Symbolic Computation Techniques in Satisfiability Checking

Erika Ábrahám
Computer Science Department
RWTH Aachen University
Aachen, Germany
abraham@cs.rwth-aachen.de

***Abstract*—Satisfiability Checking is a relatively young research area, aiming at the development of efficient software technologies for checking the satisfiability of existentially quantified logical formulas. Besides the success story of SAT solving for propositional logic, SAT-modulo-theories (SMT) solvers offer sophisticated solutions for different theories. When targeting arithmetic theories, SMT solvers also make use of decision procedures rooted in Symbolic Computation.**

In this paper we give a brief introduction to SMT solving, discuss differences to Symbolic Computation, and illustrate the potentials and obstacles for embedding Symbolic Computation techniques in SMT solving on the example of the Cylindrical Algebraic Decomposition.

Keywords*—*Satisfiability Checking, SAT-Modulo-Theories Solving, Algebra

I. Introduction

Logical formalisations are frequently used in different research and application areas to specify, e.g., combinatorial, scheduling, planning, verification, synthesis or optimisation problems. To find solutions to the logically specified problems, *decision procedures* can be used to check the satisfiability or validity of the logical formulas.

The first decision procedures were developed in mathematical logic for algebraic theories. The achievements in the area of *Symbolic Computation* are not only of theoretical nature: efficient implementations in powerful *computer algebra systems* allow applications to practically relevant problems.

Independently, another line of research evolved in computer science. *Satisfiability Checking* [6] attacks the problem from a different perspective: Whereas Symbolic Computation rather focuses on the satisfiability problem for sets of arithmetic[1] constraints, the main aim of Satisfiability Checking is to offer (practically) efficient solutions for logical formulas with a Boolean structure. Starting with propositional logic and easier theories like equalities and uninterpreted functions, nowadays quite a number of *SAT* and *SAT modulo theories* (*SMT*) *solvers* are available for a large number of theories (including arithmetic theories). These solvers use dedicated data structures and sophisticated heuristics to achieve an efficiency that allows their application to large industrial problems.

This work was supported by the H2020-FETOPEN-2016-2017-CSA project SC^2 (712689).

[1] *Algebraic* theories are often called *arithmetic* theories in the SMT-solving context.

For propositional logic, which is known to be NP-complete, *SAT solvers* use an elegant combination of enumeration, propagation [16] and resolution [17]. Conflict-driven clause-learning and non-chronological backtracking [30], and novel implementation techniques like the two-watched-literal scheme, restarts, cache performance optimisation, etc. brought further impressive progress.

SMT solvers [5], [28] enrich propositional SAT solvers with solver modules for different theories like equalities and uninterpreted functions, bit-vector arithmetic, floating-point arithmetic, array theory, difference logic, (quantifier-free) linear real/integer/mixed arithmetic, and (quantifier-free) nonlinear real/integer/mixed arithmetic.

For arithmetic theories, SMT solvers often make use of decision procedures developed in the area of Symbolic Computation. On the one hand, these procedures must be adapted and extended before they can be embedded in an SMT solver, what might cost quite some effort. On the other hand, we can exploit the main strengths of SMT solving and apply dedicated heuristics (for each decision procedure adjusted to its nature) to improve their performance in the SMT solving context. Further potential lies in the lifting of combinatorial problems from the theory level to the Boolean level, where learning can be used for an efficient search.

In this paper we give a short introduction to the theoretical foundations of Satisfiability Checking and a nutshell-overview about state-of-the-art SMT solvers including our own SMT solver `SMT-RAT` [15]. We discuss the efficient embedding of algebraic decision procedures in SMT solvers on the example of the *cylindrical algebraic decomposition* [12].

II. SAT-Modulo-Theories Solving

SAT-modulo-theories (*SMT*) *solving* aims at deciding the satisfiability of (usually quantifier-free) first-order logic formulas over some theories. Before we give an insight into the functioning of state-of-the-art SMT solvers, we first discuss the algorithmic basics of *SAT solving* for checking the satisfiability of propositional logic formulas without any underlying theory, which is a central component at the heart of SMT technologies.

A. SAT Solving

Propositional logic formulas are Boolean combinations of Boolean variables. The *satisfiability problem* for propositional logic, which is known to be NP-complete [14], is the problem

