SIMPLY: a Compiler from a CSP Modeling Language to the SMT-LIB Format*

Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret

Departament d'Informàtica i Matemàtica Aplicada
Universitat de Girona
E-17071 Girona, Spain
{mbofill,mpalahi,suy,villaret}@ima.udg.edu

Abstract. In this paper we introduce `Simply`, a compiler from a declarative language for CSP modeling to the standard SMT-LIB format. The current version of `Simply` is able to generate problem instances falling into the quantifier free linear integer arithmetic logic. The compiler has been developed with the aim of building a system for easy CSP modeling and solving. By taking advantage of the year-over-year increase in performance of SMT solvers, we hope that such a system can serve as an alternative to other decision procedures in many applications. The compiler can also be used for easy SMT benchmark generation.

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 [2], an application that compiles specifications written in a declarative modeling language into SAT is shown to give promising results. See also [7, 17] 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 [16].

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, such as the theory of linear (integer or real) arithmetic, arrays, lists, etc., or combinations of them, in first order logic with equality [14]. 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 Hence, an SMT instance is a generalization of a boolean SAT instance in which some propositional variables have been replaced by predicates from the underlying theories, and can contain formulas

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

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 [15].

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 CSPs, even in the case of combinatorial optimization (see, e.g., [10] 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 are detailed in [11].

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 [9] 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 [5] allows the user to specify combinatorial problems in a formal language with natural language and discrete mathematics facilities.

`Simply` is intended to be a declarative programming system for easy modeling and solving of CSPs. Although the richness of its input language does not reach the level of ESSENCE or MiniZinc, its simplicity makes it really practical. The input language of `Simply` (see Fig. 1) is similar to that of EaCL [8] and MiniZinc, and its main implemented features are arrays, `Forall` sentences, comprehension lists, and some global constraints.

```
Problem:queens_8
    Data
        n:=8;
    Domains
        Dom rows=[1..n];
    Variables
        IntVar q[n]::rows;
    Constraints
        AllDifferent([q[i] | i in [1..n]]);
        Forall(i in [1..n-1]) {
            Forall(j in [i+1..n]) {
                q[i]-q[j]<>j-i;
                q[j]-q[i]<>j-i;
            }
        }
```

Fig. 1. `queens_8.y`: A naive encoding for the *8-Queens* problem.

`Simply` works in the spirit of SPEC2SAT [3], which transforms problem specifications written in NP-SPEC [1] 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 [13] instead of SAT instances. Then, the problem can be solved by using any SMT solver supporting the required theories. 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. The system can also serve as a CSP benchmark generator for SMT solvers comparison. Currently, `Simply` generates quantifier-free linear integer arithmetic formulas, but there are plans to extend the compiler in order to deal with other interesting theories such as, e.g., arrays and bit-vectors.

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, and give some details about the compiler. In Section 4 we discuss some benchmarks. Finally, in Section 5 we conclude and discuss 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 [13] 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 the use of 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) [12], 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). Notice that $Solver_T$ only needs to handle conjunctions of literals 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, as we explain in Section 3, there are two logics which are especially relevant, namely, *quantifier free 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 [13] for details). Such 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 difference logic literals can be reduced to the absence of negative cycles in finite weighted graphs, it can be decided in $\mathcal{O}(n^3)$ time by the Bellman-Ford algorithm. For this reason, many solvers give a special treatment to such kind of literals.

3 `Simply`: The tool

In this section we describe the input language and features of `Simply` through an example. Let us consider the *n-Queens* problem: given a number n of queens of the chess game, the problem is to find a position in a board of size $n \times n$, for each queen, such that no queen threatens any other (i.e., they are not in the same column, row or diagonal). A naive model of the *n-Queens* problem (for $n = 8$) in the `Simply` input language can be found in Fig. 1, with an array `q` of `n` integer variables where `q[i]`, for `i` in `1..n`, denotes the row where the queen of column `i` is placed. The problem is solvable iff there exists an assignment for `q`, according to its domain, such that all the posted constraints are satisfied, i.e., all the values of `q` are different (i.e., no two queens are in the same row) and the distances between the indexes (columns) and values (rows) of any two pair of elements of the array are distinct (i.e., no two queens are in the same diagonal).

