Satisfiability Checking: Theory and Applications

Erika Ábrahám(✉) and Gereon Kremer

RWTH Aachen University, Aachen, Germany
abraham@cs.rwth-aachen.de

Abstract. Satisfiability checking aims to develop algorithms and tools for checking the satisfiability of existentially quantified logical formulas. Besides powerful SAT solvers for solving propositional logic formulas, sophisticated SAT-modulo-theories (SMT) solvers are available for a wide range of theories, and are applied as black-box engines for many techniques in different areas. In this paper we give a short introduction to the theoretical foundations of satisfiability checking, mention some of the most popular tools, and discuss the successful embedding of SMT solvers in different technologies.

1 Introduction

First-order-logic is a powerful modelling formalism frequently used to specify problems in different areas like verification, termination analysis, test case generation, controller synthesis, equivalence checking, combinatorial tasks, scheduling, planning, and product design automation and optimisation, just to mention a few well-known examples. Once the problem is formalised, algorithms and their implementations are needed to check the validity or satisfiability of the formulas, and in case they are satisfiable, to identify satisfying solutions. Algorithms to solve this problem are called *decision procedures*.

In mathematical logic, in the early 20th century some novel decision procedures were developed for arithmetic theories. With the advent of computer systems, big efforts were made to provide automated solutions in form of practically feasible implementations of decision procedures. In the area of symbolic computation, this development led to computer algebra systems supporting all kinds of scientific computations. Another line of research, *satisfiability checking* [10], started to focus on the more specific aim of checking the *satisfiability* of *existentially* quantified logical formulas.

For Boolean propositional logic, which is known to be NP-complete, in the late '90s impressive progress was made in the area of satisfiability checking, resulting in powerful *SAT solvers*. The first idea used *resolution* for quantifier elimination [30], but it had serious problems with the explosion of the memory requirements with increasing problem size. A combination of *enumeration* and *Boolean constraint propagation* [29] brought important enhancements. Another major improvement was achieved by a novel combination of enumeration, Boolean constraint propagation and resolution, leading to *conflict-driven*

R. De Nicola and E. Kühn (Eds.): SEFM 2016, LNCS 9763, pp. 9–23, 2016.
DOI: 10.1007/978-3-319-41591-8_2

clause-learning and *non-chronological backtracking* [50]. Later on, this impressive progress was continued by novel efficient implementation techniques (*e.g.*, sophisticated decision heuristics, two-watched-literal scheme, restarts, cache performance, etc.). Also different extensions are available, for example QBF solvers for quantified Boolean formulas, Max-SAT solvers to find solutions which satisfy a maximal number of clauses, or #SAT solvers to find all satisfying solutions of a propositional logic formula. State-of-the-art SAT solvers are able to solve such impressively large propositional logic problems that they became not only applicable in industry, but one of the most important engines in, *e.g.*, hardware verification.

Driven by this success, the satisfiability checking community started to enrich propositional SAT solvers with solver modules for different theories. Nowadays, sophisticated *SAT-modulo-theories* (*SMT*) solvers are available for a wide range of 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) non-linear real/integer/mixed arithmetic. Latest research led also to functional extensions, going beyond satisfiability checking for existentially quantified formulas towards providing an unsatisfiable core for unsatisfiable input problems, proof of unsatisfiability, solving quantified formulas, and solving optimisation problems. Some solvers also exploit parallelisation to make use of multi-core hardware architectures.

The strength of SMT solvers is that they offer fully automated push-button solutions. Thanks to efficient data structures and elaborate search heuristics, their increasing efficiency is coupled with increasing popularity and success in applications. An important enabling factor to applications was the introduction of a standard input language `SMT-LIB` [8] with a first release in 2004, which allows users to specify their problems in the standard language and to feed it to different solvers to find the optimal tool for a given purpose.

The standard also enabled the collection of reference benchmark sets and the start of annual competitions [7]. The first competition took place in 2005 with 12 participating solvers in 7 divisions (theories, theory combinations, or fragments thereof) on 1360 benchmarks, which increased in 2015 to 21 solvers competing in 40 divisions on 154238 benchmarks in the main track. All these activities contributed to the consolidation of an SMT solving community and to the visibility of the SMT-solving technologies. Nowadays, SMT solvers are widely used and are key components of many techniques in different academic and industrial areas.

In the following we give a short introduction to the theoretical foundations of satisfiability checking in Sect. 2, give a nutshell-overview about state-of-the-art SMT solvers including our own SMT solver `SMT-RAT` in Sect. 3, and discuss the efficient embedding of SMT solvers in different technologies in Sect. 4. We conclude the paper in Sect. 5. For further reading on SMT solving we refer to, *e.g.*, [9,46].