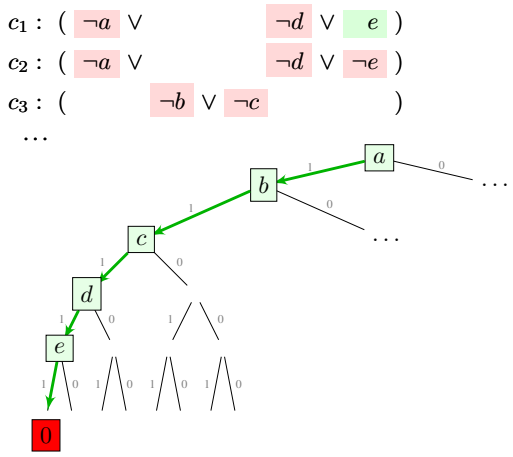

Fig. 1. Enumeration

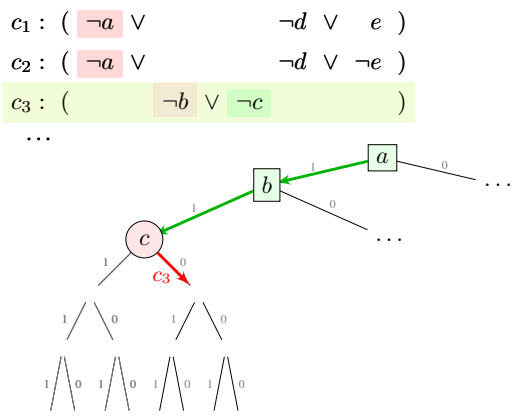

Fig. 2. Enumeration with propagation

to decide whether a propositional logic formula is satisfiable, i.e., whether there exists an assignment of values to the variables in the formula that evaluates the formula to true.

The perhaps most natural decision procedure for checking the satisfiability of such formulas is to *enumerate* all possible assignments and check whether one of them satisfies the formula. If a satisfying assignment has been found then the formula is satisfiable otherwise not.

Example 1: Figure 1 illustrates an enumeration path leading to an unsatisfying assignment.

A big step towards practical relevance was the combination of enumeration with *propagation* [16]. The formulas are first transformed into a satisfiability-equivalent formula in *conjunctive normal form*, that means into a formula being a conjunction of *clauses*, where each clause is a disjunction of *literals*, and each literal is either a variable or a negated variable. Instead of checking complete assignments, a *variable ordering heuristics* chooses a variable and a value, and the algorithm *decides* to try first assignments where the given variable is assigned the given value. In contrast to enumeration, before a next decision is taken, *Boolean constraint propagation* is used to detect implications of previously made decisions. Such an implication is detected if a clause is *unique*, which means that all of its literals but one are false and the last one is unassigned; in this case propagation assigns the value *true* to this last literal, implying that all other assignments, which would assign *false* to that last literal and thereby would lead to an unsatisfying assignment, are not considered. If propagation finds that all literals in a clause are *false* then a *conflict* is detected. As all extensions of the current partial assignment would be inevitably unsatisfying, enumeration stops at this point. *Backtracking* flips the variable value for the last decision that was not yet flipped and the search for a satisfying full assignment continues with propagating the flipped value; if no such decision exists then the formula is unsatisfiable.

Example 2: Figure 2 illustrates why the previous assignment from Example 1 can be avoided when combining enumeration with propagation: after deciding to assign *true* to both a and b, propagation in clause c_3 will imply that only the value *false* needs to be considered for c under the given partial assignment.

A further milestone in efficiency improvement merged a third basic component with enumeration and propagation. This component is *resolution*, which is a complete but for practical applications too costly decision procedure for propositional logic. The underlying idea is the following: if a clauses $c_1 = (l_1 \vee \ldots \vee l_n \vee x)$ contains a variable x and another clause $c_2 = (l'_1 \vee \ldots \vee l'_m \vee \neg x)$ its negation $\neg x$ then all assignments satisfying both clauses must satisfy either $l_1 \vee \ldots \vee l_n$ or $l'_1 \vee \ldots \vee l'_m$, i.e., they must satisfy the clause $(l_1 \vee \ldots \vee l_n \vee l'_1 \vee \ldots \vee l'_m)$. The merging point with enumeration and propagation is the way how conflicts are resolved: instead of flipping the last non-flipped decision, *conflict-driven clause learning* [30] applies resolution to follow back the chain of implications during propagation and to derive a clause which, if it would have been in the clause set at an earlier point, would have saved the procedure from running into the given conflict. The search jumps back to that earlier point and continues the search from there, starting with propagation to assure that the information in this new clause is exploited.

Example 3: Figure 3 illustrates the usage of resolution when a conflict appears. The conflicting clause is c_2, whose literals are all *false*. This holds especially for its last assigned literal $\neg e$, which is *false* because clause c_1 implied that e must be *true*. As $\neg e$ appears in c_2 and e appears in c_1, we can apply resolution. The resulting clause states that all assignments extending $a = \mathit{true}$ must set d to *false*. When resolving this conflict, the last two decisions $b = \mathit{true}$ and $d = \mathit{true}$ will be undone, the new clause will be added to the clause set, and propagation in the new clause will imply that $d = \mathit{false}$ must hold for all extensions of this partial assignment.

These algorithmical developments were implemented in numerous *SAT solvers*, which are nowadays able to solve practical problems with millions of variables. This efficiency led to the embedding of these tools into several approaches not only in different research areas like analysis, synthesis and optimisation, but also in industry, where SAT solvers are massively used for, e.g., digital circuit design and verification.