In Fig. 2 the architecture of `Simply` is depicted throughout the process of compiling and solving the *8-Queens* problem. Let the input of the compiler be the `queens_8.y` file of Fig. 1. In the compilation process all constants are replaced by their associated value, and all variables are translated into SMT integer variables. 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 inequality predicates when the domain is described as a range. The translation of the constraints typically results into a conjunction of QF_LIA predicates and,

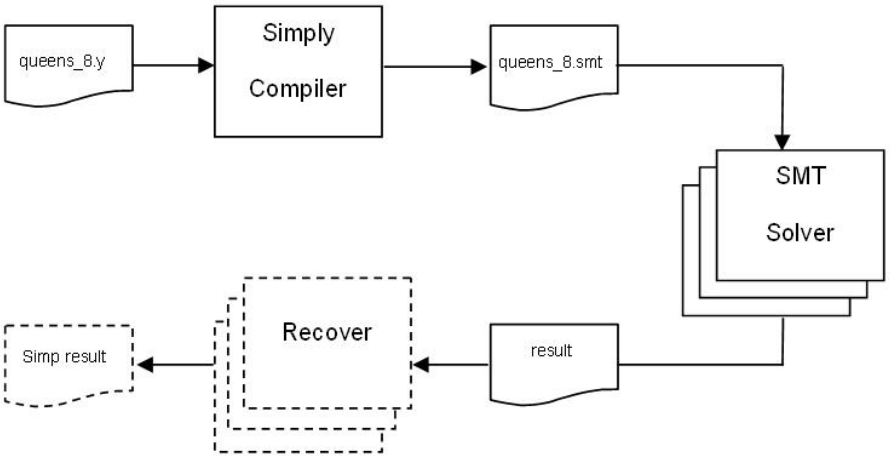

Fig. 2. The architecture of `Simply`.

in some occasions (as in this case for `queens_8.y`), into a conjunction of QF_IDL predicates. In the end, the compilation process produces a `queens_8.smt` file (see Fig. 3) in the standard SMT-LIB format.

The generated SMT problem instance can then be solved by any of the SMT solvers supporting the QF_LIA logic. Solving `queens_8.smt` with the desired SMT solver will result into a *sat* or *unsat* answer (notice that these solvers are complete). In addition, some of them (e.g., Yices [4]) can return a model (in particular, as shown in Fig. 4, the values of the SMT variables) when the problem is satisfiable. In general the names of the variables are easy to interpret. However, we have left as future work the recovering process from SMT solutions to values of the variables in the original (`queens_8.y`) file, as there is no standard format for solution output.

3.1 Structure of the `Simply` input files

Since we are interested in modeling CSPs easily, one of the goals of our tool is simplicity. For this reason, we have chosen the input language of `Simply` to be similar to that of EaCL [8] and MiniZinc [9]. The syntax of our language, with BNF-style production rules following the notation of [13], can be found in the extended version of this paper at: `http://ima.udg.edu/~villaret/simply`.

A CSP instance specification in our language has four parts:

1. `Data` definition. This is where the *constants*, that will be used in the rest of the specification, are defined. These can be either integer or boolean constants and their associated expressions must be evaluable at compilation time.

```
(benchmark queens_8.smt
  :source {Generated by Simpl.y, ima.udg.edu (ESLiP)}
  :category {testing}
  :logic QF_IDL
  :extrafuns ((q_1 Int) ..... (q_8 Int))
  :formula
 (and
  (and
   (and (>= q_1 1) (<= q_1 8))
    ...
   (and (>= q_8 1) (<= q_8 8))
  )
  (and
   (distinct  q_1 ... q_8)
   (and
     (and (distinct (- q_1 q_2) (- 2 1)) (distinct (- q_2 q_1) (- 2 1))
          ...
          (distinct (- q_1 q_8) (- 8 1)) (distinct (- q_8 q_1) (- 8 1)) )
     (and (distinct (- q_2 q_3) (- 3 2)) (distinct (- q_3 q_2) (- 3 2))
          ...
          (distinct (- q_2 q_8) (- 8 2)) (distinct (- q_8 q_2) (- 8 2)) )
      ...
     (and (distinct (- q_7 q_8) (- 8 7)) (distinct (- q_8 q_7) (- 8 7)) )
   )
  )
 )
)
```

Fig. 3. `queens_8.smt`: SMT problem resulting from the compilation of the `queens_8.y` file of Fig. 1.

```
sat (= q_1 5)(= q_2 2)(= q_3 8)(= q_4 1)
    (= q_5 4)(= q_6 7)(= q_7 3)(= q_8 6)
```

Fig. 4. The answer of Yices to the `queens_8.smt` benchmarck.

2. `Domains` definition. A *domain* characterizes the set of possible values of a variable. It can be defined by a list with ranges and individual values, or by a comprehension list. The values of these lists must be evaluable at compilation time.
3. `Variables` declaration. A *variable* can denote either an integer, a boolean, or a multidimensional array of integers or of booleans. Integer variables (and elements of integer arrays) are in fact *finite domain variables* and, hence, they must be constrained to some previously defined domain.
4. `Constraints` posting. This is where the problem is modeled, by posting the *set of constraints* that define the feasible solutions of the problem.

3.2 The Constraints

The input language deals with *formulas*, *global constraints* and the `If_Then_Else` constraint.

