

The present work was submitted to the LuFG Theory of Hybrid Systems

Master of Science Thesis

Comparison of Different Projections
in Quantifier Elimination for
Linear Real Arithmetic

Rebecca Ertl

Communicated by
Prof. Dr. Erika Ábrahám

Examiners:
Prof. Dr. Erika Ábrahám
Zerz Eva Barbara; Univ.-Prof. Dr. rer. nat. habil.

Additional Advisor: Valentin Promies

Aachen, 09.03.2025

Abstract

Linear real algebra (LRA) is used to approximate the behavior of complex systems. In various research fields, LRA formulas describe the system's behavior. For example, software verification and synthesis analyze the system behavior with the help of the LRA formulae. The description of relevant behavior for systems in LRA needs quantified variables.

LRA formulae are suitable if the system's behavior is linear. For analysis, the existence of behavior is often important, but not the exact one. Through quantified LRA the behavior is expressed at a suitable level of abstraction. If the formula is satisfiable, the approximated behavior can occur. However, the quantified formulae must be substituted by their quantifier-free equivalent for the analysis. Checking the satisfiability is too complex for quantified formulae otherwise.

In this thesis, the quantifier elimination algorithm, defined by David Monniaux in 2008, is presented. The option to update e.g., used projection makes the algorithm relevant. Furthermore, the algorithm's flexibility allows the use of modern methods. Since a novel projection method, FMplex, has been developed the algorithm's performance with this projection is an interesting topic. Moreover, to improve the behavior, variations to the algorithm are defined. Tests, comparing the variations, and two different projections offer a valid analysis of the results.

Contents

1	Introduction	7
2	Theoretical Foundations	10
2.1	Constraints in LRA	10
2.2	LRA Formulae with Quantification	12
2.3	Satisfiability of Formulae in LRA	17
2.4	Quantifier Elimination	18
3	The Quantifier Elimination Algorithm	22
3.1	General Description of the Algorithm	22
3.2	In-depth Description of Relevant Behavior	27
4	Variations of the Algorithm	33
4.1	Rotate and Repeat	33
4.2	Splitting up the Execution	38
4.3	Focus on the Active Constraints	41
5	Implementation in SMTRAT	47
5.1	Overview of the Variations	47
5.2	Statistics of the Execution	48
5.3	Validation of the Behavior	49
6	Benchmarks on the Implementation	52
6.1	Setting for the Benchmarks	52
6.2	Execution of the Benchmarks	53
7	Analysis of the Results	55
7.1	Summary of the Performance	55
7.2	Comparison between Projections FM and FMplex	58
7.3	Discussion of the Variants	60
7.4	Dynamic Activation of Variants	64
8	Conclusion	67
8.1	Summary	67
8.2	Discussion of Future Use	68
8.3	Future work	68
	Bibliography	71

Chapter 1

Introduction

In 2008 the author David Monniaux [1] defined a *quantifier elimination algorithm*. The algorithm calculates an *equivalent formula* F' without quantification to the input *linear real arithmetic* (LRA) formula $\exists.x(F)$. Additionally, the form will be in *conjunctive normal form* (CNF) or *disjunctive normal form* (DNF). The eliminated quantification and the output form enable fast satisfiability checking for F' .

Consequently, the algorithm is important to check the *satisfiability* of general LRA formulae with quantification. One core mechanism of the presented algorithm differentiates it from other approaches. The algorithm chooses a model with the help of a *satisfiability modulo theories* (SMT) solver. Then the model is *generalized* to an entire *conjunction*. This conjunction defines a larger area around the model. Then the elimination of the quantification is executed on this generalization.

The calculation of the generalization is summarized. First, all constraints of the LRA formula F will be collected. Important is, that the generalization is not allowed to include models from $\neg F$. Otherwise, all conjunctions built from the constraints are allowed as generalizations. The author, David Monniaux, uses a two-step approach to calculate the generalization. In the first step, *all constraints* are used to build the conjunction. However, this conjunction does include redundant constraints. To improve the algorithm's efficiency a second generalization step has been introduced. The second step will check for each constraint if it can be removed. After finding a minimal conjunction, *without redundancies*, the quantifiers are eliminated. In this regard, the generalization is projected. The method to build a *conjunction of (in)equalities* in LRA enables the presented algorithm [1] to be compatible with several *standard projections*.

The *projection* of the generalization is collected, to analyze the overlap of all projections with the input formula F . These projections guide the choice for the next model, since we need the projections to overlap the entire F . If the whole formula is overlapped, the criterion for termination is reached. Then the output, the disjunction of all projections, is equivalent to the original formula.

The original algorithm is defined for specific LRA formulae. This restriction has been relaxed since the projections implemented in the *Satisfiability-Modulo-Theories Real Algebra Toolbox* (SMT-RAT) [2] accept all kinds of LRA formulae as input. Moreover, the *eliminated quantification* must be *existential*. Since *universal quantifiers* can be transformed into existential ones, this does not lead to restrictions on the input. However, the implementation needs to handle the transformation of the input.

Calculating the *quantifier-free equivalent* to an input formula is important for checking the satisfiability. Because of the high complexity, the satisfiability of quantified formulae is reasoned by looking at the equivalent one. The algorithm allows a flexible choice regarding the SMT-solver and projection. This creates compatibility with new research and makes the algorithm relevant. Because the novel projection method *FMplex* [3] has been developed, testing the algorithm in combination with *FMplex* is interesting. Having such modern projection can improve performance and allow for new application areas. Two projections are compared to evaluate the projection's effect on the performance. For this the new method *FMplex* is compared to *Fourier-Motzkin* (FM).

Quantifier elimination is relevant because of the underlying mathematical basics of many systems. Using (in)equalities to *approximate complex behavior* is practical and often necessary. Without a clear mathematical description, the system's behavior can become unpredictable. The presented algorithm specializes in one mathematical theory, the LRA. These are *Boolean combinations of linear (in)equalities*, and they can be used to approximate the behavior of various processes. Therefore, these formulae are researched for automated reasoning in multiple scientific areas e.g. software verification. Moreover, the description of systems in LRA is practical and fast. Consequently, various applications using an SMT-solver in combination with LRA have been developed. In research fields, which span from the area of software verification [4, 5] to chemistry [6, 5] or biology [7, 5].

The existence of a solution for the formulae is proven, to show their satisfiability. Deciding for the complexity of the satisfiability check, are the formula's structure and theoretical background. The quantification of a formula's variables does lead to a heightened complexity. However, the quantification of a variable is needed to express the existence of a behavior. For example, existential quantification defines that at least one variable value, satisfying the formula, must exist. Negating existential quantification excludes behavior for the quantified variable. Having quantification allows you to express behavior elegantly and understandably. However, the quantification does complicate the check for the satisfiability of the formulae. The strategy is to eliminate all quantifiers before any satisfiability checking is executed.

This problem was identified quite early, with Joseph Fourier defining a method for variable elimination in 1826. So, research into quantifier elimination algorithms has been going on for a long time. The *Tarski-Seidenberg Theorem* [8, 9], has been crucial for advancing satisfiability solving. For every quantified formula, a quantifier-free equivalent exists. Based on this research multiple algorithms to eliminate quantifiers have been formulated. These describe how to find a quantifier-free equivalent F' to formula $\exists.x(F)$. Different algorithms have been defined for the task of eliminating quantifiers. In this thesis, the algorithm by Monniaux [1] is implemented and the potential future use is discussed. Many algorithms for quantifier elimination have niches for their best performance. Thus, multiple options are offered in the implementation. Through this choice, an adjustment into a suitable niche is possible.

Regarding this, we present two projections, the *Fourier-Motzkin (FM)* and the advanced *FMplex* projection. [3] Testing the algorithm with two projections, allows to evaluate its future uses thoroughly. While FM is established and can be optimized to perform fast for simple cases, *FMplex* has shown competitive behavior for eliminating many variables at once.

Additionally, multiple *variants* to the standard description of the algorithm are offered. These have been chosen based on their potential to improve the algorithm's

performance. The decision regarding the adjustments is based on the improvements for specific behavior. Tests have been carried out to avoid changes that decrease the overall performance. These tests have also been used to identify the input structure needed to profit from the variants. The final version is based on multiple cycles of identifying the causes for a long runtime and adjusting the behavior. Then the adjustment is trialed in tests to check if the behavior improved. Through these cycles, the three presented changes have been identified as relevant. The changes avoid the worst-case behavior and focus on the generalization and the input.

Furthermore, the restrictions on the formulae that the algorithm will be applied to have been relaxed. This is possible because of the SMT-solver and the projections. The algorithm's implementation for *SMT-RAT* will be tested with standard LRA [10] benchmarks. The *Cylindrical Algebraic Coverings* (CalC) algorithm [11] is chosen as a comparison for the performance evaluation. Multiple statistics about the variants' behavior are collected. The comparison and the statistics are necessary for reasoning about the algorithm's performance. Based on the results, an in-depth analysis of the performance, with a focus on the input formula is given.

The underlying theory is detailed, to understand the mechanisms behind the algorithm's behavior. Consequently, the algorithm by Monniaux [1] is presented by the following structure. First, the theory 2 will be explained and afterward, the algorithm 3 will follow. The part on the algorithm describes the standard behavior 3.1, and details the core mechanism 3.2. Followed by an explanation of the variants 4 of the behavior. Then, the implementation 5 discusses the code structure and decisions relevant to the benchmark execution 6. Afterward, we analyze the benchmarks 7 and the collected statistics. Lastly, we conclude 8 with a summary, and ideas for the algorithm's future.

Chapter 2

Theoretical Foundations

2.1 Constraints in LRA

The *linear real arithmetic (LRA)* is an arithmetic that has *variables* $\mathbf{x}$ over $\mathbb{R}$. If the formula has multiple variables $\{x_1, x_2, \dots, x_n\}$, we write $\mathbf{x}$. The variables, possibly multiplied by a *rational constant* are added to construct a linear polynomial P . Constants $c \in \mathbb{Q}$ are also allowed in the sum. Terms are also the sum of multiple linear polynomials $P(\mathbf{x})$. The *constraint in LRA* is a comparison $P_1 \sim P_2$ of two polynomials P_1 and P_2 . But this form for constraints is not practical for calculation, since both sides are polynomial. Therefore, for the definition in this thesis, the chosen form is stricter.

Definition 2.1.1 (Constraints in LRA). *Let P be a polynomial, with variables $\{x_1, x_2, \dots, x_n\} \in \mathbb{R}^n$, $x_0 = 1$, and constant factors $\{a_0, a_1, a_2, \dots, a_n\} \in \mathbb{Q}$. The constraint c , using one comparison operator $\sim \in \{=, \neq, <, >, \leq, \geq\}$ has the following structure:*

$$c := P(\mathbf{x}) \sim 0 \equiv \sum_{i=0}^n a_i x_i \sim 0$$

While the above definition is used, it is also possible to have a definition that restrict the comparison operators to $\{<, \leq\}$. However, the definition allows for all comparison operators in the constraints. Through *logic operators* the *comparison operators* can be transformed into each other. Moreover, combining LRA formulae with *Boolean logic* results in new LRA formulae. When the *negation* ($\neg$) is applied to a constraint $F := P \leq 0$, the result $\neg F = -P < 0$ has a *strict less* ($<$) operator. All further comparison operators can be derived from *less than or equals* ($\leq$) and *less* ($<$) through the choice of a suitable Boolean structure.

The following substitutions are used for finding an equivalent formula.

Example 2.1.1 (Substitution of Comparison Operators). *Necessary to define constraints $c := P(\mathbf{x}) \sim 0$ with $\sim \in \{=, \neq, <, >, \leq, \geq\}$ are strict and non-strict inequalities of one kind. To have a constraint in LRA, the comparison operators are either less ($<$) and less than or equals ($\leq$).*

Inequalities of the opposite kind, ($>$) or ($\geq$) are substituted by multiplying P with -1 :

$$F := P(\mathbf{x}) > 0 \equiv -P(\mathbf{x}) < 0.$$

To define the equality ($=$) or not-equals ($\neq$) two LRA constraints using one of the comparison operators $\{<, \leq\}$ are combined with the Boolean operator $\wedge$ (AND).

Equalities are defined by:

$$P(\mathbf{x}) \leq 0 \wedge P(\mathbf{x}) \geq 0 \equiv P(\mathbf{x}) = 0$$

Not equals is derived by replacing the inequalities with strict ones:

$$P(\mathbf{x}) < 0 \wedge P(\mathbf{x}) > 0 \equiv P(\mathbf{x}) \neq 0$$

For calculations with the constraints their *satisfiability* is most interesting. To check if a constraint is satisfiable, an *assignment* for the polynomial variables consistent with the comparison operator must be found. The consistency is checked by substituting the variable in the constraint with the assigned value. If the comparison operator is not violated, the assignment is consistent and called a solution for the constraint.

The dimension of the solution can vary for (in)equalities. We explain this by giving an example.

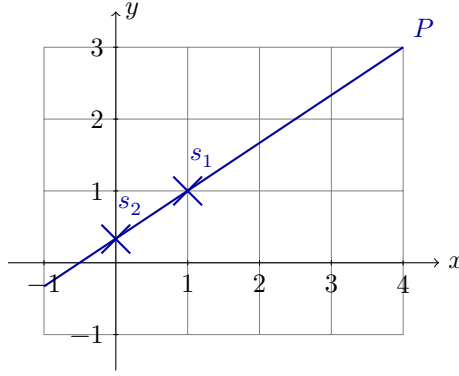


Figure 2.1: The equality $c_1 := P = 0$ is shown with two solutions marked.

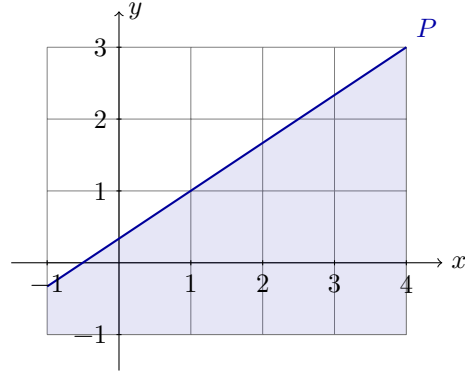


Figure 2.2: Variable values consistent with $c_2 := P \leq 0$ are in the blue area.

Example 2.1.2 (Satisfiability of Constraints in LRA). Consider the 2-dimensional polynomial $P(x, y) = (-2)x + 3y + (-1)$ with variables x and y . The polynomial's constant factors are $a_x = -2$ for x , $a_y = 3$ for y with the added constant $a_0 = -1$.

Using $P(x, y)$ as the left side of a constraint $c := P(x, y) \sim 0$, builds different LRA constraints. Depending on the comparison operator $\sim$, the solutions for c vary.

Defining the equality $c_1 := P(x, y) = 0$ will result in a constraint with the solutions on the line. One possible combination of values s_1 as solution is $x = 1$ and $y = 1$, because $P(-1, 1) = (-2) + 3 + (-1) = 0$. Another allowed variable assignment s_2 would be the combination $x = 0$ and $y = \frac{1}{3}$. The constraints and the solution are shown in Figure 2.1.

In the case of an inequality $c_2 := P(x, y) \leq 0$, the constraint is satisfiable for the half-space to the bottom. The solutions given in Figure 2.2 are either directly on the line $P(x, y)$ or in the space below the line.

Using strict inequalities less ($<$) or greater ($>$), will exclude the line from the solutions. Constraints with solutions in the upper half-space use one of the comparison operators $\{>, \geq\}$ to express greater.

The constraints in LRA are (in)equalities but multiple algorithms are defined for either equalities or inequalities. For example, the Fourier-Motzkin (FM) algorithm is defined specifically for conjunctions of inequalities with quantification. To meet this requirement, the LRA formulae can be substituted by equal ones in the correct form.

Moreover, the exact form of the constraints is relevant to allow fast comparisons between two constraints. Relevant is one typical representation of LRA constraints, the *normal form*. The *normal form* restricts the inequalities to one kind (less), and applies strict rules on the right and left sides of the formula.

Definition 2.1.2 (Normal Form for LRA Constraints). *The constraint in normal form is defined by: $c := P(x) \sim 0$, with $\sim \in \{=, <, \leq\}$*

Calculations must consider fewer polynomials and comparison operators, through the normal form. Moreover, all LRA constraints with the same solutions can be transformed into an equivalent form.

The example below explains the transformation of one constraint c_2 into the constraint c_1 in normal form.

Example 2.1.3 (Transforming LRA Constraints to Normal Form). *Constraint c_1 is from the first example 2.1.2: $c_1 := (-2)x + 3y - 1 \leq 0$*

The second constraint c_2 is the following: $c_2 := 2x \geq 3y - 1$

To prove the equivalence of both constraints, the constraint c_2 is normalized.

1) Substitution of subtraction by addition:

$c_2 := 2x \geq 3y - 1 \equiv 2x \geq 3y + (-1)$ (Change of the constants sign)

2) Compare against 0:

$2x \geq 3y + (-1) \equiv 2x + (-3)y + 1 \geq 0$ (Use addition/subtraction)

3) Change of the comparison operator:

$2x + (-3)y + 1 \geq 0 \equiv (-2)x + 3y - 1 \leq 0 \equiv c_1$ (Multiplication with -1)

Consequently, both formulae are equivalent.

After the transformation to normal form, all constraints have the same structure. By exploiting this knowledge, efficient computations can be executed.

Constraints express simple linear relations between the variables, as seen in the first example 2.1.2. Boolean combinations of multiple constraints, the LRA formula describes complex linear behaviors.

2.2 LRA Formulae with Quantification

The background theory for LRA formulae is *linear real arithmetic* (LRA). Multiple constraints are connected to one formula F by *Boolean logic operators*. As logic operators, $\wedge$ (and) or $\vee$ (or) are allowed, together with the negation $\neg$ (not) to describe the constraint's interactions. By building combinations of the basic Boolean operators, logic formulae with more complex logic operators $\Rightarrow$ (implies), $\oplus$ (xor) and $\Leftrightarrow$ (equivalent) can be defined. Moreover, both types of quantifiers, existential ($\exists$) and universal ($\forall$) can be used to quantify the formula's variables.

Through quantification, the reasoning about satisfiability is more complex. However, multiple LRA formulae with and without quantification have the same solutions. To execute the satisfiability check, an equivalent formula to the one with quantification needs to be found. The presented algorithm defines a method to calculate a *quantifier-free* formula F' equivalent to $\exists.\mathbf{x}(F)$ with quantified variables $\mathbf{x}$.

Exactly as for the constraints, the existence of a solution for the formula is interesting. The *satisfiability check* considers both the *logic* and *theory* of the formula. If a formula is satisfiable, at least one assignment α for the variables $\mathbf{x}$ must be consistent with the formula. The assignment is consistent with formula F , if the formula F is *true* after all variables are substituted by their assignment. For the formulae in LRA two types of satisfiability checking are relevant and defined as follows.

Definition 2.2.1 (Satisfiability Checking for LRA Formulae). *For the LRA formula F , two types of satisfiability checks are:*

- Check if a solution for F exists: $\text{Sat}(F)$
- The variable assignment α satisfies a formula F : $\alpha(\mathbf{x}) \models F$

The first type of SAT-check is *satisfiable (SAT)* if a solution for F exists. When no solution exists the result is *unsatisfiable (UNSAT)*.

For the second type, the variable assignment α is substituted into F . The result is *SAT* if the assignment is consistent with F . When the assignment violates F , the result is *UNSAT*.

We give an intuition to the satisfiability checking for LRA formulae by an example. Since the example 2.2.1 is simple, the solution can be found graphically. In the example, the satisfiability of two formulae, with and without quantification is compared.

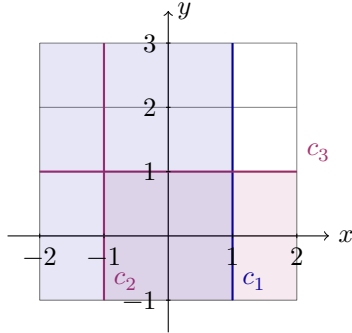


Figure 2.3: The satisfiable areas for relevant subformulae of F_1 are depicted. The blue area is satisfiable for c_1 and magenta for $(c_2 \wedge c_3)$.