Remark 1: One of the enabling factors for this success was the active community support, resulting in a standardised input language, a huge benchmark collection, competitions

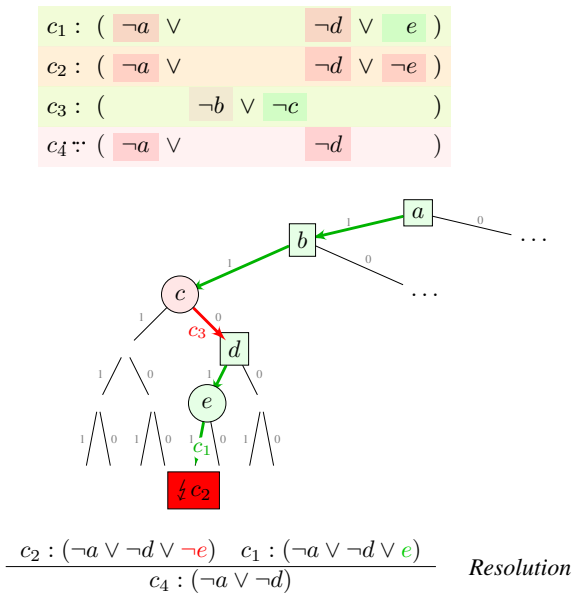

Fig. 3. Combining enumeration, propagation and resolution

since 2002, the SAT Live! forum as a community platform, the organisation of dedicated conferences, etc.

B. SMT Solving

As propositional logic is sometimes too weak for modelling, also more expressive logics and decision procedures for them were considered in the area *SAT-modulo-theories* (*SMT*) *solving*.

Here we describe the *less lazy* SMT solving approach, which takes as input a logical formula over some theories in conjunctive normal form, and builds its *Boolean skeleton* by replacing each theory atom by a fresh Boolean variable. The resulting formula is passed to a SAT solver to find solutions for the Boolean structure of the problem. After each decision and propagation loop of the SAT solver, it consults a *theory solver*, which implements a decision procedure for *sets* of theory constraints, asking whether those theory constraints whose abstraction variable is *true* and the negation of those whose abstraction variable is *false*[2] are together consistent. If they are consistent, the SAT solver continues its search. Otherwise, the theory solver returns an *explanation* for the inconsistency. Such an explanation is a tautology in the theory, usually (but not necessarily) stating that an inconsistent subset of the theory solver's passed constraints cannot hold together. The SAT solver makes use of this explanation to *refine* the Boolean abstraction by adding the abstraction of the explanation to its clause set, and handling this new clause similarly to a Boolean conflict.

Example 4: Figure 4 illustrates the SMT-solving procedure on an example. In the input formula, first the constraints $x < 0$, $x > 2$, $x^2 = 1$ and $x^2 < 0$ are abstracted by the Boolean variables a, b, c and d. The resulting propositional logic formula is passed to a SAT solver, who first might decide

[2]Those constraints whose abstraction variable is *false* do not need to be considered if the input formula does not contain any negated theory constraints.

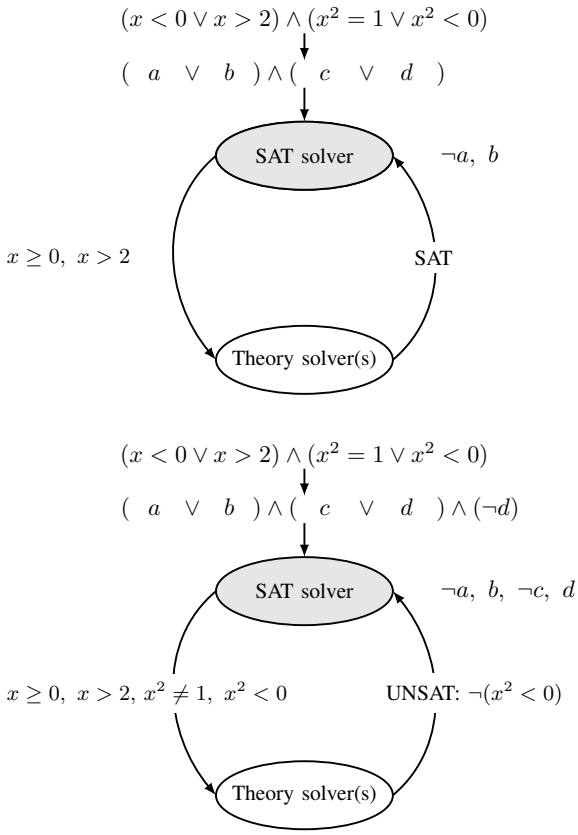

Fig. 4. Less lazy SMT solving example

a = *false*, implying by propagation b = *true*. Before the next decision, the consistency of the corresponding constraints $x \geq 0$ and $x > 2$ is checked by a theory solver. As they are consistent, the SAT solver might decide c = *false*, from which we imply d = *true* by propagation. The previous constraint set sent to the theory solver gets extended by the constraints $x^2 \neq 1$ and $x^2 < 0$. The set of these four constraints is inconsistent, an explanation being $\neg(x^2 < 0)$. The SAT solver will now learn the abstraction $\neg d$ of this explanation to refine the abstraction and to assure that in the remaining search d will not be set to *true* and thus that the search will not run into the same theory conflict again.

