SMT Solving for Arithmetic Theories: Theory and Tool Support

Erika Ábrahám and Gereon Kremer
RWTH Aachen University
Aachen, Germany
abraham@cs.rwth-aachen.de *gereon.kremer@cs.rwth-aachen.de*

***Abstract*—*Satisfiability checking* aims to develop algorithms and tools for checking the satisfiability of logical formulas. Driven by the impressive success of SAT solvers for propositional logic, *Satisfiability-Modulo-Theories* (*SMT*) *solvers* were developed to extend the scope also to different theories. Today, SMT solving is widely used in many applications, for example verification, synthesis, planning or product design automation.**

In this tutorial paper we give a short introduction to the foundations of SMT solving, describe some popular SMT solvers and illustrate their usage. We also present our own solver SMT-RAT, which was developed to support the strategic combination of different decision procedures, putting a focus on arithmetic theories.

***Keywords*-Logic, Algorithms, Algebra, Arithmetic, Computer aided analysis**

I. Introduction

The problem to decide the *satisfiability* of first order logic formulas is central in many research and applications areas like, e.g., program verification and termination analysis, symbolic execution, test-case generation, program and controller synthesis, combinatorial optimisation, product design, planning and scheduling, just to mention a few. For propositional logic formulas, since the early 1960s steadily improving algorithms were devised and implemented in *SAT solvers*, which nowadays can successfully solve huge industrial problems with millions of variables. Driven by this success, the scope was expanded to richer logics, including the theory of equalities and uninterpreted functions, array theory, bit-vector and floating-point arithmetic, difference logic, and linear and non-linear arithmetic, launching a new field of research named *Satisfiability Modulo Theories* (*SMT*) *solving* [1], [2].

In this tutorial paper we give a brief introduction to the foundations of SMT solving, discuss some of the theories that are supported by the SMT-LIB standard, describe a few SMT solvers, illustrate their usage, and mention some further main achievements of the community around SMT solving. We also showcase our SMT solver SMT-RAT as an example for solvers dedicated to arithmetic problems, and highlight its ability to strategically combine different decision procedures.

II. SMT Problems

Given a first-order logic formula over some theories, the *satisfiability problem* poses the question whether there exists a model that evaluates the formula to true. To solve the satisfiability problem for propositional logic, whose formulas are Boolean combinations of Boolean variables (propositions), we need to decide whether we can substitute Boolean values for the propositions appearing in a formula such that the formula evaluates to true. As a second example, for quantifier-free real arithmetic formulas, being Boolean combinations of polynomial equations and inequalities over real-valued variables, the problem consists of deciding whether we can assign satisfying real values to the formula's variables.

Different logics come with different expressive power for problem encoding on the one hand, but also with different worst-case complexity for solving the satisfiability problem on the other hand. Propositional logic, for which the satisfiability problem is known to be NP complete, is well-suited to model combinatorial problems over finite domains. Despite NP-completeness, state-of-the-art SAT solvers are able to solve impressively large problems stemming from practical applications. Our second example, quantifier-free real arithmetic is more expressive and can be used to model non-linear problems over continuous domains; despite a singly-exponential theoretical bound, currently available tools need in worst case doubly exponential time and space to solve the satisfiability problem, limiting their applicability. Therefore, the selection of the logic for problem encoding is an important decision: the logic must be powerful enough to encode the problem at hand, but it should not be unnecessarily expressive as otherwise solvers might fail to solve the problem in practically useful time.

SMT solving originally aims at solving *quantifier-free* first-order logic formulas over different theories, though there are also SMT solvers that support quantifiers. To illustrate the logical focus of SMT solving, we informally describe a few representative (quantifier-free) theories along with example formulas.

Real, integer and mixed real-integer arithmetic

$$x^2 + y^2 < 1 \land x \geq 1 \textit{ with } x, y \in \mathbb{R}$$

In these arithmetic theories, variables range over the reals or the integers. Besides addition and multiplication as functions, comparison predicates can be used to specify equalities and inequalities as atomic constraints. Formulas are Boolean combinations of such atomic constraints. The theories are called linear if they do not use multiplication and non-linear otherwise.

Equalities and uninterpreted functions

$$a = b \vee f(a,b) \neq f(b,a) \textit{ with } a,b \in D$$

If the satisfiability of a formula involving some functions is too hard to determine, sometimes it is helpful to simplify the problem by hiding the meaning of functions, resulting in *uninterpreted* functions. This relaxed formula is satisfiable if and only if there exist variable values and function interpretations that make the formula true. Note that the abstraction is over-approximating: if the abstraction is unsatisfiable then the original formula is also unsatisfiable, however, a satisfying solution is not necessarily consistent with the original function interpretations. In the latter case we can refine the abstraction by additionally encoding some satisfiability-relevant function properties.

