A System for CSP Solving through Satisfiability Modulo Theories[1]

Miquel Bofill[2] Miquel Palahí[3] and Mateu Villaret[4]

Departament d'Informàtica i Matemàtica Aplicada
Universitat de Girona
E-17071 Girona, Spain

Abstract

In this paper we discuss work in progress on the design and implementation of `Simply`, a system for modeling and solving Constraint Satisfaction Problems (CSP). During the last years, there has been a dramatic improvement on performance of SAT solvers, and solving CSPs by translation into propositional formulas has become a real choice in many cases. The advances in SAT technology have been adapted for more expressive (yet decidable) logics, e.g., in the framework of SAT Modulo Theories (SMT). `Simply` is intended to be a declarative programming system for easy CSP modeling which generates SMT formulas for solving these CSP. The utility and interest of `Simply` is twofold: on the one hand, it serves as a benchmark generator for CSP solving with different state-of-the-art SMT solvers and, on the other hand, the system aims at taking advantage of the highly increasing performance of these solvers.

Keywords: Constraint Satisfaction Problems, Propositional Satisfiability, Satisfiability Modulo Theories

1 Introduction

Over the last decade there have been important advances in logic based techniques and tools. Advances have been especially significant in the field of propositional satisfiability (SAT), to the point that nowadays modern SAT solvers can tackle real-world problem instances with millions of variables. Hence, SAT solvers have become a viable engine for solving combinatorial discrete problems. For instance, in [4], an application that compiles specifications written in a declarative modeling language into SAT instances is shown to give promising results. See also [9,19] for some applications of SAT technology on industrial problems. Interesting comparisons between SAT and Constraint Satisfaction Problem (CSP) encodings and techniques can be found in [18,8].

[1] Partially supported by the Spanish Ministry of Science and Innovation through the project SuRoS (ref. TIN2008-04547/TIN)

[2] Email: mbofill@ima.udg.edu

[3] Email: u1038503@correu.udg.edu

[4] Email: villaret@ima.udg.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

SAT techniques have been adapted for more expressive logics. For instance, in the case of *SAT Modulo Theories (SMT)*, the problem is to decide the satisfiability of a formula with respect to a decidable background theory, such as the theory of linear (integer or real) arithmetic, arrays, lists, etc., or combinations of them, in first order logic with equality [16,14,7,2,1]. Input formulas are often syntactically restricted, for example, to be quantifier-free, so that the problem is still decidable. Hence, an SMT instance is a generalization of a Boolean SAT instance in which some propositional variables have been replaced by predicates from the underlying theories, and can contain formulas like, e.g.,

$$f(f(x) - f(y)) \neq f(z) \wedge x + z \leq y \wedge y \leq x \Rightarrow z < 0$$

providing a much richer modeling language than plain propositional formulas. Adaptations of SAT techniques to the SMT framework have been described in [17].

The main application area of SMT is hardware and software verification. However, the available theories do not restrict the usage of SMT to verification problems and, in fact, they allow to encode many problems outside the verification area in a very natural way. There are already promising results in the direction of adapting SMT techniques for solving Constraint Satisfaction Problems, even in the case of combinatorial optimization (see, e.g., [12] for an application of an SMT solver on an optimization problem, being competitive with the best weighted CSP solver with its best heuristic on that problem). Fundamental challenges on SMT for Constraint Programming (CP) and Optimization have been detailed in [13].

Since the beginning of CSP solving, its *holy grail* has been to obtain a declarative language that allows users to easily specify their problem and forget about the techniques required to solve it. There are a lot of successful systems in this direction, just to comment on two of them: MiniZinc [11] proposes to be a "standard CSP modeling language" that can be translated into a kind of intermediate code called FlatZinc for which several solvers provide specialized front-ends; Essence [6] allows the user to specify combinatorial problems in a mixture of natural language and discrete mathematics manner.

