A System for Solving Constraint Satisfaction Problems with SMT

Miquel Bofill, Josep Suy, and Mateu Villaret*

Departament d'Informàtica i Matemàtica Aplicada
Universitat de Girona
E-17003 Girona, Spain
{mbofill,suy,villaret}@ima.udg.edu

Abstract. SAT Modulo Theories (SMT) consists of deciding the satisfiability of a formula with respect to a decidable background theory, such as linear integer arithmetic, bit-vectors, etc, in first-order logic with equality. SMT has its roots in the field of verification. It is known that the SAT technology offers an interesting, efficient and scalable method for constraint solving, as many experimentations have shown. Although there already exist some results pointing out the adequacy of SMT techniques for constraint solving, there are no available tools to extensively explore such adequacy. In this paper we introduce a tool for translating FlatZinc (MiniZinc intermediate code) instances of constraint satisfaction problems to the standard SMT-LIB language. It can be used for deciding satisfiability as well as for optimization. The tool determines the required logic for solving each instance. The obtained results suggest that SMT can be effectively used to solve CSPs.

1 Introduction

Over the last decade there have been important advances in the Boolean satisfiability (SAT) solving techniques, 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 [2], an application that compiles specifications written in a declarative modeling language into SAT is shown to give promising results. Interesting comparisons between SAT and Constraint Satisfaction Problems (CSP) encodings and techniques can be found in [13].

SAT techniques have been adapted for more expressive logics. For instance, in the case of *Satisfiability Modulo Theories (SMT)*, the problem is to decide the satisfiability of a formula with respect to a decidable background theory (or combinations of them) in first order logic with equality [9, 11]. Some of these theories are (quantifier free) *Linear Integer Arithmetic* (QF_LIA), *Integer Difference Logic* (QF_IDL), *Linear Real Arithmetic* (QF_LRA), *Uninterpreted Functions* (QF_UF), *Non-linear Integer Arithmetic* (QF_NIA), etc [10].

* All authors partially supported by the Spanish Ministry of Science and Innovation through the project SuRoS (ref. TIN2008-04547/TIN)

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. Adaptations of SAT techniques to the SMT framework have been described in [12]. Although most SMT solvers are restricted to decidable quantifier free fragments of their logics, this suffices for many applications. The main application area of SMT is hardware and software verification. Nevertheless, there are already promising results in the direction of adapting SMT techniques for solving CSPs (see e.g. [1]) even in the case of combinatorial optimization (see e.g. [7]). Fundamental challenges on SMT for constraint programming and optimization are detailed in [8].

Since the beginning of Constraint Programming (CP), its *holy grail* has been to obtain a declarative language allowing users to easily specify their problem and forget about the techniques required to solve it. Among many others [2, 4, 6], *MiniZinc* [6] is proposed to be a standard CP modeling language. CSP models and data are written in the MiniZinc language which, after compilation, result into CSP instances codified in a sort of intermediate code called *FlatZinc*. Several solvers, such as Gecode, ECL^{i}PS^e and SICStus Prolog, provide specialized backends for this intermediate language.

In this paper we introduce a tool called `fzn2smt`[1] for solving FlatZinc CSP instances through SMT. Our work is similar to that of [1], where a compiler from a declarative language to the standard SMT-LIB language [10] was developed, and to that of FznTini [5], that solves FlatZinc CSP instances through SAT. As FznTini, our system `fzn2smt` does not only solve decision problems but also optimization problems, and uses FlatZinc as input language, supporting all its standard data types and constraints. The logic required for solving each instance is determined automatically by `fzn2smt` during the translation.

2 Architecture of the Tool

The architecture of `fzn2smt` is depicted in Fig. 1 throughout the process of compiling and solving. The input of the compiler is a FlatZinc instance, which is translated into an SMT instance (in the standard SMT-LIB format) and fed into an SMT solver. Due to the large number of existing SMT solvers, each one supporting different combinations of theories, the user can choose which solver to use (default being Yices [3]).

FlatZinc has three solving options: `solve satisfy`, `solve minimize` x and `solve maximize` x, where x is an integer variable. Since currently optimization is not supported in the SMT-LIB language, we have naively implemented it by means of iterative calls performing a binary search on the domain of the variable to optimize. Moreover, since there is no standard output model in the SMT-LIB language[2], we need a specialized *recover module* for each solver in order to obtain the answers in the standard FlatZinc output format. In this work we have only

[1] `fzn2smt` can be downloaded from `http://ima.udg.edu/Recerca/GrupESLIP.html`.

