A Lazy SMT-Solver for a Non-Linear Subset of Real Algebra

Erika Ábrahám[1] and Ulrich Loup[1] and Florian Corzilius[1] and Thomas Sturm[2]

[1] RWTH Aachen University, Germany
[2] Universidad de Cantabria, Spain

Abstract. SAT-solving is a highly relevant research area with plenty of industrial applications. SMT-solving over the reals, extending SAT with theories, has its main focus on linear arithmetic. However, there are only few solvers being capable of more expressive but still decidable logics like the first-order theory of the reals with addition and multiplication – real algebra.
One of the main requests on theory solvers that must be fulfilled for their efficient embedding into a lazy DPLL-based SMT-solver is *incrementality*, which is not supported by currently available theory solvers for real algebra. In this paper we address the extension of an existing theory-solving algorithm, the virtual substitution method, to support incrementality.

1 Introduction

The *satisfiability problem* poses the question 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 is 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], and this for even linear input formulas.

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 does not make unnecessary work and 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 minimal in the sense that if we remove a constraint the remaining ones become satisfiable.
- *Backtracking:* If a conflict occurs, either in the Boolean or in the theory domain, the solver should be able to remove some constraints and continue the check at an earlier state.

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 and backtracking, and embed it into an SMT-solver. The generation of minimal infeasible subsets is future work and should further optimize our tool.

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.

Related work. We are only aware of the SMT-solvers Z3 [dMB08], HySAT [FHT+07] and ABsolver [BPT07] which are able to handle nonlinear real arithmetic constraints. The algorithm implemented in HySAT and currently in its successor tool iSAT uses interval constraint propagation to check real constraints. This technique is only pseudo-complete, i.e. it sometimes cannot solve the problem with a clear satisfiability result. Nevertheless it is in practice more efficient compared to solvers based upon exact methods [FHT+07]. The structures of ABsolver and Z3 are more similar to our SMT-solver planned. However to our knowledge, Z3 does not support full real-algebraic constraints. The authors of ABsolver do not address the issues of incrementality and backtracking. Though ABsolver computes minimal infeasible subsets, we did not find any information how they are generated.

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 in Section 3 and give some first experimental results in Section 4. We conclude the paper in Section 5.

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 variables x according to the following abstract grammar:

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

Syntactic sugar like $\mathrm{True}, \mathrm{False}, /, -, \wedge, \rightarrow, \forall, \ldots$ 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 skip sometimes 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 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 2.2, as an algorithm for the theory solver, which is very efficient but restricted in the degree of polynomials that can be handled.

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 the last years. One of the main achievements in the field of SAT-solving is the DPLL-algorithm, which is capable of solving existential Boolean formulas and furthermore, capable of performing consistency checks with other logical theories. Thus, DPLL-based decision procedures can be applied to logics richer than propositional logic, by abstracting all non-propositional atomic formulas by propositional variables. This approach is called lazy *SAT-modulo-theories* (*SMT*) solving. More comprehensive information on this topic can be found in [KS08].

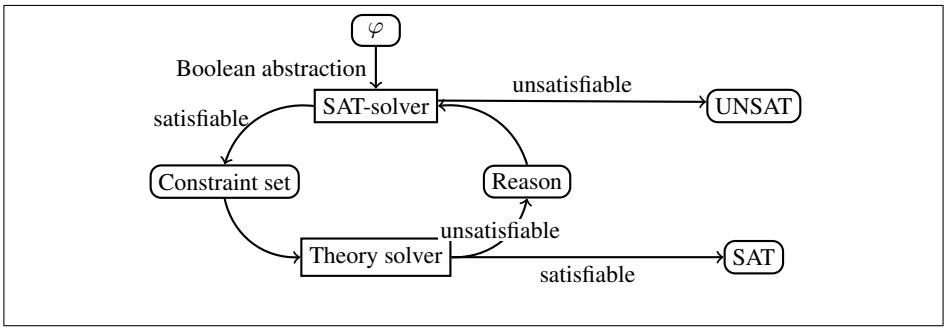


Fig. 1: The basic scheme of DPLL-based full lazy SMT-solving

The basic scheme of lazy DPLL-based SMT-solving is roughly as follows (cf. 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, a *minimal 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 again an assignment for the refined Boolean problem, which in turn has to be verified by the theory solver. Continuing this iteration in the end decides the satisfiability of the input formula.