In this paper we discuss work in progress on the design and implementation of `Simply`, a new system for modeling and solving CSPs which is aimed at taking advantage of the increasing efficiency of SMT solvers. Its characteristics are the following. On the one hand, the input language of `Simply` is similar to that of EaCL [10], and its main implemented features are arrays, FOR sentences, comprehension lists and some global constraints. Although currently the declarativeness of this language does not reach the level of Essence or MiniZinc, its simplicity makes it really practical. On the other hand, `Simply` works in the spirit of Spec2Sat [5], which transforms problem specifications written in NPSpec [3] into SAT instances in DIMACS format. However, as said, the input language of `Simply` is similar to that of EaCL and, most importantly, it generates SMT instances according to the standard SMT-LIB language [15] instead of plain SAT instances. Then, the problem can be solved by using any state-of-the-art SMT solver.

Thanks to the higher level of expressivity of SMT, the resulting SMT instances are smaller than if they where just plain SAT and, in many cases, as we show in Section 4, the problems can be solved in a reasonable amount of time by state-of-the-art SMT solvers. Our aim is to take advantage of the race for efficiency between

SMT solvers, and to expand the application of SMT techniques to new areas. Also, our system can serve as a benchmark generator for CSP solving through SMT.

`Simply` is being developed in HASKELL. At the current stage of development, it can only generate SMT formulas within the *quantifier-free Linear Integer Arithmetic* logic, but there are plans to extend the language in order to deal with other logics and theories and like, e.g., arrays and bit vectors. Linux and Windows binaries of `Simply`, as well as documentation and some benchmarks, can be found in: `http://ima.udg.edu/~villaret/simply`.

The rest of the paper is structured as follows. In Section 2 we recall some basic concepts on SMT. In Section 3 we introduce the tool and its language. In Section 4 we discuss on some benchmarks. Finally, in Section 5 we conclude and discuss on further work.

2 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first-order formula with respect to some background first-order theory. That is, an SMT instance is a first-order formula where some function and predicate symbols have predefined interpretations, according to the background theories. Examples of theories are *Equality and Uninterpreted Functions*, *Linear Integer Arithmetic*, *Linear Real Arithmetic*, their fragments *Integer Difference Logic* and *Real Difference Logic*, *Arrays* (useful in modeling and verifying software programs), *Bit-Vectors* (useful in modeling and verifying hardware designs), or combinations of them (see [15] for details). Most SMT-solvers are restricted to decidable quantifier-free fragments of their logics, but this suffices for many applications. Usually, SMT-solvers deal with problems with thousands of clauses like, e.g., $x+3 < y \vee y = f(f(x+2)) \vee g(y) \leq 1$, containing atoms over combined theories, and involving functions with no predefined interpretation, i.e., uninterpreted functions.

There are two main approaches to solve SMT instances, namely, the *eager* and the *lazy* approach. In the eager approach, the formula is translated into an equisatisfiable propositional formula. This allows to use off-the-shelf SAT solvers, but has important drawbacks like, e.g., exponential memory blow-ups. For this reason most, if not all, state-of-the-art SMT solvers implement a lazy approach, which does not involve a translation into SAT. One of these approaches is DPLL(T) [14], which consists of a general DPLL(X) engine, very similar in nature to a SAT solver, whose parameter X is instantiated with a specialized solver $Solver_T$ for a given theory T, producing a DPLL(T) system. The basic idea is making the DPLL(X) engine and $Solver_T$ work in cooperation: while the DPLL(X) engine is in charge of enumerating (partial) propositional models, $Solver_T$ is responsible for checking whether these models are consistent with the given theory T (for example, if T is the theory of *Linear Integer Arithmetic* and the current Boolean model contains $x + 2y \leq 0$, $-y - z \leq 0$ and $x - 2z > 1$, then $Solver_T$ has to detect that the current Boolean assignment is T-inconsistent). Note that $Solver_T$ only needs to handle conjunctions of predicates from theory T. In this way, given a formula F and a theory T, we are determining whether there is a model of $T \cup \{F\}$.

