiom]
9.  0 = $sum(X0,$uminus(X0)) [theory axiom]
10.  ~$less(X0,X0) [theory axiom]
11.  ~$less(X1,X2) | ~$less(X0,X1) | $less(X0,X2) [theory axiom]
15.  $uminus($uminus(X0)) = X0 [theory axiom]
19.  $less(f($sum(X0,1)),0) [cnf transformation 2]
20.  $less(0,f(g(X0))) [cnf transformation 4]
21.  0 = $sum($uminus(X0),X0) [superposition 9,15]
64.  $less(X0,f(g(X1))) | ~$less(X0,0) [resolution 11,20]
144.  $less(f($sum(X27,$sum(X28,1))),0) [superposition 19,6]
166.  ~$less(f(g(X0)),0) [resolution 64,10]
176.  $less(f($sum(X8,0)),0) [superposition 144,21]
182.  $less(f(X8),0) [forward demodulation 176,7]
183.  $false [resolution 182,166]
```

Figure 55: SMT formulas that we already used in Figure 12 (above) and the corresponding unsat-proof generated by *Vampire* (below), for which our implementation cannot generate an EXAMPLE that successfully validates.

7 Build the Bridge to E-Matching

E-Matching performs a controlled number of quantifier instantiations based on patterns provided in the input to stay time-efficient. A failure of *E-Matching* (i.e., it returning *unknown*), as it was the case for the *Bob* scenario in Section 1.1, is therefore often caused by a lack of triggering terms that would be necessary to discover the contradiction. To prevent this, we use the information our technique extracts from the unsat-proofs to include further formulas for the input that only syntactically match patterns to trigger the necessary quantifier instantiations.

We pursue an approach that is otherwise applied manually and in a sense corresponds to the inverse of an implicit quantifier instantiation. We use applications of uninterpreted functions with quantified arguments from the patterns (e.g., $f(x_0, x_1)$), where we replace the quantified variables with their instantiations (e.g., $f(0, 3)$). In order to include them in the input without changing its satisfiability, we declare a fresh uninterpreted *dummy* function that takes the function application as an argument, and to which we impose no further constraints.

Figure 56 shows how we used this approach to automatically modify the input from Figure 6 so that *E-Matching* succeeds in proving *unsat*.

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

(assert (forall ((x0 Int) (x1 Int))
        (! (=> (and (> x0 (- 0 1)) (> x1 2)) (> (f x0 x1) 7)) :pattern (f x0 x1))))
(assert (forall ((x2 Int) (x3 Int))
        (! (=> (and (< x2 1) (< x3 4)) (= (f x2 x3) 6)) :pattern (f x2 x3))))
(assert (forall ((x4 Int) (x5 Int)) (= (f x4 x5) (f x5 x4))))

(declare-fun dummy0 (Int) Bool)
(assert (dummy0 (f 0 3)))
```

Figure 56: SMT formulas from Figure 6 (with patterns), where we automatically added a *dummy* function and a further formula to trigger the quantifier instantiations necessary for *E-Matching* to succeed in proving *unsat*.

At the time of writing, the implementation is not yet mature enough to be generally used, but conceptually the approach is successful. We leave the further development and evaluation of this approach open for future work (Section 9).

8 Related Work

The idea behind this thesis, i.e., extracting information from unsat-proofs to explain the contradictions derived in them, has to the best of our knowledge not yet been pursued. Research in this area does not generally address the problems of simplifying unsat-proofs and making them human-readable. Instead, there is ample work on the correctness of unsat-proofs. In addition, there have been efforts to develop a uniform format for SMT proofs.

Proof Checking. Unsat-proofs are first and foremost certificates of the correctness of a prover's *unsat* answer, which can be used to automatically check the correctness of the prover. Such proof checkers are the main topic of current research [19, 20]. A major problem with unsat-proofs is their length, which, as we have observed, is often significantly higher than the length of the input. This imposes several challenges on proof checkers. Moskal [21] presents a technique that encodes unsat-proofs and thus enables faster proof checking.