Bit-vector arithmetic

$$x + 4 = (y << 1) \textit{ with } x,y \in \mathbb{B}^8$$

Bit-vector variables can be used to encode the behaviour of integer variables in programming languages – most notably including overflow semantics – using a fixed number of bits. As such, they are commonly used for program verification.

Floating-point arithmetic

$$x/0.0 = NaN \textit{ with } x \in \mathbb{F}_{23,8}$$

Similarly as bit-vectors for integers, floating-point variables can be used to encode the behaviour of floating-point variables in programs based on the IEEE 754 standard, including $\pm\infty$, NaN and different rounding modes. Also this theory is useful to encode program executions, most typically in the context of program analysis.

Other theories with SMT-support are suited to encode the behaviour of, e.g., arrays, enumerations, algebraic data types, tuples or even recursive data types.

For many applications, the problem encoding requires a combination of these theories. For example, both bit-vectors and floating-point constraints might be necessary to encode program executions. Therefore, some SMT solvers offer support for certain combinations of theories, based on a generic scheme to combine decision procedures for different theories if the theory combination fulfils certain properties.

III. SMT Solving

There are two fundamentally different SMT-solving approaches. *Eager SMT-solving*, commonly used for bit-vectors or uninterpreted functions, can be applied to theories that are not more expressive than propositional logic. This technique eliminates theory constraints first ("eagerly") by transforming formulas to satisfiability-equivalent propositional logic formulas. Afterwards, SAT solvers can be used to check the propositional logic formula for satisfiability.

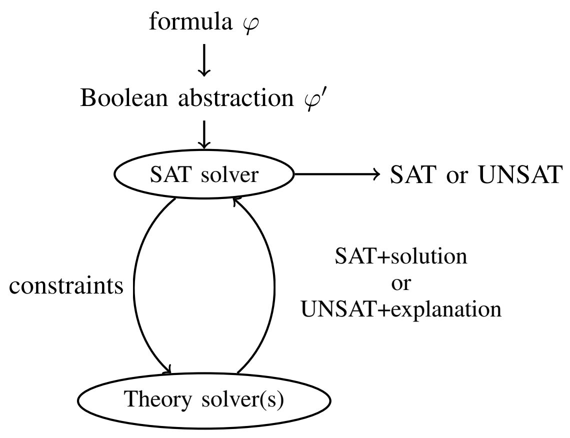

Figure 1. Lazy SMT solving

This approach has some similarities to polynomial reductions of NP complete problems: to solve a certain problem, we reduce it to another problem for which we know how to achieve the solution.

The other approach is called *lazy SMT-solving* and relies on the interaction of a SAT solver with one or more *theory solver(s)* as illustrated in Figure 1. For an input formula φ, first its *Boolean abstraction* φ' is computed by replacing every theory constraint c by a fresh proposition x_c. A SAT solver is employed to compute a solution for the abstraction φ'. Every such solution induces a set of theory constraints consisting of all constraints c for which x_c is assigned to *true*, and the negation $\neg c$ of all constraints c with x_c being *false*. The theory solver is invoked to decide whether this set of theory constraints is consistent in the theory. If the theory solver determines consistency then φ is satisfiable. Otherwise, the abstraction is refined by learning (the Boolean abstraction of) an explanation for the inconsistency, and the SAT solver searches for further solutions. If the abstraction turns out to be unsatisfiable then the input formula is unsatisfiable, too.

The above-described approach, also called *full lazy* SMT solving, consults the theory solver only for complete Boolean solutions. A more popular variant, *less lazy* SMT solving, invokes the theory solver more frequently on partial assignments to avoid that the SAT solver makes unnecessary effort to complete a partial assignment that is already conflicting in the theory.

Besides SMT solving, in this paper we will not further discuss other approaches for solving satisfiability problems, we only mention some of them at this point. *Linear programming* aims at finding an optimal solution (with respect to an objective) for a set of linear real-arithmetic constraints; prominent solvers include *glpk*, *CPLEX*, *Gurobi* or *SCIP*, but these methods are also integrated into *Excel*, *Maple*, *MATLAB* or *Mathematica*. *Constraint programming* has its roots in artificial intelligence and supports traditionally finite domains; some popular solvers include *Gecode*, *HaifaCSP*, *OR-Tools* or *sunny-cp*. *Answer set programming* came out of logic programming and aims at finding stable models for possibly quantified formulas; some examples for tools are *ASPeRiX*, *Clasp*, *DLV* and *Smodels*. Another related topic is (interactive) *theorem proving*, where axiomatic systems are defined and used for deduction; some well-known tools are *Coq*, *Isabelle/HOL*, *Lean*, *SPASS* and *Vampire*.