Figure 5 lists some decision procedures which could be implemented inside a theory solver for some arithmetic theories. The *simplex* algorithm, the *ellipsoid method* [27], the *Gauß* and the *Fourier-Motzkin variable elimination* algorithms, or the incomplete method of *interval constraint propagation* [23], [25] are applicable for linear real arithmetic, whereas in the linear integer case *cutting planes* [19], *branch and bound* [20], the *Omega test* [33], *bit-blasting* [11] and *interval constraint propagation* can be used. Satisfiability checking for non-linear real arithmetic, which has an exponential time-complexity, can be done using the *cylindrical algebraic decomposition* [12] method, methods based on *Gröbner bases* [35], the *virtual substitution* [36] method, or *interval constraint propagation*. Some incomplete methods for non-linear integer arithmetic, for which satisfiability is undecidable, are the generalised branch-and-bound [] and bit-blasting.

Linear real arithmetic:

- Simplex
- Ellipsoid method
- Gauß and Fourier-Motzkin variable elimination (mostly preprocessing)
- Interval constraint propagation (incomplete)

Non-linear real arithmetic:

- Cylindrical algebraic decomposition
- Gröbner bases (mostly preprocessing/simplification)
- Virtual substitution (incomplete)
- Interval constraint propagation (incomplete)

Linear integer arithmetic:

- Cutting planes, Gomory cuts
- Branch-and-bound (incomplete)
- Omega test
- Bit-blasting (eager)
- Interval constraint propagation (incomplete)

Non-linear integer arithmetic:

- Generalised branch-and-bound (incomplete)
- Bit-blasting (eager, incomplete)

Fig. 5. Some decision procedures for arithmetic theories

Examples for SMT solvers that can cope with arithmetic problems (either in a complete or in an incomplete manner) are `Alt-Ergo` [13], `AProVE` [11], `CVC4` [3], `iSAT3` [22], [34], `MathSAT5` [10], `MiniSmt` [38], `OpenSMT2` [9], `SMT-RAT` [15], `veriT` [7], `Yices2` [21], and `Z3` [32]. However, just a few of these solvers use algebraic decision procedures beyond the simplex method.

Implementations for algebraic decision procedures can be found in many computer algebra systems and other algebraic tools like, e.g., `CoCoALib` [1], `Maple` [29], `Mathematica` [37], `Maxima` [31], `QEPCAD` [8], `Reduce` [24] or `Singular` [18], just to mention a few.

Unfortunately, we cannot just plug in such an algebraic decision procedure as a theory solver into an SMT solver, because the SMT context puts certain special requirements on its theory solvers.

- The above example illustrates that the theory solver must check the consistency of sequences of problems with increasing constraint sets, until the problem gets inconsistent (or until a full solution is found). If the consistency check is done independently for each set, the computational effort would be very high. However, for most methods it is possible to re-use computations from the previous checks to reduce the computational effort. This ability is called *incrementality*.
- As explained above, if a set of constraints turns out to be inconsistent, in order to refine the Boolean abstraction, the theory solver should return an *explanation* to the SAT solver. As explanations of smaller size usually stronger prune the search space, it is advantageous to generate *minimal* (if any constraint is removed then the set is feasible) or even *minimum* (smallest under all) explanations.
- Finally, if a (Boolean or theory) conflict occurs, the SAT solver will undo some of its assignments. Dually, the theory solver should also be able to *backtrack* by removing some constraints from its passed constraints.

Further problems raise from the fact that most implementations are not available as a library, or they are not thread-safe and thus does not support parallelisation approaches.

As available implementations of algebraic decision procedures in general do not provide the above functionalities, there is a need to *adapt* them to satisfy the SMT solvers' requirements. Such adaptations are sometimes tricky, but they can lead to elegant novel solutions. The European Communication and Support Action *SC^2: Satisfiability Checking and Symbolic Computation* [2] aims at supporting such developments by bridging the two communities.

Such adaptations for decision procedures for non-linear arithmetic problems stay in the focus of our `SMT-RAT` solver [15], whose structure is depicted in Figure 6. Based on some standard libraries, the `CArL` library offers functionalities for basic arithmetic computations with arithmetic objects. Based on `CArL`, we implemented a set of different modules for decision procedures and methods for preprocessing, simplification etc. Most modules implement adapted algebraic decision procedures that satisfy the SMT requirements (incrementality, generation of explanations, backtracking). These `SMT-RAT` modules can be connected by a user-defined *strategy* to combine the strengths of different decision procedures. For example, the user could specify to use, as a theory solver, the simplex module if the problem is linear. If this is not the case, it could call the virtual substitution module, which offers quantifier elimination for variables that appear at most quadratically in the constraints. This elimination generates a set of sub-problems; if one of these sub-problems has a too high degree for the virtual substitution, the cylindrical algebraic decomposition module could be invoked to check the satisfiability of those subproblems.

