Integrating Virtual Substitution into Strategic SMT Solving

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Florian Corzilius

aus Geldrop (Niederlande)

Berichter	Prof. Dr. Erika Ábrahám
	Ass.-Prof. Pascal Fontaine, PhD
Tag der mündlichen Prüfung	21.10.2016

Abstract

This thesis addresses the integration of *real algebraic procedures* as *theory solvers* into *satisfiability modulo theories (SMT) solvers*, in order to check *nonlinear real- and integer-arithmetic* formulas for *satisfiability*. There are plenty of procedures to choose from and we aim for a general framework that allows us to select and combine them. The main part of this thesis concerns one specific example of the aforementioned integration: the *virtual substitution method*. Here we also optimize this method with respect to satisfiability checking and extend it such that it can be used for integer arithmetic.
We present the results of this thesis in the following order:

1. The design, functionality and features of the toolbox `SMT-RAT`, which implements a framework for a *strategical combination of procedures* for, e. g., real and integer arithmetic. This toolbox has been developed as part of this thesis and can also be used to assemble an SMT solver.

2. An *SMT compliant* theory solver based on the virtual substitution, that works *incrementally* and supports *backtracking* and *infeasible subset generation*.

3. Heuristics for strategic choices during a satisfiability check of this theory solver and optimizations, which exploit *local conflicts* and *bounds on the variable's domains*.

4. An evaluation of the SMT compliant features, the heuristics and the optimizations by integrating this theory solver into an SMT solver, which we assembled with `SMT-RAT`. We also combine this theory solver with other procedures and compare with state-of-the-art solvers.

5. For the purpose of checking nonlinear integer-arithmetic formulas for satisfiability, we present and evaluate an embedding of the virtual substitution into a *branch-and-bound framework*, which we specifically tailored for the case where a combination of procedures is used.

6. An optimization of the computation of the rational functions, which represent the *probability of reaching certain states* of *parametric discrete-time Markov chains*. Here we make use of a synergy of the *greatest common divisor* calculation of two polynomials, *polynomial factorization* and intermediate result caching.

Zusammenfassung

Diese Doktorarbeit behandelt die Integrierung *reell-algebraischer Prozeduren* als *Theory-Solver* in *Satisfiability-Modulo-Theories-Solver* (SMT-Solver), um *nichtlineare reell- und ganzzahlig-arithmetische Formeln* auf *Erfüllbarkeit* zu überprüfen. Es gibt viele Prozeduren, welche sich hierfür anbieten, und wir streben ein generelles Framework an, mit dem man diese auswählen und kombinieren kann. Der Hauptteil dieser Doktorarbeit beschäftigt sich mit einem spezifischen Beispiel für die obengenannte Integrierung: der *Virtual-Substitution-Methode*. Hierbei wird diese Methode auch bezüglich der Erfüllbarkeitsüberprüfung optimiert und auf die Anwendbarkeit auf ganzzahlige Arithmetik erweitert.
Wir stellen die Resultate dieser Doktorarbeit in der folgenden Reihenfolge vor:

1. Das Design, die Funktionsweise und die Bestandteile der Toolbox `SMT-RAT`, welche ein Framework für die *strategische Kombination von Prozeduren* für, zum Beispiel, reelle und ganzzahlige Arithmetik implementiert. Diese Toolbox wurde als Teil dieser Doktorarbeit entwickelt und kann auch dafür benutzt werden einen SMT-Solver zusammenzustellen.

2. Einen *SMT-konformen* Theory-Solver basierend auf der Virtual-Substitution-Methode, was bedeutet, dass er *inkrementell* arbeitet und *Backtracking* sowie das *Generieren unerfüllbarer Teilmengen* unterstützt.

3. Heuristiken für die vielen Wahlmöglichkeiten während der Erfüllbarkeitsüberprüfung dieses Theory-Solvers und Optimierungen, welche *lokale Konflikte* und *Schranken auf den Variablendomänen* ausnutzen.

4. Eine Auswertung der SMT-konformen Eigenschaften, der Heuristiken sowie Optimierungen mittels einer Integrierung dieses Theory-Solvers in einen SMT-Solver, welchen wir mit `SMT-RAT` zusammengestellt haben.

5. Zwecks der Erfüllbarkeitsüberprüfung nichtlinearer ganzzahlig-arithmetischer Formeln, präsentieren und evaluieren wir eine Einbettung der Virtual-Substitution-Methode in ein *Branch-and-Bound-Framework*, das wir spezifisch auf den Fall der Kombination von Prozeduren zugeschnitten haben.

6. Eine Optimierung der Berechnung rationaler Funktionen, welche die *Wahrscheinlichkeit repräsentieren bestimmte Zustände parametrischer Markow-Ketten mit diskreter Zeit zu erreichen*. Hierbei verwenden wir ein Zusammenspiel der Berechnung des größten gemeinsamen Teilers zweier Polynome, von Polynomfaktorisierungen und von Zwischenergebnis-Caching.

Acknowledgements

Reminiscing recent years, I am a little bit overwhelmed. I certainly never took it for granted that I got the chance to be a PhD-student and work on a topic I absolutely adored. I lived my own childhood dream by being a scientist, meeting so many fantastic people with whom I share invaluable experiences and visiting spectacular places all over the world.

I am so thankful that Erika Ábrahám gave me this chance and I thank her particularly for all the patience, guidance and marvelous discussions. I am also deeply grateful that I was part of her group. It was a great joy and really inspiring to work with Xin Chen, Nils Jansen, Gereon Kremer, Ulrich Loup, Johanna Nellen and Stefan Schupp. We had a lot of fun together and some unforgettable evenings.

I also want to thank all those I had the pleasure to collaborate with over previous years: Erika Ábrahám, Bernd Becker, Nils Jansen, Sebastian Junges, Joost-Pieter Katoen, Gereon Kremer, Ulrich Loup, Karsten Scheibler, Stefan Schupp, Matthias Volk and Ralf Wimmer.

Furthermore, special thanks go to Pascal Fontaine and Thomas Sturm for the many inspiring discussions in our meetings.

Being a PhD-student involves lots of traveling, engaging projects and deadlines, but it unfortunately comes at the cost of missing out on time with the ones you love. I am very thankful that my mother, my father and my sisters, nonetheless, always supported me and for all the faith they had in me. This holds none more so than for my wife, Becky, who was patient, if plans had to be altered, who pushed me, if my motivation faded, and who revised every single line of this thesis.

Contents

CHAPTER 1

Introduction

Thinking back to mathematics lessons at school, most students started struggling when it came to solving a word problem; that is a mathematical exercise presented in text form. No doubt the first obstacle for this task is the translation of text to mathematical notation or its geometrical interpretation. Then we need to recognize patterns and identify techniques which might help to find a solution to the posed question. In the majority of cases these questions ask for one solution or maybe an optimal solution. School prepares young people using these exercises with the assumption that they will encounter many such problems, if not in daily life but in work life, especially if it concerns a rather complex topic.

As a university student, in particular for natural sciences, one has to deal with proving mathematical statements. Although we do not aim at a single solution as for word problems, we also need to recognize patterns and identify techniques, but this time in order to find a finite series of implications which either prove or disprove the statement.

About 1920, the mathematician David Hilbert started a project seeking

- a precise formal language in order to specify, for instance, word problems or statements that we want to prove, and
- a finite set of axioms, which are provably consistent and can be applied (in a finite sequence) in order to prove any statement or solve any word problem, which could be expressed in the formal language.

In 1931, Kurt Gödel disabused the people who believed that this project could ever accomplish its goal. He proved in his incompleteness theorems [Göd31] that, in general, there is no formal language and set of axioms, which fulfill these needs.

Nevertheless, the foundations of *computability theory* were laid, which aims to classify problems according to their decidability and (if decidable) according to the complexity of solving them.

It is undeniable how crucial it is to know about a problem's decidability and complexity before trying to solve it. For instance, let us have a look at the tenth problem of the famous 23 problems published by Hilbert himself in 1900 [Hil02]:

> *"Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers."*

Without loss of generality, we are looking for integer values which we assign to the variables $x_1, .., x_n$ in a polynomial $p(x_1, .., x_n)$ such that it evaluates to zero, i. e., for an *integer solution* of the equation $p(x_1, .., x_n) = 0$. In 1970, Martin Davis, Yuri Matiyasevich, Hilary Putnam and Julia Robinson showed that there is no *decision procedure* that always determines (in a finite number of operations) whether an integer solution for a given equation exists [Mat70][Mat72].

If we assume that p is *linear*, that is we can bring it to the form $a_1x_1 + .. + a_nx_n$, where $a_1, .., a_n$ are integers, the problem becomes decidable. Even if we consider a system of equations and inequalities (where the relation symbol is $<$, $\leq$, $>$ or $\geq$) over linear polynomials, which means that all of these equations and inequalities have to hold at the same time, the problem is decidable. However, the number of operations, which have to be done in order to determine whether an integer solution exists, grows exponentially as the number of variables (n) increases. If we ask for the existence of a real solution instead, that is we relax the requirement that the solution must be integer, this number of operations grows only polynomially as n increases. Even if we require this solution to be optimal (with respect to a linear objective function), it still grows polynomially.

Let us go back to our initial setting of word problem solving and theorem proving. In both cases, it is imperative that we are able to logically connect sub-problems or sub-statements. In a language we would usually use, e. g., the words "and", "or", or "implies", which logically connect two statements, or the word "not" in order to negate a given statement. In *mathematical logic* there are equivalent *Boolean operations*. Regarding the just listed logical connectives, we usually use the symbols $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies) and $\neg$ (not). If we use these *Boolean operators* in order to connect *propositions*, which can be either true or false, we obtain a *propositional formula*. For example, consider the following statements, where we have attached the translation to a propositional formula:

1. If $\overbrace{\text{Mario is hungry}}^{(b_1}$ $\overbrace{\text{and}}^{\wedge}$ $\overbrace{\text{currently too lazy to cook}}^{b_2)}$, $\overbrace{\text{then}}^{\rightarrow}$ $\overbrace{\text{he puts a frozen pizza in the oven.}}^{b_3}$
2. $\overbrace{\text{Mario is hungry.}}^{b_1}$
3. $\overbrace{\text{Mario is currently too lazy to cook.}}^{b_2}$

From these three statements, it follows that Mario puts a frozen pizza in the oven, that is b_3 must

be true. Therefore, we obtain a solution to the above statements, if we assign the value true to the variables b_1, b_2 and b_3.

In general, it is not easy to find solutions for propositional formulas. As we can assign either true or false to each Boolean variable, there are clearly finitely many candidates which come into question. Hence, we are dealing with a decidable problem. However, these candidates count 2^n, if the number of variables in the formula is n. Unfortunately, this is also the worst case complexity of any algorithm which can check whether a solution exists for a given propositional formula if P≠NP [Coo71].

Nonetheless, scientists kept on designing ever improving algorithms exploiting the inherent structure of propositional formulas, for instance of formulas which have their origins in a scientific or industrial application. A breakthrough was achieved by Martin Davis, Hilary Putnam, George Logemann and Donald W. Loveland in 1962 [DLL62], which ultimately led to today's well-known *SAT solvers* [MSS99][MMZ+01]. They are designated for checking a propositional formula for *satisfiability*, i. e., for finding out whether the formula has a solution (and is *satisfiable*) or not (and is *unsatisfiable*). SAT solvers apply a combination of an inference mechanism, such as we have applied in the previous example, and *conflict driven learning*. Nowadays, these solvers are able to check propositional formulas with up to ten million variables for *satisfiability* in only a few minutes or often even a few seconds [JBRS12].

The success story of SAT solvers shows that despite the high complexity, which must be accepted in the worst case when solving such problems, we can achieve a remarkable performance with sophisticated algorithms equipped with a learning ability. In the last decade, scientists have started an attempt to utilize SAT solvers in order to check formulas for satisfiability, which do not only combine propositions with Boolean operators but also *constraints*, e. g., the aforementioned equations and inequalities comparing linear or nonlinear polynomials in real or integer valued variables. We refer to these formulas as *satisfiability modulo theories* (SMT) formulas and present an example in order to show their applicability.

Consider the problem in Figure 1.1. We want to know whether we can position three pizzas of different sizes on a baking tray such that they do not overlap. We can encode this problem to an SMT formula, which contains the real-valued variables

- r_1, r_2 and r_3 denoting the radii of the three pizzas we choose to put onto the baking tray and
- x_1, y_1, x_2, y_2, x_3 and y_3 denoting the coordinates of the two-dimensional center points of these pizzas.

Without loss of generality, we can assume that the baking tray's bottom left corner is at $(0, 0)$. Then we can encode this problem by the SMT formula

$$\varphi_{\text{pizza}} \quad = \quad \varphi_{\text{choose}} \quad \wedge \quad \varphi_{\text{on-baking-tray}} \quad \wedge \quad \varphi_{\text{no-overlapping}}$$

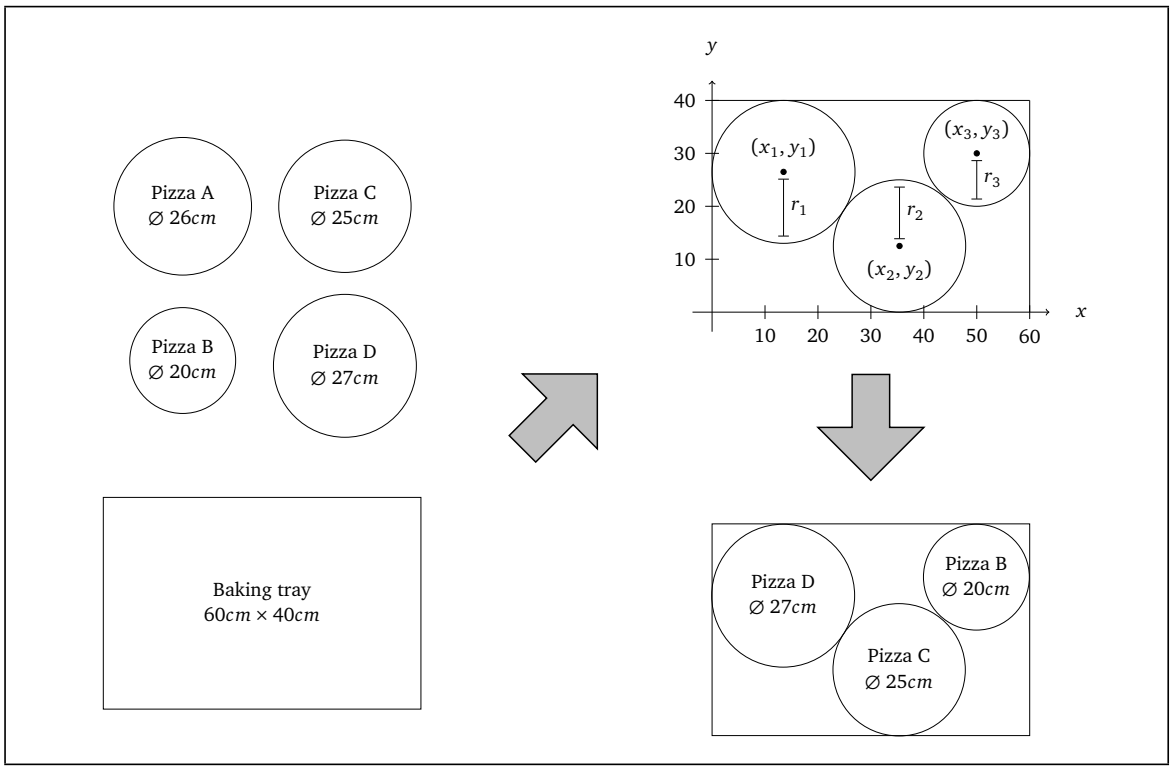


Figure 1.1: The pizza problem: Can you fit three pizzas of different sizes on a baking tray?

with

$$\varphi_{\text{choose}} = \bigwedge_{i=1}^{3} (r_i = 13 \vee r_i = 10 \vee r_i = \tfrac{25}{2} \vee r_i = \tfrac{27}{2}) \quad \wedge \quad \bigwedge_{i=1}^{2} \bigwedge_{j=i+1}^{3} \neg(r_i = r_j)$$

$$\varphi_{\text{on-baking-tray}} = \bigwedge_{i=1}^{3} (x_i + r_i \leq 60 \quad \wedge \quad x_i - r_i \geq 0 \quad \wedge \quad y_i + r_i \leq 40 \quad \wedge \quad y_i - r_i \geq 0)$$

$$\varphi_{\text{no-overlapping}} = \bigwedge_{i=1}^{2} \bigwedge_{j=i+1}^{3} (x_i - x_j)^2 + (y_i - y_j)^2 \geq (r_i + r_j)^2$$

The formula φ_{choose} ensures that we can only choose three pairwise different pizzas, where $\varphi_{\text{on-baking-tray}}$ specifies that the three chosen pizzas are all within the margins of the baking tray. The formula $\varphi_{\text{no-overlapping}}$ defines, by the use of the Euclidean distance, that these pizzas do not overlap. Therefore, the three constraints in $\varphi_{\text{no-overlapping}}$ are nonlinear.

Asking for the satisfiability of this *nonlinear real-arithmetic* (NRA) formula, has been proved to be decidable by Alfred Tarski in 1948 [Tar48]. A complete procedure for NRA, that is a procedure which, given an NRA formula, can always determine whether it is satisfiable or not, is the *cylindrical algebraic decomposition* (CAD) method introduced by George E. Collins in 1975 [Col75]. The worst case complexity of this procedure grows exponentially as the number of variables in the formulas grows [DH88]. Indeed, the CAD method tends not to perform very well on many examples. Fortunately, there are further incomplete procedures for NRA, which

often perform better than the CAD method. We can use *Gröbner bases* in order to exclude that there is a common complex solution for a set of equations, ruling out the existence of real-valued solutions [BWK93]. *Interval constraint propagation* (ICP) narrows down an initially given interval over-approximation of possible solutions and thereby might detect that no solution exists [FHT+07][GGI+10]. However, ICP can only guarantee to find or rule out solutions within a certain precision. The *virtual substitution* (VS) method, on the other hand, is limited in the degree of the variables in a given formula [Wei97]. Nonetheless, it is applicable to a wide range of problems.

The CAD and VS methods are originally designed as *quantifier elimination procedures* for NRA formulas, where some variables are existentially or universally quantified (i. e., $\exists x.\varphi \triangleq$ there exists a value for x such that φ holds or $\forall x.\varphi \triangleq$ for all values for x φ holds). Our setting, where we want to check a formula for satisfiability, is equivalent to determining the formula's truth value (true or false) if we assume that all of its variables are existentially quantified (in the form $\exists x_1..\exists x_n.\varphi$, where φ is quantifier-free and contains only the variables $x_1,..,x_n$). Therefore, we can directly use the VS and CAD methods, if we let them eliminate all (quantified) variables of this formula. It also works, if the formula is not *conjunctive*. For instance, in the previous example, $\varphi_{\text{on-baking-tray}}$ and $\varphi_{\text{no-overlapping}}$ are conjunctive, which means that all constraints have to hold. In contrast to this, φ_{choose} encodes a combinational problem. It is enough to fulfill at least one equation in each of the formula's disjunctions in order to make it satisfiable and there are $4 \cdot 3 \cdot 2 = 24$ possible ways to achieve this. With both the CAD and the VS method we have to deal with all constraints in the formula at once, even though it would be enough to only consider some of them as it is the case for φ_{choose}.

SMT solving has the intention that a SAT solver chooses some constraints of a given formula, which assure that the Boolean structure of the formula is satisfied as long as they are satisfiable. Therefore, we need to check a conjunction of constraints for satisfiability, which is achieved with a *theory solver*. For NRA formulas, this theory solver could implement, for instance, the CAD or the VS method and only has to deal with conjunctions of some of the constraints of a given formula. There are already tools which provide implementations based on the CAD and the VS method. These tools, which are often referred to as *computer algebra systems*, have their origins in a field of research, which addresses, i. a., NRA formulas for a much longer time than SMT solving: *symbolic computation*. It stands to reason that we can plug in these tools as theory solvers in an SMT solver. However, for an optimal collaboration with the SMT solver's SAT solver, a theory solver needs to fulfill certain requirements.

1. The theory solver only aims to check a conjunction of constraints for satisfiability. Instead of using, e. g., a quantifier elimination procedure, which can handle more general formulas, the theory solver should be optimized for this special purpose.

2. During SMT solving we invoke the theory solver frequently, each time asking for the satisfiability of a conjunction of constraints. Often, this conjunction is only slightly changed

between two consecutive theory solver invocations. Therefore, the theory solver optimally keeps as much information from previous invocations as possible in order to speed up the performance of the current one.

3. In case the theory solver detects that a conjunction of constraints is unsatisfiable, the SMT solver's SAT solver excludes this combination from its search. If the theory solver also provides a subset of the constraints, which is still unsatisfiable, the SAT solver can exclude all combinations, which contain these constraints instead. This can dramatically reduce the number of theory solver invocations that are needed during the SMT solving process.

We have listed some of the existing methods, which can be utilized for checking an NRA formula and in particular a conjunction of NRA constraints for satisfiability. However, it highly depends on the formula, which of these methods performs best. Due to the high worst-case complexity it often means that one method finds out the formula's satisfiability within seconds or minutes where the other method does not yield a result within hours, days or even much longer. Optimally, we would be able to choose from a set of implementations for each of these methods, but it remains the problem of which method would be the best choice for a given formula. Moreover, each of the aforementioned methods has its own heuristics and choices of sub-procedures, which raises opportunities for a performance tuning. Finally, we can also combine these methods as it has been suggested in both symbolic computation [DSW98] as well as SMT solving [dMP13].

SMT solving has already been successfully applied to academic and industrial problems [BKM14]. For instance, it made a contribution towards the detection of design errors in the logical functioning of modern digital electronic chips. It has also been useful within the context of safety-critical embedded software. Furthermore, SMT solvers have been employed in order to prove correctness or detect bugs of programs, using the programmer's invariants/assumptions or simulating a parametrized run violating some properties. With similar techniques, we can also find vulnerabilities for a security attack. For all of these applications it would be of great use to be able to also pose satisfiability checks involving NRA formulas. In contrast to the already widely used logics for SMT solving, which achieved the successes just described, we have a greater choice of methods for NRA and, more importantly, they are fairly difficult to understand and implement. Fortunately, plenty of expertise exists within the field of research of symbolic computation. A recent initiative in the context of the H2020-FETOPEN-CSA project SC^2 aims to bring together scientists from symbolic computation and SMT solving, with the goal that they become aware of each other's achievements and are able to unify their strengths in order to tackle practical problems [ÁAB+16].

1.1 Contributions and structure of this thesis

In this chapter, we have already provided an insight into SMT solving of NRA formulas and the challenges we have to deal with when designing and implementing theory solvers based on the

plenty and mostly rather complex methods we could use for this purpose. In Chapter 2, we specify the syntax and semantics of the formulas we deal with and introduce the most important procedures on which the contributions of this paper rely on: SAT solving, SMT solving and the virtual substitution method. This lays the foundations for the contributions of this thesis, which we explain in the chapters 4 - 8 in detail and are summarized in the following paragraphs. We conclude this thesis in Chapter 9.

A toolbox for strategic and parallel SMT solving As mentioned before, there are many methods which can be applied for the satisfiability check of an NRA formula. It is in general unclear which method performs best for a given formula and the right choice can influence whether we are able to determine the satisfiability within seconds/minutes or hours/days. Moreover, we want to be able to easily combine these methods.

Under these circumstances we had to rethink the common approach for SMT solving and came up with a novel *framework*, which defines a *common interface* for implementations of procedures that contribute to a formula's satisfiability check. In the first place, these so called *modules* were intended to provide NRA procedures in the form of theory solvers that are *SMT compliant*, that is they comply with the aforementioned requirements for a performant integration into an SMT solver. However, it turned out to be an interface which is general enough to bear the entire interaction within an SMT solver. This extremely modular approach aims specifically at the possibility of combining modules according to a *user-defined strategy*, which specifies which module is used to solve a formula for satisfiability based on the solving history and the formula's properties. In addition, we allow modules to *run in parallel* which makes it possible to overcome the uncertainty about which method performs best. In Chapter 3, we present our open-source `C++` toolbox `SMT-RAT`, which implements this framework and provides a set of modules. It *can easily be extended* by further modules and provides an *intuitive user-interface* for the creation of solving strategies. Such a strategy can yield, e. g., a theory solver, which can be embedded into a state-of-the-art SMT solver. In addition, the provided modules allow us to specify a strategy that serves as an SMT solver. Using such a strategy, `SMT-RAT` already took part in the international annual competition between SMT solvers.

The work on `SMT-RAT` led to two publications. In [6] we contributed

- the novel design of a framework which allows us to combine modules representing theory solvers according to a user-defined strategy,
- an SMT-compliant module implementing the virtual substitution
- and modules, which implement Gröbner bases (only for equations) and the cylindrical algebraic decomposition (only for univariate polynomials).

We carried on the development of `SMT-RAT`, yielding the contributions which we published in [2] and are summarized as follows:

- We designed a more powerful strategy class for the combination of modules, which in particular provides the possibility of running modules in parallel. We added a better user interface for the creation of such a strategy.
- We added a module, which incorporates a SAT solver and extends `SMT-RAT` so that a user can define a strategy that constitutes an SMT solver.
- We provided SMT-compliant modules which implement a preprocessing, the *simplex method* as proposed in [DdM06] and interval constraint propagation similar to [GGI+10].
- We further optimized the existing module, which implements the VS, and extended the modules implementing Gröbner bases (now: SMT compliant and usable for formula simplification) and the CAD (now: SMT compliant and applicable on arbitrary conjunctions of NRA constraints).
- We extended the shared interface of the modules allowing them to exchange more information. It also provides the new feature of calling a module's satisfiability check such that it avoids hard obstacles during solving at the price of possibly not finding any conclusive answer.

Virtual substitution in SMT The virtual substitution has already shown its applicability thanks to the `Redlog`-package [DS97] of the computer algebra system `Reduce`. It performs quantifier elimination on an NRA formula, which allows us, in particular, to check a conjunction of NRA constraints for satisfiability due to the aforementioned reasons. Hence, it is a promising candidate for an integration into a theory solver of an SMT solver which aims to solve NRA formulas for satisfiability. As we also pointed out, we optimally require this theory solver to be SMT compliant, which was not yet the case for `Redlog`'s implementation of the virtual substitution in 2011.

In Chapter 4 we push forward the ideas of my diploma thesis [Cor10], offering solutions for some questions, which were left open, and generalizing the theorems, which also yields more concise proofs. The presented contributions were published in [7] and can be summarized as follows:

- We define an algorithm that uses the virtual substitution in order to perform a satisfiability check of a conjunction of NRA constraints. For this more specific scenario than the one which `Redlog` is dedicated for, we implement a depth-first search in order to find a solution.
- As we keep track of the origins of intermediate results and store intermediate conflicts during this depth-first search, we are able to belatedly add and remove constraints to the conjunction of constraints we check for satisfiability.
- Moreover, we present and prove a theorem, from which it directly follows how to construct infeasible subsets of these constraints, in case they have no common solution.

Improving the performance of the virtual substitution in SMT The virtual substitution can only be applied if the degree of at least one variable is less than three in one constraint. Assuming that we are able to check a conjunction of NRA constraints for satisfiability with the VS, the worst-case complexity grows exponentially as the number of variables in this conjunction grows. There are many choices that have to be made within the algorithm we define in Chapter 4 and they are crucial to avoid this worst case. Where one choice might yield a solution in the underlying depth-first search, the other choice can provoke a far worse performance or even the case where the VS cannot be applied, that is all variables have a degree that is greater than two in all constraints of interest.

We concentrate on the satisfiability check of a conjunction of NRA constraints in the algorithm, which we present in Chapter 4. We also keep track of the origins of intermediate results and store intermediate conflicts. This provides a lot of potential in order to exploit local information during the satisfiability check aiming at an improvement of its performance. Local conflicts, for instance, might help us to prune the search space of the presented depth-first search.

In Chapter 5 we present

- our ideas for the heuristics we use for the choices of concern and experimentally evaluate different approaches for this purpose,
- a principle to guide the construction of local conflicts so that we can prune a larger part of the search space when resolving them,
- a method to detect local conflicts before taking all possibilities into account and
- a specialization of the main theorem in [Wei97] that specifies how to eliminate a quantifier in the virtual substitution. This specialization exploits the constraints, which are direct sub-formulas of a conjunction and represent an upper or lower bound for a variable, in order to narrow down the candidates to consider during a satisfiability check with the VS. Furthermore, it simplifies intermediate results.

Virtual substitution for integer arithmetic Up to this point, we have considered NRA formulas for a satisfiability check. If we further restrict ourselves to find integer instead of arbitrary real solutions for the variables, we have already seen that the question of whether a given formula is satisfiable is not decidable in general. If the formula is linear or if all variables are lower and upper bounded, this problem becomes satisfiable. There are two approaches which take advantage of this. The first approach adds lower and upper bounds, then encodes the problem's integer domains and mathematical operations upon them to a propositional formula and checks it for satisfiability. If a solution has been detected, we can construct an integer solution. Otherwise, we widen the lower and upper bounds and try again. The second approach applies interval constraint propagation. Here, we can also add progressively widened lower and upper bounds in order to increase the likeliness of a termination.

For linear formulas, the approach used the most in practice is *branch-and-bound*. Firstly, it searches for a real solution by the use of, e. g., the simplex method. If no real solution can be found, then there is no integer-valued solution. Otherwise, if the found solution provides a non-integer value for any variable, which is supposed to be assigned only integers, we make a case distinction (*branch*). Firstly, we check whether we can find an integer solution by recursively invoking branch-and-bound, but with the variable in question to be *upper-bounded* by the next smaller integer than the found non-integer real solution. If this does not provide an integer solution, we analogously check if we can find one, but this time with the variable being *lower-bounded* by the next greater integer than the found non-integer real solution.

This approach cannot directly be applied to a procedure, which checks the satisfiability of NRA formulas, such as the VS and CAD method. The VS method, for instance, provides parametrized solutions, meaning that we need to construct the numeric solution of some variables in order to be able to construct the numeric solution of another variable. We already mentioned that we can make use of several procedures for NRA. It is crucial for the practicability of an SMT solver for NRA to take advantage of this choice and if we want to use branch-and-bound on top of NRA procedures, this argument still holds.

In Chapter 7, we elaborate on the following contributions from [1]:

- We present a general framework for branch-and-bound in SMT solving providing the opportunity to
  1. demand a branching, which is shared among all procedures involved in the SMT solving process and affects the global reasoning process,
  2. while keeping this branching locally bound to the constraints that are co-responsible for the branching.
- We formalize how to detect the variable and value, which are necessary for an application of the aforementioned branch-and-bound methodology, when using it on top of the virtual substitution method as NRA procedure. Moreover, we specify how to construct the constraints, which are co-responsible for the branching, in this case.

A synergy of the greatest common divisor calculation, factorization and intermediate result caching Algebraic operations on polynomials form the foundations of procedures, such as the virtual substitution and the cylindrical algebraic decomposition method. They also occur in other settings, for instance, when calculating the reachability probabilities of *parametric discrete-time Markov chains* (PDTMCs) [Daw04]. During this calculation we constantly simplify the intermediate results, which are rational functions, i. e., a fraction of two polynomials. This simplification involves the rather expensive *greatest common divisor computation* (gcd) of two polynomials and, indeed, experimental results show that this operation forms one of the bottlenecks when it comes to the calculation of the reachability probabilities of PDTMCs.

In Chapter 8, we present the ideas which were published as part of [3]. It aims to speed up the gcd calculation of polynomials and thereby the simplification of rational functions by

- caching a *(partial) factorization* of occurring polynomials,
- maintaining this factorization during the usual operations on polynomials (+, ·, etc.) and
- exploiting this additional information in an algorithm that calculates the gcd of two polynomials. Moreover, this algorithm refines the already cached polynomial factorizations as a side effect.

1.2 Relevant publications

The aforementioned contributions yielded 7 peer-reviewed publications, which are listed in Section 1.2.1. We also published a technical report, which is listed separately in Section 1.2.2. Further publications, in which the author has been involved, are listed in Section 1.3.

1.2.1 Peer-reviewed publications

[1] Gereon Kremer, Florian Corzilius, and Erika Ábrahám. A generalised branch-and-bound approach and its application in SAT modulo nonlinear integer arithmetic. In *Proc. of CASC*, volume 9890 of *LNCS*, pages 315–335. Springer, 2016.

[2] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT: An open source C++ toolbox for strategic and parallel SMT solving. In *Proc. of SAT*, volume 9340 of *LNCS*, pages 360–368. Springer, 2015.

[3] Nils Jansen, Florian Corzilius, Matthias Volk, Ralf Wimmer, Erika Ábrahám, Joost-Pieter Katoen, and Bernd Becker. Accelerating parametric probabilistic verification. In *Proc. of QEST*, volume 8657 of *LNCS*, pages 404–420. Springer, 2014.

[4] Sebastian Junges, Ulrich Loup, Florian Corzilius, and Erika Ábrahám. On Gröbner bases in the context of satisfiability-modulo-theories solving over the real numbers. In *Proc. of CAI*, volume 8080 of *LNCS*, pages 186–198. Springer, 2013.

[5] Ulrich Loup, Karsten Scheibler, Florian Corzilius, Erika Ábrahám, and Bernd Becker. A symbiosis of interval constraint propagation and cylindrical algebraic decomposition. In *Proc. of CADE*, volume 7898 of *LNCS*, pages 193–207. Springer, 2013.

[6] Florian Corzilius, Ulrich Loup, Sebastian Junges, and Erika Ábrahám. SMT-RAT: An SMT-compliant nonlinear real arithmetic toolbox. In *Proc. of SAT*, volume 7317 of *LNCS*, pages 442–448. Springer, 2012.

[7] Florian Corzilius and Erika Ábrahám. Virtual substitution for SMT-solving. In *Proc. of FCT*, volume 6914 of *LNCS*, pages 360–371, 2011.

1.2.2 Technical reports

[8] Nils Jansen, Florian Corzilius, Matthias Volk, Ralf Wimmer, Erika Ábrahám, Joost-Pieter Katoen, and Bernd Becker. Accelerating parametric probabilistic verification. *CoRR*, abs/1312.3979, 2013.

[9] Sebastian Junges, Ulrich Loup, Florian Corzilius, and Erika Ábrahám. On Gröbner bases in the context of satisfiability-modulo-theories solving over the real numbers. *Technical report at RWTH Aachen University*, AIB-2013-08, 2013.

1.2.3 Contributions made by the author

For all publications, which we listed in Section 1.2.1, the author contributed to the discussions, to the development of the ideas, to their formulation and in most cases to their implementation. Most publications have been achieved in collaboration with many scientists, thus we emphasize the specific contributions made by the author of this thesis.

The main focus of this thesis is clearly the virtual substitution in the context of SMT solving. The ideas, which contribute to an SMT-compliant theory solver based on the virtual substitution have been elaborated by the author in fruitful discussions with his supervisor Erika Ábrahám. The results were published in [7] and led to a fully operative SMT compliant theory solver for non-linear real arithmetic, which has been entirely implemented by the author.

Parallel to this, Ulrich Loup worked on an SMT compliant theory solver based on the cylindrical algebraic decomposition. Moreover, Sebastian Junges, who was a research student at the time, worked on an SMT-compliant theory solver using Gröbner bases. One of the main goals was also to *combine* these theory solvers, which yielded the ideas of the toolbox `SMT-RAT` in 2012. The design of the first version of `SMT-RAT` was accomplished in continuous discussions between the author, Ulrich Loup, Sebastian Junges and Erika Ábrahám. The toolbox was implemented for the greater part by the author. This work was published in [6].

Together with Karsten Scheibler and Bernd Becker from the University of Freiburg, Ulrich Loup, Erika Ábrahám and the author developed an idea for a combination of interval constraint propagation with the cylindrical algebraic decomposition, which was published in [5]. The corresponding implementation and realization of the publication was mostly carried out by Ulrich Loup and Karsten Scheibler. The author transferred these ideas to the virtual substitution, which is presented in Section 5.4.

The work of Sebastian Junges on an SMT compliant theory solver using Gröbner bases yielded the contributions, which were published in [4]. The results were mainly achieved by himself and Ulrich Loup. Together with Erika Ábrahám, the author contributed to the discussions on this topic. Moreover, he provided the implementation of further features in `SMT-RAT`.

The continuation of the work in the context of Matthias Volk's Bachelor thesis, which has been co-supervised by Nils Jansen and the author, resulted in the contributions, which were published in [3]. Many discussions about this topic, as well as the writing process, have been

supported by Ralf Wimmer, Erika Ábrahám, Joost-Pieter Katoen and Bernd Becker. The author mainly contributed the ideas for an optimization of the greatest common divisor calculation of polynomials and implemented these ideas together with Matthias Volk.

After its first publication in 2012, the toolbox `SMT-RAT` has always been under further development. The author co-supervised the Diploma thesis of Henrik Schmitz, which led, with the aid of discussions with Sebastian Junges and Erika Ábrahám, to the development of a new strategy, which also allows sub-strategies to run in parallel. The implementation was undertaken mostly by Henrik Schmitz. Moreover, the author co-supervised Stefan Schupp's master thesis about the integration of interval constraint propagation into `SMT-RAT`, where the implementation was accomplished by Stefan Schupp. Additionally, the author implemented a module in `SMT-RAT`, which enables it to be used as an SMT solver, and a module, which implements an SMT-compliant theory solver based on the simplex method as presented in [DdM06]. All these new developments along with a new implementation of a theory solver based on the cylindrical algebraic decomposition implemented by Gereon Kremer, were published in [2].

Together with Gereon Kremer and Erika Ábrahám, the author developed an integration of the virtual substitution and cylindrical algebraic decomposition into a branch-and-bound framework such that we can use these procedures for integer arithmetic. The results of this work were published in [1]. Here, the author specifically contributed the integration of the virtual substitution, which is also presented in Section 7.1. The other parts of Chapter 7 are strongly oriented towards [1], which was contributed, for the greater part, by Gereon Kremer.

1.3 Further publications

[10] Erika Ábrahám, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer, and Jacopo Mauro. Zephyrus2: On the fly deployment optimization using SMT and CP technologies. In *Proc. of SETTA*, pages 229–245. Springer, 2016.

[11] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Joost-Pieter Katoen, Erika Ábrahám, and Harold Bruintjes. Parameter synthesis for probabilistic systems. In *MBMV'16*, pages 72–74. Albert-Ludwigs-Universität Freiburg, 2016.

[12] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Harold Bruintjes, Joost-Pieter Katoen, and Erika Ábrahám. PROPhESY: A PRObabilistic ParamEter SYthesis Tool. In *Proc. of CAV*, volume 9207 of *LNCS*, pages 214–231. Springer, 2015.

[13] Erika Ábrahám, Nadine Bergner, Philipp Brauner, Florian Corzilius, Nils Jansen, Thiemo Leonhardt, Ulrich Loup, Johanna Nellen, and Ulrik Schroeder. On collaboratively conveying computer science to pupils. In *Proc. of KOLI*, pages 132–137. ACM Press, 2011.

[14] Erika Ábrahám, Florian Corzilius, Ulrich Loup, and Thomas Sturm. A lazy SMT-solver

for a non-linear subset of real algebra. In *Dagstuhl Seminar Proceedings*, volume 10271. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2010.

CHAPTER 2

Foundations

After some basic notations and definitions for numbers, sets, functions (Section 2.1) and graphs (Section 2.2), we introduce the syntax and semantics of first-order arithmetic formulas in Section 2.3, followed by some normalizations, which we can apply to these formulas in Section 2.4. In Section 2.5, we introduce SAT solving, which is a decision procedure for a special class of the formulas as defined in Section 2.3. With the understanding of the principles of SAT solving, we present the general framework of an SMT solver in Section 2.6, which is capable of solving formulas of a richer class than the one which can be solved by a SAT solver. In Section 2.7, we introduce the virtual substitution method, which forms the basis of the contributions of this thesis presented in the Chapters 4 - 7.

2.1 Numbers, sets and functions

Within this thesis we use *Boolean constants* from $\mathbb{B} := \{\text{true}, \text{false}\}$ and *integer numbers/values (or just integers)* from $\mathbb{Z} := \{..,-2,-1,0,1,2,..\}$. The *(set of) natural numbers/values* $\mathbb{N} := \{1,2,..\}$ are the positive non-zero integers and by $\mathbb{N}_0$ we refer to $\mathbb{N} \cup \{0\}$. We also use the *rational numbers/values (or just rationals)* from $\mathbb{Q}$, which are integers or fractions of integers with a non-zero denominator, for instance $\frac{1}{1}, \frac{-1}{1}, \frac{2}{1}, \frac{-2}{1}, \frac{2}{2}, \frac{-2}{2}, \frac{1}{2}, \frac{-1}{2}, \ldots$. We also make use of the *real numbers/values (or just reals)* from $\mathbb{R}$, which are all numbers representing a point on a continuous and infinite line. They consist of the rationals and *irrationals*, where the latter can be, i. a., *algebraic numbers*, for instance $\sqrt{2}$, and *transcendental numbers*, for instance π.

Given a set M, we denote the *cardinality* of M (the number of elements in M) by $|M|$, where $|M| = \infty$ if M is not finite. We denote the *power set* of a set M by $\mathbb{P}(M)$, which is the set of all subsets of M. If $|M| = \infty$, we obtain the set of all finite subsets of M by $\mathbb{P}_{<\infty}(M)$ and the set of all infinite subsets of M by $\mathbb{P}_{\infty}(M) := \mathbb{P}(M) \setminus \mathbb{P}_{<\infty}(M)$. Subsets of $\mathbb{R}$ can also be represented by

intervals. We distinguish between the following types of intervals

$$\begin{aligned}
[l,u] &:= \{d \in \mathbb{R} |\ l \leq d \leq u\} && \text{(left- and right-closed)}\\
(l',u') &:= \{d \in \mathbb{R} |\ l' < d < u'\} && \text{(left- and right-open)}\\
[l,u') &:= \{d \in \mathbb{R} |\ l \leq d < u'\} && \text{(left-closed and right-open)}\\
(l',u] &:= \{d \in \mathbb{R} |\ l' < d \leq u\} && \text{(left-open and right-closed interval)}
\end{aligned}$$

where $l \in \mathbb{R}$, $l' \in \mathbb{R} \cup \{-\infty\}$ form the *lower bound* and $u \in \mathbb{R}$, $u' \in \mathbb{R} \cup \{\infty\}$ form the *upper bound*. In the case that an interval is left- and right-closed we call it *closed*, otherwise it is *open*. Moreover, if $l' = -\infty$, we call the interval *left-unbounded* and *left-bounded*, otherwise. If $u' = \infty$, we call the interval *right-unbounded* and *right-bounded*, otherwise. If an interval is left- and right-bounded, we refer to it as a *bounded interval*, otherwise it is an *unbounded interval*. If $l > u$, $l' \geq u$, $l \geq u'$ or $l' \geq u'$, the interval is empty and we represent it by the *empty interval* $\emptyset$. The closed interval containing exactly one element, i. e., $[d,d]$ with $d \in \mathbb{R}$, is called a *point interval*. We denote the *set of all intervals* by $\mathbb{I}$.

A *function* $f\colon\ M \to M'$ maps elements from a set M, which we call the *domain* of f and denote by $\mathrm{Dom}(f)$, to elements from a set M', which we call the *codomain* of f and denote by $\mathrm{Codom}(f)$. We see functions also as relations $f \subseteq M \times M'$ with $f(m) = m'$ if and only if $(m,m') \in f$. For a function $f\colon\ M \to M'$ it holds that for all $m \in M$ there exists no $m'_1, m'_2 \in M'$ with $m'_1 \neq m'_2$, $f(m) = m'_1$ and $f(m) = m'_2$. If f *is undefined for an element* $m \in M$, i. e., there exists no $m' \in M'$ with $(m,m') \in f$, we denote this by $f(m) = \perp$ and call f a *partial function*.

Example 1 *Consider the following functions.*

1. *The* signum function *maps a real value* $d \in \mathbb{R}$ *to its sign and is defined by*

$$\mathrm{sgn}\colon\ \mathbb{R} \to \{-1,0,1\}\colon\ d \mapsto \begin{cases} 1 & ,\ d > 0\\ -1 & ,\ d < 0\\ 0 & ,\ \textit{otherwise.}\end{cases}$$

2. *We denote the* greatest common divisor *and* least common multiple *of a finite and non-empty set of non-zero integers by*

$$\gcd\colon\ \mathbb{P}_{<\infty}(\mathbb{Z} \setminus \{0\}) \setminus \{\emptyset\} \to \mathbb{N}$$

   and

$$\mathrm{lcm}\colon\ \mathbb{P}_{<\infty}(\mathbb{Z} \setminus \{0\}) \setminus \{\emptyset\} \to \mathbb{N},$$

   respectively.

3. *The* minimum *and* maximum *of a non-empty subset of the real numbers are denoted by* $\min\colon\ \mathbb{P}(\mathbb{R}) \to \mathbb{R}$ *and* $\max\colon\ \mathbb{P}(\mathbb{R}) \to \mathbb{R}$*, respectively. They are partial functions, as, for instance,* $\max(\mathbb{N})$*,* $\max(\emptyset)$*,* $\min(\mathbb{R})$ *and* $\min((0,1])$ *are undefined.*

4. *The* floor function *is defined by*

$$\lfloor \cdot \rfloor \colon \mathbb{R} \to \mathbb{Z} : d \mapsto \max(\{d' \in \mathbb{Z} |\ d' \leq d\})$$

and the ceiling function *is defined by*

$$\lceil \cdot \rceil \colon \mathbb{R} \to \mathbb{Z} : d \mapsto \min(\{d' \in \mathbb{Z} |\ d' \geq d\}).$$

5. *The* diameter *of an interval is defined by*

$$\Delta_{\mathbb{I}} \colon \mathbb{I} \to \mathbb{R} \cup \{\infty\} : I \mapsto \begin{cases} 0 & , I = \emptyset \\ \infty & , I \text{ is unbounded} \\ u - l & , \text{otherwise, where } l \text{ is the lower and } u \text{ is the upper bound of } I. \end{cases}$$

Note that the diameter of point intervals and the empty interval is 0*.*

2.2 Graphs

At some points in this thesis, we use graphs for illustration purposes and in the data structures of the presented algorithms.

Definition 1 (Directed graph) *A* digraph *or* directed graph *is a tuple* $G := (V, E)$ *of a set* V *of* vertices *and a set* $E \subseteq V \times V$ *of* (directed) edges*.*

Two examples of directed graphs are shown in Figure 2.1.

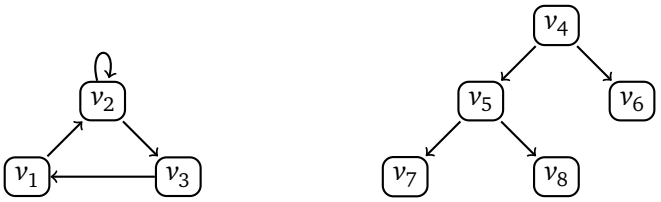


Figure 2.1: The digraph $G = (\{v_1, v_2, v_3\}, \{(v_1, v_2), (v_2, v_2), (v_2, v_3), (v_3, v_1)\})$ on the left; The directed tree $T = (\{v_4, v_5, v_6, v_7, v_8\}, \{(v_4, v_5), (v_4, v_6), (v_5, v_7), (v_5, v_8)\})$ on the right.

Definition 2 (Path, cycle) *Given a digraph* $G = (V, E)$*, a* (finite) path ω (of length $n \in \mathbb{N}$ from v_1 to v_{n+1}) in G *is a sequence*

$$\omega := v_1 v_2 \dots v_n v_{n+1}$$

where $v_i \in V$ $(1 \leq i \leq n+1)$ *and* $(v_i, v_{i+1}) \in E$ $(1 \leq i \leq n)$*. Given two vertices* $v, v' \in V$*, we say that* v' *is* reachable *from* v *in* G*, if there is a path from* v *to* v' *in* G*.*

If $n > 1$*,* $v_1 = v_{n+1}$ *and* $v_1, .., v_n$ *are pairwise different, we call* ω *a* simple cycle*.*

In Figure 2.1 the sequences $v_1v_2v_3$ and $v_4v_5v_7$ are paths, therefore, i. a., v_3 is reachable from v_1 and v_2. Moreover, v_5 and v_7 are reachable from v_4. The sequences v_2v_2 and $v_1v_2v_3v_1$ are simple cycles.

Definition 3 (Directed tree) *A* directed tree $T := (V, E)$ *is a digraph such that either* $V = \emptyset$ *or the following conditions hold:*

1. *There exists exactly one* root $v_r \in V$ *with* $(v, v_r) \notin E$ *for all* $v \in V$ *and all vertices* $v \in V \backslash \{v_r\}$ *are reachable from* v_r *by exactly one path.*
2. *There exists no simple cycle in* T.

If $(v, v') \in E$ *we call* v *the* father *of* v' *and* v' *a* child *of* v. *Vertices without children are called* leaves. *Let* $v \in V$ *and* $V_v \subseteq V$ *be the set containing all vertices reachable from* v *in* T. *Then* $(V_v, E \cap (V_v \times V_v))$ *is called a* subtree of T with the root v.

In Figure 2.1 the digraph on the right is a directed tree T with the root v_4. In this tree, the vertex v_5 has the father v_4 and the children v_7 and v_8, which are leaves. The subtree of T with root v_5 is defined by $(\{v_5, v_7, v_8\}, \{(v_5, v_7), (v_5, v_8)\})$.

2.3 Real and integer arithmetic

As the virtual substitution and it's adaption to SMT solving are clearly one of the major parts of the contribution of this thesis, we do not restrict ourselves to the arithmetic formulas which SMT solvers accept as input, but introduce the more general first-order arithmetic formulas. This is especially necessary, as the original virtual substitution method, which we explain in Section 2.7, operates on these formulas.

2.3.1 Syntax

Variables are one of the basic ingredients in a formula. In the context of this thesis, variables have one of the following three *domains*. Firstly, variables are *Boolean* (or *propositional*) if their domain is $\mathbb{B}$. They allow us to state propositions which can be either true or false, such as "is a certain condition/property fulfilled". We denote the infinite *set of all Boolean variables* by $\mathrm{VAR}_{\mathbb{B}}$. Secondly, variables are *real-valued* if their domain is $\mathbb{R}$. They give us the opportunity to reason about continuous quantities which occur, for instance, in physics, such as time, temperature, velocity or weight. They are also imperative when proving mathematical theorems. Thirdly, variables are *integer-valued* if their domain is $\mathbb{Z}$. We use these variables, if we want to express, for instance, unknown quantities of certain objects. We denote the *set of all real-valued variables* by $\mathrm{VAR}_{\mathbb{R}}$ and the *set of all integer-valued variables* by $\mathrm{VAR}_{\mathbb{Z}}$. The *set of all arithmetic variables* is then $\mathrm{VAR}_{\mathbb{R},\mathbb{Z}} = \mathrm{VAR}_{\mathbb{R}} \cup \mathrm{VAR}_{\mathbb{Z}}$. We define the *domain of a Boolean or arithmetic variable* by $\mathrm{Dom}\colon \mathrm{VAR}_{\mathbb{B}} \cup \mathrm{VAR}_{\mathbb{R},\mathbb{Z}} \to \{\mathbb{B}, \mathbb{Z}, \mathbb{R}\}$.

We can now construct the first order formulas, which we consider in this thesis.

Definition 4 (Syntax of arithmetic formulas) *Syntactically an* (arithmetic) formula φ *is defined by the following abstract grammar:*

$$\begin{aligned} p &::= 1 \mid x \mid (p+p) \mid (p-p) \mid (p \cdot p) \\ \varphi &::= b \mid p < p \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\exists x.\varphi) \end{aligned}$$

Hence, it allows us to combine Boolean variables $b \in \mathrm{VAR}_{\mathbb{B}}$ *and* constraints $p < p$ *with the* Boolean operators $\neg$ *and* $\wedge$*. Moreover, we can* existentially quantify *an arithmetic variable* $x \in \mathrm{VAR}_{\mathbb{R},\mathbb{Z}}$ *with* $\exists x.\varphi$ *making* x quantified in φ*. Constraints compare two* polynomials *using the* relation (symbol) $<$ *(written: "less than"). A polynomial is either the constant* 1*, a real- or integer-valued variable* $x \in \mathrm{VAR}_{\mathbb{R},\mathbb{Z}}$*, or the* sum *(+),* difference *(−) or* product *(·) of two polynomials.*

Note that in the definition of general first-order formulas as it can be found in the literature [Bur98], the constraints correspond to *predicates* and polynomials correspond to *terms*. A formula as defined in Definition 4 is then a Boolean combination of Boolean variables and predicates, which are composed by possibly quantified arithmetic variables and the predicate and function symbols of the *signature* $\tau := \{1, +, -, \cdot, <\}$. Then, the *set of all arithmetic formulas* is denoted by $\mathrm{FO}(\tau)$. Furthermore, the *set of all arithmetic constraints* is denoted by $\mathrm{CS} \subset \mathrm{FO}(\tau)$ and the *set of all polynomials* is denoted by POL.

We also allow syntactic sugar such as the *disjunction* ($\vee$), *implication* ($\rightarrow$), *exclusive disjunction* ($\oplus$) and *equivalence* ($\leftrightarrow$) of two formulas φ_1 and φ_2. These Boolean operators can be expressed based on the grammar in Definition 4 using the following equivalences (in this order):

$$\varphi_1 \oplus \varphi_2 \equiv (\neg\varphi_1) \leftrightarrow \varphi_2 \tag{2.1}$$

$$\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \tag{2.2}$$

$$\varphi_1 \rightarrow \varphi_2 \equiv (\neg\varphi_1) \vee \varphi_2 \tag{2.3}$$

$$\varphi_1 \vee \varphi_2 \equiv \neg((\neg\varphi_1) \wedge (\neg\varphi_2)) \tag{2.4}$$

Furthermore we allow the Boolean constants true and false within a formula, where $\mathrm{true} \equiv \varphi \vee \neg\varphi$ and $\mathrm{false} \equiv \neg\mathrm{true}$ for any formula φ. We can also *quantify* a variable v *universally* in a formula φ by $\forall v.\varphi$ which is equivalent to $\neg\exists v.(\neg\varphi)$.

We allow further relation symbols as syntactic sugar. Besides the already introduced *strict relation* $<$, there are the strict relations $>$ (written: "greater than") and $\neq$ (written: "not equal to"). Additionally, we have the *weak relations* $\leq$, $\geq$ and $=$ (written: "less than or equal to", "greater than or equal to" and "equal to", respectively). We refer to the *set of all relations* by $\mathrm{REL} := \{<, >, =, \neq, \leq, \geq\}$. We call constraints with a weak relation *weak constraints* and *strict constraints* otherwise. We obtain the *set of all constraints in a formula* with $C_{\sim}\colon \mathrm{FO}(\tau) \rightarrow \mathbb{P}(\mathrm{CS})$ and the *relation of a constraint* with $\mathrm{rel}\colon \mathrm{CS} \rightarrow \mathrm{REL}$.

We can transform a constraint comparing two polynomials p_1 and p_2 with one of these rela-

tions to an equivalent formula as defined by the grammar in Definition 4 using the following equivalences (in this order):

$$\begin{aligned} p_1 = p_2 &\equiv (p_1 \leq p_2 \wedge p_1 \geq p_2) \\ p_1 \geq p_2 &\equiv p_2 \leq p_1 \\ p_1 \leq p_2 &\equiv \neg(p_1 > p_2) \\ p_1 \neq p_2 &\equiv (p_1 < p_2 \vee p_1 > p_2) \\ p_1 > p_2 &\equiv p_2 < p_1 \end{aligned}$$

Throughout this thesis we omit parentheses and the multiplication symbol $\cdot$ where it does not lead to confusion, assuming that $\cdot$ binds stronger than $+$ and $-$, $\neg$ binds stronger than $\wedge$ and quantifiers bind weaker than any other Boolean operator. The product of the same arithmetic variable x n times, with $n \in \mathbb{N}_0$, can be denoted by x^n, which we refer to as *x to the power of n*. Moreover we allow any integer within a polynomial, which can be composed by the constant 1 and the arithmetic operations as expected. By this we do not allow non-integer rationals within a polynomial, which forms no restriction.

We further assume that if a variable $x \in \mathrm{VAR}_{\mathbb{R},\mathbb{Z}}$ is bound by a quantifier $Qx.\psi$ ($Q \in \{\exists, \forall\}$) within the formula φ, it holds, w. l. o. g., that x occurs in φ only within ψ. We call such variables *bound*. If a variable is not bound it is *free*.

We define the *set of all variables in a polynomial or formula* by

$$\mathrm{Vars}\colon (\mathrm{POL} \cup \mathrm{FO}(\tau)) \to \mathbb{P}(\mathrm{VAR}_{\mathbb{R},\mathbb{Z}} \cup \mathrm{VAR}_{\mathbb{B}})$$

and the *set of all free variables in a formula* by

$$\mathrm{FreeVars}\colon (\mathrm{POL} \cup \mathrm{FO}(\tau)) \to \mathbb{P}(\mathrm{VAR}_{\mathbb{R},\mathbb{Z}} \cup \mathrm{VAR}_{\mathbb{B}}).$$

Note that for any polynomial $p \in \mathrm{POL}$, it holds that $\mathrm{Vars}(p) \cap \mathrm{VAR}_{\mathbb{B}} = \emptyset$ (polynomials contain no Boolean variables), and for any formula $\varphi \in \mathrm{FO}(\tau)$, it holds that $(\mathrm{Vars}(\varphi) \setminus \mathrm{FreeVars}(\varphi)) \cap \mathrm{VAR}_{\mathbb{B}} = \emptyset$ (Boolean variables are always free).

Given a formula $\varphi \in \mathrm{FO}(\tau)$, if all variables in $\mathrm{Vars}(\varphi)$ are real-valued, all variables are integer-valued or all variables are Boolean, φ is a *real arithmetic*, *integer arithmetic*, or *Boolean/ propositional* formula, respectively. If $\mathrm{Vars}(\varphi)$ contains both integer- and real-valued variables, but no Boolean variables we call φ a *mixed integer-real arithmetic* formula. If $\mathrm{Vars}(\varphi)$ contains Boolean variables and only real-valued, only integer-valued or both integer- and real-valued variables, φ is a *real arithmetic, integer arithmetic or mixed integer-real arithmetic formula with Boolean variables*. We call a formula φ *quantifier-free*, if $\mathrm{Vars}(\varphi) = \mathrm{FreeVars}(\varphi)$, and a *sentence*, if $\mathrm{FreeVars}(\varphi) = \emptyset$.

Given a polynomial $p \in \mathrm{POL}$ (or constraint $c \in \mathrm{CS}$), if $|\mathrm{Vars}(p)| = 0$ (or $|\mathrm{Vars}(c)| = 0$) we call p (or c) *constant*, if $|\mathrm{Vars}(p)| = 1$ (or $|\mathrm{Vars}(c)| = 1$) we call p (or c) *univariate*, and otherwise

multivariate.

Example 2 *Consider the formulas*

$$\begin{aligned}
\varphi_1 &= (\neg b_1 \vee b_2) \leftrightarrow (b_1 \rightarrow b_2) \\
\varphi_2 &= (x_2^2 + 3) \cdot (1 + 2x_1^2) < 0 \\
\varphi_3 &= z_1 = 2 \\
\varphi_4 &= 2z_1 + 1 \geq 3x_1 \\
\varphi_5 &= \varphi_1 \rightarrow \varphi_3 \\
&= ((\neg b_1 \vee b_2) \leftrightarrow (b_1 \rightarrow b_2)) \rightarrow z_1 = 2 \\
\varphi_6 &= \neg((x_2^2 + 3) \cdot (1 + 2x_1^2) < 0 \vee \neg(2z_1 + 1 \geq 3x_1 \rightarrow b_2)) \\
\varphi_7 &= \exists x_1 . \varphi_6 \\
&= \exists x_1 . \neg((x_2^2 + 3) \cdot (1 + 2x_1^2) < 0 \vee \neg(2z_1 + 1 \geq 3x_1 \rightarrow b_2)) \\
\varphi_8 &= \exists x_1 . \forall x_2 .\ x_1 x_2 = x_2
\end{aligned}$$

where $\mathrm{Dom}(b_1), \mathrm{Dom}(b_2) = \mathbb{B}$, $\mathrm{Dom}(z_1) = \mathbb{Z}$ *and* $\mathrm{Dom}(x_1) = \mathrm{Dom}(x_2) = \mathbb{R}$. *Then,* φ_1 *is a propositional formula and* φ_2, φ_3 *and* φ_4 *are constraints. Furthermore,* φ_2 *is a real arithmetic formula,* φ_3 *is an integer arithmetic formula and* φ_4 *is a mixed integer-real arithmetic formula. The formula* φ_5 *is an integer arithmetic formula with Boolean variables and* φ_6 *is a mixed integer-real arithmetic formula with Boolean variables. All formulas except* φ_7 *and* φ_8 *are quantifier-free, where in* φ_7 *the only quantified variable is* x_1 *and, hence,* b_2, x_2 *and* z_1 *are free variables in* φ_7. *In* φ_8 *all variables are quantified, so it is a sentence. The left-hand side of* φ_2 *is a multivariate polynomial and its right-hand side is constant. The left-hand sides of* φ_3 *and* φ_4 *are univariate polynomials. Moreover,* φ_2 *and* φ_4 *are multivariate constraints and* φ_3 *is a univariate constraint.*

Definition 5 (Substitution) *The* substitution *of an arithmetic variable by a polynomial in a formula or polynomial is defined by*

$$\cdot[\,\cdot\,/\,\cdot\,]\colon\ (\mathrm{POL} \cup \mathrm{FO}(\tau)) \times \mathrm{POL} \times \mathrm{VAR}_{\mathbb{R},\mathbb{Z}} \rightarrow \mathrm{POL} \cup \mathrm{FO}(\tau)$$

such that

$$\begin{aligned}
1[p/x] &:= 1 \\
x[p/x] &:= p \\
y[p/x] &:= y && , y \neq x \\
(p_1 + p_2)[p/x] &:= (p_1[p/x] + p_2[p/x]) \\
(p_1 - p_2)[p/x] &:= (p_1[p/x] - p_2[p/x]) \\
(p_1 \cdot p_2)[p/x] &:= (p_1[p/x] \cdot p_2[p/x]) \\
b[p/x] &:= b
\end{aligned}$$

$$
\begin{aligned}
(p_1 < p_2)[p/x] &:= (p_1[p/x] < p_2[p/x]) \\
(\neg\varphi_1)[p/x] &:= (\neg\varphi_1[p/x]) \\
(\varphi_1 \wedge \varphi_2)[p/x] &:= (\varphi_1[p/x] \wedge \varphi_2[p/x]) \\
(\exists x.\varphi_1)[p/x] &:= (\exists x.\varphi_1) \\
(\exists y.\varphi_1)[p/x] &:= (\exists y.\varphi_1[p/x]) \qquad , y \neq x
\end{aligned}
$$

with $x, y \in \mathrm{VAR}_{\mathbb{R},\mathbb{Z}}$, $b \in \mathrm{VAR}_{\mathbb{B}}$, $p, p_1, p_2 \in \mathrm{POL}$ *and* $\varphi_1, \varphi_2 \in \mathrm{FO}(\tau)$. *A substitution* $\varphi[p/x]$ *means that every free occurrence of* x *in* φ *is replaced by* p. *Furthermore, we define that*

$$\varphi[p_1/x_1]..[p_n/x_n] := (..(\varphi[p_1/x_1])..)[p_n/x_n],$$

where $x_1, .., x_n$ *are arithmetic variables and* $p_1, .., p_n$ *are polynomials.*

Example 3 *Consider the following substitutions in two of the formulas from Example 2:*

$$
\begin{aligned}
((x_2^2+3)\cdot(1+2x_1^2)<0)[1/x_1] &= (x_2^2+3)\cdot(1+2\cdot 1^2)<0 \\
&= (x_2^2+3)\cdot 3<0 \\
(2z_1+1\geq 3x_1)[z_1/x_1][2/z_1] &= (2z_1+1\geq 3z_1)[2/z_1] \\
&= 5\geq 6
\end{aligned}
$$

Given a formula φ, we obtain its *real relaxation* $\varphi^{\mathbb{R}}$ by substituting φ's integer-valued variables $\mathrm{Vars}(\varphi) \cap \mathrm{VAR}_{\mathbb{Z}} = \{z_1, .., z_n\}$ by fresh real-valued variables $x_1, .., x_n \notin \mathrm{Vars}(\varphi)$, respectively, i. e., $\varphi^{\mathbb{R}} = \varphi[x_1/z_1]..[x_n/z_n]$. We obtain the *Boolean abstraction* $\varphi^{\mathbb{B}}$ of a formula φ by replacing φ's constraints $C_{\sim}(\varphi) = \{c_1, .., c_n\}$ by fresh Boolean variables $b_1, .., b_n \notin \mathrm{Vars}(\varphi)$, respectively. Then, we denote the *Boolean abstraction mapping* by the function $\mathrm{abstr}_{\varphi}^{\mathbb{B}}\colon\ C_{\sim}(\varphi) \to \mathrm{VAR}_{\mathbb{B}} : c_i \mapsto b_i$.

2.3.2 Semantics

In Section 2.3.1 we defined how a formula can be composed syntactically, but we have not yet fixed how we interpret the meaning of a formula. For this purpose, we usually need an interpretation giving all yet uninterpreted identifiers in a formula a meaning. For an arithmetic formula $\varphi \in \mathrm{FO}(\tau)$, we use the τ*-structure* $\mathfrak{A}$ mapping 1, $+$, $-$, $\cdot$ and $<$ to their expected meaning, which we do not axiomatize here and denote in the following by $1_{\mathfrak{A}}$, $+_{\mathfrak{A}}$, $-_{\mathfrak{A}}$, $\cdot_{\mathfrak{A}}$ and $<_{\mathfrak{A}}$. The only unspecified identifiers which have not yet received a fixed meaning are the variables in φ. Therefore, we specify the semantics of an arithmetic formula only with respect to assignments of values to the formula's variables $\mathrm{Vars}(\varphi)$, such that we assign to a variable only values from its domain.

Definition 6 (Assignment) *An* assignment (of values to variables) *is a possibly partial function*

$$\alpha\colon\ (\mathrm{VAR}_{\mathbb{B}} \cup \mathrm{VAR}_{\mathbb{R},\mathbb{Z}}) \to (\mathbb{B} \cup \mathbb{R}) : v \mapsto d \in \mathrm{Dom}(v).$$

We denote the set of all assignments *by* ASS.

Given a formula (or polynomial) φ, an assignment α is called a (full) assignment for φ *if* $\text{FreeVars}(\varphi) \subseteq \text{Dom}(\alpha)$ *and a* partial assignment for φ*, otherwise. The* set of all full or partial assignments *for φ is denoted by* $\text{Assigns}(\varphi)$ *and* $\text{partialAssigns}(\varphi)$*, respectively. Note that if* $\text{FreeVars}(\varphi) = \emptyset$*,* $\text{Assigns}(\varphi)$ *contains the* empty assignment $\alpha_\perp$ *with* $\text{Dom}(\alpha_\perp) = \emptyset$*. Moreover, the assignment α' is called an* extension *of the assignment α, if* $\text{Dom}(\alpha) \subset \text{Dom}(\alpha')$ *and for all* $v \in \text{Dom}(\alpha)$ *it holds that* $\alpha'(v) = \alpha(v)$*.*

We can adapt an assignment such that it maps a variable to a given value in the variable's domain by

$$\cdot\,[\,\cdot\,/\,\cdot\,]\colon\ \text{ASS} \times (\mathbb{B} \cup \mathbb{R}) \times (\text{VAR}_{\mathbb{B}} \cup \text{VAR}_{\mathbb{R},\mathbb{Z}}) \to \text{ASS} : \alpha[d/v] \mapsto \alpha'$$

where

$$\alpha'(v') \;=\; \begin{cases} d & \text{, if } v' = v \\ \alpha(v') & \text{, otherwise.} \end{cases}$$

Definition 7 (Formula and polynomial evaluation) *Given a full assignment α for a polynomial or an arithmetic formula φ, we can* evaluate φ under α *by*

$$[\![\,\cdot\,]\!]^{\cdot}\ :\ ((\text{FO}(\tau) \cup \text{POL}) \times \text{ASS}) \to (\mathbb{B} \cup \mathbb{R}),$$

which is defined inductively with respect to the abstract grammar in Definition 4 by

$$\begin{array}{lcl}
[\![1]\!]^\alpha & := & 1_{\mathfrak{A}} \\
[\![x]\!]^\alpha & := & \alpha(x) \\
[\![p_1 + p_2]\!]^\alpha & := & [\![p_1]\!]^\alpha +_{\mathfrak{A}} [\![p_2]\!]^\alpha \\
[\![p_1 - p_2]\!]^\alpha & := & [\![p_1]\!]^\alpha -_{\mathfrak{A}} [\![p_2]\!]^\alpha \\
[\![p_1 \cdot p_2]\!]^\alpha & := & [\![p_1]\!]^\alpha \cdot_{\mathfrak{A}} [\![p_2]\!]^\alpha \\
[\![b]\!]^\alpha & := & \alpha(b) \\
[\![p_1 < p_2]\!]^\alpha & := & [\![p_1]\!]^\alpha <_{\mathfrak{A}} [\![p_2]\!]^\alpha \\
[\![\neg\varphi_1]\!]^\alpha & := & \begin{cases} \text{false} & \text{, if } [\![\varphi_1]\!]^\alpha = \text{true} \\ \text{true} & \text{, otherwise} \end{cases} \\
[\![\varphi_1 \wedge \varphi_2]\!]^\alpha & := & [\![\varphi_1]\!]^\alpha = \text{true} \text{ and } [\![\varphi_2]\!]^\alpha = \text{true} \\
[\![\exists v.\varphi_1]\!]^\alpha & := & \text{exists } d \in \text{Dom}(v) \text{ such that } [\![\varphi_1]\!]^{\alpha[d/v]} = \text{true}
\end{array}$$

where $x \in \text{Dom}(\alpha)$ is an integer- or real-valued variable, $b \in \text{Dom}(\alpha)$ is a Boolean variable, p_1 and p_2 are polynomials and φ_1 and φ_2 are arithmetic formulas with $\text{Vars}(p_1)$*,* $\text{Vars}(p_2)$*,* $\text{FreeVars}(\varphi_1)$ *and* $\text{FreeVars}(\varphi_2)$ *containing only variables from* $\text{Dom}(\alpha)$*.*

Now we can determine whether a formula φ evaluates to true or false under a given full assignment for φ, which leads us to the next definition.