IV. The SMT Library

The increasing activities and success in the area of SMT solving raised the need for a common community framework, providing a platform for communication and exchange, and defining standards for input languages and solver interfaces. This need was satisfied by the *Satisfiability Modulo Theories Library* (*SMT-LIB*) initiative [3].

SMT-LIB established and maintains a selection of theory and logic specifications as well as the *SMT-LIB standard* which serves as the common input language for most SMT solvers. Furthermore, SMT-LIB provides a large collection of benchmarks, which are used not only for tool development and testing but also as the basis for the evaluations in annual competitions among SMT solvers.

A. Theories

Theories are the basic building blocks of SMT logics. As of December 2017, SMT-LIB specifies seven different theories: Core, ArraysEx, FixedSizeBitVectors, FloatingPoint, Ints, Reals and Reals_Ints. We exemplarily present the two theories Core and Reals_Ints here; for the others we refer to the SMT-LIB website. For all theories, datatypes and the syntax for functions and predicates are defined by a formal language. As the aim is to fix a standard *syntax* for inputs, the semantics is given only by informal descriptions.

The Core theory describes the Boolean framework of first-order logic. The excerpt shown in Figure 2 defines the Boolean domain with two constants *true* and *false*, as well as a number of operators below it. Note that *not* and *and* are defined as Boolean operators while = and *ite* have a *type parameter* `A` and thus can be instantiated with different argument types.

The Reals_Ints theory (see Figure 3) describes all theory constructs related to either reals or integers. This includes numeric constants, arithmetic operations like addition and multiplications and comparisons, but also functions like *mod*, *abs* or conversions *to_real* and *to_int*.

These standard theory specifications sometimes differ from what actual solvers implement for a variety of reasons.

Solvability SMT solvers rely on embedded decision procedures, which sometimes cannot handle the full theory but only a fragment of it. Hence, sometimes only inputs with constraints that are suitable for the given algorithm can be considered. Problematic functions are for example *div*

```
:sorts ((Bool 0))
:funs (
  (true Bool)
  (false Bool)
  (not Bool Bool)
  (and Bool Bool Bool :left-assoc)
  ...
  (par (A) (= A A Bool :chainable))
  (par (A) (ite Bool A A A))
  ...
)
```

Figure 2. SMT-LIB Core theory

```
:sorts ((Int 0) (Real 0))
:funs (
  (NUMERAL Int)
  (- Int Int)
  (- Int Int Int :left-assoc)
  (+ Int Int Int :left-assoc)
  (* Int Int Int :left-assoc)
  (div Int Int Int :left-assoc)
  (mod Int Int Int)
  (abs Int Int)
  (<  Int Int Bool :chainable)
  ...
  (DECIMAL Real)
  (- Real Real)
  (- Real Real Real :left-assoc)
  (+ Real Real Real :left-assoc)
  (* Real Real Real :left-assoc)
  (/ Real Real Real :left-assoc)
  (<  Real Real Bool :chainable)
  ...
  (to_real Int Real)
  (to_int Real Int)
  ...
)
```

Figure 3. SMT-LIB Reals_Ints theory

and *abs*. Though sometimes these functions can be reduced to constructs from the supported theory fragment, not all solvers implement such reductions.

Interoperability Under the given definition in the standard, *Int* and *Real* can only interact after being converted using *to_real* or *to_int* which is highly inconvenient. However, as *Int* is a subset of *Real* and all the common arithmetic operations and comparisons are compatible for the two domains, most solvers allow to seamlessly move between the two domains.

Underspecification In some cases the semantics is underspecified and allows for a certain freedom in interpretation. E.g., the result of a division by zero can be defined arbitrarily as long as division remains a total function. This can lead to differing interpretations by different solvers, which is why solvers may also reject such inputs in practice.