Remark 2: Above we discussed only the less lazy technique. Besides *full* lazy solving that consults the theory solver only for complete assignments, also *eager* SMT solving can be applied to theories that are not more expressive than propositional logic; eager approaches transform an input formula into a satisfiability-equivalent propositional logic formula, whose satisfiability can be checked by a SAT solver. Furthermore, there are also SMT solving techniques, which more closely integrate theory-solving parts into the SAT-solving mechanism.

Similarly to SAT solving, a great driving force for the impressive development of SMT solving was the introduction of an SMT-LIB [4] as standard input language in 2004, what allowed also the collection of benchmark sets for different theories and the start of competitions in 2005.

III. Embedding the Cylindrical Algebraic Decomposition in SMT Solving

The *cylindrical algebraic decomposition* (*CAD*) method [12] is a complete quantifier elimination method for real algebra. As such, in the context of SMT-solving, the CAD

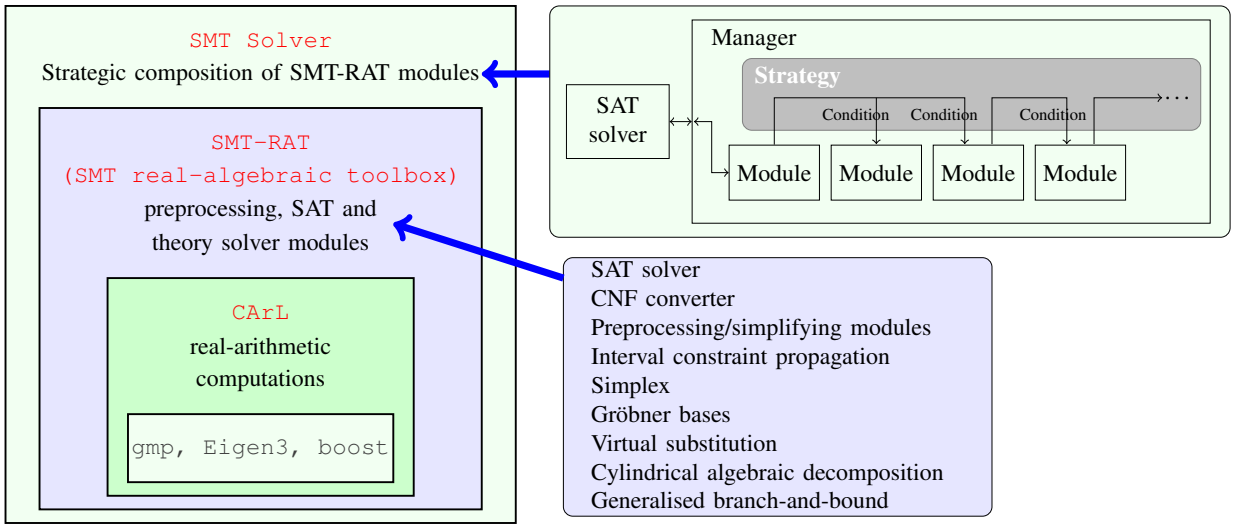

Fig. 6. The structure of our SMT−RAT library

method can be employed as a theory solver to check the satisfiability of non-linear real-arithmetic problems. In the following we first briefly explain the basic ideas behind the CAD method, and give two examples of how to exploit its functionalities in an SMT solver.

A. The Cylindrical Algebraic Decomposition

First we give some insights of how the CAD method can be used to check the satisfiability of a set of polynomial constraints, each of them comparing a polynomial to zero. Due to space limitations, we cannot give any mathematically precise description here, and we restrict ourselves to the intuition and the algorithmic aspects.

Intuitively, the CAD idea is based on the observation that the algebraic varieties of a set of polynomials (the sets of the points at which the polynomials evaluate to zero) partition the state space into finitely many connected regions, over which the polynomials are sign-invariant. Thus if we can construct such a partitioning for the polynomials in our polynomial constraints, we can take a single sample point from each partition and check whether one of those samples satisfies all polynomial constraints. If yes, we have found a solution, otherwise the set of constraints in unsatisfiable.

The construction of such a partitioning with the CAD method (the cylindrical algebraic decomposition of the state space) is illustrated on an example in Figure 7. In its first phase, the CAD method applies some projection operators, which generate for a set of n-dimensional polynomials in the variables $x_1, \ldots, x_n$ a set of $(n-1)$-dimensional polynomials in the variables $x_1, \ldots, x_{n-1}$ with the following property: at each n-dimensional point at which the number or the order of the zeros of the polynomials change when we move in the x_n-dimension (for example where the zero surfaces of two polynomials cross each other), the projection of this n-dimensional point to the $(n-1)$-dimensional space will be a zero of one of the projected $(n-1)$-dimensional polynomials.