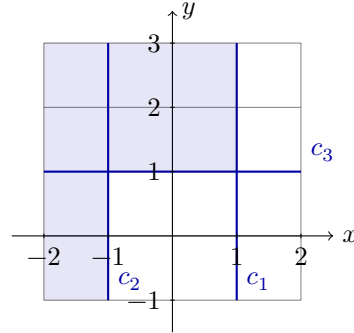


Figure 2.4: The area marked in blue satisfies $F_1 := c_1 \wedge \neg(c_2 \wedge c_3)$.

Example 2.2.1 (Satisfiability of LRA Formulae). *Two formulae, with and without quantification, will be defined with the following inequalities:*

$$c_1 := x \leq 1, c_2 := x \geq -1 \text{ and } c_3 := y \leq 1$$

$$\text{Formula } F_1 := c_1 \wedge \neg(c_2 \wedge c_3) = x \leq 1 \wedge \neg(x \geq -1 \wedge y \leq 1).$$

Figure 2.3 shows the satisfiable areas for relevant subformulae of F_1 . To find the solutions for F_1 first the entire area satisfying c_1 is marked. Then the negated conjunction $\neg(c_2 \wedge c_3)$ is considered. By removing the area where both subformulae c_1 and $(c_2 \wedge c_3)$ overlap from the solutions. The entire area with solutions for formula F_1 is marked in Figure 2.4.

By introducing the constraint $c_4 = x > 1 = \neg c_1$ for F_2 , the abstraction will include a negation. To build the abstraction, both c_1 and c_4 will have a as an assigned Boolean variable. The constraint c_2 is represented by b and c_3 by c .

Substituting the constraints of F_2 by their assigned Boolean variables constructs the abstraction $\neg a \vee (a \wedge \neg(b \wedge c))$.

Additionally, the potential quantification of the formulae needs to be explained. Since the presented algorithm processes quantified formulae, an in-depth look is taken at quantified formulae in LRA.

2.2.1 Quantified Formulae in LRA

Formulae in LRA with *quantification* need at least one variable x bound by $Q \in \{\forall, \exists\}$. To this regard, quantifier-variable pair $Q.x$, with $Q \in \{\exists, \forall\}$ will be added from the left to a formula. Binding a variable x by an existential $\exists$ quantifier states that the formula holds for at least one assignment $\alpha(x)$. If a universal quantification is used, all assignments of the bound variable must be part of a solution for the formula.

Definition 2.2.3 (Quantified Formulae in LRA). *The addition of the quantifier-variable pair $Q.x_i$ form the left quantifies the variable x_i of formula F : $Q.x_i(F)$ and $Q \in \{\forall, \exists\}$*

To quantify n variables x_i , with $i \in \{1, \dots, n\}$, for each variable one pair $Q_i.x_i$ is added to the formula F : $Q_1.x_1 \dots Q_n.x_n(F)$ with $Q_i \in \{\forall, \exists\}$

If the quantifiers are of one kind for all n variables this can be written as $Q.x(F)$ with x representing the variables and $Q \in \{\forall, \exists\}$.

The quantification is possible for the *entire formula* or also for *nested subformulae*. Variables without a quantification are called *free*, the other ones are *quantified*. If the variable x is quantified for a nested subformula S but not the entire formula F , all variable occurrences outside of the subformula S are free. The variable x is quantified only for the nested subformula S . Through that a hierarchy of the quantification is built, from the innermost quantified subformula to the outer one.

Eliminating quantifiers needs to respect the quantifier hierarchy, starting at the innermost nesting. The quantifier elimination algorithm, also called a projection, must be executed.

Multiple rules are in place in case the structure of the quantified formula needs to be changed. The transformation of quantifiers, the rules for their positions, and the distribution of quantifiers over conjunctions and disjunctions will be explained in detail.

Quantifier Negation

The first rule on *quantifier negation* allows to *transform quantifiers* of one kind into the other one.

Lemma 2.2.1 (Quantifier Negation). *The definition is for the quantifier negation since the transformation of quantifiers is a special case of the negation.*

The following equivalences are defined for quantified formula F :

$$\neg(\exists.x(F)) \equiv \forall.x(\neg F)$$

$$\neg(\forall.x(F)) \equiv \exists.x(\neg F)$$

$$\exists.x(F) \equiv \neg(\forall.x(\neg F))$$

$$\forall.x(F) \equiv \neg(\exists.x(\neg F))$$

This rule is especially important since many projections e.g., Fourier-Motzkin or FMplex only regard existentially quantified formulae. Consequently, transforming universal quantifiers into existential ones is mandatory to enable the execution of multiple relevant algorithms.

Quantifier Equivalences

The second rule based on *quantifier equivalences* defines the equivalent formulae with a change of to the quantifiers' *ordering*.

Lemma 2.2.2 (Change of Quantifier Ordering). *Any two quantifier-variable pairs directly after each other can swap their orderings, if the quantifiers are of the same kind:*

$$Q.x_1Q.x_2(F(x_1, x_2)) \equiv Q.x_2Q.x_1(F(x_1, x_2)), \text{ with } Q \in \{\exists, \forall\}$$

Repeatedly applied swaps result in an interchangeable ordering of quantifiers of the same kind. This allows the collection of variables $\mathbf{x}$ into one quantified block $Q.\mathbf{x}$ at the same nesting:

$$Q.x_1 \dots Q.x_n(F(\mathbf{x})) \equiv Q.\mathbf{x}(F(\mathbf{x})), \text{ with } Q \in \{\exists, \forall\}$$

We give an example to display how to collect multiple variables into blocks.

Example 2.2.3 (Change of Quantifier Positions). *To the formula $\exists.x_1\exists.x_2\forall.x_3\exists.x_4(F(\mathbf{x}))$, with $\mathbf{x} := \{x_1, x_2, x_3, x_4\}$, the rule to change the quantifiers ordering is applied.*

The quantifier-variables pairs $\forall.x_3$ and $\exists.x_4$ have fixed positions, since their neighbors quantifiers have the opposite kind.

For swapping the positions of the two outer quantifiers, we introduce $\mathbf{y} = \{x_1, x_2\}$. The changed positions of the quantification and the block building are shown:

$$\begin{aligned} & \exists.x_1\exists.x_2\forall.x_3\exists.x_4(F(\mathbf{x})) \\ & \equiv \exists.x_2\exists.x_1\forall.x_3\exists.x_4(F(\mathbf{x})) \\ & \equiv \exists.\mathbf{y}\forall.x_3\exists.x_4(F(\mathbf{y}, x_3, x_4)) \end{aligned}$$

Collecting quantifier-variable pairs of *one kind* as *blocks* is practical for applying projections. Eliminating an entire block increases the projections' flexibility. The chosen algorithm can reorder the quantified variable as needed. The blocks allow a high performance for specialized algorithms eliminating multiple variables, such as the FMplex.

The third rule is on *distributing the quantification* over the entire the formula.

Lemma 2.2.3 (Distribution of the Quantifiers). *The formulae regarded are either conjunctions $\bigwedge_{s_j \in S} s_j$ or disjunctions $\bigvee_{s_j \in S} s_j$, with S the set of conjuncts or disjuncts.*

Distributing the quantifiers needs to respect the following equivalences:

$$\begin{aligned} & \exists.x(s_1 \vee s_2 \vee \dots \vee s_n) \equiv \exists.x(s_1) \vee \exists.x(s_2) \vee \dots \vee \exists.x(s_n) \\ & \exists.x(s_1 \wedge s_2 \wedge \dots \wedge s_n) \not\equiv \exists.x(s_1) \wedge \exists.x(s_2) \wedge \dots \wedge \exists.x(s_n) \\ & \forall.x(s_1 \wedge s_2 \wedge \dots \wedge s_n) \equiv \forall.x(s_1) \wedge \forall.x(s_2) \wedge \dots \wedge \forall.x(s_n) \\ & \forall.x(s_1 \vee s_2 \vee \dots \vee s_n) \not\equiv \forall.x(s_1) \vee \forall.x(s_2) \vee \dots \vee \forall.x(s_n) \end{aligned}$$

Effectively this results in a move of the quantification past the logic operator into the formula. Through the change, the quantifier elimination can be applied directly to the subformulae.

2.3 Satisfiability of Formulae in LRA

2.3.1 SMT Solving for LRA Formulae

The area of *satisfiability modulo theories (SMT)* determines if a solution for LRA formulae exists. Moreover, if a solution is found, the solver returns *satisfiable (SAT)* and gives back a model satisfying the formula as proof. The model is an assignment for the variables that satisfy the input formula. If no solution is found, the solver will return *unsatisfiable (UNSAT)* and a set of conflicting constraints as reasoning.

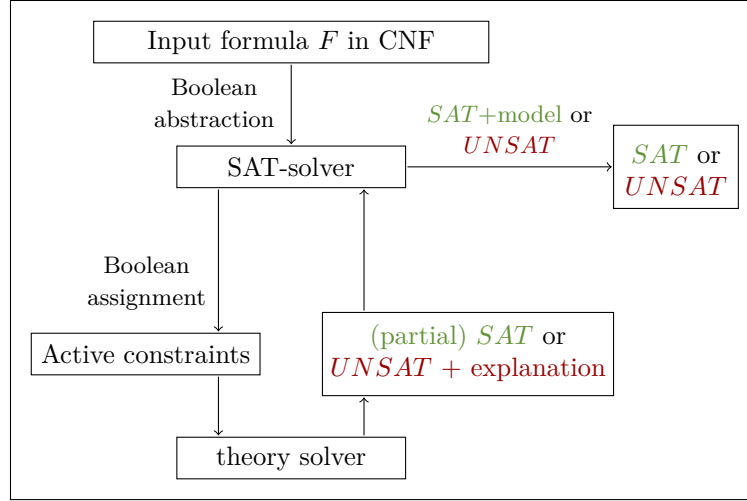


Figure 2.6: The model depicts the behavior of a (less) lazy SMT-solver.

To check the satisfiability of LRA formulae, the solver considers the formula's *logic* and *theory* separately. First, logic solving is executed for the input by an SAT-solver. The SAT-solver targets finding a satisfiable *Boolean assignment* for the Boolean skeleton. In this regard, a specialized algorithm with intelligent decision procedures is employed.

As a second step, the consistency of the assignment with the underlying theory is evaluated. Accordingly, a call to the theory solver, transferring the assignment is executed. If the theory is consistent with the assignment, the result is *SAT*, and the proof is a model for the formula. If the theory solver detects a conflict, the result is *UNSAT*. Then the reasoning is given back to the SAT-solver to apply conflict resolution. The SAT-solver will then suggest a new assignment based on its updated Boolean skeleton.

The described alternation between theory and logic solving is called *lazy*, because the decision procedure for the Boolean component is followed by a consistency check of the theory. Refining this *lazy strategy* led to further approaches like the *less lazy* one, implemented in SMT-RAT [12].

The behavior of *(less) lazy* SMT-solvers is explained next. Figure 2.6 depicts an SMT-solver with a lazy strategy, to deepen the understanding. The need to alternate between evaluating the logic and theory results in the cyclic behavior visible in Figure 2.6. While the lazy approach will start the theory solving on a full assignment for the Boolean skeleton, in the less lazy approach the consistency check is already called

for partial Boolean assignments [13]. Through the earlier call of the theory solver, conflicts are resolved faster.

The constraints are categorized into *active* and *passive* depending on the constraints needed to calculate the assignment. Every constraint, represented by a Boolean variable in the assignment is called *active* and needs to be considered for the satisfiability check. All other constraints are regarded as *passive* and are not involved in evaluating if the formula is satisfiable. [14]

Definition 2.3.1 (Active and Passive Constraints). *Let α be a (partial) Boolean assignment of the Boolean skeleton. The assigned Boolean variables build the set B .*

The constraint c represented by a variable $b \in B$ is called active. Otherwise, the constraint c is called passive.

With an example below, we explain how the satisfiability checking is done for formulae in LRA. The example uses an approach with a partial assignment.

Example 2.3.1 (Satisfiability of a Formula with LRA as Theory). *The satisfiability of formula $F_1 := c_1 \wedge \neg(c_2 \wedge c_3)$ is determined, with a graphical solution in Figure 2.4.*

Following three constraints are included in F_1 : $c_1 := x \leq 1$, $c_2 := x \geq -1$ and $c_3 := y \leq 1$

0. Transfer F_1 to the solver

1. Logic solving (take the abstraction from example 3.2.2)

Construction of the Boolean abstraction: $B_{F_1} := a \wedge \neg(b \wedge c)$

Transform to CNF: $B_{F_1} := a \wedge (\neg b \vee \neg c)$

Partial assignment α : $\alpha(a) = 1$, $\alpha(b) = 0$, $\alpha(c)$ is not relevant

2. Check the consistency of the assignment α with the theory

Active constraints are c_1 and c_2 because a and b are assigned. So c_1 and $\neg c_2$ are regarded for the theory check. The constraint c_2 is negated because the assignment $\alpha(b)$ is 0.

3. Theory solving (use the figure 2.4 to find the solution)

Solution consistent with c_1 and $\neg c_2$: $\alpha(x) := -2$, y flexible e.g. $\alpha(y) := 0$

4. Give back SAT and as proof the model $m := (-2, 0)$.

2.4 Quantifier Elimination

2.4.1 Background to Quantifier Elimination

Tarski described the first *quantifier elimination algorithm* (QE algorithm) for real arithmetic in 1951 [8]. Later the *Tarski-Seidenberg theorem* formulated, the *projection* onto a lower dimension for polynomials with rational coefficients. [9, 8] Crucial for the performance of QE algorithms is identifying regions for projection. These *regions* need to be connected and invariant regarding their satisfiability. Then one model for each region is chosen for the satisfiability checking. We *generalize* the solution for the model to the entire region. The quantifiers are eliminated by projecting from these regions, not the formula. The presented algorithm also has this strategy to generalize the knowledge about one model to a larger area.

To achieve a more efficient strategy, the author Monniaux [1] employs an SMT-solver to guide the decision for suitable regions. For an input formula $Q.xF(x)$, first, an SMT-solver is called to decide on the satisfiability of F . If the result is *SAT*, the

model given as proof will be extended to a *conjunction satisfying the formula*, which is then projected.

The decision, to project based on the model's *generalization to a conjunction* distinguishes the algorithm [1] from others. We consider two different projections for LRA formulae in this thesis. For both projections, Fourier-Motzkin (FM) and FMplex a general description is given, without details.

2.4.2 Projections for Variable Elimination

The projections differ in how many quantified variables they will regard for one elimination step. Eliminating one variable at a time, the Fourier-Motzkin (FM) projection is quite simple. In contrast multiple existentially quantified variables are considered by the FMplex [15] projection at once.

First the simpler method Fourier-Motzkin (FM) elimination will be described and afterward an in-depth look is taken at the advanced FMplex projection.

2.4.3 Fourier-Motzkin Projection

Joseph Fourier introduced the Fourier-Motzkin (FM) elimination algorithm in 1827. Later, in 1936, Motzkin rediscovered the procedure. The algorithm considers conjunctions of linear non-strict inequalities with existentially quantified variables. The *upper* and *lower bounds* of variable x need to be identified, to eliminate x .

Definition 2.4.1 (Upper and Lower Bounds). *Let $\bigwedge_{c_j \in C} c_j$ be a conjunction of constraints C with the n variables $x_i \in V$. The constraints have the form $c_j := \sum_{i=0}^n a_i x_i \leq 0$, with $x_0 := 1$. We identify if a constraint is the upper or lower bound of a variable x_i by looking at the constant factor a_i .*

$$a_i = \begin{cases} < 0 & (\text{Lower Bound}) \\ = 0 & (\text{No Bound}) \\ > 0 & (\text{Upper Bound}) \end{cases}$$

To emphasize that a constraint c_l is a lower bound for variable x , often $c_l := l \leq x$ is written. For the upper bound the constraints are written in the form $c_u := x \leq u$. Building combinations of one variable's upper and a lower bound results in a constraint c with the variable eliminated.

Definition 2.4.2 (Elimination of a Variable). *Let $x \leq u$ be an upper bound and $l \leq x$ a lower bound for x . To eliminate x the bounds are combined to a new constraint $c_e := l \leq u$.*

The Fourier-Motzkin Theorem defined the central elimination of the variable from the entire conjunction.

Theorem 2.4.1 (Fourier-Motzkin Theorem). *To eliminate a variable x_i from a conjunction $F = \bigwedge_{c_j \in C} c_j$ the upper and lower bounds for the variable are identified. New inequalities $c_e \in C_{elim}$ with x_i eliminated are built from the bounds.*

The projection P , written as $\text{proj}(F)$, is built from the constraints in C_{elim} and the set of inequalities $C_{p'}$ without x_i .

$$P := \text{proj}(F) \equiv \left(\bigwedge_{c_e \in C_{elim}} c_e \right) \wedge \left(\bigwedge_{c_{p'} \in C_{p'}} c_{p'} \right)$$

The strategy is improved by removing redundant bounds from the P .

An example applies the FM to eliminate the variable y from a formula. Moreover, Figures 2.7 and 2.8 support the explanation graphically.

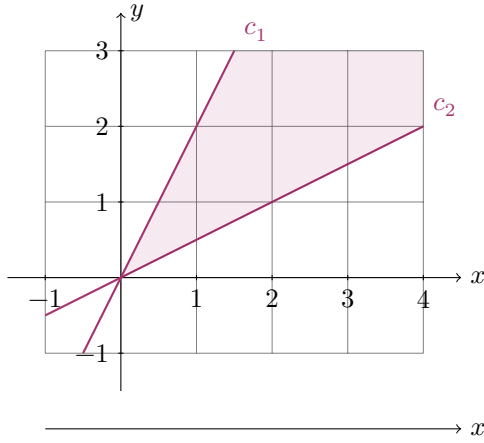


Figure 2.7: Function F is shown, with the cone shaded magenta.

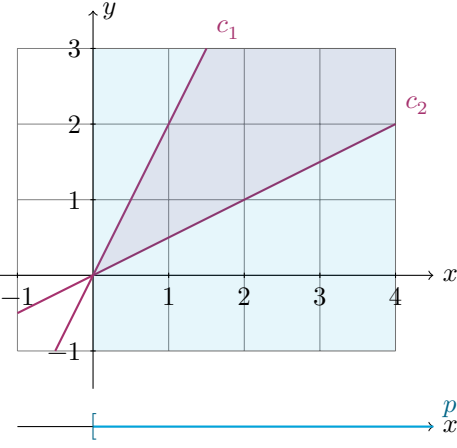


Figure 2.8: The projection p of the cone $c_1 \wedge c_2$ is displayed in blue.

Example 2.4.1 (Quantifier Elimination with Fourier-Motzkin). *The formula F , visualized in Figure 2.7 is a cone defined by the conjunction $(c_1 \wedge c_2)$.*

The constraints are: $c_1 := -2x + y \leq 0$, $c_2 := x - 2y \leq 0$

The following steps are necessary to eliminate variable y from $\exists.y(F)$:

1) Identification of lower/upper bounds

Lower bound: $x - 2y \leq 0$, because factor $-2 < 0$

Upper bound: $-2x + y \leq 0$, because $1 > 0$

2) Combine lower and upper bounds to eliminate y

Lower bound: $x - 2y \leq 0 \equiv \frac{1}{2}x \leq y$

Upper bound: $-2x + y \leq 0 \equiv y \leq 2x$

Combine upper and lower bounds to eliminate y .

Projection: $\frac{1}{2}x \leq 2x \equiv 0 \leq \frac{3}{2}x \equiv 0 \leq x$

For the graphical solution, the projection is marked by the positive half-space including 0 for x .

The figure 2.8 shows the result, $p := -x \leq 0$ additionally with only the x -dimension.

Multiple variables are eliminated through one application of FM for each variable. Adding all potential combinations of upper and lower bounds to the projections leads to an exploding number of constraints in the worst case.

2.4.4 FMplex projection

However, not all upper and lower bounds are necessary for calculating the projection. Consequently, the FMplex will use a *divide-and-conquer* approach to optimize the performance of FM. To benefit from the advantages of splitting up the search for a projection, the FMplex will ignore all redundant bounds. The resulting FMplex [15] projection can eliminate *multiple* quantifiers at once. More importantly it lowers the complexity from doubly to singly exponential. The method is implemented into the SMT-solver SMT-RAT [2] and intelligently prunes the search tree for optimization.