[2] There are even solvers that only return `sat, unsat` or `unknown`.

built the one for Yices. As a byproduct, `fzn2smt` also generates the corresponding SMT instance and model.

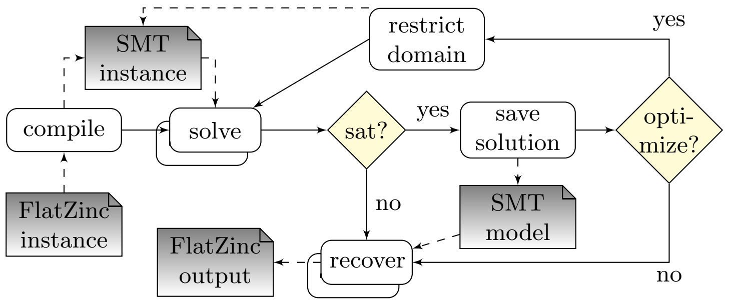

Fig. 1. The compiling and solving process of `fzn2smt`.

3 Some Hints on the Translation

FlatZinc has two categories of data: constants and variables. It provides three scalar types (booleans, integers and floats) and two compound types (sets of integers and one-dimensional integer-indexed arrays). Scalar type domains can be bounded by an interval or a list. Our translation goes as follows:

– Scalar types are translated into their equivalent in SMT (`boolean` to `Bool`, `integer` to `Int` and `float` to `Real`). Constraining a variable to its domain results in a disjunction of equalities when the domain is an explicit enumeration of values, or into a conjunction of two inequalities when the domain is described as a range. Instantiated data (i.e., constants and instantiated variables) is always replaced by its value.
– Arrays have two parametric possible translations:
  - Using QF_UF. Each array is translated into an uninterpreted function with the same name. E.g., `array[1..8] of var int:p` is translated into the uninterpreted function `p()` of type `Int` $\rightarrow$ `Int`. Accesses to the array `p` such as `p[i]`, where `i` is a constant, are translated into `p(i)`. For undetermined references to arrays, FlatZinc provides the constraint `array_int_element(i,t,p)`, that we transform into `p(i) = t` $\wedge$ `1` $\leq$ `i` $\leq$ `8`. Constraints referring to all positions of an array are expanded: e.g., `array_bool_and(p,res)` becomes `res = and(p(1)...p(8))`.
  - Decomposing the vector into as many base type variables as elements are in the vector. E.g., `array[1..8] of var int:p` is translated into `p_1,...,p_8` integer variables.

We want to remark that the SMT theory of arrays involves read and write operations and, hence, is intended to be used for modelling state change

of programs with arrays. However, since there are only read operations in CP models (i.e., there is no notion of state) it suffices to use an uninterpreted function for every array, translating every read operation of the form `read(a,i)` into `a(i)`. This is enough since the theory of equality and uninterpreted functions guarantees that `a(i)=a(j)` whenever `i=j`. Moreover, deciding satisfiability of sets of equalities involving uninterpreted functions is far more cheaper than using the arrays theory. Nevertheless, preliminary experimentation on FlatZinc instances has shown us that decomposing arrays into their elements provides a better performance than the UF approach.
– Sets can only be defined on a range or a list of integers. For this reason, we simply use a Boolean variable for every possible element, indicating whether it belongs to the set or not. In order to make it easier for some set restrictions, each Boolean variable has a partner 0-1 integer variable. Arrays of sets are always expanded.

We have implemented the translation into SMT of all FlatZinc constraints. During the compilation process it is crucial to detect the logic needed. For example, the `int_times(a,b,c)` constraint states `a*b=c`. This falls into linear arithmetic if `a` or `b` are constants or instantiated variables. Otherwise, non-linear arithmetic is necessary. Since few SMT solvers support non-linear arithmetic, when `a` and `b` are variables, but bounded by an interval or a list, we linearize this constraint by enumerating the possible values of the variable with smaller domain as follows: `a=1→1*b=c, a=2→2*b=c, ...`

4 Benchmarking and Conclusions

We have run the most of the FlatZinc instances of the problems provided with the MiniZinc 1.0.3 distribution[3]. In Table 1 we report the number of solved instances (within parenthesis) and the total time spent in them for each solver. The times are the sum of the translation, when needed (e.g., `fzn2smt` translates from FlatZinc to the SMT-LIB format), plus the solving time. We indicate in boldface the cases with more solved instances, breaking ties by total time. The experiments have been performed on an Intel Core i5 CPU at 2.66 GHz, with 4GB of RAM, running openSUSE 11.2 (kernel 2.6.31).