Definition 8 (Satisfiability of arithmetic formulas) *For a given arithmetic formula φ we define its* set of solutions *by*

$$\Theta(\varphi) := \{\alpha \in \text{Assigns}(\varphi) |\ [\![\varphi]\!]^\alpha = \text{true}\}.$$

An arithmetic formula φ is satisfiable *if and only if $\Theta(\varphi) \neq \emptyset$. If φ is not satisfiable, we call it* unsatisfiable *and, if $\Theta(\varphi) = \text{Assigns}(\varphi)$, we call φ* valid *or alternatively a* tautology.

Throughout this thesis we also use the term *solution space* instead of *set of solutions and we refer to a formula φ's set of solutions or solution space for a variable $v \in \text{Vars}(\varphi)$* by $\{d \in \text{Dom}(v) |\ \exists \alpha \in \Theta(\varphi).\ \alpha(v) = d\}$.

Example 4 *Consider the following three formulas from Example 2:*

$$\begin{aligned} \varphi_1 &= (\neg b_1 \vee b_2) \leftrightarrow (b_1 \rightarrow b_2) \\ \varphi_2 &= (x_2^2 + 3) \cdot (1 + 2x_1^2) < 0 \\ \varphi_5 &= ((\neg b_1 \vee b_2) \leftrightarrow (b_1 \rightarrow b_2)) \ \rightarrow \ z_1 = 2 \end{aligned}$$

Considering Equation (2.3)*, the left-hand side and the right-hand side of the equivalence in φ_1 are indeed equivalent and, therefore, $\Theta(\varphi_1) = \text{Assigns}(\varphi_1)$, which makes it a tautology. As x_1^2 and x_2^2 are non-negative, the left-hand side of the constraint in φ_2 is always positive. Hence, $\Theta(\varphi_2) = \emptyset$, which makes φ_2 unsatisfiable. As φ_1, which forms the left-hand side of the implication in φ_5, is valid, φ_5 is satisfied if and only if its right-hand side $z_1 = 2$ is satisfied. As $z_1 = 2$ can be satisfied by assigning 2 to z_1, φ_5 is satisfiable with $\Theta(\varphi_5) = \{\alpha \in \text{Assigns}(\varphi_5) |\ \alpha(z_1) = 2\}$.*

2.4 Normalizations

The normalization process aims to specify a set of transformations, which can be applied to an input, as long as it is not yet in a (unique) normal form. Thereby the transformations assure that certain properties of the input are not changed. We want to normalize polynomials and formulas in order to ease the generalization of procedures which operate on them. In the following, we firstly present how we normalize polynomials. Based on normalized polynomials, we define a unique normal form for constraints followed by three well known, but not unique, normal forms for formulas which contain Boolean operators or quantifiers.

2.4.1 Polynomials

A polynomial p as defined in the grammar of Definition 4 is an element of the ring $\mathbb{Z}[x_1, .., x_n]$, if we assume that $\text{Vars}(p) = \{x_1, .., x_n\}$. Among others, the following axioms hold, where p_1, p_2 and p_3 are polynomials:

$$p_1 + p_2 \quad = \quad p_2 + p_1 \qquad (+ \text{ is commutative}) \tag{2.5}$$

$$p_1 \cdot p_2 = p_2 \cdot p_1 \quad (\cdot \text{ is commutative}) \tag{2.6}$$

$$p_1 \cdot (p_2 + p_3) = (p_1 \cdot p_2) + (p_1 \cdot p_3) \quad (\cdot \text{ is distributive with respect to } +) \tag{2.7}$$

Using these axioms, we can syntactically transform a polynomial. Our aim is to define a normal form such that it offers a unique representation for all polynomials which can be obtained by applying a finite sequence of these transformations to this representation.

As a first step towards normalized polynomials, we have to fix an order on the arithmetic variables which can occur in a polynomial, which we refer to as *variable order*.

Definition 9 (Monomial) *Given a finite set of arithmetic variables* $\{x_1, .., x_n\} \subset \mathrm{VAR}_{\mathbb{R},\mathbb{Z}}$ *and a variable order* $x_1 < x_2 < .. < x_{n-1} < x_n$ *a* monomial *is defined by the power product*

$$x_1^{e_1} \cdot \cdots \cdot x_n^{e_n}$$

where $e_i \in \mathbb{N}_0$, $1 \leq i \leq n$, *is called the* exponent (of the variable x_i). *We denote the* set of all monomials over the variables $x_1, .., x_n$ *by* $M[x_1, .., x_n]$ *and define the* degree of a monomial *as follows:*

$$\deg\colon\ M[x_1, .., x_n] \to \mathbb{N}_0 : \quad x_1^{e_1} \cdot \cdots \cdot x_n^{e_n} \mapsto \sum_{i=1}^{n} e_i$$

In the course of this thesis, we always use indexed arithmetic variables, if more than one variable occurs. We assume the variable order to be as presented in Definition 9, that is the variable with the lower index is smaller in the variable order. If an integer-valued variable is compared with a real-valued variable, we assume the integer-valued variable to be smaller in the variable order.

We now want to define a normal form for polynomials based on monomials. For this purpose we need an order on monomials, which can be used analogically to the variable orders in monomials.

Definition 10 (Reverse lexicographical order) *Given two monomials* $m_1, m_2 \in M[x_1, .., x_n]$ *with* $m_1 = x_1^{e_{1,1}} \cdot \cdot\, x_n^{e_{n,1}}$ *and* $m_2 = x_1^{e_{1,2}} \cdot \cdot\, x_n^{e_{n,2}}$, *it holds that*

$$m_1 < m_2 \text{ (in the reverse lexicographical order)}$$
$$\Leftrightarrow$$
$$\exists i \in \{1, .., n\}.\ e_{i,1} < e_{i,2} \ \wedge\ \forall j \in \{i+1, .., n\}.\ e_{j,1} = e_{j,2}.$$

Based on the variable and monomial order we can define a normal form for polynomials.

Definition 11 (Normalized polynomial, term) *Let* $p \in \mathrm{POL}$ *and let* $\mathrm{Vars}(p) = \{x_1, .., x_n\}$. *Using Equations (2.5-2.7), we can transform* p *to the* normalized polynomial

$$a_1 \cdot \overbrace{x_1^{e_{1,1}} \cdot \cdot\, x_n^{e_{n,1}}}^{m_1} + \cdot\cdot + a_k \cdot \overbrace{x_1^{e_{1,k}} \cdot \cdot\, x_n^{e_{n,k}}}^{m_k},$$

where $a_1, .., a_k$ *are the coefficients with either* $a_1, .., a_k \in \mathbb{Z} \setminus \{0\}$ *or* p *is the* zero polynomial

($k = 1$ and $a_1 = e_{1,1} = .. = e_{n,1} = 0$). For the monomials $m_1, .., m_k \in M[x_1, .., x_n]$ it holds that $m_k < m_{k-1} < .. < m_2 < m_1$ (in reverse lexicographical order). We call the product $t_j := a_j \cdot m_j$ of a coefficient and monomial a term *($1 \leq j \leq k$). The term $a_1 \cdot m_1$ is called the* leading term *with a_1 being the* leading coefficient. *By $\deg(t_j) := \deg(m_j)$ we denote the* degree *of the term t_j. If $\deg(t_k) = 0$ then $t_k = a_k$ is* constant *and forms the* constant part of p. *Otherwise the constant part of p is 0. We denote the* set of all (normalized) polynomials over the variables $x_1, .., x_n$ *by $\mathbb{Z}[x_1, .., x_n]$.*

The (total) degree of normalized polynomials $p = t_1 + .. + t_k$ *is defined by*

$$\deg\colon \text{POL} \to \mathbb{N}_0 : p \mapsto \max(\{\deg(t_j) |\ 1 \leq j \leq k\})$$

and the degree of a variable in a normalized polynomial *is defined by*

$$\deg\colon \{x_1, .., x_n\} \times \mathbb{Z}[x_1, .., x_n] \to \mathbb{N}_0 : (x_i, p) \mapsto \max(\{e_{i,j} |\ 1 \leq j \leq k\}).$$

We call a polynomial p *constant*, if $\deg(p) = 0$, *linear*, if $\deg(p) = 1$, and *nonlinear*, otherwise. Note that we do not allow rationals in polynomials as defined by the grammar of Definition 4. The same holds for polynomials in normal form. Fortunately, this is no restriction, as we can easily transform a polynomial with rationals to a polynomial as defined by the grammar of Definition 4, by first transforming it to a normal form analogously to Definition 11 and then multiplying the result's coefficients by the least common multiple of their denominators.

Example 5 *We transform the polynomial, which forms the left-hand side of the constraint in φ_2 from Example 2, to a normalized polynomial using Equations (2.5-2.7) as follows:*

$$\begin{aligned}
& (x_2^2 + 3) \cdot (1 + 2x_1^2) \\
\overset{Eq.\ 2.7}{=}\ & (x_2^2 + 3) + (x_2^2 + 3)2x_1^2 \\
\overset{Eq.\ 2.6}{=}\ & (x_2^2 + 3) + 2x_1^2(x_2^2 + 3) \\
\overset{Eq.\ 2.7}{=}\ & x_2^2 + 3 + x_2^2 \cdot 2x_1^2 + 3 \cdot 2x_1^2 \\
\overset{Eq.\ 2.6}{=}\ & x_2^2 + 3 + 2x_1^2x_2^2 + 6x_1^2 \\
\overset{Eq.\ 2.5}{=}\ & 2x_1^2x_2^2 + x_2^2 + 6x_1^2 + 3
\end{aligned}$$

The resulting polynomial is normalized, as its monomials are normalized and in reverse lexicographical order

$$x_1^0x_2^0 < x_1^2x_2^0 < x_1^0x_2^2 < x_1^2x_2^2.$$

The leading term is $2x_1^2x_2^2$, the leading coefficient is 2 and the constant part is 3. The degree of the resulting polynomial is $\max(\{4, 2, 2, 0\}) = 4$ and its degree in x_1 and x_2 is 2 in both cases.

Throughout this thesis, polynomials are always expected to be in normal form no matter whether they are from POL or we know, that they contain the variables $x_1, .., x_n$ and are from $\mathbb{Z}[x_1, .., x_n]$. For the sake of the ability of distinguishing normalized polynomials, we also need an order for them.

Definition 12 (Polynomial order) *Given two polynomials* $p_1, p_2 \in \mathbb{Z}[x_1, .., x_n]$ *with* $p_1 = a_{1,1} \cdot m_{1,1} + .. + a_{1,k_1} \cdot m_{1,k_1}$ *and* $p_2 = a_{2,1} \cdot m_{2,1} + .. + a_{2,k_2} \cdot m_{2,k_2}$ *it holds that*

$$p_1 < p_2 \text{ (in the polynomial order)}$$
$$\Leftrightarrow$$
$$\exists i \in \{1, .., \min(\{k_1, k_2\})\}.\ (m_{1,i} < m_{2,i} \vee (m_{1,i} = m_{2,i} \wedge a_{1,i} < a_{2,i}))$$
$$\wedge\ \forall j \in \{1, .., i-1\}.\ m_{1,j} = m_{2,j} \wedge a_{1,j} = a_{2,j}$$

with $(m_{1,j} = m_{2,j}) := \neg(m_{1,j} < m_{2,j} \vee m_{2,j} < m_{1,j})$.

We use this order to distinguish two polynomials p_1 and p_2 by simply checking whether $p_1 < p_2$ or $p_2 < p_1$.

In the context of satisfiability checking and especially the virtual substitution as explained in Section 2.7, we are often interested in the full assignments for a polynomial p which evaluate it to 0. We call the set of these assignments the *zeros of* p and denote it by

$$\text{zeros}\colon\ \mathbb{Z}[x_1, .., x_n] \to \text{ASS} : p \mapsto \{\alpha \in \text{Assigns}(p) |\ [\![p]\!]^\alpha = 0\}.$$

We can divide a polynomial $p := \sum_{i=1}^{k} a_i m_i$ into the product of it's *content*, which is denoted by

$$\text{cont}\colon\ \mathbb{Z}[x_1, .., x_n] \to \mathbb{Z} : p \mapsto \begin{cases} \text{sgn}(a_1) \cdot \gcd(\{a_1, .., a_k\}) & , p \neq 0 \\ 0 & , \text{otherwise} \end{cases}$$

and *primitive part*, which is denoted by

$$\text{prim}\colon\ \mathbb{Z}[x_1, .., x_n] \to \mathbb{Z}[x_1, .., x_n] : \begin{cases} p \mapsto \sum_{i=1}^{k} \frac{a_i}{\text{cont}(p)} m_i & , \text{ if } p \neq 0 \\ 0 \mapsto 0 & , \text{otherwise} \end{cases}$$

i. e., $p = \text{cont}(p) \cdot \text{prim}(p)$. Note that the coefficients of $\text{prim}(p)$ are integers and that $\text{zeros}(p) = \text{zeros}(\text{prim}(p))$. If $\text{prim}(p) = p$ we call p *primitive*.

The normal form of the result of a multiplication of two polynomials in normal form with k_1 and k_2 terms, respectively, has in the worst case $k_1 \cdot k_2$ terms. We accept this cost, as many of the operations we perform on polynomials iterate over their terms and therefore we need to expand the polynomial anyway. Additionally, we present an alternative representation of polynomials in Chapter 8, which tries to avoid the complexity of polynomial multiplication while storing polynomials as products of polynomials as introduced in Definition 13.

Definition 13 (Polynomial factorization) *Given a primitive polynomial* $p \neq 0$*, a* factorization *of* p *is defined by*

$$\{p_1^{e_1}, \ldots, p_n^{e_n}\}$$

with the $n \in \mathbb{N}_0$ factors $p_i^{e_i}$*, such that the bases* $p_1, .., p_n \notin \{0, 1\}$ *are pairwise different primitive polynomials, the exponents are* $e_1, .., e_n \in \mathbb{N}$ *and it holds that* $p = \prod_{i=1}^{n} p_i^{e_i}$ *(which is defined to*

be 1 *for* $n = 0$*). We denote the* set of all polynomial factorizations *by* FAC.

Note that the only factorization of the polynomial 1 is $\emptyset$. Note that a factorization is a set, which is ordered by the bases of its factors according to the polynomial order of Definition 12.

The factorization of a primitive polynomial p is not unique in many cases. However, if it is unique then the (only) factorization of p is $\{p^1\}$ and we call p *irreducible*. If the bases of all factors in a factorization of p are irreducible, we call it a *full factorization* and otherwise a *(partial) factorization*. Note that every non-constant primitive polynomial has a unique full factorization.

Example 6 *Consider the polynomial*

$$p \quad = \quad 6x_1^4x_2^2 + 3x_1^2x_2^2 + 18x_1^4 + 9x_1^2$$

which is the product of the result $2x_1^2x_2^2 + x_2^2 + 6x_1^2 + 3$ *from Example 5 and the polynomial* $3x_1^2$*. The primitive part of* p *is*

$$\text{prim}(p) \quad = \quad 2x_1^4x_2^2 + x_1^2x_2^2 + 6x_1^4 + 3x_1^2$$

and the content of p *is* $\text{cont}(p) = 3$*. A valid factorization of* $\text{prim}(p)$ *is, for instance,*

$$\{x_1^2,\ (2x_1^2x_2^2 + x_2^2 + 6x_1^2 + 3)^1\},$$

as x_1 *and* $2x_1^2x_2^2 + x_2^2 + 6x_1^2 + 3$ *are both primitive. The full factorization of* $\text{prim}(p)$ *is*

$$\{x_1^2,\ (2x_1^2 + 1)^1,\ (x_2^2 + 3)^1\},$$

as x_1*,* $2x_1^2 + 1$ *and* $x_2^2 + 3$ *are all irreducible.*

2.4.2 Formulas

When transforming a formula φ in order to attain a normal form, we have to make sure that the result ψ of a transformation is equivalent to φ, where φ and ψ are *equivalent*, if and only if $\Theta(\varphi) = \Theta(\psi)$. If we allow a transformation to eliminate all occurrences of some variables in φ or to introduce fresh variables, that is variables which are not yet elements of $\text{Vars}(\varphi)$, we need a more general property than equivalence.

Definition 14 (Equisatisfiability of formulas) *Two formulas* φ *and* ψ *are* equisatisfiable *if and only if* $\Theta(\varphi) \neq \emptyset \iff \Theta(\psi) \neq \emptyset$ *.*

Note that two equisatisfiable formulas φ and ψ with $\text{Vars}(\varphi) \neq \text{Vars}(\psi)$ are not equivalent, as $\Theta(\varphi) \neq \Theta(\psi)$. All transformations, which we use in the remainder of this section in order to achieve a normal form, keep the formula to transform and the result of the transformation equisatisfiable.

2.4.2.1 Constraints

In general, we only apply a simple normalization to constraints with the intention of minimizing the cases we must consider in the procedures presented in this thesis.

Definition 15 (Normalized constraint) *Given a real or mixed-integer-real arithmetic constraint* $c := p_1 \sim p_2$ *(*$\sim \in$ REL*) as defined by the grammar of Definition 4, where we also take the relations into account which we defined as syntactic sugar, the* normal form *of* c *is depicted in the following table, where* $p := \text{prim}(p_1 - p_2)$ *and* $d := \text{cont}(p_1 - p_2)$*:*

			$<$	$\leq$	$=$	$\neq$	$\geq$	$>$
$p = 0$			false	true	true	false	true	false
$p = 1$	$\wedge$	$d < 0$	true	true	false	true	false	false
$p = 1$	$\wedge$	$d > 0$	false	false	false	true	true	true
$p \notin \{0,1\}$	$\wedge$	$d < 0$	$p > 0$	$p \geq 0$	$p = 0$	$p \neq 0$	$p \leq 0$	$p < 0$
$p \notin \{0,1\}$	$\wedge$	$d > 0$	$p < 0$	$p \leq 0$	$p = 0$	$p \neq 0$	$p \geq 0$	$p > 0$

By this normalization we either transform a constraint to a Boolean constant or reach the state that the normalized constraint's right-hand side is 0 and that it's left-hand side is a non-constant primitive polynomial. From here on, we assume constraints to be normalized.

Example 7 *Considering the constraints occurring in Example 2 and Example 3, we normalize them as follows:*

$$\begin{aligned}
(x_2^2 + 3) \cdot (1 + 2x_1^2) < 0 &= 2x_1^2x_2^2 + x_2^2 + 6x_1^2 + 3 < 0 \\
z_1 = 2 &= z_1 - 2 = 0 \\
2z_1 + 1 \geq 3x_1 &= 3x_1 - 2z_1 - 1 \leq 0 \\
(x_2^2 + 3) \cdot 3 < 0 &= x_2^2 + 3 < 0 \\
5 \geq 6 &= \text{false} \\
x_1x_2 = x_2 &= x_1x_2 - x_2 = 0
\end{aligned}$$

In Example 5 we have already seen how to normalize the left-hand side of $(x_2^2 + 3) \cdot (1 + 2x_1^2) < 0$ *and the left-hand sides of* $z_1 = 2$ *and* $x_1x_2 = x_2$ *are normalized polynomials straight away, if we subtract* 2 *and* x_2 *from both the left- and the right-hand side, respectively. The three constraints are normalized, as their left-hand sides are primitive and their right-hand sides are zero (the last row of the table in Definition 15 applies). It is a bit more complicated for* $2z_1 + 1 \geq 3x_1$*. After subtracting its right-hand side and ordering the terms on the left-hand side, we obtain* $-3x_1 + 2z_1 + 1 \geq 0$*. The primitive part of the left-hand side is* $3x_1 - 2z_1 - 1$ *and its content is* -1*. From the table in Definition 15 we can look up the result in the fourth row and fifth column. The primitive part of* $(x_2^2 + 3) \cdot 3 < 0$ *is* $x_2^2 + 3$ *and its content is* 3*, therefore we can find its normalized version in the last row and first column. The last constraint* $5 \geq 6$ *can obviously be evaluated to* false*. In the normalization process however, we would achieve this information automatically by subtracting the*

constraint by 6 *resulting in* $-1 \geq 0$*. Here the primitive part is* 1 *and the content is* -1*, therefore we can find the normalized version of the constraint in the second row and fifth column, which is indeed* false*.*

For integer arithmetic constraints, we can achieve further normalizations in addition to those in Definition 15.

Definition 16 (Normalized integer arithmetic constraint) *Let* $p \sim 0$ *be an integer arithmetic constraint (in normal form according to Definition 15, therefore* $p = \sum_{i=1}^{k} a_i m_i$*). Furthermore, we assume, w. l. o. g., that* $\sim \in \{<, \leq, =, \geq, >\}$ *(as* $p \neq 0$ *is equivalent* $(p < 0 \vee p > 0)$*). The normal form of* $p \sim 0$ *is depicted in the following table:*

	$<$	$\leq$	$=$	$\geq$	$>$
$d \in \mathbb{Z}$	$r + d + 1 \leq 0$	$r + d \leq 0$	$r + d = 0$	$r + d \geq 0$	$r + d - 1 \geq 0$
$d \notin \mathbb{Z}$	$r + \lceil d \rceil \leq 0$	$r + \lceil d \rceil \leq 0$	false	$r + \lfloor d \rfloor \geq 0$	$r + \lfloor d \rfloor \geq 0$

with

$$r = \begin{cases} \sum_{i=1}^{k-1} \frac{a_i}{\gcd(\{a_1,..,a_{k-1}\})} \cdot m_i & \text{, if } \deg(m_k) = 0 \\ \sum_{i=1}^{k} \frac{a_i}{\gcd(\{a_1,..,a_k\})} \cdot m_i & \text{, otherwise} \end{cases}$$

and

$$d = \begin{cases} \frac{a_k}{\gcd(\{a_1,..,a_{k-1}\})} & \text{, if } \deg(m_k) = 0 \\ 0 & \text{, otherwise.} \end{cases}$$

Therefore, normalized integer-arithmetic constraints contain only weak relations. Over the course of this thesis we assume an integer arithmetic constraint to be normalized as given by Definition 16.

A similar normalization was already presented in [DDA09] for linear integer-arithmetic constraints and it is based on the fact that $r + d = 0$ only has a solution if $d \in \mathbb{Z}$. A formal proof can be found in [NW88].

The main idea of why the normalization in Definition 16 results in an equisatisfiable constraint can be explained as follows. We divide the left-hand side of the constraint $c := p \sim 0$ by the greatest common divisor g of the coefficients of p's non-constant terms, which does not change the set of solutions of c as g is a positive rational and c's right-hand side is 0. By the definition of the greatest common divisor, we know that for all coefficients a_i of p's non-constant terms it must hold that $\frac{a_i}{g} \in \mathbb{Z}$. Given a solution of c, we know that all monomials are evaluated to an integer as the product of two integers is an integer. Hence, for a given integer solution of c, r is also evaluated to an integer, as the sum of two integers is integral. If c is an equation, it follows that it has no solution if d is not integral (Column 3 of the table in Definition 16). If the relation symbol of c is $<$, any solution of c must evaluate r to an integer which is strictly less than $-d$ or, equivalently, to an integer which is less than or equal to the next smaller integer than $-d$. This is $-\lceil d \rceil$, if d is not integer, or $-d - 1$, otherwise. If the relation symbol of c is $\leq$, any solution of

c must evaluate r to an integer which is less than or equal to $-d$, which is $-\lceil d \rceil$, if d is not an integer, or $-d$, otherwise. If the relation symbol of c is $>$ or $\geq$, we can simply multiply it by -1, which results in a constraint with the relation symbol $<$ or $\leq$, respectively, and apply the previous reasoning.

In the remainder of this thesis, we obtain the *polynomial* on the left-hand side *of a (normalized) constraint* by

$$\mathrm{Pol}\colon\ \mathrm{CS} \to \mathrm{POL} : p \sim 0 \mapsto p$$

and define the *set of all polynomials in a formula* by

$$\mathrm{Pols}\colon\ \mathrm{FO}(\tau) \to \mathbb{P}(\mathrm{POL}) : \varphi \mapsto \{\mathrm{Pol}(c') |\ c' \in C_{\sim}(\varphi)\}.$$

2.4.2.2 Quantifier-free formulas with Boolean operators

In Equations (2.1-2.4) we already had a glimpse of the most important rules which we need for syntactical transformations of a formula. This implies that we do not change the semantics when using these equivalences for transformation, and therefore obtain an equivalent (and equisatisfiable) result. The last equation belongs to *De Morgan's laws*, which are, given two formulas φ_1 and φ_2, defined by the following equivalences:

$$\neg(\varphi_1 \vee \varphi_2) \equiv\ (\neg\varphi_1 \wedge \neg\varphi_2) \tag{2.8}$$

$$\neg(\varphi_1 \wedge \varphi_2) \equiv\ (\neg\varphi_1 \vee \neg\varphi_2) \tag{2.9}$$

Moreover, two subsequent negations cancel each other out, i. e.,

$$\neg(\neg(\varphi)) \equiv\ \varphi, \tag{2.10}$$

where φ is an arithmetic formula. There is one more rather natural set of equivalences, for which we need to define how to *invert* a relation:

$$\mathrm{inv}\colon\ \mathrm{REL} \to \mathrm{REL} : \sim \mapsto \begin{cases} \geq & \text{, if } \sim \text{ is } < \\ > & \text{, if } \sim \text{ is } \leq \\ \neq & \text{, if } \sim \text{ is } = \\ = & \text{, if } \sim \text{ is } \neq \\ < & \text{, if } \sim \text{ is } \geq \\ \leq & \text{, otherwise.} \end{cases}$$

Then the following equivalences hold:

$$\neg(p \sim 0) \equiv\ p\ \mathrm{inv}(\sim)\ 0 \tag{2.11}$$

$$\neg(\mathrm{true}) \equiv\ \mathrm{false} \tag{2.12}$$

$$\neg(\text{false}) \equiv \text{ true} \tag{2.13}$$

Definition 17 (Negation normal form) *A quantifier-free formula φ in* negation normal form *(NNF) is defined by the abstract grammar*

$$\varphi \quad ::= \quad \text{false} \mid \text{true} \mid b \mid \neg b \mid c \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi)$$

where b is a Boolean variable and c is a constraint.

The main characteristic of a formula in NNF is that it does not contain negations, i. e., $\neg$, except in front of a Boolean variable. We can transform any quantifier-free formula to NNF by first applying Equations (2.1-2.4) in order to obtain a formula containing only the Boolean operators $\neg$, $\wedge$ and $\vee$. Afterwards, we apply the Equations (2.8-2.13), from left to right, respectively, until reaching a fix-point, which is then a formula in NNF.

The number of transformations to be made in order to attain a formula in NNF from a given quantifier-free formula φ grows linearly as the number of Boolean operations and constraints in φ increases.

We also want to make use of the commutative, associative and distributive properties of the Boolean operators $\wedge$ and $\vee$

$$\varphi_1 \wedge \varphi_2 \equiv \varphi_2 \wedge \varphi_1 \tag{2.14}$$
$$\varphi_1 \vee \varphi_2 \equiv \varphi_2 \vee \varphi_1 \tag{2.15}$$
$$(\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \equiv \varphi_1 \wedge (\varphi_2 \wedge \varphi_3) \equiv \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \tag{2.16}$$
$$(\varphi_1 \vee \varphi_2) \vee \varphi_3 \equiv \varphi_1 \vee (\varphi_2 \vee \varphi_3) \equiv \varphi_1 \vee \varphi_2 \vee \varphi_3 \tag{2.17}$$
$$(\varphi_1 \wedge \varphi_2) \vee \varphi_3 \equiv (\varphi_1 \vee \varphi_3) \wedge (\varphi_2 \vee \varphi_3) \tag{2.18}$$
$$(\varphi_1 \vee \varphi_2) \wedge \varphi_3 \equiv (\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3) \tag{2.19}$$

with φ_1, φ_2 and φ_3 being formulas. We can therefore assume that nested conjunctions and disjunctions as given on the left-hand sides of the Equations (2.16-2.17) are always transformed to the corresponding right-hand side.

Example 8 *We can transform the formula φ_6, as introduced in Example 2 but with normalized constraints, to NNF in the following steps:*

$$\begin{aligned}
& \neg(2x_1^2x_2^2 + x_2^2 + 6x_1^2 + 3 < 0 \vee \neg(3x_1 - 2z_1 - 1 \leq 0 \ \rightarrow b_2)) \\
\overset{Eq.\ 2.8}{=}\ & \neg(2x_1^2x_2^2 + x_2^2 + 6x_1^2 + 3 < 0) \ \wedge \ \neg(\neg(3x_1 - 2z_1 - 1 \leq 0 \ \rightarrow b_2)) \\
\overset{Eq.\ 2.10}{=}\ & \neg(2x_1^2x_2^2 + x_2^2 + 6x_1^2 + 3 < 0) \ \wedge \ (3x_1 - 2z_1 - 1 \leq 0 \ \rightarrow b_2) \\
\overset{Eq.\ 2.11}{=}\ & 2x_1^2x_2^2 + x_2^2 + 6x_1^2 + 3 \geq 0 \ \wedge \ (3x_1 - 2z_1 - 1 \leq 0 \ \rightarrow b_2) \\
\overset{Eq.\ 2.3}{=}\ & 2x_1^2x_2^2 + x_2^2 + 6x_1^2 + 3 \geq 0 \ \wedge \ (\neg(3x_1 - 2z_1 - 1 \leq 0) \ \vee b_2) \\
\overset{Eq.\ 2.11}{=}\ & 2x_1^2x_2^2 + x_2^2 + 6x_1^2 + 3 \geq 0 \ \wedge \ (3x_1 - 2z_1 - 1 > 0 \ \vee b_2)
\end{aligned}$$

In the area of satisfiability checking, especially for propositional formulas, we often use a normal form which allows us to have a supplemental definition for satisfiability.

Definition 18 (Conjunctive normal form) *A quantifier-free formula φ in* conjunctive normal form *(CNF) is defined by the abstract grammar*

$$\begin{array}{rcll} \text{at} & ::= & b & | \quad c \\ l & ::= & \text{at} & | \neg\text{at} \\ ls & ::= & l \vee ls & | \, l \vee l \\ \text{cl} & ::= & (ls) & | \quad l \\ \varphi & ::= & \text{cl} \wedge \varphi & | \quad \text{cl} \end{array}$$

where at *is called an* atom *and is either a Boolean variable b or a constraint c. A* literal *l is either an atom or its negation, which we refer to as positive and negative literal, respectively. We denote the* set of all atoms *by* $\text{At} := \text{VAR}_{\mathbb{B}} \cup \text{CS}$ *and the* set of all literals *by* $\text{Lit} := \text{At} \cup \{\neg\text{at} | \ \text{at} \in \text{At}\}$. *The* inverse of a literal *is defined by*

$$\overline{\cdot} : \text{Lit} \to \text{Lit} : l \mapsto \begin{cases} \text{at} & \text{, if } l = \neg\text{at} \\ \neg\text{at} & \text{, otherwise.} \end{cases}$$

A clause cl *is either a disjunction of literals or consists of exactly one literal making the clause* unary. *We denote the* set of all clauses *by*

$$\text{Cl} := \{\bigvee_{l \in M} l | \ M \in \mathbb{P}_{<\infty}(\text{Lit})\},$$

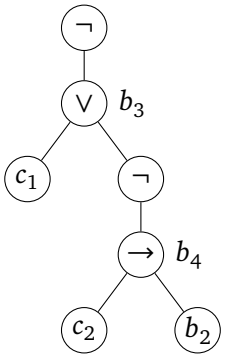
where we leave out infinite clauses and clauses, which contain a literal more than once. For a clause $\text{cl} \in \text{Cl}$*, we write that* $l \in \text{cl}$*, if* cl *contains the literal l.*

We can transform any quantifier-free formula to CNF by a procedure called *Tseitin's encoding* [Tse83]. Given a formula φ which we want to transform to CNF, the main idea of Tseitin's encoding is to introduce a fresh Boolean variable b for each Boolean operation in φ and assure the equivalence between b and the sub-formula formed by the Boolean operation. If the formula, which we want to transform to CNF, is already in NNF, it is enough to assure that b implies this sub-formula instead. Note that the result of Tseitin's encoding is equisatisfiable but not equivalent to the input formula as it introduces fresh Boolean variables.

Example 9 *The result of Example 8 is in CNF and, indeed, we can use the Equations (2.1-2.4, 2.8-2.10, 2.14-2.19) in order to syntactically transform a formula to CNF. However, the number of transformations to be made this way, in order to obtain a formula in CNF from a given quantifier-free formula φ, may grow exponentially as the number of Boolean operations in φ increases.*

Alternatively, we can use Tseitin's encoding to transform φ_6 to CNF. We assume that there are no consecutive negations, as they can easily be cancelled out by the use of Equation (2.10)*. Then, it is sufficient to introduce a fresh Boolean variable b for the Boolean operations, which are not a negation.*

The formula can be represented by the following parse tree, where $c_1 := 2x_1^2x_2^2 + x_2^2 + 6x_1^2 + 3 < 0$, $c_2 := 3x_1 - 2z_1 - 1 \leq 0$ *and* b_3 *and* b_4 *are Boolean variables, which Tseitin's encoding introduces for the two Boolean operations, which are not negations.*



Tseitin's encoding transforms the formula in the following steps to CNF:

$$
\begin{aligned}
& \neg(c_1 \vee \neg(c_2 \to b_2)) \\
= \quad & \neg b_3 \wedge (b_3 \leftrightarrow (c_1 \vee \neg b_4)) \wedge (b_4 \leftrightarrow (c_2 \to b_2)) \\
\overset{Eq.\ (2.2)}{=} \quad & \neg b_3 \wedge (b_3 \to (c_1 \vee \neg b_4)) \wedge ((c_1 \vee \neg b_4) \to b_3) \\
& \wedge (b_4 \to (c_2 \to b_2)) \wedge ((c_2 \to b_2) \to b_4) \\
\overset{Eq.\ (2.3)}{=} \quad & \neg b_3 \wedge (\neg b_3 \vee (c_1 \vee \neg b_4)) \wedge (\neg(c_1 \vee \neg b_4) \vee b_3) \\
& \wedge (\neg b_4 \vee (\neg c_2 \vee b_2)) \wedge (\neg(\neg c_2 \vee b_2) \vee b_4) \\
\overset{Eq.\ (2.8)}{=} \quad & \neg b_3 \wedge (\neg b_3 \vee (c_1 \vee \neg b_4)) \wedge ((\neg c_1 \wedge b_4) \vee b_3) \\
& \wedge (\neg b_4 \vee (\neg c_2 \vee b_2)) \wedge ((c_2 \wedge \neg b_2) \vee b_4) \\
\overset{Eq.\ (2.18)}{=} \quad & \neg b_3 \wedge (\neg b_3 \vee (c_1 \vee \neg b_4)) \wedge ((\neg c_1 \vee b_3) \wedge (b_4 \vee b_3)) \\
& \wedge (\neg b_4 \vee (\neg c_2 \vee b_2)) \wedge ((c_2 \vee b_4) \wedge (\neg b_2 \vee b_4)) \\
\overset{Eq.\ (2.16\text{–}2.17)}{=} \quad & \neg b_3 \wedge (\neg b_3 \vee c_1 \vee \neg b_4) \wedge (\neg c_1 \vee b_3) \wedge (b_4 \vee b_3) \\
& \wedge (\neg b_4 \vee \neg c_2 \vee b_2) \wedge (c_2 \vee b_4) \wedge (\neg b_2 \vee b_4)
\end{aligned}
$$

Given a clause cl and a (partial) assignment α for cl, we call cl *satisfied (under the assignment* α*)* if at least one literal l in cl is *satisfied (under the assignment* α*)*, i. e., α defines a value for l and $[\![l]\!]^\alpha = \text{true}$. Note that cl can be satisfied even if some variables in Vars(cl) are not assigned by α. If α is a full assignment for cl and $[\![\text{cl}]\!]^\alpha = \text{false}$, we call cl *conflicting (under the assignment* α*)*, which only happens, if for all literals l in cl it holds that $[\![l]\!]^\alpha = \text{false}$. If cl is not satisfied and α is defined on all but one variable in Vars(cl), we call cl *unit (under the assignment* α*)*. The search for a satisfying assignment for a formula in CNF can be achieved by finding an assignment for the formula, such that in each clause at least one literal is satisfied. State-of-the-art algorithms for checking the satisfiability of propositional formulas, as introduced in Section 2.5, use this fact and, for this reason, transform their input to CNF.

It is possible to apply equivalence transformations based on the Equations (2.1-2.4, 2.8-2.19) to a formula, which is in CNF or NNF, resulting in a formula which is still in CNF or NNF, respectively.

Therefore, neither the CNF nor the NNF of a given formula is unique.

2.4.2.3 Quantified formulas with Boolean operators

In Section 2.3.1 we already introduced an assumption on quantified formulas, which is that if a variable v is bound by a quantifier $Qv.\psi$ ($Q \in \{\exists, \forall\}$) within the formula φ, it holds, w. l. o. g., that v occurs in φ only within ψ. We can achieve this normal form by simply substituting such a quantified variable v in $Qv.\psi$ by a fresh variable, if v also occurs outside of ψ.

If we assume that a formula as defined by the grammar of Definition 4 fulfills this criterion, we can make use of the equivalences

$$\begin{array}{lcl} \neg(\exists v.\varphi) & \equiv & \forall v.(\neg\varphi) \\ \neg(\forall v.\varphi) & \equiv & \exists v.(\neg\varphi) \\ (Qv.\varphi \wedge \psi) & \equiv & (Qv.(\varphi \wedge \psi)) \\ (\varphi \wedge Qv.\psi) & \equiv & (Qv.(\varphi \wedge \psi)) \end{array} \tag{2.20}$$

with v being a variable, φ and ψ being formulas and $Q \in \{\exists, \forall\}$. Applying these equivalences from left to right, respectively, to a given formula until reaching a fix-point attains the following normal form.

Definition 19 (Prenex normal form) *A formula φ in* prenex normal form *(PNF) is defined by the abstract grammar*

$$\psi \quad ::= \quad \varphi \mid \exists v.\psi \mid \forall v.\psi$$

where v is a variable and φ is a quantifier-free formula.

If the PNF of a formula ψ contains only existential quantifiers, i. e., is of the form $\exists v_1 .. \exists v_n.\varphi$ with φ being a quantifier-free formula, ψ is satisfiable/valid if and only if φ is satisfiable/valid.

Similar to formulas in CNF and NNF, we can apply equivalence transformations to a formula in PNF such that we obtain a formula which is still in PNF. Hence, there is in general also no unique PNF of a given formula.

2.5 SAT solving

If we only consider propositional formulas, this is seemingly a huge limitation compared to general formulas with arithmetic constraints and quantifiers as defined in Section 2.3. However, checking propositional formulas for satisfiability, which we refer to as *SAT solving*, has been very successfully applied to industry and research in recent decades. The reason for this story of success is twofold.

Firstly, we can encode a vast quantity of diverse problems into propositional logic. In [MS08] and [CES+09] a wide range of applications for SAT solving is listed. In some of them, such as computer aided design of electronic circuits [Lar92] or model-checking of finite-state systems [BCCZ99], it is often natural to use propositional logic. In other applications, such as software

verification, program termination analysis and planning, it might be necessary to abstract from the real problem in order to achieve a propositional formula. Due to the high complexity of these problems, an abstraction can be crucial for finding a practical solution. Unfortunately, it can be a tedious or even impossible task to break down the problem to an encoding in propositional logic. In the field of SAT modulo theories (SMT) solving, which is introduced in Section 2.6, we allow more general formulas as input and check their satisfiability by an automatic and usually lazy encoding to propositional logic which is passed interactively to an internal SAT solver. Hence, SMT solving opens the door for further problems for SAT solving.

The second reason for the success of SAT solving is clearly the tremendous progress in improving the performance of SAT solvers, which we have observed in the last two decades. It makes SAT solving available for problem instances of real applications with up to ten million variables and clauses [JBRS12]. These instances are solved within a few minutes or often even only a few seconds, which is remarkable if we consider that checking the satisfiability of propositional formulas was one of the first problems proven to be NP complete [Coo71]. The reason for this performance lies in the nature of the problem instances of real applications. Compared to randomly generated problems, they possess certain structures which are strongly exploited by modern SAT solvers.

In the following we describe the main ideas of the algorithm which is used in almost all state-of-the-art SAT solvers. This algorithm is based on *conflict-driven clause learning* (CDCL), which extends the *DPLL algorithm*, introduced in 1962 by Martin Davis, Hilary Putnam, George Logemann and Donald W. Loveland [DLL62], by non-chronological backtracking and learning. The main algorithm and its sub-procedures, which we explain in the following, are based on `MiniSat` [ES04], since we use its implementation as a basis for the SAT solver within our own SMT solver, which we introduce in Chapter 3. We chose `MiniSat` as it is a compact and extensible open-source software, which nowadays still forms the basis of some of the best performing SAT solvers and SMT solvers. For a broad overview and further details on SAT solving we refer to [Han09]

2.5.1 Data structures and sub-procedures

Algorithm 1 describes a procedure to check a given propositional formula for satisfiability with a CDCL-based implementation similar to the one of `MiniSat`. During runtime it manipulates certain data structures, which do not directly appear in the pseudo code.

`clauses`: This is the set of clauses in the CNF of the input formula.

`learneds`: These are the clauses, which we learn after the conflict analysis.

`assigns`: It stores the assignments of Boolean constants to Boolean variables. We store the variable assignment along with the *decision level* in which it took place and its *antecedent*. The antecedent is either the clause we used via unit propagation in order to obtain the

variable assignment or $\bot$, if the variable was assigned owing to a *decision* (the terms "decision", "decision level" and "antecedent" are explained in detail in the remainder of this section).

`activities`: Activities of Boolean variables, which are used to decide to which unassigned Boolean variable we assign a Boolean constant next.

These data structures are not only essential for the understanding of CDCL but also play an important role at some points later in this thesis. There are further data structures defined in `MiniSat`, which we do not introduce here, such as the watch lists which help to find unit and conflicting clauses efficiently. For more details we refer to [ES04].

In the following we first explain the sub-procedures, which are used in Algorithm 1.

decide(): If an unassigned variable exists, this procedure chooses one with the highest activity according to `activities` and assigns to it a heuristically determined Boolean constant. We call this choice a *decision* and the variable assignment is stored in `assigns` with antecedent $\bot$. Afterwards, this procedure returns true. If, however, all variables are already assigned, this procedure just returns false. Note that we start a new *decision level* just before each decision and say that a variable was *assigned in the i-th decision level*, if i decision levels were already started.

propagate(): As long as no clause (neither in `clauses` nor in `learneds`) is conflicting under the currently found partial assignment in `assigns` and a unit clause cl exists, this procedure assigns to the unassigned variable b in cl a Boolean constant such that cl is satisfied. We call this process *Boolean constraint propagation* (BCP). If during the BCP a conflicting clause is detected, this procedure returns the pointer to this clause, otherwise, if BCP stops since no unit clause exists, it returns `null`.

addClause(clause cl, bool learned): If the given clause cl is not unary, this procedure adds cl either to `clauses`, if the second argument learned is false, or otherwise to `learneds`. If cl is unit, we assign the yet unassigned variable such that cl is satisfied. If, otherwise, cl is conflicting, we try to resolve this conflict:

- The literals in cl were assigned at different decision levels. Then, we *backtrack* to the second highest decision level i, that is we undo all assignment which were made in a decision level j with $j > i$. Afterwards we return i.
- All literals in cl were assigned at the same decision level i with $i > 0$. Then, we backtrack to decision level 0 and return 0.
- All literals in cl were assigned at the decision level 0. This means that we cannot resolve the conflict and return -1.

analyzeConflict(clausePointer confl): Assume that cl is the clause to which confl points. The procedure analyzeConflict, which we refer to as the *conflict analysis*, calculates and returns

the *conflict clause* $\text{cl}' = \text{analyze}(\text{cl})$, where

$$\text{analyze}\colon \text{Cl} \to \text{Cl} : \text{cl} \mapsto \begin{cases} \text{cl} & \text{, if cl is asserting} \\ \text{analyze}(\text{bRes}(\text{antecedent}(b), \text{cl}, b)) & \text{, otherwise} \end{cases}$$

with

- b being the last variable, which was assigned in cl,
- a clause being *asserting* if it contains exactly one literal whose variable has been assigned in the current decision level and all literals are assigned to false,
- $\text{antecedent}\colon \text{VAR}_{\mathbb{B}} \to \text{Cl}$ being defined for variables whose values were implied by propagation and it determines the clause, which was during propagation unit and implied the assignment for the given Boolean variable, and
- the *binary resolution* of two clauses $\text{cl} = (l_1 \vee .. \vee l_n \vee \neg b)$ and $\text{cl}' = (l'_1 \vee .. \vee l'_m \vee b)$ being defined by

$$\text{bRes}\colon \text{Cl} \times \text{Cl} \times \text{VAR}_{\mathbb{B}} \to \text{Cl} : (\text{cl}, \text{cl}', b) \mapsto (l_1 \vee .. \vee l_n \vee l'_1 \vee .. \vee l'_m).$$

As we have used $\text{cl} = \text{antecedent}(b)$ during propagation as a unit clause to assign false to b, cl contains the literal $\neg b$. We furthermore know that cl' contains b. Therefore, the result of $\text{bRes}(\text{cl}, \text{cl}', b)$ does not contain b. This way we cancel out exactly one Boolean variable of the current decision level with each recursive invocation of analyze(..) and always reach an asserting clause in a finite number of recursive invocations. In this procedure we also update the activities (in `activities`) of the variables, which took part in the conflict analysis. According to the *variable state independent decaying sum* (VSIDS) decision heuristic [MMZ+01], we increment the activities of these variables by a value and increment this value afterwards. This ensures that the most recent conflicts influence the activities of the variables the most.

simplify(): This procedure assumes that we are currently in decision level 0 and can therefore (optionally) simplify the clauses in `clauses` and `learneds` regarding the assignments of decision level 0. Note that these assignments are directly implied by unary clauses or the propagation before any decision has been made. A valid simplification would be to remove all clauses which are satisfied and after this we could also remove all literals, which are assigned but not satisfied, from the clauses.

forget(): This procedure removes clauses from `learneds` according to some heuristics. Usually we do not remove clauses with only two literals or clauses which recently took part in the conflict analysis. This procedure is called if the number of learned clauses exceeds a certain threshold. If this is the case, the threshold is increased by some factor, which ensures the completeness of CDCL.

Algorithm 1 The CDCL-based SAT solving algorithm similar to the implementation of `MiniSat`.

```
check(propositional formula φ)
begin
 1:   // initialize current decision level
 2:   dl := 0
 3:   // add clauses in CNF of φ
 4:   for each clause cl in CNF of φ do
 5:     if addClause(cl , false) = −1 then return unsat      // conflicting unary clause added
 6:   end for
 7:   // start search for satisfying assignment
 8:   while true do
 9:     confl :=propagate()                                  // apply unit propagation
10:     if confl ≠ null then
11:       if dl = 0 then return unsat                        // conflict cannot be resolved
12:       // create conflict clause and update activities
13:       cl := analyzeConflict(confl)
14:       // add clause, backtrack to its second highest decision level and store it in dl
15:       dl := addClause(cl, true)
16:     else
17:       if dl = 0 then simplify()                          // try to simplify considered clauses
18:       if "enough clauses learned" then forget()          // try to forget learned clauses
19:       if "enough conflicts occurred" then restart()      // backtrack to decision level 0
20:       // assign a Boolean constant to an unassigned variable with highest activity
21:       dl := dl + 1
22:       if decide() = false then
23:         return sat                                       // all variables are assigned
24:       end if
25:     end while
end
```

restart(): This procedure is called periodically, such that the number of detected conflicts between two consecutive restarts eventually increases, which ensures the completeness of CDCL. If it is invoked, it backtracks to decision level 0.

2.5.2 Main algorithm

Algorithm 1 first initializes a variable representing the index of the current decision level (Line 2). Then it stores all non-unary clauses in the CNF of the input formula to `clauses` (Line 5). If a conflict is detected during adding these clauses, the algorithm returns `unsat`.

The main loop of Algorithm 1 first applies BCP (Line 9). If it results in a conflict (Line 10), we check whether the current decision level is 0 and the conflict cannot be resolved for this reason (Line 11). In this case the algorithm returns `unsat`. Note that it is possible that we do not detect a conflict at decision level 0 before BCP has been used (e. g., if we add the clauses $(\neg b_1 \vee \neg b_2)$, (b_1) and (b_2) in this order). Otherwise, a conflict clause is calculated via conflict analysis and

added to the learned clauses with the procedure addClause.

If BCP does not lead to a conflict, we increment the stored current decision level (Line 21) and try to assign a Boolean constant to an unassigned variable (Line 22). If all variables are already assigned, we have a satisfying assignment (in `assigns`) for the input formula and return `sat` (Line 23). There are three optimizations, which we apply under certain conditions, before we decide to which variable we assign a Boolean constant next. Firstly, we try to simplify the considered clauses (in `clauses` and `learneds`) if the current decision level is 0 (Line 17). Secondly, if the number of learned clauses exceeds a certain threshold, we heuristically forget some of them and increase the threshold (Line 18). The last optimization is to periodically backtrack to decision level 0 (restart), where we ensure that the number of conflicts between two consecutive restarts eventually increases (Line 19).

2.5.3 Correctness and completeness

Theorem 1 *Given a propositional formula φ as input, Algorithm 1 always terminates with* `sat` *if φ is satisfiable and with* `unsat` *otherwise.*

Proof 1 Correctness: If the algorithm returns `sat`, all variables are assigned to a Boolean constant and no clause, neither in `clauses` nor in `learneds`, is conflicting as we would otherwise have detected a conflicting clause during propagation. The clauses in `clauses` together with the unary clauses, which we directly assigned at decision level 0, form the CNF of the input formula φ and they are, therefore, equisatisfiable to φ. We conclude that the satisfying assignment in `assigns` for the clauses in `clauses` implies that φ is indeed satisfiable.

If the algorithm returns `unsat` in Line 5, then there exists a conflicting clause at decision level 0. We can repeatedly apply binary resolution as it is done in the conflict analysis, but this time until the resulting clause does not contain any variable from the current decision level. As we are in decision level 0 the resulting clause is empty and forms a contradiction. We obtain this contradiction by applying binary resolution, which is sound and complete [DP60], to the clauses in `clauses` and `learneds`, and all clauses in `learneds` were derived by applying binary resolution to the clauses in `clauses`. This means, that we can infer a conflict from the clauses in the CNF of φ, which forms a proof of the unsatisfiability of φ.

Completeness: In order to prove the completeness of Algorithm 1 we must show that we eventually leave the main loop, which starts at (Line 8). For this purpose we consider the current partial assignments $\alpha^i \in \text{partialAssigns}(\varphi)$ at decision level i for each $i \in \{0,..,\text{dl}\}$ and let α^j be the empty assignment for $j > \text{dl}$ ($j \in \mathbb{N}$). We define a partial order on the sequences $\alpha_i = (\text{dl}_i, \alpha_i^1, .., \alpha_i^{\text{dl}_i})$, such that for two sequences α_1, α_2 it holds that $\alpha_1 < \alpha_2$ if there exists a number $k \in \{1,..,\text{dl}_1\}$ such that α_1^k is an extension of α_2^k ($\alpha_1^k \neq \alpha_2^k$) and for $i \in \{1,..,k-1\}$ it holds that $\alpha_1^i = \alpha_2^i$. If we ignore the sub-procedures forget() and restart(),

it holds that the corresponding sequence α to the current partial assignment at the start of the main loop of Algorithm 1 is in this order greater than the corresponding sequence α_{i+1} to the current partial assignment at the end of the main loop, i. e., $\alpha_{i+1} < \alpha_i$. As |partialAssigns(φ)| is finite, there is no infinite sequence of assignments in partialAssigns(φ) which is decreasing in our partial order and, therefore, the algorithm must terminate. It remains to show that (1) $\alpha_{i+1} < \alpha_i$ and (2) that Algorithm 1 is also complete if we do not ignore forget() and restart().

1. As the propagation at Line 9 only extends α_i in the current decision level, it holds for α_{i+1} that $\alpha_{i+1} < \alpha_i$. If we have a conflict, we backtrack at Line 15 to the decision level dl and extend the assignment by assigning one more variable in the decision level dl. Therefore, the α_{i+1} is in all decision levels before dl equal to α_i, but contains one more assignment at decision level dl, which implies that $\alpha_{i+1} < \alpha_i$. If no conflict occurred during propagation, we assign to a not yet assigned variable a Boolean constant in a new decision level, therefore, it still holds that $\alpha_{i+1} < \alpha_i$ as α_{i+1} is equal to α_i in all decision levels but the new one where it consists of one assignment instead of none (as in α_i). Note that simplify() does not affect the current assignment but only removes clauses which do not affect the propagation, as they are satisfied in decision level 0.
2. The sub-procedure restart() backtracks to decision level 0 and, hence, α_{i+1} is then not greater in our order than α_i. However, it is ensured that the number of detected conflicts between two consecutive restarts eventually increases. Therefore, this number is high enough at some point, such that the aforementioned reasoning for the completeness of this algorithm applies. The sub-procedure forget() does not affect the current partial assignment, but it removes learned clauses which were derived by resolution from the original clause set. Thus, clause learning only speeds up the search but does not affect completeness. □

2.6 SMT solving

SAT solvers, as introduced in Section 2.5, can determine the satisfiability of propositional formulas. Due to the fantastic performance of SAT solvers on industrial examples and the resulting upcoming success, the question arises how to broaden their field of application. *Satisfiability modulo theories* (SMT) formulas are Boolean combinations of not only Boolean variables, but also *theory predicates*. In Section 2.3, we have already seen an example of theory predicates: arithmetic constraints. In general, SMT formulas can contain theory predicates constraining arrays, bit-vectors or uninterpreted variables/predicates and function symbols, to name but a few possible theories. This thesis only concerns quantifier-free nonlinear real and integer arithmetic formulas with Boolean variables as SMT formulas.

2.6.1 Applications

SMT solving has a rapidly growing and already wide field of application. A very prominent area of research, where SMT solving is widely used, is model checking [BK08]. For instance, we can use SMT solvers for bounded model checking on hybrid automata, which are defined in [ACH+95]. A more recent example forms parametric probabilistic model-checking [12]. Besides model checking, SMT solving has also proven beneficial in further techniques which are used for program analysis. To name but a few, it has been applied in static program checking [FLL+02], test-case generation [PVL11], model-based testing [Pel13], run-time analysis [DLT16] and termination analysis [GBE+14]. SMT solving is even used in order to automatically improve a program [SOE14] or directly synthesize a program [SGF10]. If we transfer the techniques from program analysis, it is also possible to find deep bugs and security vulnerabilities using symbolic execution [PC13]. Another field of research, which profits from SMT solvers, is automated theorem proving, such as the work in [Lei13]. As a consequence, SMT solvers are already integrated as a component in the famous theorem prover Isabelle/HOL [BBP13]. SMT solving also takes place in recent developments in scheduling and planning. For instance, work has been done on rotating workforce scheduling [Erk13], many-core scheduling [TPGM14] and resource-constrained project scheduling [ABP+11]. As an example for planning, SMT solvers are used in planning problems with mixed and continuous change over time [BGMG15] or integrated task and motion planning [NPM+14].

2.6.2 Checking first-order formulas for satisfiability: State-of-the-art

Research in the area of solving quantifier-free first-order formulas for satisfiability has been carried out long before SMT solving emerged. For instance, an algorithm based on congruence closure for solving conjunctions of *equations with uninterpreted variables/predicates and function symbols* for satisfiability was published in 1980 in [NO80]. Procedures to solve quantifier-free arithmetic formulas without multiplication in polynomials but with uninterpreted predicate and function symbols was also presented around this time in [Sho79][Sho84].

For linear real-arithmetic formulas, work was carried out much earlier. The *Fourier-Motzkin variable elimination* procedure, which dates back to 1826, was published in a work of Jean-Baptiste Joseph Fourier [Fou26] and also independently invented by Theodore Motzkin. It can detect the satisfiability of a conjunction of linear real-arithmetic constraints. However, its complexity is double exponential in the number of variables, which the formula to solve contains, and therefore has a limited applicability. A second method, which can solve these kind of formulas is the *simplex method* [Dan63]. Even though its complexity is still single exponential in the number of variables in the given formula, it is the state-of-the-art procedure used nowadays for these kind of problems. This is due to the fact that the worst case complexity only occurs with very artificial examples. In practice, the performance of simplex is rather comparable to a linear complexity in the number of the formula's variables. The *ellipsoid method* [Kha80] can detect the satisfiability of such

conjunctions even with a polynomial worst complexity. However, in practice it tends to be slower than the simplex method.

For the satisfiability check of nonlinear real-arithmetic formulas, there are several incomplete as well as complete procedures available. For instance, the *virtual substitution* (VS), which we introduce in greater detail in Section 2.7 and is dealt with for the most part of this thesis, is restricted with respect to the degree of the variables in the formula, which we want to check for satisfiability. But on the other hand, we can also use this method for quantified real-arithmetic formulas. The same holds for the *cylindrical algebraic decomposition* [Col75] (CAD), which is a complete procedure for nonlinear real-arithmetic formulas. However, we gain the completeness at the expense of a higher worst-case complexity, which is double exponential for the CAD instead of single exponential, as it is for the VS, in the number of variables in a quantifier-free real-arithmetic formula, if we want to check it for satisfiability. In practice, we cannot say that either the VS or the CAD solves a formula faster. It tends to be the case, that if the VS can determine the satisfiability of a formula, it achieves this in the majority of the examples, which we had at hand, faster than the CAD. Another incomplete procedure for nonlinear real-arithmetic applies a Gröbner bases [BWK93] computation, which mainly implements Buchberger's algorithm [Buc65] and can determine in some cases whether a conjunction of nonlinear equations is unsatisfiable. In comparison to these algebraic procedures, the *interval constraint propagation* (ICP) also uses numerical approaches, which often yields a better performance in practice than one of the algebraic procedures. In Section 5.4.3, we explain this method in further detail. However, it is also incomplete for nonlinear real-arithmetic formulas and cannot handle quantifiers.

Procedures for linear integer-arithmetic or linear mixed integer-real arithmetic build upon those for linear real-arithmetic. For example, the Omega test [Pug91] uses the idea of the Fourier-Motzkin variable elimination and, therefore, shares the restriction that it can only detect the satisfiability of conjunctions of arithmetic constraints. The most commonly used approach to solve these kind of formulas, however, is *branch-and-bound* [Sch86], which was originally applied on top of the simplex method. The main idea is simple: If the simplex method detects the unsatisfiability of the given formula's real relaxation then there is in particular no integer solution. This observation does not only hold for the simplex method, but any procedure for real-arithmetic, if we use it for the real relaxation of a given integer-arithmetic or mixed integer-real arithmetic formula. In the case that we find out that the real relaxation is satisfiable and we obtain a solution which maps all variables to integers, we also find a solution for the original formula. Again, we can transfer this observation to any procedure for real-arithmetic formulas. If, on the contrary, there is one integer variable z which is mapped by the solution to a value $d \in \mathbb{R} \setminus \mathbb{Z}$, we rerun the satisfiability check, but this time adding the constraint $z \leq \lfloor d \rfloor$, and if this is unsatisfiable, we rerun the satisfiability check adding $z \geq \lceil d \rceil$, instead. However, this approach, which we refer to as *branch-and-bound* [Sch86], does not always terminate. Moreover, it cannot be applied directly to any procedure for real-arithmetic. For instance, the virtual substitution constructs symbolic solutions possibly containing dedicated representatives for arbitrary small values. In order to

construct a solution from this, which maps all variables to values, we would need to take the dependencies within the virtual substitution's solution into account. In Chapter 7 we introduce an approach, which makes branch-and-bound applicable to the virtual substitution.

This brings us to nonlinear integer-arithmetic or nonlinear mixed integer-real arithmetic. In general, the determination of the satisfiability of a formula of this type is undecidable [Mat70][Mat72]. If we bound the domains of the variables in the formula, that is we assure that there is a finite upper and lower bound for each of them, which makes the variables' domains finite, the problem becomes obviously decidable. In the context of SMT solving and only for nonlinear integer-arithmetic, the most common approach, which we refer to as bit-blasting [FGM+07], uses this fact. Here we add upper and lower bounds on the variables' domains, encode these domains and the arithmetic operations upon them to propositional logic and check it with a SAT solver for satisfiability. If we find a solution, we can reconstruct an integer solution for the given integer arithmetic formula. Otherwise, we widen the added upper and lower bounds on the variables' domains and repeat the previous step. Interval constraint propagation can also be used to detect the satisfiability of a nonlinear integer-arithmetic and even nonlinear mixed integer-real arithmetic formula. As already mentioned, in Chapter 6 we introduce another solution for this which is based on the virtual substitution. This work was published together with an adaption for the cylindrical algebraic decomposition for integer arithmetic in [1].

2.6.3 The rise of SMT solving

At the beginning of the 21st century, SAT solving became extremely successful both in research and industry. In order to extend its field of application and to exploit its performance at the point where problems with a complex Boolean structure are at hand, research has been conducted towards an incorporation of SAT solvers. One of them aimed to achieve a better performing solution for an automatic satisfiability check of first-order formulas, in particular those without quantification.

One approach, which we refer to as *eager SMT solving*, encodes the given first-order formula to a propositional formula and checks it for satisfiability with a SAT solver. If this propositional formula is equisatisfiable to the encoded formula, we can directly imply the satisfiability of the encoded formula. As an example, the SPARSE *method* [BV02] encodes an arbitrary Boolean combination of equalities between two uninterpreted variables to an equisatisfiable propositional formula. A second example is introduced in [SSB02] for *difference logic*, which concerns linear arithmetic formulas, where the constraints are of the form $x_1 - x_2 - d \sim 0$, with x_1, x_2 being arithmetic variables, $d \in \mathbb{Z}$ and $\sim$ being an arbitrary relation symbol. It might also be the case that we can only imply the satisfiability of the original formula if the propositional formula resulting from the encoding is detected to be either satisfiable or unsatisfiable. For instance, the previously explained bit-blasting for nonlinear integer arithmetic uses an encoding to a propositional formula and can only imply the encoded formula's satisfiability if the propositional formula is satisfiable.

The greatest advantage of eager SMT solving is clearly that we can simply use the SAT solver

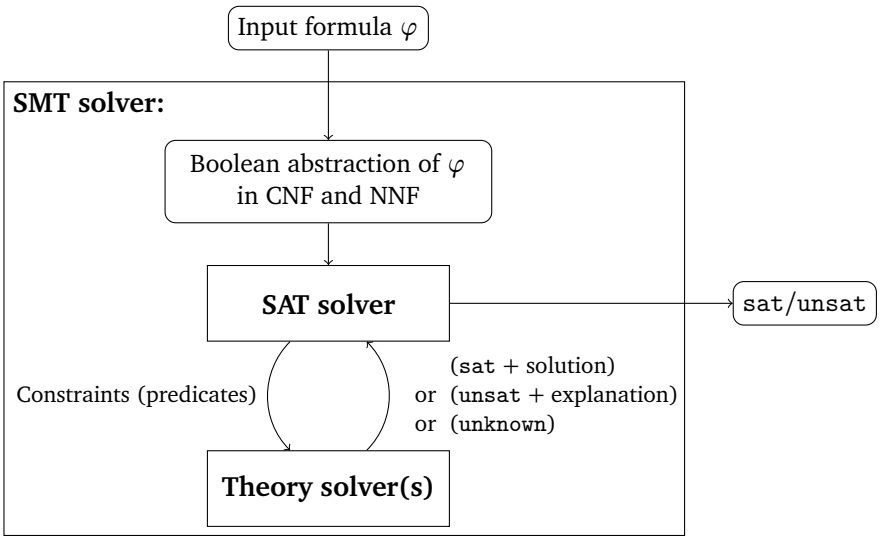


Figure 2.2: Lazy SMT solving framework.

as a black box and thereby immediately benefit from the most recent performance-improving achievements for SAT solving. Nevertheless, scientists started thinking about a tighter collaboration of the SAT solver and the existing decision procedures for first-order logics such as the ones we introduced in Section 2.6.2. The first step towards *lazy SMT solving* as it is known and widely used nowadays have been done in [BDS02] and [dMR02]. The main idea is illustrated in Figure 2.2. A lazy SMT solver consists of two main components, a SAT solver and a collection of theory solvers. The SAT solver implements the presented algorithm of Section 2.5 and each theory solver implements a procedure which checks a conjunction of predicates for satisfiability. The predicates can, for instance, be equations with uninterpreted variables and function symbols or arithmetic constraints, where we again distinguish on the one hand between linear and nonlinear constraints and on the other hand between real, integer and mixed integer-real arithmetic constraints. In Section 2.6.2 we presented procedures which would be candidates for the implementation of a theory solver.

Let us take a closer look at the functionality of a lazy SMT solver. The input formula φ of an SMT solver can be an arbitrary Boolean combination of predicates and Boolean variables. Without loss of generality, we assume that φ is an arithmetic formula as it was introduced in Definition 4. First, we transform φ into CNF. Then, we transform the result into NNF as well, which simply resolves negations in front of constraints. The resulting formula ψ is then a conjunction of clauses, where all constraints occur only in positive literals. Afterwards we create the Boolean abstraction $\psi^{\mathbb{B}}$ of ψ with $\text{abstr}_{\psi}^{\mathbb{B}}$ being the corresponding Boolean abstraction mapping. Now the SAT solver checks $\psi^{\mathbb{B}}$ for satisfiability. If $\psi^{\mathbb{B}}$ is unsatisfiable, the SMT solver returns `unsat`. Otherwise, the SAT solver has found a solution $\alpha_{\psi^{\mathbb{B}}}$. Then we let the theory solver check the conjunction of the constraints $c \in C_{\sim}(\psi)$ with $\alpha_{\psi^{\mathbb{B}}}(\text{abstr}_{\psi}^{\mathbb{B}}(c)) = \text{true}$, which are the constraints whose Boolean abstraction variables are assigned by the SAT solver's found solution to true. We refer to this

check as a *theory call*. If a theory solver detects that this conjunction is satisfiable, that is it returns `sat`, the SMT solver returns also `sat`. We also expect that the theory solver provides a solution α in this case. The SMT solver uses α and $\alpha_{\psi^{\mathbb{B}}}$ in order to construct a solution for its input φ by $\alpha_{\varphi} = \alpha \cup \{(b, \alpha_{\psi^{\mathbb{B}}}(b)) |\ b \in \text{Vars}(\varphi)\}$. If the theory solver returns `unsat`, we exclude all Boolean assignments of $\psi^{\mathbb{B}}$ which contain this *theory conflict* by adding the clause

$$\bigvee_{\substack{c \in C_{\sim}(\psi) \\ \alpha_{\psi^{\mathbb{B}}}(\text{abstr}_{\psi}^{\mathbb{B}}(c)) = \text{true}}} \neg\text{abstr}_{\psi}^{\mathbb{B}}(c)$$

to the set of clauses, which the SAT solver considers for a satisfiability check, as a learned clause. For the SAT solver (and its current assignment) this clause is conflicting. We proceed the SAT solving process at Line 13 of Algorithm 1 with this conflicting clause as explained in Section 2.5. That means that the SAT solver tries to find a satisfying assignment which does not conflict with this learned clause. This process is repeated until either a satisfying assignment of the Boolean skeleton is found such that the corresponding theory call returns `sat` or the SAT solver detects that the Boolean skeleton is unsatisfiable with respect to the learned clauses, which exclude theory conflicts.

Example 10 *Consider the formula*

$$\varphi \quad = \quad x_1 > 0 \ \wedge\ x_2 - 1 = 0 \ \wedge\ (x_1 - x_2 = 0 \ \vee\ x_2^2 < 0) \ \wedge\ (x_1 < 0 \ \vee\ x_1 - 2 \leq 0).$$

As it is already in CNF and NNF, we directly create its Boolean abstraction

$$\psi^{\mathbb{B}} \quad = \quad b_1 \ \wedge\ b_2 \ \wedge\ (b_3 \ \vee\ b_4) \ \wedge\ (b_5 \ \vee\ b_6).$$

The SAT solver then starts checking $\psi^{\mathbb{B}}$ for satisfiability. The addition of the two unary clauses yields that true *is assigned to the variables b_1 and b_2 in decision level* 0*. In decision level* 1 *we choose to assign* false *to b_3 and BCP implies that* true *has to be assigned to b_4. In decision level* 2*, we assign* false *to b_5 and BCP implies that we have to assign* true *to the last remaining unassigned variable b_6. As there is no conflicting clause, we obtain the satisfying assignment*

$$\alpha_{\psi^{\mathbb{B}}} = \{(b_1, \text{true}),\ (b_2, \text{true}),\ (b_3, \text{false}),\ (b_4, \text{true}),\ (b_5, \text{false}),\ (b_6, \text{true})\}.$$

Now we check the conjunction of constraints

$$x_1 > 0 \ \wedge\ x_2 - 1 = 0 \ \wedge\ x_2^2 < 0 \ \wedge\ x_1 - 2 \leq 0$$

with the theory solver for satisfiability. It is unsatisfiable, hence we add the clause

$$(\neg b_1 \ \vee\ \neg b_2 \ \vee\ \neg b_4 \ \vee\ \neg b_6)$$

to the SAT solver. This clause is already asserting, therefore we backtrack to decision level 1 *where BCP implies that we must assign* false *to* b_6 *and as a consequence* true *to* b_5*. The corresponding theory call for*

$$x_1 > 0 \ \wedge \ x_2 - 1 = 0 \ \wedge \ x_2^2 < 0 \ \wedge \ x_1 < 0$$

yields again `unsat`*. The SAT solver now learns the clause*

$$(\neg b_1 \ \vee \ \neg b_2 \ \vee \ \neg b_4 \ \vee \ \neg b_5).$$

Conflict analysis additionally yields the asserting clause $(\neg b_1 \vee \neg b_2 \vee \neg b_4)$*, hence we backtrack to decision level* 0 *and imply with BCP that we must assign* false *to* b_4 *and therefore* true *to* b_3*. In decision level* 1*, we choose to assign* false *to* b_5 *and due to BCP we assign* true *to* b_6*. The corresponding theory call for*

$$x_1 > 0 \ \wedge \ x_2 - 1 = 0 \ \wedge \ x_1 - x_2 = 0 \ \wedge \ x_1 - 2 \leq 0$$

yields `sat` *this time and the SMT solver returns also* `sat`*.*

The just described SMT solving framework is said to be *full lazy*, as it always constructs a full satisfying assignment of the Boolean abstraction before it makes a theory call. If we invoke the theory solver more often, for instance, every time a decision level is finished, we call it *less lazy*.

