Symbolic computation and satisfiability checking (Editorial)

James H. Davenport, Matthew England, Alberto Griggio, Thomas Sturm, Cesare Tinelli

Author's Version:
This paper has been published in the
Journal of Symbolic Computation Volume 100 (Elsevier).
The final authenticated version is available online at
https://doi.org/10.1016/j.jsc.2019.07.017

Symbolic Computation and Satisfiability Checking Editorial

James H. Davenport

Faculty of Science, University of Bath, UK

Matthew England

Faculty of Engineering, Environment and Computing, Coventry University, UK

Alberto Griggio

Embedded Systems Unit, Fondazione Bruno Kessler, Italy

Thomas Sturm

CNRS, Inria, and the University of Lorraine,France;
MPI Informatics and Saarland University, Germany

Cesare Tinelli

Department of Computer Science, University of Iowa, USA

Abstract

The two communities of Symbolic Computation and Satisfiability Checking have recently found themselves tackling similar problems and having a growing interest in each other's technology. This special issue presents articles whose contribution is of interest to, and is influenced by, both communities. Given the context of this journal we start this editorial with a more thorough overview of Satisfiability Checking, and then turn to Symbolic Computation and the potentials and challenges for collaboration. The collection of articles in this issue is evidence of the already existing fruitful work at the intersection of these communities.

Email addresses: J.H.Davenport@bath.ac.uk (James H. Davenport), Matthew.England@coventry.ac.uk (Matthew England), Griggio@fbk.eu (Alberto Griggio), thomas@thomas-sturm.de (Thomas Sturm), cesare-tinelli@uiowa.edu (Cesare Tinelli)

Keywords: satisfiability checking, symbolic computation, community integration

1. Two Communities

1.1. The SAT Problem

The Satisfiability Checking community is focussed on the problem of determining whether a given logical formula is satisfiable, that is, whether there is an interpretation of certain uninterpreted symbols that evaluates the formula to `true`. In the case where the uninterpreted symbols are all Boolean[1] variables this becomes the classic SAT Problem, the first problem to be proven NP-Complete (**?**).

Despite its theoretical complexity, today instances of the SAT problem involving hundreds of thousands of variables are solved routinely. This is usually done by dedicated *SAT-solvers*, many of which are available free and open source. SAT-solvers are now integrated in a wide variety of industrial domains, perhaps most notably formal verification. A textbook charting the rise of the SAT-solver is the Handbook of Satisfiability by **?**.

Modern SAT-solvers mostly rely on a combination of enumeration, propagation, and resolution. *Enumeration* refers to the natural solution of enumerating all possible variable assignments. In the case of an unsatisfiable formula with n variables this means checking 2^n assignments. Propagation and resolution help tackle this exponential search space: we introduce them next alongside the algorithms which first employed them.

1.1.1. The DPLL Algorithm

Propagation was the key feature of the DPLL algorithm by **?**. We first transform[2] the formula into *conjunctive normal form*: the formula becomes a conjunction of clauses, where each clause is a disjunction of literals (variables or their negation). We then proceed with an enumerative search but after each variable is set we seek implications for other variables. Such implications happen if the latest assignment renders a clause *unit* (all unassigned literals

[1] For our purposes the term Boolean algebra refers to propositional logic.

[2] Naively, this might be expensive, but the transformation of **?** (reprint of work from the 1960s) is in PTIME. The transformed formula is equi-satisfiable with the original (not neccessarily equivalent).

except one become `false`). Furthermore, if all literals in a clause are rendered `false` by an assignment, then no further analysis is required in this part of the search space. We instead backtrack to the last free decision (one made as part of the search rather than propagation).