The FMplex uses a *divide-and-conquer strategy* for calculating the projection. [3] Through the splitting all upper/lower bounds that are not the lowest/largest can be disregarded. The first definition of FMplex is most suitable for reasoning about the satisfiability of formulae. The method was extended to be also usable for quantifier elimination and thus, can be used as fast variable elimination algorithm.

For the adaption of FMplex to a quantifier elimination algorithm, the *divide-and-conquer* approach [3] was chosen. The *divide step* produces a disjunction of projections for the sub-problems. Crucial to enable quantifier elimination is the introduction of a *conquer step*, combining all results efficiently into a conjunction. The result is a procedure, that can eliminate equalities and strict inequalities. [3]

Chapter 3

The Quantifier Elimination Algorithm

The algorithm calculates $F' \equiv \exists.x(F)$, the *equivalent quantifier free* formula F' to $\exists.x(F)$ in LRA. To derive the equivalent quantifier free formula a typical mechanism for QE-strategies is employed. Not the entire formula will be substituted at once, the algorithm will partition the formula into one or many smaller parts which are satisfiable. How the formula's sub-parts are identified is characteristic of the algorithm, which *generalizes* a satisfiable model for the formula to an area around it.

This *generalization* is executed by building a *conjunction* of the formula's constraints in a way that selects an area around the model. Afterward, the generalization will be *projected*, and this projection is an equivalent quantifier-free representation for the entire area it overlaps of the formula. Logically repeating these steps, choosing a model, generalizing the model and projecting will result in an equivalent representation F' of the entire $\exists.x(F)$.

The algorithm's performance depends on these three steps. The strategic decision for a model leading to a large generalization and efficient quantifier elimination is necessary for robust performance.

Before explaining the 3 characteristic steps in-depth, the main functionality and structure of the algorithm are outlined.

3.1 General Description of the Algorithm

First, the algorithm potentially will *pre-process* the input, to have the specified form for the quantified formula in LRA. Accepted by the algorithm [1] are inputs of the form $\exists.x(F)$, which have existentially quantified variables, $\exists.x$, in front of one unquantified formula F in LRA.

Executing the algorithm on formulae with a different form is possible, but the implementation needs to handle necessary *transformations*. Also, repeated executions of the algorithm, mandatory for eliminating *quantification with alternation*, need to be considered by the implementation. These details will be regarded during the detailed description of the implementation.

For a suitable LRA formula with quantification $\exists.x_1\exists.x_2...\exists.x_n(F)$, the algorithm starts by separating the unquantified formula F from the quantifier-variable pairs.

After extracting F , the algorithm starts the loop calculating F' , the quantifier-free formula equivalent to the input. This calculation is an iterative procedure, with every loop step i building on the results of the prior loop step $i - 1$. The algorithm 1 executing the quantifier elimination, EXISTELIM is given as pseudocode.

Algorithm 1 The algorithm EXISTELIM for the elimination of existential quantifiers. [1].

```

EXISTELIM (  $\exists \mathbf{x} F$  )
1   $H \leftarrow F$ 
2   $O \leftarrow false$ 
3  while  $Sat(H)$  is satisfiable (call SMT-solver)
4  do  $m \leftarrow \text{model of } H$                                 //  $m \models H$ 
5       $M_1 \leftarrow \text{GENERALIZE1}(F, m)$                     //  $M_1 \implies F$ 
6       $M_2 \leftarrow \text{GENERALIZE2}(\neg F, M_1)$               //  $(M_2 \wedge \neg F) \equiv false$ 
7       $p \leftarrow \text{PROJECT}(M_2, \mathbf{x})$                       //  $p \equiv \exists \mathbf{x} M_2$ 
8       $O \leftarrow O \vee p$ 
9       $H \leftarrow H \wedge \neg p$ 
10 return  $O$                                                 //  $O \equiv \exists \mathbf{x}(F)$ 

```

Inside this loop the following 3 relevant steps are executed for every iteration i : choosing the *model* m for formula H , *generalizing* m to a *conjunction* M , and projecting the generalization $p_i = \text{proj}(M)$. Before the next iteration $i + 1$, the algorithm excludes the projection p_i from the decision for future models. To push for the continued progress of the iterations. By removing the projection from being considered for model picking, the decisions for areas to project are strategic and intelligent.

After n iterations the final equivalent formula F' is a *disjunction* of all calculated projections p_i , with $i \in \{1, 2, \dots, n\}$ and as such will include the exact same models as $\exists \mathbf{x}(F)$. To collect the calculated projections and to track the procedure's progress, the algorithm prepares two formulae H and O .

Central for the algorithm's progress is that the loop is executed, as long as *at least one model* for H not included in any projection is found. Consequently, the loop checks the condition $Sat(H) = UNSAT$ and continues to execute until formula H is unsatisfiable. Ensuring the correct behavior is done by initializing formula H to F and removing every projection p_i from formula H . By updating H in each iteration to the *conjunction* ($H := H \wedge \neg p_i$), of H with the negated projection $\neg p_i$. As such the formula H tracks any sub-parts of the formula F that need to be considered for the model picking.

Through this mechanism, the decision for a model and its generalization is guided effectively. When the result is calculated, the formula H is *unsatisfiable* since all models of F are represented by at least one projection p_i and thus removed from H .

The second formula O is initialized to *false* since all projections are collected as the *disjunction* in formula O . When the algorithm is finished, the quantifier-free representation $F' \equiv O$, equivalent to the input formula $\exists \mathbf{x}(F)$, is returned.

Next, an overview of the functions of the loop is given, based on the algorithm EXISTELIM 1. The three functions GENERALIZE1, GENERALIZE2 and the *projection* are the focus. Since the loop execution depends on the satisfiability of formula H , one satisfiability check is the executed to evaluate the loop guard.

In the first execution, formula H is the unquantified formula F of the input. When check has the result SAT , the SMT-solver used for the check will also give back the proof, a *model* m of the formula H . Then a conjunction, which must include m , is built by calling two functions, GENERALIZE1 and GENERALIZE2. Moreover, the conjunction is not allowed to overlap with $\neg F$. That both conditions hold, is assured by the functionality of GENERALIZE1 and GENERALIZE2.

First GENERALIZE1 is called to build a conjunction M_1 from all constraints of input F . Crucial for the performance is the second generalization step, GENERALIZE2. To achieve a fast termination, the generalization must overlap a large area. Consequently, the GENERALIZE2 call removes constraints from M_1 to build a larger conjunction. Through the removal of redundant constraints, the generalization M_2 is generated. The result of GENERALIZE2 depends strongly on the order of the constraints removal.

For the third step, the currently regarded quantified variables $\mathbf{x}$ are needed. The projection is applied to the generalization M_2 , to eliminate this block of existentially quantified variables $\mathbf{x}$. Before the next iteration, the update to formulae H and O based on the projection p is executed. To update formula O the current iterations projection p_i is added as a disjunct to formula O , $O := (O \vee p_i)$. Since H needs to have the projection removed for the next iteration, the conjunction $H = (H \wedge \neg p_i)$ of H and the negated projection is built.

Since the projection of the conjunction is the key mechanism of the algorithm, we explain both steps with an example.

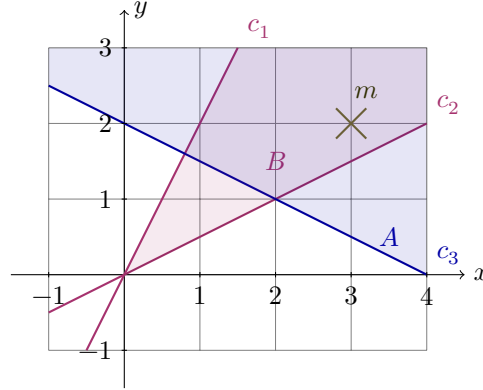


Figure 3.1: The subformulae of F are shown. The blue halfspace satisfies A . The solutions for B build the magenta cone. Moreover, the position of model $m := (3, 2)$ is marked.

Example 3.1.1 (Generalization and Projection). *The formula $\exists x \exists y (F)$ with 3 constraints is defined: $F := (c_1 \wedge c_2) \vee c_3$*

With the 3 constraints:

$$c_1 := y \leq 2x \equiv y - 2x \leq 0, \quad c_2 := 2y \geq x \equiv -2y + x \leq 0, \quad c_3 := y \geq -\frac{1}{2}x + 2 \equiv -y + \frac{1}{2}x + 2 \leq 0$$

To emphasize the Boolean structure of F , the formula is also defined as $F := B \vee A$. The left disjunct $B := (c_1 \wedge c_2)$ is a cone and the right one $A := c_3$ defines a half-space.

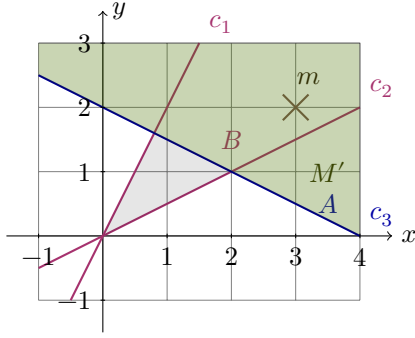


Figure 3.2: The generalization $M'=A$ is marked in green.

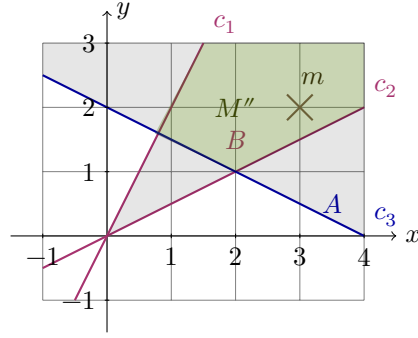


Figure 3.3: A second generalization M'' of the model m is depicted.

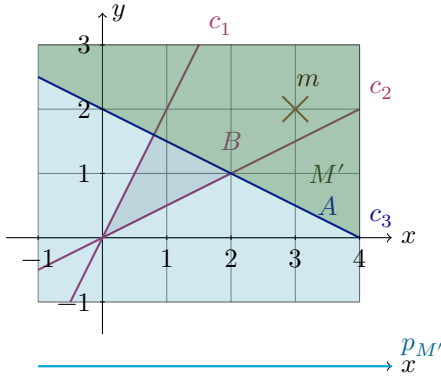


Figure 3.4: The projection $p_{M'}$ (blue) of M' (green) is visualized. Below the 2D graph, the projection is depicted with x as dimension.

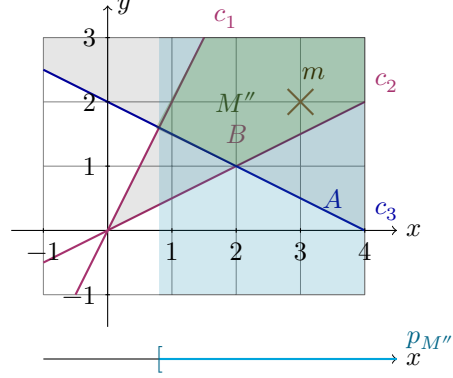


Figure 3.5: The projection $p_{M''}$ (blue) of M'' (green) is shown. Additionally, the projection $p_{M''}$ is depicted with x as dimension.

In Figure 3.1 we mark both subformulae of F and depict the chosen model $m := (3, 2)$. Multiple conjunctions are potential generalizations for the model m .

The generalizations M' and M'' for the example are visualized in Figures 3.2 and 3.3. With the first option $M' := A \equiv c_3$ and the second option $M'' := B \wedge A \equiv c_1 \wedge c_2 \wedge c_3$.

The variable y is eliminated for the projection. To compare the projection size for both generalizations, we use the Figures 3.4 and 3.5. Projecting from the generalization M' results in a termination, because the projection $p_{M'}$ is true. If the generalization M'' is chosen, the projection $p_{M''}$ does not overlap significant areas of the formula F .

The problem in this case is that M'' has the exact same conjuncts as M' , except for the addition of c_1 and c_2 . Both unnecessary conjuncts result in the smaller area overlapped by the generalization, whose projection is then subsumed.

Recognizing potential subsumptions is an important target, for having a small output and avoiding unnecessary projections.

When M' is the generalization choice, one loop iteration, and projection are needed

until the termination. To explain the behavior with generalization choice M'' , we continue with the example. The second iteration of projecting the generalization of a model is the focus.

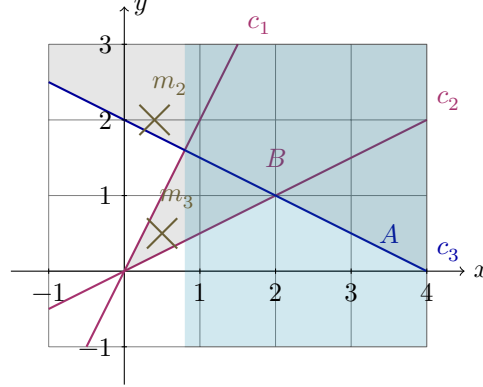


Figure 3.6: The models m_2 and m_3 of F are from areas not overlapping with the projection of M'' .

Example 3.1.2 (Consequences of the Model Choice). *In case M'' is chosen, removing the projection does leave multiple options for models of F to choose from. The models m_2 and m_3 from both potential areas are marked in Figure 3.6. The decision between the choices m_2 and m_3 is significant.*

Since the projection m_3 will not include models with $x < 0$. Consequently, the choice for m_3 leads to the execution of a third iteration. This shows the influence of the model choice on the performance.

As such model m_2 with the generalization A is chosen. Through collecting the projection of A , all prior projections are redundant.

Not every projection is necessary to build the equivalent formula F' . The *deciding models* are inside exactly one of all generalizations. The generalization must be A to include models with $x < 0$ in the projection.

Recognizing such models overlapped by the projection of one specific generalization would be practical. Instead of identifying these deciding models directly, the idea is to exclude all other areas. To find these models and project their generalization. The algorithm's performance is improved, if the search for these models is faster.

For a comprehensive overview of the significance of the model choice Figures 3.7 and 3.8 show projections with a less optimal model. Neither projection is large enough to achieve termination in the first iteration. In both cases, at least one additional iteration is necessary.

Such model choices happen because no information on the expected generalization size is considered for the decision. The SMT-solver decides independently from the need for a generalization overlapping a large area. The choice for the model is based on finding an assignment for all variables as fast as possible. This creates a strong dependency because the algorithm needs conjunctions with few constraints and a large projection for progress.

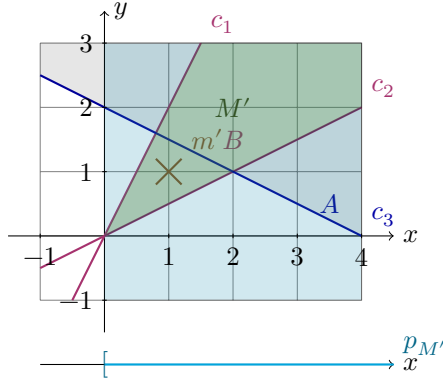


Figure 3.7: The projection $p_{M'}$ (blue) for the model m' is marked along with the generalization M' (green).

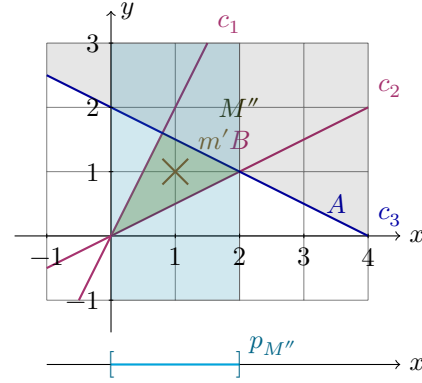


Figure 3.8: Another generalization M'' (green) for model m' is marked. The projection $p_{M''}$ (blue) is smaller than for M' .

After this first description of the algorithm's methodology, we explain the algorithm's core mechanism in depth.

3.2 In-depth Description of Relevant Behavior

Multiple *variants* of the algorithm are offered. By describing the behavior in detail, we build an understanding of the changes. Moreover, the insights will be the basis for the *statistics* collected. These are used in the analysis to explain the benchmarking performance.

Figure 3.9 gives an overview of the algorithm's *modules* and the *interactions* between them. Every loop iteration starts with checking the satisfiability of H , and if the result is *SAT*, the solver will give a model m as proof. Then based on model $m \models H$, the following three steps characteristic for the algorithm are executed. These are the core functions for the model generalization (`GENERALIZE1`, `GENERALIZE2`) and the *projection*. The behavior of the generalization is a deciding factor for the performance and will be analyzed.

To optimize those steps, the decision for a model [1] was one target of a variation in the original publication. In this thesis, the choice for variations was different, since other performance factors had been identified in tests.

These three steps are interesting targets for variations to the algorithm. To reason about the effect of different projections, two established methods, the *FM* and *FMplex* have been chosen for executing benchmarks. Moreover, the two generalization methods specified in the algorithm have been altered and both their standard behavior and the variations are detailed next.

3.2.1 The Generalization

After the model m satisfying H has been chosen, the generalization is constructed, as seen in the execution overview in Figure 3.9. The generalization is built from the constraints of F , and different potential conjunctions, satisfying a model m , can

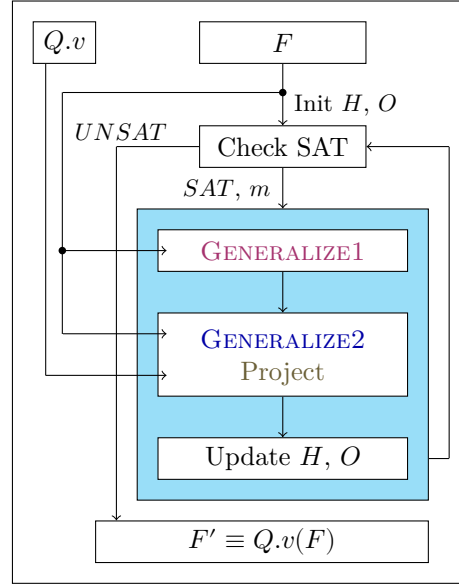


Figure 3.9: The algorithms behavior abstracted to the relevant modules. The three characteristic steps are marked.

Algorithm 2 Generalization of the model m satisfying F to a conjunction M_1 . [1].

GENERALIZE1 (F, m)

```

1   $M_1 \leftarrow true$ 
2  for all  $c \in C$ , with  $C$  all constraints of  $F$ 
3  do
4      if  $m \models c$ 
5          then  $M_1 \leftarrow M_1 \wedge c$ 
6          else  $M_1 \leftarrow M_1 \wedge \neg c$ 
7  return  $M_1$                                 //condition:  $M_1 \implies F$ 

```

be built. The generalization needs to represent a subset of the formula's models. Consequently, the inclusion of models for $\neg F$ is not allowed. As such the condition $M_1 \implies F$ needs to hold for the result.

First GENERALIZE1, building M_1 is executed. This will extend the model m to a larger area around it. The definition for GENERALIZE1 is given in algorithm 2.

For constructing M_1 all constraints of the original input formula F are collected. To achieve this behavior the code of GENERALIZE1 loops over the full set of constraints C . Before the constraint c is added to the conjunction M_1 line 4 checks $m \models c$. For this check the model m is substituted into the constraint c and then the satisfiability of c is evaluated. If the constraint c is satisfied by the model, c is directly added to the conjunction M_1 . Otherwise, if the result of the check is *UNSAT*, line 6 is executed. Then the negated constraint ($\neg c$) is considered as conjunct for M_1 . The negation satisfies model m and must be used to construct a generalization of the model.

Relevant for adding a constraint to M_1 is also, that the conjunction is projected

later. As such constraints with the comparator ($\neq$) must be treated specially. Because the projection cannot process these.

While the generalization M_1 , does include every constraint, it is not necessarily the largest generalization. Meanwhile, the number of constraints strongly influences the projection's complexity. [3, 16] Through the update mechanism, the effect is amplified. Because updates of H with the projection also influence the next iteration steps. Consequently, keeping every generalization as simple as possible is an important success factor for the algorithm.

To that extent, Monniaux [1] introduces a second generalization step, GENERALIZE2. The conjunction M_2 is smaller through the removal of redundant constraints. GENERALIZE2 iteratively checks if constraints are redundant and removes each constraint that does pass the test. Removing a constraint from the conjunction M_1 results in a larger generalization M_2 . Moreover, the removal has the direct effect of simplifying the generalization. These two benefits lead to a *less complex* and *larger* projection.

