Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

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

BACHELOR OF COMPUTER SCIENCE THESIS

AUTOMATED EXERCISE GENERATION FOR THREE SELECTED SATISFIABILITY CHECKING PROCEDURES

Maria Kazantzi

Examiners:
Prof. Dr. Erika Ábrahám

Additional Advisor:
Prof. Dr. Ulrik Schroeder

Aachen, March 21, 2022

Abstract

For this thesis, an application has been created in order to generate exercises for the students. These exercises aim to help the students practise with satisfiability checking. The methods that have been used for satisfiability checking are SAT, Fourier-Motzkin variable elimination and Virtual Substitution. In the following chapters, there will be further explanation regarding my criteria for creating a task and how the solutions should be like. Definitions of all the mentioned terms will be specified. Some related tools regarding SAT solvers have been found and are used for comparison with the algorithm used for this thesis. Furthermore, the exact implementation will be described by using UML diagrams of all the three methods and pointing out the most important functions that are implemented. The aim of this thesis is to mention the aspects of how to create good and pedagogical exercises for students. The application has been sent to some students that are currently taking part in the lecture "Satisfiability Checking" or are already familiar with satisfiability checking and were asked for a feedback. The results of the evaluation are mentioned at the end of the thesis in order to show what was achieved.

Acknowledgments

First of all, I would like to thank my family and friends who were always there for me and supported me throughout my whole bachelor. Special thanks to my supervisor Prof. Dr. Erika Abraham, who was always reachable and available to help me with all kind of questions and for introducing me to this bachelor thesis thema. Furthermore, I would like to thank Prof. Dr. Ulrik Schroeder who accepted to be the second reviewer of my thesis. Lastly, thank you to all my friends who helped with proof reading and to those who took part in the evaluation and made suggestions to improve this thesis.

Contents

Chapter 1

Introduction

In theoretical computer science and mathematical logic, the satisfiability problem is the problem to decide whether a given formula is satisfiable, i.e. whether there exists a model that satisfies the given formula. One of the most known open problems in computer science is, whether the satisfiability problem of propositional logic (SAT) can be solved in polynomial time. More specific, if the formulas in CNF 2.2.1 can be solved in polynomial time. P is the class of the problems that are solvable in a polynomial time. By polynomial time we mean that the number of the Turing machine's steps to solve a formula is bounded by a polynomial in the input length. Nondeterministic polynomial (NP) is the set of problems that the solution of the individual problems can be verified in a polynomial time. NP includes all the problems of P, since they can be solved in a polynomial time. This is because in the NP class are all the problems that can be checked in polynomial time. Since problems in P can be solved in polynomial time, checking is achieved by simply computing the result. SAT can be solved in exponetial time by brute forcing all possible variable assignments. This is possible for example, by creating a truth table. An assumption in computer science is that, no algorithm exists which solves SAT in a polynomial time and therefore, the problem lays only in the NP class and not in P.
The SAT problem is about a given propositional logic formula and the question, whether this formula evaluates to true or false, after assigning all the variables in the formula. SAT was the first problem whose NP-completeness has been proven [Tov84]. NP-complete is a class of problems. If the solution of one of the problems in the class can be quickly computed then all the problems in the set can be quickly solved.
The second method used for satisfiability checking is the Fourier-Motzkin variable elimination. This is a mathematical algorithm which eliminates variables one by one from a system of linear inequalities [HLL90]. By using this method and repeating the process, it is possible to obtain a system with at most one variable and we can see whether the whole system has a solution or not.
Virtual Substitution is another method to check for satisfiability. It is similar to the Fourier-Motzkin method but with quantifier elimination.
In the scope of this work, solvers for each will be introduced. The goal of this thesis is to create understandable and interesting exercises for students, either for examination or practise. By using the programming language C++, an application for the students has been created. The programming language C++ has been chosen, because it has higher performance and modules, such as z3 1.2 that can be used to verify the results.

To achieve a better interaction between the students and tasks, a GUI has been developed. Other possibilities were to create a pdf or website. The used library for creating such an interface was Qt, because i was able to create a user interface for both operating systems.

1.1 Motivation

This application helps the students to practise their knowledge in satisfiability checking while using these three methods. The students should use the whole algorithms mentioned in the lecture "Satisfiability Checking" [ÁK16] in order to be able to solve some of the tasks. For other tasks, just a partial question is asked. They have the possibility to choose how difficult a task should be. In this way, they can practise exercises from easy to difficult. Also, the solutions are shown to help the students understand all the important steps of the solutions. This is the most important aspect, as the students learn from the solutions.

1.2 Related Work

As far as related work is concerned, the z3 tool is used for analysis of systems [DMB08]. Z3 is also known as Theorem Prover, which is a solver created by Microsoft. This tool was developed in order to solve problems that appear in software verification and analysis. Z3 is a high-performance tool used for satisfiability modulo theories (SMT). The solver uses a simplifier at the beginning [DMB08]. The simplifier applies standard algebraic reduction rules and reduces as much as possible the formulas that were given as inputs. This for instance, does not happen to the solvers that were implemented for this thesis. All the input formulas are already merged and they can not be more simplified. One resemblance with the implemented SAT algorithm is that, z3 is also based on a DPLL SAT solver that is able to handle equalities [DMB08]. Additionally, it uses the method of two-watched literals for propagation as well. In case of conflict, z3 resolves it by performing backtracking. Lastly, it returns models for formulas that are satisfiable and proof for unsatisfiable formulas. One difference is that z3 handles also bit-vectors and arrays.
Another relevant tool is CVC4. This shortcut stands for Cooperating Validity Checker. It is based on the lazy DPLL algorithm [BCD+11]. By using the CVC4 tool, it is possible to run a number of different threads at the same time. Interruption of the operations is possible, if results from other threads make them irrelevant. It can be used to prove the satisfiability of first-order formulas. Both solvers can be used in the C++ API and support real numbers.
Another related solver is SMTInterpol [CHN12]. Also this tool is based on the DPLL algorithm, but the backtracking is not necessarily chonologically. In the algorithm for this thesis the backtracking is possible only chronological. This means, that only steps backwards according the order are possible. For satisfiable formulas, models are returned. Otherwise, it produces resolution proofs. SMTInterpol and CVC4 are based on the simplex method, in the domain of theory solver for linear arithmetic.
Until now, there is a simplex solver as a known tool that generates exercises and solves them by using the simplex method. It is a pivot tool that can be used to solve linear programming problems. Other than that, there are currently no available

tools that use SAT, Fourier-Motzkin and Virtual Substitution as methods for exercise generation and solving.

Chapter 2

Basics

2.1 Criteria for Exercise Generation

The exercises that are being generated must satisfy certain quality criteria. These criteria are according to the problems, the task formulations and solutions that are provided afterwards to the students, in order to be pedagogically useful.

2.1.1 Problem

The decision of the tasks should be whether they are relevant and meaningful for the students to know this step of the process or not. Tasks must be easy to understand and solvable. They should not be very complex, because the aim is to build up the confidence of the students about their knowledge. Another aspect is what exactly should be asked. Because there are whole procedures for solving such exercises, if only certain steps should be asked or the whole procedure. Moreover, if the questions depend on following errors or not. The type of the exercise should also be taken into consideration. Whether they are multiple choice, numerical answers or free text. This aspect controls if the students can actually guess the answer or not.

2.1.2 Task Formulation

The questions should be presented in a way, so that they have the same syntax as in the lecture of "Satisfiability Checking" [ÁK16]. This is in order to avoid any misunderstandings and to be as intuitive as possible for the students. They should always fit the level of knowledge of the students and demand what they have already learnt. It is important that the exercises are fair among each other and that they take the same amount of time from student to student, when the tasks are for examination. It is desired, that the representation of the result of the exercise is simple and the calculations are not that complex for the students. Lastly, the tasks must always be clear and specific enough, so that the students know exactly what they are asked. The same goes for how the students should type their answers. For example, how many digits after a komma are needed when they write down their answer.

2.1.3 Solution

Just like tasks, solutions must also use the same syntax and algorithms as in the "Satisfiability Checking" lecture [ÁK16]. An important aspect for the solutions is that, the generated solutions should have a fair amount of steps, which clarify how the outcome was calculated. The results of the program must always be correct and easy to understand, in order to help the students learn correctly. Also, the solutions can return a small feedback to the given answer with the aim to motivate the students. Presenting definitions can be helpful in the explanation of a solution. Last but not least, the answers should be undestandable and have all the possible ways of solutions.

To be fair, all students should receive individual exercises that need the same effort to be solved. Fairness is a major point in all of the three procedures that have been implemented. There are two ways to achieve it:

1. Isomorphism: Here the task of each student looks different but has exactly the same functionality. Function to achieve this is the **changeVariableName()**. This function replaces a variable with another one.

   Example:
   $(a \vee b) \wedge (b \vee c) \Rightarrow (a \vee d) \wedge (d \vee c)$
   In this case b is replaced by d.

   Next function is the **changePositions()**. By using this function, the position of the clauses can be swapped.

   Example:
   $(a \vee b) \wedge (b \vee c) \Rightarrow (b \vee c) \wedge (a \vee b)$

   Lastly, for the Fourier-Motzkin and Virtual Substitution there is also the function **reverseInequalities()** that swaps the terms and reverses the symbol of the inequality.

   Example:
   $3 + 2y < 5 \Rightarrow 5 > 3 + 2y$

2. Fairness based on method's characteristics. This aspect in SAT is based on the number of decision levels and backtracks. If this is not the case, some students will have to apply more algorithmic steps in order to reach the solution. In the Fourier-Motzkin variable elimination method, fairness depends on the number of variables and clauses. If a student has an exercise with 3 variables then another student must have a similar amount of variables as well. In similar way lays also the fairness for Virtual Substitution, as the fairness depends on the amount of variables and clauses of the exercise. If that is not the case, the effort for the specific exercise will not be the same, as the point of those type of exercises is to eliminate the variables.

To create a task in Fourier-Motzkin with simple representable results and not complex calculations, the function **createGoodExercise()** is called. Also, in exercises for virtual substitution the function **generateSquareTerm()** is called, in order to create

a term with integer as solution of the roots. Their functionality will be explained in the implementation Chapter 4.