Current SMT solvers can deal with several theories and logics. For the purposes

of our tool, there are two logics which are especially relevant, namely, *unquantified Integer Linear Arithmetic* (QF_LIA) and its fragment *Integer Difference Logic* (QF_IDL). QF_LIA formulas are Boolean combinations of inequations between linear polynomials over integer variables. QF_IDL formulas are Boolean combinations of inequations of the form $x - y < b$ where x and y are integer variables and b is an integer constant (see [15] for details). Such kind of atoms occur in the definition of feasible solutions for many problems. For instance, they can be used to express constraints on the time elapsed between pairs of events. Since the satisfiability of conjunctions of such literals can be reduced to the absence of negative cycles in finite weighted graphs, it can be decided in $O(n^3)$ time by the Bellman-Ford algorithm. For this reason, many solvers give a special treatment to such kind of literals.

3 `Simply`: The Tool

3.1 The Input Language of `Simply`

One of the goals of our tool is simplicity, since we are interested in modeling CSPs easily. For this reason, we have chosen the input language of `Simply` to be similar to that of EaCL [10]. A CSP specification in our language has four parts:

(i) `Data` definition. This is where *constants* that will be used in the rest of the specification are defined. These can be either integer values, evaluable expressions (hence, using other values or previously defined constants) or lists of integer values.

(ii) `Domains` definition. A *domain* characterizes the set of possible values of a variable. It can be defined either as a range between two integer values, a list of integer values or a list of ranges and values.

(iii) `Variables` declaration. A *variable* can denote either an integer or a multidimensional array of integers. Integer variables are in fact *finite domain variables* and, hence, they must be constrained to some previously defined domain.

(iv) `Constraints` posting. This is where the problem is specified, by posting the *set of constraints* that defines a feasible solution of the problem.

3.1.1 The Constraints

Our input language can deal with *basic* and *global* constraints.

- Basic constraints are either integer or logical constraints. The following operators are supported: `=`, `<>`, `<`, `=<` and `>=` for integers and `Not`, `And`, `Or`, `Xor`, `Implies`, `Iff` and `If_then_else` for Booleans.
- Currently the following global constraints are supported:
  - `Sum(List,Value)`. This constraint enforces equality between `Value` and the sum of all elements of `List`. Both `Value` and the elements of `List` are allowed to be integer variables.
  - `Count(List,Value,N)`. This constraint states equality between `N` and the number of occurrences of `Value` in `List`. Here again `N`, `Value` and the elements of `List` can be integer variables.
  - `AllDifferent(List)` requires all the elements of `List` to be different.

The lists appearing in *global* constraints can be extensional, e.g., `[1,x,3,m[a]+3]`, or intensional via *comprehension* lists *à la Haskell*. This powerful and expressive feature allows us to generate complicated lists easily. We illustrate its usage with the following example: `[ m[i,j] | i in 1..3, j in 1..3, i<>j ]` results into `[ m[1,2], m[1,3], m[2,1], m[2,3], m[3,1], m[3,2] ]`. The first part of a comprehension list is the *pattern*, i.e., the expression we want to generate. Currently patterns can only be constants or integer variables (in this example, the elements of the bi-dimensional array `m`). The rest of the comprehension list is formed by two distinct kinds of expressions, namely, the *generators* (in the example `i in 1..3` and `j in 1..3`, that expand the pattern), followed by the *filters*, that restrict these expansions (like `i<>j`).

Constraints can be posted directly, or with the `If-Then-Else` and `Forall` statements. These two statements are processed at compilation time. For instance, when the compiler finds the `If-Then-Else` statement, it evaluates the `If` condition. If it is true, the constraints of the `Then` branch are posted and, otherwise, the constraints of the `Else` branch are posted [5]. It is important to notice the difference between the `If-Then-Else` statement and the `If_then_else` operator, which is not evaluated at compilation time. Consider, for instance, the following example: `If (i<4) Then { m[i]<>m[i+1]; } Else { m[i]<>m[i-1]; m[i]=m[i-2]; }`. The condition `i<4` is evaluated at compilation time. Therefore, the variable `i` cannot be a "constraint" variable, i.e., it must be a constant or a "local" variable, e.g., an index of a `Forall` statement.

