Virtual Substitution for SMT-Solving

Florian Corzilius and Erika Ábrahám

RWTH Aachen University, Germany

Abstract. SMT-solving aims at deciding satisfiability for the existential fragment of a first-order theory. A SAT-solver handles the logical part of a given problem and invokes an embedded theory solver to check consistency of theory constraints. For efficiency, the theory solver should be able to work *incrementally* and generate *infeasible subsets.* Currently available decision procedures for *real algebra* – the first-order theory of the reals with addition and multiplication – do not exhibit these features. In this paper we present an adaptation of the virtual substitution method, providing these abilities.

1 Introduction

The *satisfiability problem* poses the question of whether a given logical formula is satisfiable, i.e., whether we can assign values to the variables contained in the formula such that the formula becomes true. The development of efficient algorithms and tools (*solvers*) for satisfiability checking form an active research area in computer science. A lot of effort has been put into the development of fast solvers for the propositional satisfiability problem, called SAT. To increase expressiveness, extensions of the propositional logic with respect to first-order *theories* can be considered. The corresponding satisfiability problems are called *SAT-modulo-theories* problems, short *SMT.* SMT-solvers exist, e.g., for equality logic, uninterpreted functions, predicate logic, and linear real arithmetic.

In contrast to the above-mentioned theories, less activity can be observed for SMT-solvers supporting the first-order theory of the real ordered field, what we call *real algebra.* Our research goal is to develop an SMT-solver for real algebra, being capable of solving Boolean combinations of polynomial constraints over the reals efficiently.

Even though decidability of real algebra has been known for a long time [Tar48], the first decision procedures were not yet practicable. Since 1974 it is known that the time complexity of deciding formulas of real algebra is in worst case doubly exponential in the number of variables (dimension) contained in the formula [DH88,Wei88].

Today, several methods are available which satisfy these complexity bounds, for example the cylindrical algebraic decomposition (CAD) [CJ98] , the Gröbner basis, and the virtual substitution method [Wei98]. An overview of these methods is given in [DSW97]. There are also tools available which implement these methods. The stand-alone application `QEPCAD` is a `C++` implementation of the CAD method [Bro03]. Another example is the `REDLOG` package [DS97] of the computer algebra system `REDUCE` based on `Lisp`, which offers an optimized combination of the virtual substitution, the CAD method, and real root counting.

Currently existing solvers are not suited to solve large formulas containing arbitrary combinations of real constraints. We want to combine the advantages of highly tuned SAT-solvers and the most efficient techniques currently available for solving conjunctions of real constraints, by implementing an SMT-solver for real algebra capable of efficiently solving arbitrary Boolean combinations of real constraints.

Theory solvers should satisfy specific requirements in order to embed them into an SMT-solver efficiently:

- *Incrementality:* The theory solver should be able to accept theory constraints one after the other. After it receives a new theory constraint it should check the conjunction of the new constraint with the earlier constraints for satisfiability. For efficiency it is important that the solver makes use of the result of earlier checks.
- *(Minimal) infeasible subsets:* If the theory solver detects a conflict, it should give a reason for the unsatisfiability. The usual way is to determine an unsatisfiable subset of the constraints which is ideally minimal in the sense that if we remove a constraint the remaining ones become satisfiable.
- *Backjumping:* If a conflict occurs, either in the Boolean or in the theory domain, the solver should be able to backtrack and continue the search for a solution at an earlier state, thereby reducing the search space as far as possible.

To our knowledge, these functionalities are currently not supported by the available solvers for real algebra. In this paper we extend the virtual substitution method to support incrementality, backjumping, and the generation of infeasible subsets.

We have chosen the virtual substitution method because it is a restricted but very efficient decision procedure for a subset of real algebra. The restriction concerns the degree of polynomials. The method uses solution equations to determine the zeros of (multivariate) polynomials in a given variable. As such solution equations exist for polynomials of degree at most 4, the method is a priori restricted in the degree of polynomials. In this paper we restrict ourselves to polynomials of degree 2, however it is possible to extend our approach to polynomials up to degree 4. Furthermore, we want to develop an incremental adaptation of the CAD method and to call this complete[1] but less efficient decision procedure in order to complete the incremental implementation of the virtual substitution method. This will be part of our future work.