2.2 Definitions

In this part, basic definitions as well as the used methods will be explained in order to help with the comprehension of the terms used in this paper.

2.2.1 SAT Algorithm

The input of this method is a propositional logic formula ϕ in conjunctive normal form (CNF). CNF is a conjunction of one or more clauses, where a clause is a disjunction of literals.

Example:
$(A \vee C) \wedge (B \vee C \vee D)$ is in CNF.
$(A \wedge B) \vee C$ is not in CNF.

For this thesis the DPLL algorithm was used [ÁK16]. In this algorithm the variables are getting assigned. An assignment of variable can either be with false or true. Before explaining the algorithm, we have to make some status of the clauses clear, that occure after assignment of variables.

Satisfied: At least one literal in the clause is true.
Unsatisfied: All literals in the clause are false.
Unit: All literals but one are assigned to false.
Unresolved: All the other cases.

The algorithm starts from the left of the formula and if it finds a unit clause it propagates. This means, it assigns the last unassigned variable of the unit clause in a way such that the clause is fulfilled. The step of propagation uses unit clauses which imply the consequences of decisions. Then it checks again in case of another unit clause. If it is not the case, it assigns the first variable that has priority, depending on the variable order that is given. The default value that is used, is false. Then again, it checks whether there is a unit clause. The algorithm decides for all the variables until a unit clause is found and propagates it so that the clause is fulfilled. If there is a conflict then it backtracks the last decision and reassigns the variable that was last assigned, to the opposite value. Conflict is when the current assignment does not satisfy the formula. If the conflict continues, the backtrack goes on until all the possibilities are checked. If all the clauses of the formula are satisfied, then the algorithm returns satisfiable. If backtrack and propagation are not possible anymore, the algorithm returns unsatisfiable for the current formula. In case of a conflict in decision level 0, the formula is unsatisfiable and does not have a solution.

Example of propagation:
Given: $\Phi := ((A \vee B) \wedge (B) \wedge (\neg B \vee C))$ with the static order $A < B < C$.

Assign B = true, because (B) is a unit clause:
$\Phi = ((A \vee 1) \wedge (1) \wedge (\neg 1 \vee C))$
The clause $(\neg 1 \vee C)$ is a unit.
Assign C = true:
$\Phi = ((A \vee 1) \wedge (1) \wedge (\neg 1 \vee 1))$
Assign A = false, because it is already satisfiable and it is the default value:
$\Phi = ((0 \vee 1) \wedge (1) \wedge (\neg 1 \vee 1))$
The resulting formula is $\Phi = (1 \wedge 1 \wedge 1) \equiv 1$
The algorithm returns satisfiable for this formula.
A solution is: $A = 0$, $B = 1$, $C = 1$.

First example of backtrack:
Given: $\Phi := ((B) \wedge (A) \wedge (\neg A \vee \neg B))$ with the static order $A < B < C$.

Assign A = true, because (A) is a unit clause:
$\Phi = ((B) \wedge (1) \wedge (0 \vee \neg B))$
Assign B = true:
$\Phi = ((1) \wedge (1) \wedge (0 \vee 0))$
The third clause is unsatisfied. This means we should backtrack the last decision and reassign B = false:
$\Phi = ((0) \wedge (1) \wedge (0 \vee 1))$
Now the first clause is unsatisfied. We backtrack again but because the value of B was already switched, we reassign the value of A to false:
$\Phi = ((B) \wedge (0) \wedge (1 \vee B))$
The second clause is unsatisfied.
Because those were all the possibilities, the formula does not have a solution and the algorithm returns unsatisfiable.

Second example of backtrack:
Given: $\Phi := ((\neg A \vee B) \wedge (B \vee C \vee D))$ with the static order $A < B < C < D$.

There is no unit clause from decision level 0, so A should be first assigned.
A is assigned to false, because of the default value.
Assign A = false:
$\Phi = ((1 \vee B) \wedge (B \vee C \vee D))$
Next priority has the variable B and it is assigned to default value:
$\Phi = ((1 \vee 0) \wedge (0 \vee C \vee D))$
Priority has now C and it is assigned to false due to default value:
$\Phi = ((1 \vee 0) \wedge (0 \vee 0 \vee D))$
Because the clause $(0 \vee 0 \vee D)$ is a unit, D is assigned to true:
$\Phi = ((1 \vee 0) \wedge (0 \vee 0 \vee 1))$
The resulting formula is $\Phi = (1 \wedge 1) \equiv 1$
The algorithm returns satisfiable for this formula.
A solution is: $A = 0$, $B = 0$, $C = 0$, $D = 1$.

With the procedure of finding and direct assigning the variable of a unit clause, the algorithm is able to find mandatory assignments and therefore performs better than a simple brute force algorithm.

2.2.2 Fourier-Motzkin Variable Elimination

Before explaining the Fourier-Motzkin variable elimination method, there is another method that should be mentioned. The Gaussian method [Cha93] is also used for variable elimination. It is an algorithm used for solving systems of linear equations. It determines whether a system of linear real arithmetic equalities has a solution or not.
On the other hand, the Fourier-Motzkin method is used to solve linear inequality systems [HLL90]. By using this algorithm it can be decided if a given set of linear inequalities over the real numbers is satisfiable. The basic idea of the variable elimination is to eliminate the equality constraints with the Gaussian method and then to pick a variable from the inequalities and eliminate it. To do this, we collect requirements on the lower and upper bounds of the variable we want to eliminate. Then all the lower bounds will be combined with all the upper bounds. With this we are able to find the condition that we need. This means the lowest upper bound combined with the highest lower bound. We repeat this process until all variables are eliminated. Then, we have a system of constant inequalities, where it is trivial to see whether the resulting system is true or false.

The inequalities are in the form:

$$\sum_{j=1}^{n} a_{ij} \cdot x_j \leq b_i,$$

where a_{ij} and b_i integer/rational constants,
x_j variables,
and i the constraint's index.

After resolving for variable x_n: $a_{in} \cdot x_n \leq b_i - \sum_{j=1}^{n-1} a_{ij} \cdot x_j$

(a) $\xrightarrow{a_{in} > 0}$ $x_n \leq \frac{b_i}{a_{in}} - \sum_{j=1}^{n-1} \frac{a_{ij}}{a_{in}} \cdot x_j$ upper bound

(b) $\xrightarrow{a_{in} < 0}$ $x_n \geq \frac{b_i}{a_{in}} - \sum_{j=1}^{n-1} \frac{a_{ij}}{a_{in}} \cdot x_j$ lower bound

Example:
Given the following inequality system:
1. $x + 2y \geq 3$
2. $y \leq 5$
3. $x = 0$

Gaussian elimination is applied:
(1),(3): $2y \geq 3$ (4)

Lower bound: $y \geq \frac{3}{2}$
Upper bound: $y \leq 5$

(2),(4): $y \geq \frac{3}{2} \wedge y \leq 5$

After combining the bounds:
$\Rightarrow \frac{3}{2} \leq 5$

$\Rightarrow$ True
The algorithm returns satisfiable.

It is possible that some variables do not have upper or lower bounds. In these cases the variables have on the one side no bounds which means it is enough to just choose a value that satisfies all inequalities where the variable appears.

2.2.3 Virtual Substitution

The Virtual Substitution method constructs a finite set $T \subset \mathbb{R}$ of test candidates with $\exists x_1 ... \exists x_n \ \phi \equiv \exists x_1 ... \exists x_{n-1} \bigvee_{t \in T} \phi[t//x_n]$ for a real-algebraic formula $\exists x_1 ... \exists x_n \phi$ with $n > 0$ and ϕ quantifier-free [Kos16]. $[A//B]$ stands for virtually substituting A for B. The finite set T contains representative points from sign-invariant regions. In order to compute these regions, we first need to find the real roots of univariate polynomials [Akr80]. This is achieved by using solution equations, which exist up to polynomial degree 4.

For a polynomial $ax^2 + bx + c \in \mathbb{Z}[x]$, the real roots in x are:

1. Constant in x: the real root is any real number, if $a = 0 \wedge b = 0 \wedge c = 0$
2. Linear in x: $\xi_0 = -\frac{c}{b}$, if $a = 0 \wedge b \neq 0$
3. Quadratic in x, first solution: $\xi_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$, if $a \neq 0 \wedge b^2 - 4ac \geq 0$
4. Quadratic in x, second solution: $\xi_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$, if $a \neq 0 \wedge b^2 - 4ac > 0$

Example of real roots of univariate polynomial:
Given: $2x^2 + 4x + 2 = 0$
Step 1: Calculate real roots.
Because $a = 2$ and $b^2 - 4ac = 4^2 - 4 \cdot 2 \cdot 2 = 0 \geq 0$, real quadratic roots exist. By using the third point mentioned above, the result is -1.

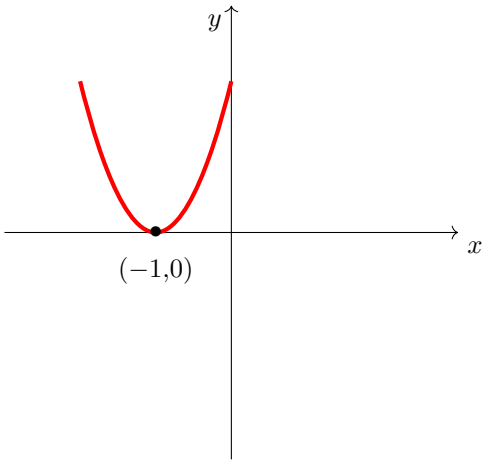

Figure 2.1: The real root of the polynomial $2x^2 + 4x + 2 = 0$

Finding real roots of a variable is also possible with multivariate polynomials. Multivariate means that we have polynomials with several variables. They can be solved exactly like the univariate polynomials with polynomial coefficients.