Above we described a *full lazy* procedure (also referred to as offline integration schema), where the theory solver checks constraints corresponding to a complete assignments only. In practice this is often disadvantageous, since 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 (also referred to as online integration schema) 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. Furthermore, in case of a conflict the theory solver should also be able to *backtrack* to previous computation states.

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 *backtracking* (cf. Section 2.1).

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. We are interested in checking satisfiability of a pure-existentially quantified formula (cf. "unquantified" in terms of SMT-LIB). Below we explain how the innermost existentially bound variable is eliminated by 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 of degree at most two. Thus we assume that all constraints in φ are of the form $f \sim 0$, $\sim \in \{=, <, >, \leq, \geq, \neq\}$, where f is a polynomial that is at most quadratic in x with polynomial coefficients.

Considering the real domain of a variable x, 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 f in x. Given a constraint $f = ax^2 + bx + c \sim 0$, $\sim \in \{=, <, >, \leq, \geq, \neq\}$, the finite endpoints of its satisfying intervals are the zeros of $f = ax^2 + bx + c$:

$$\begin{array}{ll} x_0 = -\frac{c}{b} & \textit{if } a = 0 \ \wedge \ b \neq 0 \\ x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} & \textit{if } a \neq 0 \ \wedge \ b^2 - 4ac \geq 0 \\ x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a} & \textit{if } a \neq 0 \ \wedge \ b^2 - 4ac \geq 0 \end{array}$$

A set of constraints has a common solution iff the intersection of their solution intervals is not empty. If so, this intersection contains at least one left/right endpoint of a left-/right-closed solution interval of a constraint, or a point infinitesimal greater/smaller than the left/right endpoint of a left-/right-open solution interval of a constraint. We call such points *test candidates*. Basically, the virtual substitution recursively eliminates x in φ by (i) determining all test candidates for x in all constraints in φ that contains x, and (ii) checking if one of these test candidates satisfies φ.

To check whether a test candidate e for x satisfies another constraint $g \sim 0$ in φ, we substitute all occurrences of x by e in g, yielding $g[e/x] \sim 0$, and check the resulting constraint under the solution's side conditions for consistency. Note that neither $g[e/x] \sim 0$ 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, namely ∞ or square roots. Furthermore, it would introduce new variables for infinitesimals. 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.

The virtual substitution method defines 30 substitution rules: There are six relation symbols and five possible types of test candidates corresponding to (1) the left or right endpoint of a left- or right-closed interval, (2) the right endpoint of a right-open interval, (3) the left endpoint of a left-open interval, (4) ∞, or (5) $-\infty$. To describe all substitution rules would go beyond the scope of this paper. We refer to [LW93,Wei97] for a complete description and give here two examples to demonstrate the idea:

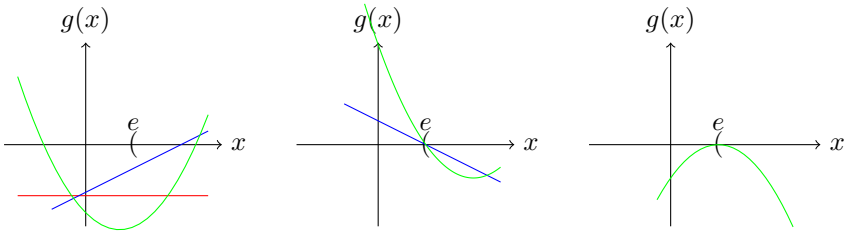
1. We first show the case for a test candidate being an left- respectively right-endpoint of a left- respectively right-closed interval used for substitution in an equation. So let e be a test candidate for x of type (1) and assume the constraint $g = 0$ occurring in φ. If we use standard substitution to replace x by e in $g = 0$, the result can be transformed to the general form $\frac{r+s*\sqrt{t}}{q} = 0$, where q, r, s and t are polynomial terms of the real algebra.
   We distinguish between the cases of s being 0 or not, i.e., if there is a square root in the term after substitution or not. In case $s = 0$ the equation $\frac{r+s*\sqrt{t}}{q} = 0$ simplifies to $\frac{r}{q} = 0$ and further to $r = 0$. In case $s \neq 0$, the constraint $\frac{r+s*\sqrt{t}}{q} = 0$ is satisfied iff $r + s * \sqrt{t} = 0$, or equivalently, iff either both r and s equal 0, or they have different signs and $|r| = |s\sqrt{t}|$. Therefore the virtual substitution replaces the constraint $g = 0$ by

$$(s = 0 \wedge r = 0) \vee (s \neq 0 \wedge r * s \leq 0 \ \wedge \ r^2 - s^2 t = 0).$$