Example 11 *Considering the input formula* φ *and its Boolean abstraction* $\psi^{\mathbb{B}}$ *of Example 10 . When the SAT solver finishes decision level* 0*, its current assignment for* $\psi^{\mathbb{B}}$ *is*

$$\alpha_{\psi^{\mathbb{B}}} = \{(b_1, \text{true}), \ (b_2, \text{true})\}.$$

Less-lazy SMT solving invokes the theory solver at this point for

$$x_1 > 0 \ \wedge \ x_2 - 1 = 0,$$

which results in `sat`*. Afterwards the SAT solver finishes decision level* 1 *with*

$$\alpha_{\psi^{\mathbb{B}}} = \{(b_1, \text{true}), \ (b_2, \text{true}), \ (b_3, \text{false}), \ (b_4, \text{true})\}$$

and the corresponding theory call

$$x_1 > 0 \ \wedge \ x_2 - 1 = 0 \ \wedge \ x_2^2 < 0$$

yields this time `unsat`*. We add a clause*

$$(\neg b_1 \ \vee \ \neg b_2 \ \vee \ \neg b_4)$$

to the SAT solver in order to exclude this theory conflict. The SAT solver backtracks to decision level

0 *and propagates that we must assign* false *to* b_4 *and therefore* true *to* b_3*. This finishes decision level* 0 *and we invoke the theory solver with*

$$x_1 > 0 \ \wedge \ x_2 - 1 = 0 \ \wedge \ x_1 - x_2 = 0.$$

As this is satisfiable, the SAT solver starts a new decision level by choosing to assign false *to* b_5*. BCP implies that we have to assign* true *to* b_6*, which concludes this decision level. The corresponding theory call for*

$$x_1 > 0 \ \wedge \ x_2 - 1 = 0 \ \wedge \ x_1 - x_2 = 0 \ \wedge \ x_1 - 2 \leq 0$$

returns `sat`*, thus the SMT solver returns* `sat`*, as we have found a satisfying full assignment of* $\psi^{\mathbb{B}}$ *which is consistent with the theory.*

Comparing full-lazy and less-lazy SMT solving, we observe in Example 10 and Example 11 that the number of theory calls seems to be quite similar. Less-lazy SMT solving actually produces one more theory call here. In practice, it highly depends on the example at hand. However, compared to full-lazy SMT solving, the complexity of the theory calls with less-lazy SMT solving is reduced, as the number of constraints to be solved is smaller. Especially for less-lazy SMT solving, we also observe that a theory call often shares the majority of the constraints with the previous theory call. If the theory solver could exploit this fact, that is not just starting a satisfiability check from scratch each time it is invoked, but use the results of the previous checks and only make a minimal effort to detect the satisfiability of the currently considered conjunction of constraints, this would boost the performance of lazy SMT solving, especially of less-lazy SMT solving.

Hence, considering two consecutive theory calls, we have to remove some of the constraints from the theory solver and add some new constraints to it, after the first theory call was finished and before we invoke the second one. The ability of a theory solver to provide an interface for a belated removing of a constraint while keeping as much information, which it gained in the former theory calls, is referred to as the *backtracking ability*. Similarly, if a theory solver provides an interface to add constraints belatedly, while keeping as much information, which it gained in the former theory calls, we say that it supports *incrementality*. In [BBC+05], for instance, first results have shown the positive impact of these features for SMT solving.

The first two theory calls in Example 10 both contained the constraint $x_2^2 < 0$, which itself is already unsatisfiable. If the theory solver would have known this fact and would have been able to communicate this, we could have saved the second theory call. In [dMRS02] the authors dealt with this fact. Instead of simply returning `unsat`, we let the theory solver also return an explanation, which is a subset of the constraints in the conjunction, which it checked for satisfiability. We call this explanation *infeasible subset* (of the checked constraints). We use this explanation in order to construct the learned clause, which excludes the discovered theory conflict from future assignments of the SAT solver. Let $c_1 \wedge \ldots \wedge c_n$ be the conjunction of constraints, for which the theory solver detected that it is unsatisfiable, and the Boolean abstraction of c_i be b_i

($1 \leq i \leq n$). Until now, we learned the clause $\neg b_1 \vee \ldots \vee \neg b_n$, which excludes all assignments for the Boolean abstraction the SAT solver considers that assign true to b_i ($1 \leq i \leq n$). Now, we have an explanation in form of an infeasible subset $C \subseteq \{c_1, \ldots, c_n\}$ and, instead, we learn the clause $\bigvee_{c_i \in C} \neg b_i$, which excludes in general more assignments for the Boolean abstraction the SAT solver considers than $\neg b_1 \vee \ldots \vee \neg b_n$.

As a consequence, a theory solver which provides a small infeasible subset, if it detects that its input is unsatisfiable, improves the performance of the SMT solver. Usually, the smaller the infeasible subset is, the more we can benefit from it in the SMT solving process. Finding the smallest infeasible subset, for instance $\{x_2^2 < 0\}$ for the first theory call in Example 10, is in practice often very hard. Instead, we usually only require an infeasible subset which is *minimal*, which means, that if we remove a constraint from it, the conjunction of the remaining constraints is satisfiable. However, even minimality is sometimes difficult to achieve, so in the end we need to find a good trade-off between creating small infeasible subsets and reducing the effort we have to make in the theory solver in order to construct them. This discussion also indicates that there is often more than one infeasible subset and as long as one infeasible subset is not a subset of the other, both are valuable information for the SMT solver.

Summarizing, a theory solver has to meet three conditions for a well performing collaboration within a lazy (especially less-lazy) SMT solver:

1. Backtracking ability
2. Incrementality
3. Infeasible subset generation

A theory solver, which fulfills these three requirements, is called *SMT compliant*. There has been a lot of work in this field of research in recent years. An SMT-compliant theory solver for conjunctions of equations with uninterpreted variables and function symbols was introduced in [NO05]. The authors of [CAMN04] presented an SMT-compliant theory solver for difference logic and [DdM06] contributed an SMT compliant theory solver for linear real and integer arithmetic, which is based on an adaption of the simplex method's first phase, branch-and-bound and the construction of cutting planes. In [dMP09], the authors deal with the creation of explanations, if Gröbner bases are used in order to detect the unsatisfiability of a conjunction of nonlinear real-arithmetic equations. We contributed in [7] an SMT-compliant theory solver for nonlinear real arithmetic, which utilizes Gröbner bases. Furthermore, we introduced an SMT-compliant theory solver also for nonlinear real arithmetic based on the virtual substitution [7]. We present these ideas and further developments in Chapter 4 of this thesis.

There are further ideas of how we can improve the collaboration of the SMT solver's SAT solver and theory solver(s), such as theory propagation or theory guided decision heuristics. For more details and a good overview of many techniques used in SMT solving, we suggest [NOT06] and [Seb07].

Recent developments brought forth an even tighter interaction of decision procedures, such as those from Section 2.6.2, and SAT solving. A generalization of this idea is presented in [JBdM13] and a precise implementation for nonlinear real arithmetic based on the cylindrical algebraic decomposition was introduced in [JdM12].

2.6.4 Tools and standards (2016)

The SMT solving community has put a lot of effort into the standardization of the SMT solver's input. Thanks to the international initiative SMT-LIB, most SMT solvers support a common input language, which is specified as the SMT-LIB standard [RT03][BFT16]. This initiative also maintains a large library of benchmarks, which comprise thousands of input examples, and an annual competition among the different SMT solvers. We used these and some other benchmarks, which are briefly described in Chapter 6 and Section 7.2, for the experimental results within this thesis.

We want to conclude this section on SMT solving giving an overview of the currently available SMT solvers and other tools which can be used to check arithmetic formulas for satisfiability. In the last decade, we have been able to observe a lot of activity for linear real and integer arithmetic. The SMT solvers, which are specifically dedicated to support these logics, for instance, `CVC4`[BCD+11], `MathSAT5`[CGSS13], `OpenSMT2`[BPST10], `SMTInterpol` [CHN12], `veriT`[BCBdODF09], `Yices2`[Dut14] and `Z3`[dMB08], can solve real world problems with hundreds of variables in often only a few seconds and they achieve this with a fantastic reliability. To the best of our knowledge, all of these SMT solvers base their implementation, which checks linear real- and integer-arithmetic formulas for satisfiability, on the groundbreaking contribution of [DdM06]. In addition, they put a lot of effort into exploiting that the input formulas show certain characteristics, where it is possible to solve or simplify them by the use of preprocessing. Moreover, it is essential to use smart heuristics for the countless choices that must be made during the SMT solving process [Gri09][KBD13]. Undoubtedly, there are more ideas which contributed to the excellent performance of the named SMT solvers, such as an integration of a linear programming solver [KBT14], breaking symmetries [DFMP11][ADFO13] or the creation of better cutting planes for linear integer arithmetic [DDA09].

Long before SMT solving was invented, nonlinear real-arithmetic formulas have been of concern in computer algebra systems. For instance, `Redlog` [DS97] is capable of not only checking quantified formulas, which are arbitrary Boolean combinations of real-arithmetic constraints and Boolean variables, but it can also eliminate quantifiers/variables yielding an equisatisfiable formula. To the best of our knowledge, `Redlog` implements an optimized combination of the virtual substitution and the cylindrical algebraic decomposition. Furthermore, it uses Gröbner bases and other ideas in order to simplify a formula.

The tool `HySAT`[FHT+07] and its successor `iSAT3`[SKB13] can also check real- and integer-arithmetic formulas for satisfiability but using a closer interaction of SAT and theory solving. Their implementations are based on a close collaboration of a SAT solver and interval constraint

propagation, which differs from the previously presented approach of lazy SMT solving. As ICP is incomplete for real arithmetic (and of course also for integer arithmetic), these tools cannot always give a conclusive answer to the question for satisfiability of a given formula. However, they are also able to deal with constraints involving exponential or trigonometric functions. There are further tools, which implement ICP, such as `dReal` [GKC13] or `raSAT` [TVKO16].

The SMT solvers `CVC4`, `MiniSmt` [ZM10] and `Z3` initially checked nonlinear real-arithmetic formulas via linearizing them, which has the big advantage that their very performant engine for linear arithmetic can be employed afterwards. However, it only yields an incomplete procedure, which cannot determine the satisfiability of most of the nonlinear formulas in the SMT-LIB benchmarks. Recent developments brought forth a new approach, which is based on a close collaboration of a SAT solver and the cylindrical algebraic decomposition [JdM12]. This approach was first implemented as part of `Z3` and, in the meanwhile, it is also used in the SMT solver `Yices2`. At this moment in time, it outperforms the aforementioned approaches in the majority of the examples in the SMT-LIB benchmarks for nonlinear real arithmetic. It forms a very good example of how CDCL reasoning can be used in algebraic procedures and has already drawn interest in the computer algebra community [ÁFSW16].

Besides ICP, the most common approach to check nonlinear integer-arithmetic formulas for satisfiability is bit-blasting. It was implemented within the tool `AProVE` [GBE+14], which is primarily dedicated to an automated generation of termination and complexity proofs. In the meantime, the SMT solvers `CVC4`, `Yices2` and `Z3` also use bit-blasting for nonlinear integer-arithmetic.

2.7 Virtual substitution

Virtual (term) substitution (VS) was first introduced in 1993 as a quantifier/variable elimination procedure for linear real-arithmetic formulas [LW93]. In contrast to the Fourier Motzkin variable elimination, the first version of the VS cannot only be applied to an existentially quantified conjunction of linear real-arithmetic constraints, but even to arbitrary Boolean combinations of such constraints where each variable can be either existentially or universally quantified.

Some years later, VS was extended to a quantifier elimination procedure for nonlinear real-arithmetic formulas [Wei97]. However, it can only eliminate quantified variables whose degree is not higher than 2. Furthermore, eliminating a quantifier might increase the degree of the remaining quantified variables.

The boundary of the degree for which a quantified variable can be eliminated could be pushed further to three [Wei94] and four [GT09]. Recently, it has been shown that quantified variables of an arbitrary but bounded degree can be eliminated by an approach based on the VS [KS15]. Unfortunately, the higher the degree of the quantified variable we eliminate, the more complex is the result of this elimination step in terms of the number of atoms in the resulting formula. This is why we restrict ourselves to VS for the quadratic case as presented in [Wei97], which has

proven to be a viable procedure for various applications by its first implementation in `Redlog`.

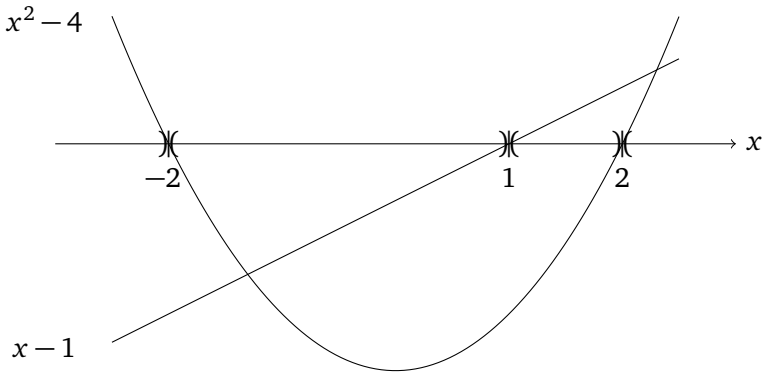
2.7.1 Constructing test candidates with side condition

Let c be a real arithmetic constraint and $A \subseteq \text{Assigns}(c)$ be a set of assignments for c which evaluate $\text{Pol}(c)$ to a value with the same sign, i. e., for all $\alpha_1, \alpha_2 \in A$ it holds that $\text{sgn}([\![\text{Pol}(c)]\!]^{\alpha_1}) = \text{sgn}([\![\text{Pol}(c)]\!]^{\alpha_2})$. Regardless of the relation in c, we know that all assignments in A are either solutions, i. e., $A \subseteq \Theta(c)$ or not, i. e., $A \cap \Theta(c) = \emptyset$. This is due to the fact that we compare $\text{Pol}(c)$ in c by some relation symbol to 0. If we consider a real arithmetic formula $\varphi^{\mathbb{R}}$ instead of a constraint, we can make the same observation. Let us partition $\text{Assigns}(\varphi^{\mathbb{R}})$ into *maximal sign invariant regions* $A_1, .., A_n$ regarding the polynomials in $\text{Pols}(\varphi^{\mathbb{R}})$, such that for all $\alpha_1, \alpha_2 \in A_i$ it holds for all $p \in \text{Pols}(\varphi^{\mathbb{R}})$ that $\text{sgn}([\![p]\!]^{\alpha_1}) = \text{sgn}([\![p]\!]^{\alpha_2})$ $(1 \leq i \leq n)$. Then A_i contains either only solutions of $\varphi^{\mathbb{R}}$, i. e., $A_i \subseteq \Theta(\varphi^{\mathbb{R}})$ or no solutions of $\varphi^{\mathbb{R}}$, i. e., $A_i \cap \Theta(\varphi^{\mathbb{R}}) = \emptyset$ $(1 \leq i \leq n)$. As a conclusion it is sufficient to check one assignment in each sign invariant region to the variables in $\varphi^{\mathbb{R}}$ in order to determine the satisfiability of $\varphi^{\mathbb{R}}$. As the number of maximal sign invariant regions is for a given finite set of polynomials always finite, we obtain the main idea of decision procedures such as the virtual substitution and the cylindrical algebraic decomposition, which construct a finite set of assignments such that each maximal sign invariant region is covered at least once.

Example 12 *Let us consider the univariate quantifier-free real-arithmetic formula*

$$\varphi^{\mathbb{R}} = (x^2 - 4 \geq 0 \ \wedge \ x - 1 < 0) \ \vee \ (x^2 - 4 = 0 \ \wedge \ x - 1 > 0)$$

The set of all polynomials in $\varphi^{\mathbb{R}}$ *is* $\text{Pols}(\varphi^{\mathbb{R}}) = \{x^2 - 4, \ x - 1\}$ *and we illustrate them in the following plot.*



We can partition Assigns($\varphi^{\mathbb{R}}$) *to the sign invariant regions*

$$\begin{aligned}
A_1 &= \{\alpha \in \text{Assigns}(\varphi^{\mathbb{R}}) |\ \alpha(x) \in (-\infty, -2)\ \} \\
A_2 &= \{\alpha \in \text{Assigns}(\varphi^{\mathbb{R}}) |\ \alpha(x) \in [-2, -2]\ \} \\
A_3 &= \{\alpha \in \text{Assigns}(\varphi^{\mathbb{R}}) |\ \alpha(x) \in (-2, 1)\ \} \\
A_4 &= \{\alpha \in \text{Assigns}(\varphi^{\mathbb{R}}) |\ \alpha(x) \in [1, 1]\ \} \\
A_5 &= \{\alpha \in \text{Assigns}(\varphi^{\mathbb{R}}) |\ \alpha(x) \in (1, 2)\ \} \\
A_6 &= \{\alpha \in \text{Assigns}(\varphi^{\mathbb{R}}) |\ \alpha(x) \in [2, 2]\ \} \\
A_7 &= \{\alpha \in \text{Assigns}(\varphi^{\mathbb{R}}) |\ \alpha(x) \in (2, \infty)\ \}
\end{aligned}$$

In the univariate case of Example 12 we observe that the values of one sign invariant region, which the assignments map to x, form either point intervals or open intervals. Moreover, the bounds of these intervals are the zeros of the polynomials in Pols($\varphi^{\mathbb{R}}$). This leads to the idea that we could use them in order to construct the finite set of assignments covering all sign invariant regions. In Example 12, the three assignments of the zeros of the polynomials in Pols($\varphi^{\mathbb{R}}$) to x, i. e., $\{\alpha \in \text{Assigns}(\varphi^{\mathbb{R}}) |\ \alpha(x) \in \{-2, 1, 2\}\}$ cover only the sign invariant regions A_2, A_4 and A_6. As the values, to which the assignments of other regions map, form open intervals, we need to find a value which is either very close to the right of (greater than) the interval's left bound or very close to the left of (less than) the interval's right bound. We decide for the former. One question remains: How close is close enough? For instance, choosing the next greater integer would lead to the additional assignments $\{\alpha \in \text{Assigns}(\varphi^{\mathbb{R}}) |\ \alpha(x) \in \{-1, 2, 3\}\}$. Now we would still not cover the sign invariant regions A_1, as its left bound is not a zero of the polynomials in Pols($\varphi^{\mathbb{R}}$) but $-\infty$, and A_5, as we did not choose a value close enough to the right of the zero of $x - 1$. Instead of considering the entire formula in order to find suitable values, we postpone this decision by introducing representatives for a sufficiently small value, which we denote by $-\infty$, and a value which is sufficiently close to the right of an open left bound d, we denote by $d + \epsilon$. This leads to the finite set of assignments $\{\alpha \in \text{Assigns}(\varphi^{\mathbb{R}}) |\ \alpha(x) \in \{-\infty, -2, -2 + \epsilon, 1, 1 + \epsilon, 2, 2 + \epsilon\}\}$ covering all sign invariant regions for some sufficiently small $-\infty$ and infinitesimal ϵ (> 0) .

The relation symbols of the constraints in $\varphi^{\mathbb{R}}$ have no influence on the sign invariant regions of the polynomials in $\varphi^{\mathbb{R}}$. However, they define the set of solutions of $\varphi^{\mathbb{R}}$. In the univariate case, as in Example 12, the assignments which satisfy a weak constraint map to values forming a set of closed intervals. For strict constraints they form open intervals. As we are interested in the satisfiability of a formula, we do not need to cover sign invariant regions which are open intervals with a left bound being a zero of a polynomial which only occurs in weak constraints. This region is satisfied if and only if the point interval, which contains this zero, is satisfied. Moreover, we do not need to consider sign invariant regions, which are point intervals that contain a zero of polynomials that only occur in strict inequalities. In Example 13 we illustrate this optimization on the formula from Example 12.

Example 13 *Consider the formula $\varphi^{\mathbb{R}}$ from Example 12. The sets of solutions of the constraints in*

$\varphi^{\mathbb{R}}$ *are*

$$\begin{aligned}
\Theta(x^2-4\geq 0) &= \{\alpha\in \text{Assigns}(\varphi^{\mathbb{R}})|\ \alpha(x)\in(-\infty,-2]\cup[2,\infty)\ \}\\
\Theta(x^2-4=0) &= \{\alpha\in \text{Assigns}(\varphi^{\mathbb{R}})|\ \alpha(x)\in[-2,-2]\cup[2,2]\ \}\\
\Theta(x-1<0) &= \{\alpha\in \text{Assigns}(\varphi^{\mathbb{R}})|\ \alpha(x)\in(-\infty,1)\ \}\\
\Theta(x-1>0) &= \{\alpha\in \text{Assigns}(\varphi^{\mathbb{R}})|\ \alpha(x)\in(1,\infty)\ \}
\end{aligned}$$

Now we can construct a finite set of assignments, such that one of them is a solution of $\varphi^{\mathbb{R}}$ *if and only if* $\varphi^{\mathbb{R}}$ *is satisfiable. For this purpose, we do not need to cover* A_3 *and* A_7*, as the polynomial* x^2-4 *occurs only in weak constraints. We also do not need to cover the sign invariant region* A_4*, as the polynomial* $x-1$ *occurs only in strict constraints. Therefore, we only check the assignments*

$$\{\alpha\in \text{Assigns}(\varphi^{\mathbb{R}})|\ \alpha(x)\in\{-\infty,-2,1+\epsilon,2\}\}$$

to find a solution of $\varphi^{\mathbb{R}}$ *for some sufficiently small* $-\infty$ *and infinitesimal* ϵ *(*>0*).*

We have seen how to construct a finite set of assignments for a given univariate real-arithmetic formula $\varphi^{\mathbb{R}}$, such that one of them is a solution of $\varphi^{\mathbb{R}}$ if and only if $\varphi^{\mathbb{R}}$ is satisfiable. This construction also guarantees us, that all assignments of this set satisfy $\varphi^{\mathbb{R}}$ if and only if $\varphi^{\mathbb{R}}$ is valid or in other words, it is satisfied by all assignments in Assigns($\varphi^{\mathbb{R}}$). We can transfer this principle to the multivariate case with n variables in order to achieve an equisatisfiable formula with $n-1$ variables. Once we have chosen a variable x we want to eliminate this way, we construct a finite set of so called *test candidates*, which form the equivalent to the values to which the constructed assignments for the univariate case map. The test candidates are obtained in almost the same way as before. For each weak constraint c_w in $\varphi^{\mathbb{R}}$, we use the zeros of Pol(c_w) in x as test candidates. For each strict constraint c_s in $\varphi^{\mathbb{R}}$, we use the zeros of Pol(c_s) in x plus an infinitesimal ϵ as test candidates. Furthermore, we use $-\infty$ as a test candidate, which, as before, represents a sufficiently small value. The main difference is that the zeros are possibly parametrized in the remaining variables and might not exist under certain conditions in the remaining variables. We also need a solution formula to determine the parametrized zeros of a polynomial in one variable and here lies the restriction of this method. For polynomials whose degree in x is higher than 4, there exists no general formula to determine its zeros in x. We even restrict ourselves to quadratic polynomials in x, as it gives us a comparably compact solution formula for the zeros in x and we have to deal with less cases. Before we get to a general definition of test candidates, we introduce an expression which emerges when using the solution formula of quadratic equations.

Definition 20 (Square root expression) *A* square root expression *is of the form*

$$\frac{p+q\sqrt{r}}{s},$$

where $p,q,r,s \in \text{POL}$. *We denote the* set of all square root expressions *by*

$$\text{SqrtEx} := \{\frac{p+q\sqrt{r}}{s} |\ p,q,r,s \in \text{POL}\}$$

and the set of all square root expressions in the variables $x_1,..,x_n$ *by*

$$\text{SqrtEx}[x_1,..,x_n] := \{\frac{p+q\sqrt{r}}{s} |\ p,q,r,s \in \text{POL}[x_1,..,x_n]\}.$$

Given a quadratic equation $p_1x^2+p_2x+p_3=0$ in the variable x, that is $p_1,p_2,p_3 \in \text{POL}$ and $x \notin \text{Vars}(p_1) \cup \text{Vars}(p_2) \cup \text{Vars}(p_3)$, the solution formula for x in $p_1x^2+p_2x+p_3=0$ considers the following three cases:

$$x_0 = \quad -\frac{p_3}{p_2} \qquad \text{, if } p_1 = 0 \ \wedge\ p_2 \neq 0 \tag{2.21}$$

$$x_1 = \quad \frac{-p_2+\sqrt{p_2^2-4p_1p_3}}{2p_1} \qquad \text{, if } p_1 \neq 0 \ \wedge\ p_2^2-4p_1p_3 \geq 0 \tag{2.22}$$

$$x_2 = \quad \frac{-p_2-\sqrt{p_2^2-4p_1p_3}}{2p_1} \qquad \text{, if } p_1 \neq 0 \ \wedge\ p_2^2-4p_1p_3 \geq 0 \tag{2.23}$$

Note that any real number is solution for x, if $p_1=0$, $p_2=0$ and $p_3=0$, therefore this case is covered, in particular, by $-\infty$. Therefore, we can summarize the appearance of the symbolic zero of x in a polynomial, which is quadratic in x, by a square root expression $\frac{p+q\sqrt{r}}{s}$, where

- for Equation (2.21), it holds that $p=-p_3$, $q=0$, $r=1$ and $s=p_2$,
- for Equation (2.22), it holds that $p=-p_2$, $q=1$, $r=p_2^2-4p_1p_3$ and $s=2p_1$, and
- for Equation (2.23), it holds that $p=-p_2$, $q=-1$, $r=p_2^2-4p_1p_3$ and $s=2p_1$.

This gives us the opportunity to generalize test candidates, taking into account that they can be $-\infty$ and supplemented by an infinitesimal ϵ.

Definition 21 (Construction of test candidates) *The* set of all test candidates *is defined by*

$$\text{TCS} \ :=\ \text{SqrtEx} \cup \{t+\epsilon |\ t \in \text{SqrtEx}\} \cup \{-\infty\}$$

The set of test candidates for an arithmetic variable in a constraint*, which is quadratic in x, is defined by*

$$\text{tcs}\colon\ \text{VAR}_{\mathbb{R},\mathbb{Z}} \times \text{CS} \to \text{TCS}:$$

$$(x,\ p_1x^2+p_2x+p_3 \sim 0) \mapsto \begin{cases} \{-\infty,\ -\frac{p_3}{p_2},\ \frac{-p_2\pm\sqrt{p_2^2-4p_1p_3}}{2p_1}\} & \text{, if } \sim \text{ is weak} \\ \{-\infty,\ -\frac{p_3}{p_2}+\epsilon,\ \frac{-p_2\pm\sqrt{p_2^2-4p_1p_3}}{2p_1}+\epsilon\} & \text{, otherwise,} \end{cases}$$

where $p_1,p_2,p_3 \in \text{POL}$ *and* $x \notin \text{Vars}(p_1) \cup \text{Vars}(p_2) \cup \text{Vars}(p_3)$.

The side condition of a test candidate *is defined by*

$$\mathrm{sc}\colon\ \mathrm{TCS} \to \mathrm{FO}(\tau) : t \mapsto \begin{cases} \mathrm{sc}(t') & \text{, if } t = t' + \epsilon \\ s \neq 0 \ \wedge\ r \geq 0 & \text{, if } t = \frac{p+q\sqrt{r}}{s} \\ \mathrm{true} & \text{, otherwise } (t = -\infty) \end{cases}$$

where p, q, r and s are polynomials and t' is a test candidate not containing ϵ.

The set of test candidates for an arithmetic variable in an arithmetic formula *where the variable occurs at most quadratically is defined by*

$$\mathrm{tcs}\colon\ (\mathrm{VAR}_{\mathbb{R},\mathbb{Z}} \times \mathrm{FO}(\tau)) \to \mathrm{TCS} : (x, \varphi) \mapsto \bigcup_{c \in C_{\sim}(\varphi)} \mathrm{tcs}(x, c).$$

The side condition of a test candidate ensures that the zero, which we used to create the test candidate, indeed exists. The side condition of the candidate $-\infty$ is valid, as it does not relate to a zero.

Compared to the univariate case, the zeros used for the construction of the test candidates can now contain variables. Therefore, their existence, exact location and order in an illustration such as in Example 12 is not clear. Fortunately, this does not affect the argumentation we made for the univariate case. No matter which real values we assign to the remaining variables in the symbolic zero, which is used in a test candidate, it still forms the left bound of one of the sign invariant regions. If we take into account all zeros of the polynomials of all constraints in a formula and also these zeros plus an infinitesimal ϵ and a sufficiently small value $-\infty$, we ensure that for any instantiation of the variables in the symbolic zeros all sign invariant regions for the variable, for which we created the test candidates, are covered. Therefore, a given real arithmetic formula $\varphi^{\mathbb{R}}$, which contains the variable x, is satisfiable if and only if there is one test candidate t for x in $\varphi^{\mathbb{R}}$, such that, on condition that $x = t$ and t's side condition $\mathrm{sc}(t)$ holds, $\varphi^{\mathbb{R}}$ is satisfiable. Furthermore, $\varphi^{\mathbb{R}}$ is valid if and only if all test candidates t for x in $\varphi^{\mathbb{R}}$, make $\varphi^{\mathbb{R}}$, on condition that $x = t$ and $\mathrm{sc}(t)$ holds, satisfiable.

2.7.2 Substituting variables by test candidates virtually

Given a real arithmetic formula $\varphi^{\mathbb{R}}$, we would usually substitute all occurrences of a real arithmetic variable x in $\varphi^{\mathbb{R}}$ by the test candidate t at hand, in order to ensure that $x = t$. As t can be $-\infty$ or contain quotients, square roots of polynomials or infinitesimals ϵ, the substitution result would not necessarily be an arithmetic formula. Hence, we do not know how to construct test candidates for the remaining variables in this result. Therefore, the virtual substitution provides rules which specify an equisatisfiable formula $\psi^{\mathbb{R}}$ to $\varphi^{\mathbb{R}}$, on condition that $x = t$ and $\mathrm{Vars}(\psi^{\mathbb{R}}) = \mathrm{Vars}(\varphi^{\mathbb{R}}) \setminus \{x\}$ (x does not occur in $\psi^{\mathbb{R}}$ and no further variables are introduced in $\psi^{\mathbb{R}}$).

Definition 22 (Virtual substitution) *The* virtual substitution of a real arithmetic variable by a test candidate in a real arithmetic formula *is defined by*

$$\cdot\,[\,\cdot\,/\!/\,\cdot\,]\colon\ \mathrm{FO}(\tau)\times\mathrm{TCS}\times\mathrm{VAR}_{\mathbb{R},\mathbb{Z}}\to\mathrm{FO}(\tau).$$

For a given real arithmetic formula $\varphi^{\mathbb{R}}$, a test candidate t and a real arithmetic variable x, we obtain $\varphi^{\mathbb{R}}[t/\!/x]$ from the virtual substitution rules*, which can be found in [Wei97]. A more detailed listing of these rules together with an alternative version which tries to avoid the growth of the remaining variable's degrees can be found in [Cor10].*

Assume that we have a constraint c, a variable $x \in \mathrm{Vars}(c)$ and a test candidate t which does not contain x. The virtual substitution rules specify an equivalent quantifier-free real-arithmetic formula to $c[t/\!/x]$ which depends on the form of t and the form of c. For the test candidate, we distinguish between the four cases that t is

1. $-\infty$,
2. a square root expression $\frac{p+q\sqrt{r}}{s}$ with $r = 1$ (is actually a fraction of two polynomials),
3. a square root expression $\frac{p+q\sqrt{r}}{s}$ with $r \neq 1$ (actually contains a square root) or
4. contains an infinitesimal.

For the constraint, the rules depend on its relation symbol and the degree of x in c. In order to obtain $\varphi^{\mathbb{R}}[t/\!/x]$ for a real arithmetic formula $\varphi^{\mathbb{R}}$, we simply replace each constraint c' in $\varphi^{\mathbb{R}}$ by the quantifier-free real-arithmetic formula $c'[t/\!/x]$, which is specified in the virtual substitution rules. The result is indeed an equisatisfiable real-arithmetic formula to $\varphi^{\mathbb{R}}$, on condition that $x = t$, such that $x \notin \mathrm{Vars}(\varphi^{\mathbb{R}}[t/\!/x]) \subseteq \mathrm{Vars}(\varphi^{\mathbb{R}})$. In the following we show two cases of the virtual substitution rules, which can be seen as the most significant ones in order to understand the concept.

Example 14 *Let t be a test candidate for x with a square root, i. e., $t = \frac{p_1+q_1\sqrt{r}}{s_1}$, which we refer to in the following as a* square root expression*. Moreover, assume we want to substitute t for x in the constraint $p = 0$. For the virtual substitution of a test candidate being a square root expression, the degree of x in p does not matter. If we replace all occurrences of x by t in p, we can transform the result to a square root expression $\frac{p_2+q_2\sqrt{r}}{s_2}$ (p_2, q_2 and s_2 are real arithmetic polynomials) with the same radicand r. This transformation is possible, since the result of an addition or multiplication of two square root expressions with the same radicand can be transformed to a square root expression with this radicand:*

$$\frac{p_3+q_3\sqrt{r}}{s_3} + \frac{p_4+q_4\sqrt{r}}{s_4} = \frac{s_4(p_3+q_3\sqrt{r})+s_3(p_4+q_4\sqrt{r})}{s_3s_4} = \frac{s_4p_3+s_3p_4+(s_4q_3+s_3q_4)\sqrt{r}}{s_3s_4}$$

$$\frac{p_3+q_3\sqrt{r}}{s_3} \cdot \frac{p_4+q_4\sqrt{r}}{s_4} = \frac{(p_3+q_3\sqrt{r})(p_4+q_4\sqrt{r})}{s_3s_4} = \frac{p_3p_4+q_3q_4r+(q_3p_4+p_3q_4)\sqrt{r}}{s_3s_4}$$

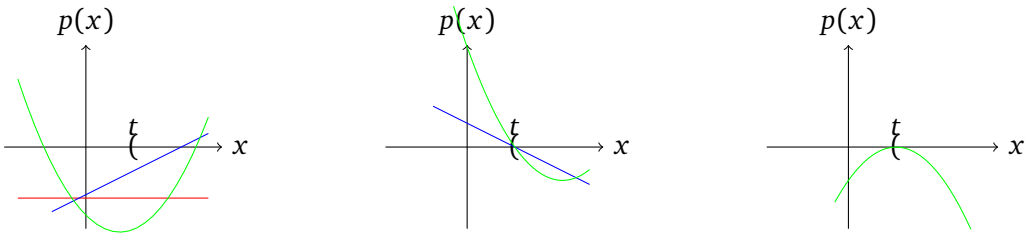
The equation $\frac{p_2+q_2\sqrt{r}}{s_2} = 0$ *holds if and only if* $p_2 + q_2\sqrt{r} = 0$*, or equivalently, if and only if either both* p_2 *and* q_2 *are equal to* 0*, or they have different signs but the same absolute value, i. e.,* $|p_2| = |q_2\sqrt{r}|$*. This can be expressed with the quantifier-free real-arithmetic formula*

$$p_2 q_2 \leq 0 \ \wedge \ p_2^2 - q_2^2 r = 0.$$

Example 15 *If we substitute the variable* x *by a test candidate* $t + \epsilon$ *(so* t *is not* $-\infty$ *and it does not contain an infinitesimal) virtually in an inequality* $p < 0$*, such that* x *occurs at most quadratic in* p*, it results in the quantifier-free real-arithmetic formula*

$$\underbrace{p[t//x] < 0}_{\text{Case 1}} \vee \underbrace{(p[t//x] = 0 \wedge p'[t//x] < 0)}_{\text{Case 2}} \vee \underbrace{(p[t//x] = 0 \wedge p'[t//x] = 0 \wedge p''[t//x] < 0)}_{\text{Case 3}}$$

where p' *and* p'' *are the first and second derivative of* p *for* x*, respectively. Here, Case* 1 *states that the polynomial* p *evaluates to a negative value for some assignment for* p*, if* x *has the value represented by* t*. This implies that* $p < 0[t + \epsilon//x]$ *must hold, as due to the density of* $\mathbb{R}$ *there must be a value in the right neighborhood of* t *such that, if* x *has this value, for any assignment for* p*, it still evaluates to a negative value. In Case* 2 *and* 3*, we assume that for some assignment of* p *it evaluates to zero, if* x *has the value represented by* t*. Both cases ensure that* p *is decreasing, if we move from* $x = t$ *to the positive* x*-direction. Then, the density of* $\mathbb{R}$ *implies again that there must be a value in the right neighborhood of* t*, where* p *is negative. The three cases for univariate polynomials can be visualized as follows:*



Case 1 *Case 2* *Case 3*

Note that we still need to apply virtual substitutions in the resulting constraints $p[t//x] = 0$*,* $p'[t//x] = 0$*,* $p[t//x] < 0$*,* $p'[t//x] < 0$*, and* $p''[t//x] < 0$*. However, it does not involve an infinitesimal anymore.*

2.7.3 Quantifier elimination with the virtual substitution

We can now formalize how to eliminate a quantified variable with the virtual substitution.

Theorem 2 *Let* $\varphi^{\mathbb{R}}$ *be a quantifier-free real-arithmetic formula with* $x \in \text{Vars}(\varphi^{\mathbb{R}})$*, which occurs at most quadratic in* $\varphi^{\mathbb{R}}$*. Then the following two equivalences hold:*

$$\exists x.\, \varphi^{\mathbb{R}} \quad \Leftrightarrow \quad \bigvee_{t \in \text{tcs}(x, \varphi^{\mathbb{R}})} (\varphi^{\mathbb{R}}[t//x] \ \wedge \ \text{sc}(t)) \tag{2.24}$$

$$\forall x.\ \varphi^{\mathbb{R}} \quad \Leftrightarrow \quad \bigwedge_{t \in \mathrm{tcs}(x,\varphi^{\mathbb{R}})} (\mathrm{sc}(t) \rightarrow \varphi^{\mathbb{R}}[t//x]) \tag{2.25}$$

Proof 2 The proof of Equation (2.24) can be found in [Wei97] and is based on the aforementioned explanations. Equation (2.25) can be implied by Equation (2.24) as follows:

$$\begin{array}{lll} \forall x.\varphi^{\mathbb{R}} & \Leftrightarrow & \neg\exists x.\neg\varphi^{\mathbb{R}} \\ & \overset{2.24}{\Leftrightarrow} & \neg(\bigvee_{t\in\mathrm{tcs}(x,\neg\varphi^{\mathbb{R}})}(\neg\varphi^{\mathbb{R}}[t//x] \wedge \mathrm{sc}(t))) \\ & \Leftrightarrow & \bigwedge_{t\in\mathrm{tcs}(x,\neg\varphi^{\mathbb{R}})}(\varphi^{\mathbb{R}}[t//x] \vee \neg\mathrm{sc}(t)) \\ & \overset{C_\sim(\varphi^{\mathbb{R}}) = C_\sim(\neg\varphi^{\mathbb{R}})}{\Leftrightarrow} & \bigwedge_{t\in\mathrm{tcs}(x,\varphi^{\mathbb{R}})}(\mathrm{sc}(t) \rightarrow \varphi^{\mathbb{R}}[t//x]) \end{array}$$

□

CHAPTER 3

SMT-RAT: Strategic and Parallel Toolbox for SMT Solving

In late 2012, there was no SMT solver which could check nonlinear real-arithmetic formulas for satisfiability in general. To the best of our knowledge, CVC4, MiniSmt and Z3 provided only incomplete procedures based on a linearization of the input formula at that time, which enabled them to solve some nonlinear real-arithmetic formulas. Moreover, there were also computer algebra systems which could handle arbitrary Boolean combinations of nonlinear real-arithmetic constraints. For instance, Redlog can even solve (or eliminate variables in) quantified nonlinear real-arithmetic formulas, which includes checking purely existentially quantified formulas for satisfiability. However, its implementation did not apply modern lazy SMT solving. Moreover, it could not be used directly as an SMT compliant theory solver, as it supported neither incrementality nor a backtracking ability nor could it provide infeasible subsets.

Given these circumstances and due to the wide range of procedures for nonlinear real arithmetic, as summarized in Section 2.6.2, we initiated the project SMT-RAT [6][2] which aimed to develop a collection of SMT-compliant theory solvers for nonlinear real arithmetic. Inspired by the ideas from [dMP13], we set ourselves the objective to also seek the ability to combine the single implementations of this collection with respect to a user defined strategy. Moreover, we wanted sub-strategies within such a strategy to be allowed to be run in parallel. This does not only make sense as most computers use multi-core processors nowadays, but also as it is often unclear which procedure promises a better performance for which characteristics of a nonlinear real-arithmetic formula, when checking it for satisfiability. For instance, we know that a standard implementation based on Gröbner bases can only (sometimes) detect that a conjunction of nonlinear real-arithmetic equations is unsatisfiable. Hence, we would not want to use such an implementation if no equations are involved, but if there are equations, it might be of use to utilize Gröbner bases or it might yield only additional overhead. Therefore, running the two sub-strategies in parallel for this case, one with the theory solver based on Gröbner bases and

one without, adopts the performance of the better choice while accepting some overhead for the use of multithreading.

The support for a strategic combination of different mathematically complex procedures puts high requirements on the design of SMT-RAT, especially as we need a clearly modular framework where the implementations of these procedures share a common interface. We also stressed that this interface is not only very general but also kept simple, so that it is easy to extend SMT-RAT by implementations of further procedures. During the design phase of SMT-RAT, it became clear that we do not need to restrict this interface to be only shared by SMT-compliant theory solver implementations. Instead, we further generalize it to be shared by an implementation of any procedure which

- checks a conjunction of formulas for satisfiability[1],
- is able to remove some of the formulas in this conjunction while keeping as much information from previous satisfiability checks as possible,
- is able to add formulas to this conjunction while keeping as much information from previous satisfiability checks as possible and
- can provide (preferably small) infeasible subsets of the formulas in the conjunction once the satisfiability check yields that it is unsatisfiable.

This sounds very familiar and, indeed, it is very close to the definition of the three requirements on SMT-compliant theory solvers. The only difference is that we allow conjunctions of arbitrary formulas instead of just constraints. Therefore, we use the term *SMT compliant* from now on for any procedure-implementation, which fulfills the above requirements.

In the context of this thesis, we only deal with quantifier-free arithmetic formulas, but the concepts which we introduce in this chapter can be extended to any quantifier-free first order formula.

3.1 Modules

In SMT-RAT, we call an SMT compliant procedure-implementation a *module*. A module m has an initially empty *(set of) received formulas* $C_{rcv}(m)$. The main function of a module is check(bool `full`), which either decides whether the conjunction of the received formulas in $C_{rcv}(m)$ is satisfiable or not, returning `sat` or `unsat`, respectively, or returns `unknown`. If the function's argument `full` is set to false, the underlying procedure of m is allowed to omit hard obstacles during solving at the cost of returning `unknown` in more cases.

We can manipulate $C_{rcv}(m)$ by adding a formula φ with add(φ) to $C_{rcv}(m)$ and removing a formula φ with remove(φ) from $C_{rcv}(m)$. The method add works ideally incrementally, that

[1] It is not a restriction to consider a conjunction of formulas instead of a general formula, as we consider a "conjunction of one formula" as the formula itself.

is we add the formula while keeping as much information from the last satisfiability check of m as possible. Analogously, remove should ideally keep as much information from the last satisfiability check of m as possible (backtracking ability). As $C_{rcv}(m)$ is usually only slightly changed between two consecutive check calls, their performance can be significantly improved if m provides incrementality and a backtracking ability. However, it is not a strict requirement.

In case m determines the unsatisfiability of $C_{rcv}(m)$ via the procedure check, it has to return an *infeasible subset* $C_{inf}(m) \subseteq C_{rcv}(m)$. Note that m should at least return the trivial infeasible subset, which is $C_{rcv}(m)$, but m's caller might benefit from a smaller infeasible subset or subsets.

Moreover, a module can specify *lemmas*, which are valid formulas. They encapsulate information which can be extracted from a module's internal state and collected by m's caller. In the course of this chapter we further specify how lemmas can be used.

Furthermore, a module can ask other modules for the satisfiability of its *(set of) passed formulas* denoted by $C_{pas}(m)$, if it invokes the procedure runBackends(bool full). It thereby delegates work to modules that may be more suitable than m itself for solving the conjunction of the formulas in $C_{pas}(m)$ for satisfiability. Which modules are used by this procedure is unknown to m and is specified by a user-defined strategy.

In the course of this section we mean the conjunction of the received formulas by received formula and analogously we mean the conjunction of the passed formulas by passed formula.

3.2 Strategy

SMT-RAT supports user-defined strategies for the composition of modules. We first define such a strategy and then explain how to interpret it.

Definition 23 (SMT-RAT strategy) *An* SMT-RAT strategy $(V, E, \mathrm{Ml}, \mathrm{Cd}, \mathrm{Pr})$ *is a directed tree* (V, E) *where the vertices and edges are labeled. The vertices are labeled by*

- *SMT-RAT modules, which is denoted by*

$$\mathrm{Ml}\colon\ V \to \textit{SMT-RAT modules},$$

and the edges are labeled by

- *Boolean combinations of formula properties, which we refer to as* conditions *and denote by*

$$\mathrm{Cd}\colon\ E \to (\mathrm{FO}(\tau) \to \mathbb{B}),$$

- *and* priority values*, which is denoted by*

$$\mathrm{Pr}\colon\ E \to \mathbb{N}.$$

The priority values of the edges in E are pairwise different, that is it holds that

$$\forall e_1, e_2 \in E.\ (e_1 \neq e_2 \rightarrow \Pr(e_1) \neq \Pr(e_2)).$$

By formula properties we mean propositions about, for instance, the Boolean structure of the formula, about the constraints in it, e. g., whether it contains equations, or about the polynomials in the formula, e. g., whether they are linear or not.

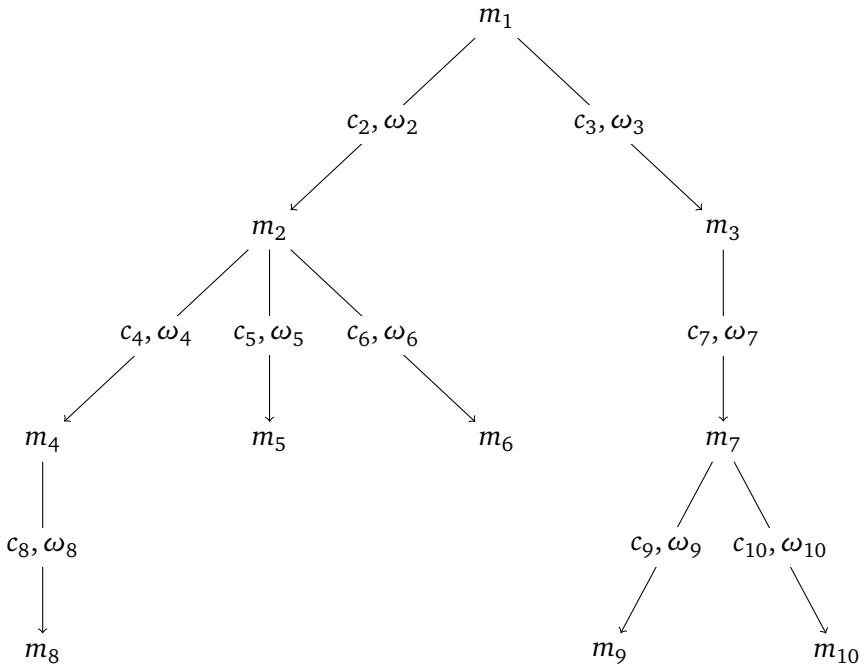


Figure 3.1: An abstract example of an SMT-RAT strategy with the modules $m_1, \ldots, m_{10}$, the conditions $c_2, \ldots, c_{10}$ and the priority values $\omega_2, \ldots, \omega_{10}$.

Consider the abstract illustration of an example for an SMT-RAT strategy in Figure 3.1. If we check a formula φ_1 for satisfiability according to this strategy, we solve it by the use of the SMT-RAT module m_1, which is the label of the root of the strategy. This means, that we pass φ_1 to m_1 via its procedure add and then invoke m_1's procedure check. While m_1 performs the satisfiability check, it might need to know the satisfiability of a formula φ_2 in order to continue. At this point we make use of the strategy again. There are two edges from m_1 leading to the modules m_2 and m_3. If the formula φ_1 fulfills the condition c_2, but not c_3, we use m_2 to check φ_2 for satisfiability in the same way we have used m_1 for φ_1. If, otherwise, φ_2 fulfills the condition c_3, but not c_2, we use m_3 instead. If φ_2 fulfills both of these conditions, we use both modules in parallel, if using another thread is allowed.[2] Otherwise, we use the successor with the higher priority (indicated by a lower priority value), that is we use m_2, if $\omega_2 < \omega_3$, and m_3, if $\omega_3 < \omega_2$. Note that by Definition 23 it holds that $\omega_2 \neq \omega_3$. The last possibility is that both conditions are

[2]Usually, we either disable a parallel mode, which implies that only one thread can be used during the whole solving process, or we allow it to use as many threads as there are cores in the processor on the machine where it is run.

not fulfilled for φ_2. In this case, the satisfiability for φ_2 is `unknown`, which might provoke that m_1 is not able to detect the satisfiability of φ_1.

The root module in a strategy can also be a module, which does nothing else but copying its set of received formulas to its set of passed formulas and invoking the backends. The backends are in that case the roots of different strategies, which are invoked to check our input formula φ_1 for satisfiability if the condition in the label of the edge to this strategy is fulfilled by φ_1.

We will see some examples later which point out that an `SMT-RAT` strategy can define a sophisticated framework for a satisfiability check by the use of this simple modular mechanism. Throughout this thesis we do not specify the priority values, if it does not lead to confusion. Moreover, if we do not specify a condition along an edge of an `SMT-RAT` strategy, this indicates that the condition is true, that means it is fulfilled by any formula's properties.

3.3 Manager

The *manager* holds the strategy $T = (V, E, \mathrm{Ml}, \mathrm{Cd}, \mathrm{Pr})$ and controls the aforementioned utilization of T for an input formula C_{input}. Initially, the manager calls the procedure check of the module m_r, being the root of T, with $C_{rcv}(m_r) = C_{input}$. Whenever a module $m = \mathrm{Ml}(v)$ calls the procedure runBackends for a $v \in V$, the manager adds a *solving task* $(\mathrm{Pr}((v, v')),\ m,\ m')$ to its priority queue Q of solving tasks (ordered by the increasing priorities), for every edge $(v,\ v') \in E$ with $m' = \mathrm{Ml}(v')$ such that $\mathrm{Cd}((v, v'))$ holds for $C_{pas}(m)$. If a core of the processor on the machine on which `SMT-RAT` is executed is available, the first solving task of Q (the one with the highest priority) is started and popped from Q. The manager thereby starts the procedure check of m' with $C_{rcv}(m') = C_{pas}(m)$ and passes the result back to m.

This means that we obtain an implementation which checks formulas for satisfiability, if we define a strategy and instantiate a manager with this strategy.

3.4 Procedures implemented as modules

The procedures, which are implemented as modules, form the heart of `SMT-RAT`. Currently, we can classify them into three groups, *preprocessing modules*, *SMT solving modules* and *theory solving modules*.

3.4.1 Preprocessing modules

Preprocessing modules implement lightweight procedures, which can detect the satisfiability of the given formula but only in some cases. Otherwise, it passes through its input formula to its backends and invokes them to perform the satisfiability check. In many cases, preprocessing modules can also simplify the formula beforehand, that is it invokes the backends on a equisatisfiable simplified formula instead.

In the following, we present two examples for a preprocessing of a given arithmetic formula φ.

- Replace each equation in $c \in C_{\sim}(\varphi)$ by

$$\bigvee_{p \in \mathcal{F}_{\mathrm{Pol}(c)}} p = 0,$$

  if the factorization $\mathcal{F}_{\mathrm{Pol}(c)}$ of c's left-hand side $\mathrm{Pol}(c)$ is not trivial. It would also be possible to resolve constraints, which are not equations and where the left-hand side's factorization is not trivial. For instance, if the relation symbol is $\neq$, we would need to use a conjunction instead of a disjunction. For the other relation symbols, we need to consider many combinations instead. Experimental results indicate that only resolving equations with a factorable left-hand side yields better performances for the involved satisfiability check.

- If $\varphi = (\varphi' \wedge dx + p = 0)$ with x being a real-valued variable, $d \in \mathbb{Q}$ and p being a polynomial not containing x, we can simplify φ to $\varphi'[-\frac{p}{d}/x]$.[3] If a sub-formula ψ of φ has this form, i. e., $\psi = (\psi' \wedge dx + p = 0)$, we can replace ψ by

$$\psi'[-\frac{p}{d}/x] \wedge dx + p = 0,$$

  which means that we cannot drop the equation in this case.

In practice, we often observe that we can apply one kind of preprocessing several times, each time resulting in a simpler formula. This does also hold, if we apply different kinds of preprocessing. Therefore, we follow the idea of a repeated application of a series of different kinds of preprocessing, either until we reach a fixed-point, that is the formula to preprocess does not change by applying this series of different kinds of preprocessing, or the number of repetitions reaches a given bound. In `SMT-RAT` we have implemented this as a *fixed-point preprocessing module*, which we denote by $\texttt{Module}_{FP}$.

3.4.2 SMT solving modules

SMT solving modules employ a SAT solver. This implies that they can, in particular, check a propositional formula for satisfiability. In general, an SMT solving module checks formulas for satisfiability by the use of an interaction with its SAT solver and backends according to the strategy.

In `SMT-RAT`, there is only one SMT solving module, which is called $\texttt{Module}_{SAT}$. It basically applies the concept of a less-lazy SMT solver and additionally utilizes lemmas that are specified by the backends (theory solvers). Algorithm 2 implements the procedure check of a $\texttt{Module}_{SAT}$ m. This algorithm forms an extension of the satisfiability check of a CDCL SAT solver as given in Algorithm 1. Therefore, m has also the members of a CDCL SAT solver, for instance,

[3] We did not define the substitution by a non-integer constant, but it is analog to the substitution of Definition 5. We achieve a polynomial as defined in Definition 4, if we multiply the result by some positive constant afterwards (as explained after Definition 11).

- `clauses` representing the set of clauses in (the Boolean abstraction of the NNF of) the input formula's CNF,
- `learneds`, which is the set of learned clauses after conflict analysis, and
- the current assignments `assigns` of Boolean constants to Boolean variables.

First, we set the current decision level to 0 and initialize the flag excl_assign by false (Line 1-2). This flag indicates that the backends could not detect the satisfiability of a conjunction of constraints, which needed to be determined in order to find out the satisfiability of *m*'s received formula. Afterwards, we calculate the NNF of the CNF of the received formula resulting in φ, and store its Boolean abstraction in $\varphi^{\mathbb{B}}$ (Line 3). The corresponding Boolean abstraction mapping is $\text{abstr}_{\varphi}^{\mathbb{B}}$ (Line 4). Then we add each clause in the Boolean abstraction of $\varphi^{\mathbb{B}}$ to `clauses`.

The main loop of Algorithm 2 has the same structure as the main loop of Algorithm 1. First, we apply Boolean constraint propagation (Line 9). In contrast to Algorithm 1, we additionally check whether the currently found partial assignment of φ's Boolean skeleton is also consistent with the theory, if no Boolean conflict occurred. As `clauses` considers the clauses in the Boolean abstraction of the received formula's CNF after transforming it also to NNF, we only need to check the conjunction of those constraints for satisfiability, where `assigns` assigns true to the constraint's Boolean abstraction (Line 12-13). We store these constraints in the passed formula and invoke the interface runBackends in order to determine the satisfiability of their conjunction. If it returns `unsat`, we add a clause to `learneds` for each infeasible subset of the backends. These clauses exclude the theory conflicts, which correspond to these infeasible subsets, from the search for a satisfying assignment just as explained in Section 2.6.3 (Line 14). From the resulting conflicting clauses, we choose one, which is conflicting at the lowest decision level, and store it in confl.

Afterwards, if either a Boolean or a theory conflict occurred, we analyze the conflict, backtrack in order to resolve it and assign the asserting literal just as it is done in Algorithm 1. The only difference is that we return `unknown` instead of `unsat`, if the conflict cannot be resolved and, additionally, the flag excl_assign is set to true. If no conflict occurred, Algorithm 2 handles the backend's lemmas. We distinguish between two types of lemmas, *urgent lemmas* and *final lemmas*. Urgent lemmas are learned each time before we make a decision and start the next decision level (Line 27). Note that it is also possible to learn urgent lemmas directly after a theory call instead, and then jump back to Boolean constraint propagation in Line 9. This would be repeated until either no more urgent lemmas are learned or a conflict is reached.

If no urgent lemmas are learned and some variables in $\varphi^{\mathbb{B}}$ are still unassigned, we make a decision in exactly the same way as it is done in Algorithm 1. Otherwise, we must now also check whether the last theory call confirmed theory consistency. If this is the case, we can return `sat` (Line 35). Otherwise, we try to learn final lemmas. If the backends do not provide any final lemmas, we exclude the partial assignment corresponding to the last theory call and set the flag, which indicates that this has ever happened, to true. Note that we exclude this partial assignment

Algorithm 2 The procedure check of a $\texttt{Module}_{SAT}$ m, which adapts CDCL-based SAT solving as introduced in Algorithm 1.

```
check(bool full)
begin
 1:   dl := 0
 2:   excl_assign := false
 3:   φ^𝔹 := Boolean abstraction of φ, which is the NNF of the CNF of the received formula
 4:   abstr_φ^𝔹 := corresponding Boolean abstraction mapping
 5:   for each clause cl in φ^𝔹 do
 6:     if addClause(cl, false) = −1 then return unsat        // conflicting unary clause added
 7:   end for
 8:   while true do
 9:     confl := propagate()                                  // apply unit propagation
10:     theory_call := sat
11:     if confl = null then                                  // check if assigns is consistent with theory
12:       C_pas(m) := {c ∈ C_∼(φ)| assigns(abstr_φ^𝔹(c)) = true}
13:       theory_call := runBackends(full)
14:       if theory_call = unsat then confl := addInfeasibleSubsets()
15:     end if
16:     if confl ≠ null then
17:       backtrack to largest decision level in confl
18:       if dl = 0 then                                      // conflict cannot be resolved
19:         if excl_assign then return unknown
20:         else return unsat
21:       else
22:         cl := analyzeConflict(confl)                      // create conflict clause
23:         dl := addClause(cl, true)                         // add conflict clause
24:       end if
25:     else
26:       ...                                 // simplify, restart, forget .. see Algorithm 1  (Line 17-19)
27:       if addUrgentLemmas() = false then                   // no urgent lemmas were added
28:         dl := dl + 1
29:         if decide() = false then                          // all variables are assigned
30:           if theory_call = unknown then
31:             if addFinalLemmas() = false then              // no final lemmas were added
32:               excl_assign := true
33:               excludeCurrentTheoryCall()
34:             end if
35:           else return sat
36:         end if
37:       end if
38:     end if
39:   end while
end
```

in the same way as we exclude theory conflicts, that is we learn a clause which specifies that for at least one of the constraints in the last theory call, we must assign false to its Boolean abstraction.

3.4.3 Branching lemmas

A good example for lemmas, which are provided by a theory solving module, are *branching lemmas*. This happens, for instance, if we invoke a theory solving module m in order to detect the satisfiability of a conjunction of mixed integer-real arithmetic constraints $\varphi = c_1 \wedge \ldots \wedge c_n$. Here, m might try to find out the satisfiability of φ's real relaxation first. If it is unsatisfiable, then φ is unsatisfiable and we can return `unsat`. If it is satisfiable, but the found solution assigns a value $d \in \mathbb{R} \setminus \mathbb{Z}$ to an integer-valued variable $z \in \text{Vars}(\varphi)$, m can provide a branching lemma of the form

$$(c'_1 \wedge \ldots \wedge c'_k) \rightarrow (z \leq \lfloor d \rfloor \vee z \geq \lceil d \rceil). \tag{3.1}$$

It demands the splitting of the domain of z at d, under the condition that the *branching premise* $c'_1 \wedge \ldots \wedge c'_k$ with $\{c'_1, \ldots, c'_k\} \subseteq \{c_1, \ldots, c_n\}$ holds. Additionally, the theory solving module can specify which of the two branches it prefers to start with. We call the Boolean abstraction $(\neg b_{c'_1} \vee \ldots \vee \neg b_{c'_k} \vee b_{z \leq \lfloor d \rfloor} \vee b_{z \geq \lceil d \rceil})$ of the branching lemma in Eq. 3.1 a *branching clause* and its last two (possibly fresh) literals *branching literals*. In our context branching lemmas are always final lemmas.

When a branching clause is added to the set of clauses considered by a $\texttt{Module}_{SAT}$, one of the branching literals (the one that was *not* preferred by the theory solving module) will be assigned false (thus, if the branching premise is true, BCP will assign true to the preferred branching literal; this way we prevent both branching literals becoming true, which would result in a theory conflict). Afterwards, we handle the branching clause just as any learned clause and benefit from the usual reasoning and learning process, which yields the best performance according to our experience.

To prevent unnecessary branchings, we always assign the value false to branching literals as decision variables. Remember that only constraints with true abstraction variables will be passed to the theory solving module. This means that only branching clauses, whose premise is true, play a role in the theory, and for those clauses only one of the branching literals.

3.4.4 Theory solving modules

Theory solving modules can only check conjunctions of constraints for satisfiability. If one of the formulas in its received formula is not a constraint or a constraint which is not supported by the procedure which the module implements, it usually returns `unknown`. A theory solving module can also return `unsat` in this case, if the conjunction of the constraints in the module's received formula, which are supported by the procedure that the module implements, is found to be unsatisfiable. In many cases, theory solving modules implement incomplete procedures. In the case that such a module gets into the position where it is not able to proceed with its satisfiability

check unless it knows the satisfiability of a certain formula, the module can consult its backends in the aforementioned process.

Module$_{Simplex}$ This module implements the SMT-compliant simplex method equipped with branch-and-bound and cutting-plane procedures as presented in [DdM06]. We apply it on the linear constraints of a conjunction φ of mixed integer-real arithmetic constraints. If this module detects that the real relaxation $\varphi_+^{\mathbb{R}}$ of the linear part φ_+ of the problem is unsatisfiable, it returns `unsat`. If it finds a solution for $\varphi_+^{\mathbb{R}}$ that also satisfies φ, it returns `sat`. If it does not satisfy φ but the real-relaxation $\varphi_{\mathbb{R}}$ of the input conjunction, which also means the relaxed nonlinear constraints, this module creates a branching lemma and returns `unknown`. Otherwise, it forwards φ to another theory solving module, and passes back the result and, if constructed, also the infeasible subsets and the lemmas to its caller.

Module$_{GB}$ The implementation of this module is based on Gröbner bases computation as presented in [4] and is also SMT compliant. It takes the polynomials of the equations $\varphi_=^{\mathbb{R}}$ in the real relaxation $\varphi^{\mathbb{R}}$ of its received formula φ and applies Buchberger's algorithm. If the calculated Gröbner base contains a constant ($\neq 0$) or more generally speaking a positive or negative definite polynomial, we know that $\varphi_=^{\mathbb{R}}$ has no complex solution and therefore neither $\varphi^{\mathbb{R}}$ nor φ have a solution. Otherwise, we push φ to this module's passed formula and invoke backends, just as a Module$_{Simplex}$ would do. Depending on its settings, a Module$_{GB}$ passes a simplified version of φ to its backends.

Module$_{\text{VS}}$ This module implements an SMT-compliant version of the virtual substitution as we present in Chapter 4.

Module$_{\text{CAD}}$ The implementation of this module is based on the cylindrical algebraic decomposition (CAD) and is also SMT-compliant. It can check a conjunction of mixed integer-real arithmetic constraints for satisfiability. As the CAD is a complete procedure for real arithmetic formulas, this module never invokes backends. Some ideas of its implementation are presented in [5]. Moreover, [1] introduces how the CAD can be adapted in order to find integer solutions where it also creates branching lemmas.

Module$_{ICP}$ This module uses interval constraint propagation (ICP) similar to the one presented in [GGI$^+$10], and lifts splitting decisions and contractions as lemmas to a preceding Module$_{SAT}$. We give a detailed example for ICP in Section 5.4.3. Given this module's received formula φ, it basically narrows down an over-approximation of φ's solution space until either reaching an empty over-approximation, in which case this module returns `unsat`, or reaching a certain precision. In the latter case, this module tries to guess a solution and, if it succeeds, it returns `sat`. Otherwise, it invokes the backends in order to check φ, equipped with the found over-approximation, for satisfiability. In Section 5.4 we show how an implementation, which is based on the virtual substitution, can take advantage of this additional

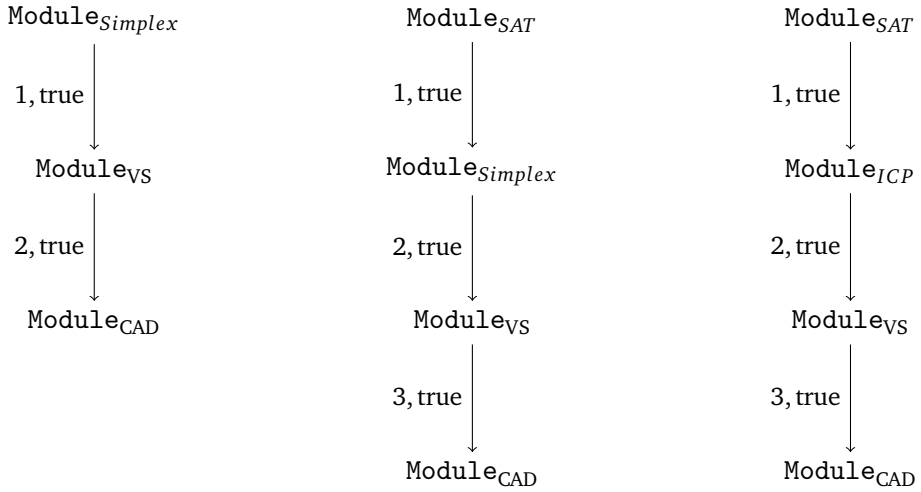


Figure 3.2: The SMT-RAT strategy on the left assembles different theory solving modules resulting in a composed theory solver, and the two SMT-RAT strategies on the right form SMT solvers for quantifier-free arithmetic formulas.

information. Moreover, we presented in [5] that a CAD-based procedure can also make use of an additional interval-based over-approximation of the solution space.

3.5 Strategy examples and their application

The SMT-RAT strategy examples in Figure 3.2 are kept very simple, that is we do not make use of branchings in these strategies, so there is no parallel solving involved, neither do we raise any conditions (true is fulfilled by any formula). We call this a *sequential SMT-RAT strategy*. However, these strategies highlight the fact that SMT-RAT is a toolbox, which provides modules that implement different procedures, rather than being an SMT solver.

The SMT-RAT strategy on the left in Figure 3.2 combines three theory solving modules. Therefore, we can only use this strategy in order to solve a conjunction φ of constraints. This strategy follows a simple but very useful scheme. First, it tries to use a Module$_{Simplex}$, which might detect that the linear part of φ is unsatisfiable or detect a solution which also satisfies the nonlinear part of φ. If this module cannot determine the satisfiability of φ, we invoke a Module$_{VS}$. Either its satisfiability check succeeds or it invokes a Module$_{CAD}$ on a formula, where possibly some variables, which occur in φ, could be eliminated.

There are many SMT solvers available and some of them do not support solving nonlinear real arithmetic. These SMT solvers can then integrate a theory solver based on the aforementioned strategy, which broadens their field of application. Figure 3.3 illustrates this cooperation.

The SMT-RAT strategy example in the middle of Figure 3.2, uses the strategy on the left as a theory solver for SMT-RAT's in-house SMT-solving implementation Module$_{SAT}$. It results in a strategy which forms an SMT solver for quantifier-free arithmetic formulas. If we extend this by

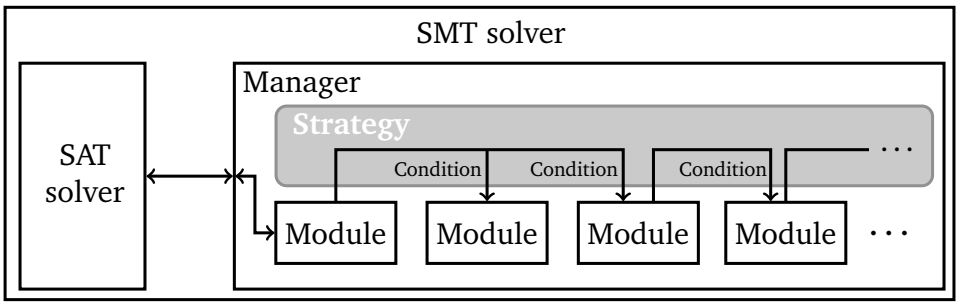


Figure 3.3: A snapshot of an SMT-RAT composition of a theory solver embedded in an SMT solver.