Example:
Given the polynomial $2yzx^2 + 2zx + 3y - 1 \in \mathbb{Z}[y,z,x]$.
It will be considered like this: 2yzx^2 + 2zx + 3y-1
Coefficients of x: $a = 2yz$, $b = 2z$ and $c = 3y - 1$

Up until now we have discussed how to describe real zeros with solution equations but not how to solve constraints yet. The method compares a multivariate polynomial to zero. We do not want to see if only the polynomial can get zero but whether a certain sign $=, <, >, \leq, \geq, \neq$ can be satisfied. The goal of the method is to check for satisfiability and return a solution or an explanation if it is unsatisfiable. There are two different cases for polynomial p. If it is constant or has no real zeros at all then we know that its sign will not change in the whole $\mathbb{R}$. Otherwise, if the polynomial is not constant, we have real zeros as defined above and we can find the possible solution intervals.

- If $p = 0$ then the interval we take is the zeros and the constant case (if $a = b = 0$) where we may take any point from the $\mathbb{R}$ to check the sign. This can be done by taking the interval $(-\infty, \infty)$.

- If $p < 0$, $p > 0$ or $p \neq 0$ then we take the opposite intervals. That means we do not want to take the zeros because $p = 0$ is false.

- If $p \leq 0$ or $p \geq 0$ then the intervals are like the second case but with the zeros included.

It is possible to have several polynomial constraints. In this case, we construct a possible common solution interval. The problem is that we do not know the exact endpoints of the intersections, but we can represent each candidate solution interval by its leftmost point. If the interval is open, then we take the leftmost point plus a very small value ϵ. For each constraint we add test candidates.

- $p = 0$, $p \leq 0$, $p \geq 0$: We take each real zero of p and $-\infty$

- $p < 0$, $p > 0$, $p \neq 0$: We take each real zero of p plus ϵ and $-\infty$

Example of test candidates generation:
Given: $2x^2 + 4x + 2 = 0$, eliminate x

According to step 1, real root of the polynomial is -1.
Step 2: Generation of test candidates.
First test candidate: $-\infty$, for all the constraints.
Second test candidate: -1, without adding an ϵ as it is not a strict inequality.

For the final step the Virtual Substitution is used. We substitute the test candidates for each occurance of the current variable in the whole formula, which in the example above is only one inequality. If we have more than one inequality, we substitute and with operand OR we connect each resulting formula that comes after substitution.
Here the standard substitution $\phi[t/x]$ can lead to formulas which contain ϵ, $-\infty$, $\sqrt{}$ or division. Therefore, we use Virtual Substitution $\phi[t//x]$ that generates real algebraic formulas which are semantically equivalent to the standard substitution but do not contain the symbols above. To do this, there are some rules that define the instructions concerning the usage of virtual substitution from a test candidate into a constraint. These rules are mention in the Chapter 8.

If the test candidate is a fraction, then we multiply every term of each inequality with the highest degree of the substituted variable that appears in every inequality.

E.g., it works as following for equalities $(ax^2 + bx + c = 0)$ $[\frac{q}{r}//x]$:

Quadratic:

$$\begin{aligned} &= (ax^2 + bx + c)[\frac{q}{r}/x] \cdot r^2 \\ &= (a\frac{q^2}{r^2} + b\frac{q}{r} + c) \cdot r^2 \\ &= aq^2 + bqr + cr^2 = 0 \end{aligned} \tag{2.1}$$

Linear:

$$\begin{aligned} &= (bx + c)[\frac{q}{r}/x] \cdot r^1 \\ &= (b\frac{q}{r} + c) \cdot r^1 \\ &= b \cdot q + c \cdot r = 0 \end{aligned} \tag{2.2}$$

This is only achievable because we have 0 on the right-hand side. Otherwise, the multiplication is not possible.

Example of Virtual Substitution:
Step 3: Virtual substitution.

1. $(2x^2 + 4x + 2 = 0)$ $[-\infty//x]$
   $\Rightarrow (2 = 0 \wedge 4 = 0 \wedge 2 = 0)$
   $\Rightarrow$ (False $\wedge$ False $\wedge$ False)
   $\Rightarrow$ False

2. $(2x^2 + 4x + 2 = 0)$ $[-1//x]$
   $\Rightarrow (2 \cdot (-1)^2 + 4 \cdot (-1) + 2 = 0)$
   $\Rightarrow 2 - 4 + 2 = 0$
   $\Rightarrow$ True

   $\Rightarrow$ False $\vee$ True
   $\Rightarrow$ True

The algorithm returns satisfiable for this formula.

Something important to point out, is the problem when the algorithm finds a test candidate with a root, that contains a different variable. The implemented algorithm is able to make calculations when there are only numbers in the root, but in case a variable is included, it is impossible.

Solution:
Every term $(ax^2 + bx + c)$ $[\frac{q+r\sqrt{t}}{s}//x]$ can be written in the form $\frac{\hat{q}+\hat{r}\sqrt{\hat{t}}}{\hat{s}}$ where q,$\hat{q}$,r,$\hat{r}$,s,$\hat{s}$,t,$\hat{t}$ the polynomials. Then the algorithm applies the Virtual Substitution rules for root.
In case of p(x) = 0 it generates these two inequalities: $(\hat{q}\hat{r} \leq 0 \wedge \hat{q}^2 - \hat{r}^2 t = 0)$

Explanation:

$$\begin{aligned} &\frac{\hat{q}+\hat{r}\sqrt{t}}{s} = 0 \Leftrightarrow \hat{q} + \hat{r}\sqrt{t} = 0 \\ &\Leftrightarrow \hat{q}\hat{r} \leq 0 \wedge \|\hat{q}\| = \|\hat{r}\sqrt{t}\| \\ &\Leftrightarrow \hat{q}\hat{r} \leq 0 \wedge \hat{q}^2 - \hat{r}^2 t = 0 \end{aligned}$$

Depending on the inequality symbol, different inequalities are created 8.3.

Example:
Given: $\Phi := ((x^2 - 2x - y \geq 0) \wedge (x \geq 0))$.

Real roots: $\frac{2+\sqrt{4+4y}}{2}$, $\frac{2-\sqrt{4+4y}}{2}$ and 0.
Test candidates: $\frac{2+\sqrt{4+4y}}{2}$, $\frac{2-\sqrt{4+4y}}{2}$, 0 and $-\infty$ for all constraints.

For this example we will only focus on the first test candidate $\frac{2+\sqrt{4+4y}}{2}$ and substitute it in the second inequality $x \geq 0$.
Resulting inequality: $\frac{2+\sqrt{4+4y}}{2} \geq 0$.
Now we substitute the variable y. We observe that it is in a root, so we apply the transform root rules mentioned above.

Polynomials: $q = 2$, $r = 1$, $t = 4 + 4y$, $s = 2$
Inequalities: $2 \cdot 1 \leq 0 \wedge 2^2 - 1(4 + 4y) = 0 \Rightarrow 2 \leq 0 \wedge -4y = 0$

With the remaining inequalities we repeat the process of finding real roots and test candidates as known.

Chapter 3

Methodology

In this part, a rough introduction of the work will be described in order to have a first overview on how each method works.

3.1 SAT

For the SAT exercises, the students have a choice of 3 difficulties (1-3). According to the difficulties the algorithm chooses how many variables and clauses it can generate for the current exercise. In order to solve this exercise, the class SATSolver is created.

3.2 Fourier-Motzkin

The Fourier-Motzkin variable elimination creates an exercise with the option to choose a difficulty (1-3) as well. The number of variables and clauses of this exercise can be adjusted according to the level of difficulty. The class FMSolver tries to solve the given exercise.

3.3 Virtual Substitution

The students can choose for the Virtual Substitution exercises whether the exercise will have one or two variables. Difficulty 1 has always one variable to substitute. Difficulty 2 has sometimes one or two, whereas with difficulty 3 the program generates exercises that include two variables. For such exercises, the class VSSolver is called to solve the current task.

More details about the algorithms are explained in the next chapter.