Unexpected specification Considering the datatype *Real*, the user would expect a solver to handle the full set of real numbers. While this expectation is usually met for *Int*,

```
(logic QF_LRA
  :theories (Reals)
  :language
  "Closed quantifier-free formulas built over
      arbitrary expansions of the Reals signature
      with free constant symbols, but containing
      only linear atoms, that is, atoms with no
      occurrences of the function symbols * and /,
       except as specified the :extensions
      attribute."

  :extensions
  "Terms with _concrete_ coefficients are also
      allowed, that is, terms of the form c, (* c
      x), or (* x c)  where x is a free constant
      and c is an integer or rational coefficient.
  - An integer coefficient is a term of the form m
       or (- m) for some numeral m.
  - A rational coefficient is a term of the form d
      , (- d) or (/ c n) for some decimal d,
      integer coefficient c and numeral n other
      than 0.
  ")
```

Figure 4. SMT-LIB QF_LRA logic

representing any arbitrary value from *Real* is not possible. Upon closer inspection, the specification actually states that *Real* does not contain all real numbers, but only the algebraic numbers.

B. Logics

Based on the above theories, SMT-LIB defines a large, but by no means exhaustive, list of logics. A logic defines a fragment of a first-order logic over one or more theories. If a certain logic is specified, this usually means that there is an interest in solving that specific logic, either because there are effective solving techniques handling exactly this fragment or because some applications need exactly this modelling language.

Common restrictions are to disallow quantification (indicated by a leading *QF_* for “quantifier-free”) or to remove certain functions from the signature. A few examples for widely used logics are *QF_LRA* (quantifier-free linear real arithmetic), *QF_NRA* (quantifier-free non-linear real arithmetic), *QF_BV* (quantifier-free bit-vectors) or *UFLRA* (quantified uninterpreted functions and linear real arithmetic). The language restrictions are given in an informal format, but aim to be precise (see Figure 4 for the example of *QF_LRA*).

C. SMT-LIB language

Based on the definitions of theories and logics, the SMT-LIB standard provides a specification of an input language for SMT problems. This language is designed to be easy to parse and to extend. Some simple examples are shown in Figures 5, 6 and 7 together with possible results (listed as comments starting with a semicolon).

```
(set-logic QF_UF)
(declare-const p Bool)
(assert (and p (not p)))
(check-sat) ; unsat
```

Figure 5. Example propositional logic problem

```
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(assert (> (+ x y) 1))
(assert (= x (+ y 1)))
(check-sat) ; sat
(get-model) ; x = 2, y = 1
```

Figure 6. Example QF_LIA problem

Figure 5 shows an example for a propositional logic problem. After specifying the logic (where we use *QF_UF* for propositional logic as there is no pure propositional logic in the SMT standard) and some variables, we can use `assert` to hand over formulas to the solver and check for satisfiability using `check-sat`.

Figure 6 shows a QF_LIA example, where `get-model` additionally asks for a model of the specified problem. A number of other commands are available, for example to compute *unsatisfiable cores*, solve a sequence of problems *incrementally* or perform *optimisations*.

A combined example is shown in Figure 7. This example uses two integer variables x and y. We first assert $f_1 : x < 1$ and $f_2 : y > 2$ and then open a new scope in which we additionally assert that $f_3 : x = y$. Now we ask the solver to check whether the given formula $x < 1 \wedge y > 2 \wedge x = y$ is satisfiable. Obviously it is not and using `get-unsat-core` we can obtain a reason for that. In this case, the reason will be f_1, f_2, f_3. Now we can leave the scope which will

```
(set-logic QF_LIA)
(set-option :produce-unsat-cores true)

(declare-const x Int)
(declare-const y Int)

(assert (! (< x 1) :named f1))
(assert (! (> y 2) :named f2))
(push 1)
(assert (! (= x y) :named f3))
(check-sat) ; unsat
(get-unsat-core) ; (f1 f2 f3)
(pop 1)
(check-sat) ; sat
(get-model) ; x = 0, y = 3
(assert (! (< y 5) :named f4))
(maximize (+ x y))
(check-sat) ; sat
(get-model) ; x = 0, y = 4
```

Figure 7. Example QF_LIA problem

remove f_3 and again ask for satisfiability. This time, the formula $x < 1 \wedge y > 2$ is satisfiable and we get a model using `get-model`, for example $x = 0, y = 3$. Additionally, we could now ask for an optimal solution: we add a new constraint $y < 5$ and tell the solver to maximise $x + y$. The problem is still satisfiable and the model should be $x = 0, y = 4$.

Note that it highly depends on the combination of solver and logic whether a specific feature is available or works as expected. While most of the commands shown in Figure 7 are part of the SMT-LIB language, every solver is free to offer custom commands. For example, `maximize` is such a custom extension, though it is implemented consistently by a variety of solvers. Even if part of the SMT-LIB language, many problems that can be expressed exceed the functionalities of current solvers. Generation of unsatisfiable cores is oftentimes implemented rather naively. Optimisation on the other hand is nicely done for linear arithmetic by several solvers, but non-linear optimisation is much harder and rarely supported, and for other theories like e.g. uninterpreted functions, its meaning is not even clearly specified.