Related work. The SMT-solvers `Z3` [dMB08], `HySAT` [FHT$^+$07] and `ABsolver` [BPT07] are able to handle nonlinear real-algebraic constraints. The algorithm implemented in `HySAT` and in its successor tool `iSAT` uses interval constraint

[1] The CAD method can handle full real algebra.

propagation to check real constraints. Though this technique is not complete, in practice it is more efficient than those based on exact methods [FHT$^+$07]. The structures of `ABsolver` and `Z3` are more similar to our approach. However, `Z3` does not support full real-algebra. Papers on `ABsolver` do not address the issue of incrementality. Though `ABsolver` computes minimal infeasible subsets, we did not find any information on how they are generated. Other approaches as, e.g., implemented in the RAHD tool [PJ09], embed real-algebraic decision procedures in a theorem proving context. The virtual substitution was also used in its original form for temporal verification within the Stanford Temporal Solver [Bjo99]. The virtual substitution is used for linear arithmetic in [Bjo10].

The remaining part of the paper is structured as follows: We give an introduction to DPLL-based SMT-solving and to virtual substitution in Section 2. We introduce our incremental virtual substitution algorithm providing infeasible subsets and give some first experimental results in Section 3. We conclude the paper in Section 4. A more detailed description of our work can be found in [Cor11].

2 Preliminaries

In this paper we focus on satisfiability checking for a subset of the existential fragment of real algebra (quadratic and beyond). *Terms* or *polynomials* p, *constraints* c, and *formulas* φ can be built upon constants $0, 1$ and real-valued variables x according to the following abstract grammar:

$$\begin{array}{rcl} p & ::= & 0 \mid 1 \mid x \mid (p+p) \mid (p \cdot p) \\ c & ::= & p = p \mid p < p \\ \varphi & ::= & c \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\exists x \varphi) \end{array}$$

Syntactic sugar like *True* , *False* , $-$, $/$, $\vee$, $\rightarrow$, $\forall$, ... is defined as usual; the equality is added for convenience but could also be defined as syntactic sugar. We define that $\vee$ binds stronger than $\wedge$, and $\wedge$ binds stronger than $\exists$, and sometimes skip the parentheses. The semantics of real algebra is as expected. We call a variable x occurring in a formula $\exists x \varphi$ *bound*; not bound variables are called *free*. Formulas with no free variables are called *sentences*. With $\mathbb{R}[x_1, \ldots, x_n]$ we denote the set of all polynomials containing variables $x_1, \ldots, x_n$.

With the real numbers $\mathbb{R}$ as the domain, the set of all true real-algebraic sentences is the first-order theory of $(\mathbb{R}, +, \cdot, 0, 1, <)$, called *real algebra*. In this paper we restrict to the existential fragment, i.e., to formulas which can be transformed into the form $\exists x_1 \ldots \exists x_n \varphi$ with φ being quantifier-free.

The satisfiability problem for real-algebraic formulas is decidable as proved around the 1930s [Tar48]. We use DPLL-based SMT-solving, introduced in Section 2.1, for the satisfiability check. An SMT-solver combines a SAT-solver, handling the Boolean structure, and a theory solver to check the theory constraints. We apply the *virtual substitution method*, introduced in Section 3, as an algorithm for the theory solver, which is very efficient but restricted in the degree of polynomials that can be handled.

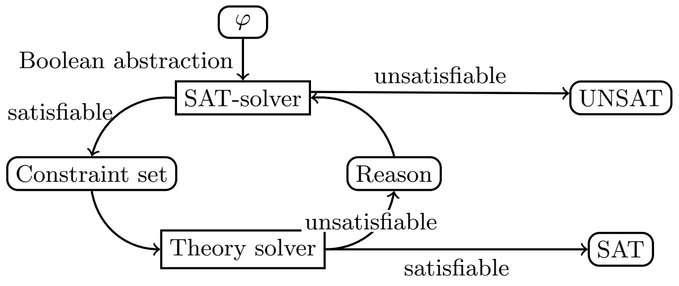

Fig. 1. The basic scheme of DPLL-based full lazy SMT-solving

2.1 Lazy SMT-solving

The propositional satisfiability problem (SAT) where the variables range over the values 1 (*True*) and 0 (*False*) is NP-complete, but SAT-solvers are quite efficient in practice due to a vast progress in SAT-solving during recent years. One of the main achievements in the field of SAT-solving is the DPLL-algorithm, which is capable of solving existential Boolean formulas.