Example 1. *Consider the formula*

$$(\alpha \vee \beta \vee \neg\gamma) \wedge (\neg\alpha \vee \neg\beta) \wedge (\beta \vee \gamma) \wedge (\neg\alpha \vee \beta \vee \neg\gamma), \tag{1}$$

which is already in conjunctive normal form with the parentheses indicating the four clauses. An initial assignment of α to `true` *would render the second clause unit and so propagation would set β to* `false`*. This in turn renders both the third and fourth clauses unit, however, propagating on one of them will give an assignment which renders the other* `false`*. So either way we backtrack to our choice and reassign α to* `false`*.*

This time there is no unit clause and so we have to make a choice for β. If we choose to assign β to `true` *then the formula is already satisfied regardless of the value for γ; while if we assigned β to* `false` *then propagation would lead to a conflict and backtracking to pick β to be* `true`*.*

1.1.2. The CDCL Algorithm

Propagation avoids exploring every branch of the search space but we may still end up replaying similar analyses in separate branches. The next milestone in the development of SAT-solvers was the idea of using resolution to avoid such repetition, as described by the conflict driven clause learning (CDCL) algorithm (**?**).

The main idea is to use *resolution* to learn the reason for a conflict and generate a new clause from that which we add to the input formula so that similar conflicts are avoided later in the search. This is done by means of a resolvent: given two clauses, one which has x as a literal and one with $\neg x$, the resolvent is the disjunction of all the other literals from both clauses. All assignments that satisfy both clauses must also satisfy the resolvent (since one of the other literals must have been satisfied).

Example 2. *Consider a formula (1) $\wedge \psi$ where (1) was the formula from Example 1 and ψ another formula in conjunctive normal form involving more variables than just α, β, γ. In the previous example we learnt that there was no satisfying solution to (1) with α assigned to* `true`*. We want to avoid repeating this analysis in different branches of the search space defined by*

the other variables in ψ. Resolution would ensure this: before the initial backtrack in Example 1 we saw that the third and fourth clauses conflicted – they wanted to propagate different assignments for γ. Thus we take the resolvent of these clauses to form $\neg\alpha \vee \beta$. This in turn causes a conflict with the second clause and their resolvent is simply $\neg\alpha$. Thus we add this to the input to ensure from then on that we only assign α to `false`.

The process of deriving resolvent clauses by analysing conflicts is the main idea of the CDCL algorithm, and together with propagation, it forms the backbone of modern SAT solvers. In fact, the CDCL algorithm can be seen as an interplay between two phases, *model search* and *proof search*, which drive each other in a feedback loop: model search tries to find a solution for the input formula, by applying decisions and propagation; when a conflict arises, the information about which propagations were performed is used by the proof search phase to learn new clauses (generated via resolution), which in turn will drive the model search phase to explore a different portion of the search space.

1.1.3. Modern SAT-solvers

We have presented above the two most important ideas to tackle the SAT problem, but there are many other innovations used in modern SAT-solvers. There is substantial engineering such as the *two watched literals* rule, heuristics for variable selection / branching (e.g. the VSIDS variable activity score), restart strategies, and techniques for clause database management (e.g. the LBD measure). Also of great importance are the preprocessing and inprocessing procedures, such as variable elimination, subsumption, and blocked clauses elimination. The best single reference for an overview of these topics is the Handbook of Satisfiability **?**.

1.2. Satisfiability Modulo Theories

The success of modern SAT solvers led naturally to the idea of applying such techniques beyond propositional calculus. The approach has been best formalised by the paradigm of Satisfiability Modulo Theories (SMT), which leads to SMT-solvers. These take as input a logical formula whose atoms are statements in a theory domain (or perhaps several different domains).

The SMT-solver will usually first convert the formula to conjunctive normal form and build its *Boolean skeleton* by replacing each theory atom by a new Boolean variable. This skeleton is passed to a SAT-solver: if it is found

to be unsatisfiable then the conclusion can be applied to the original formula also. To conclude, in contrast, a satisfying solution, one must check at some point the validity of the solution from the SAT-solver in the theory. This is usually done by calling a *theory solver*: a procedure for deciding if the set of theory constraints implied by the interpretation of Boolean variables is consistent. If not consistent then the theory solver is expected to offer an explanation: a statement that is a valid in the theory stating that a, hopefully small, subset of the constraints cannot hold together. This forms a new clause for the Boolean skeleton, generalising the inconsistency just encountered, to ensure future candidate solutions are not invalid in a similar manner. This way we are running a loop of calls to the SAT-solver and theory solver until, ideally, either a satisfying solution is found or unsatisfiability is concluded. In practice, one also admits incomplete theories and theory solvers so that SMT solvers possibly return *unknown* as a final result.