- Formulas, or boolean expressions, are basic constraints built up from integer and boolean variables and constants. The following operators are supported: `=`, `<>`, `<`, `=<` and `>=` for integers and `Not`, `And`, `Or`, `Xor`, `Implies` and `Iff` 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`. When `List` is empty, `Value` is enforced to be zero.
  - `Count(List,Value,N)`. This constraint states equality between `N` and the number of occurrences of `Value` in `List`. When `List` is empty, `N` is enforced to be zero.
  - `AllDifferent(List)` requires all the elements of `List` to be different.

  Let us remark that the elements of `List`, as well as the `Value` and `N` parameters, are allowed to be arithmetic expressions containing integer variables.
- The `If_Then_Else` (ϕ) {`C1`} {`C2`} constraint states that, when the formula ϕ is satisfied, then the constraints `C1` must be satisfied and, when ϕ is not satisfied, then the constraints `C2` must be satisfied. Let us remark that the formula ϕ does not need to be evaluable at compilation time.

Constraints can be posted either (i) directly, (ii) through the *If-Then-Else* statement (which has nothing to do with the `If_Then_Else` constraint), or (iii) through the *Forall* statement. These two statements are processed at compilation time. For instance, when the compiler finds an *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.

It is important to notice the difference between the *If-Then-Else* statement and the `If_Then_Else` constraint, whose condition, as said, 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];}
```

Since condition `i<4` must be evaluated at compilation time, `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. If `i<4` was to be evaluated during computation, then the following constraint should be posted:

```
If_Then_Else (i<4) {m[i]<>m[i+1];} {m[i]<>m[i-1]; m[i]=m[i-2];} ;
```

The semantics of a *Forall* statement is as usual and can be illustrated with the following example:

```
Forall(i in [2..4]) {m[i]<>m[i-1];}
```

This results into the replication of the constraint `m[i]<>m[i-1]` with the local variable `i` being replaced by the appropriate values (the compilation techniques used for the expansion are quite similar to the ones used for comprehension lists):

```
m[2]<>m[1]; m[3]<>m[2]; m[4]<>m[3];
```

Lists can be extensional, by directly enumerating elements and ranges e.g., `[1, x, 3..5, 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 that we want to generate. Currently, patterns must be arithmetic expressions (in this example, the elements of the bidimensional 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 (e.g., `i<>j`).

3.3 Compilation

The compiler has been implemented in Haskell. The compilation process has two steps: the first step only checks for syntactic compliance and some minor semantic details, and generates an intermediate code. The second step is the one in charge of semantic analysis and the final SMT-LIB code generation. This code generation step distinguishes between expressions that must be evaluated at compilation time (such as, for instance, the expressions in the condition of the *If-Then-Else* statement), or translated into SMT-LIB expressions (for instance a basic constraint).

The names of the variables are preserved from the input file to the resulting SMT-LIB formula. The only exceptions are arrays: for instance, `m[10]` is translated into ten SMT variables `m_1,m_2,...,m_10`.

The function in charge of the final code generation is `codeGeneration`, which receives an `IntermediateCode` and returns a `String` with the generated code. To illustrate the simplicity and convenience of using Haskell for the code generation step, in Fig. 5 we show the `codeGeneration` function for the intermediate code of the global constraint `Count`. First of all, the function unfolds `list` and obtains the value of `value`. Then, it generates the list of the comparisons between the elements of the unfolded list and the value that we are looking for. Next, it applies the `ite` operator to each element *reifying* these comparisons[1]. In the end, it returns the string that enforces the value of `times` to be equal to the sum of the reifications.

[1] Notice that the SMT-LIB `ite` operator will return 1 when the comparison is satisfied and 0 otherwise.