The projection is repeatedly applied until we get one-dimensional polynomials, whose zeros $\xi_{1,1}, \ldots, \xi_{1,k_1}$ can be isolated and ordered (for example using Sturm sequences and Cauchy bounds). Thanks to the projection property, we know that for each two successive zeros $\xi_{1,i}$ and $\xi_{1,i+1}$ of the one-dimensional polynomials, the *cylinder* $(\xi_i, \xi_{i+1}) \times \mathbb{R}$ is a *stack* of (finitely many two-dimensional) sign-invariant regions of the two-dimensional polynomials. I.e., for any (one-dimensional) sample $s_1 \in (\xi_{1,i}, \xi_{1,i+1})$, the line $\{s_1\} \times \mathbb{R}$ will hit the same sign-invariant regions of the two-dimensional polynomials.

Based on this observation, in the second *construction* phase of the CAD method we choose $2k_1+1$ samples $s_{1,1} < s_{1,2} = \xi_{1,1} < s_{1,3} < s_{1,4} = \xi_{1,2} < \ldots < s_{1,2k_1} = \xi_{1,k_1} < s_{1,2k_1+1}$, one from each sign-invariant region of the one-dimensional polynomials. For each $s_{1,i}$ of these samples we proceed as follows. We substitute $s_{1,i}$ into the two-dimensional polynomials and isolate the zeros $\xi_{2,1}, \ldots, \xi_{2,k_2}$ of the resulting (now one-dimensional) polynomials. Again, we choose $2k_2+1$ samples $s_{2,1} < s_{2,2} = \xi_{2,1} < s_{2,3} < s_{2,4} = \xi_{2,2} < \ldots < s_{2,2k_2} = \xi_{2,k_2} < s_{2,2k_2+1}$ and proceed the same way for each sample $s_{2,j}$ by substituting $(s_{1,i}, s_{2,j})$ into the three-dimensional polynomials etc.

At the end, we get of a set of n-dimensional samples, which contains from each n-dimensional connected sign-invariant region of the n-dimensional polynomials at least one point, such that we can test whether the sign conditions on the n-dimensional polynomials, as posed by the polynomial constraints, can be satisfied by any of these points.

B. Making the CAD method SMT-compliant

First we explain how our SMT−RAT implementation adapts the CAD method to be SMT-compliant. We adapted the underlying data structures to make the CAD module incremental, being able to generate explanations, and being able to backtrack.

For *incrementality*, we first need to remark that, when applied to a set S of n-dimensional polynomials, projection operators generate a set of $(n-1)$-dimensional polynomials by applying projection to single polynomials and to pairs of

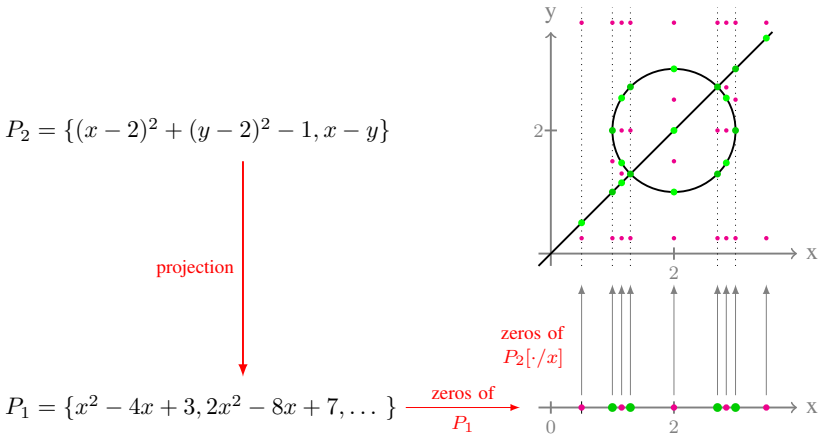

Fig. 7. The cylindrical algebraic decomposition on an example

polynomials from S. Once the CAD was applied to a set S of n-dimensional polynomials, if this set is extended with a set S' of new polynomials, to get all projections for $S \cup S'$, the old projection for S needs to be extended by (i) projections of polynomials $p \in S'$ and (ii) projections of pairs of polynomials from $(S \cup S') \times S'$. Thus the previous projection computations can be reused. As the projection sets increased monotonically, also the sample sets increase monotonically. Thus we can reuse the previous sample tree for S and extend it with samples that arise due to additional zeros of the newly computed projections.

This incremental approach is implemented in `SMT-RAT`. Efficient data structures are used for book-keeping which projections and constructions are completed and which are the ones that still need to be computed. As a side-effect, the CAD module of `SMT-RAT` is able to make partial computations in both phases, i.e., there is no need to compute the full projection before we can start constructing samples. Furthermore, it allows us to use sophisticated heuristics to guide the CAD search (e.g., to project lower-dimensional polynomials first). These properties allow to speed up satisfiability checks remarkably for satisfiable problem instances.