Quantifier-free theory	SMT-LIB name	Example theory constraints
equality and uninterpreted functions	QF_UF	$a = f(b, g(a, c))$
theory of (fixed-size) bit-vectors	QF_BV	$(\, a \mid b\,) \leq (\, a \& b\,)$
theory of arrays with extensionality	QF_AX	$\mathit{select}(\mathit{store}(a, i, v), i) = v$
floating point arithmetic	QF_FP	$x_2 = x_1 + 5(x_1 - y_1)$
real difference logic	QF_RDL	$x - y > 0$
integer difference logic	QF_IDL	
linear real arithmetic	QF_LRA	$3x + 7y - 8 \leq 0$
linear integer arithmetic	QF_LIA	
non-linear real arithmetic	QF_NRA	$x^{42} - 2yz^2 + 5 = 0$
non-linear integer arithmetic	QF_NIA	
theory of bit-vectors and bit-vector arrays extended with uninterpreted functions	QF_AUFBV	$\mathit{select}(a, \mathit{bv}[32]) < \mathit{bv}[32]$
linear real arithmetic with uninterpreted functions	QF_UFLRA	$3x + 7f(y) - 8 \geq 0$

Fig. 1. Example theory constraints from some logics that are included in the SMT-LIB standard language. The involved operators are: f, g, h are uninterpreted functions; $|$ and & are bit-wise *or* and *and*, respectively; finally, for arrays $\mathit{write}(a, i, v)$ is the array a after setting its ith field to v, whereas $\mathit{read}(a, j)$ stays for the jth field of a. For readability, the examples are not in SMT-LIB syntax, *e.g.*, they use infix notation.

2 Satisfiability Checking

Satisfiability checking aims at automated solutions for determining the satisfiability of existentially quantified first-order-logic formulas. Such formulas are Boolean combinations of *theory constraints*, where the form of the theory constraints depends on with which theory we instantiate first-order logic. For example, existentially quantified non-linear real arithmetic formulas can be built from polynomial equalities and inequalities, and their Boolean combinations. Some example theory constraints from different theories that are included in the SMT-LIB standard input language are depicted in Fig. 1. Exemplarily, we mention also two *combined* theories in the last two rows.

2.1 SAT Solving

Before we discuss SAT-modulo-theories solving for checking the satisfiability of quantifier-free first-order-logic formulas, we first make a short excursion to *SAT solving*. SAT solvers implement decision procedures to check the satisfiability of *propositional logic* formulas, being the Boolean combinations of atomic (Boolean) propositions.

Here we only explain the DPLL-style SAT solving algorithm, which is implemented in most state-of-the-art SAT solver technologies. The input formula

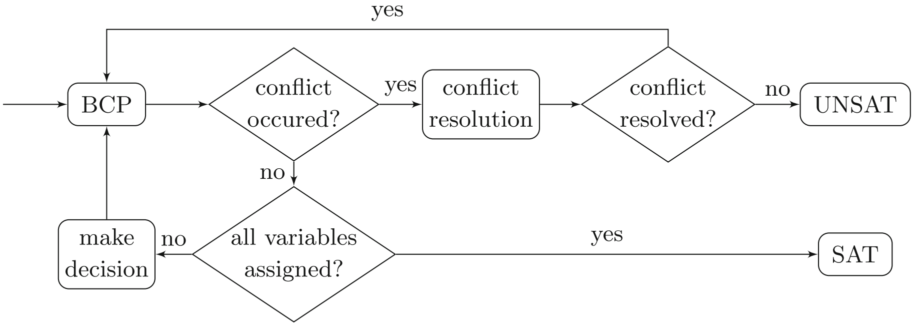

Fig. 2. The DPLL framework

is expected to be in conjunctive normal form (CNF), *i.e.*, the conjunction of *clauses*, each clause being the disjunction of *literals*, and each literal being a proposition or its negation. Each formula can be transformed into CNF in linear time and space at the cost of linearly many fresh propositions using Tseitin's transformation [67].

The DPLL algorithm has three main ingredients:

1. To explore the state space, the algorithm iteratively makes *decisions*, *i.e.*, it iteratively assigns truth values to some heuristically chosen propositions.
2. After each such decision, the algorithm applies *Boolean constraint propagation* (*BCP*) to determine further variable assignments that are implied by the last decision.
3. If BCP leads to a *conflict*, *i.e.*, if the value of a proposition is implied to be true as well as false at the same time, *conflict-driven clause-learning* and *non-chronological backtracking* [50] are applied: The algorithm follows back the chain of implications and applies *resolution* [30] to derive a reason for the conflict in form of a *conflict clause*, which is added to the solver's clause set. Backtracking removes previous decisions and their implications until the conflict clause can be satisfied.