The DPLL-algorithm can be extended to handle more expressive logics. For this approach, which is called *SAT-modulo-theories* (*SMT*) solving, a SAT-solver gets combined with a decision procedure for the satisfiability check of constraints from the underlying theory (see, e.g., [KS08]).

The basic scheme of *full lazy* DPLL-based SMT-solving is roughly as follows (see Fig. 1). The SMT-solver first creates a Boolean skeleton of the input formula, replacing all theory constraints contained in the input formula by fresh Boolean variables. The resulting Boolean formula is passed to the SAT-solver, which searches for a satisfying assignment. If it does not succeed, the formula is unsatisfiable. Otherwise, the assignment found corresponds to certain truth values for the theory constraints and has to be verified by the theory solver. If the constraints are satisfiable, then the original formula is satisfiable. Otherwise, if the theory solver detects that the conjunction of the corresponding theory constraints is unsatisfiable, it then hands over a reason for the unsatisfiability, an *infeasible subset* of the theory constraints, to the SAT-solver. The SAT-solver uses this piece of information to exclude the detected conflict from further search. Afterwards, the SAT-solver computes an assignment for the refined Boolean problem again, which in turn has to be verified by the theory solver. Continuing this iteration decides the satisfiability of the input formula in the end.

The full lazy procedure is often disadvantageous in practice, because the SAT-solver may do a lot of needless work by extending an already (in the theory domain) contradictory partial assignment. *Less lazy* variants of the procedure call the theory solver more often, already handing over constraints corresponding to partial assignments. To do so efficiently, the theory solver should accept constraints in an *incremental* fashion, where computation results of previous steps can be reused.

Note that we strictly separate the satisfiability checks in the Boolean and in the theory domains, that means, we do not consider theory propagation embedded in the DPLL search like, e.g., Yices does.

2.2 Virtual substitution

The virtual substitution method is a restricted but very efficient decision procedure for a subset of real algebra. In this paper we adapt it to support incrementality and infeasible subset generation (see Section 3). In this section we introduce virtual substitution in its original form.

We are interested in checking satisfiability of existentially quantified formulas in prenex normal form (PNF) $\exists x_1 \ldots \exists x_n \varphi$ with φ quantifier-free. The decision procedure based on virtual substitution produces a quantifier-free equivalent of a given input formula, by successively eliminating all bound variables starting with the innermost one. Below we explain how the innermost quantified variable is eliminated using virtual substitution.

Let $\exists y_1 \ldots \exists y_n \exists x \varphi$ be the input formula, where φ is a quantifier-free Boolean combination of polynomial constraints. In this paper we handle constraints, whose degree in x is at most two ($y_1, \ldots,\ y_n$ may occur with higher degree). Thus we assume that all polynomial constraints in φ are of the form $p \sim 0$, $\sim \in \{=,\ <,\ >,\ \leq,\ \geq,\ \neq\}$, where $p = ax^2 + bx + c$ is at most quadratic[2] in x.

Considering the problems domain, each constraint containing x splits it into values which satisfy the constraint and values which do not. More precisely, the satisfying values can be merged to a finite number of intervals whose endpoints are elements of $\{\infty, -\infty\} \cup \mathbb{L}_x$, where $\mathbb{L}_x$ are the zeros of p in x:

$$\begin{array}{lll}
\text{Linear in } x: & x_0 = -\frac{c}{b} & \text{, if } a = 0 \ \wedge \ b \neq 0 \\
\text{Quadratic in } x\text{, first solution:} & x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} & \text{, if } a \neq 0 \ \wedge \ b^2 - 4ac \geq 0 \\
\text{Quadratic in } x\text{, second solution:} & x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a} & \text{, if } a \neq 0 \ \wedge \ b^2 - 4ac \geq 0
\end{array}$$

The conditions on the right are called *side conditions*. Note that in the case $a = b = c = 0$ (constant in x) the solution interval for x is $(-\infty,\ \infty)$, which does not have finite endpoints.