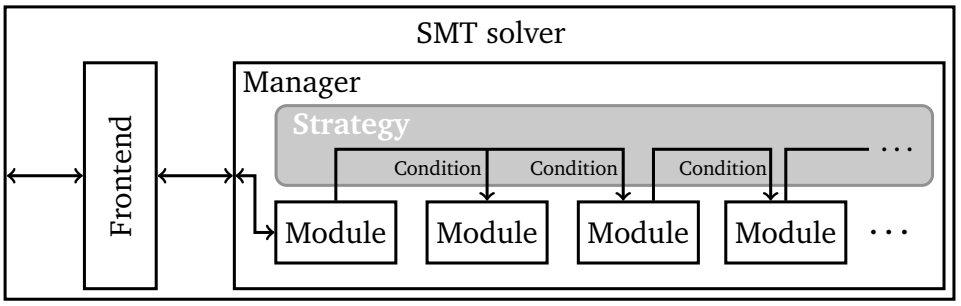


Figure 3.4: A snapshot of an SMT-RAT composition of an SMT solver.

a front-end in order to parse SMT-LIBv2 input files, which SMT-RAT also provides, we obtain an SMT solver that can be used according to custom straight away. We illustrate this framework in Figure 3.4.

The rightmost SMT-RAT strategy example in Figure 3.2 forms an SMT solver as well, but this time involving a $\texttt{Module}_{ICP}$. Hence, we use ICP to narrow down an over-approximation of a theory call's solution space. This involves lemmas being lifted to a $\texttt{Module}_{SAT}$, which represent splitting decisions and contractions of the over-approximation. As backend, the module $\texttt{Module}_{ICP}$ uses a strategy which tries to solve the input first with the virtual substitution and then with the cylindrical algebraic decomposition. Bear in mind that $\texttt{Module}_{ICP}$ passes through its received formula supplemented with a usually very narrow over-approximation of the received formula's solution space. As both $\texttt{Module}_{\text{VS}}$ and $\texttt{Module}_{\text{CAD}}$ highly benefit from this extra information, this strategy is often a performant alternative to the one in the middle of Figure 3.2.

CHAPTER 4

Virtual Substitution in SMT

In the last decade, SMT solving has already successfully brought state-of-the-art SAT solving and decision procedures for different logics together. As the SMT solver's SAT solver deals with the Boolean structure of the given SMT formula, the implementation of such a decision procedure, which is integrated in the SMT solver as a theory solver, is only supposed to check a conjunction of theory constraints for satisfiability. As a consequence, it can be optimized in order to speed up these checks, but we also require the theory solver to be SMT compliant for a stronger collaboration with the SAT solver within the SMT solver as explained in Section 2.6.

For instance, a theory solver used in an SMT solver for linear real-arithmetic needs to check a conjunction of linear real-arithmetic constraints for satisfiability. The Simplex method is originally designed to find an optimal solution of such a conjunction. In [DdM06], an SMT compliant adaption of the Simplex method is presented. In this chapter, we present an SMT compliant adaption of the virtual substitution, as introduced in Section 2.7, to check the satisfiability of a conjunction of nonlinear real-arithmetic constraints.

In this chapter, we first explain how to use the virtual substitution in general for a satisfiability check. Afterwards, we introduce a formal description of an SMT compliant theory solver through the following points:

- We introduce the data structure in which the theory solver can store its solving state in Section 4.2.1.
- In Section 4.2.2, we show how to add constraints to the theory solver incrementally, that is possibly after performing a theory check and then keeping as much information as possible in the theory solver's data structure.
- In Section 4.2.3, we present how to remove constraints from the theory solver belatedly,

that is possibly after performing a theory check and then keeping as much information as possible in the theory solver's data structure.

- How to perform a satisfiability check with the theory solver on the introduced data structure using the virtual substitution is explained in Section 4.2.4.
- The creation of infeasible subsets, in the case that the satisfiability check introduced in Section 4.2.4 detects unsatisfiability, is presented in Section 4.2.6.
- If, on contrary, the conjunction of constraints is found to be satisfiable, we explain in Section 4.2.5 how to obtain a solution.
- In Section 4.2.7, we illustrate all of these ideas on an extensive example.

Afterwards, we present in Section 4.3 how the virtual substitution can be combined with other procedures. We conclude this chapter in Section 4.4 with some ideas on further improvements, which we did not elaborate on within this thesis.

4.1 Virtual substitution for satisfiability checking

The virtual substitution as explained in Section 2.7 is a quantifier elimination procedure which applies Equation (2.24) or Equation (2.25) of Theorem 2 until all quantified variables are eliminated. With this procedure, we can decide the satisfiability/validity of real arithmetic sentences. For SMT solving, however, we are interested in the satisfiability of an, in our case, quantifier-free real-arithmetic formula $\varphi^{\mathbb{R}}$. As the satisfiability of $\varphi^{\mathbb{R}}$ is equivalent to the validity of $\exists x_1..\exists x_n.\varphi^{\mathbb{R}}$, if $\text{Vars}(\varphi^{\mathbb{R}}) = \{x_1,..,x_n\}$, we can use Theorem 2 for this purpose.

Corollary 1 *Let $\varphi^{\mathbb{R}}$ be a quantifier-free real-arithmetic formula with $x \in \text{Vars}(\varphi^{\mathbb{R}})$ occurring at most quadratic in $\varphi^{\mathbb{R}}$, then*

$$\varphi^{\mathbb{R}} \textit{ is satisfiable} \quad \Leftrightarrow \quad \bigvee_{t \in \text{tcs}(x,\varphi^{\mathbb{R}})} (\varphi^{\mathbb{R}}[t//x] \wedge \text{sc}(t)) \textit{ is satisfiable.}$$

As a direct consequence, we gain a mechanism to check whether a given formula $\varphi^{\mathbb{R}}$ is satisfiable. We choose a variable $x_n \in \text{Vars}(\varphi^{\mathbb{R}})$ and need to find only one test candidate $t_{x_n}^{i_n}$ for x_n in $\varphi^{\mathbb{R}}$, such that $\varphi^{\mathbb{R}}_{n-1} := \varphi^{\mathbb{R}}[t_{x_n}^{i_n}//x_n] \wedge \text{sc}(t_{x_n}^{i_n})$ is satisfiable. For checking the satisfiability of $\varphi^{\mathbb{R}}_{n-1}$, we do the same but now for one of the remaining variables in $\text{Vars}(\varphi^{\mathbb{R}}_{n-1})$ and so on. If $\varphi^{\mathbb{R}}$ is satisfiable, we find a test candidate $t_{x_j}^{i_j}$ for each variable $x_j \in \text{Vars}(\varphi^{\mathbb{R}}) = \{x_1,..,x_n\}$ such that

$$(..(\varphi^{\mathbb{R}}[t_{x_n}^{i_n}//x_n] \wedge \text{sc}(t_{x_n}^{i_n}))..)[t_{x_1}^{i_1}//x_1] \wedge \text{sc}(t_{x_1}^{i_1}) \quad \equiv \quad \text{true}.$$

An illustration of the thereby traversed depth-first search tree is shown in Figure 4.1.

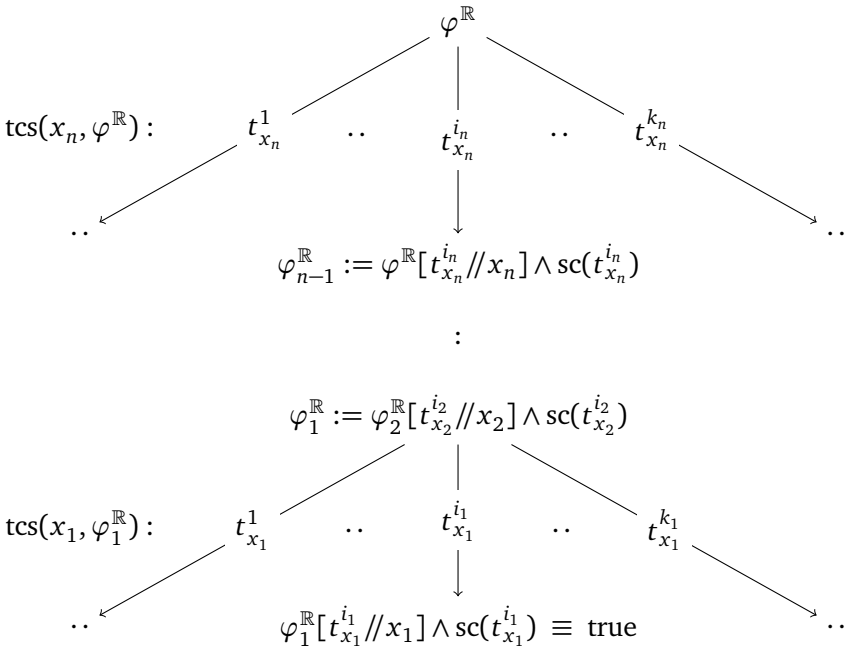


Figure 4.1: Possible depth-first search tree of the virtual substitution used for a satisfiability check.

4.2 An SMT-compliant theory solver based on the virtual substitution

We aim to design a theory solver which performs a depth-first search as illustrated in Figure 4.1. Here, we want to make use of the restriction that the formula $\varphi^{\mathbb{R}}$, which the theory solver has to check for satisfiability, is only a conjunction of real arithmetic constraints. Additionally, the theory solver has to be SMT compliant. Therefore, we need to store intermediate results in order to be able to omit their recalculation. However, it would clearly not be an option to just store the search tree as depicted in Figure 4.1, as it grows exponentially as the number of variables increases. Hence, we need to prune subtrees, which do not contain a satisfying assignment of test candidates to variables for $\varphi^{\mathbb{R}}$, and keep a record of the causes of these intermediate conflicts.

4.2.1 Data structure to store a depth-first search tree of the virtual substitution

Figure 4.1 gives us a rough idea of the requirements on a data structure, which we can use to store an intermediate result of a depth-first search for a satisfying assignment of test candidates to variables. It shows a directed tree, where the vertices are labeled by real arithmetic formulas (in Figure 4.1 by $\varphi^{\mathbb{R}}$, $\varphi^{\mathbb{R}}_{n-1}$, .., $\varphi^{\mathbb{R}}_1$, true) and the edges are labeled by test candidates (in Figure 4.1 by $t^1_{x_n}$, .., $t^{k_n}_{x_n}$, .., $t^1_{x_1}$, .., $t^{k_1}_{x_1}$).

As we want to prune unsatisfiable subtrees and store reasons for their unsatisfiability instead, we additionally want to be able to label the vertices by conflicts. Especially, for the sake of an

incremental adding and removing of information in our data structure, we must also be able to trace back the information's origins.

Definition 24 *A* virtual substitution search tree *(VSST)* $(V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC})$ *is a directed tree* (V, E) *where the vertices and edges are labeled.*

The vertices are labeled by

- *a set of formulas, which is denoted by*

$$\Phi\colon\ V \to \mathbb{P}(\text{FO}(\tau)),$$

- *a function that maps formulas to a set of sets of formulas, which is denoted by*

$$\text{orig}\colon\ V \to (\text{FO}(\tau) \to \mathbb{P}(\mathbb{P}(\text{FO}(\tau)))),$$

- *a set of sets of formulas, which is denoted by*

$$\kappa\colon\ V \to \mathbb{P}(\mathbb{P}(\text{FO}(\tau)))$$

- *and a Boolean flag, which is denoted by*

$$\text{incompl}\colon\ V \to \mathbb{B}.$$

Some vertices are also labeled by an arithmetic variable, which is denoted by the partial function

$$\text{elimvar}\colon\ V \to \text{VAR}_{\mathbb{R},\mathbb{Z}}$$

and a function that maps formulas to Boolean constants, which is denoted by the partial function

$$\text{used}\colon\ V \to (\text{FO}(\tau) \to \mathbb{B}).$$

The edges are partially labeled by a pair of a test candidate and a set of formulas, which is denoted by the partial function

$$\text{TC}\colon\ E \to \text{TCS} \times \mathbb{P}(\text{FO}(\tau)).$$

A VSST $(V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC})$ is very similar to the search tree from Figure 4.1. The main differences are the additional labels orig, κ, used and incompl of a vertex $v \in V$ and that there is now a set of a formulas instead of just a formula, where the conjunction of these formulas, i. e., $\bigwedge_{\varphi \in \Phi(v)} \varphi$, corresponds to a formula in Figure 4.1. The other labels have the following meaning:

orig : This label specifies a function $\text{orig}(v)$ that maps each formula φ in $\Phi(v)$ to a set of sets of

formulas from $\Phi(v')$, where v' is the father of v. Each set in $\text{orig}(v)(\varphi)$ forms a reason for the existence of φ. For instance, if $\text{orig}(v)(\varphi) = \{\{\varphi_1, \varphi_2\}, \{\varphi_3\}\}$, φ_1 and φ_2 are together responsible that we created φ. The formula φ_3 forms another reason why we created φ. If v is the root of (V, E) and, therefore, has no father, it holds that $\text{orig}(v)(\varphi) = \{\{\varphi\}\}$.

κ : This label forms a set of conflicts, which contains infeasible subsets of the formulas in $\Phi(v)$, if the search for a satisfying assignment of test candidates to variables yields that $\bigwedge_{\varphi \in \Phi(v)} \varphi$ is unsatisfiable. Otherwise, $\kappa(v)$ is empty.

incompl : This label specifies a Boolean flag that indicates whether the satisfiability of the conjunction of the formulas in $\Phi(v)$ cannot be determined by the virtual substitution as the degree of some constraints is to high.

A vertex $v \in V$ can also be labeled with the variable $\text{elimvar}(v)$, which we eliminate next from the formulas in $\Phi(v)$ according to Corollary 1. For such a vertex, the labeling function TC maps the edges $(v, v') \in E$ to a pair of a test candidate and a set of formulas in $\Phi(v)$, which form reasons for the existence of this test candidate. This makes sense as several constraints might provide the same test candidate, for instance, all constraints which contain $\text{elimvar}(v)$ provide the test candidate $-\infty$. Furthermore, a vertex $v \in V$ can be labeled with a function $\text{used}(v)$ that maps each formula in $\Phi(v)$ to a Boolean constant, which specifies whether the formula (which must be a constraint in this case) has been used to provide test candidates or not. Hereby we enable an incremental creation of test candidates.

We use a VSST in the intended SMT-compliant theory solver and initialize it with:

$$\begin{aligned} T_{\text{VS}} &= (V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC}) \\ &:= (\{v_1\},\ \emptyset,\ \{(v_1, \emptyset)\},\ \{(v_1, \emptyset)\},\ \{(v_1, \emptyset)\},\ \{(v_1, \emptyset)\},\ \emptyset,\ \{(v_1, \text{false})\},\ \emptyset) \end{aligned}$$

We ensure that T_{VS} always fulfills the following invariants:

1. The labeling function TC is undefined for all edges $(v, v') \in E$, i. e., $\text{TC}((v, v')) = \bot$, if elimvar is undefined for v, i. e., $\text{elimvar}(v) = \bot$.

2. If $\Phi(v)$ contains a formula which is not a constraint, elimvar is undefined, i. e., $\text{elimvar}(v) = \bot$, and there exists no more than one child v' of v which is not conflicting, i. e., $\kappa(v') = \emptyset$, and whose satisfiability is not impossible to determine with the virtual substitution, i. e., $\text{incompl}(v') = \text{false}$.

3. The function Φ maps the root of (V, E) to a set of constraints.

The meaning of the first invariant is quite natural. It states that we only construct test candidates, if the variable to eliminate next, which is the variable for which the test candidates are constructed, is specified.

The second invariant assures that we only eliminate a variable, if the considered formulas are just constraints. This is due to the fact that the search for a satisfying assignment of test candidates

Algorithm 3 Given a VSST $T_{\text{VS}} = (V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC})$ and a vertex $v \in V$, this procedure creates an empty child v' of v.

createChild(VSST T_{VS}, vertex $v \in V$)
begin
1: $v' :=$ fresh vertex not occurring in V
2: $V := V \cup \{v'\}$
3: $\Phi := \Phi \cup \{(v', \emptyset)\}$
4: incompl := incompl $\cup \{(v', \text{false})$
5: orig := orig $\cup \{(v', \emptyset)\}$
6: $\kappa := \kappa \cup \{(v', \emptyset)\}$
7: $E := E \cup \{(v, v')\}$
8: **return** v'
end

to variables, which is presented in the course of this chapter, strictly distinguishes whether we consider a conjunction of constraints or not. For the case that a vertex $v \in V$ is labeled by Φ with a set of constraints, we present an incremental approach to eliminate a variable in $\bigwedge_{c \in \Phi(v)} c$ according to Corollary 1 in Section 4.2.4. Here, we create a child v' for each test candidate t and set $\text{TC}((v, v')) = (t, C)$, where C contains the constraints which provide t, respectively, and $\Phi(v') = \{c[t//\text{elimvar}(v)] |\ c \in \Phi(v)\} \cup \{\text{sc}(t)\}$. If v is labeled by Φ with a set containing formulas not being constraints, we need a case distinction. The children of v then reflect the single cases, such that they are labeled by Φ with a set, which contains only constraints again. In this case, only one case is considered at a time. This means, that all but one child are either conflicting or marked by incompl(v) with true, which means that for this case the virtual substitution was not able to determine the satisfiability.

The root in the theory solver's VSST stores the constraints which the theory solver has to check for satisfiability, therefore the third invariant has to hold.

Before, we can present the algorithms for adding and removing constraints to a VSST, and the satisfiability check for the constraints in the root of a VSST, we introduce the following two auxiliary procedures, which enable a manipulation of a VSST.

createChild We can add vertices to a VSST with Algorithm 3. For a given vertex it creates a child with an empty set of formulas and no conflicts.

deleteSubtree We can remove vertices from a VSST with Algorithm 4. This procedure expects the edge to the vertex to delete as input and removes this edge and all vertices and edges in the subtree where the vertex to delete is the root.

Definition 25 (Conflicting and valid vertices in a VSST) *Given a VSST* $T_{\text{VS}} = (V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC})$ *and a vertex* $v \in V$*, we call it*

- conflicting*, if* $\kappa(v) \neq \emptyset$*, and*

- valid, *if it is not conflicting and* $\Phi(v) = \emptyset$.

In Figure 4.1, for instance, the leaf labeled by true corresponds to a valid vertex in a VSST.

4.2.2 Incremental adding of constraints

If we add a constraint c to the theory solver, we have to add it to the theory solver's VSST T_{VS} storing the result of the last search for a satisfying assignment of test candidates to variables. For this purpose we invoke the procedure $\text{add}_{\text{VS}}(T_{\text{VS}}, v_1, c, \{c\})$ as introduced by Algorithm 5, where v_1 is the root of T_{VS}.

Given a VSST T_{VS}, adding a formula φ to a vertex $v \in V$, where φ has the origins $M \subseteq \Phi(v')$ in v's father v' (if v is not the root, otherwise $M = \{\varphi\}$), is accomplished in two phases. First, we add φ to the vertex's formulas $\Phi(v)$ and extend the mapping of formulas to origins $\text{orig}(v)$ by $(\varphi, \{M\})$ (Line 5-6). In the case that $\varphi =$ false we add a conflict set to $\kappa(v)$ consisting of only φ (Line 2). If $\Phi(v)$ contained only constraints and a variable to eliminate is already fixed, we extend $\text{used}(v)$ by (φ, false) and reset $\text{incompl}(v)$ to false, if φ is a constraint (Line 9). This indicates, that φ has not yet provided test candidates and, in the case that $\text{incompl}(v)$ was true before, we now might be able to determine the satisfiability of the conjunction of constraints in $\Phi(v)$. If φ is not a constraint, we delete $\text{used}(v)$, $\text{elimvar}(v)$ and all children of v, which means, that this vertex is now used for a case distinction instead of a variable elimination (Line 13-16).

In the second phase of Algorithm 5 we update v's children (Line 24). If a child was introduced by reason of a variable elimination, we extend its formulas by the result of $\varphi[t//\text{elimvar}(v)]$, where t is the test candidate in the label of the edge from v to the child. We achieve this by calling add_{VS} recursively with $\{\varphi\}$ as origin (Line 27). If the child was introduced by the result of a case distinction, we have to extend the currently considered case, which is represented by the only non-conflicting child (Line 30). It might happen that due to the addition of φ no more cases have to be considered. Then, the formula considered by v is unsatisfiable and we create the conflicts according to Algorithm 11.

Algorithm 4 Given a VSST $T_{\text{VS}} = (V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC})$ and an edge $(v, v') \in E$, this procedure deletes the subtree in T_{VS} with root v'.

```
deleteSubtree(VSST T_VS, edge (v, v') ∈ E)
begin
 1:   // remove all children
 2:   while exists (v', v'') ∈ E do
 3:       deleteSubtree(T_VS, (v', v''))
 4:   end while
 5:   // remove vertex and edge to its father
 6:   E := E \ {(v, v')}
 7:   V := V \ {v'}
end
```

Algorithm 5 Given a VSST $T_{\mathrm{VS}} = (V, E, \Phi, \mathrm{orig}, \kappa, \mathrm{elimvar}, \mathrm{used}, \mathrm{incompl}, \mathrm{TC})$ and a vertex $v \in V$, this procedure adds the given formula φ to $\Phi(v)$ and propagates this update in the subtree with the root v.

```
add_VS(VSST T_VS, vertex v ∈ V, formula φ, set of formulas M)
begin
 1:   // add φ to the vertex
 2:   if φ = false then κ(v) := κ(v) ∪ {{φ}}          // formula forms trivially a conflict
 3:   if φ = true then return                          // does not influence the satisfiability
 4:   if φ ∉ Φ(v) then
 5:       Φ(v) := Φ(v) ∪ {φ}                          // add the formula
 6:       orig(v) := orig(v) ∪ {(φ, {M})}             // add the formulas origin
 7:       if elimvar(v) ≠ ⊥ then
 8:           if φ is a constraint then
 9:               used(v) := used(v) ∪ {(φ, false)}   // extend used-flags
10:               // new constraint might make it possible to determine satisfiability
11:               incompl(v) := false
12:           else
13:               elimvar := elimvar \ {(v, elimvar(v)}   // remove elimination variable
14:               used := used \ {(v, used(v)}            // remove all used-flags
15:               for all (v, v') ∈ E do
16:                   deleteSubtree(T_VS, (v, v'))        // delete all children
17:               end for
18:           end if
19:       end if
20:   else
21:       orig(v)(φ) := orig(v)(φ) ∪ {M}              // formula exists: only add the origin
22:   end if
23:   // update the children
24:   for all (v, v') ∈ E do
25:       if elimvar(v) ≠ ⊥ then
26:           (t, M_t) := TC((v, v'))                 // child with VS result
27:           add_VS(T_VS, v', φ[t//elimvar(v)], {φ})  // extend the VS result
28:       else if κ(v') = ∅ then
29:           // try to extend the considered case of the child
30:           if extendCase(T_VS, v) = false then createConflicts(T_VS, v)
31:       end if
32:   end for
end
```

Within Algorithm 5 we use the following sub-procedure.

extendCase Consider that we consider in a vertex v' one case of the case distinction which we made for the formulas considered by the father v of v'. Then assume that we add formulas to $\Phi(v)$ and, therefore, have to extend the formulas in $\Phi(v')$ as well such that it is still a valid case of the conjunction of the formulas in $\Phi(v)$. With the procedure extendCase,

Algorithm 6 Given a VSST $T_{\mathrm{VS}} = (V, E, \Phi, \mathrm{orig}, \kappa, \mathrm{elimvar}, \mathrm{used}, \mathrm{incompl}, \mathrm{TC})$ and an edge $(v, v') \in E$, this procedure extends the constraints considered by v' in $\Phi(v')$ to a case of the formulas considered by v in $\Phi(v)$, such that if $\bigwedge_{c\in\Phi(v')} c$ is satisfiable, so is $\bigwedge_{\varphi\in\Phi(v)} \varphi$.

extendCase(VSST T_{VS}, edge $(v, v') \in E$)
begin
1: *// assumption:* $\kappa(v) = \emptyset$
2: $V' :=$ children of v which are not v'
3: *// formulas considered by v while excluding a conflict for each child*
4: $\psi := \bigwedge_{\varphi\in\Phi(v)} \varphi \wedge \bigwedge_{v''\in V',\kappa(v'')\neq\emptyset} \neg(\bigwedge_{c\in K_{v''}} c)$ *// for a heuristically chosen $K_{v''} \in \kappa(v'')$*
5: $\wedge \bigwedge_{v''\in V',\kappa(v'')=\emptyset} \neg(\bigwedge_{c\in\Phi(v'')} c)$ *// exclude cases with* $\mathrm{incompl}(v'') = \mathrm{true}$
6: $\psi^{\mathbb{B}} :=$ Boolean abstraction of ψ
7: $\mathrm{abstr}^{\mathbb{B}}_{\psi} :=$ corresponding Boolean abstraction mapping
8: **if** $\psi^{\mathbb{B}}$ is satisfiable **then**
9: $\alpha :=$ satisfying assignment for $\psi^{\mathbb{B}}$
10: *// collect the constraints which have to hold according to α*
11: $M := \{c \in C_{\sim}(\psi) |\ \alpha(\mathrm{abstr}^{\mathbb{B}}_{\psi}(c)) = \mathrm{true}\}$
12: *// update v' such that it now considers the formulas in M*
13: $\mathrm{remove}_{\mathrm{VS}}(T_{\mathrm{VS}}, v', \Phi(v') \setminus M)$
14: **for all** $c \in M \setminus \Phi(v')$ **do**
15: $O_c := \{\varphi \in \Phi(v) |\ c \in C_{\sim}(\varphi)\}$ *// create origins of c*
16: $\mathrm{add}_{\mathrm{VS}}(T_{\mathrm{VS}}, v', c, O_c)$
17: **end for**
18: **return** true
19: **else**
20: deleteSubtree$(T_{\mathrm{VS}}, (v, v'))$
21: **return** false
22: **end if**
end

which we implement in Algorithm 6, we can extend the formulas in v' and update it for this purpose. This procedure determines a satisfying assignment α, if any exists, for the Boolean abstraction $\psi^{\mathbb{B}}$ of the formula

$$\psi = \bigwedge_{\varphi\in\Phi(v)} \varphi \wedge \bigwedge_{\substack{v''\in V' \\ \kappa(v'')\neq\emptyset}} \neg(\bigwedge_{c\in K_{v''}} c) \wedge \bigwedge_{\substack{v''\in V' \\ \kappa(v'')=\emptyset}} \neg(\bigwedge_{c\in\Phi(v'')} c),$$

where V' are the children of v without v' and we choose $K_{v''} \in \kappa(v'')$ heuristically. Thus, we obtain a case, which is not yet excluded by one of the conflicts in one of the children of v. Neither it is a case for which we already know that we cannot determine the satisfiability using the virtual substitution. These cases are represented by the children of v which are marked by incompl with true. Note, that if we invoke extendCase with the edge (v, v') as argument, it is ensured that for each child $v'' \neq v'$ of v either $\kappa(v'') \neq \emptyset$ or $\mathrm{incompl}(v'') =$

true.

Let us have a closer look at how each $\varphi \in \Phi(v)$ is constructed. As v is a vertex which we use for a case distinction, it cannot be the root of T_{VS}. Therefore, it must have a father v_f and each $\varphi \in \Phi(v)$ is the result of $c_f[t//\text{elimvar}(v_f)]$ for some constraints $c_f \in \Phi(v_f)$ and $\text{TC}((v_f, v)) = (t, M_t)$. Considering the rules we use to obtain this result, as depicted in [Wei97] or [Cor10], we see that it is in NNF, where in particular all literals are positive. If there is no $\varphi \in \Phi(v)$ with $\varphi = \text{false}$, the Boolean skeleton $\psi'_{\mathbb{B}}$ of $\psi' = \bigwedge_{\varphi \in \Phi(v)} \varphi$ is then obviously satisfiable (for instance, it is satisfied by the assignment which assigns true to all of its variables). As α is a satisfying assignment of $\psi^{\mathbb{B}}$ and, thus, in particular a satisfying assignment of $\psi'_{\mathbb{B}}$, it also holds that any assignment α' which assigns true to each variable x, if $\alpha(x) = \text{true}$, is a satisfying assignment of $\psi'_{\mathbb{B}}$. This is due to the fact, that $\psi'_{\mathbb{B}}$ is in NNF and only constructed by conjunctions and disjunctions. As any satisfied disjunction is still satisfied if one of its sub-formulas is true instead of false, we can prove by induction on the formula depth that $\psi'_{\mathbb{B}}$ is satisfied by α'. Therefore, ψ' is satisfiable, if the conjunction of those constraints, to whose Boolean abstraction α assigns true, is satisfiable. Hence, we have to make sure that v' contains these constraints. We add them to v' (Line 16), if they are not yet in the constraints considered by v', and remove all constraints, which are considered by v' but do not have to hold according to α (Line 13). If we add a constraint c to v' we specify that its origin in v consists of all formulas in $\Phi(v)$ which contain c (Line 15). In Section 4.2.6 we explain why this is sufficient in order to construct conflicts for v by the use of the conflicts in v' if it is conflicting. However, in many cases it still forms an over-approximation of the actual reason why we assigned true to the Boolean abstraction of c. We propose another idea as to how we might obtain better origins in Section 4.4.

If $\psi^{\mathbb{B}}$ is unsatisfiable, all cases are covered and we delete the subtree with the root v', but keep all the other children of v, which contain the conflicts we make use of in order to construct conflicts for v (Line 20). If for each child v'' of v it holds that $\text{incompl}(v'') = \text{false}$, $\bigwedge_{\varphi \in \Phi(v)} \varphi$ is indeed unsatisfiable, as the unsatisfiability of the Boolean abstraction of ψ implies the unsatisfiability of ψ, which again implies the unsatisfiability of $\bigwedge_{\varphi \in \Phi(v)} \varphi$, as $\neg(\bigwedge_{c \in K} c)$ is a tautology for all $K \in \kappa(v'')$ and $v'' \in V'$.

4.2.3 Belated removing of constraints

Removing a constraint c from the theory solver means that we have to remove it from the theory solver's VSST T_{VS}. We can achieve this, as a VSST also stores the origins of formulas in the labels of the vertices and the origins of test candidates in the labels of the edges. For this purpose we invoke the procedure $\text{remove}_{\text{VS}}(T_{\text{VS}}, v_1, \{c\})$ as introduced by Algorithm 7, where v_1 is the root of T_{VS}.

In general, we can remove the formulas in $M = \{\varphi_1, \ldots, \varphi_n\}$ and everything which has its

Algorithm 7 Given a VSST $T_{\mathrm{VS}} = (V, E, \Phi, \mathrm{orig}, \kappa, \mathrm{elimvar}, \mathrm{used}, \mathrm{incompl}, \mathrm{TC})$ and a vertex $v \in V$, this procedure removes everything from the subtree with the root v which has its origins in the given set of formulas M.

remove$_{\mathrm{VS}}$(VSST T_{VS}, vertex $v \in V$, set of formulas M)
begin
1: $M' := \emptyset$ *// set of formulas to remove from the children as origins*
2: *// remove origins from formulas in vertex*
3: **for all** $\varphi \in \Phi(v)$ **do**
4: $N := \emptyset$ *// stores the remaining origin sets*
5: **for all** $O \in \mathrm{orig}(v)(\varphi)$ **do**
6: **if** $O \cap M = \emptyset$ **then** $N := N \cup \{O\}$ *// add origin, if it contains no formula to remove*
7: **end for**
8: **if** $N = \emptyset$ **then**
9: **if** $\mathrm{used}(v) \neq \bot$ **then** $\mathrm{used}(v) := \mathrm{used}(v) \setminus \{(\varphi, \mathrm{used}(v)(\varphi))\}$
10: $\mathrm{orig}(v) := \mathrm{orig}(v) \setminus \{(\varphi, \mathrm{orig}(v)(\varphi))\}$
11: $\Phi(v) := \Phi(v) \setminus \{\varphi\}$
12: *// removing constraint might make it possible to determine satisfiability*
13: $\mathrm{incompl}(v) := \mathrm{false}$
14: $M' := M' \cup \{\varphi\}$
15: **else**
16: $\mathrm{orig}(v)(\varphi) := N$
17: **end if**
18: **end for**
19: *// remove conflicts which depend on M'*
20: **if** $\kappa(v) \neq \emptyset$ **then**
21: $\kappa(v) := \{K' \in \kappa(v) \mid K' \cap M' = \emptyset\}$
22: $v' :=$ father of v
23: **if** $\kappa(v) = \emptyset \ \wedge\ \mathrm{elimvar}(v') = \bot$ **then**
24: deleteSubtree$(T_{\mathrm{VS}}, (v', v))$ *// as we allow only one case at a time*
25: **end if**
26: **end if**
27: *// remove M' from origins in the children*
28: **for all** $(v, v') \in E$ **do**
29: **if** $\mathrm{TC}((v, v')) \neq \bot$ **then**
30: $(t, O_t) := \mathrm{TC}((v, v'))$
31: **if** $O_t \subseteq M'$ **then**
32: deleteSubtree$(T_{\mathrm{VS}}, (v, v'))$ *// remove child if origins of test candidate vanish*
33: **else**
34: $\mathrm{TC}((v, v')) := (t, O_t \setminus M')$ *// update origins of test candidate*
35: remove$_{\mathrm{VS}}(T_{\mathrm{VS}}, v', M')$
36: **end if**
37: **else** remove$_{\mathrm{VS}}(T_{\mathrm{VS}}, v', M')$
38: **end for**
end

origin in $\varphi_1, \ldots, \varphi_n$ from a subtree of T_{VS} with the root v, by invoking $\text{remove}_{\text{VS}}(T_{\text{VS}}, v, M)$. This procedure first removes all those origins in $\text{orig}(v)(\varphi)$ for each formula φ in $\Phi(v)$, which contain formulas from the given set M of origins to remove (Line 4-16). If a formula thereby runs out of origins, we delete it and add it to the set M' of origins, which we remove from v's children later (Line 9-14). In this case we also reset $\text{incompl}(v)$ to false as it now might be possible to determine the satisfiability of the conjunction of formulas in $\Phi(v)$ (Line 13).

If the vertex v is labeled with a non-empty conflict, we remove all conflicts from $\kappa(v)$ containing formulas whose origins vanished, i. e., the formulas in M' (Line 20-24). In the case that no conflict remains and this vertex is a case of the case distinction made for its father v', we delete it (Line 24). Otherwise, v is still conflicting or it considers the result of $\Phi(v')[t//\text{elimvar}(v')] \wedge \text{sc}(t)$ where t is the test candidate in the label of the edge (v', v). Hence, we recycle conflicts and virtual substitution results.

Finally, Algorithm 7 removes the formulas in M' from the origins in v's children. If the edge from v to a child is labeled by a test candidate, we also update its origins (Line 30-35). If they vanish, we remove the child (Line 32).

4.2.4 Checking a conjunction of constraints for satisfiability

As seen before, we can add and remove constraints to the theory solver, leaving a set of constraints in the root of the theory solver's VSST T_{VS}. In order to check this set of constraints for satisfiability, we invoke $\text{check}_{\text{VS}}(T_{\text{VS}}, v_1)$, which is described in Algorithm 10, passing the VSST T_{VS} and its root v_1 as input. If this procedure returns `sat`, the conjunction of constraints in $\Phi(v_1)$ is satisfiable and, if it returns `unsat`, this conjunction is unsatisfiable. If this procedure returns `unknown`, it was not possible to determine the satisfiability of $\bigwedge_{c \in \Phi(v_1)} c$ with the presented implementation of the virtual substitution method. Bear in mind, that the virtual substitution is incomplete for general real arithmetic formulas, as explained in Section 2.7. Before we go into detail about Algorithm 10, we describe two sub-procedures, which it uses.

updateEliminationVar Let us consider a vertex v in our VSST T_{VS}, such that all formulas in $\Phi(v)$ are constraints. As explained before, we then eliminate a variable according to Corollary 1. The procedure updateEliminationVar, which is described in Algorithm 8, determines the variable, which we eliminate, and updates the vertex afterwards. We choose the variable according to certain heuristics which are purely based on the constraints in $\Phi(v)$ (Line 1). This choice is vital for the performance of the virtual substitution, as a wrong choice can lead to more cases or the position where we cannot determine the satisfiability, while the right choice could have prevented this. We present a detailed description of the metrics of this heuristic choice in Section 5.1. After deciding which variable is the best to eliminate, we update the vertex in Algorithm 8. This means, that even if the variable to eliminate has been chosen before and some constraints in $\Phi(v)$ have provided test candidates, we might change this choice and delete everything which depended on it (Line 9-15). In the

Algorithm 8 Given a VSST $T_{\mathrm{VS}} = (V, E, \Phi, \mathrm{orig}, \kappa, \mathrm{elimvar}, \mathrm{used}, \mathrm{incompl}, \mathrm{TC})$ and a vertex $v \in V$, this procedure updates the variable to eliminate next.

```
updateEliminationVar(VSST T_VS, vertex v ∈ V)
begin
 1:   x := heuristically the best variable to eliminate next in Φ(v)
 2:   if elimvar(v) = ⊥ then
 3:     // no elimination variable chosen yet: set it to x
 4:     elimvar := elimvar ∪ {(v, x)}
 5:     // initialize flags to indicate if constraints were used for test candidate creation
 6:     used := used ∪ {(v, {(c, false)| c ∈ Φ(v)})}
 7:   else if elimvar(v) ≠ x ∧ changing to x improves enough then
 8:     // found a better elimination variable
 9:     elimvar(v) := x
10:     // delete all children of v
11:     while exists (v, v') ∈ E do
12:       deleteSubtree(T_VS, (v, v'))
13:     end while
14:     // update flags to indicate if constraints were used for test candidate creation
15:     used(v) := {(c, false)| c ∈ Φ(v)}
16:   end if
end
```

case, that the variable to eliminate has not yet been set, we fix the elimination variable and ensure that all constraints are mapped by used(v) to false, which means that we have not yet used these constraints in order to provide test candidates (Line 4-6).

createTCs In the procedure to check the satisfiability of the constraints in the root of the VSST T_{VS}, we also make use of the sub-procedure createTCs, which constructs for a given vertex v test candidates. Here, we assume that $\Phi(v)$ contains only constraints and that v's elimination variable elimvar(v) is already fixed. We can construct test candidates, if $\Phi(v)$ contains a constraint c, which has not yet been used in order to provide test candidates, i. e., used(v)(c) = false, and which contains the variable to eliminate but its degree in c is less than or equal to 2. Furthermore, we only construct a test candidate if it has not yet been provided by another constraint, which is the case if an edge from v to one of its children is labeled with this test candidate. Otherwise, we only extend the origins of this test candidate (Line 8) and add the result of virtually substituting the elimination variable by this test candidate in c to this child (Line 9).[1] Assume that there are constraints fulfilling the aforementioned conditions. We choose one of them according to heuristics, which are explained in Section 5.1. Let us assume that we choose the constraint c. Then we create for each test candidate t for the elimination variable elimvar(v) in c a child (Line 11), such

[1] Here, we can further optimize the incremental creation of the VSST, if we bookmark constraints for which we have not yet added the result of virtually substituting the elimination variable by this test candidate in c to this child, and only add it, if we actually want to consider this child.

Algorithm 9 Given a VSST $T_{\mathrm{VS}} = (V, E, \Phi, \mathrm{orig}, \kappa, \mathrm{elimvar}, \mathrm{used}, \mathrm{incompl}, \mathrm{TC})$ and a vertex $v \in V$ and a constraint $c \in \Phi(v)$, this procedure searches for a constraint $c \in \Phi(v)$, which can provide further test candidates. It then creates a child v' of v for each test candidate t for the variable $\mathrm{elimvar}(v)$ in c, such that the formulas considered by v' in $\Phi(v')$ are the formulas considered by v in $\Phi(v)$ where $\mathrm{elimvar}(v)$ is virtually substituted by t and t's side conditions. Then it holds that if $\bigwedge_{c \in \Phi(v')} c$ is satisfiable, so is $\bigwedge_{\varphi \in \Phi(v)} \varphi$.

createTCs(VSST T_{VS}, vertex $v \in V$)
begin
1: createdTC := false *// this variable is set to* true *if a test candidate has been created*
2: *// find heuristically the best constraint which has not yet been used for test candidate creation*
3: **while** exists $c \in \Phi(v)$ with $\mathrm{used}(v)(c) = \mathrm{false}$ **do**
4: $\mathrm{used}(v)(c) := \mathrm{true}$ *// constraint marked as used for test candidate creation*
5: **if** $0 < \deg(\mathrm{elimvar}(v), \mathrm{Pol}(c)) \leq 2$ **then**
6: **for all** $t \in \mathrm{tcs}(\mathrm{elimvar}(v), c)$ **do**
7: **if** exists $(v, v') \in E$ with $\mathrm{TC}((v, v')) = (t, O_t)$ **then**
8: $\mathrm{TC}((v, v')) := (t, O_t \cup \{c\})$
9: $\mathrm{add}_{\mathrm{VS}}(T_{\mathrm{VS}}, v', c[t//\mathrm{elimvar}(v)], \{c\})$ *// add* $c[t//\mathrm{elimvar}(v)]$ *to* v'
10: **else**
11: $v' := \mathrm{createChild}(T_{\mathrm{VS}}, v)$ *// create a child for* t
12: $\mathrm{TC} := \mathrm{TC} \cup \{((v, v'), (t, \{c\}))\}$ *// label new edge with* t*, whose origin is* c
13: $\mathrm{add}_{\mathrm{VS}}(T_{\mathrm{VS}}, v', \mathrm{sc}(t), \{c\})$ *// add* t*'s side condition to* v'
14: **for all** $c' \in \Phi(v)$ **do**
15: $\mathrm{add}_{\mathrm{VS}}(T_{\mathrm{VS}}, v', c'[t//\mathrm{elimvar}(v)], \{c'\})$ *// add* $c'[t//\mathrm{elimvar}(v)]$ *to* v'
16: **end for**
17: createdTC := true
18: **end if**
19: **end for**
20: **if** createdTC **then return** true
21: **else if** $\deg(\mathrm{elimvar}(v), \mathrm{Pol}(c)) > 2$ **then**
22: $\mathrm{incompl}(v) = \mathrm{true}$ *// we cannot determine the unsatisfiability*
23: **end if**
24: **end if**
25: **return** false
end

that the edge leading to it is labeled with t and the origins consisting of just c (Line 12). Additionally, we add t's side conditions and the results of virtually substituting the elimination variable by t in each constraint c' in $\Phi(v)$, i. e., $c'[t//\mathrm{elimvar}(v)]$, to the created child (Line 13-15). Note that the origins of t's side conditions in the created child v' consist of the constraint, which provided t, i. e., c. The origins of the virtual substitution results in v' consist only of the constraint we substituted in. Finally, this procedure returns true if test candidates were created, otherwise it returns false.

We can summarize the main idea of the procedure $\mathrm{check}_{\mathrm{VS}}$, described in Algorithm 10, as

Algorithm 10 Given a VSST $T_{\text{VS}} = (V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC})$ and a vertex $v \in V$, this procedure checks the satisfiability of the formulas in v, i. e., $\bigwedge_{\varphi \in \Phi(v)} \varphi$.

```
check_VS(VSST T_VS, vertex v ∈ V)
begin
 1:   if κ(v) ≠ ∅ then return unsat                          // v is conflicting
 2:   if Φ(v) = ∅ then return sat                             // v is valid
 3:   while true do
 4:     // choose heuristically the best child which is not conflicting
 5:     while exists (v, v') ∈ E with κ(v') = ∅ do
 6:       if check_VS(T_VS, v') = sat then return sat
 7:     end while
 8:     // create new children
 9:     if Φ(v) contains only constraints then
10:       updateEliminationVar(T_VS, v,)          // update the elimination variable
11:       if createTCs(T_VS, v) = false then
12:         // all test candidates considered and hence all children are constructed
13:         return createConflicts(T_VS, v) // construct infeasible subsets of Φ(v) and return
14:       end if
15:     else
16:       v' := createChild(T_VS, v)                        // create an empty child of v
17:       if extendCase(T_VS, (v, v')) = false then
18:         // all cases considered and hence all children constructed
19:         if v has child v'' with incompl(v'') = true then
20:           incompl(v) = true
21:         end if
22:         return createConflicts(T_VS, v) // construct infeasible subsets of Φ(v) and return
23:       end if
24:     end if
25:   end while
end
```

follows. Each vertex v considers a formula, which is equisatisfiable to the disjunction of the formulas considered by v's children, if all children have been constructed. If $\Phi(v)$ contains only constraints, all children are constructed if all test candidates for the variable to eliminate in all of these constraints were constructed with createTCs. Otherwise, all children are constructed if all cases of the formula $\bigwedge_{\varphi \in \Phi(v)} \varphi$ were considered according to the procedure extendCase.

This implies that as soon as we find a valid vertex v, the conjunction of the formulas in v's father is satisfiable and, therefore, the same holds for v's father's father and so on. It follows that the conjunction of the formulas in the root is satisfiable and, hence, we can return `sat`. In order to find a valid vertex, starting with the root, we need to alternately choose a child (and test candidate) when eliminating a variable and then a child of this child (and case) when making a case distinction until reaching a valid vertex.

If, on the other hand, all children of a vertex v are constructed and all of them are conflicting, it

Algorithm 11 Given a VSST $T_{VS} = (V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC})$ and a vertex $v \in V$, this procedure calculates infeasible subsets of $\Phi(v)$ and stores them into $\kappa(v)$.

```
createConflicts(VSST T_VS, vertex v ∈ V)
begin
 1:   // if conjunction of formulas in v are not necessarily unsatisfiable
 2:   if incompl(v) = true then return unknown
 3:   // otherwise, find the best sets to cover all conflicts in the children
 4:   κ(v) := bestConflictCoveringSets(T_VS, v)
 5:   if v is not root then        // keep children of root, which can be reused after backtracking
 6:       // delete all children of v
 7:       while exists (v, v') ∈ E do
 8:           deleteSubtree(T_VS, (v, v'))
 9:       end while
10:   end if
end
```

follows from Theorem 2 that the conjunction of the formulas considered by v must be unsatisfiable. It remains to fill $\kappa(v)$ based on the conflicts in the children with infeasible subsets of $\Phi(v)$ and delete all children, which we accomplish with Algorithm 11. We explain in Section 4.2.6 how to achieve these infeasible subsets. Note that if $\Phi(v)$ contains a formula φ which is false, it has been added to v with the procedure add_{VS} and is the result of the elimination of variable by a test candidate (or directly added to the root). Within the procedure add_{VS} we ensure that $\{\varphi\}$ is then added as a conflict to $\kappa(v)$.

On the basis of this concept, Algorithm 10 processes a given vertex v in the theory solvers VSST T_{VS}. If $\Phi(v)$ is empty, the vertex is evidently satisfiable and we return `sat` (Line 2). If $\kappa(v)$ is not empty, v is trivially conflicting and we return `unsat` (Line 1). Otherwise, we either construct children of v until one of them is satisfiable or until all children are constructed and all of them are conflicting or excluded as we cannot determine the satisfiability of their considered formulas with the virtual substitution. In the first case, this procedure returns `sat` (Line 6), and in the second case it either returns `unknown`, if for at least one child check_{VS} returned `unknown`, or it creates conflicts and returns `unsat` (Line 13). The main loop of Algorithm 10 first checks all non conflicting children (Line 5-6) and then constructs new children if possible. Here we distinguish again whether $\Phi(v)$ contains only constraints or not. In the first case, we want to create children which consider further test candidates (Line 11), after we have fixed (or updated) the variable to eliminate (Line 10). In the second case we make a case distinction. Here, we create an empty child v' (Line 16), i. e., $\Phi(v') = \emptyset$, and add the constraints, which correspond to the next case we want consider, to v' using the procedure extendCase (Line 17). If it returns false, all cases were considered and, if we could not determine the satisfiability for one of these cases, we cannot determine the satisfiability of the formulas represented by v (Line 20).

We already mentioned in the beginning of this section, that it would not be an option to store

all vertices of the search tree as depicted in Figure 4.1 when applying Algorithm 10. This is due to the fact that the number of vertices grows exponentially as the number of variables in the formula we check for satisfiability increases. Therefore, we delete in Algorithm 11 all children of the given vertex, if we know that it is conflicting and if it is not the root of the theory solver's VSST. When removing a constraint from a theory solver in an SMT solver belatedly, we can differ between two situations.

One situation is, that the SMT solver's SAT solver encounters a Boolean conflict, backtracks internally and then changes its Boolean assignment, which ultimately leads to the removal of constraints in the theory solver. In this case, it is more likely that the last theory solver call determined that its input is satisfiable. In our case, this implies that the VSST contains a path from its root to a valid vertex and therefore, after a belated removing of a constraints with $\text{remove}_{\text{VS}}$, it might not remove a lot of information which are stored in the VSST and even keep the solution path untouched (we explain in Section 4.2.5 what we exactly mean by solution path).

The second occasion where we remove constraints from a theory solver belatedly, is after it detects a theory conflict and the SMT solver resolves this conflict by changing the assignment in the SAT solver. This implies that the root of the VSST is conflicting. As we keep all children of the root in the case that it is conflicting, a belated removing of its considered formulas and, thereby, resolving of the root's conflict, is not the same as a complete restart. Many children might still be conflicting and as a consequence there are many test candidates, which we do not need to reconsider.

4.2.5 Creating a solution

Let us assume that we checked the set of constraints in the root v_1 of the theory solver's VSST $T_{\text{VS}} = (V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC})$ for satisfiability by invoking $\text{check}_{\text{VS}}(T_{\text{VS}}, v_1)$ and that this procedure returned `sat`. Then, V contains a valid vertex v_k $(k > 1)$ and a solution θ can be constructed if we collect the variable eliminations along the path

$$\omega := v_1 v_2 \dots v_{k-1} v_k$$

in T_{VS} from v_1 to v_k with

$$\theta(x) \quad = \quad \begin{cases} t & \text{, if } \exists i \in \{1, \dots, k-1\}.\ (\text{TC}((v_i, v_{i+1})) = t \ \wedge\ \text{elimvar}(v_i) = x) \\ 0 & \text{, otherwise,} \end{cases}$$

where we assume that $x \in \bigcup_{\varphi \in \Phi(v_i)} \text{Vars}(\varphi)$. Note, that it can happen that by the elimination of a variable x from a vertex v_i to the vertex v_{i+1} $(1 \leq i < k, \text{elimvar}(v_i) = x)$ further variables are dropped out. Therefore, a solution can assign any value to these variables, for instance 0.

The constructed solution θ maps variables to test candidates, which represent reals, but actually they can contain other variables. Moreover, test candidates can be a representative for

a sufficiently small value, i. e., $-\infty$, or contain a positive infinitesimal ϵ. Fortunately, we can construct a solution from this, which maps all variables to reals, using the ideas in [KSD16].

4.2.6 Generating small reasons for infeasibility

Assume that we check a conjunction of constraints $c_1 \wedge .. \wedge c_n$ with check_{VS} for satisfiability. In this scenario, by the use of the procedures add_{VS} and $\text{remove}_{\text{VS}}$ we ensure that the root v_1 of our theory solver's VSST T_{VS} contains exactly the constraints $c_1, .., c_n$. Then we invoke $\text{check}_{\text{VS}}(T_{\text{VS}}, v_1)$, which returns either `sat`, `unknown` or `unsat`. In the last case, we require an SMT-compliant theory solver also to specify infeasible subsets of $\{c_1, .., c_n\}$, which are preferably small.

Given any vertex v (including the root) in the VSST T_{VS}, we know, from the explanation in Section 4.2.4, that $\text{check}_{\text{VS}}(T_{\text{VS}}, v)$ returns false if v is conflicting. More precisely, we have the situation where $\kappa(v)$ contains conflicts. This is either the result of adding a formula φ being false to v or all children of v are constructed and all of them are conflicting. Then, we fill $\kappa(v)$ with infeasible subsets of $\Phi(v)$ based on the conflicts in the children. The following procedure implements the construction of these infeasible subsets.

bestConflictCoveringSets Let us assume that the given VSST T_{VS} is $(V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC})$. Furthermore, we assume that the given vertex v is not a leaf and all of its children $V' \subset V$ are conflicting, i. e., for all $v' \in V'$ it holds that $\kappa(v') \neq \emptyset$. In particular, there exists no $\varphi \in \Phi(v)$ with $\varphi = \text{false}$, as otherwise Algorithm 5 would have directly added φ to $\kappa(v)$ and we would not enter this procedure. It returns a set $K \subseteq \mathbb{P}(\Phi(v))$ of subsets of v's considered formulas such that for all of these subsets $K' \in K$ it holds that

$$\forall v' \in V'.\ \exists K_{v'} \in \kappa(v').\ \forall \varphi' \in K_{v'}.\ \exists N \in \text{orig}(v')(\varphi').\ N \subseteq K'. \tag{4.1}$$

It means that all sets in K cover at least one of the origins of each formula in at least one of the conflicts in each child v'. Note that an origin is in general a set of formulas in $\Phi(v)$ and that there could be more than one origin to choose from.

The construction of a *set covering* as we need it in order to fulfill Equation (4.1) turns into an NP-hard problem if we also require that it is optimal in any way, for instance to search for the smallest set covering [CTF00]. We omit this complexity by the use of an approximative approach to find some set coverings instead. The applied heuristics are presented in Section 5.2.

It remains to prove that the thereby constructed conflicts do indeed form infeasible subsets of the formulas in a vertex v. We again distinguish, whether $\Phi(v)$ contains constraints only or also formulas, which contain some Boolean complexity.

If $\Phi(v)$ contains only constraints, we use v for a variable elimination according to Corollary 1. From Theorem 3 it directly follows that all sets returned by bestConflictCoveringSets are infeasible.

Theorem 3 *Let $\varphi^{\mathbb{R}} = \bigwedge_{i=1}^{n} c_i$ be an unsatisfiable conjunction of real arithmetic constraints, which contains the variable x. Assume that it holds that*

$$\varphi^{\mathbb{R}} \overset{\text{Corollary 1}}{\Longleftrightarrow} \bigvee_{t \in \text{tcs}(x,\varphi^{\mathbb{R}})} (\varphi^{\mathbb{R}}[t//x] \wedge \text{sc}(t)). \tag{4.2}$$

Then for any subset $M \subseteq \{c_1, .., c_n\}$ of the constraints in $\varphi^{\mathbb{R}}$ it holds that

$$\forall t \in \text{tcs}(x, \varphi^{\mathbb{R}}).\ \exists N \subseteq \{c[t//x] |\ c \in M\} \cup \{\text{sc}(t)\}.\ \textstyle\bigwedge_{\psi \in N} \psi \text{ is unsatisfiable}$$
$$\Rightarrow$$
$$\textstyle\bigwedge_{c \in M} c \text{ is unsatisfiable.}$$

Proof 3 We show that if the right-hand side of the implication to prove does not hold, then it follows that the left-hand side does not hold. This implies Theorem 3.

$$\begin{aligned}
& \textstyle\bigwedge_{c \in M} c \text{ is satisfiable} \\
\overset{\text{Equation (4.2)}}{\Rightarrow}\ & \exists t \in \text{tcs}(x, \varphi^{\mathbb{R}}).\ \text{sc}(t) \wedge \textstyle\bigwedge_{c \in M} c[t//x] \text{ is satisfiable} \\
\Rightarrow\ & \exists t \in \text{tcs}(x, \varphi^{\mathbb{R}}).\ \forall N \subseteq \{c[t//x] |\ c \in M\} \cup \{\text{sc}(t)\}.\ \textstyle\bigwedge_{\psi \in N} \psi \text{ is satisfiable}
\end{aligned}$$

□

If v contains formulas in $\Phi(v)$, which are not constraints, we make a case distinction for v and the children represent all cases which needed to be checked in order to ensure that $\bigwedge_{\varphi \in \Phi(v)} \varphi$ is unsatisfiable (for more details we refer to Section 4.2.2). From Theorem 4 it directly follows that all sets returned by bestConflictCoveringSets are infeasible. Note that we propose an improvement in Section 4.4 as to how we might retrieve better conflicts for the formulas in $\Phi(v)$.

Theorem 4 *Let $\varphi^{\mathbb{R}} = \bigwedge_{i=1}^{n} \varphi_i$ be an unsatisfiable conjunction of real arithmetic formulas in NNF, where all literals are positive. Let $\varphi^{\mathbb{B}}$ be the Boolean abstraction of $\varphi^{\mathbb{R}}$ and $\text{abstr}^{\mathbb{B}}_{\varphi^{\mathbb{R}}}$ the corresponding Boolean abstraction mapping.*

Then for any subset $M \subseteq \{\varphi_1, .., \varphi_n\}$ of the formulas in $\varphi^{\mathbb{R}}$ with

$$C = \{c \in C_{\sim}(\varphi^{\mathbb{R}}) | \forall j \in \{1, .., n\}.\ \varphi_j \notin M \rightarrow c \notin C_{\sim}(\varphi_j)\} \quad (\textit{constraints that only occur in } M)$$

it holds that

$$\forall \alpha \in \Theta(\varphi^{\mathbb{B}}).\ \exists N \subseteq \{c \in C |\ \alpha(\text{abstr}^{\mathbb{B}}_{\varphi^{\mathbb{R}}}(c)) = \text{true}\}.\ \textstyle\bigwedge_{c \in N} c \text{ is unsatisfiable}$$
$$\Rightarrow$$
$$\textstyle\bigwedge_{\varphi \in M} \varphi \text{ is unsatisfiable.}$$

Proof 4 We again show that if the right-hand side of the implication to prove does not hold, then

it follows that its left-hand side does not hold implying Theorem 4.

$$
\begin{aligned}
& \varphi_M = \textstyle\bigwedge_{\varphi \in M} \varphi \quad \text{is satisfiable} \\
\overset{(*)}{\Rightarrow} \quad & \exists \alpha_M \in \Theta(\varphi_M^{\mathbb{B}}).\ \bigwedge_{\substack{c \in C \\ \alpha_M(\mathrm{abstr}^{\mathbb{B}}_{\varphi_M}(c)) = \mathrm{true}}} c \text{ is satisfiable} \\
\overset{(**)}{\Rightarrow} \quad & \exists \alpha \in \Theta(\varphi^{\mathbb{B}}).\ \bigwedge_{\substack{c \in C \\ \alpha(\mathrm{abstr}^{\mathbb{B}}_{\varphi^{\mathbb{R}}}(c)) = \mathrm{true}}} c \text{ is satisfiable} \\
\Rightarrow \quad & \exists \alpha \in \Theta(\varphi^{\mathbb{B}}).\ \forall N \subseteq \{c \in C \mid \alpha(\mathrm{abstr}^{\mathbb{B}}_{\varphi^{\mathbb{R}}}(c)) = \mathrm{true}\}.\ \textstyle\bigwedge_{c \in N} c \text{ is satisfiable}
\end{aligned}
$$

(∗): Note, that $\varphi_M^{\mathbb{B}}$ is the Boolean skeleton of φ_M and $\mathrm{abstr}^{\mathbb{B}}_{\varphi_M}$ the corresponding Boolean abstraction mapping. From φ_M being satisfiable it follows that there exists an assignment $\alpha_M^{\mathbb{R}} \in \Theta(\varphi_M)$. Then, we construct α_M as follows

$$
\alpha_M(b) = \begin{cases} \text{true} & \text{, if } \exists c \in C_\sim(\varphi^{\mathbb{R}}).\ \mathrm{abstr}^{\mathbb{B}}_{\varphi_M}(c) = b \ \wedge\ [\![c]\!]^{\alpha_M^{\mathbb{R}}} \equiv \text{true} \\ \text{false} & \text{, otherwise.} \end{cases}
$$

(∗∗): As $\varphi^{\mathbb{B}}$ is in NNF and all its literals occur only positively, $\varphi^{\mathbb{B}}$ is satisfiable (we can exclude $\varphi^{\mathbb{B}} = \text{false}$ as $\varphi_M^{\mathbb{B}}$, which consists of sub-formulas of $\varphi^{\mathbb{B}}$, is satisfiable). Then there exists an assignment $\alpha' \in \Theta(\varphi^{\mathbb{B}})$. As $\varphi^{\mathbb{B}}$ is in NNF and all its literals occur only positively, the assignment

$$
\alpha(b) = \begin{cases} \text{true} & \text{, if } \alpha_M(b) = \text{true} \\ \alpha'(b) & \text{, otherwise} \end{cases}
$$

is also satisfying $\varphi^{\mathbb{B}}$. Furthermore, we assume, w. l. o. g., that $\mathrm{abstr}^{\mathbb{B}}_{\varphi_M}$ maps the constraints, which are mapped by $\mathrm{abstr}^{\mathbb{B}}_{\varphi_M}$ to a Boolean variable, to the same Boolean variable. Then, it holds that

$$
\bigwedge_{\substack{c \in C \\ \alpha(\mathrm{abstr}^{\mathbb{B}}_{\varphi^{\mathbb{R}}}(c)) = \mathrm{true}}} c \quad = \quad \bigwedge_{\substack{c \in C \\ \alpha_M(\mathrm{abstr}^{\mathbb{B}}_{\varphi_M}(c)) = \mathrm{true}}} c
$$

is satisfiable. □

4.2.7 Example

As the data structure of the presented SMT compliant theory solver is a directed tree, we illustrate it in this example as such. Let $T_{\mathrm{VS}} = (V, E, \Phi, \mathrm{orig}, \kappa, \mathrm{elimvar}, \mathrm{used}, \mathrm{incompl}, \mathrm{TC})$ be the theory solver's VSST and the vertices $v_1, v_2 \in V$ be connected by $(v_1, v_2) \in E$, which is labeled

by $\mathrm{TC}((v_1, v_2)) = (t, O_t)$. Then, we illustrate v_1, v_2 and (v_1, v_2) as in Figure 4.2. In the case $\mathrm{incompl}(v) = \text{true}$ holds for a vertex $v \in V$, v's borders are drawn as a dashed line.

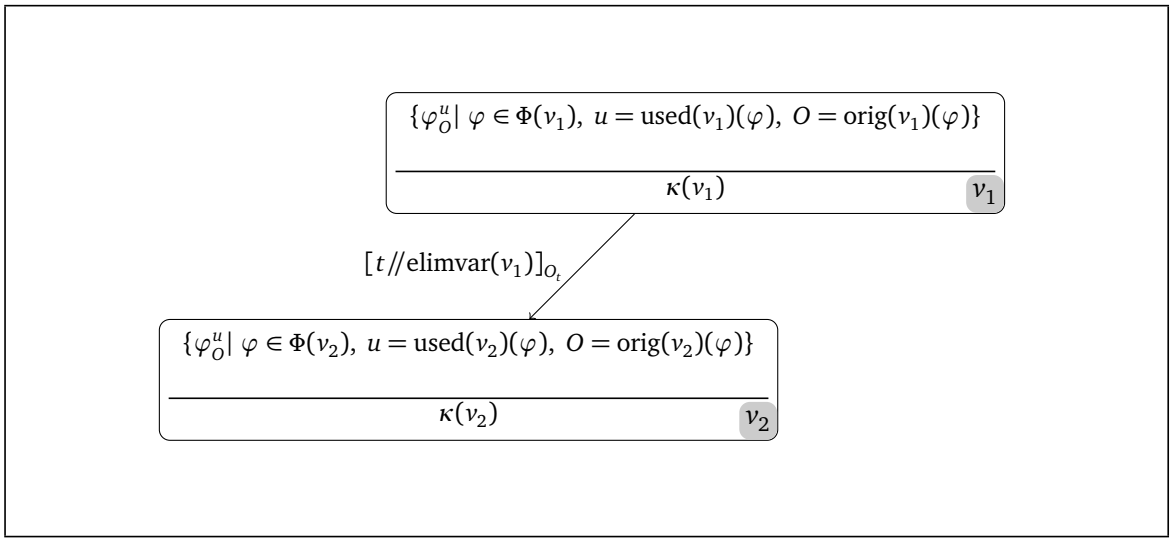


Figure 4.2: Illustration of how we present two vertices $v_1, v_2 \in V$ of a VSST $T_{\mathrm{VS}} = (V, E, \Phi, \mathrm{orig}, \kappa, \mathrm{elimvar}, \mathrm{used}, \mathrm{incompl}, \mathrm{TC})$, which are connected by a directed edge $(v_1, v_2) \in E$ that is labeled by $\mathrm{TCS}(v_1, v_2) = (t, O_t)$. In the case that $\mathrm{incompl}(v) = \text{true}$ holds for a vertex $v \in V$, v's borders are drawn as a dashed line.

We now simulate a run of an SMT solver with an SMT-compliant theory solver based on this chapter. The input formula of this SMT solver is

$$\varphi \quad = \quad x_1^2 - x_2 - 1 = 0 \ \wedge\ 6x_1 - 2x_2 - 3 \geq 0 \ \wedge\ (x_1^2 + x_2^2 - 2 = 0 \ \vee\ 2x_1 + x_2 + 2 < 0).$$

Note that we sometimes simplify the result of the occurring virtual substitutions slightly while conserving the example's expressiveness.

The SMT solver's SAT solver considers φ's Boolean abstraction $b_1 \wedge b_2 \wedge (b_3 \vee b_4)$. In the SAT solver's decision level 0, we assign b_1 and b_2 to true. As there is no Boolean conflict, we add the corresponding constraints $x_1^2 - x_2 - 1 = 0$ and $6x_1 - 2x_2 - 3 \geq 0$ to the theory solver and ask it whether $x_1^2 - x_2 - 1 = 0 \ \wedge\ 6x_1 - 2x_2 - 3 \geq 0$ is satisfiable. The theory solver's VSST is initially

$$\begin{aligned} T_{\mathrm{VS}} \quad &= \quad (V, E, \Phi, \mathrm{orig}, \kappa, \mathrm{elimvar}, \mathrm{used}, \mathrm{incompl}, \mathrm{TC}) \\ &:= \quad (\{v_1\},\ \emptyset,\ \{(v_1, \emptyset)\},\ \{(v_1, \emptyset)\},\ \{(v_1, \emptyset)\},\ \{(v_1, \emptyset)\},\ \emptyset,\ \{(v_1, \mathrm{false})\},\ \emptyset) \end{aligned}$$

and, hence, consists of the valid vertex v_1.

Now we first invoke $\mathrm{add}_{\mathrm{VS}}(T_{\mathrm{VS}}, v_1, x_1^2 - x_2 - 1 = 0, \{x_1^2 - x_2 - 1 = 0\})$ yielding the VSST on the left of Figure 4.3 followed by the call $\mathrm{add}_{\mathrm{VS}}(T_{\mathrm{VS}}, v_1, 6x_1 - 2x_2 - 3 \geq 0, \{6x_1 - 2x_2 - 3 \geq 0\})$ which results in the VSST on the right of this figure. Considering Algorithm 5, we add the respective constraint in both cases to $\Phi(v_1)$ and set their origins in $\mathrm{orig}(v_1)$ (Line 5-6). As the elimination variable of v_1 is not yet fixed and v_1 has no children, there is nothing more to be done.

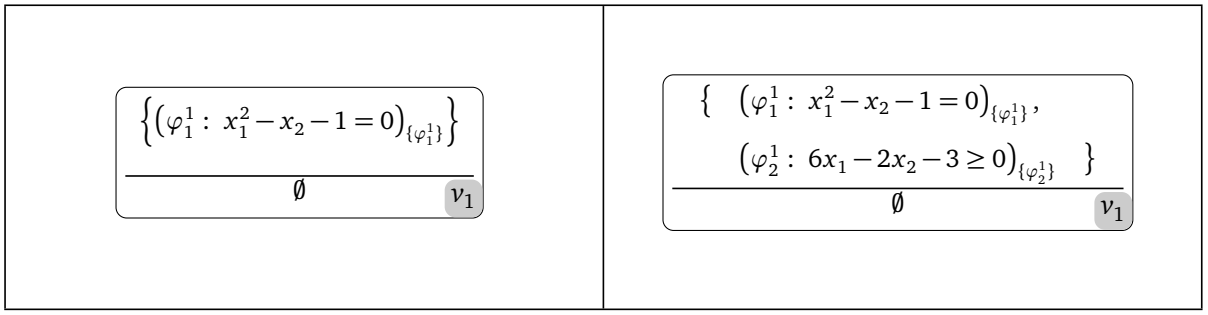


Figure 4.3: The results of first adding $x_1^2 - x_2 - 1 = 0$ to the VSST $T_{\text{VS}} = (V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC}) := (\{v_1\}, \emptyset, \{(v_1, \emptyset)\}, \{(v_1, \emptyset)\}, \{(v_1, \emptyset)\}, \{(v_1, \emptyset)\}, \emptyset, \{(v_1, \text{false})\}, \emptyset)$ by invoking $\text{add}_{\text{VS}}(T_{\text{VS}}, v_1, x_1^2 - x_2 - 1 = 0, \{x_1^2 - x_2 - 1 = 0\})$ (on the left) and then adding $6x_1 - 2x_2 - 3 \geq 0$ by invoking $\text{add}_{\text{VS}}(T_{\text{VS}}, v_1, 6x_1 - 2x_2 - 3 \geq 0, \{6x_1 - 2x_2 - 3 \geq 0\})$.

Note, that we label the formulas in a vertex within this example, e. g., $x_1^2 - x_2 - 1 = 0$ is labeled by φ_1^1 and $6x_1 - 2x_2 - 3 \geq 0$ is labeled by φ_2^1. We use these labels in order to identify the formulas. In order to keep things simple in the course of this example, we reuse these labels and also the names of the vertices, after deleting them.