The author discussed *sufficient* conditions to check for a constraint's potential removal. The original definition applied the removal of the constraint directly to formula F . This brought the challenge of finding a suitable theory that defines this removal. As a result, the basic check for the removal for $c \in C$ is updated to $(M_1 \setminus \{c\} \wedge \neg F)$ is *UNSAT*. In algorithm 3 the GENERALIZE2 used in this thesis is defined.

Algorithm 3 Removal of redundant constraints from M_1 . The result M_2 needs to satisfy $(M_2 \wedge \neg F) \equiv false$. [1].

```

GENERALIZE2 (  $\neg F, M_1$  )
1   $M_2 \leftarrow M_1$ 
2  for all  $c \in C$ , with  $C$  all conjuncts of  $M_1$ 
3  do
4      if  $Sat(\neg F \wedge (M_2 \setminus \{c\}))$  is unsatisfiable (use of SMT-solver)
5      then remove  $c$  from  $M_2$ 
6  return  $M_2$                                 //condition:  $Sat(M_2 \wedge \neg F) = UNSAT$ 

```

This definition does not allow the conjunction without the constraint to intersect with the negated original formula $\neg F$. The original definition has been adjusted to fit the solver SMT-RAT and thus executes the removal of constraints only on conjunctions. To remove constraint c , it is replaced by *true*. Or if the constraint has been added to the SMT-solver it can be removed. The change to GENERALIZE2 does fulfill the mandatory conditions. Moreover, it benefits from having a clearly defined theoretical background.

In general, *multiple* constraints could be removed at once. No model of $\neg F$ is allowed to be also a model of the generalization. Consequently, only models of F are projected. However, finding a suitable combination of constraints without an intelligent procedure is difficult. Because an assured removal is practical, the standard removal strategy is *iterative*. Thus, the conjunction will be updated after identifying one redundant constraint.

The dependency on the exact steps before creates a strong dependency on the removal order *order* of GENERALIZE2. [1] Not having a strategy with a performance focus for the removal, made trying out different ones to adjust the generalization of one target for optimization.

We explain the execution of the model generalization with an example, to support a better understanding. The example uses the same formula as in example 3.1.1. For the calculation of the generalization, GENERALIZE1 and GENERALIZE2 are used.

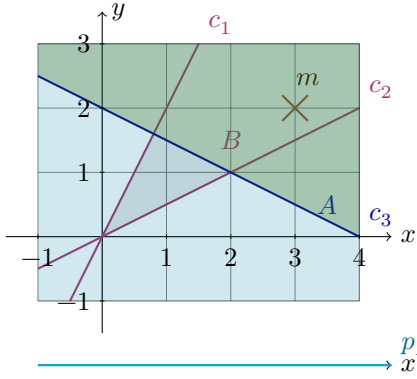


Figure 3.10: The projection p_A (blue) overlapping the entire space is visualized. Along with the generalization A (green).

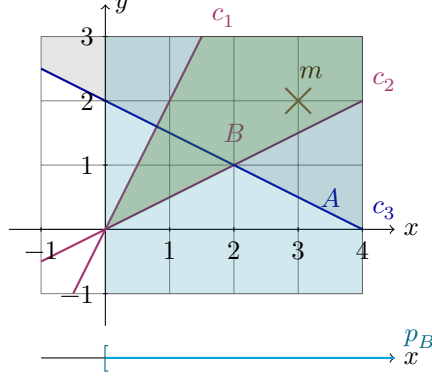


Figure 3.11: The second generalization B (green) for model m is depicted. The projection p_B (blue) of B does not overlap the area with $x < 0$.

Example 3.2.1 (Ordering of the Removal of Constraints). *The formula is $\exists x \exists y (F)$ with $F := (c_1 \wedge c_2) \vee c_3$.*

The conjunction M_1 built by GENERALIZE1 always includes all constraints. When deciding for model $m := (3, 2)$, the conjunction M_1 is $c_1 \wedge c_2 \wedge c_3$. Because the model satisfies all three constraints no negation is applied.

In a second step executed by GENERALIZE2, the redundant constraints are removed from M_1 .

Removal with the order: $c_1 < c_2 < c_3$ (from left to right)

Removal of c_i	M'_2 with removal	$\text{Sat}(\neg F \wedge M'_2)$	Update M_2
1. Remove c_1	$M'_2 := M_2 \setminus \{c_1\} \equiv B$ $\equiv c_2 \wedge c_3$	UNSAT	$M_2 := c_2 \wedge c_3$
2. Remove c_2	$M'_2 := M_2 \setminus \{c_2\} \equiv c_3$	SAT	$M_2 := c_2 \wedge c_3$
3. Remove c_2	$M'_2 := M_2 \setminus \{c_3\} \equiv c_2$	SAT	$M_2 := c_2 \wedge c_3$

Another removal with the reverse order: $c_3 < c_2 < c_1$ (from left to right)

Removal of c_i	M'_2 with removal	$\text{Sat}(\neg F \wedge M'_2)$	Update M_2
1. Remove c_3	$M'_2 := M_2 \setminus \{c_3\} \equiv c_1 \wedge c_2$	UNSAT	$M_2 := c_1 \wedge c_2$
2. Remove c_2	$M'_2 := M_2 \setminus \{c_2\} \equiv A \equiv c_1$	UNSAT	$M_2 := c_1$
3. Remove c_1	$M'_2 := M_2 \setminus \{c_1\} \equiv \text{true}$	SAT	$M_2 := c_1$

The two generated generalizations are the disjuncts A and B , of formula F . While both conjunctions are smaller than M_1 , the number of conjuncts in A and B differs. Then the quantified variable y is eliminated.

Moreover, while both are legitimate generalizations the size of their projections differs, as visible in Figures 3.10 and 3.11. The projection p_A of A overlaps the entire space and is significantly larger than that of B . This projection p_B does overlap the positive half-space including the 0 for x .

To achieve an optimal performance one must build a large projection with a small formula size. The need for a large projection stems from potential subsumptions since one projection can subsume others. This is the case in the example, with the projection of A subsuming the smaller one of B .

The order of the constraints removal has major influence on the result of GENERALIZE2. [1] In the example, two possible conjunctions A or B are calculated by GENERALIZE2. If the algorithm tries to remove c_1 first, it will be a success, but both constraints defining $B := c_2 \wedge c_3$ cannot be removed. If the algorithm uses the order c_3, c_2 and then c_1 , we find the reduced conjunction $A := c_1$, with the minimal number of constraints.

The example visualized both effects of GENERALIZE2 on the conjunction, the simplification, and the increase in size. Additionally, one potential for future improvements is contrasted by the example. While the generalization will be larger, the strategy will not find the largest possible one. The iterative approach does not find one optimal conjunction but one that cannot be reduced further.

3.2.2 Universal Quantification

While the algorithm allows for both kinds of quantification, *existential* $\exists$ and *universal* $\forall$, it cannot eliminate both types of quantification at once. To enable the elimination of universally quantified variables, $\forall.x$, the quantifier is transformed into an existential one.

The transformation $\forall.x(F) \equiv \neg\exists.x(\neg F)$, has two negations that need to be regarded. The first negation is directly to the formula F . Consequently, the algorithm is applied to the formula $\neg F$. Another negation is applied after the quantifier elimination is finished. Because one of the negations is applied to the result, the algorithm must start the quantifier elimination at the innermost nesting.

As a result, the equivalent formula $F' \equiv \forall.x(F)$ is no longer in disjunctive normal form (DNF). But $F' \equiv \neg O$ is a conjunction in conjunctive normal form (CNF).

The hierarchy of the blocks with the different quantifiers must be regarded. Each alternation results in one block of quantifiers. The quantified variables are eliminated from the *inner-most* nested block to the *outer* one. The resulting formula is equivalent regarding the exact nesting. Applying the algorithm at the innermost nesting ensures that all variables not quantified in the currently regarded block are free.

After every eliminated block, the algorithm checks if the updated formula is still quantified. Then the new innermost nested block is identified, and the quantifier elimination is repeated. Consequently, the number of repetitions is exactly the number of quantifier alternations. The implementation will handle the coordination of eliminating all blocks.

3.2.3 Removal of Not-Equals

Both projections, the FM and FMplex do not process constraints of the type *not-equals* ($\neq$). The resulting generalization would be split up and no longer a connected region. Therefore, the not-equals ($\neq$) is processed separately. Processing a constraint $P(\mathbf{x}) \neq 0 \equiv P(\mathbf{x}) < 0 \vee P(\mathbf{x}) > 0$ is done by adding one of the strict inequalities to the generalization. The decision is made by checking which constraint satisfies $m \models (P(\mathbf{x}) \sim 0)$, with $\sim \in \{<, >\}$.

3.2.4 Subsumptions

Previously, the example 3.1.1 with F from Figure 3.1 already showed the basic impact of subsumptions on GENERALIZE2. The interactions between the generalizations, formula F , and the projection are generally more complicated.

First, the subsumptions mentioned in the original algorithm and shown in the example are analyzed. This type of subsumption is about the potential generalizations for formula F . The generalization M_1 always includes all constraints of F . If another conjunction M_2 without at least one constraint c can be constructed, the first type of subsumption occurs. Since $M_1 \equiv M_2 \wedge c$, the generalization M_2 subsumes M_1 . Projecting M_1 leads to p_{M_1} subsumed by p_{M_2} . Introducing the second step GENERALIZE2, to remove redundant constraints, avoids this kind of subsumption.

Although this first kind of subsumptions is handled well, further subsumptions regarding the projection can occur. One projection can be completely included in another one. Moreover, the strategic placement of the projections is necessary for a fast termination.

The example 3.1.2 showed that a subset of the formula's models is the deciding factor for terminating the central loop. As shown in Figure 3.10 the one relevant generalization is A . Other iterations generalizations and projections are redundant. When projecting the chosen generalization, the projection can subsume other already executed projections from prior iterations. Ideally the algorithm would be able to identify relevant models and thus recognize the best choice for the projection.

These subsumptions have a negative performance impact, because of multiple factors. The first one is that projecting formulae in LRA is complex, and avoiding unnecessary projections improves the performance. Another impact is the potential for simplification of both formulae H and O after finding the projection subsuming prior ones.

Subsumptions cannot be avoided, since the cause is also the model choice. Ideally, the model would be picked with a strategy that identifies crucial areas for projection. However, the model choice does not get guidance except to be outside of areas overlapped by a projection. Consequently, the decisions made by the SMT-solver could benefit from a heuristic tailored to the algorithm.

However, other improvements to the algorithm are possible through intelligent changes. Improvements to the standard version are achieved by focusing on the performance-relevant parts. This means that GENERALIZE2 is a target for optimization. Or an overall decrease in complexity through pre-processing the input. We define all variants and explain them with the help of examples in the next chapter.

Chapter 4

Variations of the Algorithm

Multiple relevant performance influences, the subsumptions, and model choice, have been contrasted. Improvements to the algorithm depend on how well these challenges are handled or avoided. Crucial is, to produce an idea of how to avoid the worst case when the cause is unclear. Trying to focus on the mechanism for *reducing the conjunctions' number of constraints* was difficult. Even after multiple attempts to identify a suitable heuristic for ordering the constraints for removal, no general success was achieved.

The cause of the problem is that the algorithm's complexity does not scale directly with the conjunction but with their projection. However, without intelligent placement, not even large projections guarantee a fast termination. As a result, the choice for the conjunction can only give a tendency to how the algorithm will perform.

Strong factors identified for adjustments are the iterative removal of constraints in GENERALIZE2. One factor is adopting information from the satisfiability check, which improves the quality of the generalization. Another bottleneck for the execution is complex input, characterized by many variables and constraints. The problems of GENERALIZE2 are amplified by complex input. More input constraints increase the number of findable generalizations. Additionally, the removal needs to consider more constraints.

Considering these *complex interactions* between the functions, multiple *key steps* are adjusted together for performance advancement. To improve the basic definition of the algorithm problematic, behavior is highlighted and then contrasted with the adjusted one. For all chosen variants, the benefits and targets will be explained by looking at the examples

4.1 Rotate and Repeat

One target for performance improvement is the decision strategy for GENERALIZE2. The quality of GENERALIZE2 decisions is not ensured to be optimal, because of two reasons. These are the *iterative execution* of GENERALIZE2, in combination with the optimization target to calculate conjunctions with few constraints.

We motivate the need for the strategy through a short example. The example details problematic behavior and uses already-seen constraints but makes a few changes. First, one additional cone along with a new constraint is introduced. Moreover, the

formula's Boolean skeleton is changed. So that the formula is satisfiable for *two overlapping triangles*. The placement of both triangles purposefully forces the need for a least *two projections*, one for each of the triangles. To emphasize potential problems, GENERALIZE2 is assumed to generate the exact conjunction M_2 given, without specifying the order.

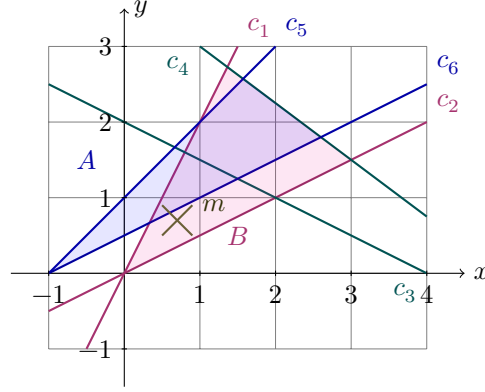


Figure 4.1: The formula $F := A \vee B$ is depicted. With the two triangles A (blue) and B (magenta) marked separately. The model m satisfying F will be generalized.

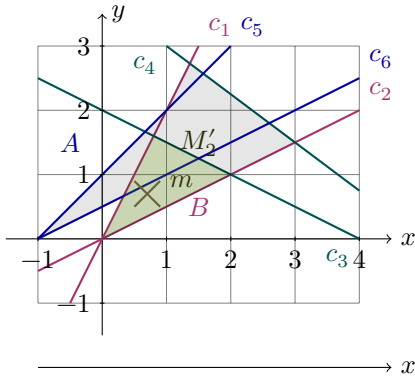


Figure 4.2: The generalization $M'_2 := c_1 \wedge c_2 \wedge c_3$ of model m is marked in green.

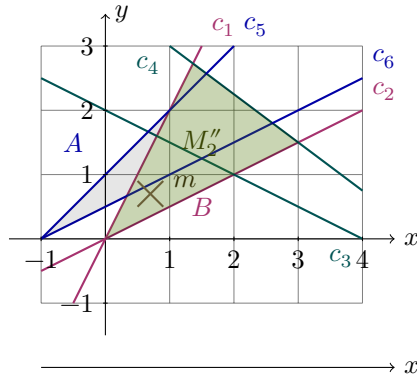


Figure 4.3: The generalization $M''_2 := B$ of model m is marked in green.

Example 4.1.1 (Motivation for 'Rotate & Repeat'). *Figure 4.1 graphically shows the example formula F , the disjunction of both triangles A and B.*

The constraints c_1, c_2 as in example 3.1.1: $c_1 := y - 2x \leq 0$, $c_2 := -2y + x \leq 0$

Constraint c_3 is adjusted to: $c_3 := y \leq -\frac{1}{2}x + 2 \equiv y + \frac{1}{2}x - 2 \leq 0$

Added are the constraints: $c_4 := y \leq \frac{15}{4} - \frac{3}{4}x \equiv y + \frac{3}{4}x - \frac{15}{4} \leq 0$

$c_5 := y \leq 1 + x \equiv y - x - 1 \leq 0$, $c_6 := \frac{1}{2} + \frac{1}{2}x \leq y \equiv \frac{1}{2} + \frac{1}{2}x - y \leq 0$

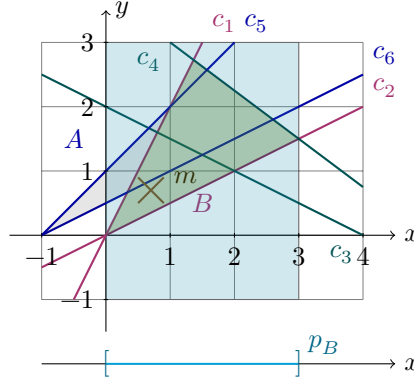


Figure 4.4: The projection p_B (blue) of B is visualized in both 2D and below on the x-axis.

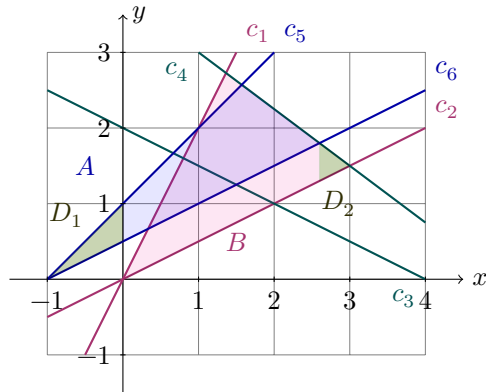


Figure 4.5: The two areas D_1 and D_2 deciding for termination are marked in green. If the projections overlap both areas, the algorithm terminates.

The quantified formula $\exists.x\exists.y(F)$, with $F := (((c_1 \wedge c_2) \wedge (c_3 \vee c_4)) \vee ((c_5 \wedge c_6) \wedge (c_3 \vee c_4)))$ uses 6 constraints c_i , $i \in \{1, 2, \dots, 6\}$. From GENERALIZE1 the conjunction $M_1 := c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge \neg c_6$ is defined.

Two generalizations M'_2 and M''_2 for the model m are given in Figures 4.2 and 4.3. The need for repetition arises from problematic generalizations like M'_2 as given in Figure 4.2. For the projection, the variable y is eliminated. Since the projection of $M'_2 := c_1 \wedge c_2 \wedge c_3$ does not overlap B or A at least two more iterations are necessary.

To depict an optimal execution the generalization $M := B = c_1 \wedge c_2 \wedge c_4$, marked in Figure 4.3 is projected. The projection, visualized in Figure 4.4 will remove the entire triangle B from consideration for the next model. In the next iteration, then a model for A is chosen and the algorithm can terminate fast.

The example gives an intuition for the need for repetition and rotation. Executing GENERALIZE2 can result in model generalizations, that cause redundant projections. To analyze the reason for the redundancies, we explain which projections are needed for termination.

Example 4.1.2 (Projections Deciding for Termination). *For the termination, both areas marked in Figure 4.5 by D_1 and D_2 need to be included in a projection. For simplicity, all generalizations for D_1 will include $c_5 \wedge c_6$. The regarded conjunctions to generalize models from the area D_2 include $c_1 \wedge c_2$.*

If the model is in area D_1 the conjunctions $c_5 \wedge c_6 \wedge c_4$, $c_5 \wedge c_6 \wedge c_3$ or also $c_5 \wedge c_6 \wedge \neg c_1$ are generalizations. Models from D_2 are included in the conjunction $c_1 \wedge c_2 \wedge c_4$.

The variable y is eliminated for the projection. Combining the projections of these generalizations will result in termination. All projections until the deciding ones including either D_1 or D_2 are redundant. However, they guide the model choice, until the deciding areas are found and a model from D_1 or D_2 is chosen.

Before deciding on the changes needed, the test results with the standard version are considered. Executions with a random order are compared to ones with a standard ordering. The potential negative impact of the implemented strict ordering can be identified in the logs. Afterward, different strategies to reorder the constraints have been regarded.

First basic options like shuffle or reverse have been trialed. However, randomization can lead to worse-case behavior. This was avoided by repeating the core steps. The final variant repeats GENERALIZE2 and the projection once. For the *first* execution, the reordering is a *randomization*. The *second* one has a more *deliberate reordering* utilizing *rotating* the constraints. For the rotation, the constraints included in the conjunction for the first execution are chosen.

The executed rotation has the following definition:

Definition 4.1.1 (Rotating the Ordering of Constraints). *The constraints $c_i \in C$, with $i \in \{1, 2, \dots, |C|\}$ defining M_1 are ordered by their index.*

For memorizing the ordering a sorted list L is used, e.g. $L := 1 < 2 < 3 < 4 < 5 < 6$ with priority from left to right.

Rotating c_i with index i using list L is done by:

Removal from list: $L' := L \setminus \{i\}$

Adding with highest priority for removal: $L'' := i < L'$

The conclusion is that the search for *deciding conjunctions* is best achieved by a decisive change in strategy. Multiple repetitions of GENERALIZE2 increase the

chances of finding necessary generalizations. The combination with randomization of the constraints ordering after GENERALIZE1 offers a weaker dependency between iterations of GENERALIZE2. So, the variant to GENERALIZE2 improves the ordering dependency.