Assume that $\{c_1, \ldots,\ c_n\}$ is the set of constraints in φ that contain x. Each constraint c_i, $1 \leq i \leq n$, has a set of solution intervals $\{I_{i,1}, \ldots,\ I_{i,k_i}\}$ for x. If the constraints have a common solution for x, then for all $i \in \{1, \ldots,\ n\}$ there exists a $j_i \in \{1, \ldots,\ k_i\}$ with

$$I \ = \ \Big(\bigcap_{i \in \{1,\ldots,\ n\}} I_{i,j_i} \Big) \ \neq \ \emptyset.$$

The intersection I is an interval, whose endpoints are both endpoints of some of the intervals we intersect. If I is left-closed, its left endpoint is in I; otherwise

[2] The coefficients of x are again polynomials, but do not contain x.

there exists an infinitesimal value we can add to the left endpoint, such that the result is an element of I. In both cases we found an element of I being a solution for x. Candidates for the left endpoint of I are the left endpoints of the possible solution intervals, i.e. $-\infty$ and all finite endpoints $x_0,\ x_1,\ x_2$ for all constraints. When searching for a satisfying solution for x, it is sufficient to consider those candidates if they belong to a left-closed interval, or those candidates plus an infinitesimal if the corresponding interval is left-open.

In other words, we check if (1) one of the left endpoints of the left-closed solution intervals, or (2) one of the left endpoints plus an infinitesimal ϵ of the left-opened intervals or (3) a very small value, which we denote by $-\infty$, fulfills all constraints. We call those points belonging to (1), (2), or (3) *test candidates*. For the proof of soundness and completeness[3] of the method see [Wei88].

Basically, the virtual substitution recursively eliminates all bound variables x in φ by (i) determining all test candidates for x in all constraints in φ containing x, and (ii) checking if one of these test candidates satisfies φ.

To check whether a test candidate t for x satisfies a constraint $\overline{p} \sim 0$ in φ, we substitute all occurrences of x by t in $\overline{p}$, yielding $\overline{p}[t/x] \sim 0$, and check the resulting constraint under the test candidate's side conditions for consistency. Note that neither $\overline{p}[t/x] \sim 0$ nor the solution conditions refer to x, but they may contain other bound variables. Thus the consistency check may involve further quantifier eliminations.

Standard substitution could lead to terms not contained in real algebra, since the test candidates include $-\infty$, square roots, and infinitesimals ϵ. Virtual substitution however, avoids these expressions in the resulting terms: it defines substitution rules yielding formulas of real algebra that are equivalent to the result of the standard substitution. However, these substitution rules may increase the degree of the polynomials.

Assume T is the set of all possible test candidates for x. Given a test candidate $t \in T$ with side conditions C_t, the virtual substitution method applies the substitution rules to all constraints in the input formula φ and conjugates the result with C_t. Considering all possible test candidates results in the formula

$$\exists y_1 \ldots \exists y_n \bigvee_{t \in T} (\varphi[t/x] \wedge C_t).$$

Note that test candidates of type (3) do not have side conditions. The virtual substitution method continues with the elimination of the next variable.

3 The Adapted Virtual Substitution Method

As discussed in Section 1, a theory solver should support incrementality, infeasible subset generation, and backjumping in order to be suited for an efficient embedding into a less lazy SMT-solver.

[3] for the restricted logic with quadratic constraints

The original virtual substitution method does not provide these functionalities yet. Nevertheless, it can be embedded into an SMT-solver. Full lazy SMT-solving does not require incrementality, but is not very profitable compared to a less lazy approach with an incremental theory solver. We could also embed a non-incremental theory solver into a less lazy SMT-solver. However, in this case the theory solver has to re-do a lot of work.