2. The second case we describe is the substitution for a test candidate of type (3) in an inequation $g < 0$. The test candidate represents in this case the left end point e of a left open interval plus an infinitesimal value. The substitution of the test candidate for x in $g < 0$ is equivalent to the following:

$$\underbrace{g[e/x] < 0}_{\text{Case 1}} \vee \underbrace{g[e/x] = 0 \wedge g'[e/x] < 0}_{\text{Case 2}} \vee \underbrace{g[e/x] = 0 \wedge g'[e/x] = 0 \wedge g''[e/x] < 0}_{\text{Case 3}}$$

   Either g maps the endpoint to a negative value satisfying Case 1, where it is surrounded by negative values due to the density of $\mathbb{R}$ or it is zero and one of its derivatives in x is negative satisfying Case 2 or 3, which implies that values in the right neighborhood of e are mapped to negative values.
   In the above formula, the substitutions $g[e/x] = 0$ and $g'[e/x] = 0$ are computed according to the first case above. The other substitutions $g[e/x] < 0$, $g'[e/x] < 0$, and $g''[e/x] < 0$ are computed using the substitution rule for test candidates of type (1) and a strict inequation, not listed here.



Case 1 Case 2 Case 3

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

possible test candidates results in the formula

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

Note that test candidates of types (4) and (5) do not have side conditions. The virtual substitution method continues with the elimination of the next variable y_n.

3 Incremental Virtual Substitution

As discussed in Section 2.1, a theory solver should support incrementality in order to be suited for an efficient embedding into a less lazy SMT-solver. Note that theory solvers in the SMT-context have as input conjunctions of theory constraints only, instead of arbitrary combinations. If the theory solver already checked a set of theory constraints for consistency, it should be able to accept one more constraint, and to check whether the conjunction of the already added constraints and the new constraint is still consistent. Furthermore, it should support backtracking to a previous solver state, thereby undoing the adding of some 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 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 main section we propose an incremental version of the virtual substitution method.

Assume that the original virtual substitution method checks the satisfiability of a formula and eliminates a variable x. The elimination yields a list of test candidates with corresponding side conditions. After the substitution step the result is a new formula being the disjunction of the substitution results for each test candidate of each constraint containing x. Note that this new formula, which does not contain the variable x any more, gets exponentially large.

If we want to support the belated addition of further constraints, possibly containing x, we must be able to belatedly substitute x in the new constraint using the previous test candidates. Furthermore, we have to find the test candidates of the new constraint for x and belatedly consider them for substitution. For this purpose we must firstly store all the received constraints and secondly the list of all determined test candidates with their corresponding side conditions.

A naive approach would be to mimic the original virtual substitution method: we could store all the abovementioned information, apply all relevant previous substitutions to new constraints, and extend the formula with new disjunctive components using test candidates from the new constraint. However, this approach would lead to very large formulas.

To reduce the data to be stored and to support incrementality, we follow an informed search instead of a breadth-first search. To understand how this can be achieved, we first describe the data model underlying our search.