Example 3. *Consider the formula*

$$(x > 0 \vee x = -1) \wedge (x^2 < 0 \vee x^2 > 9).$$

The Boolean skeleton is $(a \vee b) \wedge (c \vee d)$ *where new Boolean variables* a, b, c, d *represent the truth of the four sign conditions involving a theory variable* x*. A SAT-solver may propose the solution to this skeleton which sets* a *and* c *to* `true` *and* b *and* d *to* `false`*. A theory solver would determine that the set of constraints this produces is incompatible and produce the explanation that we cannot have both* a *and* c `true` *at once. Thus the clause* $(\neg a \vee \neg c)$ *is added to the input. Further analysis by the SAT-solver may then swap the Boolean assignment for* c *and* d*. This time the theory solver should conclude the resulting set of conditions is satisfiable in the theory (by any* $x > 3$*).*

The above description is of the DPLL(T) architecture for SMT, and the example implemented the *lazy* approach, where the theory solver is only consulted after a full Boolean assignment has been obtained. There can be less lazy versions, with the theory solver consulted after each assignment. The chapter in the Handbook of Satisfiability by **?** is a key reference for this SMT paradigm.

There are other approaches to generalising SAT-solving to theory domains. The *eager* approach, introduced in **?**, translates the original formula to an equisatisfiable Boolean formula in a single step, which is then given to a standard SAT solver. *Programmatic SAT* introduced by **?** calls user

specified code instead of a particular theory solver. The *model constructing satisfiability calculus* (MCSAT) introduced by **?** extends propagation and resolution into the theory domain: it allows guesses, not only for truth values of the theory constraints, but also for guesses for the actual theory variables, followed by propagation techniques to drive the search for other theory variables away from unsatisfiable parts of the search space.

The theory solvers discussed above implement decision procedures to decide whether a set of constraints is compatible in the given domain. For linear real arithmetic the simplex method could be used for example, while in linear integer arithmetic cutting planes or interval constraint propagation could be used. The textbook by **?** offers a good outline of the various procedures commonly employed by SMT. For some domains the procedures in question were developed within the field of Symbolic Computation and have their natural home in computer algebra systems.

1.3. Symbolic Computation

The use of computers to do exact mathematics dates back almost as far as computing itself with the first theses appearing in 1953. Early successes include algorithms to perform automatic integration which could even certify when something could not be integrated in closed form (**?**). New methods for polynomial factorisation accompanied the first general purpose algebraic software in the 1960s. One of the most impactful contributions to the field was Gröbner Bases, developed by **?**, which allowed for the efficient solution of many problems for polynomials over algebraically closed fields.

The scope of the field of Symbolic Computation was set out in the editorial of the first volume of this journal (**?**) and is summarised as "*the algorithmic solution of problems dealing with symbolic objects*". Although the aforementioned editorial explicitly included both computational algebra and computational logic over time the Symbolic Computation field has seemed to focus more on the algebraic. Indeed, the implementation tools are almost exclusively referred to as *computer algebra systems*.

Today, these vary from large commercial general purpose systems such as `Maple`, and `Mathematica` to free and open source alternatives such as `Maxima`, `Reduce`, `Sage` and `Singular`. There are also a wide variety of specialised systems dedicated to subfields including `Macaulay2` for algebraic geometry, `GAP` for group theory, `CoCoA` for commutative algebra, `Qepcad` for real quantifier elimination, and `PariGP` for number theory.