If the input has clauses consisting of a single literal, these literals will be directly assigned. Therefore, the algorithm starts with BCP, as show in Fig. 2, to detect implications. If BCP leads to a conflict, the algorithm tries to resolve the conflict. If the conflict cannot be resolved, the input formula is unsatisfiable. Otherwise, if the conflict was successfully resolved, the algorithm backtracks and continues with BCP. If BCP could be completed without any conflicts, a new decision will be made if there are any unassigned propositions. Otherwise, a satisfying solution is found.

Example 1. Assume as input the CNF $(a) \wedge (\neg a \vee b) \wedge (c \vee d) \wedge (\neg b \vee c \vee \neg d)$. First a is set to `true`. BCP implies by the second clause that b must be `true` in order to complete the current partial assignment to a full satisfying solution. As no conflict appeared and there are still unassigned variables, a new decision will be made. Assume that this decision assigns `false` to c. BCP will assign `true` to d based on the third clause, however, now the fourth clause is conflicting. Resolution applied to the last two clauses will result in the conflict clause $(\neg b \vee c)$, which is added to the clause set. Backtracking removes the last decision, and BCP implies that c must be `true`. As all variables are assigned, a complete solution if found and the algorithm returns SAT.

The above algorithm is complete for propositional logic. It should be noted that many further optimisations were proposed, which led to major improvements, but cannot be discussed here.

2.2 SMT Solving

To check the satisfiability of quantifier-free first-order-logic formulas with an underlying theory (or combined theories [54]), *SAT-modulo-theories* (*SMT*) solvers can be applied. *Eager* SMT solving approaches translate the input formula to a satisfiability-equivalent propositional logic formula, whose satisfiability can be decided using a SAT solver. In the following we focus on *lazy* SMT-solving approaches.

Lazy SMT solvers combine a SAT solver with one or more *theory solvers*. Thereby the SAT solver handles the input formula's logical structure and is responsible for finding solutions for the *Boolean skeleton* of the input formula, which is gained by substituting fresh propositions for the theory atoms. To be able to check the consistency of theory atoms, the SAT solver communicates with the theory solvers, which implement decision procedures for the underlying theory.

Figure 3 illustrates the lazy SMT solving framework. The SAT solver iteratively searches for a satisfying solutions for the Boolean skeleton. During its search, it consults the theory solver(s) to check whether the current Boolean assignment is consistent in the theory. To do so, it collects all theory constraints whose abstraction proposition is `true` and appears non-negated in the formula, and those whose abstraction proposition is `false` and appears negated in the formula. The resulting theory constraint set is sent to the theory solver(s), which checks whether it is consistent. In the *full* lazy approach, this communication takes place only for full Boolean solutions, whereas in the *less* lazy approach usually after each conflict-free BCP execution.

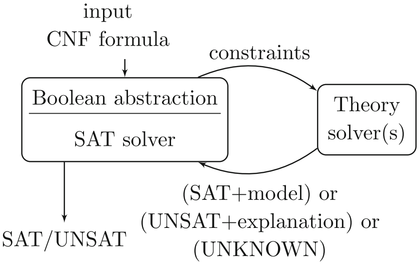

Fig. 3. The SMT solving framework

If the constraints are consistent in the theory and the SAT solver's assignment is already complete then a satisfying solution is found for the input formula. If the constraints are consistent but the Boolean assignment is not yet complete, the SAT solver continues its search. Otherwise, if the theory constraints are conflicting, the invoked theory solver returns an *explanation* for the conflict.

The explanation is often an *infeasible subset* $\{c_1, \ldots, c_n\}$ of the theory solver's input constraints, which leads to a tautology $(\neg c_1 \vee \ldots \neg c_n)$, whose abstraction can be added to the SAT solver's clause set. As the newly added clause is conflicting, conflict resolution is applied and the SAT solver continues its search in other parts of the search space.

Example 2. Assume as input the linear real-arithmetic formula

$$(x - y > 10) \wedge (x + y = 4 \vee x = 2y \vee x < y)$$

with Boolean abstraction

$$(a) \wedge (b \vee c \vee d) \ .$$

Assume that the SAT solver's current assignment is $a = \texttt{true}$, $b = \texttt{false}$, $c = \texttt{true}$ and $d = \texttt{true}$. The constraint set $\{x - y > 10, x = 2y, x < y\}$ is sent to a theory solver, which reports back inconsistency. A possible explanation is $\{x - y > 10, x < y\}$, whose abstraction $(\neg a \vee \neg d)$ assures that in the further search either a or d will be set to `false`.

The above-described approach clearly separates the Boolean search and theory solving. There are also other approaches in which Boolean and theory solving are more closely integrated.

First SMT solvers addressed more light-weight theories like equality logic and uninterpreted functions. Aiming at program verification, theories for arrays, bitvectors and floating-point arithmetic followed. Nowadays there are also highly tuned SMT solvers for linear arithmetic theories. Latest developments also allow solving non-linear arithmetic problems [26,41], quantified formulas, optimisation problems [13], and exploit parallelisation [70].