In this section we propose an incremental version of the virtual substitution method, which is also able to generate an infeasible subset of the checked constraints in case they are inconsistent. We show, by way of an example, how the consistency check works, and give first experimental results using our prototype implementation.

3.1 Data structure of the theory solver

Assume that the theory solver receives some constraints $\varphi = c_1, \ldots, c_n$. The algorithm chooses a variable x to eliminate first and computes the set T of all test candidates that the constraints provide for x (see Section 2). The formula φ is satisfied if there is a test candidate $t \in T$ such that $\varphi_t = c_1[t/x] \wedge \ldots \wedge c_n[t/x] \wedge C_t$ is satisfiable, where C_t are the side conditions of t. In contrast to the original algorithm, we proceed using a *depth-first search*, first checking consistency for a single test candidate t as long as φ_t does not turn out to be inconsistent, and switch to another test candidate otherwise. Applying the substitution $c_i[t/x]$, $1 \leq i \leq n$, may lead to further case distinctions from which we can again choose a first branch until it turns out to be unsatisfiable. If we succeed to eliminate all variables without getting any conflict in a certain branch, then the problem is satisfiable and we stop the search. Otherwise, if all branches lead to conflicts, the problem is unsatisfiable.

This search structure can be represented by a tree as shown in the examples of Figure 2. The framed nodes of this tree contain conjunctions of real constraints and are called *conjunction nodes*. The other nodes are either indexed substitutions, called *substitution nodes*, or *conflict nodes*. The indices of the substitutions are the side conditions of the test candidate it considers and the labels on the edges to a substitution are the constraints, which provided the substitutions test candidate. The successors of a node are formed as follows: Let us consider a conjunction node N containing the constraints $c_1, \ldots, c_n$. Let x and T be as defined before. For each $t \in T$ there exists at most one successor of N being the substitution $[t/x]$ indexed by the side conditions of t. The successors of a substitution node $[t/x]_{C_t}$ are the substitution cases as explained above (*). If all of these cases contain a variable-free inconsistent constraint, than the substitution node has a conflict node as successor (**). We remove variable-free consistent constraints from the constraint sets. The index of a constraint in a conjunction node refers to the constraint in the root of the tree, from which it stems. For a constraint c indexed by c_i it means, that we got c from c_i by applying substitutions. We call c_i the *original constraint* of c. The index of a conflict node is formed by a disjunction of the original constraints of the constraints, which caused this conflict. It means, that applying the substitution of the antecessor

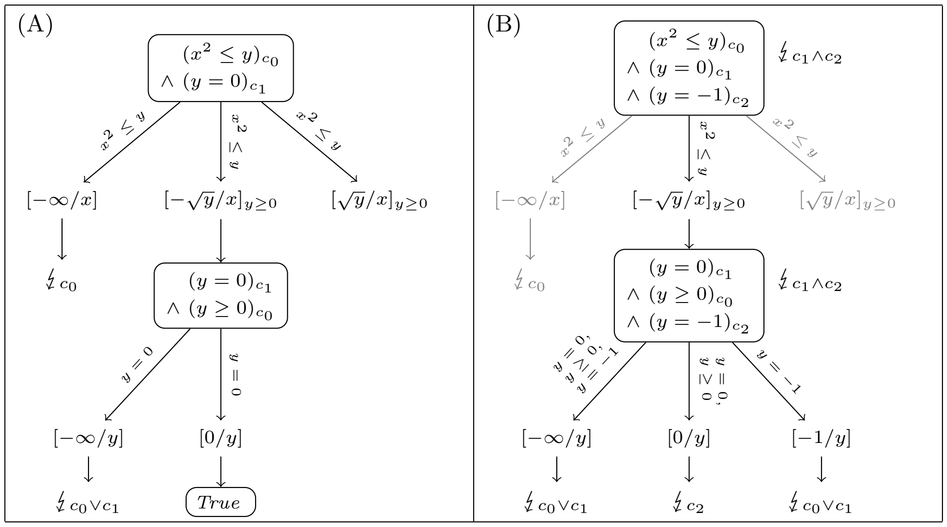
(A)
$(x^2 \leq y)_{c_0} \wedge (y = 0)_{c_1}$
$x^2 \leq y$
$[-\infty/x]$
$[-\sqrt{y}/x]_{y\geq 0}$
$[\sqrt{y}/x]_{y\geq 0}$
$↯_{c_0}$
$(y = 0)_{c_1} \wedge (y \geq 0)_{c_0}$
$y = 0$
$[-\infty/y]$
$[0/y]$
$↯_{c_0 \vee c_1}$
True
(B)
$(x^2 \leq y)_{c_0} \wedge (y = 0)_{c_1} \wedge (y = -1)_{c_2}$
$↯_{c_1 \wedge c_2}$
$(y = 0)_{c_1} \wedge (y \geq 0)_{c_0} \wedge (y = -1)_{c_2}$
$y = 0, y \geq 0, y = -1$
$y \geq 0, y = 0$
$y = -1$
$[-1/y]$
$↯_{c_2}$