After adding these two constraints we invoke their satisfiability check, i. e., $\text{check}_{\text{VS}}(T_{\text{VS}}, v_1)$. Considering Algorithm 10, we enter its main loop, as v_1 is neither conflicting nor valid. It has no children, so we skip the inner loop and, since $\Phi(v_1)$ contains only constraints, we fix an elimination variable (Line 10). For this purpose we invoke Algorithm 8. Let us assume that we choose to eliminate x_1 first. The precise heuristics for the choice of the elimination variable is introduced later in Section 5.1. As the elimination variable has not yet been fixed for v_1, we set it to x_1 and initialize the flag for all constraints in $\Phi(v_1)$, which indicates whether a constraint has already been used for test candidate creation, by false (Line 6). Afterwards, Algorithm 10 invokes Algorithm 9, which chooses a constraint that has not yet been used for test candidates creation, i. e., is mapped by $\text{used}(v_1)$ to false. In our case, both constraints fulfill this criteria and in the context of this example, we always choose the first constraint. The precise heuristics for the choice of the next constraint providing test candidates is also introduced later in Section 5.1. Therefore, we try to construct the test candidates for x_1 in $x_1^2 - x_2 - 1 = 0$. As the degree of x_1 in this constraint is less than or equal to 2, we can construct the test candidates $-\infty$, $\sqrt{x_2 + 1}$ and $-\sqrt{x_2 + 1}$. For each of them, we create a new child v_i $(2 \leq i \leq 4)$ of v_1, label the edge to it by the respective test candidate t and its origin, i. e., $\varphi_1^1 :\ x_1^2 - x_2 - 1 = 0$, and add the result of the virtual substitution of x_1 by t in each constraint $c \in \Phi(v_1)$ to v_i. We achieve this by invoking $\text{add}_{\text{VS}}(T_{\text{VS}}, (v_1, v_i), c[t /\!/ x_1], \{c\})$, which adds $c[t /\!/ x_1]$ to $\Phi(v_i)$ and sets its origin to $\{c\}$. Afterwards we set $\text{used}(v_1)(\varphi_1^1)$ to true, which means that we have used φ_1^1 for the provision of test candidates. The resulting vertices v_2, v_3 and v_4 are illustrated in Figure 4.4. Here, we can see that the child v_2 which we constructed for $-\infty$ is already conflicting, i. e., $\kappa(v_2) \neq \emptyset$, which is due to $\varphi_1^1[-\infty /\!/ x_1]$ and $\varphi_2^1[-\infty /\!/ x_1]$ $(\varphi_2^1 :\ 6x_1 - 2x_2 - 3 \geq 0)$ both resulting in false.

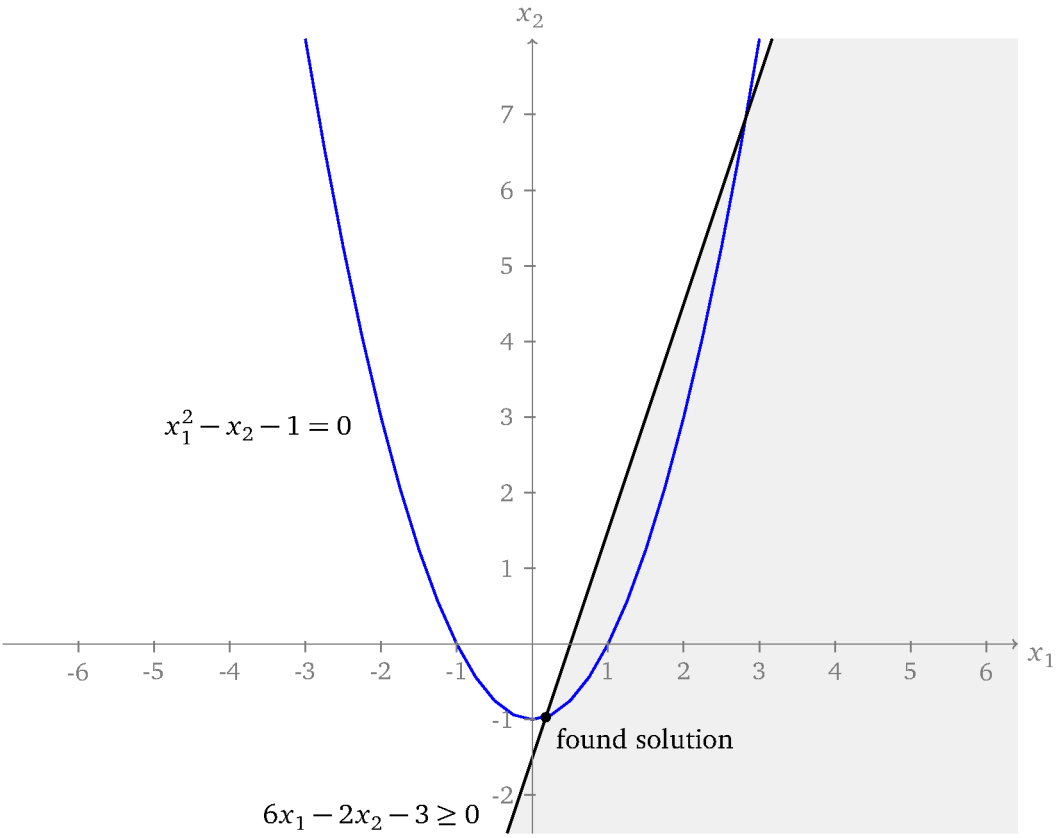


Figure 4.4: The solution sets of the constraints $x_1^2 - x_2 - 1 = 0$ and $6x_1 - 2x_2 - 3 \geq 0$ and the solution found by check_{VS} as illustrated in Figure 4.5.

Therefore, we add the formula $\varphi_1^2 :$ false once to $\Phi(v_2)$ and specify its origin by both constraints, i. e., $\text{orig}(v_2)(\varphi_1^2) = \{\{\varphi_1^1\}, \{\varphi_2^1\}\}$.

At this point, Algorithm 9 returns true, and we continue in Algorithm 10 reentering its main loop. This time, v_1 has two non-conflicting children. We choose v_3, but in general we would also make this choice heuristically as it is introduced later in Section 5.1, and recursively invoke check_{VS} for this vertex. Similar as for v_1 in the beginning of this example, we enter the main loop and skip the inner loop. In contrast to $\Phi(v_1)$, however, $\Phi(v_3)$ contains formulas, which are not constraints, i. e.,

$$\varphi_2^3 \quad = \quad 4x_2^2 - 24x_2 - 27 \leq 0 \ \vee \ (2x_2 + 3 < 0 \ \wedge \ 4x_2^2 - 24x_2 - 27 \geq 0).$$

Therefore, we have to make a case distinction for this vertex instead of eliminating a variable. For this purpose we create the empty child v_5 for v_3 and fill it with the next case invoking $\text{extendCase}(T_{\text{VS}}, (v_3, v_5))$. If we take a look at Algorithm 6 and bear in mind that v_3 has no further children, this case consists of the constraints, which are assigned to true by a satisfying assignment for the Boolean abstraction of the conjunction of the formulas in v_3. In Figure 4.5,

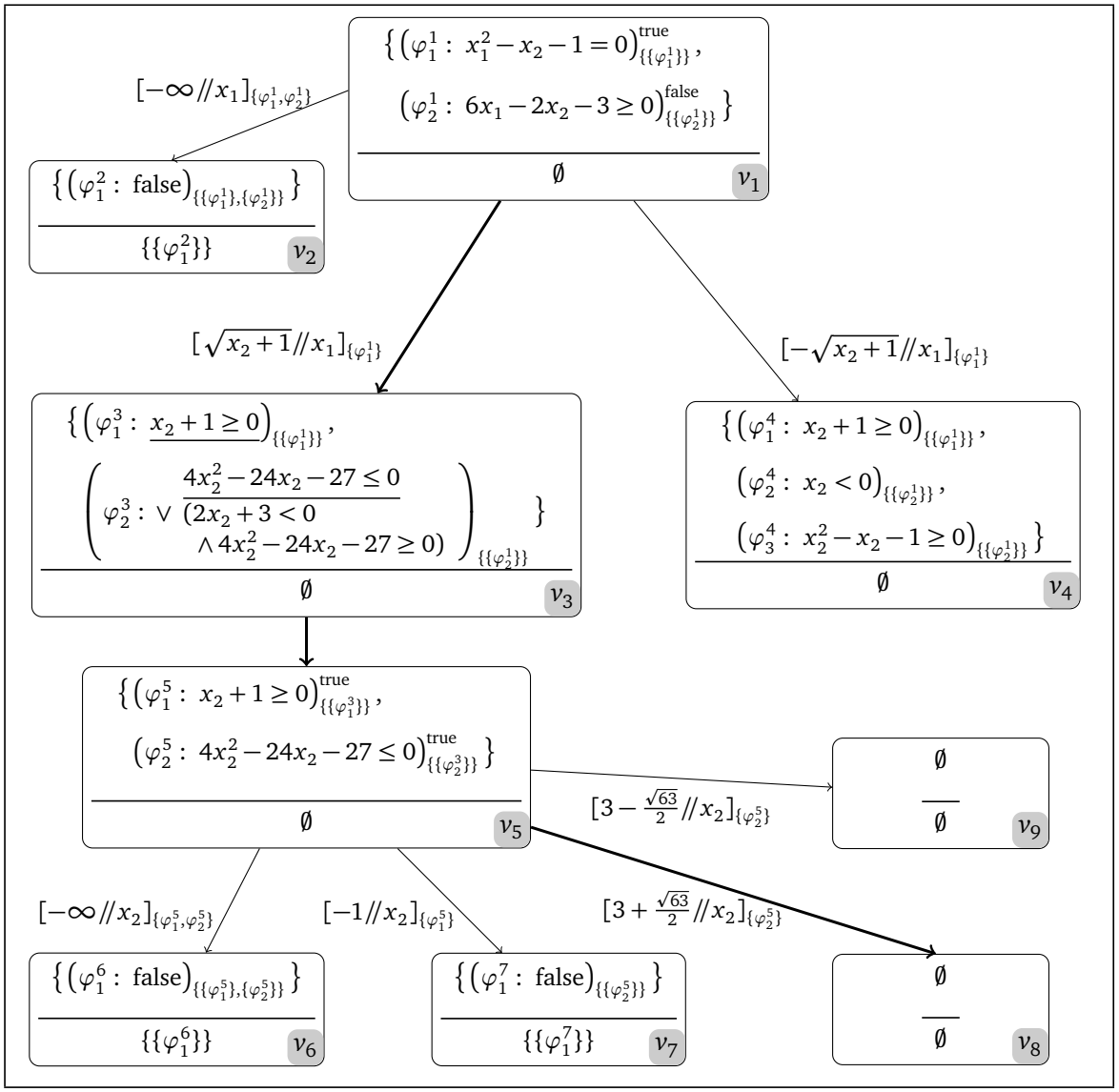


Figure 4.5: The resulting VSST T_{VS} of the satisfiability check of $x_1^2 - x_2 - 1 = 0 \wedge 6x_1 - 2x_2 - 3 \geq 0$, which we achieved by invoking $\text{check}_{\text{VS}}(T_{\text{VS}}, v_1)$ with T_{VS} and v_1 (root of T_{VS}) being initially as on the right of Figure 4.3.

we indicate these constraints by underlining them. We add them to v_5 using add_{VS}, where the origins of each added constraint are the formulas in v_3 in which they occur (Line 15-16).

Afterwards, extendCase returns true and we reenter the main loop for vertex v_3. It now has exactly one child, i. e., v_5, which represents the just added case and, as v_5 is not conflicting, we invoke $\text{check}_{\text{VS}}(T_{\text{VS}}, v_5)$. This vertex has no children initially and $\Phi(v_5)$ contains only constraints, which forms the same case as it was for v_1 in the beginning of this example. Hence, we first fix the variable to eliminate which must be x_2 as it is the only remaining variable.

Then we choose the first constraint which has not been used in order to provide test candidates,

i. e., $\varphi_1^5 : \ x_2 + 1 \geq 0$, and create a child for v_5 for each of the two test candidates $-\infty$ and -1, which it provides, in the same way as before. The formulas in the obtained children v_6 and v_7 are again the results of virtually substituting the two constraints in v_5, i. e., φ_1^5 and $\varphi_2^5 : \ 4x_2^2 - 24x_2 - 27 \leq 0$, by the test candidate, respectively. For $-\infty$ it yields false in both cases making v_5 conflicting and for the test candidate -1 the virtual substitution results in true for φ_1^5, so we do not add it, and in false for φ_2^5 also making v_6 conflicting. Therefore, we take the next constraint φ_2^5 in v_5 for test candidate creation into account. It provides the three test candidates $-\infty$, $3 + \frac{\sqrt{63}}{2}$ and $3 - \frac{\sqrt{63}}{2}$. For the first one there already exists v_5's child v_6. Considering Algorithm 6, we first add φ_2^5 to the origins of $-\infty$ by updating $\text{TC}((v_5, v_6))$ and then add the result of $\varphi_2^5[-\infty // x_2]$ to $\Phi(v_6)$ (Line 8-9). As it is false, we update the origins of $\varphi_1^6 :$ false.

For the other two test candidates $3 + \frac{\sqrt{63}}{2}$ and $3 - \frac{\sqrt{63}}{2}$ we create the two new children v_7 and v_8. As virtually substituting x_2 by both of these test candidates in each constraint in $\Phi(v_5)$ results in true respectively, there is nothing to be added to neither v_7 nor v_8. Now we have two non-conflicting children of v_5, so we recursively call check_{VS} for one of them after reentering the main loop of Algorithm 10. Let us assume that we choose v_7. As it is a valid vertex, i. e., $\Phi(v_7) = \emptyset$, we immediately return `sat` (Line 2) and jump back to the call $\text{check}_{\text{VS}}(T_{\text{VS}}, v_5)$ (Line 6). It also returns `sat`, as the recursively invoked call returned `sat`. We jump back to the call $\text{check}_{\text{VS}}(T_{\text{VS}}, v_3)$ and return `sat` again for the same reason. Now we are in the outermost call $\text{check}_{\text{VS}}(T_{\text{VS}}, v_1)$, where we also return `sat`, which means that the theory solver determines the satisfiability of $x_1^2 - x_2 - 1 = 0 \wedge 6x_1 - 2x_2 - 3 \geq 0$. The found solution of this formula is given by the pairs of elimination variables and test candidates along the path from the root of the VSST to the found valid vertex. In Figure 4.4, this path consists of the edges in bold, thus the found solution is $\{(x_1, \ \sqrt{4 + \frac{\sqrt{63}}{2}}), \ (x_2, \ 3 + \frac{\sqrt{63}}{2})\}$. Figure 4.4 illustrates the solution spaces defined by the two constraints and the found solution.

The SMT solver now knows that the conjunction of the constraints corresponding to the partial assignment of its input formula φ's Boolean abstraction is satisfiable. As a consequence, the SMT solver's SAT solver can enter the next decision level afterwards. Let us assume it assigns b_3 to false. It then implies by the use of Boolean constraint propagation that b_4 must be assigned to true. This finishes the decision level and also forms a full assignment of φ's Boolean abstraction. However, we still need to check whether the conjunction of the constraints, corresponding to the Boolean variables which are assigned to true, is satisfiable. Thus we have to check the satisfiability of

$$x_1^2 - x_2 - 1 = 0 \ \wedge \ 6x_1 - 2x_2 - 3 \geq 0 \ \wedge \ 2x_1 + x_2 + 2 < 0$$

and need to add $2x_1 + x_2 + 2 < 0$ to the theory solver, beforehand. For this purpose we invoke $\text{add}_{\text{VS}}(T_{\text{VS}}, v_1, 2x_1 + x_2 + 2 < 0, \{2x_1 + x_2 + 2 < 0\})$.

Compared to the addition of the two constraints in the beginning of this example, the VSST of the theory solver stores the result of the last satisfiability check instead of just consisting of the root vertex v_1. Therefore we do not only need to add the constraint to v_1 but also propagate

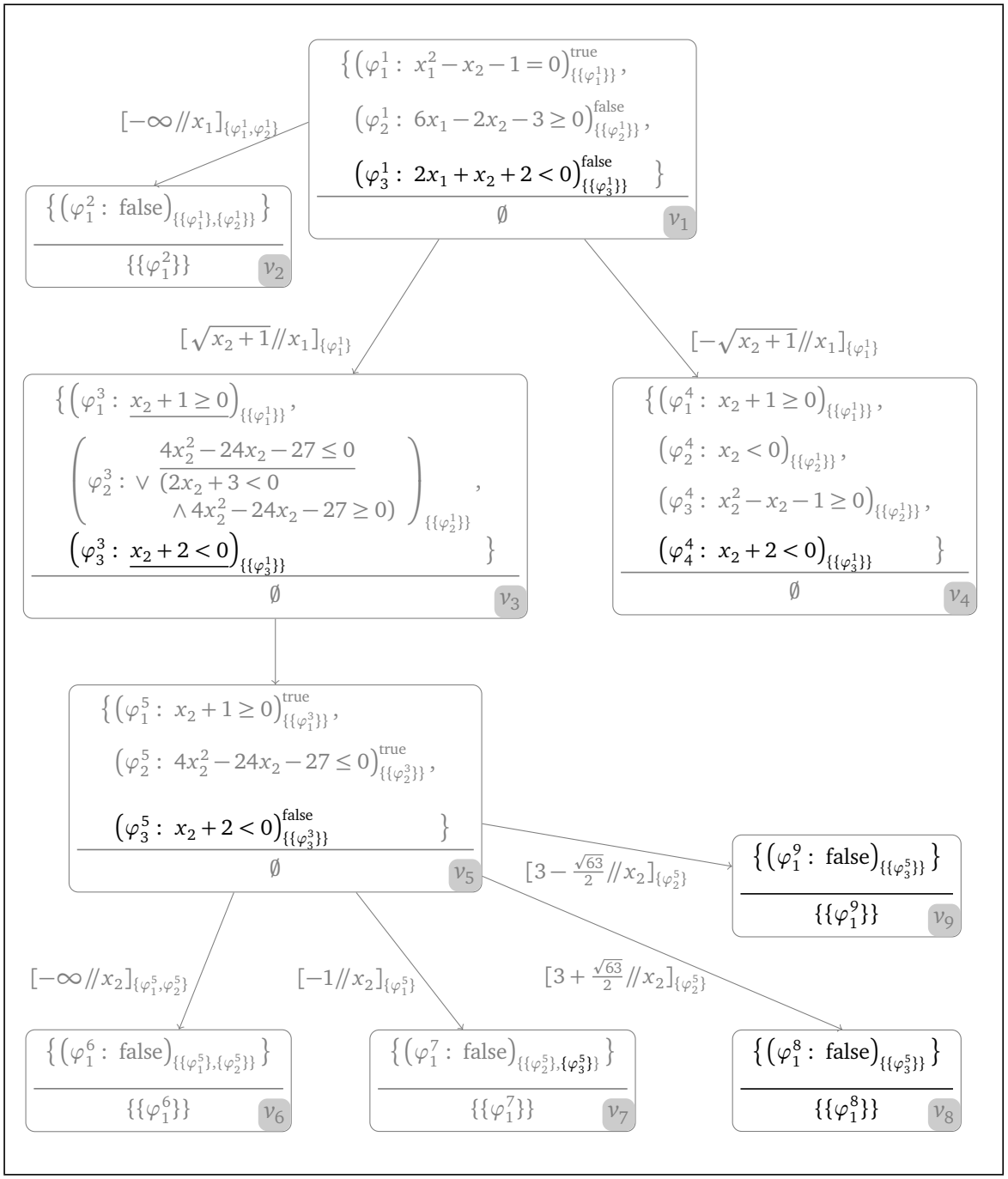


Figure 4.6: The resulting VSST T_{VS} after adding the constraint $2x_1 + x_2 + 2 < 0$ to the VSST T_{VS} of Figure 4.5 by invoking $\text{add}_{\text{VS}}(T_{\text{VS}}, v_1, 2x_1 + x_2 + 2 < 0, \{2x_1 + x_2 + 2 < 0\})$.

it throughout its successors. This means that we first add $2x_1 + x_2 + 2 < 0$ to v_1 as we have explained before, but also need to take into account the second phase of Algorithm 5 (Line 24-30). As we use v_1 for eliminating the variable x_1, its children represent the results of virtually substituting x_1 by test candidates provided by the constraints in $\Phi(v_1)$. Therefore, we need to

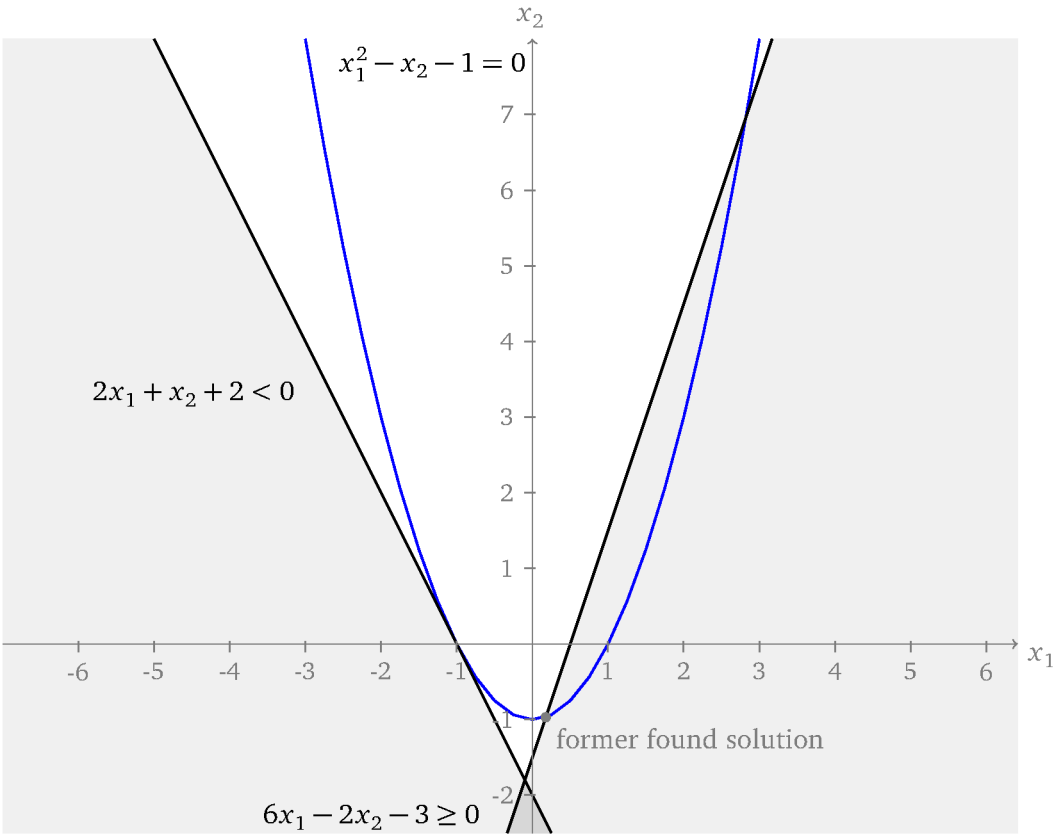


Figure 4.7: The solution sets of the constraints $x_1^2 - x_2 - 1 = 0$, $6x_1 - 2x_2 - 3 \geq 0$ and $2x_1 + x_2 + 2 < 0$.

extend the considered virtual substitution result of each child v_i $(2 \leq i \leq 4)$ by the result of virtually substituting x_1 by the test candidate on the label of the edge to this child in the added constraint $\varphi_3^1 : \ 2x_1 + x_2 + 2 < 0$. Thereby we call add_{VS} recursively.

In our case, v_1 has three children. The first one, v_2, has been created for the test candidate $-\infty$. The result of $\varphi_3^1[-\infty // x_1]$ is true, hence we do not add anything to v_2. The second child of v_1, v_3, has been created for the test candidate $\sqrt{x_2 + 1}$ and, here, the result of $\varphi_3^1[\sqrt{x_2 + 1} // x_1]$ is $x_2 + 2 < 0$. Therefore we invoke $\text{add}_{\text{VS}}(T_{\text{VS}}, v_3, x_2 + 2 < 0, \{\varphi_3^1\})$. The vertex v_3 makes a case distinction, hence we extend the currently considered case represented by v_5 after adding $\varphi_3^3 : \ x_2 + 2 < 0$ to $\Phi(v_5)$. As it is a single constraint, we have to extend the case by it, yielding the recursive call $\text{add}_{\text{VS}}(T_{\text{VS}}, v_5, x_2 + 2 < 0, \{\varphi_3^3\})$. Similar to v_1, v_5 is used for a variable elimination but this time we eliminate the variable x_2. Thus, we need to extend the virtual substitution results v_5's children consider. The first child considers the result of virtually substituting by the test candidate $-\infty$. As $x_2 + 2 < 0[-\infty // x_2]$ is true, there is nothing to be added. For all the other children, virtually substituting x_2 by the test candidates they consider, results in false. Therefore, we have to add this result to the children, which makes them all conflicting.

We have finished propagating the constraint $x_2 + 2 < 0$ in v_5 and as a consequence we also

finish the propagation of $x_2 + 2 < 0$ in v_3. We finish the call $\text{add}_{\text{VS}}(T_{\text{VS}}, v_1, 2x_1 + x_2 + 2 < 0, \{2x_1 + x_2 + 2 < 0\})$ adding the result of $2x_1 + x_2 + 2 < 0[-\sqrt{x_2+1}//x_1]$, which is also $x_2 + 2 < 0$, to the only, not yet considered child v_4. In Figure 4.6, the resulting VSST can be seen. All of the information which we gained in the previous satisfiability check with our theory solver is preserved. However, the solution is not valid anymore, which makes perfect sense, as the added constraint clearly excludes this solution. Figure 4.7, illustrates the solution spaces of all three constraints $x_1^2 - x_2 - 1 = 0$, $6x_1 - 2x_2 - 3 \geq 0$ and $2x_1 + x_2 + 2 < 0$.

After adding the constraint $2x_1 + x_2 + 2 < 0$, we now invoke the satisfiability check for the conjunction of the three constraints in $\Phi(v_1)$. We invoke $\text{check}_{\text{VS}}(T_{\text{VS}}, v_1)$ where T_{VS} is initially in the state as illustrated in Figure 4.6. As v_1 is not conflicting nor valid and has two non-conflicting children, we choose one of them and recursively invoke check_{VS}. Let us assume that we choose v_3. Just like for v_1, we can find a non-conflicting child of v_3, which is the only child v_5. This vertex, however, has only conflicting children, so we try to create a new one. We use v_5 for the elimination of the variable x_2, thus we try to find a constraint in $\Phi(v_5)$ which can provide test candidates for x_2. The only constraint which has not yet been used for test candidate creation is $\varphi_5^3 : \; x_2 + 2 < 0$. It provides the test candidates $-\infty$ and $-2 + \epsilon$ for x_2. The first one is already considered by v_6, so we only extend the origins in $\text{TC}((v_5, v_6))$ by φ_5^3. The second one creates the new child v_{10} and the result of virtually substituting x_2 in the constraints of $\Phi(v_5)$ by $-2 + \epsilon$ is always false. Hence, v_{10} is also conflicting and for the first time we obtain a case where all constraints of a vertex for variable elimination were considered for test candidate creation and all of them failed to be a solution.

This situation can be seen in Figure 4.8. In Algorithm 10, we enter Line 13 and, thus, invoke the procedure createConflicts. According to Theorem 2 we know that the conjunction of the constraints in $\Phi(v_5)$ is unsatisfiable, therefore we create in this procedure infeasible subsets of $\Phi(v_5)$ and add them to $\kappa(v_5)$. The procedure thereby applies Theorem 3, hence, we can construct infeasible subsets of $\kappa(v_5)$ by finding set coverings of the conflicts in v_5's children after mapping the formulas they contain to their origins in v_5. In our case, all conflicts in v_5's children consist only of the formula false, thus, for each child their conflict after mapping the formulas they contain to their origins in v_5 are precisely the origins of false. Considering that

$$\Phi(v_5) \;=\; \{\overbrace{x_2 + 1 \geq 0}^{\varphi_1^5},\; \overbrace{4x_2^2 - 24x_2 - 27 \leq 0}^{\varphi_2^5},\; \overbrace{x_2 + 2 < 0}^{\varphi_3^5}\},$$

we need to find a set covering of

$$\begin{aligned} M_1 &= \{\{\varphi_1^5\}, \{\varphi_2^5\}\} \\ M_2 &= \{\{\varphi_2^5\}, \{\varphi_3^5\}\} \\ M_3 &= \{\{\varphi_3^5\}\} \\ M_4 &= \{\{\varphi_3^5\}\} \\ M_5 &= \{\{\varphi_1^5\}, \{\varphi_2^5\}, \{\varphi_3^5\}\} \end{aligned}$$

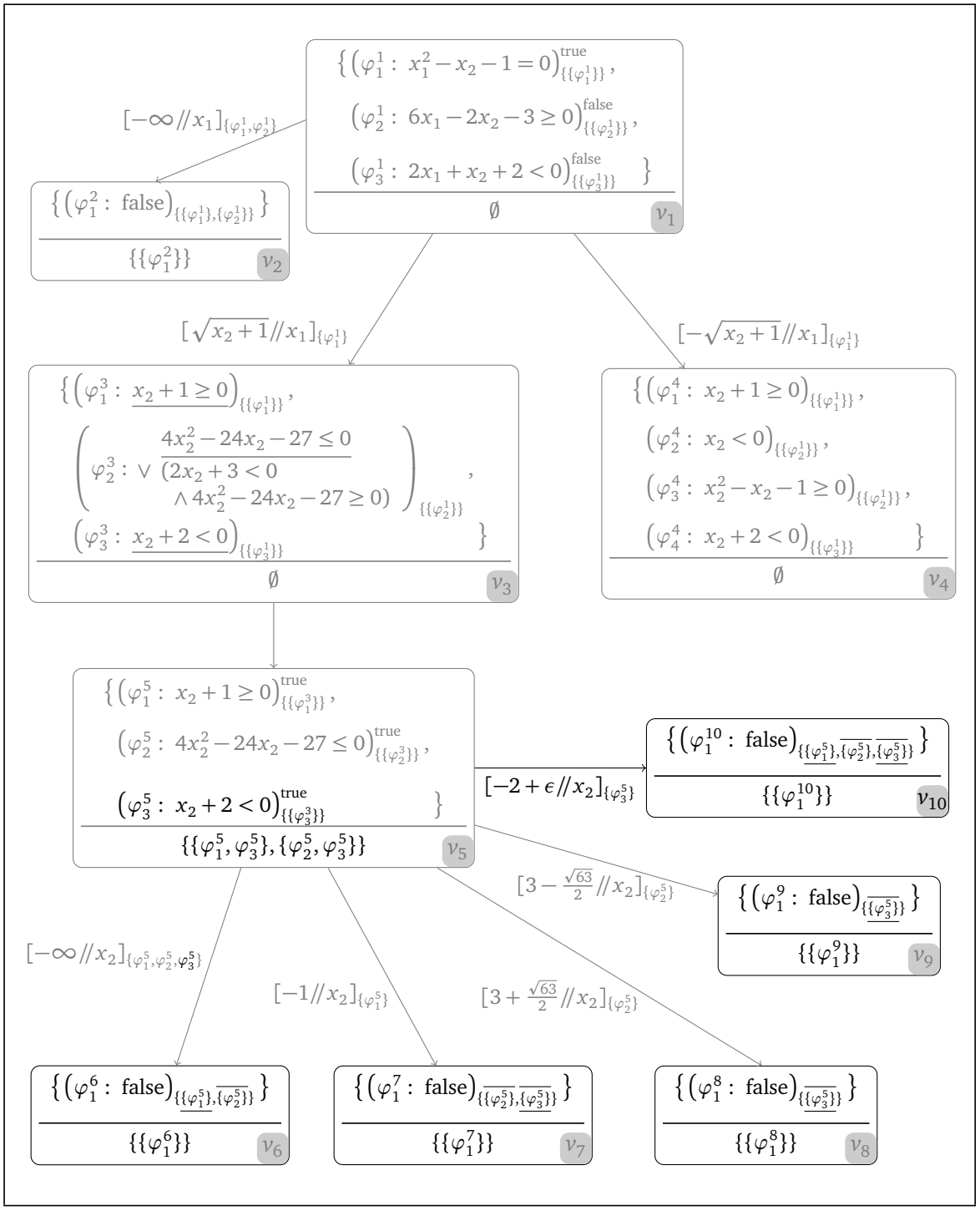


Figure 4.8: The resulting VSST T_{VS} until the *first* conflicting intermediate vertex, i. e., v_5, is reached by the satisfiability check of $x_1^2 - x_2 - 1 = 0 \wedge 6x_1 - 2x_2 - 3 \geq 0 \wedge 2x_1 + x_2 + 2 < 0$, which we invoked with $\text{check}_{\text{VS}}(T_{\text{VS}}, v_1)$ where T_{VS} and v_1 are initially as in Figure 4.6.

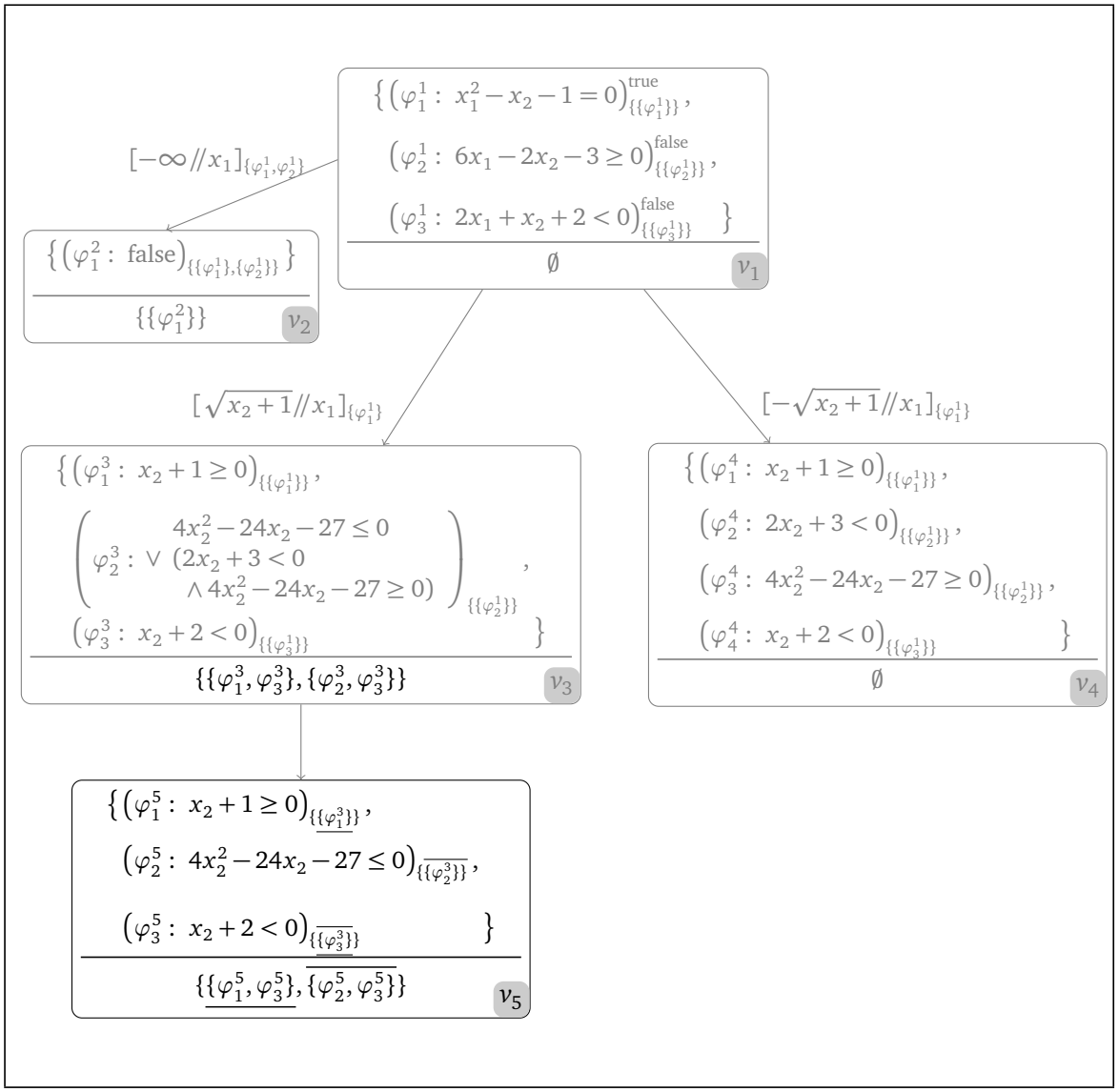


Figure 4.9: The resulting VSST T_{VS} until the *second* conflicting intermediate vertex, i. e., v_3, is reached by the satisfiability check of $x_1^2 - x_2 - 1 = 0 \wedge 6x_1 - 2x_2 - 3 \geq 0 \wedge 2x_1 + x_2 + 2 < 0$, which we invoked with $\text{check}_{\text{VS}}(T_{\text{VS}}, v_1)$ where T_{VS} and v_1 are initially as in Figure 4.8.

that is a set $M \subseteq \Phi(v_5)$ such that for all $1 \leq i \leq 5$ there exists a set $M' \in M_i$ with $M' \subseteq M$. The two smallest set coverings are $\{\varphi_1^5, \varphi_3^5\}$ and $\{\varphi_2^5, \varphi_3^5\}$. How we can compute set coverings is further explained in Section 4.2.6. In Figure 4.8, we illustrate that $\{\varphi_1^5, \varphi_3^5\}$ covers a set in each M_i $(1 \leq i \leq 5)$ by underlining the covered sets. For $\{\varphi_2^5, \varphi_3^5\}$ we use an overline instead. According to Algorithm 11 we add both set coverings as infeasible subsets of $\Phi(v_5)$ to $\kappa(v_5)$ and delete all children of v_5. Thereby we remove some information, but we keep two reasons for the infeasibility of the conjunction of the constraints in $\Phi(v_5)$. To keep both of them can provide more possibilities to create set coverings in case that conjunction of the formulas in v_5's father

is also unsatisfiable. Moreover, in a later stage of the SMT solver's satisfiability check we might remove constraints from the VSST belatedly. If a conflict of v_5 remains after this removal, we do not need to reconsider v_5 as it is still conflicting.

Now we continue check_{VS} for the vertex v_3. We reenter the main loop of Algorithm 10 and, since we use this vertex to make a case distinction, we try to achieve a new case by creating an empty vertex v_6 and invoking $\text{extendCase}(T_{\text{VS}}, (v_3, v_6))$ (Line 17). This procedure tries to find a satisfying assignment of the Boolean abstraction of the formula

$$\psi := \bigwedge_{\varphi \in \Phi(v_3)} \varphi \wedge \neg\Big(\bigwedge_{c \in K_{v_5}} c\Big)$$

where $K_{v_5} \in \kappa(v_5)$ is chosen heuristically. Let us assume that we choose $K_{v_5} = \{x_2+1 \geq 0,\ x_2+2 < 0\}$, then the Boolean abstraction of ψ is

$$b_5 \wedge (b_6 \vee (b_7 \wedge b_6)) \wedge b_8 \wedge \neg(b_5 \wedge b_8).$$

This formula is already unsatisfiable and as a consequence extendCase cannot find a new case in order to fill the just created empty vertex v_6 with the corresponding constraints. Therefore this procedure deletes v_6 and returns false (Line 20). We observe at this point, that the small infeasible subset, which we created for v_5, saves us from considering another case. It becomes clear that we benefit from good infeasible subsets even in intermediate vertices and that the quality and variety of the infeasible subsets in the children decides how good the infeasible subsets of their father will be. In Algorithm 10 we now fulfill the condition at Line 17 for the first time, meaning that all cases were considered. As we did not encounter any constraint with a degree higher than 2 in any variable, we conclude that the conjunction of the formulas in $\Phi(v_3)$ must be unsatisfiable. We simply fill $\kappa(v_3)$ with infeasible subsets of $\Phi(v_3)$ using the procedure createConflicts again and return `unsat`. Figure 4.9 shows the VSST with the two resulting infeasible subsets in $\kappa(v_3)$.

We are now back in the outermost check_{VS} call for the root v_1 of the VSST. It still has one non-conflicting child, i. e., v_4, so we invoke $\text{check}_{\text{VS}}(T_{\text{VS}}, v_4)$. In Figure 4.10, we see that it results once more in the situation that all test candidates are constructed and none of them succeeded to be part of a solution. The procedure createConflicts yields two infeasible subsets in v_4.

We reenter the main loop of Algorithm 10 for v_1 and, since all children are conflicting, we have to try to construct new children. We choose a constraint in $\Phi(v_1)$ which has not yet been used in order to provide test candidates, so let us assume that we take $\varphi_2^1 :\ 6x_1 - 2x_2 - 3 \geq 0$ for this purpose. For the first test candidate, which it provides for x_1, i. e., $-\infty$, we extend the corresponding existing child v_2 and for the second test candidate, which it provides for x_1, i. e., $\frac{2x_2+3}{6}$, we create the new child v_5. Afterwards, we invoke $\text{check}_{\text{VS}}(T_{\text{VS}}, v_5)$ which ultimately leads to the situation that all children are constructed and all of them are conflicting. We can see the gained infeasible subsets for v_5 in Figure 4.11.

Again we reenter the main loop of Algorithm 10 for v_1. All children are conflicting, thus we

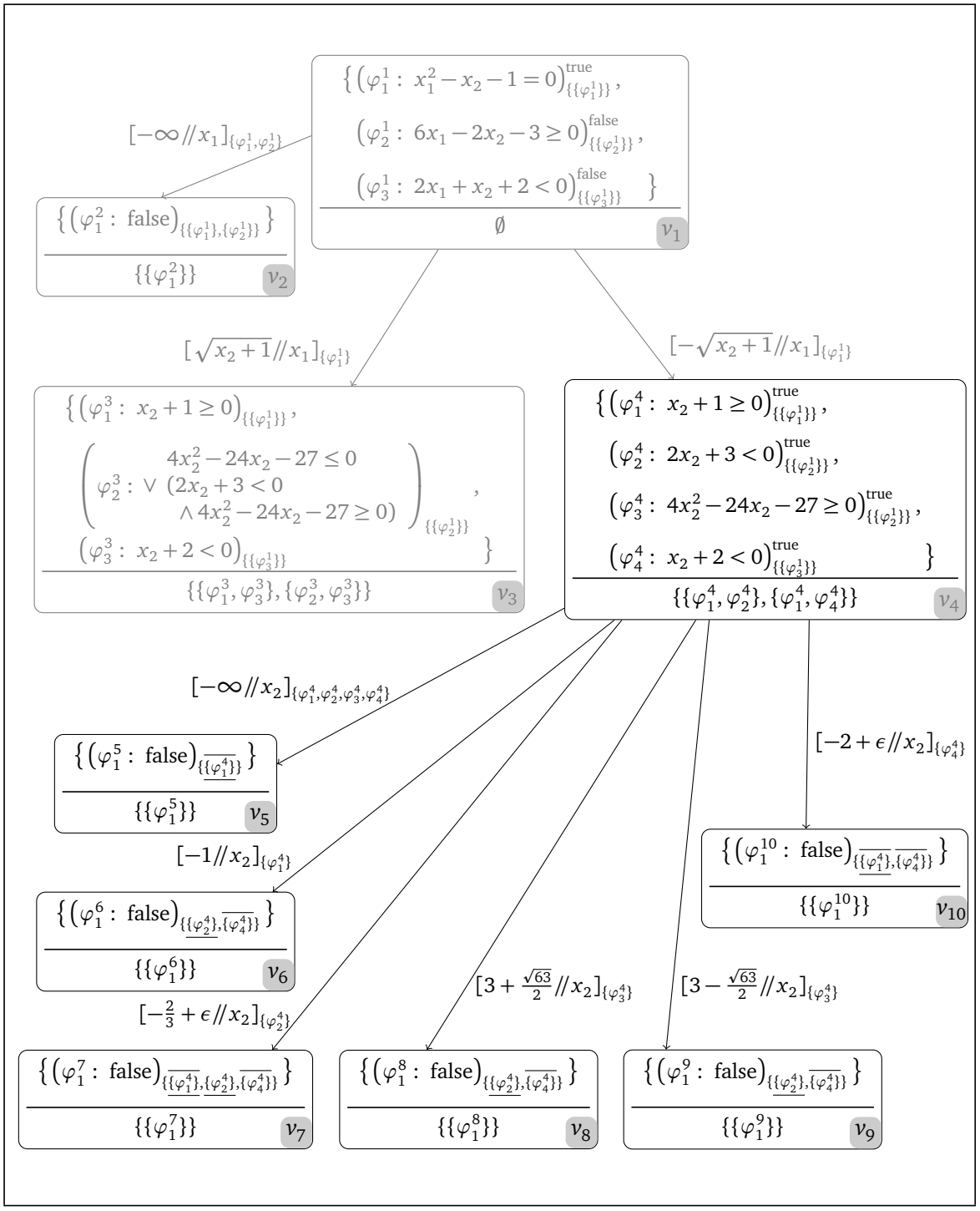


Figure 4.10: The resulting VSST T_{VS} until the *third* conflicting intermediate vertex, i. e., v_4, is reached by the satisfiability check of $x_1^2 - x_2 - 1 = 0 \wedge 6x_1 - 2x_2 - 3 \geq 0 \wedge 2x_1 + x_2 + 2 < 0$, which we invoked with $\text{check}_{\text{VS}}(T_{\text{VS}}, v_1)$ where T_{VS} and v_1 are initially as in Figure 4.9.

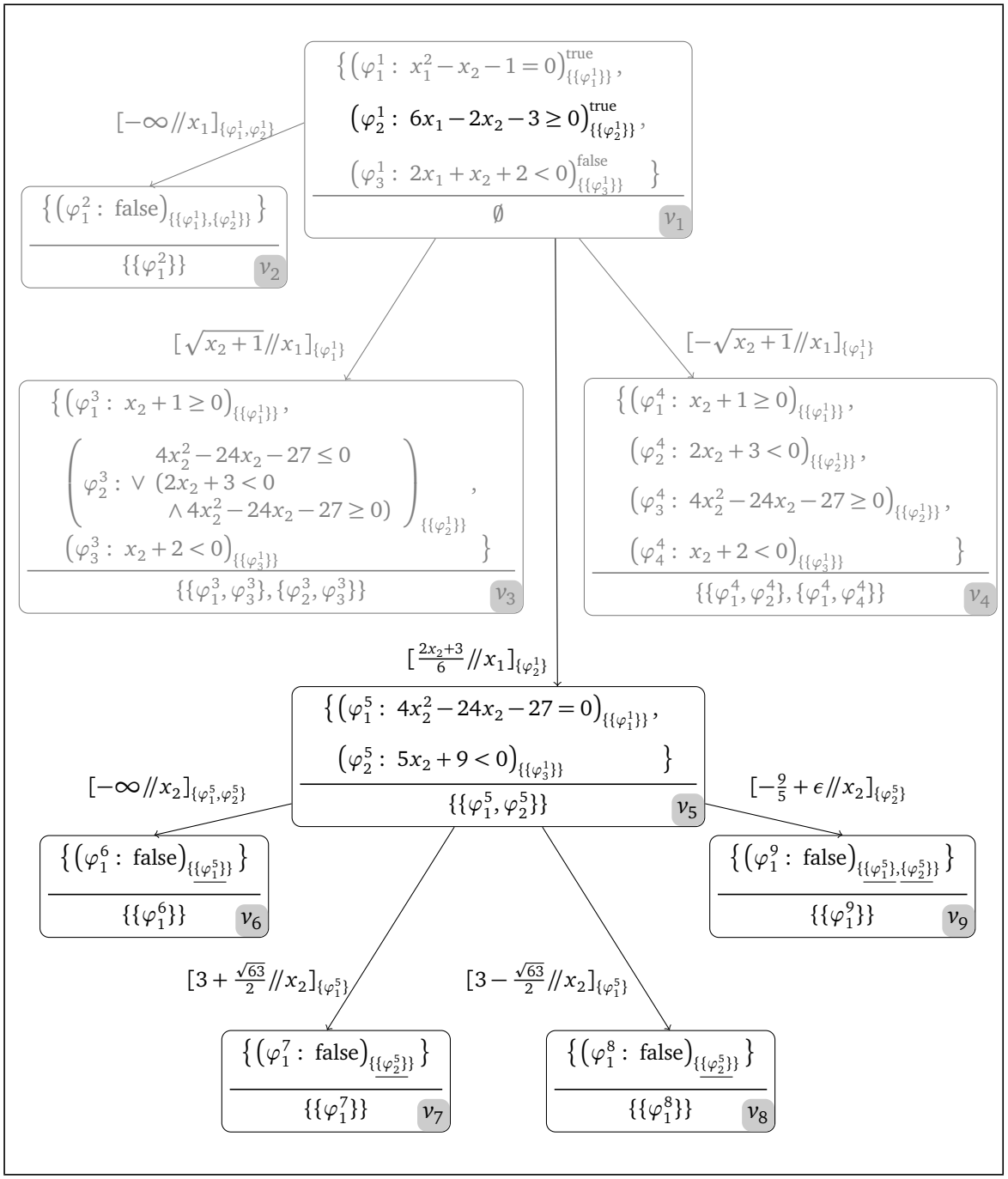


Figure 4.11: The resulting VSST T_{VS} until the *fourth* conflicting intermediate vertex, i. e., v_5, is reached by the satisfiability check of $x_1^2 - x_2 - 1 = 0 \wedge 6x_1 - 2x_2 - 3 \geq 0 \wedge 2x_1 + x_2 + 2 < 0$, which we invoked with $\text{check}_{\text{VS}}(T_{\text{VS}}, v_1)$ where T_{VS} and v_1 are initially as in Figure 4.10.

use the last remaining constraint $\varphi_3^1 : 2x_1 + x_2 + 2 < 0$ to construct test candidates. As always, we obtain the test candidate $-\infty$ and extend the corresponding child. The other test candidate, which φ_3^1 provides, is $-\frac{x_2+2}{2} + \epsilon$. We create the new child v_6 and fill $\Phi(v_6)$ with the result of

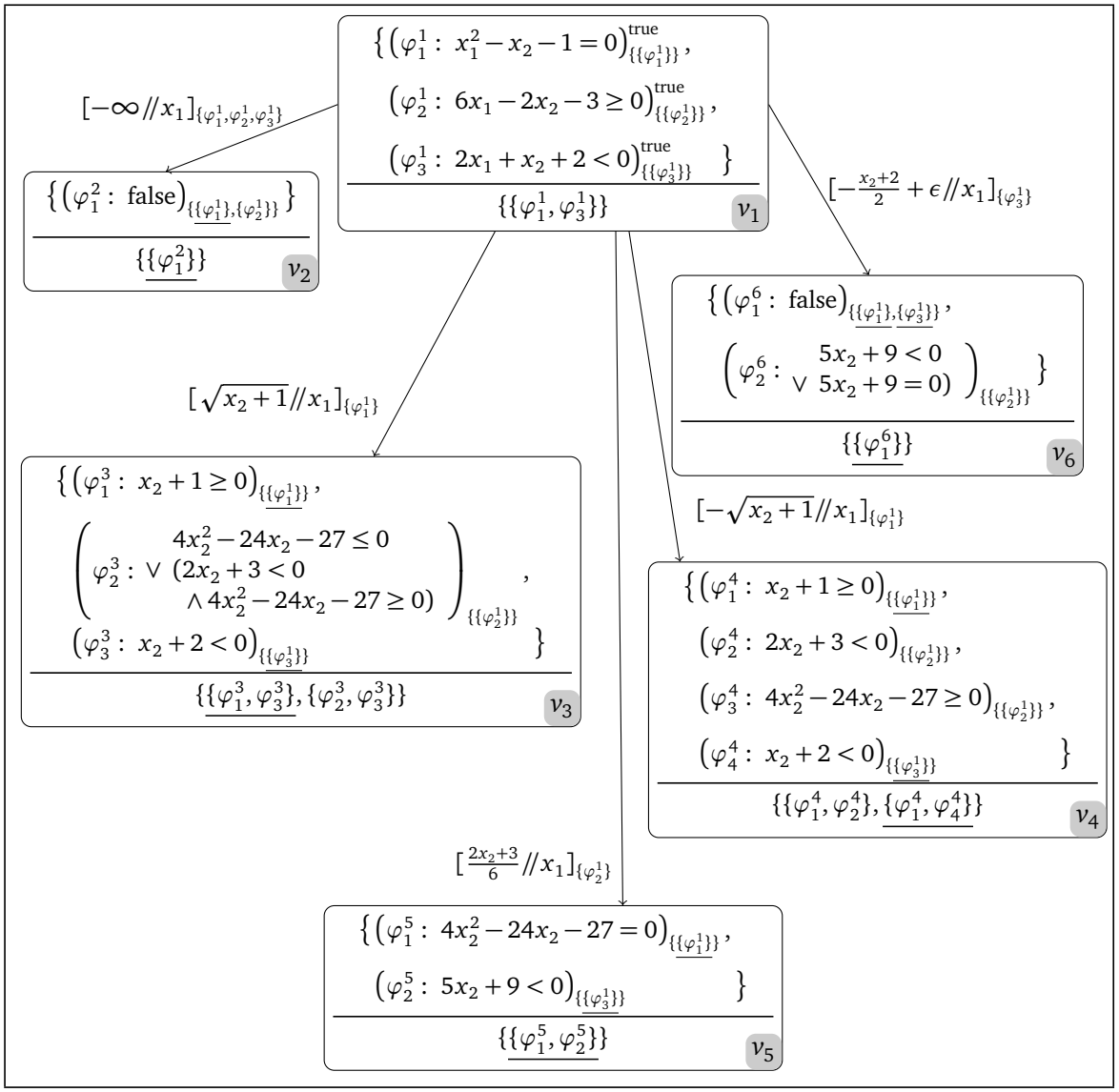


Figure 4.12: The resulting VSST T_{VS} until the *last* conflicting intermediate vertex, i. e., ν_1, is reached by the satisfiability check of $x_1^2 - x_2 - 1 = 0 \wedge 6x_1 - 2x_2 - 3 \geq 0 \wedge 2x_1 + x_2 + 2 < 0$, which we invoked with $\text{check}_{\text{VS}}(T_{\text{VS}}, \nu_1)$ where T_{VS} and ν_1 are initially as in Figure 4.11.

virtually substituting x_1 by this test candidate in all constraints of $\Phi(\nu_1)$. As for two constraints the result is false, ν_6 is conflicting. All constraints in ν_1 have now be used in order to provide test candidates and all of them are in conflict with some of the constraints in $\Phi(\nu_1)$. Therefore we apply createConflicts for the last time yielding the infeasible subset $\{x_1^2 - x_2 - 1 = 0,\ 2x_1 + x_2 + 2 < 0\}$ of the constraints whose conjunction our theory solver checked for satisfiability. As a consequence, it returns `unsat`. The resulting VSST is illustrated in Figure 4.12.

The SMT solver adds the clause $(\neg b_1 \vee \neg b_4)$ to its SAT solver and thereby excludes the conflict represented by the infeasible subset, which our theory solver has constructed. The SAT solver

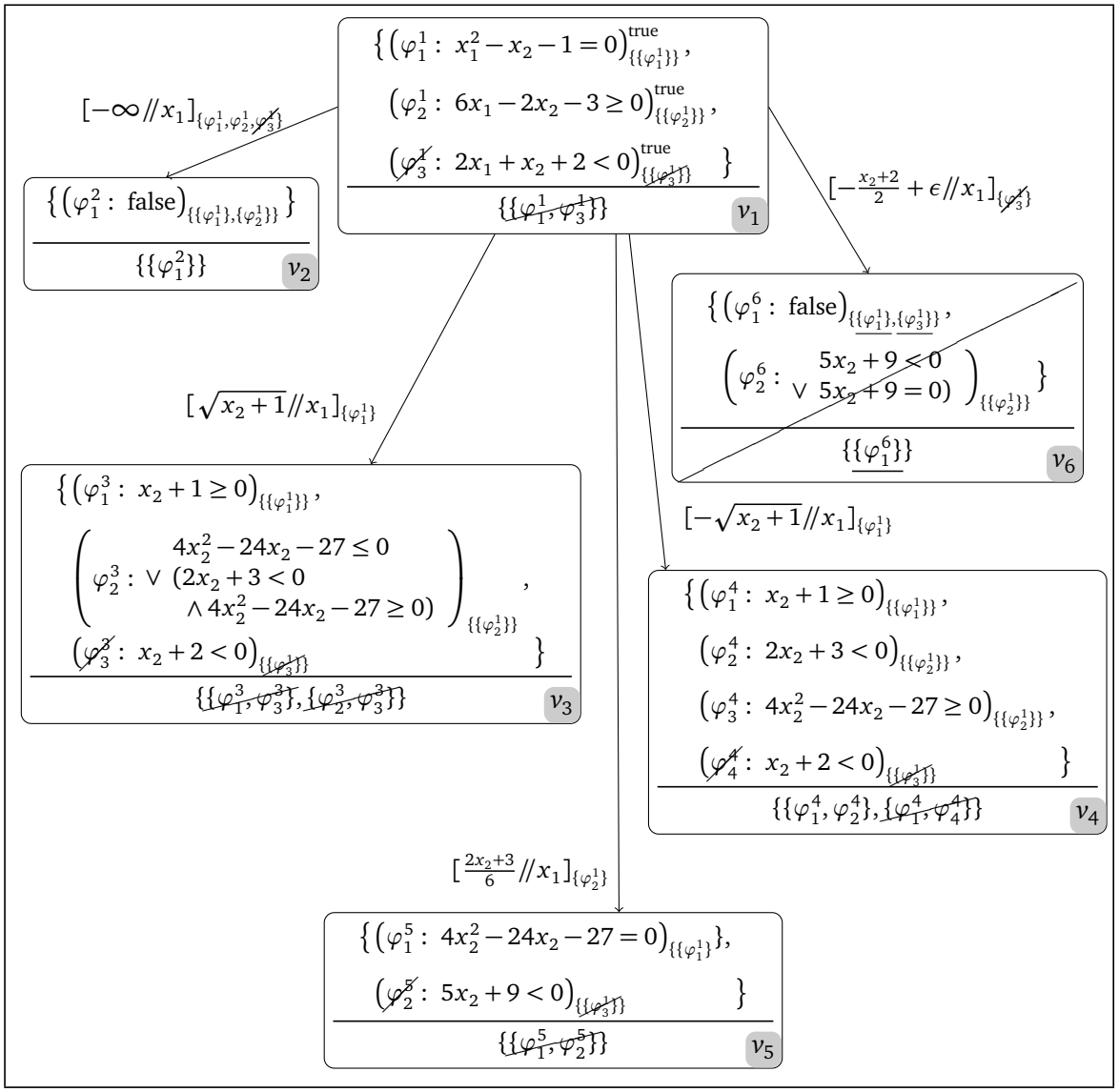


Figure 4.13: The resulting VSST after removing the constraint $2x_1 + x_2 + 2 < 0$ with $\text{remove}_{\text{VS}}(T_{\text{VS}}, \nu_1, \{2x_1 + x_2 + 2 < 0\})$, where T_{VS} and ν_1 are initally as illustrated in Figure 4.12.

backtracks to decision level 0, which leads in particular to b_4 being unassigned instead of assigned to true. Therefore we remove the corresponding constraint $2x_1 + x_2 + 2 < 0$ from our theory solver by invoking $\text{remove}_{\text{VS}}(T_{\text{VS}}, \nu_1, \{2x_1 + x_2 + 2 < 0\})$.

This is the first time in this example where Algorithm 7 is applied. The algorithm seems to be rather complicated but it is simply removing everything in our theory solver's VSST T_{VS} which can be related to $\varphi_3^1 : 2x_1 + x_2 + 2 < 0$. The resulting VSST can be seen in Figure 4.13 and in the following we explain the details on how we achieved this. We start with the origins of the formulas in $\Phi(\nu_1)$ and remove all sets in them which contain φ_3^1. If a formula has no origins left

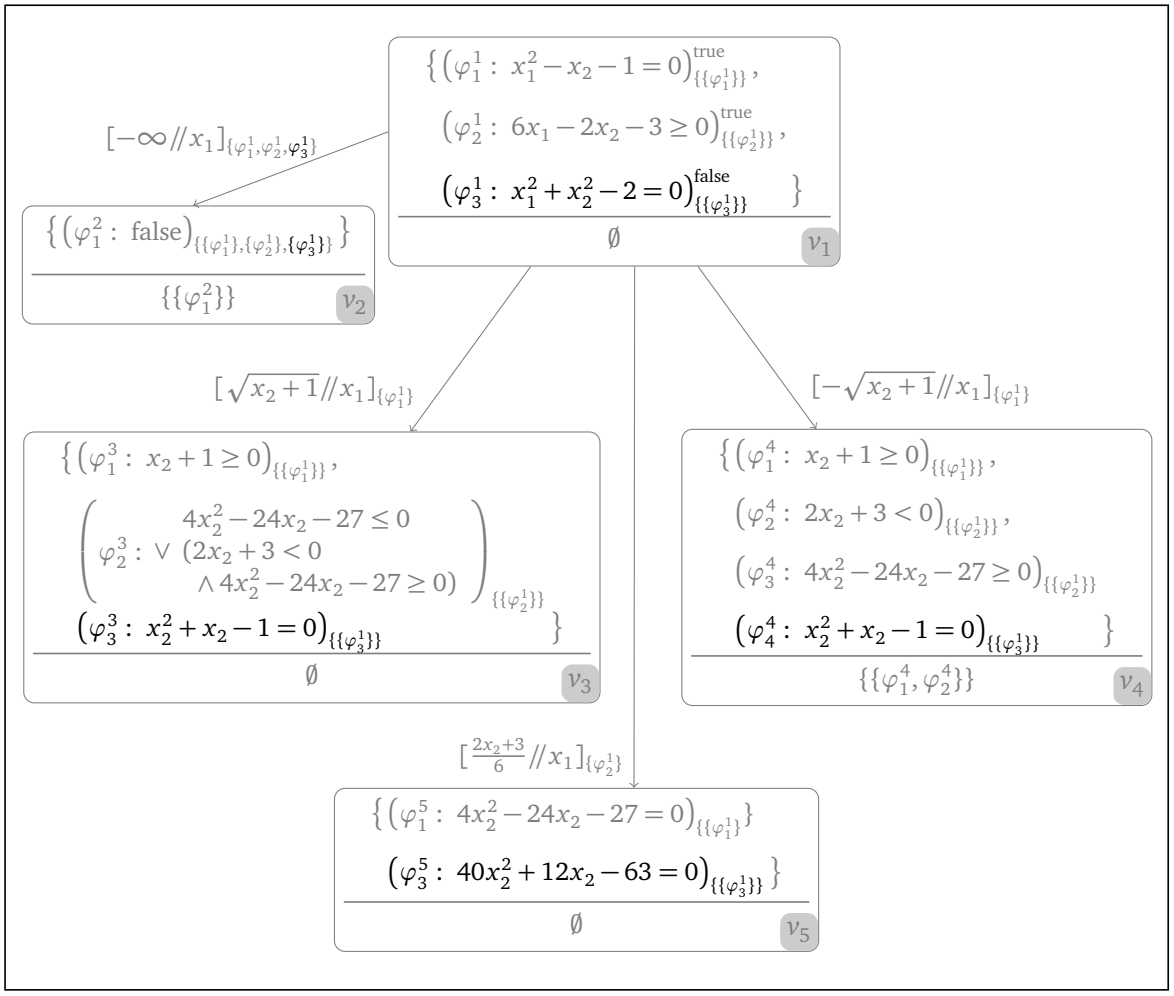


Figure 4.14: The resulting VSST after adding the constraint $x_1^2 + x_2^2 - 2 = 0$ with $\text{remove}_{\text{VS}}(T_{\text{VS}}, v_1, \{2x_1 + x_2 + 2 < 0\})$, where T_{VS} and v_1 are initially as illustrated in Figure 4.13.

afterwards, we remove this formula and its effects from T_{VS}. The only formula in $\Phi(v_1)$ containing origins with φ_3^1, is φ_3^1 itself, which is a special characteristic that only holds for the root of a VSST. As the origins of φ_3^1 are now empty we remove φ_3^1 from the remaining part of T_{VS}. We start with $\kappa(v_1)$ and remove its only infeasible subset as it contains φ_3^1 (Line 20-24).

Then, we remove φ_3^1 from the origins of the test candidates in the labels on the edges to v_1's children and afterwards remove all of its effects by recursively invoking $\text{remove}_{\text{VS}}$ for each child (Line 28-37). If a test candidate loses all its origins, we remove the subtree with the respective child as the root. In our case, we remove one of the three origins for the test candidate $-\infty$, hence, we do not remove the corresponding child v_2. Afterwards, we invoke $\text{remove}_{\text{VS}}(T_{\text{VS}}, v_2, \{\varphi_3^1\})$, which does not change anything.

The next test candidate $\sqrt{x_2 + 1}$ does not have φ_3^1 as an origin, however, invoking $\text{remove}_{\text{VS}}$ for the corresponding child v_3 empties the origins of the formula φ_3^3. Hence, we remove φ_3^3 from

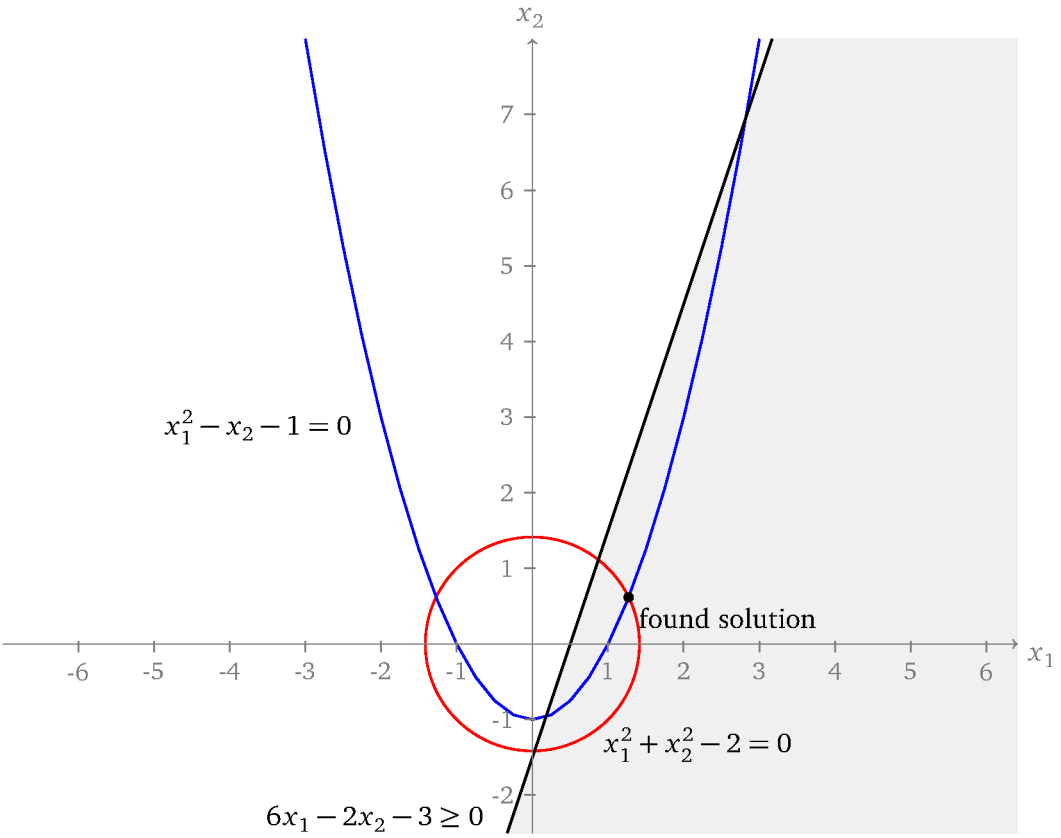


Figure 4.15: Intersection of the solution sets of the constraints $x_1^2 - x_2 - 1 = 0$, $6x_1 - 2x_2 - 3 \geq 0$ and $x_1^2 + x_2^2 - 2 = 0$ and the solution found by check_{VS}(,a) s illustrated in Figure 4.16.

$\kappa(v_3)$ yielding that no conflicts are left. We get the same result for the test candidate $\frac{2x_2+3}{6}$ and the child v_5 which represent this test candidate.

Now we consider the test candidate $-\sqrt{x_2+1}$ and its corresponding child v_4. Just as before with the test candidate $\sqrt{x_2+1}$, we remove one formula in $\Phi(v_4)$, but, fortunately, we only remove one of the two infeasible subsets from $\kappa(v_4)$. Therefore we keep the valuable information that $\Phi(v_4)$ still contains a conflict and there is no need to reconsider v_4 and the test candidate $-\sqrt{x_2+1}$.

The last test candidate $-\frac{x_2+2}{2} + \epsilon$, which has been provided by the constraints in $\Phi(v_1)$, loses its only origin with φ_3^1. Thus, we delete the vertex v_6, which corresponds to this test candidate, and conclude the call $\text{remove}_{\text{VS}}(T_{\text{VS}}, v_1, \{2x_1 + x_2 + 2 < 0\})$.

We observe that a lot of information which is stored in the VSST T_{VS} can be kept after removing the constraint, however not all results which were ever made during the previous satisfiability check are preserved. For instance, we have to reconsider the vertices v_3 and v_5, which partly makes sense as we know from our first satisfiability check which resulted in the VSST in Figure 4.4, that v_3 leads to a satisfying assignment, for instance.