Algorithm 1 The incremental virtual substitution algorithm (1)

```
bool add_new_constraint(constraint c)
begin
    add_new_constraint(c,root);                                   (1)
    return is_consistent(root);                                   (2)
end

void add_new_constraint(constraint c, element (C,S)_t)
begin
    if t = x then                                                 (1)
        C := C ∪ {(c : f)};                                       (2)
    else if t =⊥ then                                             (3)
        C := C ∪ {c};                                             (4)
    end if                                                        (5)
    for all children (C',S')_t' of (C,S)_t do                     (6)
        add_new_constraint(c,(C',S')_t');                         (7)
    end for                                                       (8)
end

bool is_consistent(element (C,S)_t)
begin
    if (C,S)_t is a leaf then                                     (1)
        return is_consistent_leaf((C,S)_t);                       (2)
    else                                                          (3)
        return is_consistent_innernode((C,S)_t);                  (4)
    end if                                                        (5)
end
```

Remember that the virtual substitution starts with a formula and applies variable elimination and substitutions to it. Both of these operations lead to branching on possible solutions: the variable elimination branches on possible test candidates yielding pairs of substitutions and corresponding substitution conditions, and the substitution itself branches also on possible substitution cases. As we want to be able to belatedly apply those operations to later arrived constraints, we must remember not only the current result but also the history of operations executed. Therefore the current solver state is stored in a tree. Each leaf corresponds to a possible solution of the constraints handed over to the theory solver. Inner nodes represent an earlier term to that either variable elimination or a substitution was applied, yielding the disjunction of the terms represented by its children.

The nodes are indexed tuples $(C,S)_t$ with C a set of polynomial constraints, S a set of substitutions, and $t \in \{\bot\ ,*\} \cup \mathit{Var}$ where *Var* is the set of real-valued variables appearing in constraints. The substitution set S contains substitutions that were or still should be applied to the constraints handed over to the solver in the branch leading to the node. The constraints C result from the original constraints by applying substitutions from S to them. The index t denotes the next operation applied to C which results in a branching on the cases represented by the children of the node. An element $(C,S)_\bot$ is

always a leaf, as the index $\perp$ denotes that no operation was applied to the constraints in C since the node was added. A node $(C,S)_x$ has children representing the cases for the different test candidates for the elimination of the variable x; the substitution sets of the children extend S with the corresponding substitution and the constraint sets of the children extend C with the corresponding side conditions. An element $(C,S)_*$ is a node in which a substitution was applied to a constraint in C; the result of the substitution was stored in a number of generated children, which represent the different substitution cases.

The search tree initially consists of a single node $(\emptyset,\emptyset)_\perp$, storing the information that the theory solver did not get any constraints yet, no substitution was yet applied, and no next operation on the constraint set was determined yet.

When the solver gets a new constraint it does a sequence of operations to check if the previously received constraints together with the new constraint are still satisfiable. Mimicking the original virtual substitution method would require that we belatedly apply all substitutions to the new constraint on all paths to leaf nodes and add all branching results as children to the leaf nodes. Furthermore, inner nodes that branch on the elimination of variables occurring in the new constraint would get new children due to new possible test candidates for the elimination, which should be further processed. When completed the search, the tree would have a leaf with an empty constraint set (or containing only tautologies) for each satisfying branch.

We do not follow this breadth-first approach, but use a more depth-first search oriented algorithm, in that we continue the search only until we get a leaf with an empty constraint set, assuring that there is at least one solution.

When the theory solver gets a new constraint, the new constraint gets added to each constraint set C of each node $(C,S)_t$ in the three with $t \neq *$. Why we do not need to add new constraints to $*$-indexed nodes will become clear later (though it would not be critical to add them, it is not necessary). Then we heuristically choose a leaf and apply substitutions and variable eliminations to the constraint set until we either get a satisfying leaf, or all children turn out to correspond to unsatisfiable branches. In the first case we are ready, whereas in the second case we delete the chosen node and continue in other branches.

The incremental virtual substitution algorithm can be described by the pseudo-code Algorithms 1 and 2. We explain the functioning of the algorithm on a small example.

Initially the search tree consists of a single node $(\emptyset, \emptyset)_\perp$. We call *add_new_constraint* with the constraint $c_1 : x^2 - y \geq 0$ as parameter to hand over the first constraint to the theory solver. The method *add_new_constraint* adds the constraint c_1 to the constraint sets of all nodes that are not indexed by $*$. There is just one such node $(\emptyset, \emptyset)_\perp$, which gets extended to $(\{c_1\}, \emptyset)_\perp$.