The compilation of a `Forall` statement can be illustrated with the following example: `Forall(i in 2..4) { m[i]<>m[i-1]; }` results into the replication of the constraint `m[i]<>m[i-1]` with the local variable `i` being replaced by the appropriate values: `{ m[2]<>m[1]; m[3]<>m[2]; m[4]<>m[3]; }`. The compilation techniques used for the expansion are quite similar to the ones used in the comprehension lists.

3.2 Architecture of the Tool

We describe the architecture of `Simply` depicted in Figure 1 throughout the process of modeling and solving a CSP. Let the input of our compiler be the text file (`csp.y`) [6]. In the compilation process all constants are replaced by their associated value, and variables are translated into SMT integer variables. The restriction of the variables to their domains results into equality and inequality predicates, for the case of single values and the case of ranges of values, respectively. The translation of the constraints typically results into a conjunction of QF_LIA predicates (see Section 2) and, in some occasions, into a conjunction of QF_IDL predicates. In the end, the compilation process produces a file `csp.smt` in the standard SMT-LIB format. The generated SMT problem instance can be solved by any of the SMT solvers supporting the QF_LIA logic. Solving the `csp.smt` with the desired SMT solver will result into a *sat* or *unsat* answer (notice that those solvers are complete). In addition, some of them can return a model (in particular, the values of the SMT

[5] The `Else` branch, contrarily to the case of the `If_then_else` operator, is optional.

[6] The parser for the language has been built using the `Happy` parser generator. This is why we use the `.y` extension.

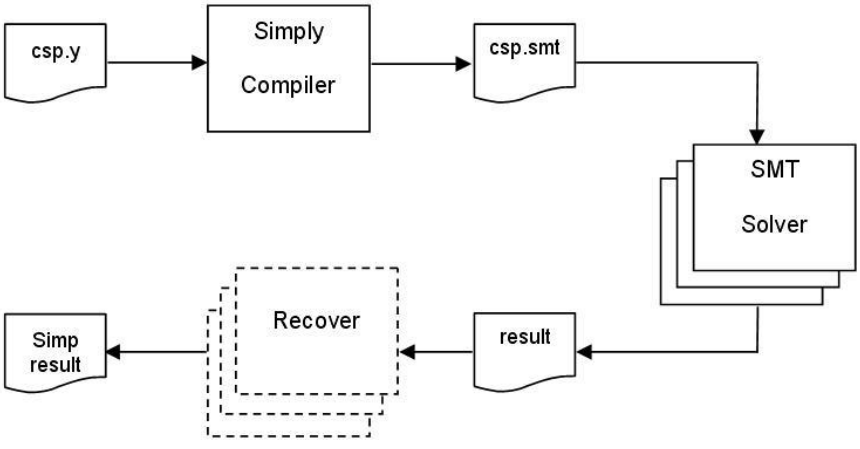

Fig. 1. The architecture of `Simply`.

variables) when the problem is satisfiable. In general the names of the variables are easy to interpret. Nevertheless, we have left as future work the recovering process from SMT solutions to values of the original variables in the `csp.y` file.

4 Complete Examples and Benchmarks

In this section we provide a complete encoding of the *Golomb's Ruler* problem and a complete encoding for a *sudoku* puzzle. We also comment on a table of benchmarks for the following problems:

- *Golomb's ruler*: A Golomb ruler of length l with n marks is defined as a set of n integers $0 \leq a_1 < a_2 < ... < a_n \leq l$ such that the $m(m-1)/2$ differences $a_j - a_i, 1 \leq i < j \leq n$ are all distinct. In our formulation, given l and n, we are interested in finding a Golomb ruler of any length not greater than l. In the benchmark table of Figure 4, the problem `golomb_ruler_n_l` means solving the problem for a ruler with n marks and length at most l.
- *Social Golfer*: Given a set of p players that want to play golf once a week, it amounts to find an arrangement for all of them into a number g of groups of size s for w weeks, in such a way that no two players play more than once in the same group. The following relationship must hold among three of the aforementioned quantities: $p = g * s$. In the benchmark table of Figure 4, the problem `social_golfer_w_g_s` means solving the problem for w weeks, g groups of size s.
- *Queens*: Given a number q of queens of the chess game, assuming that we place one queen per column in a board of $q \times q$ size, we have to find the row for each queen such that no queen threats any other. In the benchmark table of Figure 4, the problem `queens_q` means solving the problem for q queens.
- *Sudoku*: The classic puzzle.