From these experiments we can observe that `fzn2smt` is the system that solves more instances. Although SMT solvers are black-box and our experimentation is not exhaustive, looking at the results more carefully we observe that `fzn2smt` outperforms other systems in problems with many constraints and non-trivial arithmetic (cars, carseq, cutstock, jobshop, nsp, rcpsp). In [5] it was already shown that FznTini, based on (plain) SAT encoding, was competitive with specialized systems. By adding theories we go one step further. It is worthy to

[3] We have omitted the `2DBinPacking` and `QCP` benchmarks due to errors in the translation from MiniZinc to FlatZinc, some instances of `nsp` due to domain errors in specification, and the instances of `debruijn` which required more than 5 minutes for translation from MiniZinc to FlatZinc.

Table 1. Performance of different tools on FlatZinc instances. 's' stands for satisfaction and 'o' for optimization. # stands for the number of instances run of each benchmark. Time in seconds(solved). Timeout is 300 seconds for each instance.

Problem		#	G12	ECL^{i}PS^e	Gecode	FznTini	`fzn2smt`
alpha	s	1	**0.01(1)**	0.47(1)	0.07(1)	0.60(1)	0.66(1)
areas	s	4	0.13(4)	2.04(4)	**0.05(4)**	0.38(4)	4.75(4)
bibd	s	14	118.16(12)	4.18(7)	35.58(7)	**353.78(13)**	79.48(12)
cars	s	79	0.02(1)	0.81(1)	295.64(3)	1.21(1)	**2271.17(21)**
carseq	s	81	0.47(2)	3.47(2)	0.12(2)	0.06(2)	**2502.95(45)**
curriculum	s	3	0.27(2)	95.09(2)	261.82(1)	**8.70(3)**	9.97(3)
cutstock	o	121	0.01(1)	0.49(1)	0.01(1)	1.53(1)	**1066.29(20)**
debruijn_binary	s	7	37.93(7)	12.40(6)	**10.33(7)**	0.09(1)	0.65(1)
eq	s	1	**0.01(1)**	0.49(1)	**0.01(1)**	20.14(1)	0.47(1)
golfers	s	9	113.7(5)	0.59(1)	**22.64(5)**	9.51(2)	26.30(2)
golomb	o	10	**251.18(9)**	85.84(8)	23.83(8)	23.87(6)	41.88(6)
jobshop	o	74	0.11(1)	1.92(2)	22.14(2)	514.70(4)	**1113.04(22)**
kakuro	s	6	**0.06(6)**	2.97(6)	**0.06(6)**	-(0)	4.67(6)
knights	s	4	**0.05(4)**	1.99(4)	0.27(4)	0.32(4)	3.08(4)
langford	s	25	52.19(20)	121.85(20)	**34.54(20)**	310.24(18)	50.52(20)
latin-squares	s	7	**5.98(6)**	12.54(6)	15.35(6)	129.40(3)	7.54(4)
magicseq	s	9	25.17(7)	21.56(7)	**7.39(7)**	0.30(3)	11.01(4)
nmseq	s	10	281.73(6)	-(0)	**171.03(7)**	-(0)	1.42(1)
nsp	s	20	-(0)	-(0)	0.09(1)	402.22(14)	**70.28(15)**
pentominoes	s	7	113.68(4)	27.88(2)	**208.52(5)**	12.81(1)	4.85(1)
photo	o	2	0.10(2)	1.07(2)	**0.04(2)**	0.08(2)	0.78(2)
quasigroup7	s	10	**1.37(5)**	293.67(3)	3.65(5)	380.28(3)	31.51(5)
queens	s	7	88.88(7)	**36.24(7)**	0.52(3)	94.76(4)	54.61(5)
radiation	o	9	**207.90(7)**	54(6)	231.41(7)	-(0)	584.25(7)
rcpsp	o	10	11.96(2)	49.85(4)	22.54(5)	-(0)	**581.13(8)**
schur_numbers	s	3	1.30(3)	1.03(2)	**0.30(3)**	0.02(2)	1.45(3)
search_stress	s	1	**0.01(1)**	0.54(1)	**0.01(1)**	0.16(1)	0.44(1)
shortest_path	o	10	4.36(4)	292.15(6)	**141.34(7)**	-(0)	45.79(6)
slow_convergence	s	10	68.74(10)	12.84(7)	**11.03(10)**	36.98(4)	222.37(10)
steiner-triples	s	6	0.13(2)	0.50(1)	0.01(1)	65.71(2)	**96.33(5)**
still_life	o	10	17.35(8)	80.03(8)	134.12(9)	60.29(8)	**25.20(9)**
talent_scheduling	o	11	10.43(3)	-(0)	**4.03(3)**	128.10(2)	45.71(3)
template_design	o	7	126.20(2)	1.40(1)	56.99(2)	14.91(1)	**37.37(2)**
tents	s	3	0.10(3)	1.52(3)	**0.04(3)**	0.30(3)	2.50(3)
trucking_hl	o	5	**3.82(5)**	-(0)	41.56(5)	-(0)	3.90(5)
Total		596	1543(163)	1221(132)	1757(164)	2571(114)	**9004(267)**