After the addition of the constraint a consistency check is performed by the method *is_consistent*. Figure 2 shows the resulting tree after the check. At the beginning the tree consists of the root node $(\{c_1\}, \emptyset)_\perp$ being leaf marked by $\perp$, thus the method *is_consistent_leaf* gets invoked. As there are no substitutions to consider yet, the root is evaluated according to the second case in the method *is_consistent_leaf*. It markes the root by the variable x, which gets eliminated based on c_1 producing the test candidates:

1. $-\sqrt{y}$ with side conditions $1 \neq 0 \wedge 4y \geq 0$,

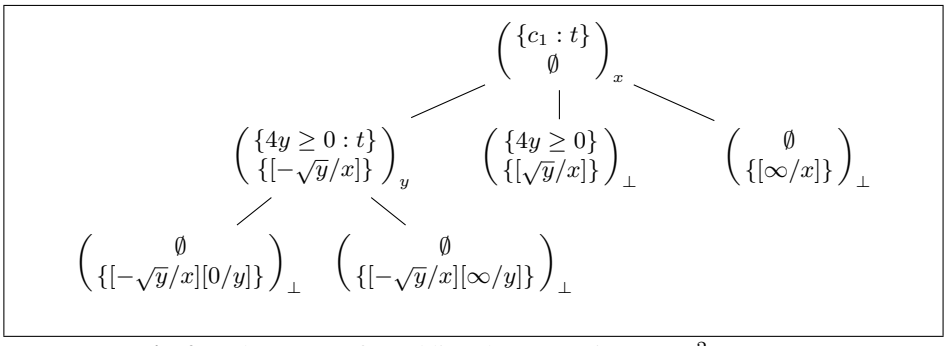


Fig. 2: Solver state after adding the constraint $c_1 : \ x^2 - y \geq 0$.

2. $\sqrt{y}$ with side conditions $1 \neq 0 \wedge 4y \geq 0$,
3. ∞ with no side conditions.

Note, that ∞ is the test candidate representing the right endpoint of the right-unbounded solution interval. The constraint c_1 in the conditions of the root gets the marked by t, which says that it was already involved to create test candidates for the elimination of x. A leaf is created for each of the generated test candidates. In the side conditions we skip tautologies. Note that if there were further constraints in the processed node they were handed over to the children.

In the next step we choose one of the just created new leaves, e.g., the left one. It still has a non-empty constraint set referring to the variable y that gets eliminated. The node gets marked by the eliminated variable y and the constraint used to generate the test candidates gets labeled by t. This step generates a leaf with an empty condition set, thus the constraint is satisfiable and we can stop the search.

Figure 3 depicts the result of adding a further constraint $c_2 : \ x^2 - 1 = 0$. Again it is appended to the constraint sets of all nodes in the search tree. Note, that the new constraint is labeled in elimination nodes by f, denoting that is was not yet used to generate test candidates. Next we select a leaf, which again is the left-most one. It has a single constraint c_2 in which we substitute $-\sqrt{y}$ for x, leading to a single new child. We apply the second substitution for $[0/y]$ to the child's constraint which results in a contradiction. All three nodes up to the y-indexed node get deleted. We decide to take its child corresponding to the test candidate ∞ for y. Complete evaluation leads again to inconsistency, and also this path gets deleted up to the y-indexed node. This node now is a leaf and we create new test candidates for y using the f-labeled constraint c_2, which now gets the label t. There is just one new test candidate for y, namely 1. Its substitution leads to a satisfying node, and the method terminates.

When the current set of constraints turns out to be unsatisfiable, the theory solver should be able a backtrack. We have the choice between two possibilities: (1) We can store all received constraints in a list. In case of backtracking we delete the current tree and construct a new one for the constraints that were not undone. (2) We can store before each decision of the SAT-solver the current tree in a stack in order to restore an earlier solver state. Currently we do (2). We will analyze in the future if it pays off or if recomputing would be more efficient.