The encoding of the Golomb's ruler problem is given in Figure 2. In `Data` we define the two constants for each instance of the problem: the length of the ruler `l` and the number of marks `n`. In `Variables` we declare the array `m` of `n` integer

variables with domain `a`, defined in `Domains` (the values from 0 to `l`). Then we post the required constraints: we firstly enforce marks to be all different, then we break some symmetries by ordering the marks and finally, we enforce all distances between any pair of marks to be different.

```
Problem: GolombRuler
{ Data { l:=55; n:=10; }
  Domains { Dom a=[0..l]; }
  Variables { IntVar m[n]::a; }
  Constraints {
        Forall(i in [1..n-1]) { m[i]<m[i+1]; }
        Forall(i in [1..n]) {
            Forall(j in [i+1..n]) {
                Forall(k in [1..n]) {
                    Forall(h in [k+1..n]) {
                        If ((i<>k) Or (j<>h)) Then
                           { m[i]-m[j]<>m[k]-m[h]; }
                    }
                }
            }
        }
    }
}
```

Fig. 2. A naive encoding of the *Golomb's ruler* problem with $l = 55$ and $n = 10$.

The encoding of the benchmarked *sudoku* is given in Figure 3. The relevant code is the last `Alldifferent` where we enforce the 9 internal squares to have the 9 different values using a comprehension list.

We have run the SMT solvers which participated in the QF_LIA division of the Satisfiability Modulo Theories Competition [7] (SMT-COMP) 2008 against some benchmarks generated with `Simply` from the previously defined problems. We want to remark that all benchmarks correspond to naive formulations of these classic CSPs. Figure 4 shows the time in seconds spent by each solver in each problem. The timeout was set to 600 seconds. The benchmarks were executed on a 3.00 GHz Intel Core 2 Duo machine with 2 Gb of RAM running under GNU/Linux 2.6. We have not included the compilation time of `Simply` code into SMT-LIB format in the table of Figure 4, because it has always been a few milliseconds.

5 Conclusion and Further Work

We have presented work in progress on a tool for easy CSP modeling and solving, whose main novelty is the generation of SMT problem instances as output. Our aim is to take advantage from the improvements that take place from year to year in SMT solvers, and also to provide a tool for benchmark generation.

[7] SMT-COMP: The Satisfiability Modulo Theories Competition (`http://www.smtcomp.org`).

```
Problem: Sudoku
{ Data { n:=9; }
  Domains { Dom a=[1..n]; }
  Variables { IntVar s[n,n]::a; }
  Constraints {
    s[1,3]=2; s[1,6]=1; s[1,8]=6;
    s[2,3]=7; s[2,6]=4; s[3,1]=5;
    s[3,7]=9; s[4,2]=1; s[4,4]=3;
    s[5,1]=8; s[5,5]=5; s[5,9]=4;
    s[6,6]=6; s[6,8]=2; s[7,3]=6;
    s[7,9]=7; s[8,4]=8; s[8,7]=3;
    s[9,2]=4; s[9,4]=9; s[9,7]=2;
    Forall(i in [1..n])
    {  AllDifferent([ s[i,j] | j in 1..n]);
       AllDifferent([ s[j,i] | j in 1..n]); }
    Forall(i in [0..2])
    {  Forall(j in [0..2])
       {  AllDifferent( [ s[f,c] | f in (1+i*3)..(3+i*3),
                                   c in (1+j*3)..(3+j*3) ] ); }
    }
  }
}
```

Fig. 3. A naive encoding of a *Sudoku* puzzle.