3 SMT Solvers

The aforementioned SMT competitions [7] compare the abilities of participating SMT solvers on `SMT-LIB` benchmark sets. The latest results from 2015 [64] give a good overview of state-of-the-art solvers and their range of applicability. Table 1 shows a rough survey of these solvers for existentially quantified logics. There is a large number of further SMT solvers, which did not participate in last year's competition. Other SMT solvers under active development, which we are aware of, are `Alt-Ergo` [25] and `iSAT3` [34,62]. Further examples for SMT solvers are `Ario`, `Barcelogic`, `Beaver`, `clasp`, `DPT`, `Fx7`, `haRVey`, `ICS`, `LPSAT`, `MiniSmt`, `Mistral`, `OpenCog`, `RDL`, `SatEEn`, `Simplics`, `Simplify`, `SMCHR`, `SONOLAR`, `Spear`, `STeP`, `SVC`, `SWORD`, and `UCLID`.

SMT-solver technologies cover a wide range of theories and their combinations. The embedding of theory decision procedures into the SMT solving context requires not only a deep understanding of the individual decision procedures, but also a careful software design. We illustrate how an SMT solver can be designed to support a broad range of logics, and how a user of such a solver can exploit the

Table 1. An overview of the SMT solvers for solving quantifier-free logical formulas that participated in SMT-COMP 2015 (for the naming of the logics see Fig. 1 and the SMT-LIB page [8]).

Solver	Website	Supported SMT-LIB logics QF_XXX
AProVE [37]	aprove.informatik.rwth-aachen.de	NIA
Boolector [55]	fmv.jku.at/boolector	ABV, AUFBV, BV, UFBV
CVC4 [6]	cvc4.cs.nyu.edu	All not involving FP
MathSAT5 [22]	mathsat.fbk.eu	All not involving integers
OpenSMT2 [18]	verify.inf.usi.ch/opensmt2	UF
raSAT [43]	github.com/tungvx/raSAT	NIA, NRA
SMTInterpol [21]	github.com/ultimate-pa/smtinterpol	All not involving BV, FP, NRA and NIA
SMT-RAT [26]	github.com/smtrat/smtrat/wiki	BV, LIA, LIRA, LRA, NIA, NIRA, NRA, UF
STP [35]	stp.github.io	BV
veriT [16]	www.verit-solver.org	All not involving BV, FP, NRA and NIA
Yices2 [32]	yices.csl.sri.com	All not involving FP and NIA
Z3 [51]	z3.codeplex.com	All

versatility, on the example of our SMT-RAT [26] solver. SMT-RAT's focus is on non-linear arithmetic. It adapts algebraic decision procedures to the needs of SMT solving and exploits powerful combinations of these procedures. Currently, it offers SMT-compliant implementations of the Fourier-Motzkin variable elimination, the simplex method [28], interval constraint propagation [36,39], methods based on Gröbner bases [68], the virtual substitution method [69], the cylindrical algebraic decomposition method [24], and a generalised branch-and-bound method. Additionally it provides a DPLL-style SAT solver as well as several preprocessing modules.

In SMT-RAT, all these procedures – including the SAT solver and preprocessing modules – are implemented in encapsulated modules, which share a common module interface. This modularisation allows for a strategic combination [52] of these solver modules: whenever a module is unable to solve a specific problem, it can forward the problem – or sub-problems – to other modules that might be better suited for the given task.

The strategic combination of solver modules is governed by a user-defined SMT-RAT *strategy*. Basically, a strategy is a directed tree, whose nodes are solver module instances, and whose edges are labelled with conditions. These conditions are evaluated in the context of a formula; an example for such a condition could

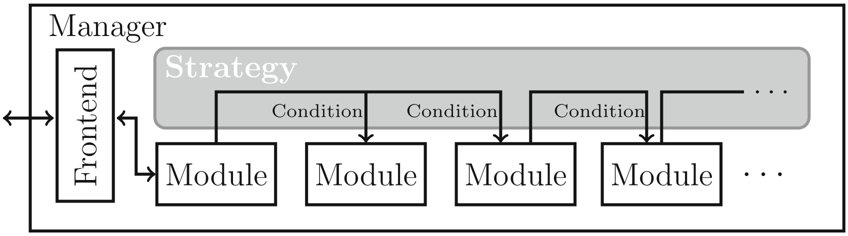

Fig. 4. Basic structure of an `SMT-RAT` strategy

be that the formula is linear, or that the maximal degree of polynomials in the formula is at most 2.