Fig. 2. Solver state after (A) adding the constraints $c_0 : (x^2 \leq y)$ and $c_1 : (y = 0)$ followed by a consistency check, and (B) adding another constraint $c_2 : (y \leq -1)$ and performing another consistency check

node to such a constraint just results (according to the substitution rules) in cases containing variable-free inconsistent constraints.

The just described tree represents the data structure the theory solver maintains. As you can see in the next subsection, it provides all the information we need for an incremental implementation of the virtual substitution method and for the generation of infeasible subsets. Moreover, the tree structure permits an informed depth-first solution search instead of the breadth-first search given by the original virtual substitution.

3.2 Functionality of the theory solver

Using the above data structure, this subsection shows, by means of an example, how to initialize the solver, how to add constraints incrementally, how to perform a consistency check, and how to generate an infeasible subset of the added constraints, if the consistency check fails.

We initialize the solver's data structure with a single conjunction node containing the empty conjunction (representing *True*). First we add two constraints $c_0 : (x^2 \leq y)$ and $c_1 : (y = 0)$ to the solver, whose conjunction is inserted into the root.

Now we provoke the consistency check of $c_0 \wedge c_1$ (Figure 2 part (A) shows the data structure after the consistency check). The solver repeats choosing a node in the data structure and creating successors for it until it either finds an empty conjunction node or no more nodes, for which we can create successors, exist. The former implies consistency, the latter inconsistency. Currently, there is only one node to choose, namely the root. As it is a conjunction node, its successors are substitution nodes. Hence, we fix a variable to eliminate next and form the test candidates for this variable, which the constraints in the root provide. Instead of creating all successors by determining all test candidates the constraints provide, the data structure allows us to create only successors corresponding to the test candidates of a single constraint. We choose the variable x for elimination and the constraint $x^2 \leq y$ for test candidate generation, as it is the only constraint containing x. The created successors are formed by the substitution $[-\infty/x]$ without a side condition and the substitutions $[-\sqrt{y}/x]$ and $[\sqrt{y}/x]$ both with the side condition $y \geq 0$.

The data structure now contains four nodes. However, we can only create successors for the three recently created substitution nodes, since the only constraint in the root containing x has already been used to create successors. We choose the left-most substitution node. Its successors are created by applying $[-\infty/x]$ to the conjunction $x^2 \leq y \wedge y = 0$ of its antecessor. It results in a conflict node with index c_0 (statement (**) in Section 3.1), denoting that the test candidate $-\infty$ for x is not a solution for $x^2 \leq y$. We call c_0 the *conflict reason* for the already fixed assignments of variables to test candidates (here just x to $-\infty$).

The next node we choose is $[-\sqrt{y}/x]_{y \geq 0}$. Applying the substitution yields a single conjunction node as successor (statement (*) in Section 3.1). We continue with the newly created node, in which only y can be eliminated. We create successors for the test candidates provided by $y = 0$, resulting in the substitution nodes $[-\infty/y]$ and $[0/y]$. We choose the former one to continue and create its successor being a conflict node with the index $c_0 \vee c_1$. The test candidate $-\infty$ for y is neither a solution $y = 0$ nor for $y \geq 0$. Therefore, the conflict reason is either c_0 or c_1.