Proof Formats. Barret, de Moura and Fontaine [22] present the approaches of various SMT solvers to proof production as well as the challenges this poses for a generic proof format in order to prevail. We also observed that the unsat-proofs generated by *Z3* and by *Vampire* are completely different, one of the reasons being the disparate underlying technique of the respective provers. A general proof format should be flexible enough to support different proof rules of the various provers and at the same time simple enough to actually provide an advantage. Besson, Frédéric and Fontaine [23] propose such a generic proof format, whose aim is to simplify proof checking as much as possible. For our purposes, however, this would be of no use, as it does not cover quantifiers and instantiations. Böhme and Weber [24] discuss design choices for proof formats, with the focus once more on checkability, but they also take human readability into account. Despite the existing efforts, however, no general format has yet become established. Manually extracting the information from the unsat-proofs thus requires expert, tool-specific knowledge. Our technique automates this process for two state-of-the-art provers, *Z3* and *Vampire*, and presents the EXAMPLES in a simple, common format.

9 Future Work

In this section, we present a list of possible extensions for the work done in this thesis together with some further explanations and possible approaches for each of them.

Generalize Implementation. Our implementation does not fully capture all the heuristics for extracting implicit quantifier instantiations from unsat-proofs generated by *Vampire* that we presented in Section 3.3.2.3. As described in Section 6.2, there is still room for improvement for syntactic identification and extraction of implicit quantifier instantiations, and also the parser could be extended. Moreover, one could include further semantic reasoning for finding concrete values.

Optimize Implementation. The evaluation (Section 6) showed that our implementation is not particularly fast for larger inputs, as we have reached the time limit for some of the complex benchmarks without generating a potential EXAMPLE. Since performance was not our main concern, optimizing the implementation would, according to our expectation, lead to the generation of successfully validated EXAMPLES for these complex benchmarks when using *Z3* with *MBQI*.

Better Output Format. Many potential users of our tool indicated that they would prefer a more familiar format for the EXAMPLES than SMT-LIB syntax (Section 4). For this purpose, one could use the work from [25] to translate SMT-LIB syntax into a human-friendly notation and integrate it with our tool.

Build the Bridge to *E-Matching*. In Section 7 we established the groundwork for an automatic generation of triggering terms for *E-Matching*. One could further develop this technique and evaluate whether it is successful for complex benchmarks.

Include Additional Provers. Our technique and its implementation are designed so that only the steps that extract information from the unsat-proof are prover-dependent, as we describe in Sections 5.3.2 and 5.3.1. Thus, one could without too much effort add another prover that instantiates quantifiers in its unsat-proofs, which possibly increases the applicability of our tool.

Integration into *Viper*. *Viper* is a language and suite of tools developed at ETH Zürich, providing an architecture on which new verification tools and prototypes can be developed simply and quickly [14]. One could extend *Viper* by integrating our work. However, since the integration of our implementation into a verifier would involve significant effort, one should consider beforehand what impact this would have and what challenges they might have to overcome. If a set of SMT formulas generated by *Viper* is unsatisfiable, then either the specification provided by the user contradicts an internal axiomatisation, or the axiomatisation is even self-contradictory. The contradiction may therefore involve formulas that are not known to the user, and so an EXAMPLE generated with our technique is not necessarily meaningful for them. Moreover, all the encodings that *Viper* generates are based on *E-Matching*, meaning that they include patterns on which the soundness depends as well, but these are ignored by *MBQI* and by *Vampire*. Therefore, one might first have to find a way to extend these algorithms to consider them after all.

10 Conclusions

The main goal of this thesis was to syntactically extract quantifier instantiations from unsat-proofs generated by *Z3* with *MBQI* and by *Vampire* and use them to explain to users why their input formulas are unsatisfiable. We have developed a comprehensive technique for this purpose. An important part of it is the validation whether the EXAMPLES we generate really satisfy this requirement, i.e., whether they actually expose the unsatisfiability. To ensure this, we use an elegant approach, some parts of which we have implemented ourselves and others we use the prover for.