notice that Gecode, G12 and ECL^{i}PS^e, since are search based systems, can take profit of annotations in order to use particular strategies orienting the search, whilst SAT and SMT do not. Surprisingly, our naive binary search approach to optimization is also giving good performance.

The obtained results suggest that SMT can be effectively used for CSP solving in a broad sense, i.e., not just for specialized problems. Hence the tool could serve for getting a big enough picture of the suitability of SMT solvers w.r.t. CSP solving in general, and to compare the performance of state-of-the-art SMT solvers outside the SMT competition.

We have not used any MiniZinc global constraint (such as, e.g., `cumulative`, `alldifferent`, ...) since they are not supported by current SMT solvers. We think that developing specialized solvers for such theories in SMT is a promising research line. Finally, we think that better results could be obtained if translating directly from the MiniZinc language to SMT. In doing so, most clever translations could be possible and probably less variables could be generated.

References

1. Bofill, M., Palahi, M., Suy, J., Villaret, M.: SIMPLY: a Compiler from a CSP Modeling Language to the SMT-LIB Format. In: Proceedings of the 8th Intl. Workshop on Constraint Modelling and Reformulation. pp. 30–44 (2009)
2. Cadoli, M., Mancini, T., Patrizi, F.: SAT as an Effective Solving Technology for Constraint Problems. In: Esposito, F., Ras, Z.W., Malerba, D., Semeraro, G. (eds.) 16th Intl. Symposium on Foundations of Intelligent Systems. LNCS, vol. 4203, pp. 540–549. Springer (2006)
3. Dutertre, B., de Moura, L.: The Yices SMT solver. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf` (August 2006)
4. Frisch, A., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A Constraint Language for Specifying Combinatorial Problems. Constraints 13(3), 268–306 (2008)
5. Huang, J.: Universal Booleanization of Constraint Models. In: Stuckey, P.J. (ed.) 14th Intl. Conf. on Principles and Practice of Constraint Programming. LNCS, vol. 5202, pp. 144–158. Springer (2008)
6. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: MiniZinc: Towards a Standard CP Modelling Language. In: Bessiere, C. (ed.) 13th Intl. Conf. on Principles and Practice of Constraint Programming. LNCS, vol. 4741, pp. 529–543. Springer (2007)
7. Nieuwenhuis, R., Oliveras, A.: On SAT Modulo Theories and Optimization Problems. In: Biere, A., Gomes, C.P. (eds.) 9th Intl. Conf. on Theory and Applications of Satisfiability Testing. LNCS, vol. 4121, pp. 156–169. Springer (2006)
8. Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Challenges in Satisfiability Modulo Theories. In: Baader, F. (ed.) 18th Intl. Conf. on Rewriting Techniques and Applications. LNCS, vol. 4533, pp. 2–18. Springer (2007)
9. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
10. Ranise, S., Tinelli, C.: The SMT-LIB Standard: Version 1.2. Tech. rep., Dept. of Comp. Science, University of Iowa (2006), available at `www.SMT-LIB.org`
11. Sebastiani, R.: Lazy Satisfiability Modulo Theories. Journal on Satisfiability, Boolean Modeling and Computation 3(3-4), 141–224 (2007)
12. Sheini, H., Sakallah, K.: From Propositional Satisfiability to Satisfiability Modulo Theories. In: Biere, A., Gomes, C.P. (eds.) 9th Intl. Conf. on Theory and Applications of Satisfiability Testing. LNCS, vol. 4121, pp. 1–9. Springer (2006)
13. Walsh, T.: SAT vs CSP. In: Dechter, R. (ed.) 6th Intl. Conf. on Principles and Practice of Constraint Programming. LNCS, vol. 1894, pp. 441–456. Springer (2000)