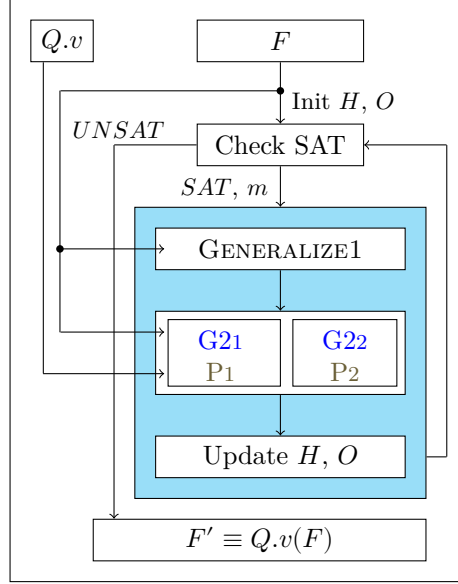


Figure 4.6: The algorithms behavior with the variant 'Rotate & Repeat'. GENERALIZE2 (G2) and the projection (P) are applied twice. With the input conjunction M_1 from the same model m .

Moreover, the repetition leads to a robust performance, with fewer dependencies on a suitable input. To implement the wanted behavior, GENERALIZE2 is executed multiple times for one model as seen in Figure 4.6. Moreover, by rotating out the constraints chosen for the other executions of GENERALIZE2, the influence of one bad ordering will be minimized.

The final variant is based on the test results. First, the constraints ordering is randomized and then one repetition with the rotating for reordering is executed. The functionality and benefits of the strategy are examined by giving an example.

Example 4.1.3 (Rotating the Ordering of Constraints). *To this regard, the standard GENERALIZE2 and a 'Rotate & Repeat' execution with 2 repetitions is compared. For the formula F of example 4.1.1 we choose model $m := (1, 1)$, with generalization $M_1 := (c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6)$.*

Specified order for removal: highest to lowest index, $L := 6 < 5 < 4 < 3 < 2 < 1$

GENERALIZE2 ('Standard'): $M_2 := c_1 \wedge c_2 \wedge c_3$

GENERALIZE2 ('Rotate & Repeat'):

1st Round: $M'_2 := c_1 \wedge c_2 \wedge c_3$; Order: $L := 6 < 5 < 4 < 3 < 2 < 1$

2nd Round: $M''_2 := c_4 \wedge c_5 \wedge c_6$; Order: $L := 3 < 2 < 1 < 6 < 5 < 4$

With the use of 'Rotate & Repeat', one conjunction including models from the deciding area D_1 is found fast. The other conjunction for the area D_2 will follow in

the second iteration. Projecting with the standard GENERALIZE2 does not include models of either D_2 or D_1 yet.

Even in the worst case, starting with the smallest possible generalization, the use of 'Rotate & Repeat' will lead to fast improvements.

Example 4.1.4 (Benefits of the Rotation). *Chosen model $m := (1, 1)$, with generalization $M_1 := (c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6)$.*

Specified order for removal: $L := 5 < 2 < 4 < 6 < 1 < 3$

GENERALIZE2 ('Standard'): $M_2 := c_1 \wedge c_3 \wedge c_6$

GENERALIZE2 ('Rotate & Repeat'):

1st Round: $M'_2 := c_1 \wedge c_3 \wedge c_6$; Order: $L := 5 < 2 < 4 < 6 < 1 < 3$

2nd Round: $M''_2 := c_1 \wedge c_2 \wedge c_4$; Order: $L := 6 < 1 < 3 < 5 < 2 < 4$

Consequently, the adjustments offer more independence from both the ordering of the removal and the model choice. Importantly, having multiple projections for the same model causes the projections to overlap. Repeating the projection increases the chance for the projection to overlap a deciding area.

Together with a *randomization* of the constraints (*Shuffle*) before the first round, 'Rotate & Repeat' is defined as the variant 'RRShuffle'. Since formulae in benchmarks are of different complexity, this effect must be acknowledged. The strategy 'RRShuffle' creates a strong overhead, in case it is not needed. The next variant focuses stronger on adjusting the input, so that the runtime is improve in general.

The following adjustment partitioning the formula into multiple subformulae, is a decisive extension to reduce the calculation's complexity. The trade-off is that the algorithm must be executed more often.

4.2 Splitting up the Execution

This variation 'Splitting' attempts to improve the algorithm's *scaling* with *complex formulae*. Multiple factors play a role when estimating the scaling efficiency, with dependencies on all three core steps of the loop and the used SMT-solver.

For the solver, the complexity of evaluating the satisfiability of a quantified formula is dependent on the number of variables and the number of constraints in a formula. Additionally, the performance of GENERALIZE1 and GENERALIZE2 had been identified as a bottleneck during tests. Both functions profit from fewer constraints.

The approach is based on pushing the quantification past the outermost logic operator. For this, the rule on *quantifier distribution* must be applied to the input. Then the quantifier elimination is executed for *each quantified subformula* separately.

The benefits of the variant 'Splitting' are demonstrated by an example. The quantified formula is the same as in the example 4.1.1, two satisfiable overlapping triangles.

Example 4.2.1 (Split up of the Formula). *The formula $\exists x \exists y (F) := \exists x \exists y (A \vee B)$ is a disjunction.*

For the split up of the formula, the distribution of the quantification is used: $\exists x \exists y (A \vee B) \equiv (\exists x \exists y (A)) \vee (\exists x \exists y (B))$

Figure 4.7 depicts formula F . To explain the split up for F , the same steps are executed again. Just with F defined on the 6 constraints c_i , $i \in \{1, 2, 3, 4, 5, 6\}$.

$F := (((c_1 \wedge c_2) \wedge (c_3 \vee c_4)) \vee ((c_5 \wedge c_6) \wedge (c_3 \vee c_4)))$

The result of applying the law to quantifier distribution is:

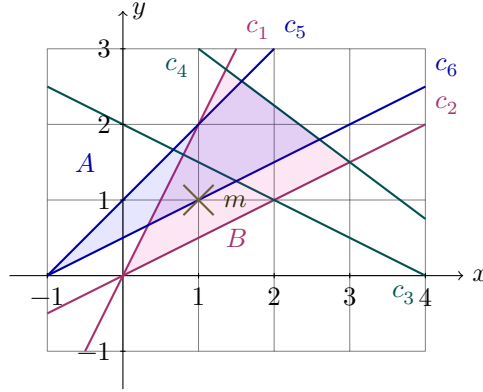


Figure 4.7: The formula $F := A \vee B$, the disjunction of two triangles A (blue) and B (magenta) is depicted.

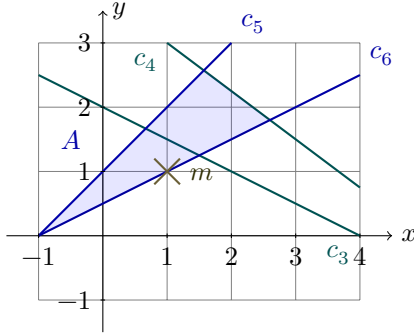


Figure 4.8: The first result of the split is formula A (blue).

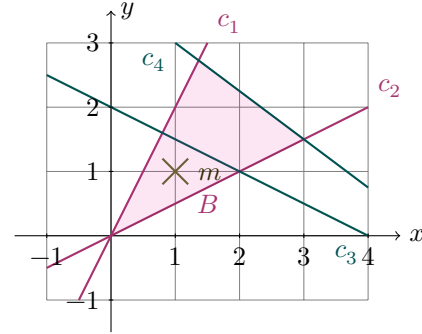


Figure 4.9: The second result of the split is formula B (magenta).

$$\begin{aligned} & \exists x \exists y (((c_1 \wedge c_2) \wedge (c_3 \vee c_4)) \vee ((c_5 \wedge c_6) \wedge (c_3 \vee c_4))) \\ & \equiv (\exists x \exists y ((c_1 \wedge c_2) \wedge (c_3 \vee c_4))) \vee (\exists x \exists y ((c_5 \wedge c_6) \wedge (c_3 \vee c_4))) \end{aligned}$$

Figures 4.8 and 4.9 and provide a visual explanation. Before the split up, multiple generalizations for the model m have been possible. For each of the subformulae A and B , using 'Rotate & Repeat' can find an optimal generalization within one execution.

These potential benefits seen in the example drive the decision for the variation with splitting. The scaling with the number of constraints and variables is improved. The idea is that the subproblems will have *fewer quantified variables and constraints*. The quantified subformulae are less complex than the original formula. However, the *lowered complexity* comes from *increasing the number of quantified formulae* to which the algorithm is applied.

This decision to split the quantified formula into multiple less complex subproblems needs careful consideration. Certain trade-offs, arising from different factors, will influence the performance with variant 'Splitting'. One important factor is the complexity of the subproblems that must be solved. Without the change, each subproblem can be overlapped by other projections. Complex subproblems are never

solved directly if a projection overlapping them is found before.

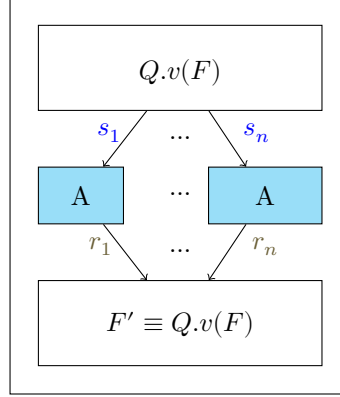


Figure 4.10: The input formula $Q.v(F)$ is split up into the subformulae $\{s_1, \dots, s_n\}$. After applying the algorithm (A) to all subformulae, the equivalent formula F' is constructed from each output $\{r_1, \dots, r_n\}$.

Both existentially and universally quantified formulae can be split up, as shown in the model in Figure 4.10. For universally quantified formulae, the split-up is executed before the quantifier transformation, which is applied afterward for each subformula.

In addition to splitting up of the formula F a refinement, has been developed. To emphasize the need for a transformation the effect is explained by considering a formula $\exists.x\exists.y(F')$, unsuitable for splitting. The example forces the capability to split up the quantified formula, by transforming F' to F .

Example 4.2.2 (Split-up with a Refined Transform). *The quantified formula $\exists.x\exists.y(F')$ with $F' := (((c_1 \wedge c_2) \vee (c_5 \wedge c_6)) \wedge (c_3 \vee c_4))$ defines the same satisfiable area as seen in the example 4.2.1. Because F' is a conjunction, no distribution of the existential quantifier is possible. Because the outer logic operator is an $\wedge$ (and). To split up the formula, the transformation of F' to F below is applied.*

For the formula $F' := (((c_1 \wedge c_2) \vee (c_5 \wedge c_6)) \wedge (c_3 \vee c_4)) \equiv ((A \vee B) \wedge C)$, three subformulae $A := (c_1 \wedge c_2)$, $B := (c_5 \wedge c_6)$ and $C := (c_3 \vee c_4)$ are defined.

Apply the distributive law:

$$F' \equiv ((A \vee B) \wedge C) \equiv ((A \wedge C) \vee (B \wedge C)) \equiv F$$

Use the definition of A, B and C:

$$F' \equiv ((A \wedge C) \vee (B \wedge C)) \equiv (((c_1 \wedge c_2) \wedge (c_3 \vee c_4)) \vee ((c_5 \wedge c_6) \wedge (c_3 \vee c_4)))$$

Split-up of $\exists.x\exists.y(F')$:

$$\begin{aligned} & \exists.x\exists.y(((c_1 \wedge c_2) \wedge (c_3 \vee c_4)) \vee ((c_5 \wedge c_6) \wedge (c_3 \vee c_4))) \\ & \equiv (\exists.x\exists.y((c_1 \wedge c_2) \wedge (c_3 \vee c_4))) \vee (\exists.x\exists.y((c_5 \wedge c_6) \wedge (c_3 \vee c_4))) \end{aligned}$$

The refinement is not added to the tests, because the actual transformation needed to improve the performance is complex. If multiple subformulae are candidates for applying the distributive law, the runtime can differ significantly. Depending on the subformula chosen for the transformation.

First tests with careful fine-tuning showed that the variation to the split-up performed well on a specific input. However, the general performance is too low to

include the variant in the tests. Still, refining the split-up by a transformation can be useful if employed specifically for a certain input.

During the first check of the splitting benchmarks, the conclusion was that focusing on a subsection of the formula does bring more benefits than expected. Consequently, another variant was developed with a similar mechanism.

4.3 Focus on the Active Constraints

The split up of the formula lowers the complexity by focusing on smaller subformulae for the quantifier elimination. Also, this strategy is beneficial if the formula has a suitable structure. One downside is that the quantifier elimination is executed separately for each subformula.

More practical would be to *focus on subformulae*, that are simple to solve. So instead of partitioning the formula into smaller subformulae, the *attention* to a *smaller part* of the formula is implicitly forced. In this regard, the conjunction is built concerning the subset of constraints deciding the model choice.

Through the concentration on a subset of all constraints, the complexity of the generalization choice is reduced. Moreover, the algorithm then *follows the reasoning of the SMT-solver*. This strategy is beneficial because the SMT-solver can strategically focus on simple subformulae. Since the entire calculation of the generalization is changed, the variant is called 'Generalize1'.

Multiple generalizations can be found for each model, but determining their quality is more difficult. Because finding the best generalization is not possible, the search needs to be fast and strategic. Thus, the guidance of the solver when building the generalizations is necessary. Before defining this variant 'Generalize1', an example illustrates the possibility of finding multiple generalizations with a similar quality.

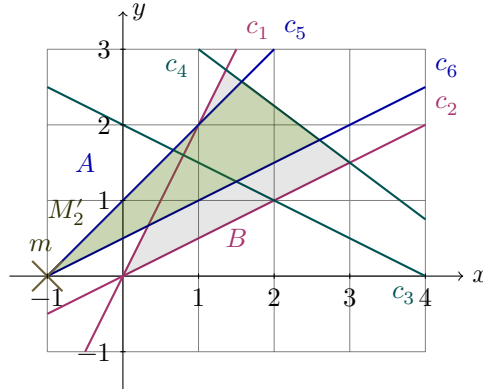


Figure 4.11: The first generalization $M'_2 := A$ (green) of model $m := (-1, 0)$.

Example 4.3.1 (Multiple Generalizations for a Formula). *The example is again on formula $F := A \vee B$, defined by the two satisfiable triangles. The generalization M_1 before the optimization is $\neg c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6$. Then the standard strategy needs to call the SMT-solver to remove redundant constraints. So, removing three constraints will lead to at least three additional solver calls.*

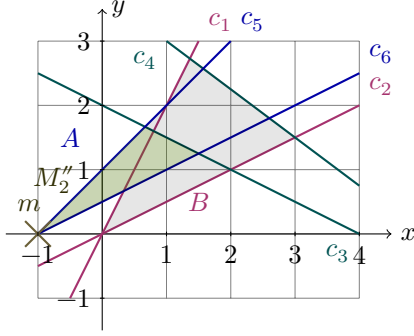


Figure 4.12: The second generalization $M_2'' := c_5 \wedge c_6 \wedge c_3$ (green) of model $m := (-1, 0)$.

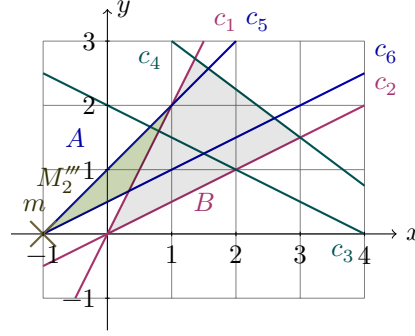


Figure 4.13: The second generalization $M_2''' := c_5 \wedge c_6 \wedge \neg c_1$ (green) of model $m := (-1, 0)$.

Figures 4.11, 4.12 and 4.13 show three possible generalizations of model $m := (-1, 0)$. All three generalizations M_2' , M_2'' and M_2''' are already minimal. If the variable y is eliminated for all conjunctions, the largest projection is generated for M_2' . Additionally, removing the projection of $M_2' := A$ for choosing the next model assures termination in the second iteration. Because the next model must be from the second deciding area.

However, the other two conjunctions' quality is similar, regarding the interactions between multiple iterations. These always need a second iteration with another model. When we choose B in the second iteration as a generalization, the algorithm will terminate for all three conjunctions. Consequently, the only difference between the generalizations is the time spent calculating them.

So, as long as one generalization is found faster, performance improvements are expected. Because not one most ideal generalization is necessary. An adjustment to reduce the time spent on GENERALIZE2 is needed. By focusing on fewer constraints, the calculation of the final generalization is simplified. The idea is to adopt the decisions of the SMT-solver for the model choice. As such we explain the new functionality, which considers the functionality of SMT-RAT. We describe how to identify the constraints relevant to the model decision from the SMT-solver.

To this extent, we consider the *last cycle of logic and theory solving*. In this cycle, the *model* was calculated to prove satisfiability. The *SMT-solver* starts each cycle with logic solving. Since SMT-RAT [12] is a *less lazy* SMT-solver, the assignment for the Boolean abstraction of F is *partial*. The theory solving regards only the active constraints, assigned in the Boolean skeleton.

So, the plan is to do the same for the generalization. Through this decision the number of constraints after the GENERALIZE1 call is lowered. Having fewer constraints in M_1 decreases the complexity of the GENERALIZE2 call.

For the variant 'Generalize1', the generalization is built from the *active constraints*. *Passive constraints* are removed directly during the GENERALIZE1 execution.

Definition 4.3.1 (Active Constraints as Generalization). *The generalization M_a constructed from active constraints does fulfill both necessary conditions:*

- $M_a \implies F$

- $Sat(\neg F \wedge M_a) = UNSAT$

We proof the correctness of this statement by explaining that all passive constraints are redundant.

Proof 4.3.1 (Building the Generalization from Active Constraints).

First we regard how the constraints are included in the active constraints C_a . These depend on a partial assignment for the formula's Boolean skeleton. Constraints represented by a Boolean variable with an assignment are active. For simplification, we handle all negated constraints with the following strategy. If an active constraint c is represented by a Boolean variable with the assignment false, the negation $\neg c$ is added to C_a .

The full assignment is built from the partial one. In this regard, all further Boolean variables are assigned at random. With the same strategy, the set of active constraint C_a can be extended to a full set of all constraints.

Because $F \wedge \neg F \equiv \text{false}$, this must also hold for the Boolean skeleton of F and $\neg F$. Consequently, any (partial) assignment sufficient to proof the satisfiability of F cannot be satisfiable for $\neg F$. Moreover, this also holds for the set of active constraints C_a . Consequently, the solutions for conjunction M_a of constraints C_a do not include any models of $\neg F$.

After explaining the necessary background, we proof that M_a does fulfill both conditions for the generalization.

Let $M_a := \bigwedge_{c_a \in C_a} c_a$ be a generalization, with C_a the set of active constraints. The set of passive constraints is C_p , and C are the constraints of F . We show that the condition $M_a \implies F$ holds.

In case $M_a \equiv \text{false}$: The implication $M_a \implies F$ is always true

The proof in case M_a is satisfiable is more involved. We start by building a generalization M_1 , which includes all constraints C . For this step, we need the definition that all constraints added to C_a must satisfy m .

Since the model m is determined concerning the active constraints C_a , all of them are added to M_1 . The further constraints $c_p \in C_p$ are added directly, if $m \models c_p$ is true. Otherwise, the constraint $\neg c_p$ is a conjunct of M_1 .

Now we apply the knowledge from the theory solving to remove redundant constraints. We know, that the conjunction of active constraints M_a does not include models of $\neg F$. So the condition $M_a \implies F$ holds, independent from adding or removing passive constraints C_p to the conjunction M_a . Therefore, the passive constraints can be added either negated or directly without influencing the result.

This allows the removal of the passive constraints $c_p \in C_P$ from M_1 . As long as all active constraints are included in the conjunction with removals. Consequently, all passive constraints are removed directly. The condition $Sat(\neg F \wedge (M_1 \setminus \{c_p\})) = UNSAT$ holds for the removal of each $c_p \in C_p$.

Since all passive constraints are removable, building the generalization M_1 directly from the active constraints is an option.

The functionality of the SMT-solver is analyzed, to implement the removal of constraints. The first idea, to focus on the active constraints is difficult to implement. The simpler strategy is to choose the constraints based on the model given by the SMT-solver.