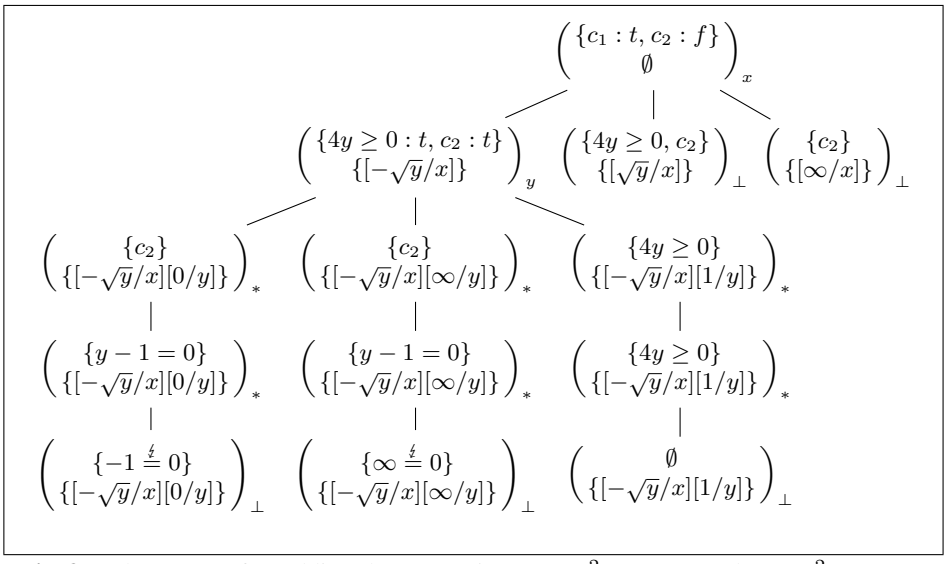


Fig. 3: Solver state after adding the constraints $c_1 :\ x^2 - y \geq 0$ and $c_2 :\ x^2 - 1 = 0$.

4 Experimental results

We are currently building a prototype implementation of the proposed algorithm and its embedding into an SMT-solver. First results show that the running times for conjunctions of constraints are comparable for the non-incremental and the incremental version of the virtual substitution algorithm. This implies that the costs of additional book-keeping remains relatively small, and that the depth-first setting has comparable running times as well as the breadth-first setting. This gives us optimal conditions for the embedding of our theory solver into an SMT-solver for quadratic case (and beyond) of the real algebra.

We created a random set of test formulas of the form

$$(x_a x_b = d) \wedge \bigwedge_{(j_0,j_1,j_2,j_3,j_4,c)\in M} \bigvee_{i=0}^{4} (x_{j_0} + x_{j_1} + x_{j_2} + x_{j_3} + x_{j_4} = c)$$

where $V = \{0, \ldots, 19\}$, $V_c = \{1, \ldots, 50\}$, $M \subseteq V^5 \times V_c$ with $|M| = 10$, the x_l are variables for all $l \in V$, and a,b,$d \in V$ with $a \neq b$. The position of the clause $(x_a x_b = d)$ in this CNF-formula is determined randomly, i.e., it is not always the first clause. Table 1 shows results characteristic for this kind of input formula. All listed example formulas are satisfiable. (We did not make any intensive benchmark testing yet, but we will give in an extended version of this paper further case studies for other unsatisfiable and satisfiable benchmarks.)

The running times show that in most of the cases less lazy solving is superior to the full lazy approach. Note that the example formulas are relatively small, thus less lazy solving has its main advantage in the facts that (1) in case a partial assignment already leads to a conflict, the theory solver needs to check smaller sets of constraints only, and (2) less lazy setting yields smaller conflicts. (As we do not yet support minimal infeasible

subset generation, we take the disjunction of the negated unsatisfiable constraints as conflicting clause.)

Since we invoke the theory solver after each decision level handing over new constraints, the incremental version has to do less work than the non-incremental one. The running times show that the book-keeping effort pays off.

Table 1 First experimental results (times depicted in seconds) of the prototype implementation of the less lazy SMT-solver using an incremental virtual substitution solver.

Less lazy with incremental theory solver:	16.589	11.8851	28.4418	0.5557	9.1395
Less lazy with non-incremental theory solver:	59.2009	73.7511	97.8212	62.4652	83.9639
Full lazy with non-incremental theory solver:	287.51	105.233	86.7371	71.8332	> 150.0
`Redlog`:	> 300.0	> 300.0	> 150.0	> 150.0	> 150.0

5 Conclusion

In this paper we proposed an incremental adaptation of the virtual substitution method. First results of our prototype implementation show that the approach is efficient.