1.4. Difficulties with Computer Algebra as Theory Solver

The history of Symbolic Computation is longer than that of Satisfiability Checking. The first SAT solver following modern design principles did not appear until the 1990s and by the time that SMT was actively developed many of today's popular computer algebra systems were already well established. So it should seem natural that SMT-solvers consider using a computer algebra system as a theory solver. However, this did not happen, for a number of reasons beyond the lack of familiarity of the communities. Computer algebra systems tend to be large standalone systems aimed at interactive problem solving with a human, while SMT-solvers tend to be made up from small, flexible and highly efficient libraries. Hence it is not trivial for an SMT-solver to share data-structures with a computer algebra system as it does with its typical solvers. More fundamentally though, the actual algorithms employed by computer algebra systems lacked in functionality necessary for successful use in SMT. To be *SMT-compliant* a theory solver must satisfy a number of criteria (**??**):

- The solver should work incrementally, i.e. after determining the consistency of a constraint set there should be the option to add an additional constraint and recheck, reusing as much computation as possible.
- The solver should have the ability to support backtracking, i.e. after determining the consistency of a constraint set there should be the option to remove a constraint and recheck, again, reusing as much computation as possible.
- In the case of unsatisfiability, the solver is expected to offer an explanation as to why the constraints cannot be satisfied, e.g. a small subset which are on their own incompatible.

These requirements are not commonly met by the algorithms in computer algebra systems, where the intended use case is usually to perform one computation to solve a single problem. To make a Symbolic Computation algorithm SMT-compliant requires much work, certainly at the implementation level, e.g. the data-structures used, but more importantly also in the mathematical theory.

2. SC-Square

We use SC^2 (SC-Square) to refer to the collaboration between the two communities of Satisfiability Checking and Symbolic Computation.

2.1. EU Project

SC^2 originates from a European Union Horizon 2020 project[3] that ran from 2016 to 2018 with the aim of bridging the gap between these communities. The project consortium consisted of: University of Bath (UK), RWTH Aachen (Germany), Fondazione Bruno Kessler (Italy), the University of Genoa (Italy), Maplesoft Europe Ltd, the University of Lorraine (France), Coventry University (UK), the University of Oxford (UK), the University of Kassel (Germany), the Max Planck Institute for Informatics (Germany), and Johannes Kepler University of Linz (Austria). In addition, the project had over 50 associates from around the world.

The main project goals were around community building, in particular through the funding of workshops and summer schools. The 2017 SC^2 Summer School in Saarbrücken, Germany included 12 extended lectures on the foundations of the two fields, key algorithms, and applications. The latter were presented by speakers from industry describing the use of such technology to verify software and systems in industries as diverse as automotive and finance. The slides and videos of some talks from the Summer School are available online[4].

The existence and maintenance of a common specification language SMT-LIB (**?**), with its accompanying collections of benchmarks and annual competitions, is seen as an important accelerator on research in Satisfiability Checking. Another key output from the project was work to extend SMT-LIB to better support the areas of relevance to SC^2 (a natural choice since there was no computer algebra alternative). The upcoming SMT-LIB 3.0 is set to include a variety of features relevant to Symbolic Computation such as new functions, quantifier elimination, real algebraic numbers, and optimisation. There have been 10 separate contributions of benchmarks linked to the SC^2 project, accounting for a third of the recent library growth. This has included a more diverse range of applications such as those from the life sciences (**?**) and economics (**?**). Also, `Maple` now supports the SMT-LIB

[3] `http://www.sc-square.org/EU-CSA.html`

[4] `http://www.sc-square.org/CSA/school/lectures.html`

language and integration with SAT/SMT solvers (**?**) as does `Reduce` which further, has a fully featured CDCL solver implemented and won the NRA category of the 2017 SMT competition[5].

The project also funded a variety of proof of concept studies including on integrations: between computer algebra system Redlog and SMT-solver VeriT, between computer algebra systems CoCoA and Maple and SMT-solver SMT-RAT, and between computer algebra system CoCoA and SMT-solver MathSAT; as well as development on new incremental versions of symbolic algorithms.