In the SMT solver's SAT solver we now apply Boolean constraint propagation. It additionally

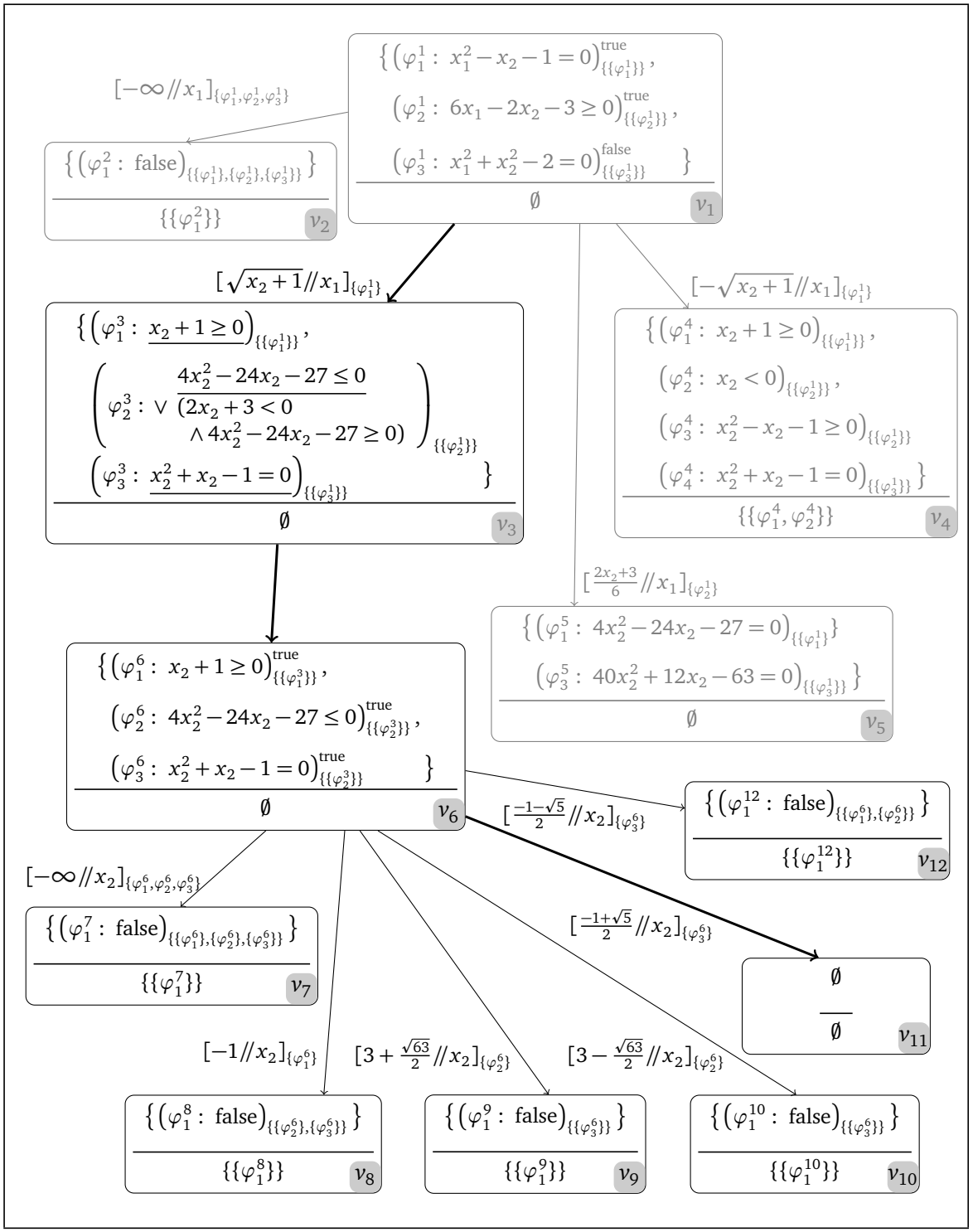


Figure 4.16: The resulting VSST T_{VS} of the satisfiability check of $x_1^2 - x_2 - 1 = 0 \wedge 6x_1 - 2x_2 - 3 \geq 0 \wedge x_1^2 + x_2^2 - 2 = 0$, which we achieved by invoking $\text{check}_{\text{VS}}(T_{\text{VS}}, \nu_1)$ with T_{VS} and ν_1 being initially as in Figure 4.14.

assigns b_4 to false and b_3 to true, therefore, we have to add the constraint $x_1^2 + x_2^2 - 2 = 0$ to the theory solver. We achieve this in the same way as in the beginning of this example with $\text{add}_{\text{VS}}(T_{\text{VS}}, v_1, x_1^2 + x_2^2 - 2 = 0, \{x_1^2 + x_2^2 - 2 = 0\})$. It results in the VSST as given by Figure 4.14

The SMT solver now has a full satisfying assignment for the Boolean skeleton of φ while excluding the conflict we have found before. Next, we want to confirm that the conjunction of the constraints which have to hold according to this assignment, i. e., $x_1^2 - x_2 - 1 = 0 \wedge 6x_1 - 2x_2 - 3 \geq 0 \wedge x_1^2 + x_2^2 - 2 = 0$, are satisfiable. If so, φ is satisfiable, otherwise, it is unsatisfiable, as we are at decision level 0. The three constraints are already added to T_{VS}, which in particular means that they are the elements of $\Phi(v_1)$. Therefore, we invoke $\text{check}_{\text{VS}}(T_{\text{VS}}, v_1)$ for the last time which results in the VSST of Figure 4.15.

Briefly worded, we achieve this as follows. We invoke check_{VS} recursively for one of the non-conflicting children of v_1, where we choose v_3. As we might remember from the first satisfiability check with check_{VS} in this example, we use v_3 for a case distinction. We create the child v_6 for the first case to consider and recursively invoke check_{VS} for this vertex. Here, we try to eliminate the remaining variable x_2 gradually constructing test candidates until one leads to a valid vertex, which is v_{11} in our case. Afterwards, the theory solver can return `sat` and so does the SMT solver. The solution of φ can again be read off the edges in of T_{VS} which are printed in bold in Figure 4.15. Therefore, the solution is $\{(x_1, \sqrt{\frac{1+\sqrt{5}}{2}}), (x_2, \frac{-1+\sqrt{5}}{2})\}$. In Figure 4.15 the solution spaces of the constraints $x_1^2 - x_2 - 1 = 0$, $6x_1 - 2x_2 - 3 \geq 0$ and $x_1^2 + x_2^2 - 2 = 0$ as well as the found solution are illustrated.

4.3 Combining virtual substitution with other procedures

Let us assume that we check the conjunction of constraints φ for satisfiability using the procedure check_{VS}. During this check we enter the sub-procedure createConflicts and fulfill the condition at Line 2 in Algorithm 11. This means that we called createConflicts(T_{VS}, v) where T_{VS} is theory solver's VSST and v a vertex in T_{VS} with $\text{incompl}(v) = \text{true}$. This can have two causes.

1. The vertex v has a child v' with $\text{incompl}(v') = \text{true}$.

2. The vertex v is used for the elimination of a variable x. All constraints in $c \in \Phi(v)$, which can be used for variable elimination, i. e., $0 < \deg(x, \text{Pol}(c)) \leq 2$, provided already test candidates, so $\text{used}(v)(c) = \text{true}$, and all of these test candidates failed to lead to a solution, which means that each of them is considered by a child v' of v with $\kappa(v') \neq \emptyset$. Moreover, there exists at least one constraint $c \in \Phi(v)$ with $\deg(x, \text{Pol}(c)) > 2$.

Algorithm 11 returns in this case `unknown`, but, instead, we can also try to solve the conjunction of formulas in $\Phi(v)$ for satisfiability with another procedure. If it returns `sat` and a solution θ', we know that φ is satisfiable and, considering Section 4.2.5, we can construct a solution θ for φ

as follows:

$$\theta(x) \quad = \quad \begin{cases} \theta'(x) & \text{, if } x \in \bigcup_{\psi \in \Phi(v)} \text{Vars}(\psi) \\ t & \text{, if } x \in \text{Vars}(\varphi) \\ & \quad \text{and } \exists i \in \{1, \ldots, k-1\}.(\text{TC}((v_i, v_{i+1})) = t \ \wedge \ \text{elimvar}(v_i) = x) \\ 0 & \text{, if } x \in \text{Vars}(\varphi). \end{cases}$$

If the procedure returns `unsat`, we need it to provide infeasible subsets of $\Phi(v)$, which we then store in $\kappa(v)$.

An example for a procedure that can be used to check the conjunction of formulas in $\Phi(v)$ for satisfiability is an implementation which is based on the cylindrical algebraic decomposition. Such an implementation would usually be complete for this purpose and, hence, always return either `sat` or `unsat`. The cylindrical algebraic decomposition, which is implemented as an `SMT-RAT` module provides infeasible subsets, as well.

Up to now, we presented the idea to invoke another procedure immediately if the aforementioned situation occurs where $\text{incompl}(v) = \text{true}$ for the call $\text{createConflicts}(T_{\text{VS}}, v)$. Instead we can also delay the use of another procedure, meaning that we keep v, mark it by "conflict creation postponed" and pursue a different branch in our VSST. At some point in the process of the satisfiability check there might only be vertices in the VSST, which are marked this way or seem to be worse to consider than invoking another procedure for some of the marked vertices.

4.4 Future work

4.4.1 Using an incremental and infeasible subset generating SAT solver for the case distinction

Algorithm 6 implements how we currently make a case distinction for a vertex v whose formulas in $\Phi(v)$ are not all constraints. It needs to find a satisfying assignment for the Boolean abstraction of the conjunction of the formulas in $\Phi(v)$ while excluding a conflict in each child of v. Clearly, a SAT solver as presented in Section 2.5 can find such a satisfying assignment or determine that the formula is unsatisfiable.

This interaction of a SAT solver and a procedure that checks the satisfiability of conjunction of constraints, where we obtain infeasible subsets of the set of these constraints, if the conjunction is unsatisfiable, strongly reminds us of SMT solving. What we actually need in order to replace the functionality of Algorithm 6, is very close to the $\texttt{Module}_{SAT}$ of `SMT-RAT`, as introduced in Section 3.1. The implementation should accept a conjunction of quantifier-free arithmetic formulas, transform it to CNF and try to find a satisfying assignment for the resulting formula's Boolean abstraction. It occasionally invokes a backend (theory solver) in order to check the satisfiability of a conjunction of constraints and in the case, that it is unsatisfiable, the implementation asks for infeasible subsets. Taking these infeasible subsets into account, the implementation searches

for another satisfying assignment for the Boolean abstraction and so on.

We furthermore require that the implementation is incremental and is able to find infeasible subsets of the set of checked quantifier-free arithmetic formulas, if their conjunction is unsatisfiable (this is not yet supported by $\texttt{Module}_{SAT}$). Then we can make use of such an implementation where we have to make a case distinction, as presented in this chapter.

4.4.2 Using SMT-RAT backends to check virtual substitution results for satisfiability

Let us consider checking a conjunction of constraints for satisfiability, as described in Section 4.2.4. We use the root of the theory solver's VSST to eliminate a variable according to Corollary 1. For each test candidate t, we create a child and add t's side conditions as well the results of virtually substituting the root's elimination variable in the constraints in the root by t to it. As the added formulas might not only be constraints but, for instance, contain a disjunction, we need to make a case distinction. If the implementation of this theory solver is part of SMT-RAT, we can directly use backends instead, as described in Chapter 3. The advantage would be that the backend could be the root of an entire SMT-RAT strategy, which not only deals with the Boolean complexity by a $\texttt{Module}_{SAT}$ but also applies preprocessing or other modules based on, for instance, the Simplex method or Gröbner bases. As mentioned before, for a well-performing embedding, we would require that all of these implementations are incremental and can generate infeasible subsets. Unfortunately, this is not yet supported by $\texttt{Module}_{SAT}$.

CHAPTER 5

Improving the Performance of the Virtual Substitution in SMT

This chapter addresses optimizations of the satisfiability check as described in Chapter 4. The presented ideas can mostly be adopted to be usable in the context of the original virtual substitution as introduced in Section 2.7.

On the one hand, we present the heuristics we make use of for the situations where we have a choice. For instance, in Section 5.1 we describe how we select the variable to eliminate according to Corollary 1 or the next constraint for the provision of test candidates. The construction of conflicts as presented in Section 4.2.6 involves an approximation of the optimal set coverings. We discuss the optimality criteria and heuristic choices we can make during this construction in Section 5.2.

On the other hand, we present optimizations which exploit local information during the search for a satisfying assignment as specified in Section 4.2.4. In Section 5.2, we present a mechanism which allows us to prune larger unsatisfiable subtrees in the VSST of the theory solver during its consistency check. Furthermore, we explain in Section 5.3 how we can detect that a vertex in the VSST is conflicting before all of its children are constructed. Finally, we present an optimization in Section 5.4 which takes advantage of variable bounds, that is upper and lower bounds on the variables' domains. As a result we can narrow down the set of test candidates we have to consider and simplify the virtual substitution results.

5.1 Choice of the elimination variable and constraint to provide test candidates for

Let us assume a vertex v of the VSST $T_{\mathrm{VS}} = (V, E, \Phi, \mathrm{orig}, \kappa, \mathrm{elimvar}, \mathrm{used}, \mathrm{incompl}, \mathrm{TC})$ such that $\Phi(v)$ contains only constraints. In this case, we use v for the elimination of a variable according

to Corollary 1 and, therefore, need to specify a variable to eliminate. Moreover, we gradually construct the test candidates of the elimination variable for the constraints in $\Phi(v)$, where we consider them one at a time.

Firstly, we need a measure to decide how good a constraint serves for the provision of test candidates for a given elimination variable. Based on this measure, we can decide how well a variable is suitable as an elimination variable for a set of constraints, as given in $\Phi(v)$.

5.1.1 Measure of quality of constraints for test candidate construction

Given a variable x and a constraint $c \in \Phi(v)$ such that $x \in \text{Vars}(c)$, the following functions, we refer to as *constraint valuations*, specify a value in order to represent different properties of x in c. This value is positive and the closer it is to 0, the more preferable it is with respect to the property it concerns.

Too high degree $\omega_{\text{thd}}(x, c)$: This value is either 1, if the degree of x is less than or equal to 2, or 2, otherwise. That way we prefer constraints for which it is possible to create test candidates for x, that is the degree in x is not too high.

It is clearly very important to give constraints a worse (and higher) rating if x occurs with a degree higher than 2, as this is the case where the virtual substitution might not give a conclusive answer. However, we have to keep in mind that even though we prefer to choose x, if it occurs in all constraints only with a degree less than or equal to 2, the remaining variables might not satisfy this property, especially after the elimination of x.

Finitely many solutions $\omega_{\text{fms}}(x, c)$: We rate c better (with 1 instead of 2), if c is an equation, the degree of x in c is less than or equal to 2 and for at least one finite test candidate $t \in \text{tcs}(x, c)$, i. e., $t \neq -\infty$, it holds that $\text{sc}(t) = \text{true}$. According to Theorem 5, we then have to consider only the test candidates for x in c instead of all test candidates for x in the constraints in $\Phi(v)$.

This is a very important valuation, as it drastically reduces the number of test candidates to consider for x. In particular, there is no need to create test candidates for x in constraints, where x occurs with a degree higher than 2.

Relation symbol $\omega_{\text{rel}}(x, c)$: We rate equations best (with 1) as they restrict the solution space most and, therefore, mutual solutions of the constraints in $\Phi(v)$ are rather represented by the test candidates for x in equations. We rate strict inequalities worse (with 3 instead of 2) than weak inequalities, as the virtual substitution by test candidates for x in strict inequalities, i. e., those where we add an infinitesimal ϵ, entail a higher complexity than the virtual substitution by test candidates for x in weak inequalities.

Elimination variable's degree $\omega_{\text{evd}}(x, c)$: This value equals to the degree of x in c, that is the lower this degree is, the better we rate c.

The virtual substitution of x by test candidates, which we construct for constraints being linear in x, result in simpler formulas compared to those we get for constraints where x occurs quadratic.

Number of variables $\omega_{\text{nv}}(x,c)$: This value equals the number of variables in c, i. e., $|\text{Vars}(c)|$. The more variables c has, the worse is its rating. Test candidates for x in c with a small ω_{nv}, prevents more complex virtual substitution results in terms of the number of variables in each constraint in the result.

We combine constraint valuations for a given variable x and a constraint $c \in \Phi(v)$ such that $x \in \text{Vars}(c)$ by

$$\omega((\omega_1,..,\omega_n),x,c) \;=\; \sum_{i=1}^{n} \frac{\overline{\omega}_i(x,c)}{(\max_\omega)^i}$$

where $\omega_1,..,\omega_n \in \{\omega_{\text{thd}},\omega_{\text{fms}},\omega_{\text{rel}},\omega_{\text{evd}},\omega_{\text{nv}}\}$ are pairwise different, $\max_\omega \in \mathbb{N}$ and for all $i \in \{1,..,n\}$ it holds that

$$\overline{\omega}_i(x,c) = \begin{cases} \omega_i(x,c) & \text{, if } \omega_i(x,c) < \max_\omega \\ \max_\omega & \text{, otherwise.} \end{cases}$$

This yields a rating of a constraint with respect to one of its variables, where we prioritize the constraint valuation ω_1 over $\omega_2,..,\omega_n$, the constraint valuation ω_2 over $\omega_3,..,\omega_n$ and so on.

Theorem 5 *Let $\varphi = c_1 \wedge .. \wedge c_n$ be a conjunction of constraints and $c \in \{c_1,..,c_n\}$ be an equation, such that*

- $x \in \text{Vars}(c)$*,*
- $\deg(x,\text{Pol}(c)) \leq 2$ *and*
- *there exists a test candidate $t \in \text{tcs}(x,c)$ with $t \neq -\infty$ and $\text{sc}(t) = \text{true}$.*

Then it holds that

$$\varphi \quad \textit{is satisfiable} \quad \Leftrightarrow \quad \bigvee_{\substack{t \in \text{tcs}(x,c) \\ t \neq -\infty}} (\varphi[t\,//\,x] \wedge \text{sc}(t)) \quad \textit{is satisfiable.}$$

Proof 5 Let $\varphi = c_1 \wedge .. \wedge c_n$ and c, x and t as defined by Theorem 5.

Then c is of the form $p_1x^2 + p_2x + p_3 = 0$ and the virtual substitution rules in [Wei97] specify that

$$(p_1x^2 + p_2x + p_3 = 0)[-\infty\,//\,x] \quad \equiv \quad p_1 = 0 \wedge p_2 = 0 \wedge p_3 = 0.$$

From $\text{sc}(t) = \text{true}$ it follows from Definition 21 that $p_1 \neq 0$ or $p_2 \neq 0$ and, therefore, $c[-\infty\,//\,x] \equiv$

false. Then, it holds that if $\alpha \in \Theta(c)$ we know that

$$\alpha(x) \in M = \{\frac{[\![p]\!]^\alpha + [\![q]\!]^\alpha \sqrt{[\![r]\!]^\alpha}}{[\![s]\!]^\alpha} \mid \frac{p + q\sqrt{r}}{s} \in \text{tcs}(x,c) \setminus \{-\infty\},\ [\![\text{sc}(t)]\!]^\alpha \equiv \text{true}\}.$$

As $\Theta(\varphi) = \bigcap_{i=1}^{n} \Theta(c_i)$, it also holds that if $\alpha \in \Theta(\varphi)$ then $\alpha(x) \in M$. Hence, it is sufficient to only check the test candidates in $\text{tcs}(x,c)$ in order to determine the satisfiability of φ. □

We still need to decide which constraint valuations we want to use and how we prioritize them. As explained before, we expect the constraint valuation ω_{fms} to be the most important one, followed by ω_{thd} as it might enable us to solve formulas where we would otherwise not find a conclusive answer. Therefore, we always use these constraint evaluations with the highest priority in this order. It will be interesting to see, which of the constraint valuations ω_{rel} and ω_{evd} are more important, hence, we will test them, once prioritizing ω_{rel} and once prioritizing ω_{evd}. The last constraint evaluation ω_{nv} seems to be the least important and we will test, whether it is better to include or exclude it. Summarized, we are going to test the following four combinations of constraint valuations in Section 6.4.

1. $(\omega_{\text{fms}}, \omega_{\text{thd}}, \omega_{\text{rel}}, \omega_{\text{evd}}, \omega_{\text{nv}})$
2. $(\omega_{\text{fms}}, \omega_{\text{thd}}, \omega_{\text{rel}}, \omega_{\text{evd}})$
3. $(\omega_{\text{fms}}, \omega_{\text{thd}}, \omega_{\text{evd}}, \omega_{\text{rel}}, \omega_{\text{nv}})$
4. $(\omega_{\text{fms}}, \omega_{\text{thd}}, \omega_{\text{evd}}, \omega_{\text{rel}})$

5.1.2 Measure of quality of variables for elimination

Given the set of constraints $\Phi(v)$ in the vertex v, we want to identify a variable occurring in these constraints which we eliminate next according to Corollary 1. This choice is vital for the performance of the virtual substitution and can even decide whether we can determine the satisfiability of a given formula or not. Therefore, we always try to choose the variable for which the constraints in $\Phi(v)$ provide the most promising test candidates with respect to the heuristic we choose according to Section 5.1.1.

Let us assume that we decide to use the constraint valuations $\omega_1, .., \omega_n$ in this order, that is ω_1 has the highest priority and ω_n the lowest. Given a variable x, we can then valuate its quality for an elimination according to Corollary 1, if we consider the *variable valuation*

$$(\overbrace{\omega((\omega_1, .., \omega_n), x, c_1)}^{\omega_{x,1}}, .., \overbrace{\omega((\omega_1, .., \omega_n), x, c_k)}^{\omega_{x,k}})$$

where $c_1, .., c_k \in \Phi(v)$ are the constraints which contain the variable x and for all $1 \leq i < k$ it holds that $\omega_{x,i} \leq \omega_{x,i+1}$. A variable valuation represents the constraint valuations for the variable x in $\Phi(v)$ written as an increasing number series.

Based on variable valuations we specify an order for the variables and then choose one of the variables which are the smallest in this order for an elimination according to Corollary 1. In the following, we present three different orders, which try to optimize the best, the average and the worst constraint valuation for a variable, respectively. For this purpose, we assume we are comparing two variables x_1 and x_2 and consider their variable valuations $(\omega_{x_1,1},..,\omega_{x_1,k_1})$ and $(\omega_{x_2,1},..,\omega_{x_2,k_2})$.

Optimizing best constraint valuation:

$$\begin{aligned} x_1 \prec_{\text{best}} x_2 \quad \Leftrightarrow \quad & \exists j \in \{0,..,\min(\{k_1,k_2\})\}. \\ & (\forall i \in \{1,..,j\}.\omega_{x_1,i} = \omega_{x_2,i}) \\ & \wedge ((j = k_1 \wedge k_2 > j) \vee (j < \min(\{k_1,k_2\}) \wedge \omega_{x_1,j+1} < \omega_{x_2,j+1})) \end{aligned}$$

Optimizing average constraint valuation:

$$x_1 \prec_{\text{avg.}} x_2 \quad \Leftrightarrow \quad (\sum_{i=0}^{k_1} \omega_{x_1,i})/k_1 \; < \; (\sum_{i=0}^{k_2} \omega_{x_2,i})/k_2$$

Optimizing worst constraint valuation:

$$\begin{aligned} x_1 \prec_{\text{worst}} x_2 \quad \Leftrightarrow \quad & \exists j \in \{0,..,\min(\{k_1,k_2\})\}. \\ & (\forall i \in \{0,..,j-1\}.\omega_{x_1,k_1-i} = \omega_{x_2,k_1-i}) \\ & \wedge ((j = k_1 \wedge k_2 > j) \vee (j < \min(\{k_1,k_2\}) \wedge \omega_{x_1,k_1-j} < \omega_{x_2,k_2-j})) \end{aligned}$$

5.2 Conflict construction and backjumping

For the generation of small reasons for infeasibility, as explained in Section 4.2.6, we considered a vertex v in the theory solver's VSST $T_{\text{VS}} = (V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC})$, such that all of its children are constructed and all of them are conflicting. Let us assume that $V' \subset V$ is the set of v's children. As mentioned before, a vertex v' is conflicting if $\kappa(v') \neq \emptyset$. Our goal is to construct infeasible subsets of the set of formulas $\Phi(v)$ by the use of the conflicts $\kappa(v')$ of v's children. Note that we checked $\bigwedge_{\varphi \in \Phi(v)} \varphi$ for satisfiability with the result that it is unsatisfiable. As we know from Equation (4.1), we obtain an infeasible subset $K \subseteq \Phi(v)$ if it covers a conflict $K_{v'} \in \kappa(v')$ in each child $v' \in V'$ after mapping the formulas $\varphi' \in K_{v'}$ to one of its origins $N \in \text{orig}(v')(\varphi')$, i. e.,

$$\forall v' \in V'.\, \exists M \in \overbrace{\{\bigcup_{i=1}^{n} N_i |\, (N_1,..,N_n) \in \prod_{i=1}^{n} \text{orig}(v')(\varphi'_i),\ \{\varphi'_1,..,\varphi'_n\} \in \kappa(v')\}}^{O_\kappa(v')}.\, M \subseteq K.$$

We refer to $O_\kappa(v')$ as the *origins of the conflicts in the vertex* v'.

This means that we have to find a set covering of $\{O_\kappa(v') |\, v' \in V'\} \subseteq \mathbb{P}(\mathbb{P}(\mathbb{P}(\Phi(v))))$, that is a

set $K \subseteq \Phi(v)$ which covers an element M of each $O_\kappa(v')$, i. e., $M \subseteq K$. It differs slightly from the original definition of the set covering problem [CTF00], where we would require that $M \in \Phi(v)$ instead of $M \subseteq \Phi(v)$ and that $M \in K$ instead of $M \subseteq K$.

Due to the aforementioned construction of $O_\kappa(v')$ for a $v' \in V'$, in the worst case it contains $|\kappa(v')| \cdot |N_1| \cdot \ldots \cdot |N_n|$ sets, which is a number that grows exponentially as n increases. However, it is seldom the case that we have more than one origin for a formula considered by a vertex in V, i. e., $\frac{\sum_{i=1}^{n} |N_i|}{n} \approx 1$. Hence, the size of $O_\kappa(v')$ grows linearly as the number of conflicts in $\kappa(v')$ increases.

Finding the smallest set covering is an NP-hard problem [CTF00]. Therefore, we only construct an over-approximation of the smallest set covering invoking setCovering($\{O_\kappa(v')|\ v' \in V'\}$), which is defined in Algorithm 12. For each $O' \in \{O_\kappa(v')|\ v' \in V'\}$, which is a set of sets, this algorithm adds the elements of the set $M \in O'$ to the set covering K, which is already covered the most by K, i. e., $|M \cap K|$ is minimal. This algorithm is correct, as we clearly cover a set in each $O' \in \{O_\kappa(v')|\ v' \in V'\}$. As we need to check each set in O' once, in order to find the one which is covered by K the most, the complexity of Algorithm 12 lies in $\mathcal{O}(n)$ where n is the number of subsets of $\Phi(v)$ in $\{O_\kappa(v')|\ v' \in V'\}$.

Algorithm 12 Given a set O of sets of subsets of a set of formulas Φ, this procedure returns a set $K \subseteq \Phi$ such that for all $O' \in O$ there exists a $M \in O'$ such that $M \subseteq K$.

setCovering(a set $O \subseteq \mathbb{P}(\mathbb{P}(\mathbb{P}(\Phi)))$ of sets of subsets of a set of formulas Φ)
begin
1: $K := \emptyset$ *// initialize set covering*
2: **for all** $O' \in O$ **do**
3: $M :=$ set in O' with $|M \cap K|$ being minimal
4: $K := K \cup M$ *// update yet found set covering*
5: **end for**
6: **return** K
end

5.2.1 Backjumping

Assume that we have a vertex v of our VSST T_{VS} with $\kappa(v) \neq \emptyset$ being constructed as explained in the beginning of this section. If we use v for a case distinction and, hence, v's father v_f for test candidate generation, we can check whether a conflict $K \in \kappa(v)$ exists which consists only of constraints that were already considered by v_f, i. e., $K \subseteq \Phi(v_f)$. As K is an infeasible set of constraints, $\bigwedge_{\varphi \in \Phi(v_f)} \varphi$ must be unsatisfiable, therefore we can add K to $\kappa(v_f)$ and do not need to consider any further test candidates for the variable elimvar(v_f), which we eliminate in the constraints of $\Phi(v_f)$. Instead we directly jump back to the father of v_f, if it is not the root, or detect the unsatisfiability of the conjunction of the constraints in the root of T_{VS}, otherwise. This is why we name this technique *backjumping*.

With the aforementioned simple and cheap check we can clearly omit checking test candidates, which speeds-up the search for a solution. We can adapt the construction of conflicts as given by Algorithm 12, such that it is more likely to encounter a case where we can apply backjumping. For this purpose we make use of the following definition in order to specify the quality of a formula during this construction.

Definition 26 (Age of a formula in a VSST) *Given a VSST* $T_{\mathrm{VS}} = (V, E, \Phi, \mathrm{orig}, \kappa, \mathrm{elimvar}, \mathrm{used}, \mathrm{incompl}, \mathrm{TC})$ *and a vertex* $v \in V$*, we define the* age of a formula $\varphi \in \Phi(v)$ *(with respect to the vertex* v*) by*

$$\mathrm{age}_{T_{\mathrm{VS}}} \colon \mathrm{FO}(\tau) \times V \to \mathbb{N}_0 : (\varphi, v) \mapsto \begin{cases} 1 & \text{, if } \varphi \in \Phi(v) \text{ and } v \text{ is root,} \\ 1 + \mathrm{age}_{T_{\mathrm{VS}}}(\varphi, v_f) & \text{, if } \varphi \in \Phi(v) \text{ and } v_f \text{ is father of } v, \\ 0 & \text{, otherwise.} \end{cases}$$

The age of a formula φ with respect to the vertex v in the VSST T_{VS} is the length of the path from the vertex v', which lies on the path from the root of T_{VS} to v and does not contain φ, i. e., $\varphi \notin \Phi(v')$, to v. In other words, this age tells us how long a formula has been passed from a node to its children and to their children and so on until it got to v. In particular it means, that the variable elimination along this path did not change φ, which is simply based on the fact that none of the variables in $\mathrm{Vars}(\varphi)$ have been eliminated yet.

As before, let us consider the case where we have a vertex v in the theory solver's VSST $T_{\mathrm{VS}} = (V, E, \Phi, \mathrm{orig}, \kappa, \mathrm{elimvar}, \mathrm{used}, \mathrm{incompl}, \mathrm{TC})$, such that all of its children are constructed and all of them are conflicting. So far we added the conflicts to $\kappa(v)$, which we create with $\mathrm{setCovering}(O)$, where $O = \{O_\kappa(v') |\ v' \in V'\}$. Now, we preprocess O before, which results in

$$O_{\mathrm{age}_{T_{\mathrm{VS}}}} = \{\ \{M \in O_\kappa(v') |\ \max(\{\mathrm{age}_{T_{\mathrm{VS}}}(\varphi, v) |\ \varphi \in M\}) \leq m\} \mid v' \in V'\}$$

with

$$m = \max(\{\min(\{\max(\{\mathrm{age}_{T_{\mathrm{VS}}}(\varphi, v) |\ \varphi \in M\}) |\ M \in O_\kappa(v')\}) |\ v' \in V'\}),$$

and then add the conflicts created by $\mathrm{setCovering}(O_{\mathrm{age}_{T_{\mathrm{VS}}}})$ to $\kappa(v)$. Here, m is the minimum of all possible set covering's formulas' maximum age. The preprocessing removes all sets of formulas in O, where the formulas' maximum age is higher than m. Note that by the construction of m it is ensured that each set of sets in $O_{\mathrm{age}_{T_{\mathrm{VS}}}}$ is not empty. Consequently, invoking $\mathrm{setCovering}(O_{\mathrm{age}_{T_{\mathrm{VS}}}})$ yields an over-approximation of the smallest set covering where its formulas' maximum age is minimal with respect to all possible set coverings of O.

5.3 Local conflict detection

The generation of small infeasible subsets, as one of the requirements on an SMT compliant theory solver, might form their most important feature. Usually we have to compute the infeasible subsets after the theory solver's check procedure detected the unsatisfiability of the conjunction of its

input constraints. In our case, where we used check_{VS} on the theory solver's VSST $T_{\text{VS}} = (V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC})$, the infeasible subsets are already stored in the conflicts $\kappa(v_r)$ of T_{VS}'s root $v_r \in V$, if check_{VS} returns `unsat`. This is due to the fact that we calculate the infeasible subsets as we perform the consistency check. Furthermore, we do not only calculate them for the root but all conflicting vertices of T_{VS}, which is necessary in order to finally construct the conflicts of the root.

Regarding the performance of a satisfiability check with check_{VS} we have already seen a scenario where smaller conflicts in conflicting vertices might be useful for other purposes. With smaller conflicts it is more likely that the maximum age of the formulas in a conflict is smaller, and, hence, it is more likely that we can apply backjumping, as introduced in Section 5.2. As we explained, backjumping saves us from considering all test candidates, which the constraints in a vertex for variable elimination provide, and, therefore, it improves the performance of the satisfiability check.

We would like to make more use of the thorough construction of conflicts in the vertices. Let us assume that during the satisfiability check with check_{VS} we construct a conflict in a vertex $v \in V$ using the procedure createConflicts(T_{VS}, v). Then, $\kappa(v)$ stores a set of conflicts which are infeasible subsets of the formulas in $\Phi(v)$. In the case that we use v for variable elimination, v is conflicting and, hence, we construct its conflicts, if all children of v are constructed and all of them are conflicting. This means that all constraints in $\Phi(v)$ have already provided test candidates and each of them led to a conflict. Now, if $\kappa(v)$ contains an infeasible subset M of $\Phi(v)$ with $M \neq \Phi(v)$, which means that $M \subset \Phi(v)$, it follows from Corollary 1 that it would have been sufficient to only consider the test candidates provided by the constraints in M in order to detect that their conjunction is unsatisfiable. As introduced by Definition 27, we say that the constraints in M form a local conflict. This definition generalizes the described situation by allowing a conjunction of arbitrary formulas instead of a conjunction of constraints. A local conflict is actually just a synonym for an infeasible subset but in context of the elimination of a variable with the virtual substitution.

Definition 27 (Local conflict) *Let $\varphi^{\mathbb{R}} = \varphi_1 \wedge .. \wedge \varphi_n$ be a quantifier-free real-arithmetic formula. Furthermore, let $M \subset \{\varphi_1, .., \varphi_n\}$ be a strict subset of the sub-formulas of $\varphi^{\mathbb{R}}$ and $\varphi_M^{\mathbb{R}} = \bigwedge_{\varphi \in M} \varphi$ with $x \in \text{Vars}(\varphi_M^{\mathbb{R}})$ occurs at most quadratically in $\varphi_M^{\mathbb{R}}$. Then the formulas in M form a* local conflict *if*

$$\bigvee_{t \in \text{tcs}(x, \varphi_M^{\mathbb{R}})} (\varphi_M^{\mathbb{R}}[t /\!/ x] \wedge \text{sc}(t)) \text{ is unsatisfiable.}$$

With the help of the conflicts in v's children and using a similar principle as for the construction of conflicts with the procedure createConflicts, we can detect a local conflict before constructing and checking all test candidates provided by the constraints in $\Phi(v)$. For this purpose we use the procedure which is implemented by Algorithm 13, i. e., we invoke localConflict(T_{VS}, v), before we create new children in the procedure check_{VS} as given by Algorithm 10 (Line 8).

Algorithm 13 Given a VSST $T_{\text{VS}} = (V, E, \Phi, \text{orig}, \kappa, \text{elimvar}, \text{used}, \text{incompl}, \text{TC})$ and a vertex $v \in V$, this procedure returns true if we have a local conflict for v.

```
localConflict(VSST T_VS, vertex v ∈ V)
begin
 1:   if elimvar(v) = ⊥ then return false        // only if vertex is used for variable elimination
 2:   // collect all constraints, which have been used to create test candidates
 3:   M := {c ∈ Φ(v)| used(v)(c) = true}
 4:   for all (v, v') ∈ E do
 5:     (t, O_t) := TC((v, v'))
 6:     if O_t ∩ M ≠ ∅ then                       // test candidate has origin in M
 7:       // if v' is not conflicting or no conflict mapped back to its origins
 8:       // is covered by M, then we do not have a local conflict for M
 9:       if κ(v') = ∅ or ∀K ∈ O_κ(v'). K ⊈ M then return false
10:     end if
11:   end for
12:   // in each child a conflict after mapping it back to its origins is covered by M,
13:   // then M is a local conflict and we add it to κ(v)
14:   κ(v) := κ(v) ∪ {M}
15:   return true
end
```

Algorithm 13 only detects local conflicts for vertices which are used for variable elimination and returns false (Line 1), otherwise. It then collects all constraints in $\Phi(v)$, which have been used for the provision of test candidates (Line 3), and checks whether all children are conflicting and whether these constraints cover one of the origins of the conflicts in each child (Line 14). If this is the case, we add the constraints as a conflict to $\kappa(v)$ (Line 14) and return true, which means that a local conflict has been found.

5.4 Exploiting variable bounds

In many real world examples, we deal with variables that have upper or lower bounds. For instance, variables which represent time, temperature, or velocity come naturally with bounds. Time must be non-negative and, depending on the example, we can usually predict certain bounds for the temperature or velocity (room temperature < 50 degrees, $0 \leq$ car velocity < 400 km/h etc.). We can represent these bounds with constraints as defined in Definition 28.

Definition 28 (Variable Bound Constraint) *A constraint of the form*

$$ax - d \sim 0$$

with x being an arithmetic variable, $a \in \mathbb{N}$, $d \in \mathbb{Z}$ and $\sim \in \{=, \leq, <, \geq, >\}$ is called a (variable) bound constraint.

We denote the set of all bound constraints *by* $\mathrm{CS}_{\mathrm{VB}} \subset \mathrm{CS}$ *and we refer to a bound constraint* $c \in \mathrm{CS}_{\mathrm{VB}}$ *as a* lower bound constraint, *if* $\mathrm{rel}(c) \in \{=, \geq, >\}$, *and as an* upper bound constraint, *if* $\mathrm{rel}(c) \in \{=, \leq, <\}$. *Note that a bound constraint with the relation* $=$ *forms a lower as well as an upper bound constraint.*

In the following definition, we generalize how we obtain and represent the variables' bounds for a given formula.

Definition 29 (Variable bounds) *Let* φ *be an arbitrary arithmetic formula. Then we can, w. l. o. g., decompose* φ *to*

$$c_1 \wedge .. \wedge c_n \wedge \varphi_1 \wedge .. \wedge \varphi_m,$$

where $n, m \in \mathbb{N}_0$, $n + m > 0$, $c_1, .., c_n$ *are bound constraints and* $\varphi_1 \wedge .. \wedge \varphi_m$ *are arithmetic formulas which are not bound constraints.*[a]

We obtain the bounds *of a given variable in a given formula with*

$$\mathrm{vb}\colon (\mathrm{VAR}_{\mathbb{R},\mathbb{Z}} \times \mathrm{FO}(\tau)) \to \mathbb{I} : (c_1 \wedge .. \wedge c_n \wedge \varphi_1 \wedge .. \wedge \varphi_m, x) \mapsto \mathrm{vb}'(x, \{c_1, .., c_n\})$$

where $\mathrm{vb}'\colon (\mathrm{VAR}_{\mathbb{R},\mathbb{Z}} \times \mathbb{P}(\mathrm{CS}_{\mathrm{VB}})) \to \mathbb{I}$ *maps a variable* x *and a set of bound constraints* C *as depicted in the following table*

$\sup(x, C, \to)$ $\inf(x, C, \downarrow)$	$\bot$	$a'x - d' \leq 0$	$a'x - d' < 0$
$\bot$	$(-\infty, \infty)$	$(-\infty, \frac{d'}{a'}]$	$(-\infty, \frac{d'}{a'})$
$ax - d \geq 0$	$[\frac{d}{a}, \infty)$	$[\frac{d}{a}, \frac{d'}{a'}]$, *if* $\frac{d'}{a'} > \frac{d}{a}$ $\emptyset$, *otherwise*	$[\frac{d}{a}, \frac{d'}{a'})$, *if* $\frac{d'}{a'} \geq \frac{d}{a}$ $\emptyset$, *otherwise*
$ax - d > 0$	$(\frac{d}{a}, \infty)$	$(\frac{d}{a}, \frac{d'}{a'}]$, *if* $\frac{d'}{a'} \geq \frac{d}{a}$ $\emptyset$, *otherwise*	$(\frac{d}{a}, \frac{d'}{a'})$, *if* $\frac{d'}{a'} \geq \frac{d}{a}$ $\emptyset$, *otherwise*

with

$$\sup\colon (\mathrm{VAR}_{\mathbb{R},\mathbb{Z}} \times \mathbb{P}(\mathrm{CS}_{\mathrm{VB}})) \to \mathrm{CS}_{\mathrm{VB}} \cup \{\bot\} :$$
$$(x, C) \mapsto \begin{cases} ax - d < 0 & , \text{if } \exists\ ax - d < 0 \in C. \\ & \quad \forall\ a'x - d' \sim' 0 \in C.\ \sim' \in \{=, \leq, <\} \to \frac{d}{a} \leq \frac{d'}{a'} \\ ax - d \sim 0 & , \text{if } \exists\ \sim \in \{=, \leq\} \wedge ax - d \sim 0 \in C \wedge \\ & \quad \forall\ a'x - d' \sim' 0 \in C.\ \sim' \in \{=, \leq, <\} \to \frac{d}{a} < \frac{d'}{a'} \\ \bot & , \text{otherwise} \end{cases}$$

$$\inf\colon (\mathrm{VAR}_{\mathbb{R},\mathbb{Z}} \times \mathbb{P}(\mathrm{CS}_{\mathrm{VB}})) \to \mathrm{CS}_{\mathrm{VB}} \cup \{\bot\} :$$
$$(x, C) \mapsto \begin{cases} ax - d > 0 & \text{, if } \exists\ ax - d > 0 \in C. \\ & \quad \forall\ a'x - d' \sim' 0 \in C.\ \sim' \in \{=, \geq, >\} \to \frac{d}{a} \geq \frac{d'}{a'} \\ ax - d \sim 0 & \text{, if } \sim \in \{=, \geq\} \wedge ax - d \sim 0 \in C \wedge \\ & \quad \forall\ a'x - d' \sim' 0 \in C.\ \sim' \in \{=, \geq, >\} \to \frac{d}{a} > \frac{d'}{a'} \\ \bot & \text{, otherwise} \end{cases}$$

[a] Note that due to φ being normalized, $\varphi_1, .., \varphi_m$ are not conjunctions.

In other words, the bounds of a variable x in a formula φ are defined by the strictest bound constraints, which have to hold such that φ can be satisfied. These strictest bound constraints are the strictest lower bound constraint of x, if any lower bound constraint for x exists, and the strictest upper bound constraint of x, if any upper bound constraint for x exists. For a strictest lower or upper bound constraint c of x it holds that for all other lower respectively upper bounds c' of x the formula $c \implies c'$ is valid. We represent a variable's bounds by an interval, where the strictest lower bound constraint specifies the interval's lower bound and the strictest upper bound constraint specifies the interval's upper bound.

5.4.1 Interval arithmetic

Similarly as with numbers (Z, $\mathbb{Q}$, $\mathbb{R}$, etc.), we can compute with intervals. All common arithmetic operations, such as the addition ($+$), subtraction ($-$), multiplication ($\cdot$) and division ($/$) of two intervals are defined. We can also compute the i-th root ($\sqrt[i]{\cdot}$) of an interval for $i \in \mathbb{N}$, as the bounds of an interval are real valued which allows us to use arbitrarily nested root expressions, as it can be seen in the following example.

Example 16

$$\begin{array}{lcll} [-1,3) & + & [0,2] & = [-1,5) \\ (1,\infty) & - & [1,1] & = (0,\infty) \\ (1,2] & \cdot & [3,4) & = (3,8) \\ (-\infty,0] & \cdot & (-2,0] & = [0,\infty) \\ [1,2] & / & (2,3] & = [\frac{1}{3},1) \\ & & \sqrt[3]{[2,\sqrt{5}]} & = [\sqrt[3]{2}, \sqrt[6]{5}] \end{array}$$

Dividing an interval by an interval, which contains zero, might not result in a *convex set* M. That means there exist values $d_1, d_2 \in M$ such that there exists a value $d' \in \mathbb{R} \setminus M$ with $d_1 < d' < d_2$. As the convex subsets of $\mathbb{R}$ are exactly the sets we can represent with a single interval, there is no interval representation for M.

Example 17

$$[1,2] \ / \ [-2,3] \ = \ (-\infty, -\tfrac{1}{2}] \cup [\tfrac{1}{3}, \infty)$$

Therefore, interval arithmetic is not closed under division.

We will not give further details on how all of these arithmetic operations on intervals are defined and refer to [Kul09] instead. The following property, which is always preserved by any arithmetic operation $\circ \in \{+, -, \cdot, /\}$ on intervals, is essential for using them in our context where we have to guarantee correctness.

$$\begin{array}{rrcl} \forall\ I_1, I_2 \in \mathbb{I}. & I_1 \circ I_2 & = & \{d_1 \circ d_2 |\ d_1 \in I_1,\ d_2 \in I_2\} \\ \forall\ I \in \mathbb{I}. & \sqrt{I} & = & \{\sqrt{d} |\ d \in I,\ d \geq 0\} \end{array} \tag{5.1}$$

In the implementation of the ideas we present in the course of this section, we use `IEEE` standard floating points instead of arbitrary real numbers for the bound values of the intervals. This improves the performance of the arithmetic operations, however it does not guarantee that the Property 5.1 is still fulfilled. Instead we have to mitigate this property such that the arithmetic operations only include the results of all combinations of the elements of the (two) input interval(s):

$$\begin{array}{rrcl} \forall\ I_1, I_2 \in \mathbb{I}. & \forall\ d_1 \in I_1,\ d_2 \in I_2. & d_1 \circ d_2 \in I_1 \circ I_2 \\ \forall\ I \in \mathbb{I}. & \forall\ d \in I,\ d \geq 0. & \sqrt{d} \in \sqrt{I} \end{array} \tag{5.2}$$

Therefore, correctness can still be guaranteed, if Equation (5.2) is fulfilled.

We can also compare intervals by $I_1 < I_2$, which is true if $I_1 \cap I_2 = \emptyset$ and the upper bound of I_1 is less than or equal to the lower bound of I_2. However, the equality of two intervals I_1 and I_2 does not follow from $\neg(I_1 < I_2) \wedge \neg(I_2 < I_1)$ as it does for numbers. For instance, $I_1 = (-\infty, 1)$ and $I_2 = [0, 1]$ satisfy this condition, but are obviously not equal. As expected, two intervals are equal if their lower bounds are equal and either both are closed or both are open and their upper bounds are equal and either both are closed or both are open. Similar to the τ-structure $\mathfrak{A}$, the τ-structure $\mathfrak{B}$ maps $+$, $-$, $\cdot$ and $<$ to the aforementioned semantics for interval arithmetic denoted by $+_{\mathfrak{B}}$, $-_{\mathfrak{B}}$, $\cdot_{\mathfrak{B}}$ and $<_{\mathfrak{B}}$. Additionally, it maps an $I \in \mathbb{I}$, $\sqrt[i]{\cdot}$ and $=$ to the just described semantics denoted by $I_{\mathfrak{B}}$, $\sqrt[i]{\cdot}_{\mathfrak{B}}$ and $=_{\mathfrak{B}}$.

If we directly apply the n-th exponentiation to an interval I instead of computing $\prod_{i=1}^{n} I$, we might obtain a different result. More precisely, we know that I^n cannot contain any negative numbers if n is even. For instance, $(-1, \infty)^2 = [0, \infty)$, where using the standard multiplication of intervals yields $(-1, \infty) \cdot (-1, \infty) = (-\infty, \infty)$. We call this the *wrapping effect* and there are more effects which can be exploited in interval arithmetic for which we refer to [MKC09].

5.4.2 Evaluation and simplification of formulas using variable bounds

The main purpose of intervals in this thesis is to use them as an over-approximation of the solution space of a formula for a variable. Similarly as for numbers, we also want to evaluate expressions such as polynomials, constraints and formulas by an assignment of variables to intervals.

Definition 30 (Interval assignment) *An* assignment of intervals to real-valued variables *(short* interval assignment*) is defined by*

$$B\colon\ \mathrm{VAR}_{\mathbb{R}} \to \mathbb{I} : v \mapsto I \subseteq \mathrm{Dom}(v).$$

Analogously to Definition 6, we consider an interval assignment B to be *full* for a given formula (or polynomial) φ, if $\mathrm{FreeVars}(\varphi) \subseteq \mathrm{Dom}(B)$. Within this thesis we only consider full interval assignments and denote the *set of all full interval assignments* by $\mathbb{I}\mathrm{ASS}$.

Just as for assignments to values, we can adapt an interval assignment such that it maps a variable to a given interval by

$$\cdot\,[\,\cdot\,/\,\cdot\,]\colon\ \mathbb{I}\mathrm{ASS} \times \mathbb{I} \times \mathrm{VAR}_{\mathbb{R},\mathbb{Z}} \to \mathbb{I}\mathrm{ASS} : B[I/x] \mapsto B'$$

where

$$B'(x') \;=\; \begin{cases} I & \text{, if } x' = x \\ B(x') & \text{, otherwise.} \end{cases}$$

For the evaluation of a formula or a polynomial under an interval assignment, it is sufficient in this context if we restrict ourselves to quantifier-free arithmetic formulas without Boolean variables. A generalization to arbitrary arithmetic formulas is straight forward.

Definition 31 (Formula and polynomial evaluation) *Given a full interval assignment B for a polynomial or a quantifier-free arithmetic formula φ without Boolean variables, we can* evaluate it under B *by*

$$[\![\,\cdot\,]\!]^{\cdot}\ :\ (\mathrm{FO}(\tau) \times \mathbb{I}\mathrm{ASS}) \to \mathrm{FO}(\tau),$$

which is defined inductively with respect to the abstract grammar in Definition 4

$$\begin{array}{lcl}
[\![x]\!]^B & := & B(x)_{\mathfrak{B}} \\
[\![1]\!]^B & := & [1,1]_{\mathfrak{B}} \\
[\![p_1 + p_2]\!]^B & := & [\![p_1]\!]^B +_{\mathfrak{B}} [\![p_2]\!]^B \\
[\![p_1 - p_2]\!]^B & := & [\![p_1]\!]^B -_{\mathfrak{B}} [\![p_2]\!]^B \\
[\![p_1 \cdot p_2]\!]^B & := & [\![p_1]\!]^B \cdot_{\mathfrak{B}} [\![p_2]\!]^B \\
[\![p_1 < p_2]\!]^B & := & \begin{cases} \text{true} & \text{, if } [\![p_1]\!]^B <_{\mathfrak{B}} [\![p_2]\!]^B \\ \text{false} & \text{, if } [\![p_2]\!]^B <_{\mathfrak{B}} [\![p_1]\!]^B \\ p_1 < p_2 & \text{, otherwise} \end{cases} \\
[\![\neg\varphi_1]\!]^B & := & \begin{cases} \text{false} & \text{, if } [\![\varphi_1]\!]^B = \text{true} \\ \text{true} & \text{, if } [\![\varphi_1]\!]^B = \text{false} \\ \varphi_1 & \text{, otherwise} \end{cases}
\end{array}$$

$$[\![\varphi_1 \wedge \varphi_2]\!]^B := \begin{cases} \text{false} & \text{, if } [\![\varphi_1]\!]^B = \text{false or } [\![\varphi_2]\!]^B = \text{false} \\ \varphi_1 & \text{, if } [\![\varphi_2]\!]^B = \text{true} \\ \varphi_2 & \text{, if } [\![\varphi_1]\!]^B = \text{true} \\ \varphi_1 \wedge \varphi_2 & \text{, otherwise} \end{cases}$$

where $x \in \mathrm{Dom}(B)$ is an arithmetic variable, p_1 and p_2 are polynomials and φ_1 and φ_2 are quantifier-free arithmetic formulas without Boolean variables.

Definition 31 is very similar to Definition 7, however, we cannot evaluate constraints to true or false, if the evaluations of the left- and right-hand sides intersect. Instead, the evaluation then keeps the constraints, which implies that the evaluation of a formula does not always result in a Boolean constant, but can also yield a (simplified) formula.

Example 18 *Consider the interval assignment B with*

$$\begin{aligned} B(x_1) &= (-\infty, 0] \\ \textit{and}\quad B(x_2) &= (1, 2]. \end{aligned}$$

Then evaluating the formula $2x_1 + 1 < x_2 \wedge \neg(x_1 x_2 < 0)$ under B yields

$$[\![\ \overbrace{2x_1 + 1 < x_2}^{\substack{([2,2]_{\mathfrak{B}} \cdot_{\mathfrak{B}} (-\infty,0]_{\mathfrak{B}}) +_{\mathfrak{B}} [1,1]_{\mathfrak{B}} <_{\mathfrak{B}} (1,2]_{\mathfrak{B}} \\ \equiv \\ (-\infty,1]_{\mathfrak{B}} <_{\mathfrak{B}} (1,2]_{\mathfrak{B}} \\ \equiv \\ \text{true}}} \wedge \neg(\overbrace{x_1 x_2 < 0}^{\substack{(-\infty,0]_{\mathfrak{B}} \cdot_{\mathfrak{B}} (1,2]_{\mathfrak{B}} <_{\mathfrak{B}} [0,0]_{\mathfrak{B}} \\ \equiv \\ (-\infty,0]_{\mathfrak{B}} <_{\mathfrak{B}} [0,0]_{\mathfrak{B}}}})]\!]^B \equiv \neg(x_1 x_2 < 0)$$

5.4.3 Interval constraint propagation

Given a formula φ with its decomposition $\varphi = c_1 \wedge .. \wedge c_n \wedge \varphi_1 \wedge .. \wedge \varphi_m$ according to Definition 29, we have seen how to obtain bounds for the variables in $\mathrm{Vars}(\varphi)$. However, we did only consider the bound constraints $c_1 \wedge .. \wedge c_n$ for these bounds and ignored the other constraints $\{\varphi_i |\ 1 \leq i \leq m,\ \varphi_i \text{ is a constraint}\}$, which have to hold such that φ can be satisfied. In Example 19 we illustrate the main idea of how *interval constraint propagation* (ICP) uses these constraints to refine an over-approximation of φ's solution space for its variables step by step. We only provide this informal illustration and refer to [FHT+07] for more details. An alternative procedure, which uses contraction with an interval-based Newton method [HR97] instead of propagation, can be found in [GGI+10].

Example 19 *Consider the real-arithmetic formula*

$$\varphi \quad = \quad 2x_2 + 1 \geq 0 \ \wedge\ x_1 + 1 > 0 \ \wedge\ x_1^2 + x_2^2 - 2 = 0 \ \wedge\ x_1^2 - x_2 - 1 = 0 \ \wedge\ 6x_1 - 2x_2 - 3 \geq 0$$

which is illustrated by Figure 5.1. Its only solution is $\{(x_1, \sqrt{\frac{\sqrt{5}+1}{2}}), (x_2, \frac{\sqrt{5}-1}{2})\}$, which is the right

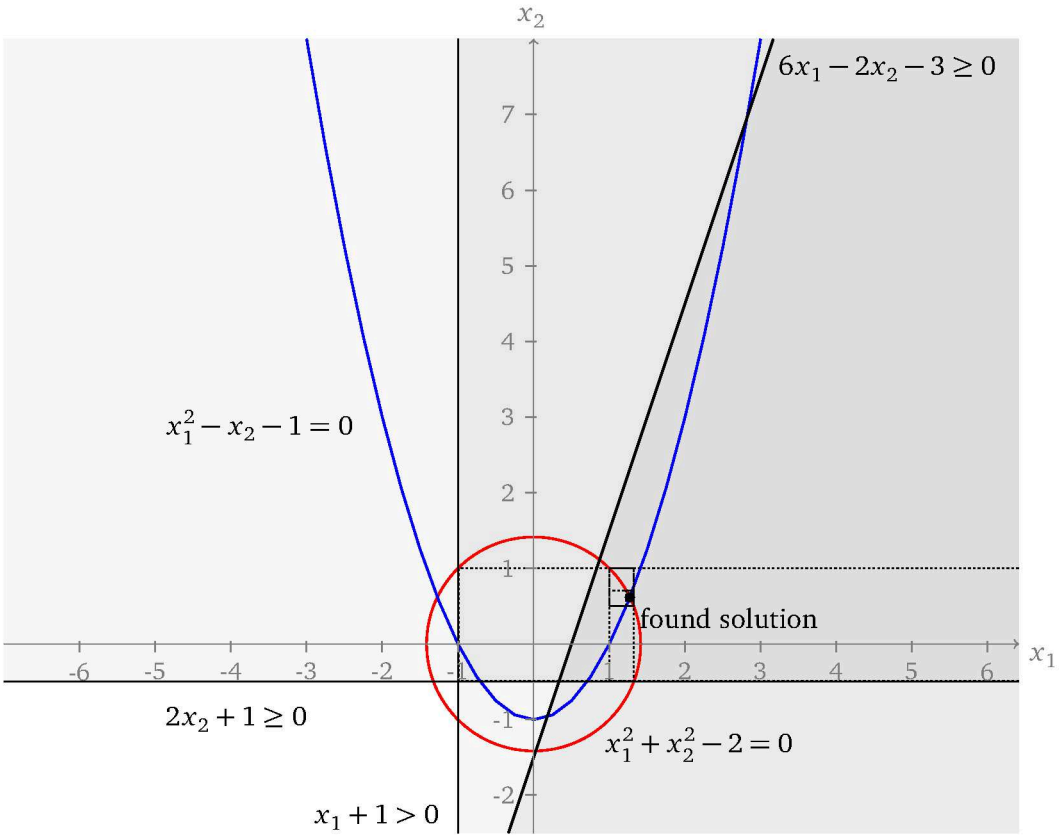


Figure 5.1: Intersection of the solution sets of the constraints $2x_2+1 \geq 0$, $x_1+1 > 0$, $x_1^2+x_2^2-2 = 0$, $x_1^2-x_2-1 = 0$ and $6x_1-2x_2-3 \geq 0$. The dotted boxes show how we narrow down an over-approximation of the intersection approaching its only element at $(\sim 1.272, \sim 0.618)$.

of the two intersection points of the equations $x_1^2 + x_2^2 - 2 = 0$ *and* $x_1^2 - x_2 - 1 = 0$ *in Figure 5.1. The bound constraints of* φ *are* $\{2x_2 + 1 \geq 0,\ x_1 + 1 > 0\}$ *and we obtain:*

$$\begin{aligned} \mathrm{vb}(x_1, \varphi) &= (-1, \infty) \\ \mathrm{vb}(x_2, \varphi) &= [-\tfrac{1}{2}, \infty) \end{aligned}$$

In the following we informally apply ICP to clarify its principle. Let us consider the constraints $x_1^2 + x_2^2 - 2 = 0$ *and* $x_1^2 - x_2 - 1 = 0$*. If we replace all nonlinear monomials, i. e.,* x_1^2 *and* x_2^2*, by fresh real arithmetic variables* x_3 *and* x_4 *we obtain*

$$\begin{aligned} c_1 &: \quad x_3 + x_4 - 2 = 0 \\ c_2 &: \quad x_3 - x_2 - 1 = 0 \\ c_3 &: \quad x_3 - x_1^2 = 0 \\ c_4 &: \quad x_4 - x_2^2 = 0 \end{aligned}$$

Usually, we would also replace multivariate linear polynomials without their constant part, i. e.,

$6x_1 - 2x_2$, $x_3 + x_4$ *and* $x_3 - x_2$, *by fresh real-arithmetic variables, but we omit this here for the sake of simplicity. For the same reason, we do not use the constraint* $6x_1 - 2x_2 - 3 \geq 0$ *for propagation. Using the bounds of* x_1 *and* x_2, *which are defined by* $\mathrm{vb}(x_1, \varphi)$ *and* $\mathrm{vb}(x_2, \varphi)$, *we initialize the interval assignment B, mapping each variable in* φ *to an over-approximation of* φ*'s solution space for the variable by:*

$$\begin{aligned} B(x_1) &= (-1, \infty) \\ B(x_2) &= [-\tfrac{1}{2}, \infty) \\ B(x_3) &= (-\infty, \infty) \\ B(x_4) &= (-\infty, \infty) \end{aligned}$$

Due to the special form of the equations c_1, c_2, c_3 *and* c_4, *we can solve them for each variable occurring in them. For instance, if we solve* c_3 *for* x_3, *we get* $x_3 = x_1^2$ *or, if we solve* c_4 *for* x_2, *we get* $x_2 = \pm\sqrt{x_4}$. *Let* s_{x_i,c_j} *define the right-hand sides of these solution equations for* $i, j \in \{1, .., 4\}$ *with* $s_{x_i,c_j} = \bot$, *if* $x_i \notin \mathrm{Vars}(c_j)$. *Otherwise,* $s_{x_i,c_j} = \frac{\sqrt[n]{p}}{d}$ *with* $n \in \mathbb{N}$, $d \in \mathbb{Z}$ *and* p *being a polynomial. Then (just for this example) an* ICP-step, *which narrows down the over-approximating solution space of one of the variables* x_i *with one of the equations* c_j *(*$i, j \in \{1, .., 4\}$*), is defined by*

$$\mathrm{icp}\colon (\mathbb{I} \times \{x_1, .., x_4\} \times \{c_1, .., c_4\}) \to \mathbb{I}\colon (I, x_i, c_j) \mapsto \begin{cases} I & , \text{ if } s_{x_i,c_j} = \bot \\ I \cap \frac{\sqrt[n]{[\![p]\!]^B}}{[d,d]} & , \text{ if } s_{x_i,c_j} = \frac{\sqrt[n]{p}}{d} \\ I \cap \pm\frac{\sqrt[n]{[\![p]\!]^B}}{[d,d]} & , \text{ if } s_{x_i,c_j} = \pm\frac{\sqrt[n]{p}}{d} \end{cases}$$

Now we can update $B(x_3)$ *by an ICP-step using* c_3:

$$B(x_3) \;=\; \mathrm{icp}(B(x_3), x_3, c_3) = \; (-\infty, \infty) \cap (-1, \infty)^2 = \; [0, \infty)$$

With the same idea we can narrow down the over-approximation of φ*'s solution space for its variables as follows. We also illustrate this process in Figure 5.1 by the dotted boxes.*

$$\begin{array}{lllllll} B(x_4) & := \mathrm{icp}(B(x_4), x_4, c_4) & = (-\infty, \infty) & \cap & [-\frac{1}{2}, \infty)^2 & = [0, \infty) \\ B(x_3) & := \mathrm{icp}(B(x_3), x_3, c_1) & = [0, \infty) & \cap & ([2,2] - [0, \infty)) & = [0, 2] \\ B(x_2) & := \mathrm{icp}(B(x_2), x_2, c_2) & = [-\frac{1}{2}, \infty) & \cap & ([0,2] - [1,1]) & = [-\frac{1}{2}, 1] \\ B(x_3) & := \mathrm{icp}(B(x_3), x_3, c_2) & = [0, 2] & \cap & ([-\frac{1}{2}, 1] + [1,1]) & = [\frac{1}{2}, 2] \\ B(x_4) & := \mathrm{icp}(B(x_4), x_4, c_4) & = [0, \infty) & \cap & [-\frac{1}{2}, 1]^2 & = [\frac{1}{4}, 1] \\ B(x_3) & := \mathrm{icp}(B(x_3), x_3, c_1) & = [\frac{1}{2}, 2] & \cap & ([2,2] - [\frac{1}{4}, 1]) & = [1, \frac{7}{4}] \\ B(x_1) & := \mathrm{icp}(B(x_1), x_1, c_3) & = (-1, \infty) & \cap & \pm\sqrt{[1, \frac{7}{4}]} & = [1, \frac{\sqrt{7}}{2}] \\ B(x_2) & := \mathrm{icp}(B(x_2), x_2, c_4) & = [-\frac{1}{2}, 1] & \cap & \pm\sqrt{[\frac{1}{4}, 1]} & = [-\frac{1}{2}, -\frac{1}{2}] \cup [\frac{1}{2}, 1] \end{array}$$

At this point, the over-approximation of φ*'s solution space for* x_2 *is not convex and here are three possibilities as to how we can proceed:*

1. *We could start to operate on non-convex sets represented by sets of intervals. Then an arith-*

metic operation of two of these sets of intervals $M_1, M_2 \subset \mathbb{I}$ would entail applying an interval arithmetic operation for each combination of an element from M_1 with an element from M_2. In our scenario, however, this is not a good option as a repeated application of such an operation might lead to a combinatorial explosion.

2. *We could form the* convex hull *of the intervals in $M \subset \mathbb{I}$ representing the non-convex set. This is simply achieved by taking the minimal lower-bound of all intervals as the lower bound of the convex hull and the maximal upper-bound of all intervals as the upper bound of the convex hull. In many cases this can be a coarse over-approximation, though.*

3. *Considering that $B(x_2) = [-\frac{1}{2}, -\frac{1}{2}] \cup [\frac{1}{2}, 1]$, we can also try to first deal with $x_2 \in [-\frac{1}{2}, -\frac{1}{2}]$ and, if we could rule out that it contains any solution for φ, we continue with $x_2 \in [\frac{1}{2}, 1]$. Instead of handling this by a simple case splitting, we can also lift this decision to an involved SAT solver as presented in Section 3.1.*

Let us assume that we follow the third possibility and consider $x_2 \in [-\frac{1}{2}, -\frac{1}{2}]$, first. Then we get a conflict straight away if we update $B(x_3)$ using c_2.

$$B(x_3) \;:=\; \text{icp}(B(x_3), x_3, c_2) \;=\; [1, \tfrac{3}{2}] \;\cap\; ([-\tfrac{1}{2}, -\tfrac{1}{2}] + [1,1]) \;=\; \emptyset$$

Therefore, we now consider $x_2 \in [\frac{1}{2}, 1]$. If we repeat the following four updates, we approach the solution $\{(x_1, \sqrt{\frac{1+\sqrt{5}}{2}}), (x_2, \frac{-1+\sqrt{5}}{2})\}$, which lies approximately at $(1.272, 0.618)$ in Figure 5.1, for both variables x_1 and x_2 from below and from above.

$$\begin{array}{lllllll}
B(x_3) & := & \text{icp}(B(x_3), x_3, c_2) & = [1, \frac{7}{4}] & \cap & ([\frac{1}{2}, 1] + [1,1]) & = [\frac{3}{2}, \frac{7}{4}] \\
B(x_4) & := & \text{icp}(B(x_4), x_4, c_1) & = [\frac{1}{4}, 1] & \cap & ([2,2] - [\frac{3}{2}, \frac{7}{4}]) & = [\frac{1}{4}, \frac{1}{2}] \\
B(x_2) & := & \text{icp}(B(x_2), x_2, c_4) & = [\frac{1}{2}, 1] & \cap & \pm\sqrt{[\frac{1}{4}, \frac{1}{2}]} & = [\frac{1}{2}, \frac{1}{\sqrt{2}}] \\
B(x_1) & := & \text{icp}(B(x_1), x_1, c_3) & = [1, \frac{\sqrt{7}}{2}] & \cap & \pm\sqrt{[\frac{3}{2}, \frac{7}{4}]} & = [\frac{\sqrt{3}}{\sqrt{2}}, \frac{\sqrt{7}}{2}]
\end{array}$$

Unfortunately, we would never yield point intervals as an over-approximation of φ's solution space for the variables x_1 and x_2, if we only apply this procedure. That is, we never obtain the exact solution in this case, but we can isolate it with an infinite precision. Furthermore, we cannot guarantee at any point during the application of this procedure, that the over-approximation of the formula's solution space contains a solution. Let us assume, that we consider the formula $\hat{\varphi} = \varphi \wedge x_1^2 + x_2^2 - 2 + \epsilon = 0$ where ϵ is a very small real number with $\epsilon > 0$. Clearly, the equations $x_1^2 + x_2^2 - 2 = 0$ and $x_1^2 + x_2^2 - 2 + \epsilon = 0$ have no common solutions, as the former circle is enclosed by the latter one. However, ICP would yield a very similar over-approximation to the one we found for φ, but containing an intersection point of each circle with the parabola. In order to be able to isolate both intersection points and detect that they are not a solution, afterwards, we need to split the over-approximation of $\hat{\varphi}$'s solution space for at least one of the two variables x_1 and x_2 at a point between these intersection points and consider the two resulting halves independently.

This example also shows that we need to define a termination criteria in order the ensure the completeness of ICP. More precisely, we need to know when to stop approaching a possible solution as it was the case in our example. The standard termination criteria for ICP is to check whether there exists no ICP-step, which reduces the diameter of the over-approximation of the formula's solution space for a variable more than a certain contraction threshold μ.

5.4.4 Using variable bounds to filter out test candidates

We have seen how to obtain an initial over-approximation B of a given formula φ's solution space using Definition 29. Furthermore, we illustrated how to refine this over-approximation with ICP in Example 19. Even though ICP could not determine the satisfiability of the formula in the example, there are cases where it can. If, for instance, it refines B such that it maps a variable to an empty interval, we know that φ is unsatisfiable. We could also test any assignment α such that for each variable x it holds that $\alpha(x) \in B(x)$. If it is a satisfying assignment, φ is obviously satisfiable. It might seem to be a rather arbitrary step to do this, but especially as B might be exactly the solution space, it makes sense to perform this cheap test.

The question remains as to how to deal with a case similar to Example 19, where we could refine the initial over-approximation B of a given formula φ's solution space, but not in a way that we can follow the satisfiability of φ. Of course, we can invoke another procedure for this purpose. For instance, the CAD is a complete procedure for real-arithmetic formulas. As we expect that the performance of ICP is better than the one of the CAD, one possible strategy for solving a real-arithmetic formula would be to use ICP first and if it cannot determine the formula's satisfiability, we use the CAD. However, we would discard the valuable information that ICP possibly narrowed down the formula's solution space. If we invoke the CAD additionally with the constraints which represent this tighter solution space, we must make sure that the CAD implementation makes use of this extra information. In [5] we presented an adaption of the operations used in the CAD such that they exploit the variables' bounds.

In this section, we want to make use of them in order to improve the performance of the virtual substitution. Let us consider its main idea which was defined in Theorem 2. Given a real-arithmetic formula $\varphi^{\mathbb{R}}$ with the real-valued variable $x \in \mathrm{Vars}(\varphi^{\mathbb{R}})$, this theorem gives us an equivalent formula which only contains the variables $\mathrm{Vars}(\varphi) \setminus \{x\}$:

$$\exists x.\, \varphi^{\mathbb{R}} \quad \Leftrightarrow \quad \bigvee_{t \in \mathrm{tcs}(x, \varphi^{\mathbb{R}})} (\varphi^{\mathbb{R}}[t /\!/ x] \wedge \mathrm{sc}(t))$$

Remember that the equivalent formula holds if there exists a test candidate $t \in \mathrm{tcs}(x, \varphi^{\mathbb{R}})$ for x in $\varphi^{\mathbb{R}}$, such that the result of virtually substituting x in $\varphi^{\mathbb{R}}$ by t holds under consideration of t's side conditions $\mathrm{sc}(t)$.