D. Benchmarks

The SMT-LIB initiative hosts a large selection of open-access benchmark inputs. This does not only allow researchers to test their solvers, but also gives a common basis to compare applicability and efficiency of different approaches. Though doubts about the representativeness of the benchmarks and the generality of results obtained from these benchmarks are always justifiable, the selection of benchmarks proves to be very valuable in practice.

As of 2017, there are in total more than 257.000 benchmarks available for different logics. The quantifier-free arithmetic theories have about 10.000 linear benchmarks (in *QF_LRA*, *QF_LIA*, *QF_RDL* and *QF_IDL*) and about 35.000 non-linear benchmarks (in *QF_NRA* and *QF_NIA*). These benchmarks are regularly updated, hence new applications can submit interesting benchmarks and in this way implicitly guide future developments in SMT-solving.

V. Theory Solving Modules

The lazy SMT-solving approach presented in Section II provides a framework that can be instantiated with different theories. Independently of the underlying theory, the SAT solver takes over responsibility for the Boolean structure of the input problem, such that the theory solver needs to deal with *sets* (*conjunctions*) of theory constraints only and may even offload internal case-splitting to the SAT solver. We now discuss some decision procedures as potential candidates to be implemented as theory solver modules for arithmetic theories.

The best-known approach to solve linear real-arithmetic constraint systems is the *simplex* algorithm. Although it exhibits an exponential worst-case complexity, it performs very good in practice and is used in large industrial settings to solve linear optimisation problems. In [4] a variant of simplex suitable for SMT solving was presented, which was extended in [5] to allow also optimisation queries. Other extensions target linear integer arithmetic [6] or non-linear arithmetic via linearisation techniques [7].

Another approach for solving linear real-arithmetic constraint systems is the *Fourier-Motzkin* variable elimination. Though it is usually inferior to the simplex algorithm performance-wise, in contast to simplex it can be used also for quantifier elimination. For example, [8] employs Fourier-Motzkin variable elimination in the context of quantified linear real arithmetic.

As for non-linear arithmetic, only a few algebraic procedures were adapted as theory solvers. The *cylindrical algebraic decomposition* due to Collins [9] is a complete quantifier elimination procedure for real-algebraic problems and has spawned a lot of research in computer algebra. It was adapted and embedded as a theory solver in the SMT-RAT solver [10] and has seen various improvements since [11], [12]. An elegant adaptation of this algorithm, which significantly differs from the presented SMT-solving approach, was published in [13] an implemented in the Z3 solver.

The *virtual substitution* due to Weispfenning [14] is a quantifier elimination procedure that is limited to an arbitrary but fixed polynomial degree, in practice degree two or three for most solvers. Nevertheless, it proved to be very effective in practical SMT solving both on its own and as a pre-processor to the CAD method [15].

Gröbner bases can be used to check the consistency of polynomial equation systems in the complex domain. Although for the real domain they can only witness unsatisfiability in general, they can be a valuable tool for certain problems as shown in [16].

Last but not least, also *interval constraint propagation* [17], a technique based on interval arithmetic, can be used for solving non-linear real arithmetic problems [18], [19].

The sheer number of decision procedures rooted in different areas and embedded into SMT solvers could suggest that such embeddings are merely a matter of implementation. It indeed is for a very naive embedding, but SMT solving usually sees drastic speedups and becomes feasible in practice only if certain features are available. Unfortunately, many decision procedures do not have these features, therefore they need to be adapted in order to enable an efficent SMT embedding.

We call a theory solver *SMT compliant* if it is *incremental*, can generate *infeasible subsets* or other types of *explanations* for inconsistency, and supports *backtracking*. These notions are explained in the following.

In *less lazy* SMT-solving, the constraint sets of most theory calls are extensions of the previous theory call by constraints corresponding to the recent extension of an incomplete Boolean assignment since the last theory call. If the theory solver does not check each set independently but instead it re-uses information gained during previous checks then we say that it works *incrementally*.

If the theory solver determines inconsistency of its received constraints then the SAT solver needs to exclude from its further search all Boolean solutions that would induce that the given constraints should hold together. It could of course do so by adding the information that the given constraints together should not be selected. However, inconsistency is often caused just by a small subset of the constraints. Therefore, it is advantageous if the theory solver provides an *explanation* for the inconsistency in form of a theory lemma (tautology), which is often given as an *infeasible subset* of the input constraints. Unsurprisingly, smaller explanations often lead to stronger performance benefits.