Figure 4 illustrates how such a strategy drives the solving procedure. A dedicated initial module (the root of the tree) receives the input formula and starts processing. If an executing module wants to pass on (sub-)problems to other modules, the conditions on the edges to its children are tested whether they hold for the given (sub-)problem. For each edge, if its condition holds, the child module will be invoked to solve the (sub-)problem. If a call to a child-module terminates, the calling module uses the returned result to continue its solving process. Note that this way also parallel execution can be implemented. Note furthermore that also the child modules can invoke further modules on their (sub-)problems.

In this framework, we can easily generate and test novel combinations of solving techniques, extend the range of supported logics, or employ parallel execution without the need to modify the previous implementation. Given a module that implements a certain decision procedure, it can be directly embedded within a strategy and thus participate in the overall solving process. Still, it does not save us the burden of handling the combination of two or more different theories. Theory combination schemes like Nelson-Oppen [54] are not yet implemented in `SMT-RAT` which is why it only supports a relatively small number of individual logics.

4 Applications

After the previous introduction to satisfiability checking and SMT solvers, let us turn to applications. In the following we mention some applications from the most popular areas. SMT solvers are employed in such a wide context that we cannot claim completeness, not only regarding single applications, but even regarding the application domains.

Program Verification. The perhaps most prominent SMT application example is program verification. In this area, the success of explicit model checking is complemented with symbolic and deductive approaches.

Bounded model checking [11] can be used to unroll the transition relation and to generate, for increasing path lengths, formulas that state the existence of a property-violating path. SMT solvers can be used to check the involved formulas for satisfiability, *i.e.*, to determine whether counterexamples exist. Whereas the basic approach cannot prove correctness but is rather suited to find counterexamples, it can be extended with, *e.g.*, k-induction to be able to prove the correctness of programs.

Deductive verification approaches generate verification conditions; if these conditions hold, the program is provably correct. In this context, SMT solvers can be used to check whether the verification conditions hold. Further methods related to, *e.g.*, invariant generation, interpolation and predicate abstraction can be invoked to increase the verification success.

Examples for tools in this area, which embed SMT-solving technologies, are `CBMC` [47] (bounded model checker for C and C++ programs), `IC3` [17,48] (induction-based verification approach), `PKIND` [42] (a parallel k-induction-based model checker), the Microsoft software model checkers `Boogie` [15] (intermediate-language verification) and `SLAM` [5] (device driver verification), the `Rodin` platform [31] for formal development in Event-B, and the SRI tool `SAL` [60] (infinite bounded model checker).

Symbolic Execution. Besides static analysis, SMT solvers are also used for *symbolic execution*. For example, the `Avalanche` tool [3] was developed to identify input data that reproduces critical bugs and vulnerabilities in programs. The tool is based on the `Valgrind` dynamic instrumentation framework. It analyses the target program by tracing and produces modified input data sets (corresponding to different execution paths) from the collected data. Finally, every possible execution path in the target program is traversed and checked for critical runtime defects. This way, buggy traces can be identified from a single test case.

Test-Case Generation. Due to the growing size of software, verification is not always applicable. Though the importance of thorough testing is undisputed among software engineers, crafting meaningful *test cases* remains a complex task. An ideal set of test cases should cover every possible code path and be reasonably concise and readable.

SMT-solving can be of help also in this area. The basic idea is similar to that of bounded model checking: we can encode paths with certain properties, *e.g.*, assuming that certain branches are followed or that certain loops are executed a given number of times, and use SMT solvers to find paths satisfying the given requirements. The work [19] reports on a successful application to generate test cases that cover most of the source code of the GNU Coreutils which "arguably are the single most heavily tested set of open-source programs in existence". The resulting code coverage was improved significantly and ten individual new bugs were found, therefrom three existing since at least 1992. Another approach for automated test case generation with SMT solving and abstract interpretation is proposed in [56].

Superoptimiser Compiler Backends. In the area of compiler construction, *superoptimisation* techniques assist to find optimal instruction sequences that are semantically equivalent to the original code fragment. The tool `Souper` [65] uses an SMT solver to automatically find *optimisations* missed by LLVM (low-level virtual machine) bit-code optimisers. Another work from this area is [57], where a simulator is used to evaluate the correctness of a candidate program on concrete test cases. If a candidate passes all test cases, the search technique verifies the *equivalence* of the candidate program and the reference program on all possible inputs using an SMT solver.

Termination Analysis. An important question in formal verification is whether a given program *terminates*. Though this question is undecidable in general, an active field of research has emerged on finding provable upper bounds on the runtime of a program. Usually, finding such complexity bounds requires solving non-linear integer problems. SAT and SMT techniques are routinely used by all leading termination analysis tools, for example `AProVE` [37], $\mathsf{T_TT_2}$ [45] or `NaTT`[71].