Hence, the first possibility to improve the performance of the virtual substitution is to reduce the number of test candidates which we have to take into account. As the following theorem

shows, we can use the bounds of the variables in $\varphi^{\mathbb{R}}$ in order to filter out the test candidates which cannot satisfy $\varphi^{\mathbb{R}}$.

Theorem 6 *Let $\varphi^{\mathbb{R}}$ be a quantifier-free real-arithmetic formula where the real-valued variable $x \in \mathrm{Vars}(\varphi^{\mathbb{R}})$ occurs at most quadratic in $\varphi^{\mathbb{R}}$. Then it holds that*

$$\exists x.\ \varphi^{\mathbb{R}} \quad \Leftrightarrow \quad \bigvee_{t \in \mathrm{tcs}_{\mathbb{I}}(x, \varphi^{\mathbb{R}}, \mathrm{vb}(x, \varphi^{\mathbb{R}}))} (\varphi^{\mathbb{R}}[t//x] \wedge \mathrm{sc}(t)). \tag{5.3}$$

where

$$\mathrm{tcs}_{\mathbb{I}}\colon\ (\mathrm{VAR}_{\mathbb{R},\mathbb{Z}} \times \mathrm{FO}(\tau) \times \mathbb{I}\mathrm{ASS}) \to \mathrm{TCS}:$$

$$(x, \varphi, B) \mapsto \{t \in \mathrm{tcs}(x, c) | c \in C_{\sim}(\varphi),\ \square(x, t, B) \neq \emptyset\}$$

with

$$\square\colon\ (\mathrm{VAR}_{\mathbb{R},\mathbb{Z}} \times \mathrm{TCS} \times \mathbb{I}\mathrm{ASS}) \to \mathbb{P}(\mathbb{R}):$$

$$(x, t, B) \mapsto \begin{cases} B(x) & \text{, if } t = -\infty \text{ and } B(x) \text{ is left unbounded} \\ \emptyset & \text{, if } t = -\infty \text{ and } B(x) \text{ is left bounded} \\ (I_1 \cup I_2) \cap B(x) & \text{, if } t = \frac{p+q\sqrt{r}}{s} \text{ and } \frac{[\![p]\!]^B + [\![q]\!]^B \sqrt{[\![r]\!]^B}}{[\![s]\!]^B} = I_1 \cup I_2 \text{ [1]} \\ (I_1 + \epsilon \cap B(x)) \cup (I_2 + \epsilon \cap B(x)) & \text{, if } t = \frac{p+q\sqrt{r}}{s} + \epsilon \text{ and } \frac{[\![p]\!]^B + [\![q]\!]^B \sqrt{[\![r]\!]^B}}{[\![s]\!]^B} = I_1 \cup I_2 \end{cases}$$

and we over-approximate $I + \epsilon$ for $I \in \mathbb{I}$ by choosing a small value $e \in \mathbb{R}$ ($e > 0$) and using the following rules:

$$\begin{array}{lcll} (a, \infty) + \epsilon & \subseteq & (a, \infty) & \\ [a, \infty) + \epsilon & \subseteq & (a, \infty) & \\ (a, b) + \epsilon & \subseteq & (a, b] & , \ b \neq \infty \\ [a, b) + \epsilon & \subseteq & (a, b] & , \ b \neq \infty \\ (a, b] + \epsilon & \subseteq & (a, b + e] & \\ [a, b] + \epsilon & \subseteq & (a, b + e] & \end{array}$$

Proof 6 **"⇐":** This direction is trivially fulfilled as

$$\begin{array}{cc} & \bigvee_{t \in \mathrm{tcs}_{\mathbb{I}}(x, \varphi^{\mathbb{R}}, \mathrm{vb}(x, \varphi^{\mathbb{R}}))} (\varphi^{\mathbb{R}}[t//x] \wedge \mathrm{sc}(t)) \\ \overset{\mathrm{tcs}_{\mathbb{I}}(x, \varphi^{\mathbb{R}}, \mathrm{vb}(x, \varphi^{\mathbb{R}})) \subseteq \mathrm{tcs}(x, \varphi^{\mathbb{R}})}{\Rightarrow} & \bigvee_{t \in \mathrm{tcs}(x, \varphi^{\mathbb{R}})} (\varphi^{\mathbb{R}}[t//x] \wedge \mathrm{sc}(t)) \\ \overset{\text{Thm.2}}{\Leftrightarrow} & \exists x.\varphi^{\mathbb{R}} \end{array}$$

"⇒":

$$\exists x.\varphi^{\mathbb{R}} \overset{\text{Thm.2}}{\Longleftrightarrow} \bigvee_{t \in \text{tcs}(x,\varphi^{\mathbb{R}})} (\varphi^{\mathbb{R}}[t//x] \wedge \text{sc}(t))$$

$$\overset{(*)}{\Rightarrow} \bigvee_{\substack{t \in \text{tcs}(x,\varphi^{\mathbb{R}}) \\ \square(x,t,B) \neq \emptyset}} (\varphi^{\mathbb{R}}[t//x] \wedge \text{sc}(t)) \quad \vee \overbrace{\bigvee_{\substack{t \in \text{tcs}(x,\varphi^{\mathbb{R}}) \\ \square(x,t,B) = \emptyset}} (\varphi^{\mathbb{R}}[t//x] \wedge \text{sc}(t))}^{\text{false}}$$

$$\overset{(**)}{\Rightarrow} \bigvee_{t \in \text{tcs}_{\mathbb{I}}(x,\varphi^{\mathbb{R}},\text{vb}(x,\varphi^{\mathbb{R}}))} (\varphi^{\mathbb{R}}[t//x] \wedge \text{sc}(t))$$

$(*)$ Here, we simply split the test candidates into two disjoint sets.

$(**)$ If for a $t \in \text{tcs}(x,\varphi^{\mathbb{R}})$ it holds that $\square(x,t,B) = \emptyset$, we know that no assignment in an over-approximation of the solution space of $\varphi^{\mathbb{R}}$ satisfies $x = t$. This means that $\varphi_{B,t}[t//x]$ is already unsatisfiable, where $\varphi_{B,t}$ consists of the bound constraints which define B and share variables with t. Therefore, we in particular know that

$$\bigvee_{\substack{t \in \text{tcs}(x,\varphi^{\mathbb{R}}) \\ \square(x,t,B) = \emptyset}} (\varphi^{\mathbb{R}}[t//x] \wedge \text{sc}(t)) \quad \equiv \quad \text{false}.$$

□

Given an over-approximation of the solution space of a real-arithmetic formula $\varphi^{\mathbb{R}}$,Theorem 6 states that we do not need to consider all test candidates for the variable to eliminate in $\varphi^{\mathbb{R}}$, but only those which can lie in this over-approximation.

Example 20 *Considering Figure 5.1 and the tighter over-approximation*

$$\begin{aligned} B(x_1) &\in [\tfrac{\sqrt{3}}{\sqrt{2}}, \tfrac{\sqrt{7}}{2}] \\ B(x_2) &\in [\tfrac{1}{2}, \tfrac{1}{\sqrt{2}}] \end{aligned}$$

which we obtained in Example 19 using ICP, any test candidate provided by the constraint $6x_1 - 2x_2 - 3 \geq 0$ *can be omitted. If we would, for instance, eliminate* x_1 *first,* $6x_1 - 2x_2 - 3 \geq 0$ *provides the test candidates* $t_1 = -\infty$ *and* $t_2 = \frac{2x_2+3}{6}$*. As*

$$\begin{aligned} & \square(x,t_1,B) = \emptyset \\ \textit{and} \quad & \square(x,t_2,B) = \tfrac{[2,2]\cdot[\frac{1}{2},\frac{1}{\sqrt{2}}]+[3,3]}{[6,6]} \cap [\tfrac{\sqrt{3}}{\sqrt{2}}, \tfrac{\sqrt{7}}{2}] = [\tfrac{2}{3}, \tfrac{3+\sqrt{2}}{6}] \cap [\tfrac{\sqrt{3}}{\sqrt{2}}, \tfrac{\sqrt{7}}{2}] = \emptyset \end{aligned}$$

we can disregard them according to Theorem 6. In Figure 5.1, we can see that the polynomial $x_1 - x_2$ *defining this constraint does not intersect with B. As* t_1 *represents the points on this line, it cannot form a solution.*

The constraint $x_1^2 + x_2^2 - 2 = 0$*, on the other hand, provides besides* $-\infty$ *the test candidates*

$$t_3 = -\sqrt{-x_2^2+2} \text{ and } \quad t_4 = \sqrt{-x_2^2+2}$$

and in this case we can only disregard t_3*, as*

$$\begin{array}{rrcl} & \square(x,t_3,B) & = & (-\sqrt{-[\frac{1}{2},\frac{1}{\sqrt{2}}]^2+[2,2]}) \;\cap\; [\frac{\sqrt{3}}{\sqrt{2}},\frac{\sqrt{7}}{2}] \\ & & = & [-\frac{\sqrt{7}}{2},-\frac{\sqrt{3}}{\sqrt{2}}] \;\cap\; [\frac{\sqrt{3}}{\sqrt{2}},\frac{\sqrt{7}}{2}] \;=\; \emptyset \\ \text{and} & \square(x,t_4,B) & = & (\sqrt{-[\frac{1}{2},\frac{1}{\sqrt{2}}]^2+[2,2]}) \;\cap\; [\frac{\sqrt{3}}{\sqrt{2}},\frac{\sqrt{7}}{2}] \;=\; [\frac{\sqrt{3}}{\sqrt{2}},\frac{\sqrt{7}}{2}]. \end{array}$$

Again, we can illustrate this in Figure 5.1. Where t_3 *represents the points on the circle given by* $x_1^2 + x_2^2 - 2 = 0$ *but with* $x_1 \leq 0$ *and, hence, left of the* x_2*-axis,* t_4 *represents the points on this circle with* $x_1 \geq 0$ *and, therefore, right of the* x_2*-axis. As the only solution is on the latter half of the circle, we only need to consider* t_4*.*

5.4.5 Simplifying formulas with respect to variable bounds

In Definition 31, we have specified how to evaluate a formula under an interval assignment. In contrast to the evaluation under a normal assignment, the result can be an interval or a simplified formula. This is exactly what we can make use of when applying the virtual substitution. Given a real-arithmetic formula $\varphi^{\mathbb{R}}$ and an over-approximation of its solution space B, the variable elimination step as given by Theorem 6 can be reformulated to

$$\exists x.\, \varphi^{\mathbb{R}} \quad \Leftrightarrow \bigvee_{t \in \text{tcs}_{\mathbb{I}}(x,[\![\varphi^{\mathbb{R}}]\!]^B \wedge \varphi_B, \text{vb}(x,[\![\varphi^{\mathbb{R}}]\!]^B))} [\![([\![\varphi^{\mathbb{R}}]\!]^B \wedge \varphi_B)[t/\!/x] \;\wedge\; \text{sc}(t)]\!]^B \wedge \varphi_{B[(-\infty,\infty)/x]} \tag{5.4}$$

where φ_B and $\varphi_{B[(-\infty,\infty)/x]}$ are the formulas representing the variables' bounds in B and $B[(-\infty,\infty)/x]$, respectively. Note, that for the latter we explicitly remove the variable bounds of x.

Example 21 *Consider the real-arithmetic formula from Example 19*

$$\varphi \;=\; 2x_2{+}1 \geq 0 \;\wedge\; x_1{+}1 > 0 \;\wedge\; x_1^2{+}x_2^2{-}2 = 0 \;\wedge\; x_1^2{-}x_2{-}1 = 0 \;\wedge\; 6x_1{-}2x_2{-}3 \geq 0$$

which is also illustrated by Figure 5.1 and the tighter over-approximation

$$\begin{array}{rcl} B(x_1) & = & [\frac{\sqrt{3}}{\sqrt{2}},\frac{\sqrt{7}}{2}] \\ B(x_2) & = & [\frac{1}{2},\frac{1}{\sqrt{2}}] \end{array}$$

which we obtained in Example 19 using ICP. We over-approximate B in order to cast off the radical

expressions yielding

$$\begin{aligned} B(x_1) &= [\tfrac{153}{125}, \tfrac{1323}{1000}] \\ B(x_2) &= [\tfrac{1}{2}, \tfrac{177}{250}]. \end{aligned}$$

Now we replace the bound constraints

$$2x_2 + 1 \geq 0 \quad \wedge \quad x_1 + 1 > 0$$

by the bound constraints

$$125x_1 - 153 \geq 0 \quad \wedge \quad 1000x_1 - 1323 \leq 0 \quad \wedge \quad 2x_2 - 1 \geq 0 \quad \wedge \quad 250x_2 - 177 \leq 0$$

yielding

$$\begin{aligned} \varphi' = \quad & 125x_1 - 153 \geq 0 \quad \wedge \quad 1000x_1 - 1323 \leq 0 \quad \wedge \quad 2x_2 - 1 \geq 0 \\ \wedge \quad & 250x_2 - 177 \leq 0 \quad \wedge \quad x_1^2 + x_2^2 - 2 = 0 \quad \wedge \quad x_1^2 - x_2 - 1 = 0 \\ \wedge \quad & 6x_1 - 2x_2 - 3 \geq 0 \end{aligned}$$

As

$$\begin{aligned} [\![6x_1 - 2x_2 - 3 \geq 0]\!]^B &\equiv [6,6] \cdot [\tfrac{153}{125}, \tfrac{1323}{1000}] - [2,2] \cdot [\tfrac{1}{2}, \tfrac{177}{250}] - [3,3] \geq 0 \\ &\equiv ([\tfrac{3672}{500}, \tfrac{3969}{500}] - [\tfrac{500}{500}, \tfrac{708}{500}]) - [\tfrac{1500}{500}, \tfrac{1500}{500}] \geq 0 \\ &\equiv [\tfrac{1464}{500}, \tfrac{1969}{500}] \geq 0 \\ &\equiv \text{true} \end{aligned}$$

we can simplify φ' to

$$\begin{aligned} [\![\varphi']\!]^B \wedge \varphi_B = \quad & 125x_1 - 153 \geq 0 \quad \wedge \quad 1000x_1 - 1323 \leq 0 \quad \wedge \quad 2x_2 - 1 \geq 0 \\ \wedge \quad & 250x_2 - 177 \leq 0 \quad \wedge \quad x_1^2 + x_2^2 - 2 = 0 \quad \wedge \quad x_1^2 - x_2 - 1 = 0 \end{aligned}$$

Hence, we get rid of the constraint for which we would not have created any test candidates according to Theorem 6, anyway. In general, simplifying the formula does not make Theorem 6 superfluous. For instance, the constraint $x_1^2 + x_2^2 - 2 = 0$ is still part of $[\![\varphi']\!]^B \wedge \varphi_B$, but according to Theorem 6 it only provides one test candidate instead of three, i. e., $t_4 = \sqrt{-x_2^2 + 2} \in \text{tcs}_{\mathbb{I}}(x_1, [\![\varphi']\!]^B \wedge \varphi_B, \text{vb}(x_1, [\![\varphi']\!]^B \wedge \varphi_B))$. Let us compute

$$\varphi'' = [\![([\![\varphi']\!]^B \wedge \varphi_B)[t_4 // x_1] \wedge \text{sc}(t_4)]\!]^B \wedge \varphi_{B[(-\infty,\infty)/x_1]}$$

in order to see the impact of Equation (5.4)*. The side condition of t_4 is $\text{sc}(t_4) = x_2^2 - 2 \leq 0$ and*

$$\begin{aligned} ([\![\varphi']\!]^B \wedge \varphi_B)[t_4 // x_1] = \quad & 2x_2 - 1 \geq 0 \quad \wedge \quad 250x_2 - 177 \leq 0 \\ \wedge \quad & x_2^2 + x_2 - 1 = 0 \quad \wedge \quad 15625x_2^2 - 7841 \leq 0 \\ \wedge \quad & 1000000x_2^2 - 249671 \geq 0 \end{aligned}$$

As

$$\begin{array}{rrcl} & [\![15625x_2^2 - 7841 \leq 0]\!]^B & \equiv & \text{true} \\ \textit{and} & [\![1000000x_2^2 - 249671 \geq 0]\!]^B & \equiv & \text{true,} \end{array}$$

it follows that

$$\begin{array}{rcccccc} \varphi'' & = & & 2x_2 - 1 \geq 0 & \wedge & 250x_2 - 177 \leq 0 \\ & & \wedge & x_2^2 + x_2 - 1 = 0 & \wedge & x_2^2 - 2 \leq 0 \end{array}$$

Without the simplification of Equation (5.4)*, where we also would need to virtually substitute* x_1 *by* t_4 *in* $6x_1 - 2x_2 - 3 \geq 0$*, we would obtain the result*

$$\begin{array}{llllll} & & 2x_2 - 1 \geq 0 & \wedge & 250x_2 - 177 \leq 0 & \\ \wedge & & x_2^2 + x_2 - 1 = 0 & \wedge & x_2^2 - 2 \leq 0 & \\ \wedge & & 15625x_2^2 - 7841 \leq 0 & \wedge & 1000000x_2^2 - 249671 \geq 0 & \\ \wedge & (& (x_2 < 0 & \wedge & x_2^2 + 1 \geq 0) & \\ & \vee & x_2^2 - 1 \leq 0 & & & \\ & \vee & (x_2^2 - 2 = 0 & \wedge & x_2 = 0) &) \end{array}$$

Hence, this simplification saves us from considering a further seven constraints and prevents us from having to take a case splitting into account, as there is a disjunction involved in the non-simplified formula.

CHAPTER 6

Experimental Results for Real Arithmetic

We have experimentally evaluated the contributions of Chapter 4 and Chapter 5 using our toolbox `SMT-RAT`, which we introduced in Chapter 3.

6.1 Benchmark sets

In this section we describe the benchmark sets, which we used in our experimental results. We also specify the number of variables, the maximum degree and the Boolean complexity of an average instance for each benchmark set. The *Boolean complexity* of a given formula is higher the more solutions the formula's Boolean abstraction has, where we say that a conjunction of constraints has *no Boolean complexity*.

We have used five of the seven benchmark sets for quantifier-free nonlinear real arithmetic from the SMT competition 2016.

Hong: These are 20 crafted and dimension dependent examples as they were used in [Hon91]. The nth example ($1 \leq n \leq 20$) has the form

$$\sum_{i=1}^{n} x_i^2 < 1 \ \wedge \ \prod_{i=1}^{n} x_i > 1.$$

An example has therefore n variables, a maximum degree of n (because of the second constraint) and no Boolean complexity.

Hycomp: These are 2102 instances generated by the model checker `HyComp` [CGMT15] for nonlinear hybrid automata using the quantifier-free SMT encoding as presented in [CMT12]. On average, an example has 44 variables, a maximum degree of 3 and almost no Boolean complexity.

Kissing: These are 45 crafted and dimension dependent examples of the *kissing number problem*. Here we seek an arrangement of n non-overlapping unit spheres such that each sphere touches another given sphere at one point only. The average number of variables per example is 36 and each example has the maximum degree 2. The examples have no Boolean complexity.

MetiTarski: These are 7713 examples, which are proof obligations generated by automatic the theorem prover `Meti-Tarski` for real-valued special functions [AP10]. The number of variables averages 3 and the maximum degree is on average 3, but measures 44 for some examples. Almost all examples have no Boolean complexity.

Zankl: These are 166 examples, mostly generated by the termination analysis of term rewriting systems [FGM+07]. The average number of variables in the examples is 81, so it is relatively high, and the average maximum degree is 3. All examples have no Boolean complexity.

For the two benchmark sets of the SMT competition 2016, which we did not use, none of the presented techniques of this thesis can solve any instance. Therefore, we did not consider them for a comparison.

Additionally, we tested on the following benchmark sets.

Bounce: These 180 examples describe whether a thrown ball (starting at an initial height and moving with a decreasing vertical motion along some direction), which is allowed to bounce n times, eventually falls into a hole in the ground at a given distance. The average number of variables in the examples is 35. All examples have the maximum degree 3 and a relatively high Boolean complexity.

Keymaera: These are 421 examples, which were mostly generated by the verification tool `KeYmaera` for hybrid systems in the context of [PQR09]. On average, the number of variables in an example is 9 and the maximal degree is 2. The examples have a relatively low Boolean complexity.

Rect: These 91 examples describe whether we can fit a given set of rectangles within a certain rectangular area. The average number of variables in the examples is 15 and the maximum degree of each example is 2. All examples have a high Boolean complexity.

Witness: These 100 examples were generated in a formal verification process as described in [RS15]. The average number of variables and average maximum degree are both 3. The examples have no Boolean complexity.

6.2 Settings

The experiments, which we report on in this chapter, are structured as follows. First we evaluate the ideas from Chapter 4 and Chapter 5 in order to work out how much they contribute on different

benchmark sets. For this purpose we always use a simple SMT-RAT strategy consisting only of a $\texttt{Module}_{SAT}$ with a $\texttt{Module}_{\text{VS}}$ as backend. This represents an SMT solver with a theory solver based on the virtual substitution as presented in Chapter 4 and equipped with the optimizations of Chapter 5. Afterwards, we compare different strategies with and without a $\texttt{Module}_{\text{VS}}$ for the purpose of seeing how well it contributes to a better performance of a strategic combination of procedures for real arithmetic. Finally, we compare these strategies to state-of-the-art tools that can be used to check the satisfiability of a real arithmetic formula.

All experiments were carried out on AMD Opteron 6172 processors. Every solver was allowed to use up to 4 GB of memory and 200 seconds of wall clock time.

6.3 An SMT-compliant theory solver based on the virtual substitution

First, we evaluate the ideas from Chapter 4, which adapt the virtual substitution for a better performing integration into an SMT solver. Here, we test the $\texttt{Module}_{\text{VS}}$

1. supporting incrementality and backtracking (instead of starting each theory call from scratch) and

2. generating an infeasible subset if a conjunction of constraints is found to be unsatisfiable (instead of simply returning the infeasible subset containing all of these constraints).

We test one setting, where the $\texttt{Module}_{\text{VS}}$ supports: neither of the two features ($\texttt{Module}_{\text{VS}}^{\cancel{\text{SMT}}}$), only the first ($\texttt{Module}_{\text{VS}}^{\text{Inc}}$), only the second ($\texttt{Module}_{\text{VS}}^{\text{IS}}$) and both features ($\texttt{Module}_{\text{VS}}^{\text{SMT}}$). We compare these settings by employing each of them in a simple SMT-RAT strategy, consisting of a $\texttt{Module}_{SAT}$ and a $\texttt{Module}_{\text{VS}}$ as illustrated in Figure 6.1. We only use the two benchmark sets BOUNCE and RECT, which provide a suitable Boolean complexity in order to illustrate the use of these SMT related features.

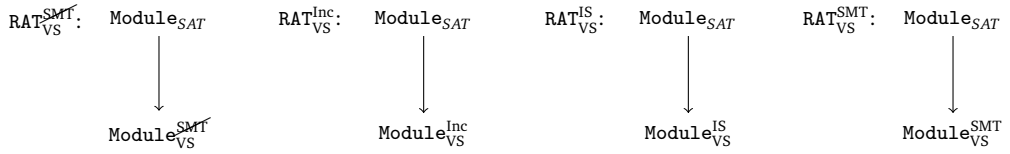


Figure 6.1: Four SMT-RAT strategies, which combine a $\texttt{Module}_{SAT}$ with a $\texttt{Module}_{\text{VS}}$ of different settings as backend

Table 6.1 displays the experimental results for this comparison. The first column shows how the theory solver based on the virtual substitution, as presented in Chapter 4, performs neither supporting incrementality, backtracking nor infeasible subset generation. In the second column, where just incrementality and backtracking are supported, we only gain 7 instances for the benchmark BOUNCE, however, we win 18 satisfiable and lose 11 unsatisfiable examples. Interestingly,

Solvers→ Benchmarks↓	$\text{RAT}_{\text{VS}}^{\not\text{SMT}}$ #	time	$\text{RAT}_{\text{VS}}^{\text{Inc}}$ #	time	$\text{RAT}_{\text{VS}}^{\text{IS}}$ #	time	$\text{RAT}_{\text{VS}}^{\text{SMT}}$ #	time
BOUNCE (180)	147	374.6	154	763.0	161	1811.9	**165**	**809.7**
- sat	105	360.0	123	757.2	119	1797.2	**123**	**750.0**
- unsat	**42**	**14.5**	31	5.8	42	14.7	42	59.8
- unkn.	0	0.0	**26**	**288.2**	0	0.0	15	297.3
RECT (91)	20	282.0	20	281.2	26	407.2	**28**	**758.4**
- sat	14	79.1	14	90.2	19	381.3	**21**	**733.8**
- unsat	6	202.9	6	190.9	7	25.9	**7**	**24.6**
- unkn.	0	0.0	0	0.0	0	0.0	0	0.0
ALL (271)	167	656.6	174	1044.2	187	2219.1	**193**	**1568.1**
- sat	119	439.2	137	847.4	138	2178.5	**144**	**1483.8**
- unsat	48	217.4	37	196.7	**49**	**40.6**	49	84.3
- unkn.	0	0.0	**26**	**288.2**	0	0.0	15	297.3

Table 6.1: Comparison of the four SMT-RAT strategies of Figure 6.1 (the column # contains the number of solved instances and the column *time* contains the amount of seconds needed for solving these instances).

none of the examples in BOUNCE lead to a timeout with the additional features. Instead, we end up in a case where the virtual substitution cannot be applied and `unknown` is returned for 26 examples. This is due to the fact that incrementality can influence the solving process. For instance, if we have to choose the next variable to eliminate or next constraint to create test candidates for, there might be several choices of the same quality according to our heuristics. Let us assume that in one theory call we choose the variable x for elimination for a given vertex in the VSST of our theory solver. In the next theory call, if we consider the same vertex, the variable y has now an identical quality according to our heuristics for a variable elimination, so we could choose either x or y. In an incremental call, where x has already been chosen, we do not change this decision in order to reuse results gained already. If incrementality is not supported, we start from scratch and, in this case, we might choose y instead. If we are unlucky, as for many instances of BOUNCE, this leads to a worse performance or the case, where the virtual substitution cannot determine the satisfiability of the given conjunction of constraints. More thorough heuristics could avoid this behavior.

In the third column of Table 6.1, we only support infeasible subset generation. Compared to the first column, this feature clearly improves the performance, as we can solve a further 12% of the examples. For BOUNCE, the infeasible subsets are on average 45% smaller than just returning all checked constraints and for RECT even 50% smaller. In the last column, the results were achieved with an entirely SMT-compliant setting of the $\texttt{Module}_{\text{VS}}$ gaining us almost 16% solved examples in comparison to using a non-SMT-compliant setting.

6.4 Choice of the elimination variable and constraint to provide test candidates for

The choice of the next variable to eliminate, as well as the choice of the next constraint to create test candidates for, have a significant influence on the performance of an implementation of the

Benchmarks↓	Solvers→	$Opt_{avg} + CV_{RDV}$ #	$Opt_{avg} + CV_{RDV}$ time	$Opt_{avg} + CV_{RD}$ #	$Opt_{avg} + CV_{RD}$ time	$Opt_{avg} + CV_{DRV}$ #	$Opt_{avg} + CV_{DRV}$ time	$Opt_{avg} + CV_{DR}$ #	$Opt_{avg} + CV_{DR}$ time
BOUNCE	sat	108	865.0	**108**	**862.0**	108	941.3	108	925.8
(180)	unsat	**17**	**0.9**	**17**	**0.9**	17	1.0	17	1.0
	unkn.	29	759.9	29	829.9	29	969.2	29	954.1
HONG	sat	0	0.0	0	0.0	0	0.0	0	0.0
(20)	unsat	1	0.1	1	0.2	1	0.1	**1**	**< 0.1**
	unkn.	19	3.9	19	6.3	19	4.5	19	5.4
HYCOMP	sat	0	0.0	0	0.0	0	0.0	0	0.0
(2102)	unsat	1398	5975.1	1415	7390.0	1435	6773.3	**1464**	**7506.6**
	unkn.	0	0.0	0	0.0	0	0.0	0	0.0
KEYMAERA	sat	0	0.0	0	0.0	0	0.0	0	0.0
(421)	unsat	381	19.0	381	23.3	381	18.3	**381**	**16.3**
	unkn.	14	165.1	14	174.3	13	5.9	12	5.5
KISSING	sat	7	36.1	7	46.3	**7**	**27.0**	7	53.5
(45)	unsat	0	0.0	0	0.0	0	0.0	0	0.0
	unkn.	0	0.0	0	0.0	0	0.0	0	0.0
METITARSKI	sat	3709	659.2	3709	794.9	**3716**	**619.8**	3713	612.4
(7713)	unsat	1511	70.5	1508	119.5	**1527**	**76.4**	1521	70.5
	unkn.	2493	792.1	2496	900.7	2470	744.8	2479	741.5
RECT	sat	20	1089.6	**20**	**617.5**	19	515.2	18	477.7
(91)	unsat	7	36.3	**7**	**35.8**	7	36.7	**7**	**35.8**
	unkn.	0	0.0	0	0.0	0	0.0	0	0.0
WITNESS	sat	**28**	**15.1**	**28**	**15.1**	28	15.6	28	15.2
(100)	unsat	0	0.0	0	0.0	0	0.0	0	0.0
	unkn.	72	2086.4	72	2084.0	72	2093.0	72	2089.4
ZANKL	sat	19	225.7	**20**	**485.3**	19	169.3	19	371.6
(166)	unsat	**9**	**8.9**	9	13.5	8	1.8	8	0.8
	unkn.	6	11.0	6	10.4	8	85.4	8	82.9
ALL		7215	9001.4	7230	10404.5	7273	9195.7	**7292**	**10087.3**
(10838)	sat	3891	2890.7	3892	2821.2	**3897**	**2288.2**	3893	2456.3
	unsat	3324	6110.7	3338	7583.3	3376	6907.5	**3399**	**7631.0**
	unkn.	2633	3818.3	2636	4005.7	2611	3902.8	2619	3878.7

Table 6.2: Comparison of four different constraint valuations for the variable valuation Opt_{avg} (the column # contains the number of solved instances and the column *time* contains the amount of seconds needed for solving these instances).

virtual substitution as introduced in Chapter 4. We can combine the constraint valuations as introduced in Section 5.1.1 in various ways and, moreover, use each of these combinations with the strategies for choosing the next variable to eliminate as introduced in Section 5.1.2. For the constraint valuations we confine ourselves to the four combinations, which we introduced at the end of Section 5.1.1. Hence, given a constraint and an elimination variable, which occurs in this constraint, we use the constraint valuations:

1. $CV_{RDV} = (\omega_{fms}, \omega_{thd}, \omega_{rel}, \omega_{evd}, \omega_{nv})$ (rate the constraint's relation symbol higher than the elimination variable's degree and take the number of variables in the constraint into account)

2. $CV_{RD} = (\omega_{fms}, \omega_{thd}, \omega_{rel}, \omega_{evd})$ (rate the constraint's relation symbol higher than the elimination variable's degree)

3. $CV_{DRV} = (\omega_{fms}, \omega_{thd}, \omega_{evd}, \omega_{rel}, \omega_{nv})$ (rate the elimination variable's degree higher than the constraint's relation symbol and take the number of variables in the constraint into account)

Benchmarks↓	Solvers→	$Opt_{worst} + CV_{RDV}$ #	$Opt_{worst} + CV_{RDV}$ time	$Opt_{worst} + CV_{RD}$ #	$Opt_{worst} + CV_{RD}$ time	$Opt_{worst} + CV_{DRV}$ #	$Opt_{worst} + CV_{DRV}$ time	$Opt_{worst} + CV_{DR}$ #	$Opt_{worst} + CV_{DR}$ time
BOUNCE	sat	105	1481.7	108	1763.7	119	742.5	**123**	**717.7**
(180)	unsat	17	0.8	17	0.9	**27**	**71.8**	9	0.4
	unkn.	29	793.4	29	1212.4	30	726.4	48	505.0
HONG	sat	0	0.0	0	0.0	0	0.0	0	0.0
(20)	unsat	**1**	**< 0.1**	**1**	**< 0.1**	**1**	**< 0.1**	**1**	**< 0.1**
	unkn.	19	5.0	19	6.1	19	5.3	19	4.7
HYCOMP	sat	0	0.0	0	0.0	0	0.0	0	0.0
(2102)	unsat	1531	9341.2	1584	10028.4	**1595**	**10683.2**	1543	8716.7
	unkn.	0	0.0	0	0.0	0	0.0	0	0.0
KEYMAERA	sat	0	0.0	0	0.0	0	0.0	0	0.0
(421)	unsat	379	16.0	380	18.7	405	57.0	**406**	**74.1**
	unkn.	18	321.2	17	24.8	15	284.3	15	108.9
KISSING	sat	**8**	**168.9**	8	170.6	8	173.7	8	190.6
(45)	unsat	0	0.0	0	0.0	0	0.0	0	0.0
	unkn.	0	0.0	0	0.0	0	0.0	0	0.0
METITARSKI	sat	3797	558.8	3797	580.8	**3807**	**552.5**	3805	551.6
(7713)	unsat	1519	60.8	1520	71.1	1522	60.9	**1525**	**59.7**
	unkn.	2397	653.9	2396	670.3	2384	648.8	2383	643.0
RECT	sat	23	794.1	21	483.2	**24**	**997.8**	20	261.1
(91)	unsat	8	211.6	9	374.2	7	25.3	**9**	**360.9**
	unkn.	0	0.0	0	0.0	0	0.0	0	0.0
WITNESS	sat	**28**	**15.3**	28	15.4	28	15.4	28	15.4
(100)	unsat	0	0.0	0	0.0	0	0.0	0	0.0
	unkn.	72	2084.9	72	2093.2	72	2106.6	72	2101.6
ZANKL	sat	23	540.9	**25**	**240.8**	20	199.6	24	300.4
(166)	unsat	13	34.6	14	154.1	11	4.5	**15**	**259.2**
	unkn.	7	228.4	5	1.1	7	108.4	6	100.9
ALL		7452	13224.8	7512	13902.1	**7574**	**13584.3**	7516	11507.8
(10838)	sat	3984	3559.7	3987	3254.6	4006	2681.6	**4008**	**2036.7**
	unsat	3468	9665.1	3525	10647.5	**3568**	**10902.7**	3508	9471.2
	unkn.	2542	4086.8	2538	4007.9	2527	3879.7	2543	3464.1

Table 6.3: Comparison of four different constraint valuations for the variable valuation Opt_{worst} (the column # contains the number of solved instances and the column *time* contains the amount of seconds needed for solving these instances).

4. $CV_{DR} = (\omega_{fms}, \omega_{thd}, \omega_{evd}, \omega_{rel})$ (rate the elimination variable's degree higher than the constraint's relation symbol)

We combine these four constraint valuations with the three heuristics for choosing the next variable to eliminate. For a given set of constraints (considered by a vertex in the VSST of the theory solver) they optimize

1. the average constraint valuation (Opt_{avg}),
2. the worst constraint valuation (Opt_{worst}) and
3. the best constraint valuation (Opt_{best}).

Therefore, we tested 12 different settings for a $\texttt{Module}_{VS}$ in an `SMT-RAT` strategy, which again only combine a $\texttt{Module}_{SAT}$ with a $\texttt{Module}_{VS}$ as backend. We tested on all benchmark sets[1] and

[1]We do not show the results for the benchmark sets HONG and WITNESS, as all settings performed equally on these examples (1 unsatisfiable instance of HONG in less than 0.1 seconds and 28 satisfiable instances of WITNESS in $\sim$ 15 seconds).

Benchmarks↓	Solvers→	$Opt_{best} + CV_{RDV}$ #	$Opt_{best} + CV_{RDV}$ time	$Opt_{best} + CV_{RD}$ #	$Opt_{best} + CV_{RD}$ time	$Opt_{best} + CV_{DRV}$ #	$Opt_{best} + CV_{DRV}$ time	$Opt_{best} + CV_{DR}$ #	$Opt_{best} + CV_{DR}$ time
BOUNCE	sat	119	324.9	**123**	**813.6**	103	1574.8	123	1001.9
(180)	unsat	24	2.5	**42**	**57.0**	9	0.4	31	7.5
	unkn.	33	241.5	15	301.9	33	1651.5	26	629.6
HONG	sat	0	0.0	0	0.0	0	0.0	0	0.0
(20)	unsat	1	0.1	**1**	**< 0.1**	**1**	**< 0.1**	**1**	**< 0.1**
	unkn.	19	4.2	19	1.4	19	7.9	19	4.3
HYCOMP	sat	0	0.0	0	0.0	0	0.0	0	0.0
(2102)	unsat	1518	7219.7	**1813**	**6461.0**	1682	5388.7	1812	8333.4
	unkn.	0	0.0	0	0.0	0	0.0	0	0.0
KEYMAERA	sat	0	0.0	0	0.0	0	0.0	0	0.0
(421)	unsat	399	110.6	399	50.9	374	22.3	**401**	**31.7**
	unkn.	19	78.7	21	328.4	28	190.6	20	139.3
KISSING	sat	7	35.8	7	56.8	**7**	**28.1**	7	54.7
(45)	unsat	0	0.0	0	0.0	0	0.0	0	0.0
	unkn.	1	114.7	1	49.3	1	87.2	1	69.7
METITARSKI	sat	**3831**	**635.2**	3819	509.3	3792	620.9	3808	582.2
(7713)	unsat	**1467**	**69.9**	1465	22.8	1458	68.9	1465	60.3
	unkn.	2415	498.2	2429	422.2	2463	479.9	2440	454.0
RECT	sat	17	248.2	15	221.8	**18**	**471.1**	17	394.7
(91)	unsat	**8**	**213.5**	**7**	**31.5**	7	23.3	7	23.0
	unkn.	0	0.0	0	0.0	0	0.0	0	0.0
WITNESS	sat	**28**	**15.3**	28	14.8	28	15.4	28	15.7
(100)	unsat	0	0.0	0	0.0	0	0.0	0	0.0
	unkn.	72	2092.7	72	2073.3	72	2107.1	72	2110.0
ZANKL	sat	**19**	**364.1**	18	237.4	18	202.0	18	380.3
(166)	unsat	**10**	**58.3**	6	1.7	8	61.7	7	18.2
	unkn.	7	60.9	8	78.7	6	14.9	7	45.5
ALL		7448	9298.1	**7743**	**8441.7**	7505	8477.6	7725	10903.7
(10838)	sat	**4021**	**1623.6**	4010	1853.7	3966	2912.3	4001	2429.6
	unsat	3427	7674.5	**3733**	**6588.0**	3539	5565.3	3724	8474.2
	unkn.	2566	3090.9	2565	3255.3	2622	4539.2	2585	3452.3

Table 6.4: Comparison of four different constraint valuations for the variable valuation Opt_{best} (the column # contains the number of solved instances and the column *time* contains the amount of seconds needed for solving these instances).

the results are presented in Table 6.2 ($Opt_{avg} + [CV_{RDV}, CV_{RD}, CV_{DRV}, CV_{DR}]$), Table 6.3 ($Opt_{worst} + [CV_{RDV}, CV_{RD}, CV_{DRV}, CV_{DR}]$) and Table 6.4 ($Opt_{best} + [CV_{RDV}, CV_{RD}, CV_{DRV}, CV_{DR}]$).

Summarizing all results, Opt_{avg} does not seem to be a good choice. It is not superior to Opt_{worst} or Opt_{best} in any the benchmark sets, apart from the unsatisfiable instances of METITARSKI. Here, it can solve, in combination with the constraint valuation CV_{DRV}, 2 more instances than the second best combination for these instances, $Opt_{worst} + CV_{DR}$. Considering the best results of Opt_{worst} and Opt_{best}, Opt_{best} solves 7758 examples where Opt_{worst} solves 7574. If we have a look at the single benchmark sets, Opt_{best} performs especially well on HYCOMP if combined with CV_{RD} or CV_{DR}. Here it solves over 200 examples more than the best setting with Opt_{worst}. However, on all the other benchmark sets, apart from BOUNCE and the satisfiable instances of METITARSKI, Opt_{worst} is better for at least one setting.

We can also observe a significant impact of the choice of constraint valuation. Comparing, for instance, Opt_{best} in combination with CV_{RDV} or CV_{RD} shows a difference of 302 solved instances just for HYCOMP. The disparity between the combinations $Opt_{best} + CV_{RD}$ and $Opt_{best} + CV_{DRV}$ are also considerable. The same holds, if we use Opt_{worst} as heuristics for choosing the next variable

to eliminate. Here, using CV_{DRV} solves more satisfiable instances of BOUNCE than using CV_{DR}, but at the same time fewer unsatisfiable instances. Moreover, using CV_{DRV} or CV_{DR} instead of CV_{RDV} or CV_{RD} solves 405/406 in contrast to 379/380 instances of KEYMAERA.

Another interesting fact is that there is no setting, which is superior in the majority of the benchmark sets. The best candidate varies not only when changing the benchmark set but also if we only consider the satisfiable or unsatisfiable instances, respectively.

The experimental results confirm that settings, which involve Opt_{worst} instead of Opt_{best} tend to omit ending up in a case, where the virtual substitution cannot detect the satisfiability. This makes sense, as we try to avoid vertices in the VSST with constraints which are worse according to our heuristics. A setting with Opt_{best}, though, aims at finding a solution and, indeed, it solves more satisfiable instances than with Opt_{worst}. However, the difference is marginal.

Two of the settings with Opt_{best} solve many more unsatisfiable instances than any other setting, which mainly relies on the good performance on the benchmark set HYCOMP. For these instances it is important in which order we eliminate the variables and, if we use Opt_{best} in combination with a constraint valuation which does not take the number of variables in the constraints into account, we achieve a better variable elimination order in this specific case. For all further experiments we use $Opt_{best} + CV_{RD}$.

6.5 Backjumping, local conflict detection and exploiting variable bounds

Table 6.5 shows the results of the evaluation of the last three contributions from Chapter 5, which aim to improve the performance of the virtual substitution in SMT. For all results in this table we used an `SMT-RAT`-based SMT solver combining a $\texttt{Module}_{SAT}$ with a theory solving module $\texttt{Module}_{VS}$, as introduced in Chapter 4. Additionally, we enabled backjumping (Section 5.2), local conflict detection (Section 5.3) and variable bounds exploitation (Section 5.4) in the columns 2, 3 and 4, respectively. This means that the difference between the solvers which produced the results in the first and the second (third, fourth) column is only the additional usage of backjumping (local conflict detection, variable bounds exploitation).

The results indicate that backjumping has almost no effect. It slightly improves the overall performance and, hence, we can solve one more instance. Backjumping takes a minor short cut as it detects unsatisfiability before checking all test candidates. However, for the given examples and chosen heuristics, it only comes into use for less than one percent of the instances of our benchmark sets.

Local conflict detection introduces some overhead but, compared to backjumping, it can detect conflicting vertices more often and in most cases earlier. The test results show that due to the overhead we run into timeout for some satisfiable instances of METITARSKI and one unsatisfiable instance of KEYMAERA where, without backjumping, we can solve these instances. However it enables us to solve a further 59 unsatisfiable instances of METITARSKI.

Benchmarks↓	Solvers→	plain #	plain time	with backjumping #	with backjumping time	with local conflict #	with local conflict time	with variable bounds #	with variable bounds time
BOUNCE	sat	123	813.6	**123**	**811.6**	123	812.6	122	1598.1
(180)	unsat	42	57.0	42	56.9	**42**	**56.6**	37	15.7
	unkn.	15	301.9	15	300.9	15	306.5	20	669.1
HONG	sat	0	0.0	0	0.0	0	0.0	0	0.0
(20)	unsat	**1**	**< 0.1**	**1**	**< 0.1**	**1**	**< 0.1**	**1**	**< 0.1**
	unkn.	19	1.4	19	1.1	19	1.3	19	1.2
HYCOMP	sat	0	0.0	0	0.0	0	0.0	0	0.0
(2102)	unsat	1813	6461.0	**1813**	**6453.0**	1813	6685.1	1805	7025.3
	unkn.	0	0.0	0	0.0	0	0.0	0	0.0
KEYMAERA	sat	0	0.0	0	0.0	0	0.0	0	0.0
(421)	unsat	399	14.0	399	14.0	398	11.7	**403**	**113.2**
	unkn.	21	328.4	21	326.5	20	326.9	18	280.5
KISSING	sat	7	56.8	**7**	**56.7**	7	57.6	7	69.1
(45)	unsat	0	0.0	0	0.0	0	0.0	0	0.0
	unkn.	1	49.3	1	49.3	1	49.3	1	104.6
METITARSKI	sat	3819	509.3	3819	509.4	3815	519.4	**3906**	**453.5**
(7713)	unsat	1465	22.8	1465	22.7	1524	23.6	**1776**	**24.9**
	unkn.	2429	422.2	2429	422.7	2374	393.4	2031	379.2
RECT	sat	15	221.8	16	421.4	16	572.2	**17**	**316.2**
(91)	unsat	7	31.5	7	31.4	**7**	**29.2**	7	34.5
	unkn.	0	0.0	0	0.0	0	0.0	0	0.0
WITNESS	sat	28	14.8	28	14.8	28	14.8	**28**	**10.0**
(100)	unsat	0	0.0	0	0.0	0	0.0	0	0.0
	unkn.	72	2073.3	72	2068.4	72	2074.9	72	2407.0
ZANKL	sat	18	237.4	18	236.6	18	234.4	**22**	**218.2**
(166)	unsat	6	1.7	6	1.7	6	0.9	**16**	**3.5**
	unkn.	8	78.7	8	78.3	8	92.3	6	106.4
all		7743	8441.7	7744	8630.1	7798	9017.9	**8147**	**9882.4**
(10838)	sat	4010	1853.7	4011	2050.4	4007	2210.9	**4102**	**2665.2**
	unsat	3733	6588.0	3733	6579.7	3791	6807.0	**4045**	**7217.2**
	unkn.	2565	3255.3	2565	3247.4	2509	3244.6	2167	3948.0

Table 6.5: Results of the evaluation of additionally using the backjumping, local conflict detection and the employment of variable bounds (the column # contains the number of solved instances and the column *time* contains the amount of seconds needed for solving these instances).

The utilization of variable bounds as explained in Section 5.4 is a cheap filter and significantly narrows down the set of test candidates. We are able to solve a further 404 instances, which is clearly a considerable improvement. It only slightly worsens the performance for the benchmark sets HYCOMP and BOUNCE, due to an unlucky influence on the variable elimination order.

6.6 Comparison of SMT-RAT strategies with state-of-the-art tools

We evaluated the three SMT-RAT strategies on the left of Figure 6.2. These strategies do not involve a $\texttt{Module}_{\text{CAD}}$. We compare them with Redlog[2], for which we also disabled the use of the cylindrical algebraic decomposition method (by means of "`off rlqefb;`"). In constrast to the previously used SMT-RAT strategies in this chapter, the first SMT-RAT strategy of Figure 6.2 uses a $\texttt{Module}_{Simplex}$ before a $\texttt{Module}_{\text{VS}}$ is invoked. It only detects whether the linear constraints of a theory call already form a conflict and occasionally finds a solution for the linear constraints,

[2](from `http://svn.code.sf.net/p/reduce-algebra`, Revision 3758)

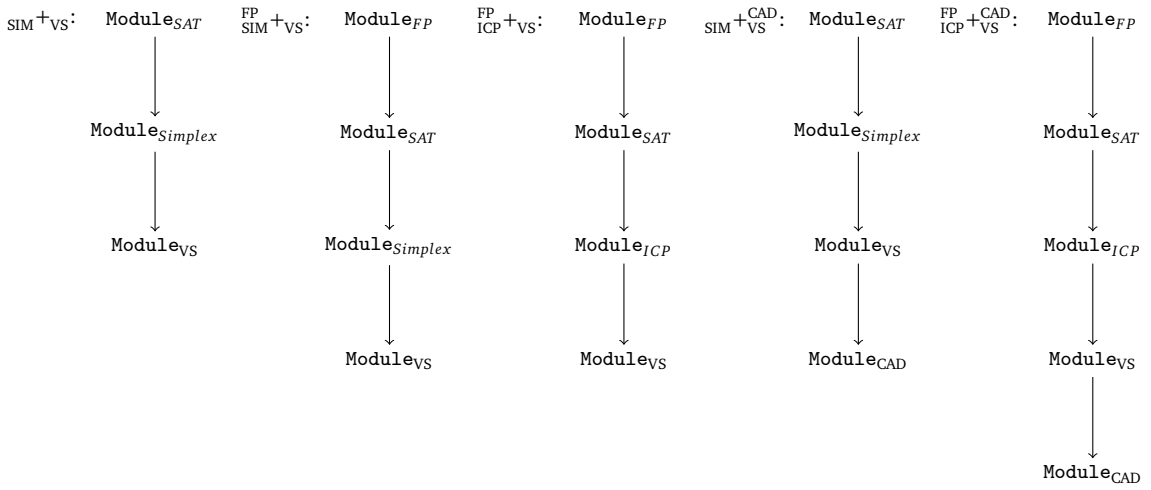


Figure 6.2: SMT-RAT strategies, which combine the modules $\texttt{Module}_{FP}$, $\texttt{Module}_{SAT}$, $\texttt{Module}_{Simplex}$, $\texttt{Module}_{ICP}$ and $\texttt{Module}_{\text{CAD}}$ with the $\texttt{Module}_{\text{VS}}$ in different ways.

which also satisfies the nonlinear constraints. The second strategy uses a preceding $\texttt{Module}_{FP}$, which tries to simplify the input SMT formula. The third strategy only differs from the second one, in that it uses a $\texttt{Module}_{ICP}$ instead of a $\texttt{Module}_{Simplex}$. This module can detect unsatisfiability in many cases. Moreover, it refines the variables' lower and upper bounds, which in turn can be exploited by a $\texttt{Module}_{\text{VS}}$, as explained in Section 5.4.

The results are depicted in Table 6.6. Comparing the first and the last column, we can see that an SMT solver using a theory solver as presented in this thesis, does not only solve more examples with high Boolean complexity (BOUNCE and RECT), but also performs better on most of the other benchmark sets. Altogether, this strategy solves more than 1000 additional instances if compared with Redlog (where the cylindrical algebraic decomposition method is disabled) and this is achieved in a fifth of the time Redlog requires. Nonetheless, Redlog solves more unsatisfiable examples, especially for the benchmark sets HYCOMP and ZANKL. One explanation for this is that the polynomial factorization, which Redlog uses, can factorize polynomials which cannot be factorized with the factorization of GiNaC [BFK02], on which SMT-RAT relies. In case we can factorize a polynomial in a constraint, for which we want to create test candidates, we construct them by the use of the zeros of the single factors (modulo multiplicity). This way the virtual substitution can also cope with constraints, in which the variable to eliminate has a degree that is higher than 2. Note that by the additional use of a $\texttt{Module}_{Simplex}$ we solve 150 examples more than without the use of a $\texttt{Module}_{Simplex}$. These are mostly those examples, for which a solution of the linear constraints also satisfies the nonlinear ones. It also improves the performance on RECT. In fact, without the use of this module our approach performs less well than Redlog. This is rather surprising, as this benchmark set has a high Boolean complexity. Having a closer look at these examples, they are a conjunction of one nonlinear inequality and some clauses, in which all constraints are linear. Redlog seems to handle the nonlinear constraint first, that is it eliminates its two variables first, resulting in a formula that can be drastically simplified.

Benchmarks↓	Solvers→	$_{\text{SIM}}{}^{+}{}_{\text{VS}}$ #	$_{\text{SIM}}{}^{+}{}_{\text{VS}}$ time	$^{\text{FP}}_{\text{SIM}}{}^{+}{}_{\text{VS}}$ #	$^{\text{FP}}_{\text{SIM}}{}^{+}{}_{\text{VS}}$ time	$^{\text{FP}}_{\text{ICP}}{}^{+}{}_{\text{VS}}$ #	$^{\text{FP}}_{\text{ICP}}{}^{+}{}_{\text{VS}}$ time	Redlog$_{\not\text{CAD}}$ #	Redlog$_{\not\text{CAD}}$ time
BOUNCE	sat	123	1042.9	**123**	**543.1**	91	469.7	117	2139.5
(180)	unsat	57	158.4	57	82.9	**57**	**22.3**	56	1056.6
	unkn.	0	0.0	0	0.0	0	0.0	–	–
HONG	sat	0	0.0	0	0.0	0	0.0	0	0.0
(20)	unsat	1	< 0.1	1	0.1	**20**	**11.1**	1	0.5
	unkn.	19	1.3	19	7.8	0	0.0	–	–
HYCOMP	sat	0	0.0	0	0.0	0	0.0	0	0.0
(2102)	unsat	1814	8781.5	1587	9275.1	1634	9990.9	**1958**	**28267.0**
	unkn.	0	0.0	0	0.0	0	0.0	–	–
KEYMAERA	sat	0	0.0	0	0.0	0	0.0	0	0.0
(421)	unsat	410	62.6	409	27.5	410	26.6	**415**	**216.9**
	unkn.	10	8.5	9	2.5	0	0.0	–	–
KISSING	sat	7	73.7	7	74.0	**13**	**79.7**	4	2.2
(45)	unsat	0	0.0	0	0.0	0	0.0	0	0.0
	unkn.	1	113.9	1	111.7	0	0.0	–	–
METITARSKI	sat	4187	393.8	4041	989.0	**4620**	**5932.9**	2958	11770.8
(7713)	unsat	1887	30.0	1927	107.0	**2368**	**293.7**	1922	7583.4
	unkn.	1636	242.9	1744	384.5	356	3060.0	–	–
RECT	sat	**26**	**336.7**	22	468.8	16	806.3	21	307.7
(91)	unsat	**14**	**368.2**	11	277.9	7	31.1	12	829.8
	unkn.	0	0.0	0	0.0	2	10.9	–	–
WITNESS	sat	28	10.3	28	11.2	**66**	**2078.8**	5	69.4
(100)	unsat	0	0.0	0	0.0	**15**	**1057.6**	0	0.0
	unkn.	72	2877.3	72	2880.4	0	0.0	–	–
ZANKL	sat	26	244.9	25	167.3	24	145.2	**66**	**816.1**
(166)	unsat	17	21.5	19	7.2	19	6.9	**44**	**385.1**
	unkn.	4	1.1	4	0.5	3	0.1	–	–
all		8597	11524.6	8257	12031.1	**9360**	**20952.9**	7579	53444.9
(10838)	sat	4397	2102.3	4246	2253.4	**4830**	**9512.7**	3171	15105.7
	unsat	4200	9422.2	4011	9777.7	**4530**	**11440.2**	4408	38339.2
	unkn.	1742	3245.0	1849	3387.4	361	3071.0	–	–

Table 6.6: Comparison of the first three SMT-RAT strategies from Figure 6.2, which do not involve the CAD, with Redlog when disabling that it uses the CAD (the column # contains the number of solved instances and the column *time* contains the amount of seconds needed for solving these instances).

As it can be seen in the second column, the preprocessing of the Module$_{FP}$ has a negative effect on the virtual substitution, in particular if we consider the benchmark set HYCOMP. As we have seen in the experimental results for the comparison of the heuristics of the virtual substitution, this benchmark set is specifically sensitive for minor changes of the variable elimination order and the preprocessing seems to trigger a disadvantageous one. For the benchmark set METITARSKI, however, we can solve more unsatisfiable instances with preprocessing, which mainly relies on the fact that the preprocessing itself can already solve many instances. On the other hand, we lose a lot of satisfiable instance for this set.

The use of a Module$_{ICP}$, as with the third strategy, solves many additional unsatisfiable examples. In the benchmark set HONG, for instance, we do not even need to involve the virtual substitution. Nonetheless, the collaboration of interval constraint propagation with the virtual substitution that exploits variable bounds is a fruitful one. For the benchmark sets KISSING, METITARSKI and WITNESS, we can solve considerably more examples. In particular, examples from METITARSKI and WITNESS often specify upper and lower bounds for the variables, which they

involve. This increases the chances that ICP can effectively refine these bounds. Note that by disabling the techniques that exploit the variable bounds in the virtual substitution we can solve 103 examples less.

Benchmarks↓	Solvers→	$_{SIM}^{CAD}$ #	$_{SIM}^{CAD}$ time	$_{ICP}^{FP}$ #	$_{ICP}^{FP}$ time	Redlog #	Redlog time	Z3 #	Z3 time
BOUNCE	sat	123	1028.9	91	466.6	118	446.9	**123**	**8.9**
(180)	unsat	57	153.4	57	22.0	56	85.1	**57**	**3.5**
HONG	sat	0	0.0	0	0.0	0	0.0	0	0.0
(20)	unsat	3	0.1	**20**	**10.3**	6	12.4	8	5.6
HYCOMP	sat	0	0.0	0	0.0	0	0.0	0	0.0
(2102)	unsat	1788	7233.3	1589	6626.0	1959	28622.9	**2091**	**2894.8**
KEYMAERA	sat	0	0.0	0	0.0	0	0.0	0	0.0
(421)	unsat	415	125.1	411	55.2	419	217.9	**420**	**10.6**
KISSING	sat	7	74.6	13	79.6	6	3.7	**31**	**1247.0**
(45)	unsat	0	0.0	0	0.0	0	0.0	0	0.0
METITARSKI	sat	4871	1018.2	4717	3070.4	4898	6471.2	**5025**	**296.2**
(7713)	unsat	2488	2681.4	2588	1162.2	2628	3612.1	**2682**	**384.8**
RECT	sat	26	404.3	14	391.2	21	308.8	**48**	**35.2**
(91)	unsat	**14**	**486.8**	7	50.0	12	834.3	12	12.4
WITNESS	sat	**65**	**1975.3**	28	11.2	5	68.6	4	99.8
(100)	unsat	0	0.0	0	0.0	0	0.0	**18**	**45.8**
ZANKL	sat	29	236.6	27	113.9	**71**	**308.7**	60	120.4
(166)	unsat	17	13.7	19	7.0	**48**	**57.5**	27	2.2
all		9903	15431.7	9581	12065.5	10247	41050.0	**10606**	**5167.0**
(10838)	sat	5121	4737.8	4890	4132.8	5119	7607.8	**5291**	**1807.3**
	unsat	4782	10693.9	4691	7932.7	5128	33442.2	**5315**	**3359.7**

Table 6.7: Comparison of the last two SMT-RAT strategies from Figure 6.2 with the state-of-the-art solvers Redlog and Z3 (the column # contains the number of solved instances and the column *time* contains the amount of seconds needed for solving these instances).

We also evaluated SMT-RAT strategies (the two rightmost strategies in Figure 6.2) that involve our implementation of the cylindrical algebraic decomposition method. This procedure is complete for nonlinear real arithmetic, therefore both strategies always determine the satisfiability of a given example, if they terminate within the timeout. We compared them to Redlog[3], which also uses the virtual substitution method in combination with the CAD method, and Z3 (Version 4.4.1), which is the currently fastest SMT solver for nonlinear real arithmetic. Z3 uses an interaction of SAT solving and the CAD method, which is even tighter than the usual framework for less-lazy SMT solving.

Table 6.7 shows that Redlog and Z3 can solve 344 and 703 examples more, respectively, than the best SMT-RAT strategy. However, one SMT-RAT strategy (which corresponds to the second column) solves more instances of the benchmark set HONG, which we must accredit to the interval constraint propagation used. Moreover, the other SMT-RAT strategy (which corresponds to the first column) can solve the most unsatisfiable examples of RECT and the most satisfiable instances of WITNESS. For the latter, the third SMT-RAT strategy of Figure 6.2, which does not involve a $\texttt{Module}_{\text{CAD}}$, solves even more instances and additionally some of the unsatisfiable examples (see the third column of Table 6.6). Overall, Z3 performs best in most of the benchmark sets.

As mentioned before, SMT-RAT would benefit a lot from a polynomial factorization that is as

[3] (from http://svn.code.sf.net/p/reduce-algebra, Revision 3758)

good as those used by Redlog or Z3. Using the CAD method, Redlog solves many more examples than without. From the previous results in Table 6.6, we conclude that their implementation of the CAD method performs better than the one in SMT-RAT, which raises our hopes to catch up with Redlog and maybe Z3 as soon as we compensate this deficit. Due to the very good performance of Z3, we seek to integrate a similar approach into the SMT-RAT framework. Here, we would not only be able to use the CAD method, but also the virtual substitution method, for such an integration.

6.7 Parallel SMT-RAT strategies

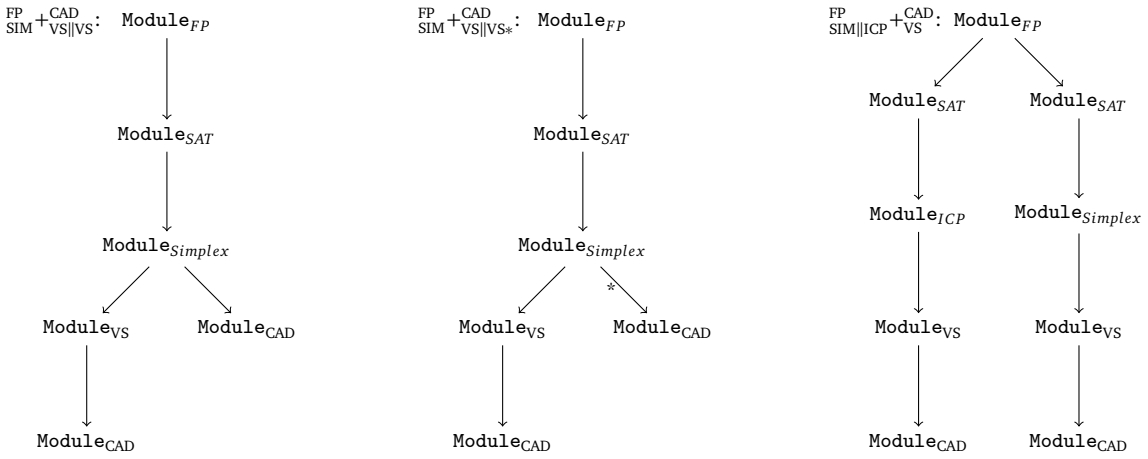


Figure 6.3: Parallel SMT-RAT strategies, which combine the modules $\texttt{Module}_{FP}$, $\texttt{Module}_{SAT}$, $\texttt{Module}_{Simplex}$, $\texttt{Module}_{ICP}$ and $\texttt{Module}_{\text{CAD}}$ with the $\texttt{Module}_{\text{VS}}$ in different ways. By $*$ we denote the condition that the all constraints of the given formula are strict inequalities.

In the previous experiments we concentrated on the contributions of this thesis for a theory solver based on the virtual substitution. We also evaluated combinations of it with other procedures. However, we have not yet used all features, which SMT-RAT strategies provide, and specifically left out the option to use parallel sub-strategies and conditions within an SMT-RAT strategy. These features provide us with an immense number of possibilities and we only illustrate the most prominent strengths and weaknesses of parallel SMT-RAT strategies in this section.

We tested how much impact the approaches from this thesis have on the currently best performing SMT-RAT strategy from Figure 6.2 (the fourth strategy with the results in the first column of Table 6.7). Here, we simply removed the $\texttt{Module}_{\text{VS}}$ from the strategy. The results seem to be clear, as we can solve approximately 1500 additional examples and this within less time, when using a $\texttt{Module}_{\text{VS}}$. However, there are still hundreds of instances, where the strategy without a $\texttt{Module}_{\text{VS}}$ performs better. Therefore, we test by use of the first strategy from Figure 6.3 running both options in parallel, that is when invoking a theory call, we run after an initial check with the $\texttt{Module}_{Simplex}$ on the one hand a $\texttt{Module}_{\text{CAD}}$ and on the other hand a $\texttt{Module}_{\text{VS}}$ followed by

a $\texttt{Module}_{\text{CAD}}$. In contrast to the best strategy from Figure 6.2, we now use a $\texttt{Module}_{FP}$ before applying a $\texttt{Module}_{SAT}$. Just as a $\texttt{Module}_{\text{VS}}$, a $\texttt{Module}_{FP}$ can eliminate variables which reduces the dimension of the problem. In contrast to a $\texttt{Module}_{\text{VS}}$, a $\texttt{Module}_{FP}$ only uses equations, which can be solved for one variable, for this purpose. For some examples, a $\texttt{Module}_{\text{CAD}}$ prefers only a preceding elimination of equations instead of a more thorough variable elimination by the cost of more complex polynomials in the constraints as it is done by a $\texttt{Module}_{\text{VS}}$. With this strategy we try to trigger exactly these examples.

Benchmarks↓	Solvers→	$^{\text{FP}}_{\text{SIM}}+^{\text{CAD}}_{\text{VS}\|\|\text{CAD}}$ #	time	$^{\text{FP}}_{\text{SIM}}+^{\text{CAD}}_{\text{VS}\|\|\text{CAD}*}$ #	time	$^{\text{FP}}_{\text{SIM}\|\|\text{ICP}}+^{\text{CAD}}_{\text{VS}}$ #	time	$^{\text{CAD}}_{\text{SIM}}+_{\text{VS}}\,\|\|\,^{\text{FP}}_{\text{ICP}}+^{\text{CAD}}_{\text{VS}}$ #	time
BOUNCE	sat	22	293.3	66	49.5	122	720.8	**123**	**731.1**
(180)	unsat	57	73.1	24	3.1	**57**	**30.0**	57	31.5
HONG	sat	0	0.0	0	0.0	0	0.0	0	0.0
(20)	unsat	3	0.1	3	0.1	**20**	**0.8**	20	1.0
HYCOMP	sat	0	0.0	0	0.0	0	0.0	0	0.0
(2102)	unsat	1253	3557.8	1260	4478.8	1576	7314.3	**1833**	**7543.5**
KEYMAERA	sat	0	0.0	0	0.0	0	0.0	0	0.0
(421)	unsat	381	16.7	382	208.2	381	16.7	**408**	**16.2**
KISSING	sat	14	40.8	**14**	**39.2**	13	106.7	12	110.6
(45)	unsat	0	0.0	0	0.0	0	0.0	0	0.0
METITARSKI	sat	4876	1918.0	**4948**	**1847.5**	4921	1271.6	4862	1159.2
(7713)	unsat	2474	2550.7	2476	2112.4	**2591**	**1088.1**	2556	950.1
RECT	sat	6	9.4	10	165.6	28	677.6	**30**	**327.1**
(91)	unsat	2	0.8	3	1.1	**13**	**387.3**	11	194.4
WITNESS	sat	4	1.5	4	1.5	**62**	**1962.6**	47	1023.6
(100)	unsat	0	0.0	0	0.0	0	0.0	0	0.0
ZANKL	sat	14	6.7	14	4.1	**28**	**222.8**	26	14.9
(166)	unsat	18	55.8	18	54.1	**18**	**1.3**	18	3.2
all		9124	8524.6	9222	8965.2	9830	13800.6	**10003**	**12106.6**
(10838)	sat	4936	2269.7	5056	2107.4	**5174**	**4962.1**	5100	3366.5
	unsat	4188	6255.0	4166	6857.7	4656	8838.5	**4903**	**8740.0**

Table 6.8: Comparison of the parallel SMT-RAT strategies from Figure 6.3 and the strategy, which runs the last two strategies from Figure 6.2 in parallel (the column # contains the number of solved instances and the column *time* contains the amount of seconds needed for solving these instances).

The results are depicted in the first column of Table 6.8. For METITARSKI, we could solve more satisfiable instances than any other SMT-RAT strategy. The same holds for KISSING. In general, this strategy performs worse, although we have to bear in mind that the preceding $\texttt{Module}_{FP}$ distorts the result. However, there is a second explanation, why for many instances, we do not always perform as well as the best option when running two sub-strategies in parallel (when ignoring the overhead for multithreading). Instead there are some examples where we actually adopt the worst performance. In the case we run modules as backends in parallel and one of them determines the satisfiability of the formula in question, we need to terminate the other modules' satisfiability checks. We require that modules then terminate in a consistent state such that we can reuse them later (incrementally). In general, we achieve this by a recurring query in these backends that checks whether they are allowed to stop their satisfiability check. The more frequent a backend makes this query, the faster all backends terminate after one of them determined the satisfiability. Unfortunately, more queries worsen the performance, hence, we

should not make them in the most basic operations which are invoked very often. In a $\texttt{Module}_{\text{VS}}$, for instance, such an operation could be the squaring of an extremely large polynomial, which might take a very long time. In a $\texttt{Module}_{\text{CAD}}$, we also rely on arithmetic operations on polynomials and, moreover, need to refine real algebraic number representations when comparing them. Both operations can take again a long time, if applied on large polynomials, that is polynomials with many terms, huge coefficients or a high degree.

Considering the examples, on which a $\texttt{Module}_{\text{CAD}}$ performs better than the combination of a $\texttt{Module}_{\text{VS}}$ with a $\texttt{Module}_{\text{CAD}}$, we observe that they often only contain strict inequalities. This is a perfect situation to use the conditions, which SMT-RAT strategies provide. Adding this condition to the branch, which only uses a $\texttt{Module}_{\text{CAD}}$, disables its invocation for all conjunctions of constraints, which also contain weak inequalities or equations. This strategy corresponds to the second one from Figure 6.3 and the results are given by the second column in Table 6.8. It shows that the addition of this condition improves the performance for almost all benchmark sets gaining us almost a hundred additional solved instances.

The last two columns in Table 6.8 show the results for the last strategy from Figure 6.3 and a strategy which simply runs the last two strategies from Figure 6.2 in parallel. For both strategies we can find benchmark sets, in which they perform better, and the second strategy solves overall more examples than any other SMT-RAT strategy we have used so far.