2.2. Workshop Series

The SC^2 project initiated an annual International Workshop on Satisfiability Checking and Symbolic Computation:

- SC^2 2016: 24th September 2016 in Timisoara, Romania. As part of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2016). Proceedings edited by **?**.
- SC^2 2017: 29th July 2017 in Kaiserslautern, Germany. Adjacent to the International Symposium on Symbolic and Algebraic Computation (ISSAC 2017). Proceedings edited by **?**.
- SC^2 2018: 11th July 2018 in Oxford, UK. Part of the Federated Logic Conference (FLoC 2018). Proceedings edited by **?**.
- SC^2 2019: 10th July 2019 in Bern, Switzerland. Part of the SIAM Conference on Applied Algebraic Geometry. Proceedings to appear, edited by Abbott and Griggio.

Although the EU project finished in 2018 there is considerable effort being spent to continue the workshop series. A constitution has been approved which sets out requirements: such as the conference alternating between affiliation with conferences in each community; and the need for two chairs each year, one from each community.

Most articles in this special issue are based on preliminary work presented at an SC^2 workshop, or one of the other SC^2 special sessions that took place at: the 2016 Conference on Applications of Computer Algebra (ACA 2016);

[5] `http://smtcomp.sourceforge.net/2017/results-NRA.shtml`

the 2016 International Workshop on Computer Algebra in Scientific Computing (CASC 2016); the 2017 International Conference on Mathematical Aspects of Computer and Information Sciences (MACIS 2017); and the 2018 International Congress on Mathematical Software (ICMS 2018).

3. Articles of this Special Issue

The special issue comprises seven articles. A call was made following the 2017 SC^2 workshop, and submission was open to all work on the the intersection of the two communities (no requirement for affiliation with the EU project). Of the seven articles, four have authors funded by the project (although all had taken part in the project workshops and special sessions). An average of 3.3 reviews per article was obtained with all reviewed by at least one expert in each of the two communities. All accepted articles underwent a multi-stage revision process to take into account the high standards of this journal.

3.1. Non-linear real arithmetic

SMT-LIB organises the theory solver domains into logics. From these, the one which has probably found the greatest intersection with Symbolic Computation has been non-linear real arithmetic. Here the atoms of the logical formulas are sign-conditions on non-linear multivariate polynomials, whose variables take real values. Questions in this theory are decidable, a result uncovered by **?**.

The decision procedures most commonly used for non-linear real arithmetic include *(Partial) Cylindrical Algebraic Decomposition* (**??**) and *Virtual Substitution* (**??**), familiar to readers of this journal as key algorithms of Symbolic Computation, developed for quantifier elimination in real closed fields. Given a non-linear arithmetic formula as described above which is preceded with quantifications on some of the variables involved, *Quantifier Elimination* (QE) means to find an equivalent unquantified version. For example, QE would transform $\exists x, ax^2 + bx + c = 0 \wedge a \neq 0$ to the equivalent unquantified statement $b^2 - 4ac \geq 0$. Satisfiability Checking is hence a sub-problem of QE for the case where all variables are existentially quantified (so the answer is boolean rather than an expression in the unquantified variables). Although SMT-LIB is now expanding to encompass QE also.

Two articles of the present special issue concern Cylindrical Algebraic Decomposition (CAD). A CAD is a decomposition of ordered $\mathbb{R}^n$ space into

cells arranged cylindrically, meaning the projections of any pair of cells with respect to the variable ordering are either equal or disjoint. The projections form an induced CAD of the lower dimensional space. The cells are (semi)-algebraic meaning each can be described with a finite sequence of polynomial constraints. To study a logical formula for SMT or QE a CAD must be produced *truth-invariant* to the formula (so the formula is either `true` or `false` on each cell). We can then check satisfiability by testing a finite number of sample points, and in the case of QE form the equivalent formulae from the cell descriptions. CAD traditionally consists of two stages: first projection, which identifies polynomials which will form boundaries in the decomposition; and then lifting, where the decomposition is produced by repeated use of real root isolation on these polynomials rendered univariate by substitution at sample points of the CAD for the space below.

The article by Ábrahám and Kremer describes how they have adapted the original CAD algorithm to satisfy the requirements for SMT-compliance outlined in Section 1.4. Their algorithm builds a CAD incrementally, by performing one projection step at a time and then moving to lifting to perform the additional real root isolation and cell decomposition required by the new additions. They maintain a history of computation allowing for backtracking, developing data structures quite different to existing CAD implementations. Their work is implemented in their SMT-solver `SMT-RAT`.