Program Synthesis. The paper [66] presents an SMT-based approach for component-based *program synthesis*. In this work, the synthesis problem is reduced to a satisfiability checking problem and an SMT solver is employed to synthesise bit-vector manipulation programs, padding-based encryption schemes, and block cipher modes of operations.

Planning. *Planning as satisfiability* was introduced in the area of artificial intelligence by Kautz and Selman in 1990 for domain-independent planning. This approach was limited to asynchronous discrete systems expressible in propositional logic, therefore SAT solvers could be applied. Later the approach was extended with numeric state variables and continuous time. Both of these extensions include integer- and real-valued state variables, which cannot be effectively handled by SAT solvers. As described in [59], SMT solvers can successfully solve such problems, but one needs to pay attention that the problem encoding is done carefully. To mention some further examples, SMT solvers in the area of planning were also applied for sequential numeric planning [61]. Another example is the work [38] which combines planning as satisfiability and SMT to perform efficient reasoning about actions that occupy realistic time. SMT solving for integrated task and motion planning is discussed in [53].

Scheduling. Many practical problems involve the *scheduling* of some tasks or processes. Oftentimes, their nature is not only combinatorial but also involves arithmetic constraints. For example, we might need to consider running times or certain resource demands in order to satisfy deadlines or to assure that enough memory is available for execution. SMT solvers, being designed to handle both combinatorial as well as arithmetic aspects, have been applied for numerous scheduling problems. The work [14] uses SMT solvers to solve resource-constrained project scheduling problems, where minimum as well as maximum delays between tasks are considered. Other examples are, *e.g.*, [2,27,58,72].

Cloud Applications. With the rise of cloud platforms, web applications have become much more flexible regarding scalability. However, *designing* a cloud application that consists of multiple components – for example database backends, webservers and a load balancer – poses the question, how many individual components are needed and how they shall be distributed among virtual machines. This problem has been solved in the `Zephyrus` tool [20] by the employment of constraint solving techniques. Based on an ongoing work of the authors with the tool developers we can state that SMT solvers equipped with linear optimisation are a valuable addition for such applications and can outperform previously used solutions. Implementing optimisation techniques for SMT solvers has seen a lot of progress in the last few years, for example in [13,49,63], thus we expect further interesting applications in domains that are so far dominated by constraint programming techniques.

Another work [12] is devoted the the analysis of *cloud contracts*, which capture architectural requirements in datacenters. The contracts are checked using the `SecGuru` tool, which is based on SMT solving and models network configurations in bit-vector arithmetic. `SecGuru` was also used to automatically validate network connectivity policies [40].

Hybrid Systems Reachability Analysis. *Hybrid systems* are systems with mixed discrete-continuous behaviour, typical examples being physical plants whose behaviour is controlled by a discrete controller. While the controller senses the plant state and executes control actions in a discrete manner, the dynamic state of the plant evolves continuously. For such systems, *reachability analysis* can be applied to assure that the plant never reaches any critical states.

One way to employ SMT solving technologies for reachability analysis is, similarly to programs, bounded model checking. However, as the dynamic behaviour is usually modelled using differential equations, the invoked SMT solvers need to be able to deal with the theory of differential equations. Suitable solvers for this task are `dReach` [23,44] and `iSAT-ODE` [33]. Beyond reachability analysis, the recent work [4] shows how various verification problems for complex synchronous hybrid PALS (physically asynchronous, logically synchronous) models can be reduced to SMT solving.

5 Conclusion

In this paper we gave a short introduction to SAT and SMT solving, discussed software design issues, and gave a number of SMT-solving applications.

The research area of satisfiability checking is highly active. New results and novel software engineering solutions constantly improve the power and the practical applicability of solver technologies. This holds not only for the efficiency, but also for the functionality of SMT solvers: Latest developments show that SMT solvers can also be successfully extended to handle, *e.g.*, quantified formulas or optimisation problems.

We expect this trend to be continued, on the one hand because there are still unused potentials (for example through building closer interactions with the symbolic computation community to improve on non-linear arithmetic theories [1]), and on the other hand because there is still a wide variety of problems, whose solutions could be improved by using SMT solving.