There is a lot of future work to do. Firstly, we have to finish the implementation of the incremental theory solver. We are already working on its efficient embedding into an SMT solver. The next step will be the development of an incremental adaptation of the CAD method. This allows us to combine those decision procedures in a style as done in `Redlog`, to be able to handle full real algebra. The generation of minimal infeasible subsets is another important feature we are working on.

The incremental theory solver has to handle case splitting in the theory domain due to the depth-first search in the tree of solution sets. It would be also interesting to consider to shift those decisions into the SAT domain in order to speed up the search by conflict learning.

References

BPT07. Andreas Bauer, Markus Pister, and Michael Tautschnig. Tool-support for the analysis of hybrid systems and models. In *Proceedings of the 2007 Conference on Design, Automation and Test in Europe (DATE), Nice, France*, pages 924–929. European Design and Automation Association, 2007.

Bro03. Christopher W. Brown. Qepcad b: a program for computing with semi-algebraic sets using CADs. *SIGSAM Bull.*, 37(4):97–108, 2003.

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

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

dMB08. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, 2008. Springer-Verlag.

DS97. Andreas Dolzmann and Thomas Sturm. REDLOG: Computer algebra meets computer logic. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 31(2):2–9, June 1997.

DSW97. Andreas Dolzmann, Thomas Sturm, and Volker Weispfenning. Real quantifier elimination in practice, July 20 1997.

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. *JSAT*, 1(3-4):209–236, 2007.

KS08. Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Springer-Verlag Berlin Heidelberg, 2008.

LW93. Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5):450–462, 1993.

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

Wei88. Volker Weispfenning. The complexity of linear problems in fields. *J. Symbolic Comput.*, 5(1-2):3–27, 1988. MathSciNet review: 89g:11123.

Wei97. Volker Weispfenning. Quantifier elimination for real algebra - the quadratic case andbeyond. *Appl. Algebra Eng. Commun. Comput*, 8(2):85–101, 1997.

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, Wien-New York, 1998. Springer-Verlag.

Algorithm 2 The incremental virtual substitution algorithm (2)

bool is_consistent_leaf$((C,S)_t)$
begin
  if $t = \bot$ and exists substitution $s \in S$ applicable to a $c \in C$ **then** (1)
    $t := *$; (2)
    substitute s in c yielding $\bigvee_{i=1,\ldots,n} \bigwedge_{j=1,\ldots,k_i} c_{i,j}$; (3)
    for all $i = 1,\ldots,n$ **do** (4)
      add child $(C\backslash\{c\} \cup \{c_{i,j} | j = 1,\ldots,k_i\},S)_\bot$ to $(C,S)_*$ (5)
    end for (6)
  else if ($t = \bot$ and exists $c \in C$ containing a variable x) or (7)
    ($t = x$ and exists $(c : f) \in C$ containing x) **then** (7)
    if $t = \bot$ **then** (8)
      $t := x$; (9)
      $C := \{(c' : f) | c' \in C, c' \neq c\} \cup \{(c : t)\}$; (10)
    else (11)
      $C := C\backslash\{(c : f)\} \cup \{(c : t)\}$; (12)
    end if (13)
    solve c for x yielding test candidates e_i, $i = 1,\ldots,n$, with (14)
      side conditions $\bigwedge_{j=1,\ldots,k_i} c_{i,j}$ for each i; (15)
    for all $i = 1,\ldots,n$ **do** (16)
      $C' := \{c' | (c' : t) \in C, c' \neq c\} \cup \{c_{i,j} | j = 1,\ldots,k_i\}$; (17)
      $S' := S \cup \{[e_i/x]\}$; (18)
      add child $(C',S')_\bot$ to $(C,S)_x$ (19)
    end for (20)
  end if (21)
  if $(C,S)_t$ is still a leaf **then** (22)
    return (C is empty); (23)
  else (24)
    return is_consistent($(C,S)_t$); (25)
  end if (26)
end

bool is_consistent_innernode$((C,S)_t)$
begin
  for all children $(C',S')_{t'}$ of $(C,S)_t$ **do** (1)
    if is_consistent($(C',S')_{t'}$) **then** (2)
      return true; (3)
    else (4)
      remove child $(C',S')_{t'}$; (5)
    end if (6)
  end for (7)
  return is_consistent($(C,S)_t$); (8)
end