The formula is converted to *negation normal form* (NNF), to reproduce which constraints are important for the solver decision. Since SMT-RAT has a specific NNF

implemented, this definition of the NNF with *no negations* is used. If a constraint c is negated in the NNF, $\neg c$ is replaced by $c' \equiv \neg c$.

All active constraints must be satisfied by the model. Afterward, the subset of constraints satisfying the model is identified. However, no distinction between active and passive constraints is made. The model can but does not have to satisfy passive constraints. As such no distinction is possible.

So, for the variant ‘Generalize1’ the constraints for the generalization are collected from the NNF. The following special definition of the NNF is used.

Definition 4.3.2 (Negated Constraints in the NNF). *The negation normal form (NNF) has no occurrences of negated constraints $\neg c_i$.*

Every constraint $\neg c_i$ is replaced by the equivalent constraint c'_i , with $c'_i := \neg c_i$.

After the transformation to NNF, the formula’s constraints are collected. Then the standard GENERALIZE1 call is replaced by an adjusted one. The change ensures that GENERALIZE1 builds a conjunction of the relevant and not all constraints.

The following definition will formalize the strategy to identify these constraints.

Definition 4.3.3. (Identification of Relevant Constraints)

The constraints C_{NNF} are extracted from the specified NNF of a formula.

Constraint $c_i \in C_{NNF}$ is relevant if $m \models c_i$, the constraint is satisfied by the model m .

All relevant constraints build the set C_r , which is used to construct the generalization M_1 in the GENERALIZE1 call.

The new behavior builds the conjunction of all relevant constraints. To ensure the described behavior, algorithm 4 for GENERALIZE1 is updated.

Algorithm 4 The generalization of the model m is executed with all conjuncts of F_{NNF} . The formula F_{NNF} is the special NNF without negations for F . The result is the conjunction M_1 constructed from a subset of all constraints.

```

GENERALIZE1-ONLYSAT (  $F, m$  )
1   $M_1 \leftarrow true$ 
2   $F_{NNF} \leftarrow$  Transform  $F$  to NNF
3  for all  $c \in C$ , with  $C$  all constraints of  $F_{NNF}$ 
4  do
5      if  $m \models c$ 
6          then  $M_1 \leftarrow M_1 \wedge c$ 
7  return  $M_1$                                  $// M_1 \Rightarrow F$ 

```

If GENERALIZE1 is called, then the formula F in NNF is satisfiable and the set of constraints C_{NNF} must include at least one relevant constraint c_i . Otherwise, for formulae without NNF the behavior of the updated GENERALIZE1 is undefined.

With the help of an example, we display the need for the formula to be in NNF. By attempting to generalize the formula F' not in NNF, with $F' \equiv F$.

Example 4.3.2 (Violation of Condition for Generalization). *The formula $\exists x \exists y (F)$, with $F := (c_1 \wedge c_2) \vee c_3$ is visualized in Figure 3.1. For the explanation F' , with the constraints $c'_1 := \neg c_1$ and $c'_2 := \neg c_2$ is defined.*

$$F' := \neg(\neg c_1 \vee \neg c_2) \vee c_3 \equiv \neg(c'_1 \vee c'_2) \vee c_3$$

Generalizing the model $m = (0, 0)$ with the variant results in M'_1 , and as comparison M_1 is calculated with the standard procedure.

For this the check $m \models c$ and $c \in \{c'_1, c'_2, c_3\}$ is executed. In the standard version the constraint c is added to M_1 either directly or negated. The updated version builds M'_1 from constraints with a SAT result.

Add Constraint	Result	Update M_1	Update M'_1
1. Check for $m \models c'_1$	UNSAT	Add $\neg c'_1$ to M_1 , $M_1 := \neg c'_1$	no changes to M'_1 , $M'_1 := \text{true}$
2. Check for $m \models c'_2$	UNSAT	add $\neg c'_2$ to M_1 , $M_1 := \neg c'_1 \wedge \neg c'_2$	no changes to M'_1 , $M'_1 := \text{true}$
3. Check for $m \models c_3$	UNSAT	add $\neg c_3$ to M_1 , $M_1 := \neg c'_1 \wedge \neg c'_2 \wedge \neg c_3$	no changes to M'_1 , $M'_1 := \text{true}$

Generalization (standard): $M_1 := \neg c'_1 \wedge \neg c'_2 \wedge \neg c_3 = c_1 \wedge c_2 \wedge \neg c_3$

Generalization (optimized): $M'_1 := \text{true}$, since no conjuncts have been added.

Violation of the defined condition $M'_1 \implies F'$, since the model $m' := (-1, -1)$ for M'_1 is no model for F' . The wrong form of F' causes errors in the generalization.

The planned adjustment ‘Generalize1’ will be outlined by an example. The example is on the same formula as already shown in the figure 3.1. Again, we contrast both the standard and adjusted GENERALIZE1 approaches.

Example 4.3.3 (Generalization with the Variant). The quantified formula in NNF, is again: $\exists x \exists y. y(F)$, with $F := (c_1 \wedge c_2) \vee c_3$

The model $m := (0, 0)$ is generalized to M_1 by the standard GENERALIZE1. Additionally, the conjunction M'_1 is built by the optimized version:

Add Constraint	Result	Update M_1	Update M'_1
1. Check for $m \models c_1$	SAT	Add c_1 to M_1 , $M_1 := c_1$	add c_1 to M'_1 , $M'_1 := c_1$
2. Check for $m \models c_2$	SAT	add c_2 to M_1 , $M_1 := c_1 \wedge c_2$	add c_2 to M'_1 , $M'_1 := c_1 \wedge c_2$
3. Check for $m \models c_3$	UNSAT	add $\neg c_3$ to M_1 , $M_1 := c_1 \wedge c_2 \wedge \neg c_3$	no changes to M'_1 , $M'_1 := c_1 \wedge c_2$

Generalization (standard): $M_1 := c_1 \wedge c_2 \wedge \neg c_3$

Generalization (optimized): $M'_1 := c_1 \wedge c_2$

This generalization M'_1 has one constraint less than the M_1 with the standard GENERALIZE1 technique. The constraint c_3 is not relevant. Because both $c_1 \wedge c_2 \wedge \neg c_3$ and $c_1 \wedge c_2 \wedge c_3$ have no overlap with $\neg F$. Consequently, a performance improvement is expected.

The example showed that the optimized strategy leads to a conjunction with fewer constraints. However, the benefits depend on the chosen model, if the model $m := (3, 2)$ is generalized, both versions calculate the same result, $M_1 \equiv M'_1 \equiv c_1 \wedge c_2 \wedge c_3$.

The benefits are based on the exclusion of constraints before the GENERALIZE2 call. As described, the GENERALIZE2 call significantly influences the runtime. Thus, the option to exclude constraints intelligently leads to improvements.

Benchmarks check the exact effect on the performance later. Before the *benchmarking*, we describe the *implementation* of the algorithm and the variants in SMT-RAT.

Chapter 5

Implementation in SMTRAT

5.1 Overview of the Variations

The algorithm is implemented as a *quantifier elimination strategy* into SMT-RAT [2]. Whenever possible the already existing functionality is adopted into the code. The implementation improves the readability and flexibility of the code through the structuring. The abstractions given in Figures 3.9, 4.6, and 4.10 are considered, by the implementation.

For example, the code must be able to repeat the *generalization* and *projection* with different orderings for the constraints. By joining the steps GENERALIZE2 and the projection into one function call, simple access to this functionality is achieved. Switching between both options for the GENERALIZE1 call is done by setting a Boolean flag.

The modernized structure enables *fast* and *simple adjustments*. The final code is published on the SMT-RAT repository [2], with the *variants* set to the necessary *parameters*.

All variants of the algorithm are implemented and will be presented. Options inside the code activate the variants and enable *combinations* of *multiple variants*. Activating the variants inside the code offers simple access to the standard version of the algorithm. Still, meaningful and interesting choices are possible with the advanced option to edit the pre-set options.

The code utilizes both *FM* and *FMplex* of SMT-RAT for the projection. This allows us to gain significant insights into the effectiveness of the FM and FMplex. Enabling the comparison of multiple quantifier elimination methods was one of the core targets when implementing the algorithm. If further projections need to be studied, the code's existing structure can be extended to allow other options. Such changes are simple because the variables for swapping between multiple choices are already in place.

Furthermore, we provide the option to choose between the *Simplex* and *LRA modules* as SMT-solver. The decision is between the LRA strategy, an established module, and the newly implemented Simplex strategy. Keeping the solver strategy flexible allows quick adjustments to the implementation. Moreover, the tests with different strategies give additional insights into the algorithm's behavior. Importantly conclusions about the compatibility with other solvers are enabled. Having the option will play a crucial role in deciding if an extension of the algorithm to use other SMT-

solvers e.g. z3 [17] is an interesting target for future research.

For the analysis relevant options are the ones that will influence the algorithm's performance. Additional functionality is introduced into the algorithm, to achieve a faster runtime. The variants target different problems. Activating variants is done by adjusting the parameters needed. Since multiple options were trialed before the final decision, a change to customize the variants based on knowledge about the input is possible. For example, the 'Shuffle' could be replaced by a reversal of the ordering.

Table 5.1 gives a summary of all presented variants. The first variant, 'RRShuffle' attempts to *decrease the influence of randomness*. In the 'Standard' variant the runtime depends strongly on the static ordering of constraints for the GENERALIZE2 call. The influence of the ordering is minimized, through the randomization after GENERALIZE1. Moreover 'RRShuffle' repeats the steps GENERALIZE2 and the projection. Multiple options are implemented to rearrange the constraints ordering before the second generalization. These are, for example rotating, shuffling, and reversing. Because the performance for the rotating was the best, this option is in the final decision.

Another identified performance influence was the size of the input formula. The variant 'Splitting' *distributes the quantification*, to avoid processing large formulae. Then the algorithm is applied to the quantified subformulae. Fine tuning for the 'Splitting' can be done, by transforming to input formula. Replacing disjunctions with conjunctions and vice versa is possible by applying the laws of Boolean algebra. Multiple considerations are needed to profit from the transformation of the input. The complexity can increase if the transformation is executed in general. As such this option to transform has not been included in this thesis.

The last variant 'Generalize1' *reduces the number of constraints* regarded to build the final generalization strongly. All three implemented optimizations are also activated together for the 'AllOptim' variant. Furthermore, intelligent decisions for activating the options are possible because of the extensive statistics implemented for logging. All relevant statistics will be covered in detail. Discussing the algorithm's performance requires insights into the statistics.

5.2 Statistics of the Execution

Discussing the algorithm's performance requires an understanding of the statistics. Thus, all *relevant statistics*, summarized in Table 5.2, are explained in detail. All statistics derived can be classified into different categories. The first category is the *structure of the input formulae*. Another category is the timers for the overall runtime and relevant execution steps. Lastly, values for the *core functionality*, GENERALIZE2, and *projections* are logged.

The following factors of the *input and output structure* are collected. Information on the number of quantifier alternations, quantified variables, variables in the formula, and the number of constraints of the input is collected. Occurrences of the same constraint are counted multiple times. This keeps the analysis simple while offering implicit information on the formula length. The category includes the structure of the output formula. To this extent, the number of constraints in the output is logged, with the same method as for the input.

The second category is the *timing of the different modules* used in the algorithm. For this statistic, the following modules are important enough to warrant a separate timer: the general algorithm, the SMT-solver, the projection, and the GENERALIZE2

call. The separate timer for these modules allows a more direct look at the runtime. The timers support identifying causes for timeouts and checking up on the effectiveness of all variants.

Furthermore, the *projection timer* is required to thoroughly compare the performance of both projections, FM and FMplex. While the time needed for the calculation is important, the size of the calculated projection might be a stronger performance indicator. Moreover, the structure has a crucial influence on the SMT-solver's performance. However, logging the interactions between the SMT-solver and the formula structure is too complex. Consequently, the timing of the overall runtime allows crucial insights into the projections' efficiency. One target is to identify if the performance after an efficient projection is lacking due to the algorithm's design decisions.

As a third category, further *criteria to compare all variants* are derived. These are the iterations of GENERALIZE2 and GENERALIZE1 calls needed. Both are tracked separately, since the variant 'RRShuffle' executes multiple GENERALIZE2 calls for each model and GENERALIZE1 call. Moreover, the number of constraints building the calculated generalizations, M_1 and M_2 , is logged. Another logged criterion is the number of identified subformulae, to track the effectiveness of the variant 'Splitting'.

Through the collection of statistics and their detailed inspection, knowledgeable explanations about the variants' runtime behavior are facilitated. This increases the potential for the algorithm's adoption in the future because the substantiated reasoning allows the utilization of the variants.

5.3 Validation of the Behavior

The correctness of the implementation is checked by *comparing* the results against an established *quantifier elimination algorithm*. While *Cylindrical Algebraic Covering* (CalC) [11] specializes in *nonlinear real arithmetic*, the algorithm is still well-suited for comparison. One added specialized module to SMT-RAT, executes the comparison automatically. If this module is chosen for the execution, the quantifier elimination for an input will be executed twice. Once by the tested variant and additionally by the 'CalC'. Because both the tested variant and the 'CalC' calculate an equivalent quantifier-free formula to the input. The algorithm's behavior is validated, by checking the output results for equivalency.

In this regard the formulae F'_{CalC} and $F'_{variant}$ are combined by an $\oplus$ (xor) to $F_{Check} := F'_{CalC} \oplus F'_{variant}$. If the formulae are the equivalent, the condition $Sat(F_{Check}) = UNSAT$ holds in general. Since equivalent formulae must have the same models. Thus, no model m satisfiable for one but not the other formula is found.

Table 5.1 The table gives an overview over all variants and their parameters. Moreover the expected improvements to the behavior are explained.

Standard	
Active Adjustments	• None
Parameters	• None
Expected Improvements	• Standard behavior
RRShuffle	
Active Adjustments	• Randomization of the Constraints Ordering (Shuffle) • Repetition of GENERALIZE2 and Projection • Reordering by Rotation between Repetitions
Parameters	• Reordering after GENERALIZE1: Shuffle • Number of Executions: 2 • Reordering before Repetition: Rotation
Expected Improvements	• Less dependency on good standard ordering • Protection against the worst case behavior • Increase chances to choose at least one good generalization
Splitting	
Active Adjustments	• Distribution of the Quantification • Quantifier Elimination for the Quantified Subformulae
Parameters	• (Not in Use: Transform of Formula (e.g. Disjunction to Conjunction))
Expected Improvements	• Simplification of the calculation • Smaller input formulae • Less input variables and constraints • Increased number of executions
Generalize1	
Active Adjustments	• Focus on Relevant Constraints • Direct Removal of Redundant Constraint (in GENERALIZE1-ONLYSAT)
Parameters	• None
Expected Improvements	• Targets the bottleneck GENERALIZE2 • Less constraints processed by GENERALIZE2 • Direct improvement to the runtime
AllOptim	
Active Adjustments	• Activates all three Single Change Variants
Parameters	• No additional Parameters
Expected Improvements	• Benefits from multiple variants • Interactions between variants are not optimized
AllOptimInt	
Active Adjustments	• Dynamic Activation of the Variants
Parameters	• Input Structure e.g. Number of Eliminated Variables
Expected Improvements	• Input based decision to active variants • Improved interactions between variants

Table 5.2 Relevant collected statistics are summarized in the table.

Category 1 – Input/Output structure
• # Constraints of Input • # Input Variables • # Eliminated Variables • # Quantifier Alternations • # Constraints of Output
Category 2 – Timers
(The timer also collects how often it is called)
• Timer – Overall • Timer – GENERALIZE2 • Timer – Projection • Timer – SMT-solver
Category 3 – Statistics of Core Steps
• # (Sub)formulae for Quantifier Elimination • # Constraints in Generalization M_1 (After GENERALIZE1) • # Constraints in Generalization M_2 (After GENERALIZE2) • # Constraints for all Projections • # GENERALIZE1 Executions • # GENERALIZE2 Executions

Chapter 6

Benchmarks on the Implementation

The tests are motivated by the need to gain *insights* into the *algorithm's behavior*. The focus is *collecting the information* needed to discuss the performance of all variants. To evaluate the effectiveness one benchmark is executed for each single variant. Moreover, the performance of the 'Standard' algorithm with no changes and 'AllOptim' with all 3 variants activated is interesting. Comparing the performance between these benchmarks allows conclusions about the interactions between the variants.

Each of the variants is expected to have different strengths. Because the adjustments focus on different bottlenecks, such as dependencies on the number or the order of constraints. For example, randomization and repetition in 'RRShuffle' improve the runtime behavior if the standard ordering is bad. Otherwise, if the original ordering for the constraints is ideal an increase in the running time is expected. In this regard, one interesting analysis is to identify when an adjustment to the algorithm is necessary. This allows us to find an optimal set of options for the execution. Interactions between variants can also be regarded for the final setting.

6.1 Setting for the Benchmarks

When executing the benchmarks the entire test *benchmark data* provided by SMT-LIB [10] was used. Then the algorithm's performance is checked against the 2439 formulae from 5 subsets. The formulae for these test sets differ significantly in their complexity. In the original publication, the '2010-Monniaux-QE' test set (Mjollnir) was introduced along with the algorithm and the toolkit 'Mjollnir' [18]. The '2010-Monniaux-QE' test set is still relevant [18] and extended to formulae *without quantifier alternations*. The 1800 formulae of the test set have all quantification except for the one at the *inner-most nesting* removed. These added benchmark formulae support the analysis of the performance influence of quantifier alternations.

The algorithm achieved the top performance for the benchmark in the first publication [1] in 2008. Achieving these strong results is based on the algorithm's strategy. In contrast to the original publication, achieving a top performance has not been the focus. For example, the execution of GENERALIZE2 will always call the SMT-solver new. If an incremental approach with minimal changes to the solver back end is im-

plemented the performance could be improved. Difficulties with unexplained errors such as 'Segmentation Fault' are the leading reason for deciding against these efficient strategies.

The difference between both implementations is their purpose. The presented *variants* focus on enabling the analytics of performance influences. Consequently, the comprehensive statistics measurements identify the conditions for performance improvements. Moreover, *comparing* the effects of *two different projections*, the FM and FMplex, will be a central point of the discussion. To give information on how the performance relates to existing quantifier elimination methods, the *Cylindrical Algebraic Covering* (CalC) [11] is chosen as a comparison. Since 'CalC' is the standard choice in SMT-RAT for quantifier elimination tasks. While 'CalC' does not specialize in formulae in LRA, the performance in tests is strong. Both considerations make the 'CalC' practical and suitable for classifying the performance of the implemented variants.

Before deciding on the final setting for the benchmarks, the parameter choices for all variants must be set. To this extent, tests have been carried out to find e.g. the number of repetitions necessary. These tests executed on the subset of 2022 formulae, the '2010-Monniaux-QE' and 'keymaera' benchmarks, are sufficient to conclude the effectiveness of the parameters. Different parameter values were compared against each other, to check if the chosen one is suitable for the analysis. The final setting for all variants and the targeted improvements are given in Table 5.1. The setting chosen to define a variant is employed once for all tests.

6.2 Execution of the Benchmarks

The benchmarks are executed for all variants presented in Table 5.1. If no specification is given, all benchmarks are executed with *FMplex as projection*. Consequently, the FMplex is active for tests with 'Standard', 'RRShuffle', 'Splitting', 'Generalize1', 'Alloptim', and 'AlloptimInt'. The main target of the benchmarks is the identification of performance criteria. Simulations were performed with *computing resources* granted by *RWTH Aachen University* under project 'thes1858'. The setting for all benchmarks is a timeout of 60 seconds and 2 GB memory. For the benchmarks with the variants, the *satisfiability of the output* is determined. To measure the profits from the algorithm's strategy to calculate output in DNF or CNF.

One of the questions is if the projection influences the algorithm's runtime. In benchmarks for quantifier elimination on conjunctions in LRA, the FMplex has been better. [3] In this setting, the projected conjunctions are less complex. Moreover, the complex interactions between the model choice and the projection are difficult to analyze. As such a simpler question will be answered. We first reason about the *relationship between the projection size and the runtime*. Then we compare the projections FM and FMplex concerning the size of their results.