Explanations are generated by our `SMT-RAT` CAD module as follows. If all samples are computed and none of them satisfies the sign conditions specified by the input polynomial constraint set, we compute for each sample the set of all input polynomial constraints whose sign conditions are violated by that sample. To achieve an explanation of unsatisfiability, we compute a *covering set* that contains for each sample at least one polynomial constraint from its set of violated constraints. This covering set is an unsatisfiable subset of the input constraint set, as it contains for each sample at least one violated polynomial constraint. To generate smaller explanations, with some additional effort also minimal covering sets can be computed.

Backtracking can be supported by a dedicated data structure. For each projected polynomial and each constructed sample we can remember its "parents" (i.e., for the projection the polynomials that were projected, and for sample construction the polynomials whose zeros define the boundaries of the cell that the sample represents). As projection and sample construction are computed in a recursive manner, these dependencies define chains in the projection and sample trees. If a polynomial constraint gets removed by backtracking, all the chains that are rooted in the polynomial of that constraint are removed recursively. Note that we need to pay special attention to multiple "parents" (e.g., if the same polynomial was generated by several projection steps or if projected polynomials have a common zero); such object are removed only when all their "parents" disappear.

C. The Z3 Approach

As an example for a deeply novel adaptation of an algebraic decision procedure in an SMT solver, we briefly explain the embedding of the CAD method in the SMT solver `Z3` [32], [26].

Instead of the strictly distinguished handling of the Boolean structure of a problem and its theory constraints, this SMT approach combines Boolean decision, propagation and conflict resolution with theory decision, propagation and conflict resolution. Intuitively, the idea is as follows.

Given a real-arithmetic formula in conjunctive normal form whose satisfiability we want to decide, we assume a static variable order $x_1, \dots, x_n$ of the theory variables appearing in the theory constraints of the formula. Starting with x_1, the procedure iteratively assigns values $\alpha(x_i)$ to the theory variables x_i. Before assigning x_i, for all clauses with constraints in $x_1, \dots, x_i$, if none of their constraints in $x_1, \dots, x_{i-1}$ is satisfied by the current partial assignment, then one of their constraints in $x_1, \dots, x_i$ is chosen that should be satisfied by the forthcoming assignment to x_i. Substituting the current partial assignment $\alpha(x_1), \dots, \alpha(x_{i-1})$ into those chosen constraints results in a set of univariate polynomial constraints (in x_i). The procedure now checks whether they have a common solution. If yes, a common solution is assigned to x_i.

Otherwise, the method determines an infeasible subset S of those polynomial constraints and constructs a (partial) CAD just for the polynomials of the constraints in S. Next, the method computes an algebraic description D of the CAD cell that contains the current partial solution $\alpha(x_1), \dots, \alpha(x_{i-1})$. Finally, the method learns a new clause $(\wedge_{c \in S} c) \rightarrow \neg D$. After

backtracking and conflict resolution, the method continues as described below, until either a satisfying solution is found or a conflict is detected before making any decisions (in which case the problem is unsatisfiable).

The above embedding of the CAD method in SMT solving is extremely efficient. One reason is that due to the careful selection of variable values, lots of unsatisfying CAD cells are not considered at all. Furthermore, the complete CAD for all polynomials in the input formula will for most problems never be computed. Instead, CADs for relatively small constraint sets combined with learning help to exclude large unsatisfying cells from further search.

IV. Conclusion

After having presented the algorithmic basics of SMT solving and the embedding of decision procedures rooted in symbolic computation into the SMT solving context on the example of the cylindrical algebraic decomposition, we conclude the paper with a short discussion on where the current research development moves.

One thread focuses on further improving the *scalability* of SMT solvers by, e.g., performance optimisation (better lemmas, heuristics, cache behaviour, ...), further novel combinations of decision procedures, developing dedicated SMT solvers tuned to be efficient for certain problem types, or parallelisation. Highly interesting are also *functionality extensions* to enable, e.g., the generation of unsatisfiable cores also in the presence of theories, the generation of proofs in the case of unsatisfiability, the computation of interpolants, the handling of quantified arithmetic formulas, and last but not least linear and non-linear (global) optimisation.

References

[1] Abbott, J., Bigatti, A.M.: What is new in CoCoA? In: Proc. of ICMS'14. LNCS, vol. 8592, pp. 352–458. Springer (2014)

[2] Ábrahám, E., Abbott, J., Becker, B., Bigatti, A.M., Brain, M., Buchberger, B., Cimatti, A., Davenport, J.H., England, M., Fontaine, P., Forrest, S., Griggio, A., Kroening, D., Seiler, W.M., Sturm, T.: Sc^2: Satisfiability checking meets symbolic computation. In: Proc. of CICM'16. LNCS, vol. 9791, pp. 28–43. Springer (2016)

[3] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proc. of CAV'11. LNCS, vol. 6806, pp. 171–177. Springer (2011)

[4] Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)