The article by England, Bradford and Davenport considers how the process of CAD construction can be improved when the input contains equational constraints (equations which must be satisfied for the formula to hold). Intuitively each of these decrease the solution space by one dimension, and the authors show that a corresponding decrease in the theoretical complexity of CAD can be achieved. The work may act as a tutorial on CAD with equational constraints with all necessary background and the current state of the art.

The article by Brown and Vale-Enriquez presents work for a partial solver for non-linear real arithmetic, i.e. a solver that can sometimes decide the satisfiability, and if not can at least simplify the input formula to ease the use of something like CAD. The partial solver is based on existing simplification algorithms, which (like the CAD of Ábrahám and Kremer) have had to be heavily modified for SMT-compliance. These simplification algorithms have also been adapted to produce the desired explanations for unsatisfiability required for SMT-compliance. This work is implemented in the `Tarksi` system, developed by Brown to implement the non-uniform CAD (NuCAD)

algorithm developed by **??** which is an example of work in Symbolic Computation inspired by techniques from Satisfiability Checking (most notably those of **?**).

3.2. Linear Arithmetic

The article by Bromberger, Sturm and Weidenbach presents a new decision procedure for linear integer arithmetic. Although this theory is decidable, the authors explain that the leading solvers currently neglect termination on some classes of problems in favour of efficiency on all other problems. As an alternative, the authors present an extension to an existing calculus so it becomes complete and terminating but stays efficient. The procedure is based on a mixture of model-driven reasoning and quantifier elimination techniques.

The article by Abbott, Bigatti, Palezzato and Robbiano concerns the computation of minimal polynomials in P/I where P is a polynomial ring and I an ideal. Such non-linear objects can capture fundamental information about linear algebra. The work can also improve the computation of Gröbner Bases, which although a technique for algebraically closed fields such as $\mathbb{C}$ is regularly used as an initial unsatisfiability check for real arithmetic, and is well known as a useful pre-processing for CAD (**?**). Their algorithms can also assist with polynomial factorization, useful for many procedures (notably CAD, where experience shows that it improves performance even when not logically necessary). The authors implement their work in the computer algebra system `CoCoA` but also `CoCoALib`: a free library of C++ code that can be particularly easily integrated with other C-like systems.

3.3. Symbolic Computation for Better SAT-solvers

The article by Horáček and Kreuzer is the only one in the issue to focus on the Boolean domain, as in Section 1.1. The idea is to use the equivalence between the Boolean SAT-problem and polynomial system solving over a finite field. The former may be attacked with SAT-solvers as described above while the latter by techniques such as Gröbner Bases, which originate from Symbolic Computation. The algebraic solver can be called to support the SAT-solver at selected points, or be in run in parallel, periodically exchanging information. The article in this issue is focussed on the method to convert a formula from the conjunctive normal form (CNF) used by the SAT-solver to the algebraic normal form (ANF) required for the algebraic techniques.

They develop a blockwise algorithm to producing fewer and lower degree polynomials.

3.4. SC² Combinations for Combinatorics

The article by Bright, Kotsireas and Ganesh investigates the Williamson Conjecture, an open problem in combinatorics on the existence of certain matrices, useful for error correcting codes. Through a combination of SAT-solver and computer algebra system the authors are able to enumerate all cases up to a much larger dimension than achieved before, uncover new cases, and gain new insight into the distribution of these matrices. Rather than the standard SMT approach outlined in Section 1.2 they use programmatic SAT: here rather than a full CNF encoding instances can be a set of CNF clauses and user code that encodes constraints too cumbersome to be written in CNF. The authors have applied these techniques to a variety of other combinatorial problems: an intersection of the two communities not predicted by the EU grant but documented by the workshop series.

Acknowledgements

Much of the editorial work for this special issue, as well as the work for many of the articles, was supported by EU H2020-FETOPEN-2016-2017-CSA project SC² (712689). The editors thank all the external reviewers for their detailed and thoughtful comment on the articles in this special issue.

References