After such a conflict has occurred, the SAT solver backtracks by undoing some of its decisions about which part of the search space should be explored next. Whenever this happens, some constraints will also be removed from the constraint set, which the theory solver needs to check for consistency. We say that a theory solver can *backtrack* if it can remove single constraints from its current set of constraints, without loosing all previously collected information about the solving process. If this is possible then, after backtracking, the theory solver can continue its search incrementally, without a full restart of its computations. On the one hand, the ability to backtrack oftentimes yields a significant performance increase at the point of backtracking, but on the other hand it might also cause a remarkable computational overhead for bookkeeping.

VI. SMT SOLVERS

There is a large number of SMT solvers available. Here we briefly describe some of them with functionalities for non-linear real arithmetic. More solvers for different logics can be found on the SMT-LIB home page [3] and on the home pages of the SMT competitions.

AProVE [20] has been originally developed for program termination analysis. In that context, it needs to solve a large number of non-linear integer problems quickly, for which it uses bit-blasting. AProVE has been the winner in the respective logic in several editions of the SMT competition.

CVC4 [21] is one of the largest open-source solvers with a broad functionality, including linear arithmetic, bit-vectors, arrays, strings, uninterpreted functions and several combinations of these theories. CVC4 tackles non-linear arithmetic with a bit-blasting approach similar to [20] for integer problems and with different linearisation techniques for problems over the reals. For some theories also quantification is supported.

iSAT3 [22] is based on the fast but incomplete interval constraint propagation method and is able to handle, beyond the usual arithmetic constructs, also extensions like trigonometric, exponential and logarithmic functions in order to verify industrial *C* programs. Unfortunately, iSAT3 does not support the SMT-LIB language but it comes with an own custom format.

MiniSmt [23] was developed for termination analysis. It interfaces the SAT solver *MiniSat* [24], the SMT solver *Yices*, and libraries from *TTT2*. The approach is incomplete and solves satisfiable instances only, but if it succeeds then it computes satisfying models quickly. Besides bit-blasting for integers, also non-integrals are reduced to the integral setting using an extended version of rational arithmetic.

raSAT [25] employs a variation of interval constraint propagation techniques for solving non-linear arithmetic problems in an incomplete manner. Its approach combines over-approximation and under-approximation to effectively solve both satisfiable and unsatisfiable problems.

Yices 2 [26] is an SMT solver supporting the theories of uninterpreted functions with equality, real and integer arithmetic, bitvectors, scalar types, and tuples. The solver module for non-linear real arithmetic is based on the same technique as implemented in *Z3*.

Z3 [27] is one of the most well-known SMT solvers. Besides arithmetic theories, Z3 supports the theories of bit-vectors, arrays, datatypes, equalities and uninterpreted functions, and quantifiers. For non-linear real arithmetic, it instantiates an elegant variation of the cylindrical algebraic decomposition method in the model-constructing satisfiability calculus framework [13].

VII. THE SMT-RAT SOLVER

In this section we introduce our free and open-source solver *SMT-RAT* (*SMT Real Arithmetic Toolbox*) [28], which was developed with a focus on non-linear arithmetic problems.

One of its main features is that it offers a library of solver *modules* which can be combined strategically by the user. The combination is specified by a *strategy*, which defines a tree of modules. The root node receives the input problem, which it might solve alone, but it might also pass on (sub-)problems to its children. Different types of problems can be passed on to different children, specified by syntactic conditions on the (sub-)problems. Examples for such syntactic conditions are for example whether the (sub-)problem contains uninterpreted functions or bit-vectors, for arithmetic problems only equalities, only strict inequalities, only linear constraints, or non-linear constraints of certain degrees. The conditions are evaluated at runtime, and if the conditions are satisfied for several children then they can be called in parallel. This way, multiple backends can work on the same task, providing an effective way to exploit multi-core processors.