[5] Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, chap. 26, pp. 825–885. IOS Press (2009)

[6] Biere, A., Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)

[7] Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: Proc. of CADE-22. LNCS, vol. 5663, pp. 151–156. Springer (2009)

[8] Brown, C.W.: QEPCAD: A program for computing with semi-algebraic sets using CADs. ACM SIGSAM Bulletin 37(4), 97–108 (2003)

[9] Bruttomesso, R., et al.: The OpenSMT solver. In: Proc. of TACAS'10. LNCS, vol. 6015, pp. 150–153. Springer (2010)

[10] Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. of TACAS'13, LNCS, vol. 7795, pp. 93–107. Springer (2013)

[11] Codish, M., Fekete, Y., Fuhs, C., Giesl, J., Waldmann, J.: Exotic semiring constraints. In: Proc. of SMT'13. EPiC Series, vol. 20, pp. 88–97. EasyChair (2013)

[12] Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Automata Theory and Formal Languages. LNCS, vol. 33, pp. 134–183. Springer (1975)

[13] Conchon, S., Iguernelala, M., Mebsout, A.: A collaborative framework for non-linear integer arithmetic reasoning in Alt-Ergo. In: Proc. of SYNASC'13. pp. 161–168. IEEE (2013)

[14] Cook, S.A.: The complexity of theorem-proving procedures. In: Proc. of STOC'71. pp. 151–158. ACM (1971)

[15] Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: An open source C++ toolbox for strategic and parallel SMT solving. In: Proc. of SAT'15. LNCS, vol. 9340, pp. 360–368. Springer (2015)

[16] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (1962)

[17] Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM 7(3), 201–215 (Jul 1960)

[18] Decker, W., Greuel, G.M., Pfister, G., Schönemann, H.: Singular 4-0-2 — A computer algebra system for polynomial computations. `http://www.singular.uni-kl.de` (2015)

[19] Dillig, I., Dillig, T., Aiken, A.: Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In: Proc. of CAV'09. LNCS, vol. 5643, pp. 233–247. Springer (2009)

[20] Doig, A.G., Land, B.H., Doig, A.G.: An automatic method for solving discrete programming problems. Econometrica 28, 497–520 (1960)

[21] Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Proc. of CAV'06. LNCS, vol. 4144, pp. 81–94. Springer (2006)

[22] Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. Journal on Satisfiability, Boolean Modeling and Computation 1(3-4), 209–236 (2007)

[23] Gao, S., Ganai, M., Ivančić, F., Gupta, A., Sankaranarayanan, S., Clarke, E.M.: Integrating ICP and LRA solvers for deciding nonlinear real arithmetic problems. In: Proc. of FMCAD'10. pp. 81–90. IEEE (2010)

[24] Hearn, A.C.: `REDUCE`: The first forty years. In: Proc. of A3L. pp. 19–24. Books on Demand GmbH (2005)

[25] Herbort, S., Ratz, D.: Improving the efficiency of a nonlinear-system-solver using a componentwise Newton method. Tech. Rep. 2/1997, Inst. für Angewandte Mathematik, University of Karlsruhe (1997)

[26] Jovanovic, D., de Moura, L.M.: Solving non-linear arithmetic. In: Proc. of IJCAR'12. LNAI, vol. 7364, pp. 339–354. Springer (2012)

[27] Khačiyan, L.C.: Polynomial algorithm for linear programming. Soviet Doklady 244, 1093–1096 (1979), typed translation

[28] Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer (2008)

[29] Maplesoft. `http://www.maplesoft.com/`

[30] Marques-silva, J.P., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48, 506–521 (1999)

[31] Maxima, a computer algebra system. `http://maxima.sourceforge.net/`

[32] de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. of TACAS'08. LNCS, vol. 4963, pp. 337–340. Springer (2008)

[33] Pugh, W.: The Omega test: A fast and practical integer programming algorithm for dependence analysis. Commun. ACM 8, 4–13 (1992)

[34] Scheibler, K., Kupferschmid, S., Becker, B.: Recent improvements in the SMT solver iSAT. In: Proc. of MBMV'13. pp. 231–241. Institut für Angewandte Mikroelektronik und Datentechnik, Fakultät für Informatik und Elektrotechnik, Universität Rostock (2013)

[35] Weispfenning, V.: A new approach to quantifier elimination for real algebra. In: Quantifier Elimination and Cylindrical Algebraic Decomposition. pp. 376–392. Texts and Monographs in Symbolic Computation, Springer (1998)

[36] Weispfenning, V.: Quantifier elimination for real algebra - the quadratic case and beyond. Appl. Algebra Eng. Commun. Comput. 8(2), 85–101 (1997)

[37] Wolfram, S.: Mathematica: A System for Doing Mathematics by Computer. Addison-Wesley (1988)

[38] Zankl, H., Middeldorp, A.: Satisfiability of non-linear (ir)rational arithmetic. In: Proc. of LPAR'10. LNAI, vol. 6355, pp. 481–500. Springer (2010)