	Z3.2	MathSAT-4.2	CVC3-1.5	Barcelogic 1.3	Yices 1.0.10
golomb_ruler_7_25	0.18	0.66	out of memory	0.30	0.21
golomb_ruler_8_34	8.34	5.45	out of memory	43.32	0.64
golomb_ruler_9_44	147.35	13.70	out of memory	timeout	586.96
golomb_ruler_10_55	timeout	timeout	out of memory	timeout	timeout
social_golfer_5_4_4	0.12	2.52	timeout	9.48	0.89
social_golfer_6_5_5	13.86	49.83	timeout	timeout	4.47
social_golfer_9_8_4	timeout	timeout	timeout	timeout	timeout
queens_8	0.00	0.09	25.12	0.11	0.01
queens_16	0.18	7.16	timeout	8.16	3.04
queens_24	6.67	197.63	timeout	298.50	23.54
sudoku	0.00	0.68	105.61	1.70	0.41

Fig. 4. Time in seconds spent on some classic problems (naive version).

Much work is still to be done in the development of `Simply` to make it competitive with other tools for CSP solving. We distinguish among three aspects:

5.1 Features of the Tool

Probably the most important aspect of future work is to study the way of obtaining better SMT encodings from our source language. This can be done either by obtaining less naive translations of constraints, especially for global ones, and by introducing new theories and logics. For instance, (uni-dimensional) arrays of integers of `Simply` programs can be flattened into integer variables (as we currently do)

or they can be directly translated to SMT array variables. Nevertheless, since SMT solvers highly differ in the treatment given to different theories and logics, much experimentation has to be done in order to decide a suitable encoding for every construct. The positive aspect of this is that we can look at `Simply` as a platform for SMT benchmarking.

From the aforementioned experimentation, we would like `Simply` to be able to automatically determine a suitable logic for each problem. Another option could be letting the user indicate the desired target logic.

As said in Section 3.2, we need to provide a translation-back module for every SMT solver, in order to translate the model found (if any) to a set of values for the original `Simply` program variables. This module will not be unique since there is no standard language for the SMT solver solution [8] answers.

Finally, a less important aspect that has to be improved is error messaging.

5.2 The Input Language

The input language has to be extended in several directions. For example, input/output operations should be added in order to, e.g., be able to load a problem instance from a file. We also want to let the users define their own predicate constraints. Concerning the declarativeness of the language, we plan to add more global constraints as, for instance, *cumulative*, *circuit* and *element*. We also plan to add *set* variables and therefore to provide global constraints on them. We believe that the use of the *bit-vectors* theory for codifying those constraints can result in compact SMT instances that can be solved efficiently.

The addition of optimization capabilities to `Simply` heavily depends on having those capabilities already implemented in SMT solvers. However, apart from the work reported in [12], optimization is not the main interest of the SMT community and so it is not supported in the currently available SMT solvers. Certainly, a possibility to circumvent the fact that current SMT solvers do not provide support for optimization, could be to transform the optimization problem into a decision problem by adding the objective function as an additional constraint, and then compute the optimum with (e.g. linear or binary) search. However, this would have the drawback of having to start from scratch at each search.

5.3 Benchmarking

We have done some benchmarking between different SMT solvers on a few classic constraint satisfaction problems. Many additional experiments need to be done in order to know which kind of problems can be solved in reasonable time by using our tool. For this reason we plan, on the one hand, to do some experiments with industrial problems and, on the other hand, to compare the performance of state-of-the-art SMT solvers with other CSP solvers on the same problems. However, let us recall that `Simply` is intended to be, not only a CSP solving tool, but also a benchmark generator for SMT.

[8] There exist SMT solvers that cannot even be queried for a model.

Acknowledgement

We want to thank the helpful comments and cooperation of Josep Suy.

References

[1] Armando, A., C. Castellini and E. Giunchiglia, *SAT-based procedures for temporal reasoning*, in: S. Biundo and M. Fox, editors, *5th European Conference on Planning, ECP'99*, Lecture Notes in Computer Science **1809** (2000), pp. 97–108.

[2] Barrett, C. W., D. L. Dill and A. Stump, *Checking satisfiability of first-order formulas by incremental translation to SAT*, in: E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV'02*, Lecture Notes in Computer Science **2404** (2002), pp. 236–249.

[3] Cadoli, M., G. Ianni, L. Palopoli, A. Schaerf and D. Vasile, *NP-SPEC: an executable specification language for solving all problems in NP*, Computer Languages **26** (2000), pp. 165–195.