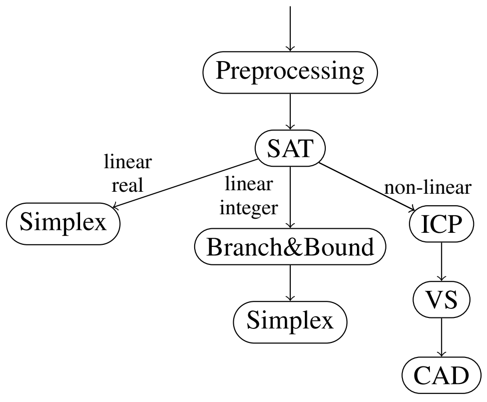

Figure 8. Example SMT-RAT strategy

```
class Strategy: public Manager {
public:
  Strategy() {
    setStrategy(
      addBackend<PP<PPSetting>>(
        addBackend<SAT<SATSetting>>({
          addBackend<Simplex<SimplexReal>>(
          ).condition(isLinear && !containsInt),
          addBackend<BB<BBSetting>>(
            addBackend<Simplex<SimplexInt>>()
          ).condition(isLinear && containsInt),
          addBackend<ICP<ICPSetting>>(
            addBackend<VS<VSSetting>>(
              addBackend<CAD<CADSetting>>()
            )
          ).condition(!isLinear)
        })
      )
    );
  }
};
```

Figure 9. Implementation of a strategy

Every module internally maintains a set of its received formulas and implements four methods: one for receiving a new input formula incrementally (*add*), one for backtracking (*remove*), one for checking the consistency of the current problem (*check*) and one for providing models for consistent problems or providing explanations for inconsistent problems (*updateModel*). The result of a consistency check can be *SAT*, *UNSAT* or *UNKNOWN*. The latter might be returned not only by incomplete methods but also if branching should be lifted from the theory to the SAT level, e.g. by the branch-and-bound method (where the branching is communicated in the form of a lemma).

SMT-RAT currently offers the following modules: a parser for the SMT-LIB language, several pre-processing techniques, a SAT solver, and a whole range of theory solvers including implementations for the simplex method, the Fourier-Motzkin variable elimination, interval constraint propagation, Gröbner bases, the virtual substitution and the cylindrical algebraic decomposition.

SMT-RAT strategies allow natural cooperation schemes with a great flexibility for solving techniques to interact in novel ways. Typically, parsing and abstracting, SAT preprocessing and solving, theory preprocessing, and fast but incomplete theory modules preceed heavy theory decision procedures with high computational costs.

An example strategy is shown in Figure 8. Every strategy implicitly starts with the parser that forwards the input to the strategy itself. In this example, after parsing, the first module performs some pre-processing techniques and forwards the – possibly simplified – formula to the SAT solver. The SAT solver forwards theory calls to its backends. Depending on the properties of the involved constraints, the strategy chooses a specific backend for every individual call. If all constraints are linear and contain real-valued variables only then simplex is used. If the current formula is linear but contains also integer variables then a simplex-based branch-and-bound approach is used. Otherwise, the formula is non-linear and a layered approach using interval constraint propagation, the virtual substitution and the cylindrical algebraic decomposition methods is applied.

The strategy does not only define a certain module to be used at some point, but also equips it with a configuration called *setting*. In our example, the two instances of the simplex module may use different heuristics tuned to the real respectively the integer domain.

The implementation of strategies is rather simple as illustrated in Figure 9 by the example of our example strategy above (the figure simplifies the real implementation by shortening class names and using placeholders for the conditions). Once a strategy is implemented this way, a SMT solver can be compiled which will behave accordingly.

The overall design has proven to be simple enough to be used and extended without prior knowledge in solver implementation. This allows us to use SMT-RAT also in education. In the context of practical courses and Bachelor and Master theses, numerous students implemented different theory solver modules within this framework, for example solving modules based on Gröbner bases [29] and interval constraint propagation [30], cutting techniques for linear integer arithmetic [31], infeasible subset generation for equalities with uninterpreted functions [32], bit-vectors including bound inference that can also be used for non-linear integer problems [33], and last but not least novel solving techniques for pseudo-Boolean problems [34].

References

[1] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

[2] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185.

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

[4] B. Dutertre and L. De Moura, "A fast linear-arithmetic solver for DPLL(T)," in *Proc. of CAV'06*. Springer, 2006, pp. 81–94.

[5] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νZ - An optimizing SMT solver," in *Proc. of TACAS'15*, vol. 15, 2015, pp. 194–199.

[6] D. Jovanovic and L. M. de Moura, "Cutting to the chase solving linear integer arithmetic." in *Proc. of CADE-23*, vol. 6803. Springer, 2011, pp. 338–353.

[7] C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "SAT modulo linear arithmetic for solving polynomial constraints," *Journal of Automated Reasoning*, vol. 48, no. 1, pp. 107–131, 2012.

[8] N. Bjørner, "Linear quantifier elimination as an abstract decision procedure," in *Proc. of IJCAR'10*. Springer, 2010, pp. 316–330.

[9] G. E. Collins, "Quantifier elimination for real closed fields by cylindrical algebraic decompostion," in *Proc. of Automata Theory and Formal Languages*. Springer, 1975, pp. 134–183.