Chapter 4

Implementation

4.1 Class Hierarchy

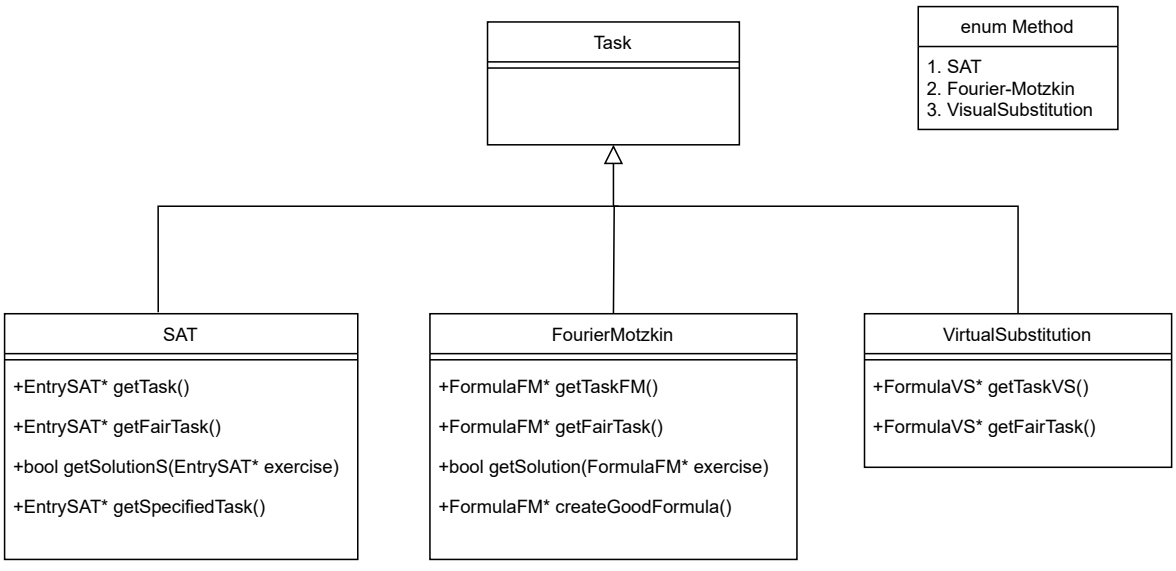

Figure 4.1: UML diagram for creating a task

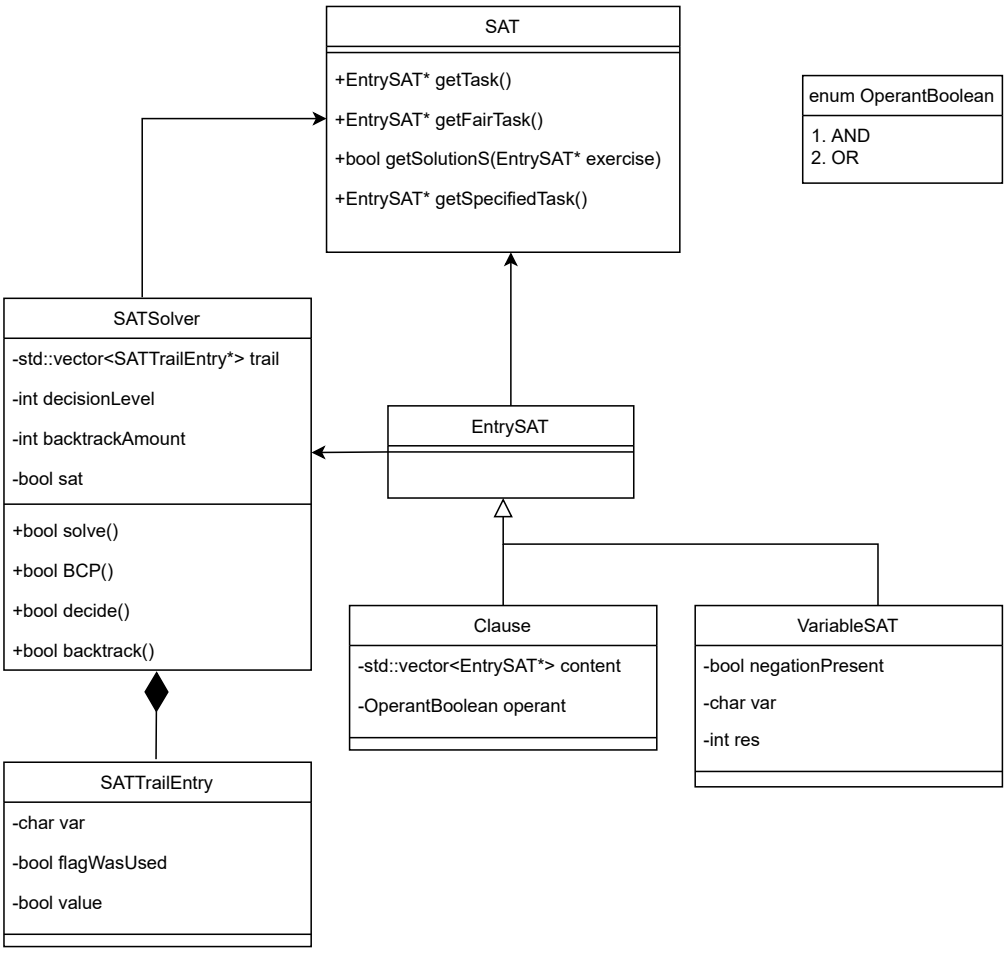

Figure 4.2: UML for SAT

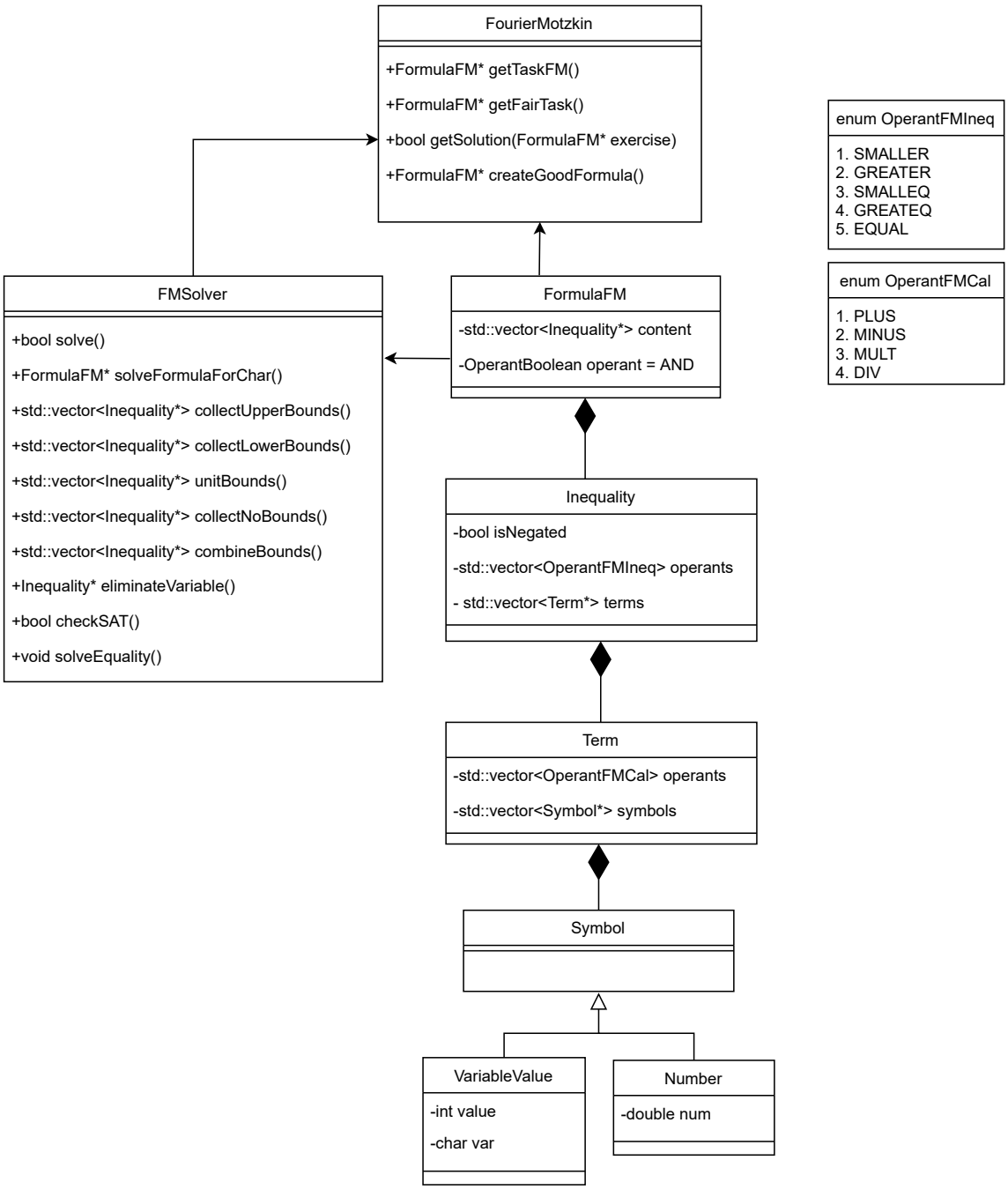

Figure 4.3: UML for Fourier-Motzkin

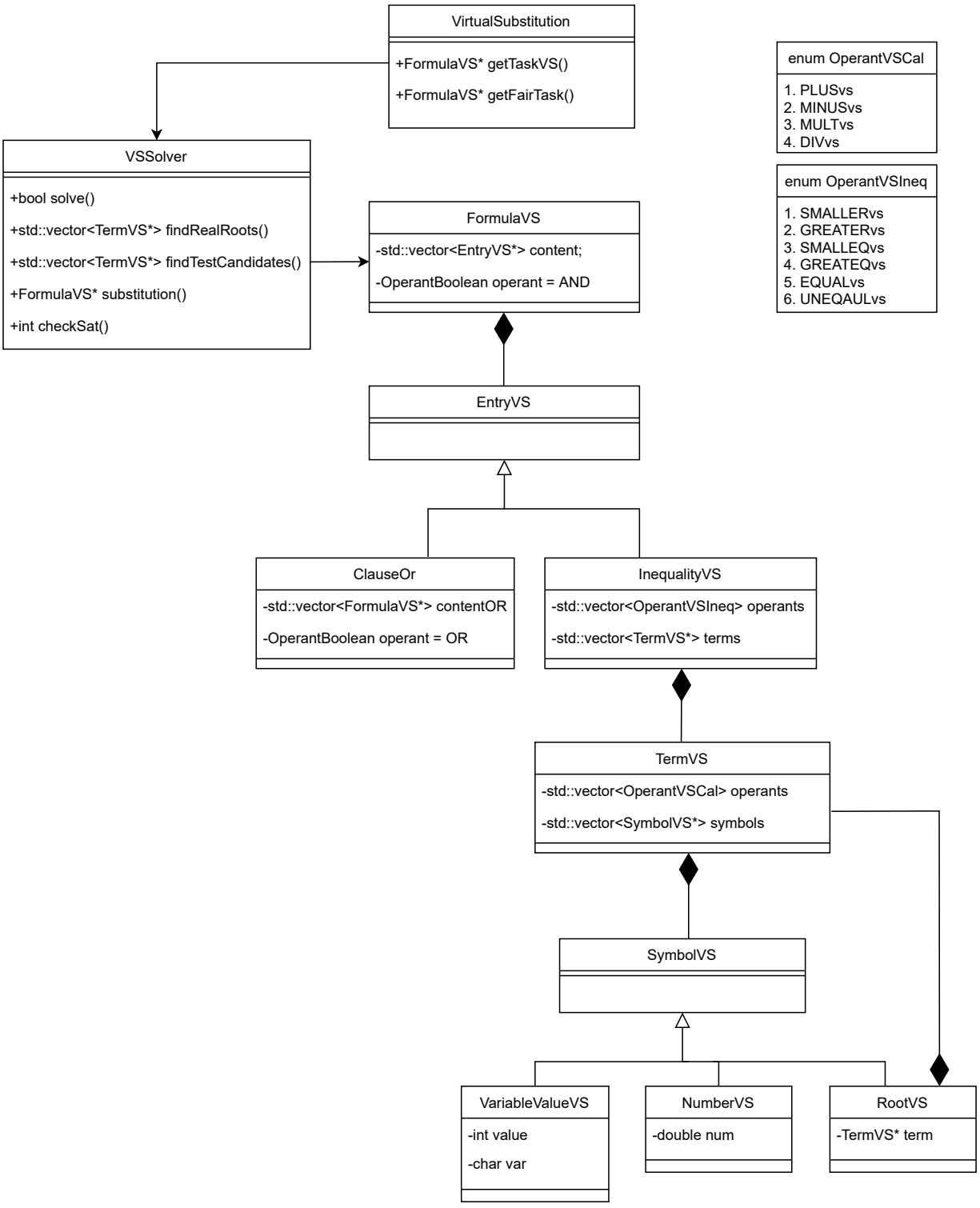

Figure 4.4: UML for Virtual Substitution

4.2 SAT Framework

The class hierarchy of the SAT problem generator is illustrated as an UML diagram in Figure 4.2.
SAT exercises are generated after the function **getTask()** is called. This function creates a vector of clauses that are connected with operand AND. In the OR clauses there is another vector of random chosen variables.
These have 3 characterics:

1. They can be either negated or not.
2. The name of the variable.
3. The *res* of the variable.

The *res* is used for the solution to see if the variable is assigned or not. If $res = 1$ then the variable is assigned to true. If $res = 0$ then the variable is assigned to false. Otherwise, the default value of *res* is -1 when the variable is unassigned.
The result of the function is an EntrySAT. EntrySAT is a clause with variables or a single variable. After creating the OR clauses the algorithm packs them all together in another EntrySAT which in this case the operand is AND. This results into a conjuctive normal form formula 2.2.1.

The function **getFairTask()** is able to take the current exercise and generate a similar one. It randomly selects which action the algorithm must follow.

There are 3 possibilities:

1. Change of the positions of variables or clauses of the current exercise.
2. Change the name of the variable throughout the whole formula.
3. Both of the above.

The difficulty stays exactly the same as selected from the previous exercise, in order to be fair.

Another function that needs to be explained is the **getSpecifiedTask()**. Here the students have the opportunity to choose a number of decision levels, backtracks and difficulties. By creating a fair task using this function, the number or decision levels and backtracks must be approximately the same. Because the difficulty of the exercise depends mostly on those two factors and both generated exercises should require the same effort to be solved.

To solve a SAT exercise the class SATSolver is created. The function **getSolutions()** tries to solve and give a solution for the given formula. The class SATSolver includes a function called **solve()** which has as input the exercise that needs to be solved.

This function looks like this:

Algorithm 1 solve(EntrySAT* exercise)

```
if !BCP(exercise) then
    sat ← false;
    return false;
end if

while true do
    if !decide(exercise) then
        sat ← true;
        return true;
    end if
    while (!BCP(exercise)) do
        if !backtrack(exercise) then
            sat ← false;
            return false;
        end if
    end while
end while
return false;
```

Let us now get in more detail about the three functions used in **solve()**.

1. The **BCP()** function is for propagation. The algorithm start searching from left to right for a unit clause that exists from the beginning. If it is the case, then regardless the default value that has to be assigned to the variable, it assigns it to true. Propagation only takes place when the clause is a unit. When from a unit clause or more unit clauses the result is unsatisfiable then the algorithm returns directly unsatisfiable for the current task.

2. If there is no unit clause the function **decide()** is called. The algorithm starts by collecting the two unassigned literals for each clause in the formula. If there are no unassigned literals, there is no decision to be made. With the variable that has priority, which is given by the static order and the algorithm assigns it to the default value. Default in the program is the variable named SATVariableDecisionInitial. It is initialized to 0 but it can also be changed to 1. After the assignment, the algorithm looks again for a unit clause starting from the leftmost clause. If a unit clause is found, **BCP()** is called and assigns the last variable. If not the same process is repeated. All the decision levels and propagations are written in the trail. For this list a new class called **SATTrailEntry** has been made in order to save every entry of the trail.

3. The last function **backtrack()** is used when the formula is unsatisfiable in the last decision level. Here the algorithm tries to find a solution by backtracking the last decision. If there was no decision made, which means the trail contains only assignments at decision level 0, then it can not take back any decisions

and the solver returns unsatisfiable. Otherwise, it deletes the last entry of the trail with backtracking and change the value from true to false, if it was true before and vice versa if it was false to true. To make sure that this conversion has not been already made, we use a flag. This flag shows if the value has already been changed or not. If yes, then we can not do the same again and wind up in an endless loop. This being the case, the algorithm goes another step back, if possible. It repeats the same process until the flag is set to false and checks if the new decision returns satisfiable as an answer. If it is again not the case, and the trail is not yet at decision level 0, we do this all over again including propagation and new decisions. If the formula is unsatisfiable and all the possibilities of assigning the variables have been taken then the solver returns unsatisfiable. If not, the formula has a solution.

4.3 Fourier-Motzkin Framework

The class hierarchy of the Fourier-Motzkin modules is shown in the Figure 4.3.
To generate an exercise to be solved with Fourier-Motzkin variable elimination, the function **getTaskFM()** is called. This function creates a new FormulaFM which contains a vector of inequalities that are connected with the operand AND. Each inequality contains a vector as well, which contains terms. Every term in the inequality contains a vector of symbols. The class **Symbol** is an upper class of **VariableValue** and **Number**. This means that symbol can either be a variable or a number. The class **TermVS** includes an enum **OperantFMCal** which has calculation symbols such as plus, minus, multiplication and division. The class **InequalityVS** on the other hand, uses the enum **OperantFMIneq**, which includes all $\sim \in \{<, >, =, \leq, \geq\}$. The variables are chosen randomly like in the SAT exercises. The same goes for the numbers, symbols and inequalities that are used. In this type of exercise, the clauses that are with operand OR connected are not present, because they were not treated in the lecture of satisfiability checking [ÁK16].

The class **FourierMotzkin** has a relevant function to the **SAT** class, **getFairTask()**. The only difference is that it also includes a function called **reverseInequalities()**. An example for this has been mentioned above 1. It is also possible to change the name of the variable or the positions of inequalities. Again, just like in SAT, the actions are chosen randomly.

Next is the **createGoodFormula()** function. By using this one, students are given the chance to have exercises in which the results are small integers and not something complex with fractions.

This function has as input three parameters:

1. The difficulty, that determines how many inequalities should be created at the beginning.
2. The mutationDepth which determines the maximal number of mutations that an inequality can have.
3. The parameter totalIneq which decides how many inequalities should be at the end.

In a while loop, the algorithm checks whether it is okay to add a new variable to one of the inequalities. By adding a variable, from one inequality results two. As long as the size of the content of the formula, which means the number of the inequalities is smaller than the parameter totalIneq, the algorithm adds new variables. At the end, the algorithm as long as the depth is not zero, mutates random inequalities with either adding or multipling by a random number or even both.

The algorithm looks like this:

1. Automatic generation of inequalities that contain only two terms with one number each.
2. Mutation by inserting variables into one inequality.
3. Mutation by adding a random number on both sides of the inequality.
4. Or mutation by multiplying each side by a random number.
5. Or both.

For personal preference the following amount is computed so that an inequality will be in average so many times mutated:

$$\text{AmountOfMutations} = (\text{number of inequalities} \cdot \text{mutationDepth}) \cdot \tfrac{2}{3}$$

Some inequalities will have more and some less mutations. It is chosen randomly which inequality will be mutated. In the class **Inequality** there is an attribute which has the current number of mutations of each inequality. With this attribute the algorithm checks if another mutation is possible for this inequality. As mentioned above, each inequality can have maximal mutation number equal to the mutationDepth.

Example:
Given: dif $= 1$, mutationDepth $= 2$, totalIneq $= 3$:
Computing the amountOfMutations $= (3 \cdot 2) \cdot \frac{2}{3} = 4$ mutations.

Step 1: $(0 < 3)$
Step 2: Insert variable x: $((0 < x) \wedge (x < 3))$
Step 3: Insert variable y: $((0 < y) \wedge (y < x) \wedge (x < 3))$
Step 4: Add a random number: $((0 + 4 < y + 4) \wedge (y < x) \wedge (x < 3))$
Step 5: Add a random number: $((4 < y + 4) \wedge (y + 1 < x + 1) \wedge (x < 3))$
Step 6: Multiply by a random number: $((4 < y + 4) \wedge (y + 1 < x + 1) \wedge (x \cdot 2 < 3 \cdot 2))$
Step 7: Add a random number: $((4 + 1 < y + 4 + 1) \wedge (y + 1 < x + 1) \wedge (x \cdot 2 < 6))$

Every time a mutation is made, the function **mergeTerm()** is called. This function simplifies the terms. In order to make it less obvious for the students the algorithm scrambles the formula such as swapping two inequalities or reversing them. For randomization, the function rand() % x is used, which in C++ gives a random number between 0 and x. In this way, a variety of exercises is being generated. An interesting feature of the function **createGoodFormula()** is that a constructor in the **Inequality** class with an input of boolean sat is called. If $sat = false$ then it creates an inequality which is unsatisfiable. Otherwise, satisfiable. To achieve this, we create the terms according to the input of this function. The first term is completely

random but the second one has an interval of the possible values in order to be the result of the inequality false or true at the end. Here the inequalities are also able to be negated at the beginning or not.

The solver of a Fourier-Motzkin task is in the class **FMSolver**. This class contains the function **solve()**, which solves the given exercise.

Algorithm 2 solve(FormulaFM* exercise)

```
std::set<char> setChars = exercise→getAllUsedVariables();
std::vector<char> vecChar(setChars.begin(), setChars.end());
sort(vecChar.begin(), vecChar.end());
FormulaFM* input = exercise→copy();

for unsigned long i = 0 ; i < vecChar.size() ; ++i do
    FormulaFM* output = solveFormulaForChar(input,vecChar.at(i));
    input = output;
end for

if checkSAT(input) then
    return true;
else
    return false;
end if
```

In this algorithm a set of all the used variables of the given exercise is created. After that, for the first variable of the set, the function **solveFormulaForChar()** is called. This function mainly eliminates the current variable. After the elimination, the same process will be repeated until all variables are eliminated, if it is possible. The input of the **solveFormulaForChar()** changes with the current variable and the current formula. The formula changes after the elimination of the first variable, and so on until there are no variables in the formula left.