We continue with the substitution node $[0/y]$ and create its only successor being an empty conjunction node. Hence, the consistency check terminates and reports consistency of $c_0 \wedge c_1$.

Next we add the constraint $c_2 : (y = -1)$ belatedly to the solver in order to demonstrate incrementality. The solver uses the resulting data structure of the previous consistency check, adds $y = -1$ to its root, and marks all nodes not being the root or one of its direct successors in order to avoid choosing them during a subsequent consistency check.

The solver checks the consistency of $c_0 \wedge c_1 \wedge c_2$ as follows (Figure 2 part (B) shows the solvers data structure after the consistency check). The only unmarked nodes for which we can create new successors are the root, $[-\sqrt{y}/x]_{y \geq 0}$, and $[\sqrt{y}/x]_{y \geq 0}$. We choose $[-\sqrt{y}/x]_{y \geq 0}$ and apply virtual substitution to the belatedly added constraint $y = -1$. The virtual substitution gives a real algebraic formula $C_1 \vee \ldots \vee C_n$ that is equivalent to the result of the standard substitution but does not contain square roots or fractions. To handle the case splitting, we copy each successor of the substitution node (including its subtree) n times and for each i, $1 \leq i \leq n$, we add the conjunction C_i, and to the ith copy. All nodes in these copied subtrees except the roots and their direct successors get marked in order to avoid choosing them in the next step. In the example there is a single successor and the resulting formula consists of one constraint, hence we just add this constraint to the successor and unmark it and its direct successors.

Now, we can choose each node in the data structure, except the empty conjunction node. We choose $[0/y]$ and create its successor, which is a conflict node with conflict reason c_2.

The next node we choose is the conjunction node $y = 0 \wedge y \geq 0 \wedge y = -1$. Until now we created only the successors considering the test candidates provided by $y = 0$. The constraint $y \geq 0$ provides the same test candidates. The constraint $y = -1$ provides another test candidate -1 for y. We create the according successor and choose it to continue the consistency check. Its resulting successor is a conflict node with conflict reason $c_0 \vee c_1$.

Now, we have created all possible successors of the considered conjunction node $y = 0 \wedge y \geq 0 \wedge y = -1$ and all lead to conflicts. It implies that its conjunction is not consistent. A conflict reason of this node must imply the conflict reason of each of its successors. We can choose for example the conflict reason $c_1 \wedge c_2$ and, indeed, $y = 0 \wedge y = -1$, which is the a result of applying $(c_1 \wedge c_2)[-\sqrt{y}/x]$, is inconsistent. Moreover, $y = 0 \wedge y = -1$ also appears in the preceding conjunction node, the root. Therefore it is also a conflict reason of the root and an infeasible subset of the constraints added to the solver.

We can use this information to jump back to the last node that implies the generated conflict reason. The constraints of that node are already conflicting, i.e., we do not need to continue the search in that subtree. We call this approach *backjumping*. In our example, we jump back to the root, which thus contains a conflict, and we report inconsistency.

3.3 Experimental results

We are currently building a prototype implementation of the introduced procedures. We have already completed an *incremental* theory solver and a *non-incremental* one, both supporting the computation of *infeasible subsets*.

To get first results for the performance of an *SMT-solver* involving our theory solver, we embedded our implementation into an existing SMT-solver. However, it does *not* yet make use of the computed infeasible subsets. The SMT-embedding allows *lazy* or *less lazy* theory solver invocation. A lazy invocation calls the theory solver only when a complete solutions for the Boolean skeleton is found. The less lazy variant invokes the theory solver for consistency check after the completion of each decision level of the SAT-solver.

Since we did not yet embed an underlying complete theory solving procedure, to test our approach we need examples that during solving do not lead to polynomials of degree higher than two. Only after embedding, e.g., the CAD method, we will be able to handle more relevant case studies.