[4] Cadoli, M., T. Mancini and F. Patrizi, *SAT as an effective solving technology for constraint problems*, in: F. Esposito, Z. W. Ras, D. Malerba and G. Semeraro, editors, *Foundations of Intelligent Systems, 16th International Symposium, ISMIS'06*, Lecture Notes in Computer Science **4203** (2006), pp. 540–549.

[5] Cadoli, M. and A. Schaerf, *Compiling problem specifications into SAT*, Artificial Intelligence **162** (2005), pp. 89–120.

[6] Frisch, A. M., W. Harvey, C. Jefferson, B. Martínez-Hernández and I. Miguel, *Essence: A constraint language for specifying combinatorial problems*, Constraints **13** (2008), pp. 268–306.

[7] Ganzinger, H., G. Hagen, R. Nieuwenhuis, A. Oliveras and C. Tinelli, *DPLL(T): Fast Decision Procedures*, in: R. Alur and D. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV'04*, Lecture Notes in Computer Science **3114** (2004), pp. 175–188.

[8] Gent, I. P., C. Jefferson and I. Miguel, *Minion: A fast scalable constraint solver*, in: G. Brewka, S. Coradeschi, A. Perini and P. Traverso, editors, *17th European Conference on Artificial Intelligence, ECAI'06*, Frontiers in Artificial Intelligence and Applications **141** (2006), pp. 98–102.

[9] Kautz, H. A., *Deconstructing planning as satisfiability*, in: *Proceedings of the Twenty-first Conference on Artificial Intelligence, AAAI'06* (2006), pp. 1524–1526.

[10] Mills, P., E. Tsang, R. Williams, J. Ford, J. Borrett and W. Park, *Eacl 1.5: An easy constraint optimisation programming language*, Technical Report CSM-324, University of Essex, Colchester, U.K. (1999).

[11] Nethercote, N., P. J. Stuckey, R. Becket, S. Brand, G. J. Duck and G. Tack, *Minizinc: Towards a standard CP modelling language*, in: C. Bessiere, editor, *Principles and Practice of Constraint Programming, 13th International Conference, CP'07*, Lecture Notes in Computer Science **4741** (2007), pp. 529–543.

[12] Nieuwenhuis, R. and A. Oliveras, *On SAT Modulo Theories and Optimization Problems*, in: *Theory and Applications of Satisfiability Testing, 9th International Conference, SAT'06*, LNCS **4121** (2006), pp. 156–169.

[13] Nieuwenhuis, R., A. Oliveras, E. Rodríguez-Carbonell and A. Rubio, *Challenges in Satisfiability Modulo Theories*, in: F. Baader, editor, *18th International Conference on Rewriting Techniques and Applications, RTA'07*, Lecture Notes in Computer Science **4533** (2007), pp. 2–18.

[14] Nieuwenhuis, R., A. Oliveras and C. Tinelli, *Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)*, Journal of the ACM **53** (2006), pp. 937–977.

[15] Ranise, S. and C. Tinelli, *The SMT-LIB Standard: Version 1.2*, Technical report, Department of Computer Science, The University of Iowa (2006), available at `www.SMT-LIB.org`.

[16] Sebastiani, R., *Lazy satisability modulo theories*, Journal on Satisfiability, Boolean Modeling and Computation **3** (2007), pp. 141–224.

[17] Sheini, H. M. and K. A. Sakallah, *From propositional satisfiability to satisfiability modulo theories*, in: A. Biere and C. P. Gomes, editors, *Theory and Applications of Satisfiability Testing, 9th International Conference, SAT'06*, Lecture Notes in Computer Science **4121** (2006), pp. 1–9.

[18] Walsh, T., *SAT vs CSP*, in: R. Dechter, editor, *Principles and Practice of Constraint Programming, 6th International Conference, CP'00*, Lecture Notes in Computer Science **1894** (2000), pp. 441–456.

[19] Zhang, H., D. Li and H. Shen, *A SAT based scheduler for tournament schedules*, in: *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT'04, Online Proceedings*, 2004, pp. 191–196.
URL http://www.satisfiability.org/SAT04/programme/74.pdf