Algorithm 3 solveFormulaForChar(FormulaFM* exercise, char c)

```
FormulaFM* res = new FormulaFM();
FormulaFM* modifiedExercise = exercise→copy();

preprocessFormula(modifiedExercise);

solveEquality(modifiedExercise,c);

std::vector<Inequality*> upper = collectUpperBounds(modifiedExercise, c);
std::vector<Inequality*> lower = collectLowerBounds(modifiedExercise, c);
std::vector<Inequality*> unit = unitBounds(upper,lower);
std::vector<Inequality*> noBounds = collectNoBounds(modifiedExercise, unit, c);
std::vector<Inequality*> resIneq = combineBounds(upper, lower , c);

for auto it : noBounds do
    res→content.push_back(it);
end for

for auto it : resIneq do
    res→content.push_back(it);
end for
return res;
```

Before eliminating the variables, the function **preprocessFormula()** is called. This function calls:

- The function **resolveNegatedIneq()**, which removes the negation from the inequalities while applying the necessary modifications. Then it resolves the zeros.
- The function **resolveZeros()** that checks each term and returns zero in case of multiplication with zero or division with zero as dividend.
- The function **merge()**, which merges the inequalities. This function is often called in the program to make the inequalities more compact. Its functionality is exactly the same as the function **mergeTerm()**, but for the whole inequality.

After the preprocess is done, the algorithm searches to find equalities in the formula that contain the current variable that must be eliminated. If such an equality exists, the three following functions are called:

First function: **NormalizeIneq()**.
If the term which contains the current variable is negated, the function changes the minus to plus and switches it to the other side of the inequality.

Example:
$4 = 5 - 2x \Rightarrow 2x + 4 = 5$

Second function: **IsolateTerm()**.
It seperates the term that includes the current variable from all the others.

Example:
$2x + 4 = 5 \Rightarrow 2x = 5 - 4 \Rightarrow 2x = 1$

Third function: **IsolateVariable()**.
Isolates the variable if it is multiplied with another symbol.

Example:
$2x = 1 \Rightarrow x = \frac{1}{2}$

After these steps, the algorithm substitutes the result of the equality to every variable in the formula that is the current eliminating variable. Then it finds the upper and lower bounds of the current variable. The function **unitBounds()** returns all the bounds together. Then with the function **collectNoBounds()**, all the inequalities from the formula that do not include lower and upper bounds of the variable are collected. Then all the upper bounds will be combined with all the lower bounds and create a new formula with the combined inequalities. Here the variable is eliminated by combining the lower bound of the variable with its upper bound. The inequalities that are collected from the **collectNoBounds()** function are directly inserted to the resulting formula. This process will be repeated so many times, until all the variables are eliminated. It is possible that a variable can not be eliminated if it has only lower or only upper bounds. At the end, this does not affect the solution, since we can just pick a solution that satisfies the remaining bound. After the algorithm eliminates as much as possible variables, it checks each resulting inequality for satisfiability.

4.4 Virtual Substitution Framework

For Virtual Substitution exercises, the class **getTaskVS()** is called. The function creates a FormulaVS. This formula contains either inequalities or clauses with operand OR. If the entry in the formula is an OR clause, then this clause has also a vector of formulas that include inequalities. Every inequality in the formula is merged with the function **merge()** and packed to the left term whereas on the right term is just a zero. This is done in order to match the presented examples shown in the lecture "Satisfiability Checking" [ÁC12]. Inequalities have a vector of terms and a vector of operands with the inequality symbols. The terms of the inequalities contain exactly like in the Fourier-Motzkin symbols, that are in the class SymbolVS. The numbers and variables of the exercises are from the classes **NumberVS** and **VariableValueVS** analog. These are underclasses of **SymbolVS**. Special term in this type of exercise is the root. The class **RootVS** is also an underclass of **SymbolVS** and contains one term under a root. The class **TermVS** uses the enum class **OperantVSCal**, which contains the same symbols as in Fourier-Motzkin. Terms contain a vector of those calculation symbols. The enum **OperantVSIneq** is used in the class **InequalityVS**, where the only difference from the enum **OperantFMIneq** is that the symbol $\neq$ also appears in the exercises. The resulting formula is connected with the operand AND. To create an exercise with square roots that are integers the algorithm uses the function **generateSquareTerm()**. This function generates a quadratic polynomial that

contains only one variable and its solutions are numbers that belong to the set of integers.
Similar to the other two methods, Virtual Substitution has also its solver in the class **VSSolver**. The **solve()** function is shown in Algorithm 4.

Algorithm 4 solve(FormulaVS* exercise)

```
std::set<char> setChars = exercise→getAllUsedVariables();
std::vector<char> vecChar(setChars.begin(), setChars.end());
sort(vecChar.begin(), vecChar.end());
FormulaVS* cur = exercise→copy();

for unsigned long i = 0 ; i < vecChar.size() ; ++i do
    cur->transformRootRule(vecChar.at(i));
    std::vector<TermVS*> testCand = getTestCand(cur, vecChar.at(i));
    removeDuplicateCandidates(&testCand);
    FormulaVS* resSubstitution = substitution(cur, testCand, vecChar.at(i));
    simplifyFormula(resSubstitution);

    if checkSat(resSubstitution, VSCheckSatDebug) == 0 then
        return false;
    end if

    cur = resSubstitution;
end for

return true;
```

The algorithm starts by sorting all the variables used in the exercise. Then the function **transformRootRule()** is called. This function searches through the whole formula to find whether there is an inequality with a root that contains the current variable. If this is the case all the necessary parameters are calculated to form the inequalities by applying the root rules 8.3.
Then the function **getTestCand()** is called. This function finds the real roots of the variable by calling **findRealRoots**. The function **findRealRoots()** in the class **InequalityVS** is shown in the upcoming algorithm.

Algorithm 5 findRealRoots(char c)

```
isolateRightSide();
std::vector<TermVS*> abcPositions = findPositionsInPolynomial(c);

if operants.at(0) == SMALLERvs || operants.at(0) == GREATERvs || operants.at(0) == UNEQUALvs then
    addEpsilon = true;
end if

if a == 0 then
    if b == 0 then
        if c is NumberVS then
            double num = abcPositions.at(2)→symbols.at(0)→num;
            if abcPositions.at(2)→operants.at(0) == MINUSvs then
                num *= -1;
            end if
            NumberVS* n = new NumberVS(num);
            TermVS* newT = new TermVS();
            newT→symbols.push_back(n);
            res.push_back(newT);
            return res;
        else
            TermVS* newT = terms.at(0)→copy();
            res.push_back(newT);
            return res;
        end if
    else
        TermVS* t = findRealRootsLinear(c);
        if addEpsilon then
            add epsilon to the terms in t
        end if
        res.push_back(t);
        return res;
    end if
else
    TermVS* t = findRealRootsLinear(c);
    if addEpsilon then
        add epsilon to the term t
    end if
    res.push_back(t);
    return res;
end if

std::vector<TermVS*> temp = findRealRootsQuadratic(c);

if addEpsilon then
    for auto it: temp do
        add epsilon to it
    end for
end if

for auto it2 : temp do
    res.push_back(it2);
end for

for unsigned long i = 0; i < res.size();++i do
    res.at(i) = mergeTerm(res.at(i));
end for
return res;
```

The algorithm calls at the beginning **merge()** to the inequality and isolates its right term. If the inequality does not contain the current variable then there is nothing to be done with the inequality and iteratively go to the next one. Using the function **findPositionsInPolynomial()**, has as output the coefficients of the searched variable in the current polynomial. The output is a vector of TermVS. The first position of this vector is always the coefficient of the quadratic current variable. The next position is the coefficient infront of the variable with degree 1 and lastly is the constant where the variable has degree 0. These positions are also described as a b and c, in order to match the syntax of the lecture [ÁK16]. If the sign infront of the coefficient is negated, then the term in the vector is multiplied by -1.

Example:
Given: $2x^2 - x + 3 - y \leq 0$
The function **findPositionsInPolynomial(x)** returns the vector [2,-1,3-y].

Afterwards, the algorithm adds an ϵ if the inequality symbol is $<$, $>$ or $\neq$. There are three possible real roots types.

1. The polynomial is neither square nor linear. Then there is only a constant in the inequality and it is returned.
2. The polynomial is linear. The function **findRealRootsLinear()** is called. To calculate the real linear roots of this polynomial the second calculation point is used 2.2.3.
3. The polynomial is quadratic. The function **findRealRootsQuadratic()** returns the real roots for this polynomial. Again, in section 2.2.3 it is mentioned in step 3 and 4 how to calculate those roots.

For each root, an ϵ is added if the inequality symbol is strict. The function **findRealRoots()** returns a vector of terms. These terms are the real roots of the current variable from the current inequality. This process will be repeated for all the inequalities of the formula in order to collect all the real roots of the current variable.

Then the algorithm creates a vector with all the test candidates. As mentioned above to the definition of the virtual substitution, the test candidates are also a vector that contains terms. In addition to that, the real root $-\infty$ is inserted for all the constraints.
After all the test candidates for the variable are listed into the vector, **removeDuplicateCandidates()** is called to remove duplicated test candidates in the vector.
Then Virtual Substitution is applied to the formula with the test candidates of the variable. If the test candidate includes an ϵ the rules for ϵ are applied 8.4. For test candidates in form $\frac{q}{r}$ the function **multiplyR()** is used to multiply the polynomial with the highest degree of the variable in order to avoid possible fractions. New inequalities are created due to the rules shown in the tables in Chapter 8.2.
The function **simplifyFormula()** is called after substitution for simplifying the resulting inequalities so that the result is more clear.
It contains the following functions:

1. **SimplifyFormula()**: Replaces the unsatisfiable inequalities with (1=0) and the satisfiable with (0=0).

2. **SummarizeFormula()**: Summarizes inequalities that already contain at least one unsatisfiable inequality to unsatisfiable.

3. **CombineBooleanAssignments()**: Removes all the satisfiable inequalities and lets all the inequalities that are undefined.