Table 1. Running times (in seconds) for 4 benchmarks

Solver	Example 1	Example 2	Example 3	Example 4
Less lazy, non-incremental, with back-jumping	18.879	15.805	22.769	13.981
Less lazy, incremental	23.879	43.200	20.103	47.124
Less lazy, non-incremental	49.217	99.714	109.582	68.955
Full lazy, non-incremental	550.988	144.826	162.805	102.458
`REDLOG`	1147.889	2104.883	1850.559	251.474

As benchmarks we have created a random set of test formulas of the form

$$(\ x_0 x_1 = c \ ) \wedge \bigwedge_{i=1}^{10} \bigvee_{j=1}^{5} (\ x_{i,j,1} + x_{i,j,2} + x_{i,j,3} + x_{i,j,4} + x_{i,j,5} = c_{i,j} \ )$$

where x_0, x_1, and all $x_{i,j,k}$ are real-valued variables from a set Var with $|\mathit{Var}| = 20$ such that x_0 and x_1 are syntactically different, and c and all $c_{i,j}$ are constants from the set $\{1, \ldots, 50\}$. The position of the single-literal clause, which is listed first above, is determined randomly, i.e., it is not always the first clause. Table 1 shows results, which are characteristic for this kind of input formula. All listed example formulas are satisfiable.

The running times show the expected result. The benefit of involving an SMT-solver can be observed by comparing the third and fourth row with the last one. The second row shows the running times of the SMT-solver using an incremental theory solver, whereas the Solver in the first row makes use of backjumping but without incrementality. Combining backjumping with incrementality has not yet been implemented but we expect it to lead to even shorter running times. The most promising improvement will be the inclusion of the infeasible subsets in the SMT environment.

4 Conclusion

In this paper we proposed an incremental adaptation of the virtual substitution method, and introduced methods for backjumping and for the generation of infeasible subsets. Our next step will be to make use of the infeasible subsets in our SMT-solver.

Our data model provides possibilities for different decision heuristics, which we also want to explore. Additionally, we will extend the degree of the constraints we can handle to its theoretical maximum of 4, and invoke a complete fall-back procedure.

References

[Bjo99] N. S. Bjorner. *Integrating Decision Procedures for Temporal Verification.* PhD thesis, Stanford University, 1999.

[Bjo10] N. S. Bjorner. Linear quantifier elimination as an abstract decision procedure. In *Proc. of Automated Reasoning*, volume 6173 of *LNCS*, pages 316–330. Springer-Verlag, 2010.

[BPT07] A. Bauer, M. Pister, and M. Tautschnig. Tool-support for the analysis of hybrid systems and models. In *DATE '07*, pages 924–929. European Design and Automation Association, 2007.

[Bro03] C. W. Brown. QEPCAD B: A program for computing with semi-algebraic sets using CADs. *SIGSAM Bull.*, 37:97–108, 2003.

[CJ98] B. F. Caviness and J. R. Johnson, editors. *Quantifier Elimination and Cylindrical Algebraic Decomposition.* Texts and Monographs in Symbolic Computation. Springer-Verlag, 1998.

[Cor11] Florian Corzilius. Virtual substitution in SMT solving. Master's thesis, RWTH Aachen University, 2011. http://www-i2.informatik.rwth-aachen.de/i2/publications/.

[DH88] J. H. Davenport and J. Heinz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5:29–35, 1988.

[dMB08] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.

[DS97] A. Dolzmann and T. Sturm. REDLOG: Computer algebra meets computer logic. *SIGSAM'97*, 31:2–9, 1997.

[DSW97] A. Dolzmann, T. Sturm, and V. Weispfenning. Real quantifier elimination in practice, 1997.

[FHT$^+$07] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling, and Computation*, 1:209–236, 2007.

[KS08] D. Kroening and O. Strichman. *Decision Procedures - An Algorithmic Point of View.* Springer-Verlag, 2008.

[PJ09] G. O. Passmore and P. B. Jackson. Combined decision techniques for the existential theory of the reals. In *Proc. of Calculemus'09/MKM'09*, volume 5625 of *LNCS*, pages 122–137. Springer-Verlag, 2009.

[Tar48] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 1948.

[Wei88] V. Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5:3–27, 1988.

[Wei98] V. Weispfenning. A new approach to quantifier elimination for real algebra. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts and Monographs in Symbolic Computation, pages 376–392. Springer-Verlag, 1998.