To look at the effect of the described adjustments benchmarks with a *single active change* have been made. Another interesting point is to study the performance of different versions. The best runtime is expected to be achieved through the *combined activation of variants*. However, the runtime can differ significantly based on the interactions between multiple variants. The 'Splitting' and 'RRShuffle' variant execute additional calculations. An unsuitable input structure can be the cause of problematic interactions between variants. For example, the repetitions might be unnecessary

if the input is too simple. Then the need to execute the generalization and projection multiple times negatively influences the runtime.

To measure such *problematic interactions*, the behavior of 'Generalize1' is compared to 'AllOptim'. Analyzing the input for distinct criteria that result in a low performance enables a more *intelligent activation* of variants. If 'AllOptim' does perform lower than 'Generalize1', the added 'Splitting' and 'RRShuffle' will be deactivated. The resulting variant 'AllOptimInt' is expected to be faster. However, for complex input, not many improvements are possible. *Dynamically activating* the variants regarding *cutoffs for input parameters* is then implemented into 'AllOptimInt'. Considering the input structure for setting the variant, avoids problematic interactions between the changes. Disabling adjustments based on the input allows us to profit from all variants.

By offering an in-depth comparison of different options, the decision on an optimal version can be tailored to the test cases.

Chapter 7

Analysis of the Results

7.1 Summary of the Performance

7.1.1 General Performance Overview

First, a *general overview* of the performance of all *variants with FMplex* as projection is given. The overall time needed for the benchmarks is compared. As well as the number of solved instances within 60 seconds and 2 GB memory. In a direct comparison, the presented algorithm showed significant limitations. For all measured criteria, the runtime, memory needed, and the number of solved instances. On the benchmark data ‘CalC’ achieved the best performance for both categories, the fastest completion time, and most instances solved.

Table 7.1 The performance on the benchmark data of 2439 formulae is summarized for all variants and the ‘CalC’. The best performance is achieved by the ‘CalC’, followed by the variant ‘AllOptim’.

Algorithm or Variant	# Solved Instances	Timeouts	Memouts	Runtime (Solved) (m)	No Result (Segfault)
Standard	1713	723	0	109.17	3
RRShuffle	1765	673	0	155.36	1
Splitting	1773	666	0	91.28	0
Generalize1	1836	603	0	111.43	0
AllOptim	1908	531	0	117.58	0
CalC [11]	2294	143	2	20.91	0

The performance for all variants and the ‘CalC’ is summarized in Table 7.1. The final runtime is calculated by building the sum over the runtime for all solved formulae. The difference between ‘CalC’ and all variants is strong enough to conclude that the cause is not inefficiencies in the program code.

More informative is the *accumulated number of solved instances* over time. Based on this criterion Figure 7.1 depicts the more fine-grained comparison. All variants and the ‘CalC’ are compared. ‘CalC’ performs best and solves most instances in under 1 second. This is visible by the steep curve in the Figure 7.1. Regardless, the expected improvements in the performance due to adjustments to the algorithm are also clearly

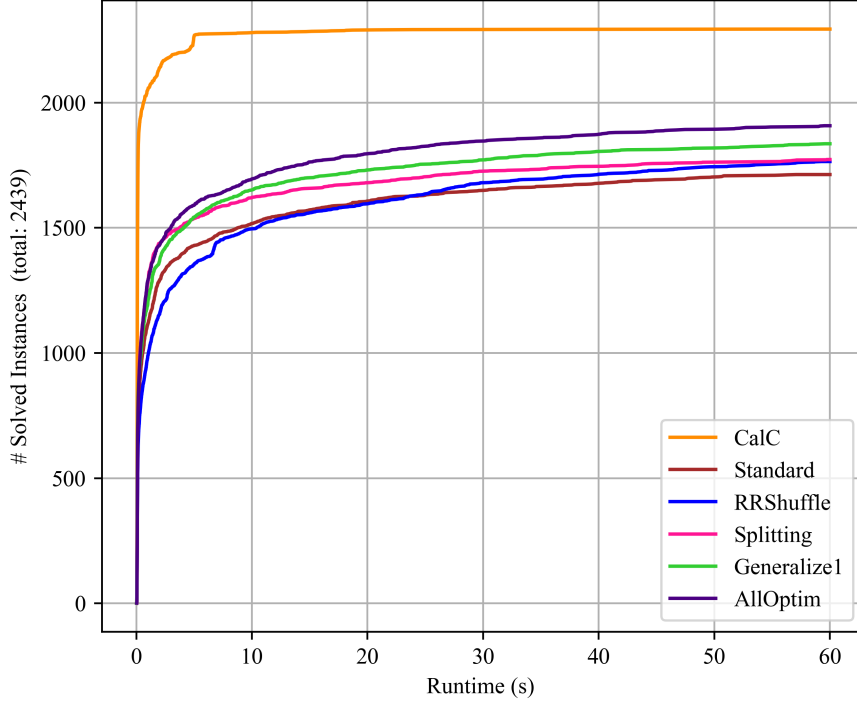


Figure 7.1: The performance of all variants is compared to the 'CalC' algorithm. For the comparison the number of solved instances over the runtime is depicted.

visible. The executions with 'Standard' and 'RRShuffle' have the lowest performance. However, the 'RRShuffle' variant performs on par with 'Splitting' for instances with a longer runtime. The variant with 'Splitting' follows these two. With a performance curve above these, the fastest variant with a single change 'Generalize1' is significantly better. Activating all variants leads to the version 'AllOptim'. Through the combined activation, the algorithm's performance improves significantly.

7.1.2 Benchmarks without Quantifier Alternations

The tests on input formulae *without quantifier alternations* are necessary for a comprehensive analysis. Then an analysis of the alternations' effect is possible. Moreover, the new formulae have multiple free variables. Executing additional tests with these formulae enables valuable insights into potential niches for the algorithm.

These 1800 formulae are the '2010-Monniaux-QE' subset of the benchmark set, with all quantifiers except for the innermost block removed. The performance is compared in Figure 7.2, with the variant 'AllOptim' achieving 376 more solved instances than 'CalC'. Using 'AllOptim' leads to 1405 solved instances and replacing the projection with FM to 1397. The weakest performance is by the 'CalC' with 1029 solved instances.

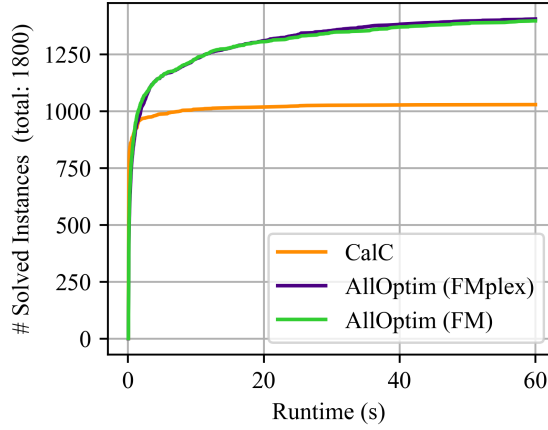


Figure 7.2: The number of solved instances is depicted on the data-set without quantifier alternations. The 'CalC', 'AllOptim' (FM) and 'AllOptim' (FMplex) are compared.

The runtime difference is examined through explanations using the derived statistics. For this, we apply the knowledge that the variant 'AllOptim' is strong for simple projections. This is a result of the complexity of both FM and FMplex. If fewer variables are projected, fewer constraints need to be considered for the projection. Thus, significant performance improvements are the result.

Table 7.2 The variant 'AllOptim', is compared to the 'CalC'. With the added information on calculating the result before the timeout.

# Eliminated Variables	AllOptim (Solved) & CalC (Timeout)	CalC (Solved) & AllOptim (Timeout)	Both Solved	Both Timeout
0, 1, 2	280	7	285	106
3, 4	170	13	326	77
5, 6	33	87	311	105

Table 7.2 compares the performance of 'CalC' and 'AllOptim'. For the comparison, the information on the number of eliminated variables is given. For the categories below with 0 to 4 eliminated variables, 'AllOptim' solves 430 more instances than the 'CalC'. Therefore, 'AllOptim' performs significantly stronger than 'CalC' in this niche. However, if 5 or 6 variables are eliminated the advantage is for 'CalC', with an additional 54 solved instances.

The time spent by the algorithm at important steps is analyzed. The responsibility for large portions of the runtime is on a few specific modules.

7.1.3 Runtime Distribution on the Modules

The *runtime distribution* is necessary to support conclusions about the cause of performance differences. For the analysis, *modules with a significant runtime* are identified.

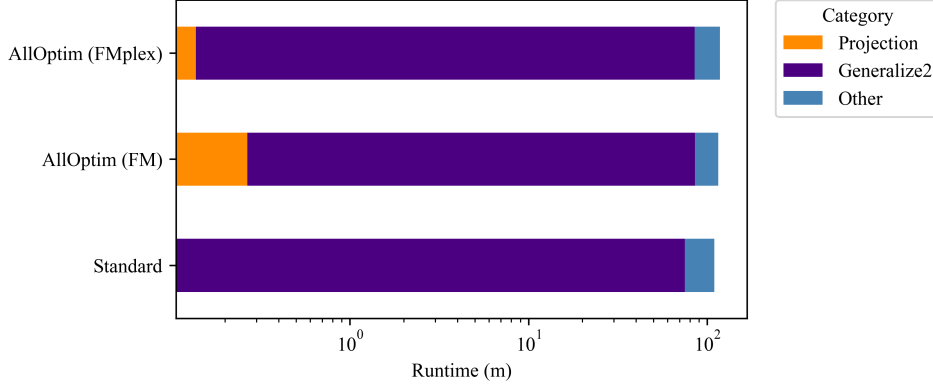


Figure 7.3: The distribution of the algorithm’s runtime for solved instances is depicted on a logarithmic scale. With respect to the three variants ‘Standard’, ‘AllOptim (FMplex)’ and ‘AllOptim (FM)’.

The overall runtime, GENERALIZE2, and projection are compared in Figure 7.3 for solved instances. The runtime necessary for projecting the generalization is under 30 seconds. That accounts for a small fraction of the overall runtime. Since *simplifying the generalization* takes the most time, the GENERALIZE2 behavior is the focus of the in-depth analysis.

Another motivation for implementing the algorithm was to facilitate the reasoning about the algorithm’s behavior regarding the input. For this reasoning, multiple dimensions of behavior are regarded. First, both projections will be compared in depth, regarding the benefits they offer based on the input. Afterward, the implemented versions are contrasted with a focus on the improvements in behavior they offer. Lastly, the option to activate the variants dynamically is explained.

7.2 Comparison between Projections FM and FMplex

The next focus is the *comparison of both projections*. Table 7.3 compares the number of solved instances for both FM and FMplex. When the execution uses FMplex, 1908 instances are solved, with 531 timeouts. Replacing FMplex with FM leads to solving 10 fewer formulae. Consequently, the results for the version with FM are 1898 solved and 541 timed-out instances.

The influence of both projections is depicted for the ‘AllOptim’ variant in Figure 7.4. By offering a plot that shows the accumulated number of solved instances against the runtime, new conclusions on the projection’s efficiency can be made. The performance is similar for fast executions under 10 seconds. However, if the runtime is between 20-35 seconds FMplex is especially good. To explain this behavior, the overall tendency in the runtime is looked at. Multiple instances are either solved fast

Table 7.3 Comparison of the performance for the variant 'AllOptim', with FM and FMplex on both benchmark datasets. The standard data has 2439 formulae and the with no alternations 1800.

Projection & Data	# Solved Instances	Timeouts	Memouts	Runtime (Solved) (m)	No Result (Segfault)
FMplex & Standard Data	1908	531	0	117.58	0
FM & Standard Data	1898	541	0	114.90	0
FMplex & No Alternations	1405	395	0	95.56	0
FM & No Alternations	1397	403	0	93.38	0

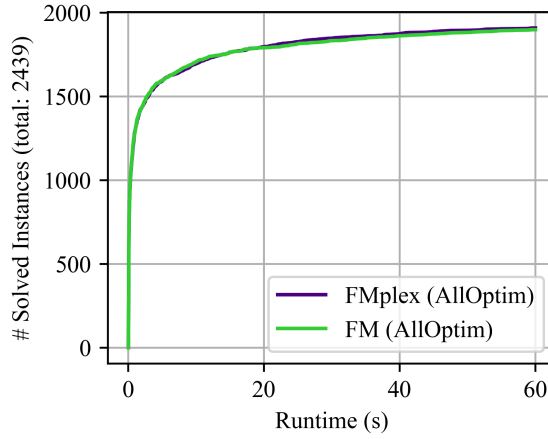


Figure 7.4: Comparison between the performance using FM and FMplex. (Variant: AllOptim)

or not at all. Consequently, FM can catch up to FMplex if the execution time is long enough. Since complex input will not be solved, the FMplex performance does stagnate.

The FMplex needs suitable input to showcase a competitive performance and profit from the novel approach. The projected formulae need to be solvable but complex. Moreover, the advantage of using FMplex increases when the formulae have many constraints. Both projections differ in the strategy for *eliminating multiple quantified variables* at once. The novel approach allows the FMplex to achieve a fast runtime on complex input. However, the additional calculations need more running time if FM is sufficient. Which projection should be preferred is analyzed concerning the input.

The quality of FM and FMplex is contrasted, based on the size of all projections needed. The information summarized in Tables 7.4 and 7.5 compares FM and FMplex. For formulae with 1 or 2 eliminated variables, the methods perform equally. The effect,

Table 7.4 The table compares the number of constraints for projections with the added information on the number of eliminated variables. For instances of the category 'FMplex (Solved) & FM (Timeout)'.

# Eliminated	# Instances	# Constraints of all Projections (Median)	
Variables: x		FMplex	FM
$0 \leq x \leq 2$	2.00	17318.00	10176.50
$3 \leq x \leq 4$	13.00	360.00	608.00
$5 \leq x \leq 6$	6.00	58.00	150.50
16.0	1.00	224.0	0.00

Table 7.5 Relevant information to compare FM and FMplex with regard to the eliminated variables and the size of all projections is given. For instances of the category 'FMplex (Timeout) & FM (Solved)'.

# Eliminated	# Instances	# Constraints of all Projections (Median)	
Variables: x		FMplex	FM
$0 \leq x \leq 2$	2.00	1929.50	1938.00
$3 \leq x \leq 4$	6.00	260.00	156.00
$5 \leq x \leq 6$	4.00	144.00	108.00
16.0	0.00	0.00	0.00

that a smaller projection size leads to a faster termination is visible in Table 7.5. The analysis focuses on Table 7.4 because FMplex solves more instances than FM. For 19 instances, with 3 to 6 eliminated variables the FMplex solves the formulae, but FM has a timeout. In these cases, executing with FMplex as projection results in a *significantly smaller projection size*. The difference is stronger if more variables are eliminated. Analyzing the difference between the projection sizes leads to the conclusion that FMplex offers strong benefits.

After covering the influence of the projections on the performance, the behavior regarding GENERALIZE2 is analyzed. To this extent, all variants are compared thoroughly. The focus is on the number of constraints included in the generalization.

7.3 Discussion of the Variants

When evaluating the behavior, the strong dependency on the *ordering of the constraints* needs to be regarded. The 'Standard' version collects all constraints occurring in the input by executing a standard function. Because this function is native to SMT-RAT further modules e.g. the SAT-solver have the same strategy to read out the formula's constraints. As a result, the SMT-solver and GENERALIZE2 are expected to use a similar ordering for their decisions.

To analyze the influence of the constraints ordering, the mechanism of both generalization steps is repeated. First, in GENERALIZE1, a *conjunction* M_1 is built from all constraints of the formula F . But the conjunction M_1 does include *redundant constraints*. Through the additional conjuncts, the size of the generalization is smaller than needed. In this regard, the second step GENERALIZE2 is executed, to build the

final generalization M_2 . All unnecessary constraints are *removed based on a certain order*.

The focus is the comparison of the variants with the help of a scatter plot. The plot depicts all instances and allows a representative expectation of the algorithm's behavior. First the effectiveness of the adjustments 'RRShuffle' and 'Generalize1' is discussed. Because the variant 'Splitting' strongly depends on the form of the input formula, it will not be analyzed separately. Lastly, the performance of 'AllOptim' is discussed.

7.3.1 Comparison between RRShuffle and Standard

Comparing the versions 'RRShuffle' and 'Standard' allows you to reason about the effect of the ordering. While 'RRShuffle' does have an additional execution, the constraints ordering is *rotated for the repetition*. This allows us to have stable and good behavior with random ordering. As seen in the overview Figure 7.1 the variant 'RRShuffle' performs significantly better than 'Standard'. With repetition, rotation, and shuffle active, 52 more instances were solved within 60 seconds.

However, the difference between both versions' behavior is more interesting. The 'Standard' behavior solves 72 instances, that timeout for 'RRShuffle'. In contrast, 122 formulae are solved by 'RRShuffle' but not 'Standard'. Together the results *differ for 194 input formulae*, solved by one but not the other version.

These results help with planning for a *future application* of the algorithm. One relevant question is whether the optimal ordering is identifiable based on the input. Then a dynamic switch between both versions could be implemented to adjust the algorithm's final behavior directly to the input.

7.3.2 Comparison of of Generalize1 and Standard

Another interesting comparison is between the versions 'Generalize1' and 'Standard' since both *use the same ordering*. By inspecting Figure 7.5, we compare both variants. 'Generalize1' solves most instances of 'Standard'. For formulae with a fast runtime, the offset in the behavior is stronger. Such a strong similarity in the runtime behavior is not seen for other comparisons. This leads to the conclusion that the performance depends strongly on the ordering's quality for 'Generalize1' and 'Standard'. If the ordering is different such a distribution does not occur. The versions 'AllOptim' or 'RRShuffle' with randomization of the ordering have instances with significant runtime differences to 'Standard'. However, the behavior of 'Generalize1' is faster than that of 'Standard' but otherwise similar.

The distribution of the instances is skewed towards the bottom, below the neutral line given in Figure 7.5. Therefore, the activation of 'Generalize1' improves the runtime in general. Further, the influence of the *number of input constraints* on the runtime is visible. The algorithm does not scale well with the number of input constraints.

The effect of the changes on the number of constraints processed by GENERALIZE2 is analyzed. The *lowered complexity* for GENERALIZE2 results in the faster 'Generalize1' runtime. For the comparison, the behavior of both algorithms is categorized. Either one of the variants has a timeout and the other variant does not, or both have the same result. These categories are used to mark the instances in the overview. In Figure 7.6 the size of the *generalization* M_2 is compared. The versions have similar be-

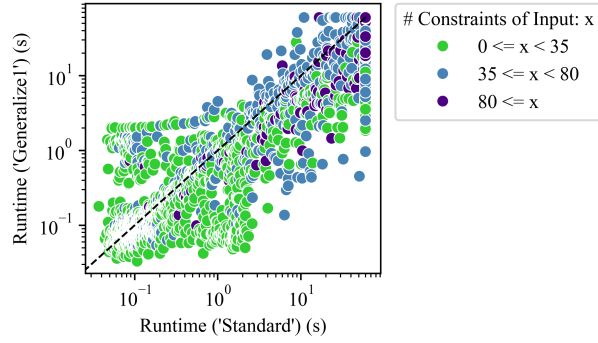


Figure 7.5: The runtime of the variant 'Standard' is contrasted with the 'Generalize1' runtime. Additionally, the connection between the runtime and the number of input constraints is visualized.

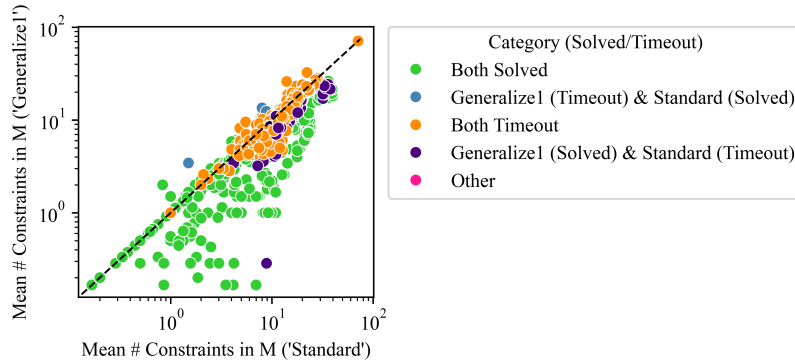


Figure 7.6: The average generalization size (after Generalize2) is shown for 'Generalize1' and 'Standard'. The distribution shows a bias towards a smaller generalization size for 'Generalize1'.

haviors, with 'Generalize1' achieving smaller generalization sizes. In a few instances, the 'Standard' execution will achieve better behavior. Since 'Generalize1' excludes a subset of constraints from the potential generalizations. If these constraints are involved in the optimal generalizations, the standard behavior will terminate faster than 'Generalize1'.