4. **ResolveDepth()**: Reduces the depth of the formula. In case the formula contains only one clause with operand OR and this clause has another clause with operant $\wedge$, it reduces it to just the last formula.

Examples:
SimplifyFormula(): $((5 < 0) \wedge (1 \geq 0)) \Rightarrow ((1 = 0) \wedge (0 = 0))$
SummarizeFormula(): $((1 = 0) \wedge (1 = 0) \wedge (x + 3 > 0)) \Rightarrow ((1 = 0))$
CombineBooleanAssignments(): $((0 = 0) \wedge (x + 3 > 0)) \Rightarrow ((x + 3 > 0))$
ResolveDepth(): $((((x + y < 6)))) \Rightarrow ((x + y < 6))$

After all the variables are substituted, the algorithm checks for satisfiability of the formula by calling **checkSat()**. This function is called recursive to all the clauses of the formula. It creates a vector with 1 for satisfiable, 0 for unsatisfiable and -1 for undefined. Undefined is the case when another variable is still in the formula and the algorithm can not compare the terms of the inequality. For a clause with operand AND all the inequalities should be true for satisfiable. When there is at least one false, unsatisfiable is returned. If there are some undefined inequalities in the clause then the whole result is undefined. For OR clauses, it needs at least one satisfiable clause to return 1. If all of them are unsatisfiable then 0 is returned. In case of some unsatisfiable and some undefined the result is again undefined and **solve()** is called again.

Chapter 5

GUI

For this thesis a user interface has been created by using Qt Creator.

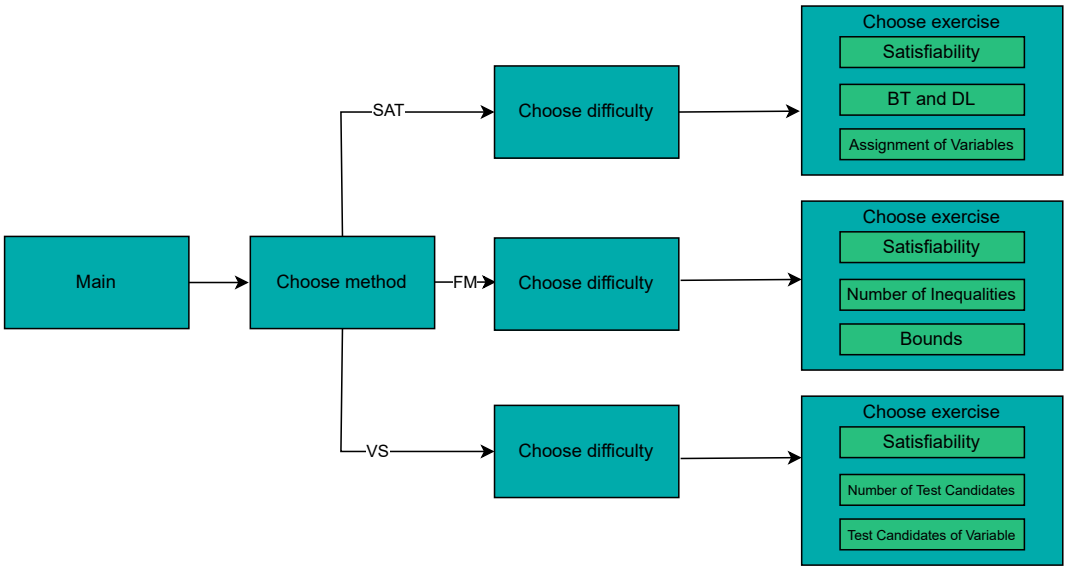

Figure 5.1: An overview of the user interface

The students can choose the method they want to exercise and then the difficulty. Each method has three types of exercises:

SAT:

1. Satisfiability: The students should answer with yes or no if the given formula is satisfiable.
2. Backtrack and decision level: The number of backtracks and decision levels is asked, that have been made in order to solve the exercise.
3. Assignment of variables: The variables are given in the solution and the students should click true if the variable is assigned to true and false otherwise. If the

exercise is unsatisfiable then there is an option to click unsatisfiable.

For all the tasks the students should solve the whole exercise in order to be able to answer these three types of questions.

Fourier-Motzkin:

1. Satisfiability: Similar to the first task of SAT, the students have to decide whether the given formula is satisfiable or not.
2. Number of inequalities: This subtask asks the number of the remaining non-trivial constraints inequalities, after the elimination of the first variable. Non-trivial constraints inequalities are the inequalities that contain at least one variable.
3. Bounds: In this type of exercise the number of the lower and upper bounds of the first variable is asked. The student should not solve the whole exercise to answer this question.

Virtual Substitution:

1. Satisfiability: The students are asked to apply the method of Virtual Substitution to the current formula and decide whether it is satisfiable or not.
2. Number of test candidates: This task requires the number of test candidates of the first variable that will be substituted.
3. Test candidates of variable: The students have a range of possible test candidates and must click the right ones for the first variable.

After the students have inserted their answer, the click button prints a label with the feedback. When the check button is clicked, another button for explanation appears. The students can see all the steps of the solutions. In SAT exercises the trail is given. For Fourier-Motzkin tasks, all the preprocess steps are included, the lower and upper bounds, their combination and the resulting formula after elimination of one variable. In the explanation of Virtual Substitution exercises the test candidates are shown and every step after substituting each test candidate.

Chapter 6

Evaluation

In this chapter, the results of the questionnaire will be presented. Eight students have been asked to download the application, use it and then anwswer some questions. Most of the students tried all the methods, whereas some others only SAT or Fourier-Motzkin exercises. All the students agreed that it took them fewer as 10 minutes to read one exercise and almost all of them understood the tasks immediately. The tasks are considered as enough meaningful to be asked for those purposes. The 7 out of 8 students found the questions of the tasks very clear. To solve an exercise it did not take them more than 10 minutes. The solutions were for the students mostly understandable but some of them consider that not all important steps were included. Of course, for difficulty 3, where the exercises are more complex, the solutions are more complicated. The numbers of the results were not too complicated. Only one student answered that it was possible to guess the answer whereas the others said either sometimes or not at all. It is possible to guess the answers for the tasks "Satisfiability". However, for the majority of the tasks it is not that easy because more answers than two are provided. The following task is for assigning the variables in SAT exercise.

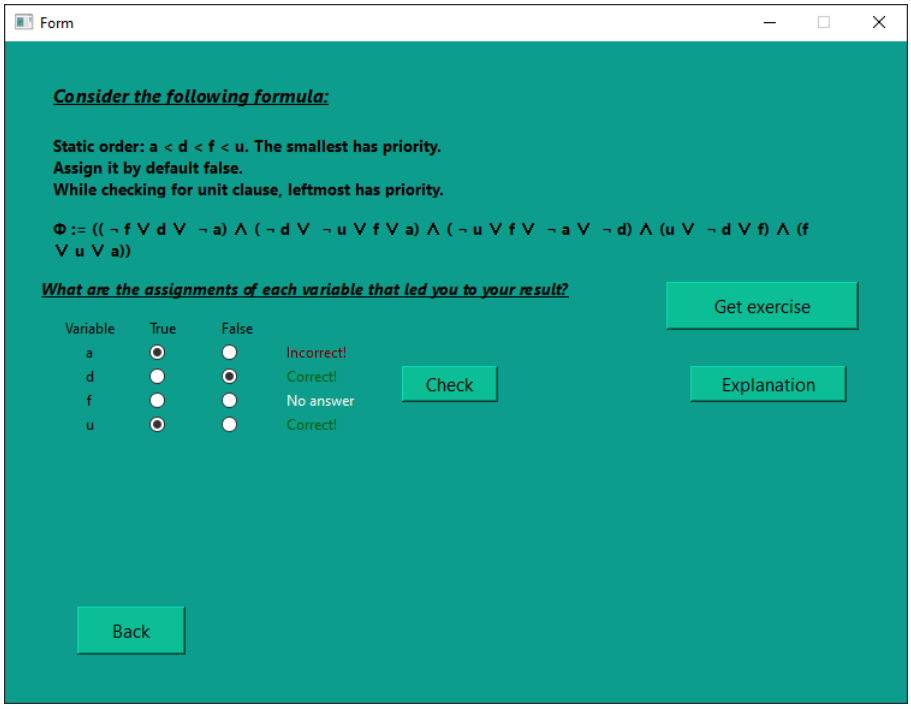

Figure 6.1: Assigning variables in SAT exercise

For those who changed the difficulty, it was noticable that the numbers of variables or clauses changed.
Some feedbacks from the students were that the application was very helpful for practising for SAT checking. Some of them wrote that the variaty of the tasks and the generated exercises is really good. Also the purpose of this application is clear, since the students are able to repeat exercises and always generate new ones. Another feedback was that the solutions could have been more detailed with more definitions. Suggestions have also been given from some students. One of them was to provide examples to the definitions that were explained in the explanation part. There were a lot of suggestions for scaling and changing the size of the window of the application and its buttons. This will be fixed in order to make it more flexible for the users to scale it depending on their needs. There was a suggestion for the Fourier-Motzkin exercises that said to make clear when there are no bounds. Also, to state at the end of the explanation if the formula is satisfiable or not. Both of them are now fixed in the application so that it is more easy for the students to understand.

Chapter 7

Conclusion

7.1 Summary

In this thesis an introduction to the satisfiability checking has been made. The used methods SAT, Fourier-Motzkin, Virtual Substitution and their solvers have been explained, in order to understand the implementation. Related work such z3 tool has been pointed out and compared to the current algorithm of the application 1.2. The application provides something that does not currently exist, since only simplex solvers are currently available. Criteria according the problems, tasks and solutions have been listed in order to create a good pedagogical application for the students [2.1]. For this thesis an exercise generator has been implemented for students to practise satisfiability checking. This application generates three types of exercises for each method. Its structure is described in the implementation Chapter 4. In the following, the user interface and its structure were presented. All the tasks have been mentioned and in a few words described. The evaluation Chapter shows the results after giving a questionnaire to some students that are familiar with satisfiability checking. The questions of this questionnaire were mainly if the application has clear questions and understandable solutions. Also if the students were able to understand the tasks immediately and if the application was helpful. In general, if the criteria that were listed at the beginning of the thesis were succesfully presented in the application. According to the results, most of the students found the exercise easy to read and to understand. The solutions were helpful but some students suggested to make the explanations more detailed and include examples. The interval for the students to read and solve an exercise lays approximately 5-10 minutes. Most of them stated that the result numbers where not complicated. Some of them wrote that it depends on the difficulty. In general, they concidered this application very helpful to practise satisfiability checking.