CHAPTER 7

Virtual Substitution for Integer Arithmetic

A popular approach to check quantifier-free linear integer arithmetic formulas $\varphi^{\mathbb{Z}}$ for satisfiability is the *branch-and-bound* framework [Sch86]. It first considers $\varphi^{\mathbb{Z}}$'s real relaxation $\varphi^{\mathbb{R}}$. If it is unsatisfiable then the integer problem is unsatisfiable too. Otherwise, if there exists a real solution then it is either integer-valued, in which case $\varphi^{\mathbb{Z}}$ is satisfiable, or it contains a non-integer value $d \in \mathbb{R} \setminus \mathbb{Z}$ for an integer-valued variable z. In the latter case a *branching* takes place: branch-and-bound reduces the relaxed solution space by excluding all values between $\lfloor d \rfloor$ and $\lceil d \rceil$ in the z-dimension, described by the formula

$$\varphi' \quad = \quad \varphi \wedge (z \leq \lfloor d \rfloor \vee z \geq \lceil d \rceil).$$

This procedure is applied iteratively, i. e., branch-and-bound will now search for real-valued solutions of φ'. It terminates if either an integer solution is found or the relaxation is unsatisfiable. Note that branch-and-bound is incomplete in general even for the decidable logic quantifier-free linear arithmetic.

The most well-known applications combine branch-and-bound with the simplex method. As branching introduces disjunctions and thus in general non-convexity, branching is implemented by case splitting: in one search branch we assume $z \leq \lfloor d \rfloor$, and in a second search branch we assume $z \geq \lceil d \rceil$. Depending on the heuristics, the search can be depth-first (full check of one of the branches, before the other branch is considered), breadth-first (check real relaxations in all current open branches before further branching is applied), or it can follow a more sophisticated strategy.

The combination of branch-and-bound with the simplex method was also explored in the SMT-solving context [DdM06]. The advantage in this setting is that we have more possibilities to design the branching.

- We can integrate a theory solver based on the simplex method as described above, implementing branch-and-bound internally in the theory solver by case splitting. It comes with the advantage that case splitting is always *local* to the current problem of the theory solver and does not affect later problems, and with the disadvantage that we cannot exploit the advantages of *learning*, i. e., to remember reasons of unsatisfiability in certain branches and use this information to speed up the search in other branches.

- Alternatively, given a non-integer solution d for a variable z found by the theory solver on a relaxed problem, we can lift the branching to the SAT solver by extending the current formula with a new clause $(z \leq \lfloor d \rfloor \vee z \geq \lceil d \rceil)$ [BNOT06]. The newly added clause must be satisfied in order to satisfy the extended formula. Therefore, the SAT solver assigns (the Boolean abstraction variable of) either $z \leq \lfloor d \rfloor$ or $z \geq \lceil d \rceil$ to true, i. e., the branching takes place. On the positive side, lifting branching information and branching decisions to the SAT solver allows us to learn from information collected in one branch, and to use this learned information to speed up the search in other branches. On the negative side, the branching is not local anymore as it is remembered in a learned clause. Therefore, it might cause unwanted splittings in later searches.

To unify advantages, `MathSAT5` [Gri12] implements a combined approach with theory-internal splitting up to a given depth and splitting at the logical level beyond this threshold.

Following the branch-and-bound approach in combination with the simplex method, we can also transfer the idea to nonlinear integer arithmetic: We can use decision procedures for nonlinear real arithmetic to find solutions for the relaxed problem and branch at non-integer solutions of integer-valued variables. However, there are some important differences. Most notably, the computational effort for checking the satisfiability of nonlinear real-arithmetic problems is *much* higher than in the linear case. If we have found a real-valued solution and apply branching to find integer solutions, the branching will *refine* the search in the virtual substitution: it will create additional test candidates, which will serve as roots for new sub-trees in the search tree. However, the search trees in both branches have a lot in common, that means, a lot of the same work has to be done for both sides of the branches. To prevent the solvers from doing much unnecessary work, we have to carefully design the branch-and-bound procedure. Here, we can make use of branching lemmas as they are provided by `SMT-RAT`, which we introduced in Section 3.4.3.

- Branching has to be lifted to the SAT solver level to enable learning, both in the form of branching lemmas as well as explanations (infeasible subsets) for unsatisfiability in different branches.

- Learning explanations will allow us to speed up the search by transferring useful information between different branches. However, we need to handle branching lemmas thoughtfully and assure that learned branching lemmas will not lead to branching for all future subproblems, but only for “similar” ones where the branching will probably be useful.

- As branching refines the search, it has to work in an incremental fashion without resetting solver states.
- If possible, the search strategies of the underlying decision procedures for nonlinear real-arithmetic have to be tuned to prefer integer solutions (and if they can choose between different integer values, they must choose the most "promising" one).
- Last but not least, as the performance of solving quantifier-free nonlinear real-arithmetic formulas for satisfiability highly improves if different theory solvers implementing different procedures are used in combination, a practically relevant branch-and-bound approach for nonlinear integer arithmetic should support this option.

7.1 Branch-and-bound with virtual substitution

In this section we present how the virtual substitution method can be embedded into the branch-and-bound framework to check the satisfiability of a given quantifier-free (nonlinear) integer-arithmetic formula. Note that the concepts of this chapter are not built upon the data structure (VSST), which we introduced in Section 4.2, but upon the more general setting of a satisfiability check with the virtual substitution which we formulated in Section 4.1. Nonetheless, all of the following ideas can be directly applied to an SMT-compliant theory solver based on the specifications of Section 4.2.

Assume that we want to check the quantifier-free integer-arithmetic formula $\varphi_n^{\mathbb{Z}}$ for satisfiability such that $\text{Vars}(\varphi_n^{\mathbb{Z}}) = \{z_1, \ldots, z_n\}$. Then we first apply the virtual substitution on the real relaxation $\varphi_n^{\mathbb{R}}$ of $\varphi_n^{\mathbb{Z}}$. If we determine unsatisfiability, we know that $\varphi_n^{\mathbb{Z}}$ is also unsatisfiable. Otherwise, if we have found a solution S with the virtual substitution for $\varphi_n^{\mathbb{R}}$, as illustrated in Figure 4.1, then S maps the variables $\text{Vars}(\varphi_n^{\mathbb{R}}) = \{z_1, \ldots, z_n\}$ to test candidates $S(z_j) = t_{z_j}^{i_j}$ $(1 \leq j \leq n)$. For integer arithmetic formulas we can omit considering strict inequalities as described in Definition 16. This saves us from considering test candidates with infinitesimals as introduced in [Wei97] and the comparably more cases they entail. Therefore, $S(z_j)$ is either $-\infty$ or of the form $\frac{q_{j,1}+q_{j,2}\sqrt{q_{j,3}}}{q_{j,4}}$ with $q_{j,1}, \ldots, q_{j,4} \in \mathbb{Z}[z_1, \ldots, z_{j-1}]$ (roots parametrized in some polynomials).

If a solution S for the relaxation $\varphi^{\mathbb{R}}$ is found then there is a true leaf in the search tree, as illustrated in Figure 4.1. We now try to construct an integer solution S^* from the parametrized solution S, as illustrated in Figure 7.1, traversing the solution path from the true leaf backwards. If the test candidate $t_{z_1}^{i_1}$ for z_1 is not $-\infty$, it does not contain any variables, thus we can determine whether its value is an integer and set $S^*(z_1)$ to this value. If $t_{z_1}^{i_1} = -\infty$, we can take any integer which is strictly smaller than all the other test candidates in $\text{tcs}(z_1, \varphi_1^{\mathbb{R}})$. Now we iterate backwards: for each test candidate $t_{z_j}^{i_j}$ on the solution path, which is not $-\infty$, we substitute the values $S^*(z_1), \ldots, S^*(z_{j-1})$ for the variables $z_1, \ldots, z_{j-1}$, resulting in

$$S^*(z_j) := S(z_j)[S^*(z_1)/z_1] \ldots [S^*(z_{j-1})/z_{j-1}],$$

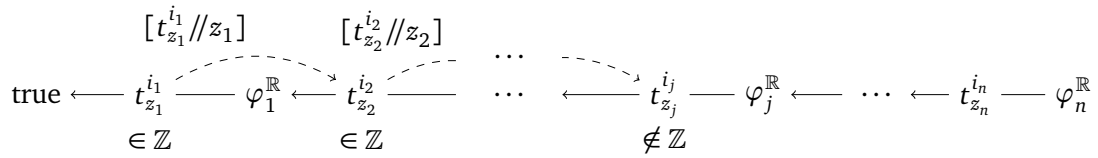


Figure 7.1: Solution path from Figure 4.1 traversed backwards from the leaf to the root.

which again does not contain any variables and we can evaluate whether its value is integer. If $t^{i_j}_{z_j} = -\infty$ then we evaluate all test candidates from $\text{tcs}(z_j, \varphi^{\mathbb{R}}_j)$ whose side conditions hold by substituting $S^*(z_1), \ldots, S^*(z_{j-1})$ for $z_1, \ldots, z_{j-1}$ in the test candidate expressions, and we set $S^*(z_j)$ to an integer value that is strictly smaller than all those test candidate values. We repeat this procedure until either a full integer solution is found or the resulting value is not integer in one dimension.

If all test candidate values are integer then VS returns `sat`. Otherwise, if we determine that $S^*(z_j)$ for some j is not integer-valued, then there is some $d \in \mathbb{Z}$ such that $S^*(z_j) \in (d-1, d)$. In this case we return the branching lemma

$$\Big(\bigwedge_{\psi \in \text{Orig}_{z_j}(S(z_j))} \psi\Big) \to (z_j \leq d-1 \vee z_j \geq d),$$

where $\text{Orig}_{z_j}(S(z_j))$ denotes the VS module's received constraints being responsible for the creation of the test candidate $S(z_j)$. We can determine this set recursively with $\text{Orig}_{z_j}(S(z_j)) := \text{Orig}_{z_j}(c)$ if we used constraint $c \in C_\sim(\varphi^{\mathbb{R}}_j)$ for generating the test candidate $S(z_j)$, and where

$$\text{Orig}_{z_j}(c) := \begin{cases} c & \text{, if } j = n \\ \text{Orig}_{z_{j+1}}(c) & \text{, if } z_j \notin \text{Vars}(c) \\ \text{Orig}_{z_{j+1}}(S(z_{j+1})) \cup \text{Orig}_{z_{j+1}}(c') & \text{, if } c' \in C_\sim(\varphi^{\mathbb{R}}_{j+1}) \text{ such that} \\ & \quad c \in C_\sim(c'[S(z_{j+1}) // z_{j+1}]) \\ \text{Orig}_{z_{j+1}}(S(z_{j+1})) & \text{, otherwise.} \end{cases}$$

Note that the last case occurs if the given constraint is introduced through a test candidate's side condition.

Basically, if we have found a non-integer valued test candidate $S^*(z_j) \notin \mathbb{Z}$, we can still continue the procedure to determine all other non-integer-valued test candidates. It would gain us the flexibility of being able to select heuristically on which variable value we want to branch, but it comes at high computational costs, as we need to compute with nested fractions and square roots. Therefore, we do not consider other heuristics but always branch on the first detected non-integer value.

This procedure is sound, as we do not prune any integer solutions. It is not complete, as it might branch infinitely often for the same variable at an always increasing or always decreasing

value. This procedure can also be used to check a quantifier-free mixed integer-real arithmetic formula for satisfiability, if we eliminate real-valued variables first.

7.2 Experimental results

We evaluated different sequential strategies for solving quantifier-free nonlinear integer arithmetic formulas (QF_NIA), using the following modules $\texttt{M}_i$:

- The SAT solver module $\texttt{Module}_{SAT}$ behaves as explained in Section 3.4.2.
- $\texttt{Module}_{\text{SAT}_{\text{Stop}}}$ works similarly except that it returns `unknown` if an invoked theory solver module returns `unknown`, instead of continuing the search for further Boolean assignments. The module $\texttt{Module}_{\text{SAT}_{\text{Stop}}}$ provides us a reference: if this module is able to solve a problem then the problem can be considered irrelevant for branch-and-bound (as branch-and-bound was not involved).
- The module $\texttt{Module}_{Simplex}$ implements the simplex method with branching lemma generation, as explained in Section 3.4.4.
- The theory solver modules $\texttt{Module}_{\text{VS}}$ (implementing VS) and $\texttt{Module}_{\text{CAD}}$ (implementing CAD) check the real relaxation of a QF_NIA input formula. If the relaxation is unsatisfiable they return `unsat`, if they coincidentally find an integer solution they return `sat`, otherwise they return `unknown` (without applying branch-and-bound).
- The VS module $\texttt{Module}_{\text{VS}_{\mathbb{Z}}}$ constructs branching lemmas as explained in Section 7.1.
- The CAD module $\texttt{Module}_{\text{CAD}_{\mathbb{Z}}}$ constructs branching lemmas as introduced in [1].
- Bit-blasting is implemented in the module $\texttt{Module}_{IntBlast}$. In our strategies it will be combined with a preceding incremental variable bound widening module $\texttt{Module}_{IncWidth}$, which constrains, for instance, first that all variables are in $[-1, 2]$, if no solution can be found, it requires all variables to be in $[-3, 4]$ etc.

All experiments were carried out on AMD Opteron 6172 processors. Every solver was allowed to use up to 4 GB of memory and 200 seconds of wall clock time.

For our experiments we used the largest benchmark sets for QF_NIA from the last SMT-COMP: APROVE, LEIPZIG (both generated by automated termination analysis) and CALYPTO (generated by sequential equivalence checking). Additionally, we crafted a new benchmark set CALYPTO_{∞} by removing all variable bound constraints from CALYPTO and thereby obtaining unbounded problems (together 8572 problem instances, see headline in Figure 7.5 for the size of each set).

Selection of a VS heuristic The `SMT-RAT` strategy $\texttt{Module}_{\text{SAT}_{\text{Stop}}} \rightarrow \texttt{Module}_{\text{VS}}$ could solve 7215 `sat` and 84 `unsat` instances, ran out of time or memory for 1146 instances, and returned `unknown`

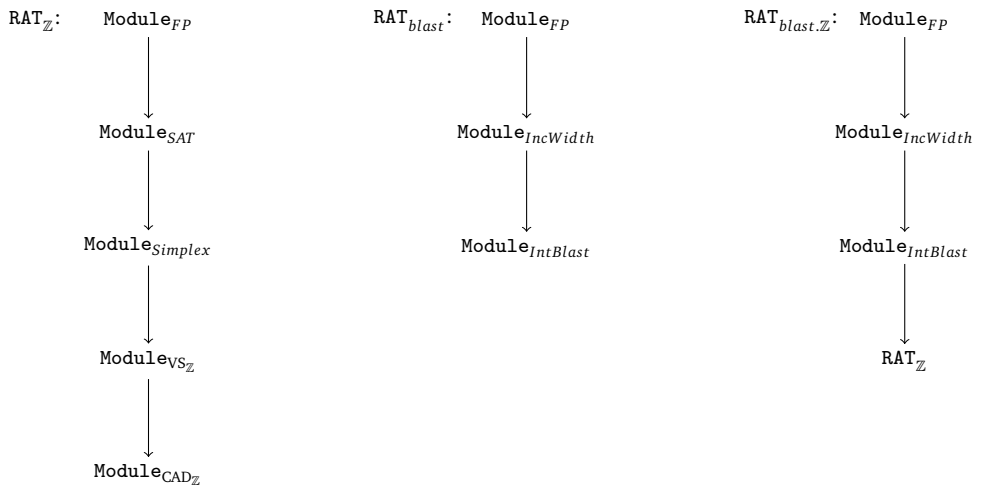


Figure 7.2: The SMT-RAT strategies used for the experimental results.

	$\mathrm{VS}_{\mathbb{R}}$		$\mathrm{VS}_{\mathbb{Z}}$	
	#	time	#	time
sat	30	714.2	**93**	**487.3**
unsat	0	0.0	**10**	**9.2**

Figure 7.3: Comparison of 2 VS heuristics on 126 (101 sat, 25 unsat) for branch-and-bound relevant instances (the column # contains the number of solved instances and the column *time* contains the number of seconds needed for solving these instances).

for 127 instances. Applying the SMT-RAT strategy Module$_{SAT}$ → Module$_{\mathrm{VS}}$ to those 127 instances, we can solve an additional 30 sat instances. If we replace the module Module$_{\mathrm{VS}}$ by the Module$_{\mathrm{VS}_{\mathbb{Z}}}$ module, which applies branching lemmas, we can solve a further 63 sat and 10 unsat instances (see Figure 7.3).

Combined strategies We crafted three strategies, depicted in Figure 7.2, to combine different theory solver modules[1]. The strategy RAT$_{blast.\mathbb{Z}}$ combines RAT$_{blast}$ and RAT$_{\mathbb{Z}}$ by first using bit-blasting up to a width of 4 bits. If this does not yield a solution, it continues to use RAT$_{\mathbb{Z}}$.

We compared these three strategies with the two fastest SMT solvers from the 2015 SMT-COMP for QF_NIA: Z3 and AProVE. Though CVC4 performed worse than these two solvers, its experimental version solved slightly more instances than AProVE in about half of the time; we did not include it here but expect it to perform between Z3 and AProVE. Figure 7.5 shows that RAT$_{\mathbb{Z}}$ and RAT$_{blast}$ complement each other well, especially for satisfiable instances. Compared to Z3 and AProVE, RAT$_{blast.\mathbb{Z}}$ solves more satisfiable instances and does this even faster by a factor of more than 10 and 6, respectively. The strategy RAT$_{\mathbb{Z}}$ solves less instances, but, as shown in Figure 7.4, this strategy solves the first 85 percent of the examples faster than any other SMT-RAT strategy or SMT solver. On unsatisfiable instances, however, Z3 is still better than SMT-RAT while

[1]Additionally, all of these strategies employ a common preprocessing.

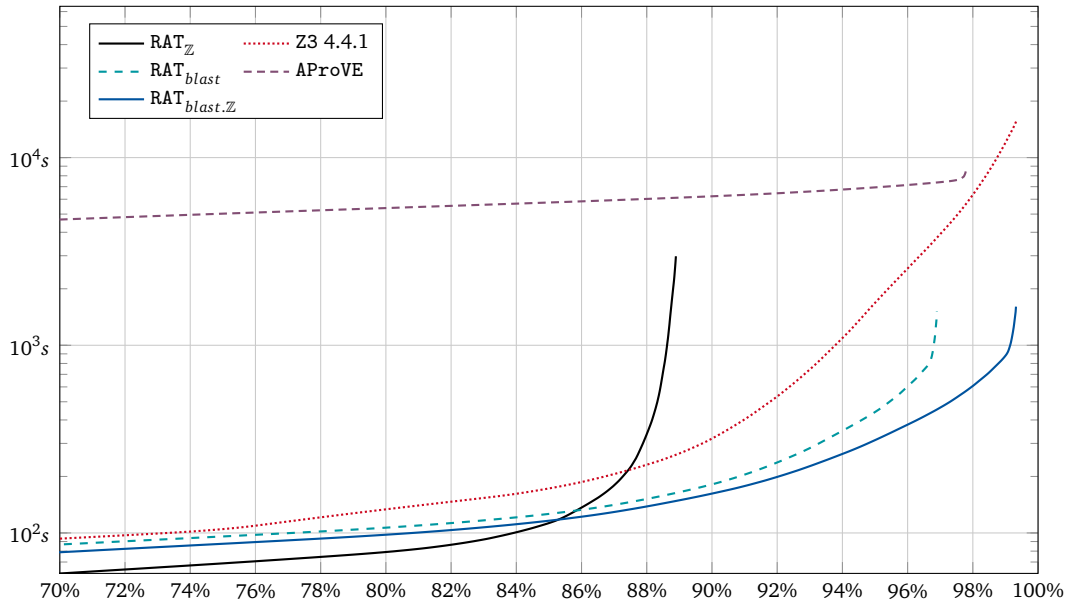


Figure 7.4: Cumulative time to solve instances from all benchmark sets.

Benchmark→ / Solver↓		APROVE (8129) #	APROVE (8129) time	CALYPTO (138) #	CALYPTO (138) time	LEIPZIG (167) #	LEIPZIG (167) time	CALYPTO$_\infty$ (138) #	CALYPTO$_\infty$ (138) time	*all* (8572) #	*all* (8572) time
RAT$_{\mathbb{Z}}$	sat	7283	2294.8	67	71.2	9	260.4	133	298.9	7492	2925.3
	unsat	73	14.3	52	40.7	0	0.0	**3**	**< 0.1**	128	55.1
RAT$_{blast}$	sat	8025	866.3	21	35.6	156	603.3	87	16.0	8289	1521.2
	unsat	12	0.4	5	0.1	0	0.0	0	0.0	17	0.5
RAT$_{blast.\mathbb{Z}}$	sat	**8025**	**780.7**	**79**	**122.3**	156	511.5	**134**	**21.8**	**8394**	**1436.3**
	unsat	71	42.6	46	127.5	0	0.0	3	0.1	120	170.2
Z3	sat	7992	14695.5	78	19.1	158	427.6	126	57.3	8354	15199.5
	unsat	**102**	**595.9**	**57**	**117.6**	0	0.0	3	2.3	**162**	**715.8**
AProVE	sat	8025	7052.2	74	559.1	**159**	**696.5**	127	685.2	8385	8993.0
	unsat	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0

Figure 7.5: Comparison of 3 SMT-RAT strategies to currently fastest SMT solvers for QF_NIA (the column # contains the number of solved instances and the column *time* contains the amount of seconds needed for solving these instances).

AProVE is not able to deduce unsatisfiability due to its pure bit-blasting approach.

We also tested all SMT-RAT strategies which use branch-and-bound, once with and once without using a branching premise. Here we could not detect any notable difference, which we mainly relate to the fact that those problem instances, for which branch-and-bound comes to application, are almost always pure conjunctions of constraints and involve only a small number of branching lemma liftings. For a more reliable evaluation a larger set of QF_NIA benchmarks would be needed.

CHAPTER 8

A Synergy of the Greatest Common Divisor Calculation, Factorization and Intermediate Result Caching

Data structures and operations on polynomials form a vital part of the foundations of, e. g., computer algebra systems or implementations of procedures based on Gröbner bases, the cylindrical algebraic decomposition or the virtual substitution. We also highly depend on polynomials, if we aim to calculate the reachability probabilities of *parametric discrete-time Markov chains* (PDTMCs). This is a parametrized version of *discrete-time Markov chains* (DTMCs), which is a modeling formalism for systems exhibiting probabilistic behavior. For more details on DTMCs we refer to [BK08]. In contrast to the state transition systems which are used to model a DTMC, a PDTMC allows us to label transitions not only with probabilistic quantities but also with rational functions over real-valued variables instead.

Definition 32 (Rational function) *A* rational function *is a quotient* $f = \frac{p_1}{p_2}$ *of two polynomials* p_1, p_2 *with* $p_2 \neq 0$[a].

[a] $p_2 \neq 0$ means that p_2 cannot be simplified to 0.

We can compute the reachability probabilities for a PDTMC as introduced in [Daw04, HHZ11], where we iteratively replace states and their incident transitions by direct transitions from the predecessors to the successors. It yields a model having only initial and absorbing states and the transitions between these states carry—as rational functions over the real-valued model parameters—the probability of reaching the absorbing states from the initial states. In [3], where we transfer the ideas from [ÁJW+10] for DTMCs to PDTMCs, we presented an alternative approach. Here, we use a state elimination strategy based on a *recursive graph decomposition* of the PDTMC into strongly connected subgraphs, which we refer to as *strongly connected components* (SCCs). Each (sub-)SCC is replaced by abstract transitions that lead from its ingoing states to its outgoing states. The resulting rational functions describe the probability of entering the SCC and

eventually leaving it.

The two aforementioned procedures build rational functions representing a given PDTMC's reachability probabilities. These rational functions might grow rapidly in the process of both procedures and thereby form one of the major bottlenecks of this methodology. As already argued in [HHZ11], the best way to stem this blow-up is the cancellation of the rational functions in every computation step, which involves—apart from *addition*, *multiplication*, and *division* of rational functions—the rather expensive calculation of the *greatest common divisor* (gcd) of two polynomials.

In this chapter we present how we can handle this problem: Additional maintenance and storage of (partial) polynomial factorizations can lead to remarkable speed-ups in the gcd computation, especially when dealing with symmetrically structured benchmarks where many similar polynomials occur. We present an optimized algorithm called $\gcd_{\mathcal{F}}$ which *operates on the (partial) factorizations* of the polynomials to compute their gcd. During the calculations, the factorizations are also refined. On this account we reformulate the arithmetic operations on rational functions such that they preserve their numerator's and denominator's factorizations, if it is possible with reasonable effort.

8.1 Factorized polynomials: Partial factorizations as polynomial representation

We can represent a polynomial p by a factorization of p as introduced in Definition 13, where we use the set $\{0^1\}$, if $p = 0$. We denote the *set of all polynomial factorizations where the polynomial can be* 0 by $\mathrm{FAC}_0 = \mathrm{FAC} \cup \{\{0^1\}\}$.

Given a polynomial p's factorization $\mathcal{F}_p = \{p_1^{e_1}, \ldots, p_n^{e_n}\} \in \mathrm{FAC}_0$ we can obtain the bases with

$$\text{bases}\colon\ \mathrm{FAC}_0 \to \mathbb{P}_{<\infty}(\mathrm{POL} \setminus \{1\}) : \{p_1^{e_1}, \ldots, p_n^{e_n}\} \mapsto \{p_1, \ldots, p_n\}$$

and we get the exponent of a base q with

$$\exp\colon\ \mathrm{POL} \times \mathrm{FAC}_0 \to \mathbb{N}_0 :\ (q, \{p_1^{e_1}, .., p_n^{e_n}\}) \mapsto \begin{cases} e_i & , \exists i \in \{1, .., n\}.\ p_i = q, \\ 0 & , \text{otherwise.} \end{cases}$$

Using the auxiliary function

$$.^{\mathrm{red}}\colon\ \mathbb{P}_{<\infty}(\mathrm{POL}^{\mathbb{N}_0}) \to \mathrm{FAC}_0 : \mathcal{F}_p \mapsto \begin{cases} \{p_i^{e_i} \in \mathcal{F}_p \mid |p_i| \neq 1 \wedge e_i > 0\} & , 0 \notin \text{bases}(\mathcal{F}_p), \\ \{0^1\} & , \text{otherwise.} \end{cases}$$

in order to achieve a well-defined polynomial factorization (or $\{0^1\}$), we can specify the following operations on polynomial factorizations (and $\{0^1\}$):

- The operation $\mathcal{F}_{p_1} \cup_{\mathcal{F}} \mathcal{F}_{p_2}$ results in a factorization of a (not necessarily least) common

multiple of two nonzero polynomials p_1 and p_2 and is defined by

$$\begin{aligned}&\cdot \cup_{\mathcal{F}} \cdot \ : \mathrm{FAC} \times \mathrm{FAC} \to \mathrm{FAC} : \\ &\quad (\mathcal{F}_{p_1}, \mathcal{F}_{p_2}) \mapsto \{q^{\max(\{\exp(q,\mathcal{F}_{p_1}),\exp(q,\mathcal{F}_{p_2})\})} \mid q \in \mathrm{bases}(\mathcal{F}_{p_1}) \cup \mathrm{bases}(\mathcal{F}_{p_2})\}^{\mathrm{red}}\end{aligned}$$

- The operation $\mathcal{F}_{p_1} \cap_{\mathcal{F}} \mathcal{F}_{p_2}$ yields a factorization of a (not necessarily greatest) common divisor of two nonzero polynomials p_1 and p_2 and is defined by

$$\begin{aligned}&\cdot \cap_{\mathcal{F}} \cdot \ : \mathrm{FAC} \times \mathrm{FAC} \to \mathrm{FAC} : \\ &\quad (\mathcal{F}_{p_1}, \mathcal{F}_{p_2}) \mapsto \{q^{\min(\{\exp(q,\mathcal{F}_{p_1}),\exp(q,\mathcal{F}_{p_2})\})} \mid q \in \mathrm{bases}(\mathcal{F}_{p_1}) \cap \mathrm{bases}(\mathcal{F}_{p_2})\}^{\mathrm{red}}\end{aligned}$$

- The binary operations $\cdot_{\mathcal{F}}$, $+_{\mathcal{F}}$ correspond to multiplication and addition, respectively, and are defined by

$$\begin{aligned}&\cdot \cdot_{\mathcal{F}} \cdot \ : \mathrm{FAC}_0 \times \mathrm{FAC}_0 \to \mathrm{FAC}_0 : \\ &\quad (\mathcal{F}_{p_1}, \mathcal{F}_{p_2}) \mapsto \{q^{\exp(q,\mathcal{F}_{p_1})+\exp(q,\mathcal{F}_{p_2})} \mid q \in \mathrm{bases}(\mathcal{F}_{p_1}) \cup \mathrm{bases}(\mathcal{F}_{p_2})\}^{\mathrm{red}}\end{aligned}$$

$$\begin{aligned}&\cdot +_{\mathcal{F}} \cdot \ : \mathrm{FAC}_0 \times \mathrm{FAC}_0 \to \mathrm{FAC}_0 : \\ &\quad (\mathcal{F}_{p_1}, \mathcal{F}_{p_2}) \mapsto \begin{cases} \mathcal{F}_{p_2} & , \ p_1 = 0 \\ \mathcal{F}_{p_1} & , \ p_2 = 0 \\ D \cdot_{\mathcal{F}} \left\{\left(\prod_{p_1' \in (\mathcal{F}_{p_1} :_{\mathcal{F}} D)} p_1'\right) + \left(\prod_{p_2' \in (\mathcal{F}_{p_2} :_{\mathcal{F}} D)} p_2'\right)\right\}^{\mathrm{red}} & , \text{otherwise} \end{cases}\end{aligned}$$

  where $D = \mathcal{F}_{p_1} \cap_{\mathcal{F}} \mathcal{F}_{p_2}$.

- The operation $:_{\mathcal{F}}$ calculates the polynomial factorization of the quotient of a polynomial p_1 and a nonzero polynomial p_2 via their factorizations $\mathcal{F}_{p_1}$ and $\mathcal{F}_{p_2}$. Note that $\mathcal{F}_{p_1} :_{\mathcal{F}} \mathcal{F}_{p_2}$ is a factorization of p_1/p_2 only if $\mathcal{F}_{p_1}$ and $\mathcal{F}_{p_2}$ are sufficiently refined and p_2 divides p_1.

$$\begin{aligned}&\cdot :_{\mathcal{F}} \cdot \ : \mathrm{FAC}_0 \times \mathrm{FAC} \to \mathrm{FAC}_0 : \\ &\quad (\mathcal{F}_{p_1}, \mathcal{F}_{p_2}) \mapsto \{q^{\max(\{0,e-\exp(q,\mathcal{F}_{p_2})\})} \mid q^e \in \mathcal{F}_{p_1}\}^{\mathrm{red}}\end{aligned}$$

Example 22 illustrates the application of the above operations.

8.2 Greatest common divisor computation of factorized polynomials

Given the factorizations $\mathcal{F}_{p_1}$ and $\mathcal{F}_{p_2}$, Algorithm 14 calculates the factorizations $\mathcal{F}_g$, $\mathcal{F}_{\frac{p_1}{g}}$, and $\mathcal{F}_{\frac{p_2}{g}}$. Intuitively, the algorithm maintains the fact that $G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1'$ is a factorization of p_1, where G contains common factors of p_1 and p_2, F_1 is going to be checked whether it contains further common factors, and F_1' does not contain any common factors. In the outer while-loop, an element $q_1^{e_1}$ to be checked is taken from F_1. In the inner while-loop, a factorization $G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2'$

Algorithm 14 gcd computation with factorization refinement

$\gcd_{\mathcal{F}}$(factorization $\mathcal{F}_{p_1}$, factorization $\mathcal{F}_{p_2}$)
begin
1: $G := (\mathcal{F}_{p_1} \cap_{\mathcal{F}} \mathcal{F}_{p_2})$
2: $F_i := \mathcal{F}_{p_i} :_{\mathcal{F}} G$ and $F_i' := \emptyset$ for $i = 1, 2$
3: **while** exists $q_1^{e_1} \in F_1$ with $q_1 \neq 1$ **do**
4: $F_1 := F_1 \setminus \{q_1^{e_1}\}$
5: **while** $q_1 \neq 1$ and exists $q_2^{e_2} \in F_2$ with $q_2 \neq 1$ **do**
6: $F_2 := F_2 \setminus \{q_2^{e_2}\}$
7: **if** $\neg$irreducible$(q_1) \vee \neg$irreducible(q_2) **then** $p := \gcd(q_1, q_2)$
8: **else** $p := 1$
9: **if** $p = 1$ **then**
10: $F_2' := F_2' \cdot_{\mathcal{F}} \{q_2^{e_2}\}$
11: **else**
12: $q_1 := \frac{q_1}{p}$
13: $F_i := F_i \cdot_{\mathcal{F}} \{p^{e_i - \min(e_1, e_2)}\}$ for $i = 1, 2$
14: $F_2' := F_2' \cdot_{\mathcal{F}} \{(\frac{q_2}{p})^{e_2}\}$
15: $G := G \cdot_{\mathcal{F}} \{p^{\min(e_1, e_2)}\}$
16: **end if**
17: **end while**
18: $F_1' := F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}\}$
19: $F_2 := F_2 \cdot_{\mathcal{F}} F_2'$
20: $F_2' := \emptyset$
21: **end while**
22: **return** (F_1', F_2, G)
end

of p_2 is maintained such that F_2' does not contain any common factors with q_1, and F_2 is still to be checked.

Now we explain the algorithm in more detail. Initially, a factorization G of a common divisor of p_1 and p_2 is set to $\mathcal{F}_{p_1} \cap_{\mathcal{F}} \mathcal{F}_{p_2}$ (Line 1). The remaining factors of p_1 and p_2 are stored in F_1 resp. F_2. The sets F_1' and F_2' contain factors of p_1 and p_2, respectively, whose greatest common divisor is 1 (Line 2). The algorithm now iteratively adds further common divisors of p_1 and p_2 to G until it is a factorization of their gcd. For this purpose, we consider all factors in F_2 for each factor in F_1 and calculate the gcd of their bases using standard gcd computation for polynomials (Line 7). Note that the main concern of Algorithm 14 is to avoid the application of this expensive operation as far as possible and to apply it to preferably simple polynomials otherwise. Where the latter is entailed by the idea of using factorizations, the former can be achieved by excluding pairs of factors for which we can cheaply decide that both are *irreducible*, i. e., they have no non-trivial divisors. If factors $q_1^{e_1} \in F_1$ and $q_2^{e_2} \in F_2$ with $p := \gcd(q_1, q_2) = 1$ are found, we just shift $q_2^{e_2}$ from F_2 to F_2' (Line 10). Otherwise, we can add $p^{\min(e_1, e_2)}$, which is the gcd of $q_1^{e_1}$ and $q_2^{e_2}$, to G and extend the factors F_1 and F_2, respectively, which could still contain common divisors, by

$p^{e_1-\min(e_1,e_2)}$ resp. $p^{e_2-\min(e_1,e_2)}$ (Line 12-15).

Furthermore, F_2' obtains the new factor $(\frac{q_2}{p})^{e_2}$, which certainly has no common divisor with any factor in F_1'. Finally, we set the basis q_1 to $\frac{q_1}{p}$, excluding the just found common divisor. If all factors in F_2 have been considered for common divisors with q_1, we can add it to F_1' and continue with the next factor in F_1, for which we must reconsider all factors in F_2' and, therefore, shift them to F_2 (Line 18-20). The algorithm terminates, if the last factor of F_1 has been processed, returning the factorizations $\mathcal{F}_p$, $\mathcal{F}_{\frac{p_1}{p}}$ and $\mathcal{F}_{\frac{p_2}{p}}$, which we can use to refine the factorizations of p_1 and p_2 via $\mathcal{F}_{p_1} := \mathcal{F}_{\frac{p_1}{p}} \cdot_{\mathcal{F}} G$ and $\mathcal{F}_{p_2} := \mathcal{F}_{\frac{p_2}{p}} \cdot_{\mathcal{F}} G$.

Example 22 *Assume we want to apply Algorithm 14 to the factorizations* $\mathcal{F}_{x_1x_2x_3} = \{(x_1x_2x_3)^1\}$ *and* $\mathcal{F}_{x_1x_2} = \{(x_1)^1, (x_2)^1\}$. *We initialize* $G = F_1' = F_2' = \{(1)^1\}$, $F_1 = \mathcal{F}_{x_1x_2x_3}$ *and* $F_2 = \mathcal{F}_{x_1x_2}$. *First, we choose the factors* $(q_1)^{e_1} = (x_1x_2x_3)^1$ *and* $(x_1)^1$ *and remove them from* F_1 *resp.* F_2. *The* gcd *of their bases is* x_1, *hence we only update* q_1 *to* $(x_2x_3)^1$ *and* G *to* $\{(x_1)^1\}$. *Then we remove the next and last element* $(x_2)^1$ *from* F_2. *Its basis and* q_1 *have the* gcd x_2 *and we therefore update* q_1 *to* $(x_3)^1$ *and* G *to* $\{(x_1)^1, (x_2)^1\}$. *Finally, we add* $(x_3)^1$ *to* F_1' *and return the expected result* $(\{(x_3)^1\}, \{(1)^1\}, \{(x_1)^1, (x_2)^1\})$. *Using these results, we can also refine* $\mathcal{F}_{x_1x_2x_3} = F_1' \cdot_{\mathcal{F}} G = \{(x_1)^1, (x_2)^1, (x_3)^1\}$ *and* $\mathcal{F}_{x_1x_2} = F_2 \cdot_{\mathcal{F}} G = \{(x_1)^1, (x_2)^1\}$.

Theorem 7 *Let* p_1 *and* p_2 *be two polynomials with factorizations* $\mathcal{F}_{p_1}$ *resp.* $\mathcal{F}_{p_2}$. *Applying Algorithm 14 to these factorizations results in* $\gcd(\mathcal{F}_{p_1}, \mathcal{F}_{p_2}) = (\mathcal{F}_{q_1}, \mathcal{F}_{q_2}, G)$ *with* G *being a factorization of the greatest common divisor* p *of* p_1 *and* p_2, *and* $\mathcal{F}_{q_1}$ *and* $\mathcal{F}_{q_2}$ *being factorizations of* $\frac{p_1}{p}$ *resp.* $\frac{p_2}{p}$.

Proof 7 We denote the product of a factorization $\mathcal{F}_p$ by $\mathcal{P}(\mathcal{F}_p) = \prod_{q^e \in \mathcal{F}_p} q^e$ and the standard greatest common divisor computation for polynomials by gcd.

We define the following Hoare-style assertion network:

GCD(factorization $\mathcal{F}_{p_1}$, factorization $\mathcal{F}_{p_2}$)
begin

1: $\{true\}$
2: $G := (\mathcal{F}_{p_1} \cap_{\mathcal{F}} \mathcal{F}_{p_2})$
3: $\{G = \mathcal{F}_{p_1} \cap_{\mathcal{F}} \mathcal{F}_{p_2}\}$
4: $F_i := \mathcal{F}_{p_i} :_{\mathcal{F}} G$ and $F_i' := \emptyset$ for $i = 1, 2$
5: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \wedge \mathcal{P}(F_1') = 1 \wedge \mathcal{P}(F_2') = 1\}$
6: **while** exists $q_1^{e_1} \in F_1$ with $q_1 \neq 1$ **do**
7: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \wedge$
$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2')) = 1 \wedge \gcd(q_1^{e_1}, \mathcal{P}(F_2')) = 1 \wedge q_1^{e_1} \in F_1\}$
8: $F_1 := F_1 \setminus \{q_1^{e_1}\}$
9: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}\} \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \wedge$
$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2')) = 1 \wedge \gcd(q_1^{e_1}, \mathcal{P}(F_2')) = 1\}$
10: **while** $q_1 \neq 1$ and exists $q_2^{e_2} \in F_2$ with $q_2 \neq 1$ **do**
11: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}\} \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \wedge$

$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2')) = 1 \wedge \gcd(q_1^{e_1}, \mathcal{P}(F_2')) = 1 \wedge q_2^{e_2} \in F_2\}$

12: $F_2 := F_2 \setminus \{q_2^{e_2}\}$

13: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}\} \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{q_2^{e_2}\} \wedge$
$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{q_2^{e_2}\})) = 1 \wedge \gcd(q_1^{e_1}, \mathcal{P}(F_2')) = 1\}$

14: **if** $\neg$irreducible$(q_1) \vee \neg$irreducible(q_2) **then** $p := \gcd(q_1, q_2)$

15: **else** $p := 1$

16: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}\} \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{q_2^{e_2}\} \wedge$
$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{q_2^{e_2}\})) = 1 \wedge \gcd(q_1^{e_1}, \mathcal{P}(F_2')) = 1 \wedge p = \gcd(q_1, q_2)\}$

17: **if** $p = 1$ **then**

18: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}\} \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{q_2^{e_2}\} \wedge$
$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{q_2^{e_2}\})) = 1 \wedge \gcd(q_1^{e_1}, \mathcal{P}(F_2')) = 1 \wedge \gcd(q_1, q_2) = 1\}$

19: $F_2' := F_2' \cdot_{\mathcal{F}} \{q_2^{e_2}\}$

20: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}\} \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \wedge$
$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2')) = 1 \wedge \gcd(q_1^{e_1}, \mathcal{P}(F_2')) = 1\}$

21: **else**

22: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}\} \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{q_2^{e_2}\} \wedge$
$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{q_2^{e_2}\})) = 1 \wedge \gcd(q_1^{e_1}, \mathcal{P}(F_2')) = 1 \wedge p = \gcd(q_1, q_2)\}$

23: $q_1 := \frac{q_1}{p}$

24: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \cdot_{\mathcal{F}} \{(q_1 \cdot p)^{e_1}\} \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{q_2^{e_2}\} \wedge$
$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{q_2^{e_2}\})) = 1 \wedge \gcd((q_1 \cdot p)^{e_1}, \mathcal{P}(F_2')) = 1 \wedge p = \gcd((q_1 \cdot p), q_2)\}$

25: $F_i := F_i \cdot_{\mathcal{F}} \{p^{e_i - \min(e_1, e_2)}\}$ for $i = 1, 2$

26: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}, p^{\min(e_1, e_2)}\} \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{(\frac{q_2}{p})^{e_2}, p^{\min(e_1, e_2)}\} \wedge$
$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{(\frac{q_2}{p})^{e_2}, p^{\min(e_1, e_2)}\})) = 1 \wedge \gcd((q_1 \cdot p)^{e_1}, \mathcal{P}(F_2')) = 1 \wedge$
$p = \gcd((q_1 \cdot p), q_2)\}$

27: $F_2' := F_2' \cdot_{\mathcal{F}} \{(\frac{q_2}{p})^{e_2}\}$

28: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}, p^{\min(e_1, e_2)}\} \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{p^{\min(e_1, e_2)}\} \wedge$
$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2' \cdot_{\mathcal{F}} \{p^{\min(e_1, e_2)}\})) = 1 \wedge \gcd((q_1 \cdot p)^{e_1}, \mathcal{P}(F_2')) = 1\}$

29: $G := G \cdot_{\mathcal{F}} \{p^{\min(e_1, e_2)}\}$

30: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}\} \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \wedge$
$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2')) = 1 \wedge \gcd(q_1^{e_1}, \mathcal{P}(F_2')) = 1\}$

31: **end if**

32: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}\} \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \wedge$
$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2')) = 1 \wedge \gcd(q_1^{e_1}, \mathcal{P}(F_2')) = 1\}$

33: **end while**

34: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}\} \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \wedge$
$\gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2')) = 1 \wedge \gcd(q_1^{e_1}, \mathcal{P}(F_2')) = 1 \wedge (q_1 = 1 \vee \mathcal{P}(F_2) = 1)$

35: $F_1' := F_1' \cdot_{\mathcal{F}} \{q_1^{e_1}\}$

36: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \cdot_{\mathcal{F}} F_2' \wedge \gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2 \cdot_{\mathcal{F}} F_2')) = 1\}$

37: $F_2 := F_2 \cdot_{\mathcal{F}} F_2'$

38: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \wedge \gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2)) = 1\}$

39: $F_2' := \emptyset$

40: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1 \cdot_{\mathcal{F}} F_1' \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \wedge \gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2)) = 1 \wedge \mathcal{P}(F_2') = 1\}$

41: **end while**

42: $\{\mathcal{F}_{p_1} = G \cdot_{\mathcal{F}} F_1' \wedge \mathcal{F}_{p_2} = G \cdot_{\mathcal{F}} F_2 \wedge \gcd(\mathcal{P}(F_1'), \mathcal{P}(F_2)) = 1\}$

43: **return** (F_1', F_2, G)

end

The above assertion network is inductive.

- For the assignments, their preconditions imply their postconditions after substituting the assigned expression for the assigned variables. (For simplicity, we handle the first if-then-else statement in lines (14)-(15) as atomic assignment as well.)
- For the if-then-else statement in lines (17)-(31), its precondition (16) implies the precondition (18) of the if-branch if the branching condition holds, and the precondition (22) of the else-branch if the condition does not hold. The postconditions (20) and (30) of both branches imply the postcondition (32) of the if-then-else statement.
- For the outer while-loop (6)-(41), its precondition (5) as well as the postcondition (40) of its body imply the precondition (7) of the body if the loop condition holds, and they both imply the postcondition (42) of the while-loop if the loop condition does not hold.
- The inner while-loop's inductivity can be shown similarly. □

That means, the assertion (42) always holds before returning, implying the correctness of the algorithm.

The algorithm is also complete, since it always terminates: We can use the sum of the degrees of all polynomials in F_1 for the outer loop as ranking function and in F_2 for the inner loop to show their termination.

8.3 Using factorized polynomials in rational functions

We represent a rational function $\frac{p_1}{p_2}$ by separate factorizations $\mathcal{F}_{p_1}$ and $\mathcal{F}_{p_2}$ for the numerator p_1 and the denominator p_2, respectively. For multiplication $\frac{p_1}{p_2} = \frac{q_1}{q_2} \cdot \frac{r_1}{r_2}$, we compute $\mathcal{F}_{p_1} = \mathcal{F}_{q_1} \cdot_{\mathcal{F}} \mathcal{F}_{r_1}$ and $\mathcal{F}_{p_2} = \mathcal{F}_{q_2} \cdot_{\mathcal{F}} \mathcal{F}_{r_2}$. Division is reduced to multiplication according to $\frac{q_1}{q_2} : \frac{r_1}{r_2} = \frac{q_1}{q_2} \cdot \frac{r_2}{r_1}$.

For the addition $\frac{p_1}{p_2} = \frac{q_1}{q_2} + \frac{r_1}{r_2}$, we compute p_2 with $\mathcal{F}_{p_2} = \mathcal{F}_{q_2} \cup_{\mathcal{F}} \mathcal{F}_{r_2}$ as a common multiple of q_2 and r_2, such that $p_2 = q_2 \cdot q_2'$ with $\mathcal{F}_{q_2'} = \mathcal{F}_{p_2} :_{\mathcal{F}} \mathcal{F}_{q_2}$, and $p_2 = r_2 \cdot r_2'$ with $\mathcal{F}_{r_2'} = \mathcal{F}_{p_2} :_{\mathcal{F}} \mathcal{F}_{r_2}$. For the numerator p_1 we first determine a common divisor s of q_1 and r_1 by $\mathcal{F}_s = \mathcal{F}_{q_1} \cap_{\mathcal{F}} \mathcal{F}_{r_1}$, such that $q_1 = s \cdot q_1'$ with $\mathcal{F}_{q_1'} = \mathcal{F}_{q_1} :_{\mathcal{F}} \mathcal{F}_s$, and $r_1 = s \cdot r_1'$ with $\mathcal{F}_{r_1'} = \mathcal{F}_{r_1} :_{\mathcal{F}} \mathcal{F}_s$. The numerator p_1 is $s \cdot (q_1' \cdot q_2' + r_1' \cdot r_2')$ with factorization $\mathcal{F}_s \cdot_{\mathcal{F}} (\mathcal{F}_{q_1'} \cdot_{\mathcal{F}} \mathcal{F}_{q_2'} +_{\mathcal{F}} \mathcal{F}_{r_1'} \cdot_{\mathcal{F}} \mathcal{F}_{r_2'})$.

The rational function $\frac{p_1}{p_2}$ resulting from the addition is further simplified by cancellation, i. e., dividing p_1 and p_2 by their greatest common divisor.

8.4 Experimental results

We developed a `C++` prototype implementation of our approach using the arithmetic library `GiNaC` [BFK02]. Moreover, we implemented the state-elimination approach used by PARAM [HHWZ10] using our optimized factorization approach to provide a more distinct comparison. All experiments were run on an Intel Core 2 Quad CPU 2.66 GHz with 4 GB of memory. We defined a timeout (TO) of 14 hours (50400 seconds) and a memory bound (MO) of 4 GB.

We report on three case studies:

- The *bounded retransmission protocol* (BRP) [HSV93] models the sending of files via an unreliable network, manifested in two lossy channels for sending and acknowledging the reception. This model is parametrized in the probability of reliability of those channels.
- The *crowds protocol* (CROWDS) [RR98] is designed for anonymous network communication using random routing, parametrized in how many members are "good" or "bad" and the probability of whether a good member delivers a message or randomly routes it to another member.
- *NAND multiplexing* (NAND) [HJ02] models how reliable computations are obtained using unreliable hardware by having a certain number of copies of a NAND unit all doing the same job. Parameters are the probabilities of faultiness of the units and of erroneous inputs.

In our experiments we compare the following two implementations with the tool PARAM:

STATE ELIM : This implementation uses the state elimination approach as it is implemented in PARAM and additionally uses the optimized factorization of polynomials as presented in Section 8.1 and Section 8.2. Comparing this implementation with PARAM shows how well the techniques of this chapter speed up the performance in general.

SCC MC : This implementation uses the SCC-based approach as briefly described in the beginning of this chapter and specified in detail in [3]. It thereby also uses the optimized factorization of polynomials and shows the quality of its applicability in a different setting.

Note that no bisimulation reduction was applied to any of the input models, which would improve the feasibility of all approaches likewise.

For all instances we list the number of states and transitions; for each tool we give the running time in seconds and the memory consumption in MB; the best time is **boldfaced**. Moreover, for our approaches we list the number of polynomials which are intermediately stored.

	Graph		SCC MC			STATE ELIM			PARAM	
Model	States	Trans.	Time	Poly	Mem	Time	Poly	Mem	Time	Mem
BRP	3528	4611	29.05	3283	48.10	**4.33**	8179	61.17	98.99	32.90
BRP	4361	5763	511.50	4247	501.71	**6.87**	9520	78.49	191.52	58.43
BRP	7048	9219	548.73	6547	281.86	**25.05**	16435	216.05	988.28	142.66
BRP	10759	13827	147.31	9231	176.89	**85.54**	26807	682.24	3511.96	304.07
BRP	21511	27651	1602.53	18443	776.48	**718.66**	53687	3134.59	34322.60	1757.12
CROWDS	198201	348349	**60.90**	13483	140.15	243.07	27340	133.91	46380.00	227.66
CROWDS	482979	728677	**35.06**	35916	478.85	247.75	65966	297.40	TO	—
CROWDS	726379	1283297	**223.24**	36649	515.61	1632.63	73704	477.10	TO	—
CROWDS	961499	1452537	**81.88**	61299	1027.78	646.76	112452	589.21	TO	—
CROWDS	1729494	2615272	**172.59**	97655	2372.35	1515.63	178885	1063.15	TO	—
CROWDS	2888763	5127151	**852.76**	110078	2345.06	12326.80	224747	2123.96	TO	—
NAND	7393	11207	8.35	15688	114.60	17.02	140057	255.13	**5.00**	10.67
NAND	14323	21567	39.71	25504	366.79	59.60	405069	926.33	**15.26**	16.89
NAND	21253	31927	100.32	35151	795.31	121.40	665584	2050.67	**29.51**	24.45
NAND	28183	42287	208.41	44799	1405.16	218.85	925324	3708.27	**50.45**	30.47
NAND	78334	121512	**639.29**	184799	3785.11	—	—	MO	1138.82	111.58

For BRP, STATE ELIM always outperforms PARAM and SCC MC by up to two orders of magnitude. On larger instances, SCC MC is faster than PARAM while on smaller ones PARAM is faster and has a smaller memory consumption.

In contrast, the crowds protocol always induces a nested SCC structure, which is very hard for PARAM since many divisions of polynomials have to be carried out. On larger benchmarks, it is therefore outperformed by more than three orders of magnitude while SCC MC performs best. This is actually measured by the timeout; using PARAM we could not retrieve results for larger instances.

To give an example where PARAM mostly performs better than our approaches, we consider NAND. Its graph is acyclic consisting mainly of single paths leading to states that have a high number of outgoing edges, i. e., many paths join at these states and diverge again. Together with a large number of different probabilities, this involves the addition of many polynomials, whose factorizations are completely stored. The SCC approach performs better here, as for acyclic graphs just the linear equation system is solved. This seems to be superior to the state elimination as implemented in our tool. We do not know about PARAM's interior for these special cases. As a solution, our implementation offers the possibility to limit the number of stored polynomials, which decreases the memory consumption at the price of losing information about the factorizations. However, an efficient strategy to manage this bounded pool of polynomials is not yet implemented. Therefore, we refrain from presenting experimental results for this scenario.

CHAPTER 9

Conclusion

Using the example of the virtual substitution, this thesis demonstrated the challenges we have to cope with when adapting a single procedure for nonlinear real arithmetic to SMT solving. As a side effect it also yields a deeper analysis of the procedure towards satisfiability checking, which helped us to provide rather general contributions, such as the detection of infeasible subsets or local conflicts and the employment of variable bounds. Moreover, we could make use of a commonly applied technique in SMT solving, branch-and-bound, in order to enable the utilization of the virtual substitution for integer arithmetic.

We experimentally evaluated the single techniques, showing that each of them improves the performance of an SMT solver with a theory solver based on the virtual substitution for most of the tested benchmark sets. We also compared our implementation to another state-of-the-art tool, which is purely based on virtual substitution, emphasizing that we can solve many more of the tested examples and this within far less time.

The presented heuristics for choosing the next variable to eliminate or next constraint to provide a test candidate for, have an immense influence on the performance of the virtual substitution. With more analysis of the solving process for examples, which could not be solved within the timeout, we might detect specific adaptions of these heuristics such that it possible to determine the satisfiability of the given example within this timeout.

One of the main messages of this thesis is that there is no overall best performing approach for SMT solving of nonlinear arithmetic formulas. Therefore, it is essential to be able to choose from a set of different procedures and optimally combine them according to some solving strategy. Within the work in the context of this thesis, we contributed the toolbox `SMT-RAT`, which provides exactly what we need for this purpose. The experimental results showed that, although the SMT solver Z3 implements an approach, which seems to be the most performant alternative for most of the used benchmark sets, for some of them we can solve more instances with other approaches.

[ÁAB+16] Erika Ábrahám, John Abbott, Bernd Becker, Anna M. Bigatti, Martin Brain, Bruno Buchberger, Alessandro Cimatti, James H. Davenport, Matthew England, Pascal Fontaine, Stephen Forrest, Alberto Griggio, Daniel Kroening, Werner M. Seiler, and Thomas Sturm. SC^2: Satisfiability checking meets symbolic computation. In *Proc. of CICM*, volume 9791 of *LNCS*, pages 28–43. Springer, 2016.

[ABP+11] Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret. Satisfiability modulo theories: An efficient approach for the resource-constrained project scheduling problem. In *Proc. of SARA*. AAAI, 2011.

[ACH+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3 – 34, 1995.

[ADFO13] Carlos Areces, David Deharbe, Pascal Fontaine, and Ezequiel Orbe. SyMT: Finding symmetries in SMT formulas. In *Proc. of SMT*, 2013.

[ÁFSW16] Erika Ábrahám, Pascal Fontaine, Thomas Sturm, and Dongming Wang. Symbolic Computation and Satisfiability Checking (Dagstuhl Seminar 15471). *Dagstuhl Reports*, 5(11):71–89, 2016.

[ÁJW+10] Erika Ábrahám, Nils Jansen, Ralf Wimmer, Joost-Pieter Katoen, and Bernd Becker. DTMC model checking by SCC reduction. In *Proc. of QEST*, pages 37–46. IEEE Computer Society, 2010.

[AP10] Behzad Akbarpour and Lawrence C. Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010.

[BBC+05] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter Rossum, Stephan Schulz, and Roberto Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Proc. of TACAS*, volume 3440 of *LNCS*, pages 317–333. Springer, 2005.

[BBP13] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.

[BCBdODF09] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient SMT-solver. In *Proc. of CADE*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proc. of CAV*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[BDS02] Clark Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proc. of CAV*, volume 2404 of *LNCS*, pages 236–249. Springer, 2002.

[BFK02] Christian Bauer, Alexander Frink, and Richard Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *Journal of Symbolic Computation*, 33(1):1–12, 2002.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[BGMG15] Daniel Bryce, Sicun Gao, David Musliner, and Robert Goldman. SMT-based nonlinear PDDL+ planning. In *Proc. of AAAI*, pages 3247–3253. AAAI Press, 2015.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[BKM14] Clark Barrett, Daniel Kroening, and Thomas Melham. Problem Solving for the 21st Century: Efficient Solvers for Satisfiability Modulo Theories. Technical Report 3, London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering, 2014. Knowledge Transfer Report.

[BNOT06] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on demand in SAT modulo theories. In *Proc. of LPAR*, volume 4246, pages 512–526. Springer, 2006.

[BPST10] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT solver. In *Proc. of TACAS*, volume 6015 of *LNCS*, pages 150–153. Springer, 2010.

[Buc65] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, 1965.

[Bur98] Stanley Burris. *Logic for Mathematics and Computer Science*. Prentice Hall, 1998.

[BV02] Randal E. Bryant and Miroslav N. Velev. Boolean satisfiability with transitivity constraints. *ACM Transactions on Computational Logic*, 3(4):604–627, 2002.

[BWK93] Thomas Becker, Volker Weispfenning, and Heinz Kredel. *Gröbner bases: A Computational Approach to Commutative Algebra*. Graduate texts in mathematics. Springer, 1993.

[CAMN04] Scott Cotton, Eugene Asarin, Oded Maler, and Peter Niebert. Some progress in satisfiability checking for difference logic. In *Proc. of FORMATS*, volume 3253 of *LNCS*, pages 263–276. Springer, 2004.

[CES+09] Koen Claessen, Niklas Eén, Mary Sheeran, Niklas Sörensson, Alexey Voronov, and Knut Åkesson. SAT-solving in practice, with a tutorial example from supervisory control. *Discrete Event Dynamic Systems*, 19(4):495–524, 2009.

[CGMT15] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Hycomp: An SMT-based model checker for hybrid systems. In *Proc. of TACAS*, volume 9035 of *LNCS*, pages 52–67. Springer, 2015.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *Proc. of TACAS*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In *Proc. of SPIN*, volume 7385 of *LNCS*, pages 248–254. Springer, 2012.

[CMT12] Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. A quantifier-free SMT encoding of non-linear hybrid automata. In *Proc. of FMCAD*, pages 187–195. IEEE, 2012.

[Col75] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 134–183. Springer, 1975.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proc. of STOC*, pages 151–158. ACM Press, 1971.

[Cor10] Florian Corzilius. Virtual Substitution in SMT Solving. Master's thesis (Diplomarbeit), RWTH Aachen University, 2010.

[CTF00] Alberto Caprara, Paolo Toth, and Matteo Fischetti. Algorithms for the set covering problem. *Annals of Operations Research*, 98(1):353–371, 2000.

[Dan63] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[Daw04] Conrado Daws. Symbolic and parametric model checking of discrete-time Markov chains. In *Proc. of ICTAC*, volume 3407 of *LNCS*, pages 280–294. Springer, 2004.

[DDA09] Isil Dillig, Thomas Dillig, and Alex Aiken. Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In *Proc. of CAV*, volume 5643 of *LNCS*, pages 233–247. Springer, 2009.

[DdM06] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. of CAV*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.

[DFMP11] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In *Proc. of CADE*, volume 6803 of *LNCS*, pages 222–236. Springer, 2011.

[DH88] James H. Davenport and Joos Heinz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5(1/2):29–35, 1988.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DLT16] Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *STTT*, 18(2):205–225, 2016.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[dMP09] Leonardo de Moura and Grant Olney Passmore. On locally minimal Nullstellensatz proofs. In *Proc. of SMT*, pages 35–42. ACM Press, 2009.

[dMP13] Leonardo de Moura and Grant Olney Passmore. The strategy challenge in SMT solving. In *Automated Reasoning and Mathematics*, volume 7788 of *LNCS*, pages 15–44. Springer, 2013.

[dMR02] Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. In *Proc. of SAT*, pages 244–251, 2002.

[dMRS02] Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proc. of CADE*, volume 2392 of *LNCS*, pages 438–455. Springer, 2002.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[DS97] Andreas Dolzmann and Thomas Sturm. REDLOG: Computer algebra meets computer logic. *SIGSAM Bulletin*, 31(2):2–9, 1997.

[DSW98] Andreas Dolzmann, Thomas Sturm, and Volker Weispfenning. Real quantifier elimination in practice. In *Algorithmic Algebra and Number Theory*, pages 221–247. Springer, 1998.

[Dut14] Bruno Dutertre. Yices 2.2. In *Proc. of CAV*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014.

[Erk13] Christoph Erkinger. Rotating Workforce Scheduling as Satisfiability Modulo Theories. Master's thesis (Diplomarbeit), Technische Universität Wien, 2013.

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proc. of SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[FGM+07] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, Réne Thiemann, and Harald Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. of SAT*, volume 4501 of *LNCS*, pages 340–354. Springer, 2007.

[FHT+07] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1(3-4):209–236, 2007.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. of PLDI*, LNCS, pages 234–245. ACM Press, 2002.

[Fou26] Jean-Baptiste Joseph Fourier. Solution d'une question particulière du calcul des inégalités. *Oeuvres II*, pages 317–328, 1826.

[GBE+14] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Steffi Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In *Proc. of IJCAR*, volume 8562 of *LNCS*, pages 184–191. Springer, 2014.

[GGI+10] Sicun Gao, Malay K. Ganai, Franjo Ivancic, Aarti Gupta, Sriram Sankaranarayanan, and Edmund M. Clarke. Integrating ICP and LRA solvers for deciding nonlinear real arithmetic problems. In *Proc. of FMCAD*, pages 81–89. IEEE, 2010.

[GKC13] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In *Proc. of CADE*, volume 7898 of *LNCS*, pages 208–214. Springer, 2013.

[Göd31] Kurt Gödel. Über formal Unentscheidbare Sätze der Principia Mathematica und Verwandter Systeme. *Monatshefte für Math. u. Physik*, 38:173–198, 1931.

[Gri09] Alberto Griggio. *An Effective SMT Engine for Formal Verification*. PhD thesis, DISI - University of Trento, 2009.

[Gri12] Alberto Griggio. A practical approach to satisfiability modulo linear integer arithmetic. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:1–27, Januar 2012.

[GT09] Chrysida Galanaki and Elias Tsigaridas. Quantifier elimination for small degree polyomials. In *Proc. of PLS*, 2009.

[Han09] *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[HHWZ10] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. PARAM: A model checker for parametric Markov models. In *Proc. of CAV*, volume 6174 of *LNCS*, pages 660–664. Springer, 2010.

[HHZ11] Ernst Moritz Hahn, Holger Hermanns, and Lijun Zhang. Probabilistic reachability for parametric Markov models. *STTT*, 13(1):3–19, 2011.

[Hil02] David Hilbert. Mathematical problems. *Bulletin of the American Mathematical Society*, 8(10):437–479, 1902.

[HJ02] Jie Han and Pieter Jonker. A system architecture solution for unreliable nanoelectronic devices. *IEEE Transactions on Nanotechnology*, 1:201–208, 2002.

[Hon91] Hoon Hong. Comparison of Several Decision Algorithms for the Existential Theory of the Reals. Technical Report 91-41, Research Institute for Symbolic Computation, Johannes Kepler University Linz, 1991.

[HR97] Stefan Herbort and Dietmar Ratz. Improving the Efficiency of a Nonlinear-System-Solver Using a Componentwise Newton Method. Technical Report 151241, Institut für Angewandte Mathematik, Universität Karslruhe (TH), 1997.

[HSV93] Leen Helmink, Alex Sellink, and Frits W. Vaandrager. Proof-checking a data link protocol. In *Proc. of TYPES*, volume 806 of *LNCS*, pages 127–165. Springer, 1993.

[JBdM13] Dejan Jovanovic, Clark Barrett, and Leonardo de Moura. The design and implementation of the model constructing satisfiability calculus. In *Proc. of FMCAD*, pages 173–180. IEEE, 2013.

[JBRS12] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33:89–92, 2012.

[JdM12] Dejan Jovanovic and Leonardo de Moura. Solving non-linear arithmetic. In *Proc. of IJCAR*, volume 7364 of *LNCS*, pages 339–354. Springer, 2012.

[KBD13] Tim King, Clark Barrett, and Bruno Dutertre. Simplex with sum of infeasibilities for SMT. In *Proc. of FMCAD*, pages 189–196. IEEE, 2013.

[KBT14] Tim King, Clark Barrett, and Cesare Tinelli. Leveraging linear and mixed integer programming for SMT. In *Proc. of FMCAD*, pages 139–146. IEEE, 2014.

[Kha80] Leonid Genrikhovich Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53 – 72, 1980.

[KS15] Marek Košta and Thomas Sturm. A generalized framework for virtual substitution. *CoRR*, abs/1501.05826, 2015.

[KSD16] Marek Košta, Thomas Sturm, and Andreas Dolzmann. Better answers to real questions. *Journal of Symbolic Computation*, 74:255 – 275, 2016.

[Kul09] Ulrich W. Kulisch. Complete interval arithmetic and its implementation on the computer. In *Numerical Validation in Current Hardware Architectures: International Dagstuhl Seminar, Dagstuhl Castle, Germany, January 6-11, 2008. Revised Papers*, volume 5492 of *LNCS*, pages 7–26. Springer, 2009.

[Lar92] Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 11(1):4–15, 1992.

[Lei13] K. Rustan M. Leino. Automating theorem proving with SMT. In *Proc. of ITP*, volume 7998 of *LNCS*, pages 2–16. Springer, 2013.

[LW93] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *Computer Journal*, 36(5):450–462, 1993.

[Mat70] Yuri V Matiyasevich. Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR*, 191(2):279–282, 1970.

[Mat72] Yuri V. Matiyasevich. Diophantine representation of enumerable predicates. *Mathematical notes of the Academy of Sciences of the USSR*, 12(1):501–504, 1972.

[MKC09] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC*, pages 530–535. ACM Press, 2001.

[MS08] Joao P. Marques-Silva. Practical applications of Boolean satisfiability. In *Proc. of WODES*, pages 74–80. IEEE, 2008.

[MSS99] Joao P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *Proc. of RTA*, volume 3467 of *LNCS*, pages 453–468. Springer, 2005.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[NPM+14] Srinivas Nedunuri, Sailesh Prabhu, Mark Moll, Swarat Chaudhuri, and Lydia E. Kavraki. SMT-based synthesis of integrated task and motion plans from plan outlines. In *Proc. of ICRA*, pages 655–662. IEEE, 2014.

[NW88] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Wiley Interscience Series in Discrete Mathematics and Optimization. Wiley, 1988.

[PC13] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Proc. of CAV*, volume 8044 of *LNCS*, pages 53–68. Springer, 2013.

[Pel13] Jan Peleska. Industrial-strength model-based testing - state of the art and current challenges. In *Proc. of MBT*, volume 111 of *EPTCS*, pages 3–28, 2013.

[PQR09] André Platzer, Jan-David Quesel, and Philipp Rümmer. Real world verification. In *Proc. of CADE*, volume 5663 of *LNCS*, pages 485–501. Springer, 2009.

[Pug91] William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proc. of ACM/IEEE Conference on Supercomputing*, pages 4–13. ACM Press, 1991.

[PVL11] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with SMT-solving and abstract interpretation. In *Proc. of NFM*, volume 6617 of *LNCS*, pages 298–312. Springer, 2011.

[RR98] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[RS15] Hadi Ravanbakhsh and Sriram Sankaranarayanan. Counterexample guided synthesis of switched controllers for reach-while-stay properties. *CoRR*, abs/1505.01180, 2015.

[RT03] Silvio Ranise and Cesare Tinelli. The SMT-LIB format: An initial proposal. In *Proc. of PDPAR*, pages 94–111, 2003.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

[Seb07] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.

[SGF10] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proc. of POPL*, pages 313–326. ACM Press, 2010.

[Sho79] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, 1979.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[SKB13] Karsten Scheibler, Stefan Kupferschmid, and Bernd Becker. Recent improvements in the SMT solver iSAT. In *Proc. of MBMV*, pages 231–241. Institut für Angewandte Mikroelektronik und Datentechnik, Fakultät für Informatik und Elektrotechnik, Universität Rostock, 2013.

[SOE14] Roopsha Samanta, Oswaldo Olivo, and E. Allen Emerson. Cost-aware automatic program repair. In *Proc. of SAS*, volume 8723 of *LNCS*, pages 268–284. Springer, 2014.

[SSB02] Ofer Strichman, Sanjit A. Seshia, and Randal E. Bryant. Deciding separation formulas with SAT. In *Proc. of CAV*, volume 2404 of *LNCS*, pages 209–222. Springer, 2002.

[Tar48] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1948.

[TPGM14] Pranav Tendulkar, Peter Poplavko, Ioannis Galanommatis, and Oded Maler. Many-core scheduling of data parallel applications using SMT solvers. In *Proc. of DSD*, pages 615–622. IEEE Computer Society, 2014.

[Tse83] Grigorii S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer, 1983.

[TVKO16] Vu Xuan Tung, To Van Khanh, and Mizuhito Ogawa. raSAT: An SMT solver for polynomial constraints. In *Proc. of IJCAR*, volume 9706 of *LNCS*, pages 228–237. Springer, 2016.

[Wei94] Volker Weispfenning. Quantifier elimination for real: the cubic case. In *Proc. of ISSAC*, pages 258–263. ACM Press, 1994.

[Wei97] Volker Weispfenning. Quantifier elimination for real algebra - the quadratic case and beyond. *Applicable Algebra in Engineering, Communication and Computing*, 8(2):85–101, 1997.

[ZM10] Harald Zankl and Aart Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In *Proc. of LPAR*, volume 6355 of *LNCS*, pages 481–500. Springer, 2010.

Index