References

1. Ábrahám, E.: Building bridges between symbolic computation and satisfiability checking. In: Proceedings of ISSAC 2015, pp. 1–6. ACM (2015)
2. Ansótegui, C., Bofill, M., Palahı, M., Suy, J., Villaret, M.: Satisfiability modulo theories: An efficient approach for the resource-constrained project scheduling problem. In: Proceedings of SARA 2011, pp. 2–9. AAAI (2011)
3. Avalanche.: Dynamic program analysis tool. http://www.ispras.ru/en/technologies/avalanche_dynamic_program_analysis_tool/
4. Bae, K., Ölveczky, P.C., Kong, S., Gao, S., Clarke, E.M.: SMT-based analysis of virtually synchronous distributed hybrid systems. In: Proceedings of HSCC 2016 (2016). (to appear)
5. Ball, T., Bounimova, E., Levin, V., De Moura, L.: Efficient evaluation of pointer predicates with Z3 SMT solver in SLAM2. Technical report, Microsoft Research (2010)
6. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
7. Barrett, C.W., de Moura, L., Stump, A.: SMT-COMP: satisfiability modulo theories competition. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 20–23. Springer, Heidelberg (2005)
8. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016). www.SMT-LIB.org
9. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Chap. 26. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press, Amsterdam (2009)
10. Biere, A., Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
11. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
12. Bjørner, N., Jayaraman, K.: Checking cloud contracts in microsoft azure. In: Natarajan, R., Barua, G., Patra, M.R. (eds.) ICDCIT 2015. LNCS, vol. 8956, pp. 21–32. Springer, Heidelberg (2015)
13. Bjørner, N., Phan, A.-D., Fleckenstein, L.: νZ - an optimizing SMT solver. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 194–199. Springer, Heidelberg (2015)
14. Bofill, M., Coll, J., Suy, J., Villaret, M.: A system for generation and visualization of resource-constrained projects. In: Proceedings of CCIA 2014. Frontiers in Artificial Intelligence and Applications, vol. 269, pp. 237–246. IOS Press (2014)
15. Boogie.: An intermediate verification language. http://research.microsoft.com/en-us/projects/boogie/
16. Bouton, Thomas, de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: an open, trustable and efficient SMT-solver. In: Schmidt, Renate A. (ed.) CADE-22. LNCS, vol. 5663, pp. 151–156. Springer, Heidelberg (2009)
17. Bradley, A.R.: SAT-based model checking without unrolling. In: Schmidt, D., Jhala, R. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)

18. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
19. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of OSDI 2008, pp. 209–224. USENIX Association (2008)
20. Catan, M., et al.: Aeolus: mastering the complexity of cloud application deployment. In: Lau, K.-K., Lamersdorf, W., Pimentel, E. (eds.) ESOCC 2013. LNCS, vol. 8135, pp. 1–3. Springer, Heidelberg (2013)
21. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: an interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012)
22. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)
23. Cimatti, A., Mover, S., Tonetta, S.: A quantifier-free SMT encoding of non-linear hybrid automata. In: Proceedings of FMCAD 2012, pp. 187–195. IEEE (2012)
24. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Brakhage, H. (ed.) Automata Theory and Formal Languages. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975)
25. Conchon, S., Iguernelala, M., Mebsout, A.: A collaborative framework for non-linear integer arithmetic reasoning in Alt-Ergo. In: Proceedings of SYNASC 2013, pp. 161–168. IEEE (2013)
26. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: Heule, M., et al. (eds.) SAT 2015. LNCS, vol. 9340, pp. 360–368. Springer, Heidelberg (2015). doi:10.1007/978-3-319-24318-4_26
27. Craciunas, S.S., Oliver, R.S.: SMT-based task- and network-level static schedule generation for time-triggered networked systems. In: Proceedings of RTNS 2014, p. 45. ACM (2014)
28. Dantzig, G.B.: Linear Programming and Extensions. Princeton University Press, Princeton (1963)
29. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
30. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
31. Déharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: Integrating SMT solvers in Rodin. Sci. Comput. Program. **94**(P2), 130–143 (2014)
32. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Heidelberg (2014)
33. Eggers, A., Ramdani, N., Nedialkov, N.S., Fränzle, M.: Improving the SAT modulo ODE approach to hybrid systems analysis by combining different enclosure methods. Softw. Syst. Model. **14**(1), 121–148 (2012)
34. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. J. Satisf. Boolean Model. Comput. **1**(3–4), 209–236 (2007)
35. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)