[10] F. Corzilius, U. Loup, S. Junges, and E. Ábrahám, "SMT-RAT: An SMT-compliant nonlinear real arithmetic toolbox," in *Proc. of SAT'12*. Springer, 2012, pp. 442–448.

[11] U. Loup, K. Scheibler, F. Corzilius, E. Ábrahám, and B. Becker, "A symbiosis of interval constraint propagation and cylindrical algebraic decomposition," in *Proc. of CADE-24*. Springer, 2013, pp. 193–207.

[12] G. Kremer, F. Corzilius, and E. Ábrahám, "A generalised branch-and-bound approach and its application in SAT modulo nonlinear integer arithmetic," in *Proc. of CASC'16*. Springer, 2016, pp. 315–335.

[13] D. Jovanović and L. De Moura, "Solving non-linear arithmetic," *ACM Communications in Computer Algebra*, vol. 46, no. 3/4, pp. 104–105, 2013.

[14] V. Weispfenning, "Quantifier elimination for real algebrathe quadratic case and beyond," *Applicable Algebra in Engineering, Communication and Computing*, vol. 8, no. 2, pp. 85–101, 1997.

[15] F. Corzilius and E. Ábrahám, "Virtual substitution for SMT-solving," in *Proc. of FCT'11*. Springer, 2011, pp. 360–371.

[16] S. Junges, U. Loup, F. Corzilius, and E. Ábrahám, "On Gröbner bases in the context of satisfiability-modulo-theories solving over the real numbers," in *Proc. of ICAI'13*. Springer, 2013, pp. 186–198.

[17] P. Van Hentenryck, D. McAllester, and D. Kapur, "Solving polynomial systems using a branch and prune approach," *SIAM J. Numer. Anal.*, vol. 34, no. 2, pp. 797–827, 1997.

[18] S. Gao, S. Kong, and E. M. Clarke, "dReal: An SMT solver for nonlinear theories over the reals," in *Proc. of CADE-24*. Springer, 2013, pp. 208–214.

[19] S. Schupp, "Interval constraint propagation in SMT-compliant decision procedures," Master's thesis, RWTH Aachen University, 2013.

[20] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl, "SAT solving for termination analysis with polynomial interpretations," in *Proc. of SAT'07*. Springer, 2007, pp. 340–354.

[21] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proc. of CAV'11*. Springer, 2011, pp. 171–177.

[22] K. Scheibler, S. Kupferschmid, and B. Becker, "Recent improvements in the SMT solver iSAT," in *Proc. of MBMV'13*. Universität Rostock, 2013, pp. 231–241.

[23] H. Zankl and A. Middeldorp, "Satisfiability of non-linear (ir)rational arithmetic," in *Proc. of LPAR'10*. Springer, 2010, pp. 481–500.

[24] N. Sörensson and N. Eén, "MiniSat 2.1 and MiniSat++ 1.0 – SAT Race 2008 Editions," *SAT*, p. 31, 2009.

[25] T. Van Khanh, V. X. Tung, and M. Ogawa, "rasat: SMT for polynomial inequality," *Formal Methods in Systems Design*, vol. 51, no. 3, pp. 462–499, 2017.

[26] B. Dutertre, "Yices 2.2," in *Proc. of CAV'14*. Springer, 2014, pp. 737–744.

[27] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. of TACAS'08*. Springer, 2008, pp. 337–340.

[28] F. Corzilius, G. Kremer, S. Junges, S. Schupp, and E. Ábrahám, "SMT-RAT: An open source c++ toolbox for strategic and parallel SMT solving," in *Proc. of SAT'15*. Springer, 2015, pp. 360–368.

[29] S. Junges, "On Gröbner bases in SMT-compliant decision procedures," Bachelor's thesis, RWTH Aachen University, 2012.

[30] S. Schupp, "Interval constraint propagation in SMT-compliant decision procedures," Master's thesis, RWTH Aachen University, 2013.

[31] D. Hütter, "SMT solving for linear integer arithmetic," Bachelor's thesis, RWTH Aachen University, 2014.

[32] L. Neuberger, "Generation of infeasible subsets in less-lazy SMT-solving for the theory of uninterpreted functions," Bachelor's thesis, RWTH Aachen University, 2015.

[33] A. Krüger, "Bitvectors in SMT-RAT and their application to integer arithmetic," Master's thesis, RWTH Aachen University, 2015.

[34] M. Grobelna, "SAT-modulo-theories solving for pseudo-Boolean constraints," Bachelor's thesis, RWTH Aachen University, 2017.