While working on this thesis, we studied quantifier instantiations in unsat-proofs in detail. We recorded the findings in this written report. Since quantifier instantiations in the unsat-proofs we encountered are neither always sufficient nor always minimal, we have introduced approaches for minimization and recovery.

Our solution is intended to improve the user experience when working with unsat-proofs, which is why we identified their expectations through a survey and designed a suitable presentation format based on its results.

We present a theoretical solution, as well as its implementation. We evaluated our implementation on manually and automatically generated benchmarks, including minimal inputs consisting of single formulas up to complex benchmarks from SMT-COMP.

References

[1] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[2] Yeting Ge and Leonardo de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 306–320, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[3] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[4] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[5] Giles Reger, Martin Suda, and Andrei Voronkov. Instantiation and pretending to be an smt solver with vampire. *CEUR Workshop Proceedings*, 1889, January 2017. 15th International Workshop on Satisfiability Modulo Theories, SMT 2017 ; Conference date: 22-07-2017 Through 23-07-2017.

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, (visited on 22.08.2021).

[7] Andrew Reynolds. Conflicts, models and heuristics for quantifier instantiation in smt. In Laura Kovacs and Andrei Voronkov, editors, *Vampire 2016. Proceedings of the 3rd Vampire Workshop*, volume 44 of *EPiC Series in Computing*, pages 1–15. EasyChair, 2017.

[8] Sascha Böhme. Proving theorems of higher-order logic with smt solvers. PhD thesis, TU München, 2011.

[9] Z3 java api. `https://z3prover.github.io/api/html/namespacecom_1_1microsoft_1_1z3.html`, (visited on 22.08.2021).

[10] Pascal Strebel. Explaining unsatisfiability proofs through examples: Implementation. `https://gitlab.inf.ethz.ch/OU-PMUELLER/student-projects/bsc-pstrebel`, 2021.

[11] Z3. `https://github.com/Z3Prover/z3`, (visited on 22.08.2021).

[12] Vampire. `https://github.com/vprover/vampire`, (visited on 22.08.2021).

[13] Alexandra Bugariu, Arshavir Ter-Gabrielyan, and Peter Müller. Identifying overly restrictive matching patterns in smt-based program verifiers. In *Formal Methods (FM)*, 2021.

[14] Peter Müller, Malte Schwerhoff, and Alexander Summers. Viper: A verification infrastructure for permission-based reasoning. pages 41–62, 01 2016.

[15] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 367–379, Cham, 2021. Springer International Publishing.

[16] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[17] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in f*. *SIGPLAN Not.*, 51(1):256–270, January 2016.

[18] Smt-comp: The 15th international satisfiability modulo theories competition, 2020. `https://smt-comp.github.io/2020/`, (visited on 22.08.2021).

[19] Rodrigo Otoni, Martin Blicha, Patrick Eugster, Antti E. J. Hyvärinen, and Natasha Sharygina. Theory-specific proof steps witnessing correctness of smt executions. In *DAC 2021 - 58th Design Automation Conference*, 2021.

[20] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. Smt proof checking using a logical framework. *Formal Methods in System Design*, 42, 02 2013.

[21] Michal Moskal. Rocket-fast proof checking for smt solvers. pages 486–500, 03 2008.

[22] Clark Barrett, Leonardo de Moura, and Pascal Fontaine. Proofs in satisfiability modulo theories. 07 2014.

[23] Frédéric Besson, Pascal Fontaine, and Laurent Théry. A flexible proof format for smt: a proposal. 08 2011.

[24] Sascha Böhme and Tjark Weber. Designing proof formats: A user's perspective — experience report —. 08 2011.

[25] Nico Darryl Hänggi. A better smt language: Design & tooling. Bachelor thesis, ETH Zürich, 2020.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Explaining Unsatisfiability Proofs through Examples

Authored by (in block letters):

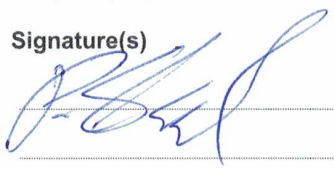
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Strebel	Pascal

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Freienwil, 01.09.2021	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.