36. 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: Proceedings of FMCAD 2010, pp. 81–90. IEEE (2010)
37. Giesl, J., et al.: Proving termination of programs automatically with AProVE. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 184–191. Springer, Heidelberg (2014)
38. Hallin, M.: SMT-Based Reasoning and Planning in TAL. Master's thesis, Linköping University (2010)
39. Herbort, S., Ratz, D.: Improving the efficiency of a nonlinear-system-solver using a componentwise Newton method. Technical report 2/1997, Inst. für Angewandte Mathematik, University of Karlsruhe (1997)
40. Jayaraman, K., Bjrner, N., Outhred, G., Kaufman, C.: Automated analysis and debugging of network connectivity policies. Technical report MSR-TR-2014-102, Microsoft Research (2014). http://research.microsoft.com/apps/pubs/default.aspx?id=225826
41. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 339–354. Springer, Heidelberg (2012)
42. Kahsai, T., Tinelli, C.: PKIND: A parallel k-induction based model checker. arXiv preprint (2011). arXiv:1111.0372
43. Khanh, T.V., Vu, X., Ogawa, M.: raSAT: SMT for polynomial inequality. In: Proceedings of SMT 2014, p. 67 (2014)
44. Kong, S., Gao, S., Chen, W., Clarke, E.: dReach :δ-reachability analysis for hybrid systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 200–205. Springer, Heidelberg (2015)
45. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean termination tool 2. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 295–304. Springer, Heidelberg (2009)
46. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, New York (2008)
47. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014)
48. Lange, T., Neuhäußer, M.R., Noll, T.: IC3 software model checking on control flow automata. In: Proceedings of FMCAD 2015, pp. 97–104. IEEE (2015)
49. Li, Y., Albarghouthi, A., Kincaid, Z., Gurfinkel, A., Chechik, M.: Symbolic optimization with SMT solvers. In: Proceedings of POPL 2014, pp. 607–618. ACM (2014)
50. Marques-silva, J.P., Sakallah, K.A.: Grasp: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**, 506–521 (1999)
51. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
52. de Moura, L., Passmore, G.O.: The strategy challenge in SMT solving. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics. LNCS, vol. 7788, pp. 15–44. Springer, Heidelberg (2013)
53. Nedunuri, S., Prabhu, S., Moll, M., Chaudhuri, S., Kavraki, L.E.: SMT-based synthesis of integrated task and motion plans from plan outlines. In: Proceedings of ICRA 2014, pp. 655–662. IEEE (2014)
54. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2), 245–257 (1979)

55. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisf. Boolean Model. Comput. **9**, 53–58 (2015)
56. Peleska, J., Vorobev, E., Lapschies, F.: Automated test case generation with SMT-solving and abstract interpretation. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 298–312. Springer, Heidelberg (2011)
57. Phothilimthana, P.M., Thakur, A., Bodik, R., Dhurjati, D.: GreenThumb: Superoptimizer construction framework. In: Proceedings of CCC 2016, pp. 261–262. ACM (2016)
58. Pike, L.: Modeling time-triggered protocols and verifying their real-time schedules. In: Proceedings of FMCAD 2007, pp. 231–238. IEEE (2007)
59. Rintanen, J.: Discretization of temporal models with application to planning with SMT. In: Proceedings of AAAI 2015, pp. 3349–3355. AAAI (2015)
60. Symbolic analysis laboratory. http://sal.csl.sri.com/introduction.shtml
61. Scala, E., Ramirez, M., Haslum, P., Thiebaux, S.: Numeric planning with disjunctive global constraints via SMT. In: Proceedings of ICASP 2016 (2016, to appear)
62. Scheibler, K., Kupferschmid, S., Becker, B.: Recent improvements in the SMT solver iSAT. In: Proceedings of MBMV 2013, pp. 231–241. Institut für Angewandte Mikroelektronik und Datentechnik, Fakultät für Informatik und Elektrotechnik, Universität Rostock (2013)
63. Sebastiani, R., Trentin, P.: OptiMathSAT: a tool for optimization modulo theories. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 447–454. Springer, Heidelberg (2015)
64. SMT-COMP 2015 result summary (2015). http://smtcomp.sourceforge.net/2015/results-summary.shtml
65. Souper. http://github.com/google/souper
66. Tiwari, A., Gascón, A., Dutertre, B.: Program synthesis using dual interpretation. In: Felty, A., Middeldorp, A. (eds.) CADE-25. Lecture Notes in Computer Science, vol. 9195, pp. 482–497. Springer, Heidelberg (2015)
67. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Siekmann, J.H., Wrightson, G. (eds.) Automation of Reasoning, pp. 466–483. Springer, New York (1983)
68. Weispfenning, V.: A new approach to quantifier elimination for real algebra. In: Caviness, B.F., Johnson, J.R. (eds.) Quantifier Elimination and Cylindrical Algebraic Decomposition. Texts and Monographs in Symbolic Computation, pp. 376–392. Springer, NEw York (1998)
69. Weispfenning, V.: Quantifier elimination for real algebra - the quadratic case and beyond. Appl. Algebra Eng. Commun. Comput. **8**(2), 85–101 (1997)
70. Wintersteiger, C.M., Hamadi, Y., de Moura, L.: A concurrent portfolio approach to SMT solving. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 715–720. Springer, Heidelberg (2009)
71. Yamada, A., Kusakari, K., Sakabe, T.: Nagoya termination tool. In: Dowek, G. (ed.) RTA-TLCA 2014. LNCS, vol. 8560, pp. 466–475. Springer, Heidelberg (2014)
72. Yuan, M., He, X., Gu, Z.: Hardware/software partitioning and static task scheduling on runtime reconfigurable FPGAs using an SMT solver. In: Proceedings of RTAS 2008, pp. 295–304. IEEE (2008)