The variants with *static ordering* perform exceptionally well for a small subset of formulas. Such instances with ideal ordering can be identified by checking if the repetitions needed are exceptionally low. The problem is that there is no comprehensive strategy to improve the order of the constraints. One relevant takeaway from the analysis is the *strong influence of the ordering* on the performance. Another the need for a small generalization size, to achieve a fast runtime.

7.3.3 Comparison of AllOptim and Standard

Since 'AllOptim' is the best-performing variant, its benefits will be analyzed thoroughly. In this regard, the difference to the 'Standard' version is contrasted.

First, the runtime comparison is given in Figure 7.7, with information on the number of input constraints as an added dimension. Because the variants do not use the same ordering, their runtime *differs strongly* even for *simple instances*. If the behavior is analyzed with a focus on the input with 35-80 constraints, the distribution of the blue points is skewed toward the bottom. Consequently, 'AllOptim' performs significantly better than 'Standard' for these instances. On the other hand, the use of 'AllOptim' leads to timeouts even if the runtime for 'Standard' is under 1 second.

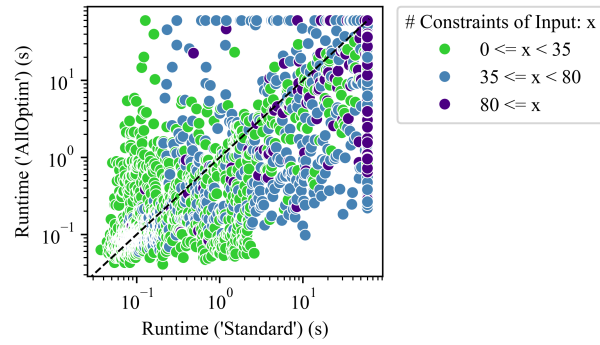


Figure 7.7: The runtime of 'AllOptim' is compared to 'Standard'. Further information on the number of input constraints is depicted.

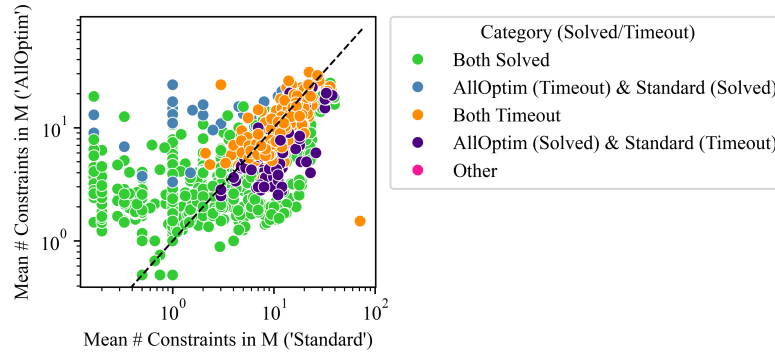


Figure 7.8: The number of constraints in the generalization, after Generalize2, is compared for the variants 'AllOptim' and 'Standard'.

The quality of a variant is evaluated by looking at the average size of the final generalization. Moreover, instances with different results for both versions are most interesting. Figure 7.8 compares both versions' *average generalization size*. Addi-

tionally, the information on a timeout is added for an evaluation. The conclusion is based on the instances in blue. These are the formulae solved by one variant but not the other one. Most of these instances have a bias towards the bottom or the top. The version with a *smaller generalization terminates* and the other has a timeout.

The correlation between a *smaller generalization* size and a *faster runtime* is visible. Evaluating the general difference between both versions is done by analyzing the distributions. Most instances have a strong offset towards the bottom, showing the benefits of using 'AllOptim'. Consequently, the focus on adjusting the mechanism for building generalizations has been successful. The complexity of both the generalization and projection steps is decreased successfully.

7.4 Dynamic Activation of Variants

The performance has improved, through the *input-based activation of the variants*. The change leads to a performance above any version with a fixed strategy, given in Table 7.6.

Table 7.6 The performance for relevant variants is compared to 'AllOptimInt' on the benchmark data of 2439 formulae.

Algorithm or Variant	# Solved Instances	Timeouts	Memouts	Runtime (Solved) (m)	No Result (Segfault)
Standard	1713	723	0	109.17	3
Generalize1	1836	603	0	111.43	0
AllOptim	1908	531	0	117.58	0
AllOptimInt	1916	523	0	99.01	0

This is also depicted in the Figure 7.9, with 'AllOptimInt' having the most solved instances.

Especially interesting is that formulae with a short runtime show the most significant improvement compared to the version with a flat activation. By deactivating unnecessary optimizations, the performance on simple input is improved. To find an intelligent decision for activating the variants, the strengths of both versions 'Generalize1' and 'AllOptim' are compared. Interestingly, the results include instances, with 'AllOptim' performing worse than 'Generalize1'. Moreover, distinct input parameters can be identified as causes. When looking at the number of eliminated quantifiers and variables.

To fine-tune the activation of the variants for 'AllOptimInt', we consider the information on *quantifier alternations* and *eliminated variables*. From Figure 7.10 the cutoff to deactivate both 'Splitting' and 'RR+Shuffle' is derived. For this, the instances categorized in blue are looked at. For less than 12 eliminated variables, the number of blue results is significantly lower than above. Another cutoff is determined by looking at Figure 7.11. Blue instances occur dominantly if the input has between 1 and 3 alternations.

Consequently, for an *input-based optimization* both 'Splitting' and 'RR+Shuffle' should be *deactivated* if both conditions do not hold. Such a version with an intelligent activation of the variants is implemented in 'AllOptimInt'. With a deactivation for the condition: '# Eliminated Variables < 12' and '# Quantifier Alternations > 3'.

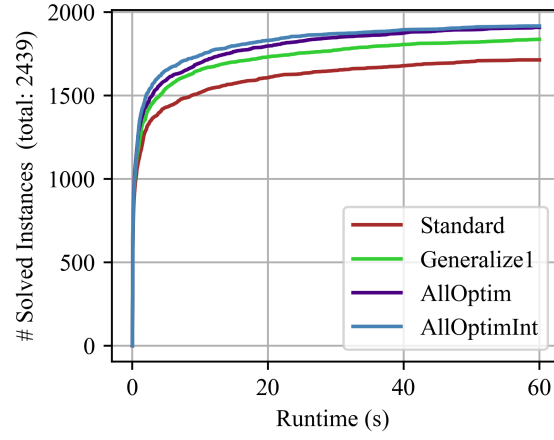


Figure 7.9: The performance of 'AllOptimInt' is compared to 'Standard', 'Generalize1' and 'AllOptim'.

While the fine-tuning is successful, the algorithm's overall behavior does not change. In the benchmarks, the number of solved instances has improved. But not to the point of a performance, competitive to the 'CalC'. Still, the fine-tuning emphasizes, that a dynamic adjustment of the algorithm has benefits. The strong focus on enabling flexible behavior, through introducing new variants has been a success.

For the last point, the entire thesis is summarized and we give a short discussion to the future use. Crucially, we present multiple ideas that are candidates for future improvements. These changes focus on enabling parallelism, suitable pre-processing and input-based decisions.

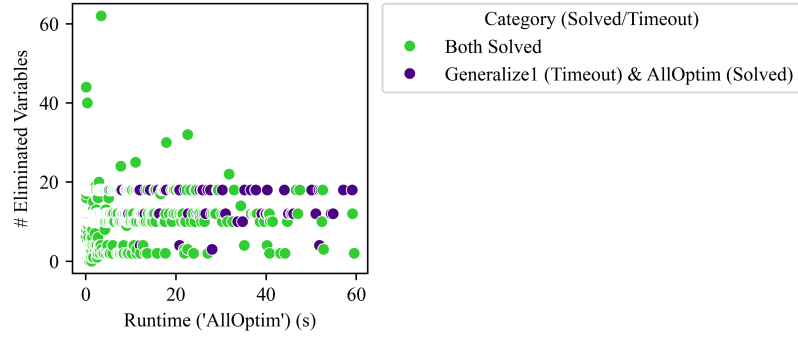


Figure 7.10: The variant 'AllOptim' is better for all instances in dark blue. To deactivate 'AllOptim' if needed, the distribution of these instances with regard to the number of eliminated variables is visualized.

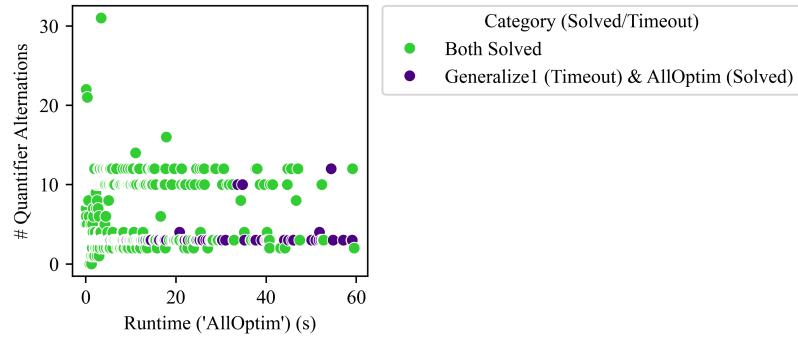


Figure 7.11: In dark blue instances with an advantage for 'AllOptim' are visualized. These occur mostly if the number of quantifier alternations is less or equal 3.

Chapter 8

Conclusion

We give a comprehensive *summary* of the algorithm, all variants, and the executed benchmarks. Moreover, we present ideas for *future use*. The proposals for further optimizations to the algorithm are based on tests that were executed but not finalized. Further, the conclusion will also present *ideas for the future based* on the data collected during the writing process.

8.1 Summary

The thesis gives a thorough analysis of the *quantifier elimination algorithm* [1]. After explaining the theoretical foundations necessary, the algorithm is detailed. In this regard, a focus is put on the behavior of the core mechanism, the generalization, and the projection.

We have identified performance bottlenecks, through tests and an analysis of the underlying theory. *Variants of the algorithm* are presented, to increase efficiency. The optimizations target different problematic behaviors, which are detailed in examples.

The first variant ‘RRShuffle’ decreases the dependency on good ordering for removing constraints in the GENERALIZE2 call. Additionally, the *repetition and rotation* lead to two overlapping projections. Through the change, the influence of one bad generalization is minimized.

For the second variant, ‘Splitting’, the decision was to *split up the input formula*. In this regard, the quantification is distributed into the formula. Then the algorithm must be applied to all subformulae, and these are smaller. Consequently, the variant has a trade-off between the number of executions and the complexity of each single execution.

The dependency on the number of input constraints motivates the general activation for the ‘Generalize1’ variant. The last variant ‘Generalize1’ improves the *generalization strategy*, through following the reasoning of the SMT-solver. While the active constraints are not directly identified, the generalization is built from a relevant subset of all constraints. Removing multiple redundant constraints before the execution of GENERALIZE2 is beneficial. The adjustment significantly reduces the generalization size.

After these explanations, we have discussed the *implementation of the variants* in SMT-RAT. The implementation handles the setting for the parameters and variant activation. Moreover, the variants chosen for the benchmarks are presented. These

include the variant ‘AllOptim’ with all adjustments active at once, and ‘AllOptimInt’ with an input-based activation.

8.2 Discussion of Future Use

Analyzing the statistics collected from the tests *identifies the criteria for a fast run time*. Significant performance improvements are achieved for smaller generalization sizes. The runtime between versions differs significantly, based on the number of constraints included in the generalization. The strong effect of removal order for the constraints, when building the final generalization has also been shown.

The comparison with the ‘CalC’ algorithm is analyzed, to discuss the algorithm’s future use in applications. The benchmarks identify one type of formulae with a performance comparable to the ‘CalC’. These are input formulae with multiple free variables and no quantifier alternations.

The discussion for the future use of ‘AllOptim’, must recognize these limitations. Moreover, the parameters to activate variants need to be fine-tuned to the input structure.

8.3 Future work

8.3.1 Sharing of Projections

The most relevant idea for *future research* would be to enable parallelism. While the splitting up of the formula is implemented, the independent parts are regarded iteratively not in parallel. This is the reason for one problem, the wait for the result of all split executions.

In an ideal *parallelized execution*, the termination could occur as soon as the redundancy of all other split executions is identified. One trivial case is e.g. splitting up a disjunction and having *true* as a result. Or *false* in case a conjunction was split up. The optimization is enabled through the sharing of the projections. For existentially quantified formulae, the *projections can be shared* between each execution on a subformula.

We substitute the split with the assumption that the algorithm will consider the subformulae separately. Then the standard behavior is used, to collect the set of all projections P_{s_1} based on s_1 . Since the formula is not split up, the entire set P_{s_1} is removed from consideration for the model choice. Thus, the behavior is equivalent to ignoring solutions found in the parallel case.

To explain the intuition on how this would function, an example will go over all necessary steps.

Example 8.3.1 (Sharing Projections). *For example, in case of a disjunction $\exists x(s_1) \vee \exists x(s_2)$ the result r_1 of subformula s_1 overlaps the entire subformula s_2 . Then instead of checking H_{s_2} to see if more projections are needed, the already collected projections are regarded. So the satisfiability of $H_{s_2} \wedge (\neg r_1)$ is checked.*

If more projections are necessary, the results from the executions before can be reused. The partial results have the form $O_{s_2} \vee \exists x(H_{s_2})$. Consequently, existing projections from r_1 can be added to O_{s_2} . So, the calculated equivalent formula can be shared with all other subformulae.

The change is also compatible without parallelized sub-routines. If no parallelization is the target, the result r_1 of the prior iteration can also be shared with later iterations.

A parameter to the strategy would be to reverse unneeded split-ups and to recombine the formulae strategically. This can be done, because $\exists.x(s_1) \vee \exists.x(s_2) \equiv \exists.x(s_1 \vee s_2)$. Analyzing the statistics to find an ideal formula complexity, using more intelligent splitting seems reasonable.

8.3.2 Pre-processing the Input

For future applications, a *stable and competitive execution time* is necessary. The number of input constraints is most relevant for the performance. All variants, including the best-performing one, struggle if the number of input constraints increases. However, the prenex-normal form (PNF) is applied before the computations. Because of PNF, the algorithm does not focus on quantified subformulae.

Identifying a *quantified subformula* as input supports fast and simple projections. A speedup is possible since all unquantified variables and constraints are no longer part of the input. The idea creates a possible overhead, the *search* for suitable subformulae. Again, parallel executions of the algorithm on quantified subformulae independent from each other are thinkable. Consequently, a suitable pre-processing would be a useful addition to the implementation.

We will explain the necessary pre-processing with the help of an example. For this, we search for quantified subformulae. If any are found, we will not search inside of these.

Example 8.3.2 (Pre-processing). *Given a formula with quantification $F_1 := c_1 \wedge c_2 \wedge c_3 \wedge (\exists.x(F_{s1}))$ the pre-processing extracts the formula $\exists.x(F_{s1})$.*

After calculating the equivalent quantifier-free formula F'_{s1} , the formula F_1 is updated by substituting the extracted formula.

Update: $F_1 := c_1 \wedge c_2 \wedge c_3 \wedge (\exists.x(F_{s1})) \equiv c_1 \wedge c_2 \wedge c_3 \wedge F'_{s1}$

The outermost nesting is regarded for input with multiple nesting levels. If the formula has the structure $F_2 := c_1 \wedge c_2 \wedge (\exists.x(c_3 \wedge (\exists.y(F_{s2}))))$ the formula $(\exists.x(c_3 \wedge (\exists.y(F_{s2}))))$ is extracted and then transformed to its PNF.

To improve the behavior even further intelligent pre-processing is thinkable. For example, continuing the search to find the innermost nested quantified formula.

8.3.3 Strategic Ordering of the Constraints

The last idea supports the *algorithm's application in a niche*. Because the input formula describes a system's behavior, the ordering of constraints can be decided by *prior information*.

The cones seen in multiple examples can be used to describe the behavior of hybrid systems. In the 'Standard' variant, the constraints' ordering depends on the formula structure. One idea is to check if the performance will improve and if the order-based decisions can be biased to pick these cones. Such a behavior can be achieved by strategically removing or adding the cones to the generalization. The benefit of such a bias would be stronger independence from the exact formula structure. However further tests are needed because the algorithm's behavior can be difficult to predict.

Already without changes, the benchmarks proved that the best variant, 'AllOptim', is well-suited for future applications. If these *fit the niche*, the input has no quantifier alternations, few quantified variables, and many free ones.

Because of the high flexibility offered, the addition of multiple specializations to tailor the algorithm's performance to a specific input is another idea. This leads to the conclusion that the algorithm defined by David Monniaux [1] can be relevant, especially in combination with *modern projections*.

Bibliography

- [1] D. Monniaux, “A quantifier elimination algorithm for linear real arithmetic,” in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2008, pp. 243–257.
- [2] Ths-rwth. (2024) SMT-RAT toolbox for strategic and parallel satisfiability-modulo-theories solving. [Online]. Available: <https://ths-rwth.github.io/smrtrat/>
- [3] V. Promies and E. Ábrahám, “A divide-and-conquer approach to variable elimination in linear real arithmetic,” in *International Symposium on Formal Methods*. Springer, 2024, pp. 131–148.
- [4] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, “Software model checking via large-block encoding,” in *2009 Formal Methods in Computer-Aided Design*. IEEE, 2009, pp. 25–32.
- [5] S. M. Kolb, S. Teso, A. Passerini, L. De Raedt *et al.*, “Learning SMT (LRA) constraints using SMT solvers,” in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*. IJCAI, 2018, pp. 2333–2340.
- [6] R. Fagerberg, C. Flamm, D. Merkle, and P. Peters, “Exploring chemistry using SMT,” in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2012, pp. 900–915.
- [7] B. Yordanov, C. M. Wintersteiger, Y. Hamadi, and H. Kugler, “SMT-based analysis of biological computation,” in *NASA Formal Methods: 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings 5*. Springer, 2013, pp. 78–92.
- [8] A. Tarski, “A decision method for elementary algebra and geometry,” in *Quantifier elimination and cylindrical algebraic decomposition*. Springer, 1998, pp. 24–84.
- [9] A. Seidenberg, “A new decision method for elementary algebra,” *Annals of Mathematics*, vol. 60, no. 2, pp. 365–374, 1954.
- [10] M. Preiner, H.-J. Schurr, C. Barrett, P. Fontaine, A. Niemetz, and C. Tinelli, “SMT-LIB release 2023 (non-incremental benchmarks),” Feb. 2024. [Online]. Available: <https://doi.org/10.5281/zenodo.10607722>

- [11] E. Ábrahám, J. H. Davenport, M. England, and G. Kremer, “Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings,” *Journal of Logical and Algebraic Methods in Programming*, vol. 119, p. 100633, 2021.
- [12] F. Corzilius, G. Kremer, S. Junges, S. Schupp, and E. Ábrahám, “SMT-RAT: an open source C++ toolbox for strategic and parallel smt solving,” in *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24–27, 2015, Proceedings 18*. Springer, 2015, pp. 360–368.
- [13] R. Sebastiani, “Lazy satisfiability modulo theories,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 3, no. 3-4, pp. 141–224, 2007.
- [14] E. Ábrahám and G. Kremer, “SMT solving for arithmetic theories: Theory and tool support,” in *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2017, pp. 1–8.
- [15] V. Promies, J. Nalbach, E. Ábrahám, and P. Kobialka, “FMplex: Exploring a bridge between fourier-motzkin and simplex,” *arXiv e-prints*, pp. arXiv–2309, 2023.
- [16] L. Khachiyan, *Fourier–Motzkin Elimination Method*. Boston, MA: Springer US, 2009, pp. 1074–1077. [Online]. Available: https://doi.org/10.1007/978-0-387-74759-0_187
- [17] L. de Moura and N. Bjørner, “Z3: an efficient smt solver,” in *2008 Tools and Algorithms for Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, March 2008, pp. 337–340. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/z3-an-efficient-smt-solver/>
- [18] D. Monniaux. (2025) Mjollnir. [Online]. Available: <https://www-verimag.imag.fr/Mjollnir.html>