7.2 Future Work

Improvements are always possible for an application. New features such as timer could be added to the application. With this feature the students know how much time they needed for an exercise. In addition, the solutions can give points to the right answers or give tips and advices to the students. Moreover, in the implementation

and more specific in Virtual Substitution, the exercises that can be generated are polynoms up to degree 2. More complex exercises can be generated but that is not the case for this implementation. Also for the Fourier-Motzkin exercises there are no clauses with operand OR. This could appear but because it was not treated in the lecture, it is left out. There were also some suggestions from the students to include a progress bar or a score for the students, but due to limited time this can not be contained in the application.

Chapter 8

Virtual Substitution Rules

$p(x) \sim 0$	$(p(x) \sim 0)\ [-\infty//x]$
$bx + c = 0$	$b = 0 \wedge c = 0$
$bx + c \neq 0$	$b \neq 0 \vee c \neq 0$
$bx + c < 0$	$b > 0 \vee (b = 0 \wedge c < 0)$
$bx + c > 0$	$b < 0 \vee (b = 0 \wedge c > 0)$
$bx + c \leq 0$	$b > 0 \vee (b = 0 \wedge c \leq 0)$
$bx + c \geq 0$	$b < 0 \vee (b = 0 \wedge c \geq 0)$
$ax^2 + bx + c = 0$	$a = 0 \wedge b = 0 \wedge c = 0$
$ax^2 + bx + c \neq 0$	$a \neq 0 \vee b \neq 0 \vee c \neq 0$
$ax^2 + bx + c < 0$	$a < 0 \vee (a = 0 \wedge b > 0) \vee (a = 0 \wedge b = 0 \wedge c < 0)$
$ax^2 + bx + c > 0$	$a > 0 \vee (a = 0 \wedge b < 0) \vee (a = 0 \wedge b = 0 \wedge c > 0)$
$ax^2 + bx + c \leq 0$	$a < 0 \vee (a = 0 \wedge b > 0) \vee (a = 0 \wedge b = 0 \wedge c \leq 0)$
$ax^2 + bx + c \geq 0$	$a > 0 \vee (a = 0 \wedge b < 0) \vee (a = 0 \wedge b = 0 \wedge c \geq 0)$

Figure 8.1: The rules of substitution by test candidate $-\infty$

$p(x) \sim 0$	$(p(x) \sim 0)\ [e//x]$ for $e = \frac{q}{r}$, let k be the maximum degree of x in p and $\delta = 1$ if k is odd, otherwise $\delta = 0$
$p(x) = 0$	$p(e) \cdot r^k = 0$
$p(x) \neq 0$	$p(e) \cdot r^k \neq 0$
$p(x) < 0$	$(r^\delta > 0 \wedge p(e) \cdot r^k < 0) \vee (r^\delta < 0 \wedge p(e) \cdot r^k > 0)$
$p(x) > 0$	$(r^\delta > 0 \wedge p(e) \cdot r^k > 0) \vee (r^\delta < 0 \wedge p(e) \cdot r^k < 0)$
$p(x) \leq 0$	$(r^\delta > 0 \wedge p(e) \cdot r^k \leq 0) \vee (r^\delta < 0 \wedge p(e) \cdot r^k \geq 0)$
$p(x) \geq 0$	$(r^\delta > 0 \wedge p(e) \cdot r^k \geq 0) \vee (r^\delta < 0 \wedge p(e) \cdot r^k \leq 0)$

Figure 8.2: The rules of substitution by test candidate e $= \frac{q}{r}$

$p(x) \sim 0$	$(p(x) \sim 0)\ [e//x]$ for $e = \frac{q+r\sqrt{t}}{s}$, let k be the maximum degree of x in p and $\delta = 1$ if k is odd, otherwise $\delta = 0$
$p(x) = 0$	$\hat{q}\hat{r} \leq 0 \wedge \hat{q}^2 - \hat{r}^2 t = 0$
$p(x) \neq 0$	$\hat{q}\hat{r} > 0 \vee \hat{q}^2 - \hat{r}^2 t \neq 0$
$p(x) < 0$	$(\hat{q}\hat{s}^\delta < 0 \wedge \hat{q}^2 - \hat{r}^2 t > 0) \vee (\hat{r}\hat{s}^\delta \leq 0 \wedge \hat{q}\hat{s}^\delta < 0) \vee (\hat{r}\hat{s}^\delta \leq 0 \wedge \hat{q}^2 - \hat{r}^2 t < 0)$
$p(x) > 0$	$(\hat{q}\hat{s}^\delta > 0 \wedge \hat{q}^2 - \hat{r}^2 t > 0) \vee (\hat{r}\hat{s}^\delta \geq 0 \wedge \hat{q}\hat{s}^\delta > 0) \vee (\hat{r}\hat{s}^\delta \geq 0 \wedge \hat{q}^2 - \hat{r}^2 t < 0)$
$p(x) \leq 0$	$(\hat{q}\hat{s}^\delta \leq 0 \wedge \hat{q}^2 - \hat{r}^2 t \geq 0) \vee (\hat{r}\hat{s}^\delta \leq 0 \wedge \hat{q}^2 - \hat{r}^2 t \leq 0)$
$p(x) \geq 0$	$(\hat{q}\hat{s}^\delta \geq 0 \wedge \hat{q}^2 - \hat{r}^2 t \geq 0) \vee (\hat{r}\hat{s}^\delta \geq 0 \wedge \hat{q}^2 - \hat{r}^2 t \leq 0)$

Figure 8.3: The rules of substitution by test candidate e = $\frac{q+r\sqrt{t}}{s}$

$p(x) \sim 0$	$(p(x) \sim 0)\ [e + \epsilon//x]$
$bx + c = 0$	$\mathrm{b} = 0 \wedge \mathrm{c} = 0$
$bx + c \neq 0$	$\mathrm{b} \neq 0 \vee \mathrm{c} \neq 0$
$bx + c < 0$	$(\mathrm{bx} + \mathrm{c} < 0) \vee ((\mathrm{bx} + \mathrm{c} = 0 \wedge \mathrm{b} < 0))$
$bx + c > 0$	$(\mathrm{bx} + \mathrm{c} > 0) \vee ((\mathrm{bx} + \mathrm{c} = 0 \wedge \mathrm{b} > 0))$
$bx + c \leq 0$	$(\mathrm{bx} + \mathrm{c} < 0) \vee ((\mathrm{bx} + \mathrm{c} = 0 \wedge \mathrm{b} < 0)) \vee ((\mathrm{b} = 0 \wedge \mathrm{c} = 0))$
$bx + c \geq 0$	$(\mathrm{bx} + \mathrm{c} > 0) \vee ((\mathrm{bx} + \mathrm{c} = 0 \wedge \mathrm{b} > 0)) \vee (\mathrm{b} = 0 \wedge \mathrm{c} = 0)$
$ax^2 + bx + c = 0$	$a = 0 \wedge b = 0 \wedge c = 0$
$ax^2 + bx + c \neq 0$	$a \neq 0 \vee b \neq 0 \vee c \neq 0$
$ax^2 + bx + c < 0$	$(ax^2+bx+c < 0) \vee (ax^2+bx+c = 0 \wedge 2ax+b < 0) \vee (ax^2+bx+c = 0 \wedge 2ax+b = 0 \wedge 2a < 0)$
$ax^2 + bx + c > 0$	$(ax^2+bx+c > 0) \vee (ax^2+bx+c = 0 \wedge 2ax+b > 0) \vee (ax^2+bx+c = 0 \wedge 2ax+b = 0 \wedge 2a > 0)$
$ax^2 + bx + c \leq 0$	$(ax^2 + bx + c < 0) \vee ((ax^2 + bx + c = 0 \wedge 2ax + b < 0)) \vee ((ax^2 + bx + c = 0 \wedge 2ax + b = 0 \wedge 2a < 0)) \vee (a = 0 \wedge b = 0 \wedge c = 0)$
$ax^2 + bx + c \geq 0$	$(ax^2 + bx + c > 0) \vee ((ax^2 + bx + c = 0 \wedge 2ax + b > 0)) \vee ((ax^2 + bx + c = 0 \wedge 2ax + b = 0 \wedge 2a > 0)) \vee (a = 0 \wedge b = 0 \wedge c = 0)$

Figure 8.4: The rules of substitution by test candidate e + ϵ

For this case we also calculate the first and second derivation of the polynomial in order to interprate these rules.

Bibliography

[ÁC12] Erika Ábrahám and Florian Corzilius. Non-linear real arithmetic: Virtual substitution. pages 23–24, 2012.

[ÁK16] Erika Ábrahám and Gereon Kremer. Satisfiability checking: Theory and applications. In *International Conference on Software Engineering and Formal Methods*, pages 9–23. Springer, 2016.

[Akr80] Alkiviadis G Akritas. The fastest exact algorithms for the isolation of the real roots of a polynomial equation. *Computing*, 24(4):299–313, 1980.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[Cha93] Vijay Chandru. Variable elimination in linear constraints. *The Computer Journal*, 36(5):463–472, 1993.

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In *International SPIN Workshop on Model Checking of Software*, pages 248–254. Springer, 2012.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[HLL90] Tien Huynh, Catherine Lassez, and Jean-Louis Lassez. Fourier algorithm revisited. In *International Conference on Algebraic and Logic Programming*, pages 117–131. Springer, 1990.

[Koš16] Marek Košta. PhD Thesis: New concepts for real quantifier elimination by virtual substitution. 2016.

[Tov84] Craig A Tovey. A simplified NP-complete satisfiability problem. *Discrete applied mathematics*, 8(1):85–89, 1984.