SMT-Solving in the Analysis and Synthesis of Hybrid Systems

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. There are several methods for the synthesis and analysis of hybrid systems that require efficient algorithms and tool for satisfiability checking. In this paper we give examples of such synthesis and analysis methods, and discuss decision procedures that can be used for this purpose.

1 Introduction

There are several methods for the synthesis and analysis of hybrid systems that require efficient algorithms and tools for satisfiability checking. For analysis, e.g., bounded model checking describes counterexamples of a fixed length by logical formulas, whose satisfiability corresponds to the existence of such a counterexample. As an example for parameter synthesis, we can state the correctness of a parameterized system by a logical formula; the solution set of the formula gives us possible safe instances of the parameters.

For discrete systems, which can be described by propositional logic formulas, SAT-solvers can be used for the satisfiability checks. For hybrid systems, having mixed discrete-continuous behavior, SMT-solvers are needed. SMT-solving extends SAT with theories, and has its main focus on linear arithmetic, which is sufficient to handle, e.g., linear hybrid systems. However, there are only few solvers for more expressive but still decidable logics like the first-order theory of the reals with addition and multiplication – real algebra. Since the synthesis and analysis of non-linear hybrid systems requires such a powerful logic, we need efficient SMT-solvers for real algebra. Our goal is to develop such an SMT-solver for the real algebra, which is both complete and efficient.

The SMT-solvers `Z3` [dMB08], `HySAT` [FHT+07], and `ABsolver` [BPT07] are able to handle nonlinear real arithmetic constraints. The algorithm implemented in `HySAT` and in its successor tool `iSAT` uses interval constraint propagation to check real constraints. This technique is very efficient but incomplete, i.e., sometimes it returns unknown as result. The structures of `ABsolver` and `Z3` are more similar to our approach but they do not support full real-algebra.

SMT-solvers combine a SAT-solver with a theory solver. Their input is a formula being a Boolean combination of propositions and theory constraints (usually in conjunctive normal form). The SAT-solver seaches for a set of theory constraints such that when they are consistent, then the formula is satisfied. The theory solver is invoked to check the theory constraint set for consistency.

In practice, this *full lazy* approach is not very efficient. *Less lazy* variants invoke the theory solver already for partial assignments with incomplete theory constraint sets. If the constraint set is consistent, it gets extended by further constraints, and the extended set is checked again for consistency. Therefore, one of the main requests on theory solvers that must be fulfilled for their efficient embedding into a less lazy SMT-solver is *incrementality*. Incrementality means, that the theory solver can check a set of theory constraints for consistency, and can re-use the result for the check of an extended theory constraint set. Incrementality is not supported by the 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.

In the following we give an introduction to the virtual substitution method in Section 2 and introduce our incremental virtual substitution algorithm in Section 3. We conclude the paper in Section 4.

2 The Virtual Substitution Method

With the real numbers $\mathbb{R}$ as domain and with addition and multiplication as operators, 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.

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 worst-case time complexity is doubly exponential in the number of variables [DH88,Wei88]. Today, several methods satisfying these complexity bounds are available, e.g., the cylindrical algebraic decomposition (CAD) [CJ98] , the Gröbner basis, and the virtual substitution method [Wei98]. Theory solvers based on these methods are, e.g., the stand-alone application `QEPCAD` [Bro03] or the `Redlog` package [DS97] of the computer algebra system `Reduce`. Though these theory solvers are efficient for conjunctions, they are not suited to solve large formulas containing arbitrary combinations of real constraints (which they usually handle by syntactic case splitting).

The *virtual substitution method* 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 handle polynomials of degree 2.

The decision procedure based on virtual substitution produces a quantifier-free equivalent of a given existentially quantified input formula, by successively eliminating all bound variables starting with the innermost one.

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 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. 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 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 such 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) $-\infty$. We refer to [LW93,Wei97] for a complete description and give here one example to demonstrate the idea: We 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

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

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

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

The virtual substitution method continues with the elimination of the next variable y_n.

3 Incremental Virtual Substitution

In this 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.

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

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
```

branching on the cases represented by the children of the node. An element $(C,S)_\perp$ 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 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 1 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 $1 \neq 0 \wedge 4y \geq 0$,
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 2 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.

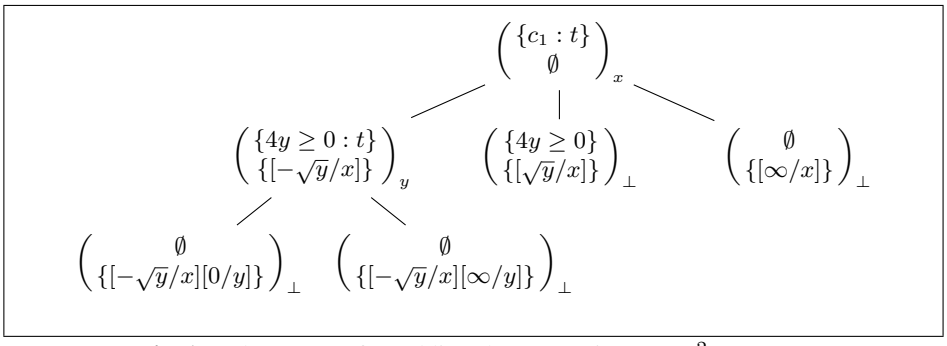

Fig. 1: Solver state after adding the constraint $c_1 : \ x^2 - y \geq 0$.

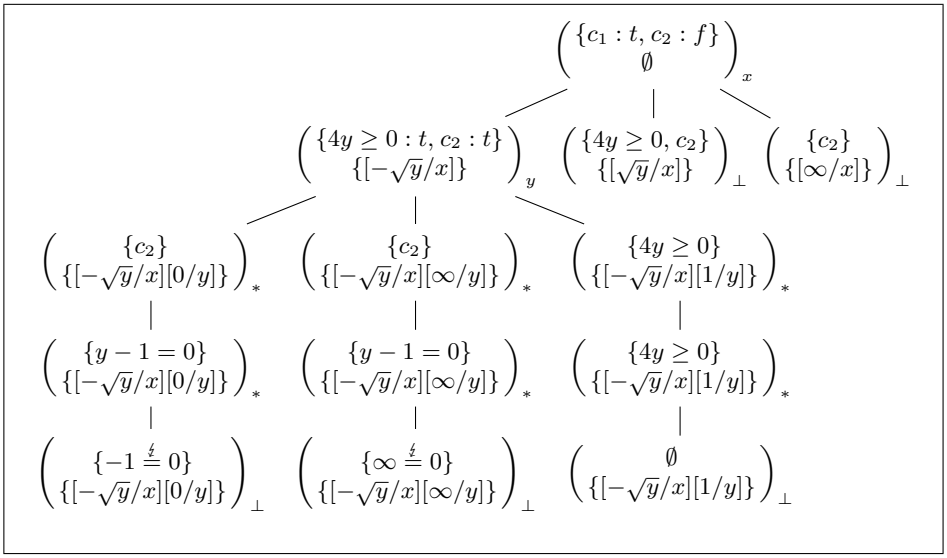

Fig. 2: Solver state after adding the constraints $c_1 : \ x^2 - y \geq 0$ and $c_2 : \ x^2 - 1 = 0$.

4 Conclusion

In this paper we proposed an incremental adaptation of the virtual substitution method. As to future work, we are already working on the efficient embedding of the incremental theory solver 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.

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.

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.

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 $t = \perp$ 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 $(C\backslash\{c\} \cup \{c_{i,j} | j = 1,\ldots,k_i\},S)_\perp$ to $(C,S)_*$ (5)
    end for (6)
  else if ($t = \perp$ and exists $c \in C$ containing a variable x) or (7)
      ($t = x$ and exists $(c : f) \in C$ containing x) **then** (7)
    if $t = \perp$ **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 $(C',S')_\perp$ 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