```
codeGeneration :: IntermediateCode -> String
codeGeneration (COUNT list value times)
   = code
     where
       l = (unfoldList list)
       v = (expValue value)
       comps = map (\x -> "(= "++ x ++" "++ v ++")") l
       reifi = map (\x -> "(ite "++ x ++" 1 0 )") comps
       sum = "(+ "++ (concat reifi) ++")"
       code = "(= "++(expValue times)++" "++sum++")"
```

Fig. 5. Haskell code for compiling the `Count` global constraint.

4 Examples and Benchmarks

In the following we describe the decisional CSPs used in our benchmarks and illustrate the use of `Simply` for one of them. The problems are the following:

- *CSPLib* [6] *problem 015, Schur's Lemma*. The problem is to put n balls labelled $\{1, \ldots, n\}$ into 3 boxes so that, for any triple of balls (x, y, z) with $x + y = z$, not all are in the same box. This problem has a solution iff $n < 14$. In the table of Fig. 7, the entry Schurl_i_j denotes the instance of the problem with i balls and j boxes.
- *CSPLib problem 030, Balanced Academic Curriculum Problem (BACP)*. The BACP objective is to design a balanced academic curriculum by assigning periods to courses in a way such that the academic load of each period is balanced, i.e., as similar as possible. The curriculum must obey the following administrative and academic regulations:
  1. Courses must be assigned within a maximum number of academic periods (`n_periods`).
  2. Each course has a number of credits or units that represent the academic effort required to successfully follow it (`course_load[n_courses]`).
  3. Some courses can have other courses as prerequisites.
  4. A minimum amount of academic credits per period is required to consider a student as full time, and a maximum amount of academic credits per period is allowed in order to avoid overload (`load_per_period_lb, load_per_period_ub`).
  5. A minimum number of courses per period is required to consider a student as full time, and a maximum number of courses per period is allowed in order to avoid overload (`courses_per_period_lb, courses_per_period_ub`).

  The goal is to assign a period to every course in a way such that the minimum and maximum academic load for each period, the minimum and maximum number of courses for each period, and the prerequisite relationships are satisfied. In Fig. 6 we can find the modeling used in the

benchmarks for this problem. Using the `Count` global constraint we restrict the number of courses per period. Notice that `course_per_period[t]` has domain `[courses_per_period_lb .. courses_per_period_lb]`. With the *Forall* statement we "load" the table of credits per period and with the global constraint `Sum` we restrict the number of credits per period. Notice again that `load_per_period[t]` has domain `[load_per_period_lb .. load_per_period_lb]`. In the table of Fig. 7, the entry Bacp_12_j denotes the instance of the BACP_12 problem with j periods.

```
Problem:bacp_12_6
Data n_courses := 66; n_periods := 6;
     load_per_period_lb := 10; load_per_period_ub := 24;
     courses_per_period_lb := 2; courses_per_period_ub := 10;
Domains
     Dom periods=[1..n_periods];
     Dom addload=[load_per_period_lb..load_per_period_ub];
     Dom addcourses=[courses_per_period_lb..courses_per_period_ub];
     Dom load=[1..5]; Dom load_ext=[0..5];
Variables
     IntVar course_load[n_courses]::load;
     IntVar acourses[n_courses]::periods;
     IntVar mload[n_courses,n_periods]::load_ext;
     IntVar load_per_period[n_periods]::addload;
     IntVar course_per_period[n_periods]::addcourses;
Constraints
// courses load
  course_load[1]=1;
  course_load[2]=3;
  ...
// course prerrequisites
  acourses[7] < acourses[1];
  ...
  Forall(t in [1..n_periods]){
     Count([acourses[j] | j in [1..n_courses]], t, course_per_period[t]);
     Forall(c in [1..n_courses]) {
       If_Then_Else( acourses[c] = t )
           { mload[c,t] = course_load[c]; }
           { mload[c,t] = 0; } ;
     }
     Sum( [ mload[i,t] | i in [1..n_courses] ], load_per_period[t] );
  }
```

Fig. 6. A modeling for the *BACP_12* problem instance with 6 periods.

– *Job-shop scheduling.* A job shop has some machines, each performing a different operation. There are some jobs to be performed and a job is a sequence

of tasks. Each task involves processing by a single machine for some duration and a machine can operate on at most one task at a time. Tasks cannot be interrupted. The goal is, given a deadline, to schedule each job such that its ending time does not exceed the deadline. In the table of Fig. 7, the entry Jobshop_i means an instance of the Job-shop problem with maximal duration i, 5 machines and 8 jobs.
- *Queens* (see Section 3). In the table of Fig. 7, the entry Queens_i means an instance of the queens problem with i queens.

We have run the SMT solvers which participated in the QF_LIA division of the Satisfiability Modulo Theories Competition[2] (SMT-COMP) 2008, namely, Z3.2, MathSAT-4.2, CVC3-1.5, Barcelogic 1.3 and Yices 1.0.10, against some benchmarks generated with `Simply` from the previous problems.[3]

For the sake of comparison, we have run some solvers of different nature on the same problems, namely, G12 MiniZinc 0.9, ECLiPSe 6.0, SICStus Prolog 4.0.7, SPEC2SAT 1.1 and Comet 1.2. The same benchmarks have been used for G12 MiniZinc 0.9, ECLiPSe 6.0 and SICStus Prolog 4.0.7 (after a translation from the MiniZinc modeling language to the FlatZinc low-level solver input language, by using the MiniZinc-to-FlatZinc translator `mzn2fzn`). SPEC2SAT transforms problem specifications written in NP-SPEC into SAT instances in DIMACS format, and thus can work in cooperation with any SAT solver supporting that format. In our tests, we have used SPEC2SAT together with zChaff 2007.3.12. With respect to Comet, only its constraint programming module has been tested. In order for the comparison to be fair, we have preserved as much as possible the modeling used in SMT and avoided the use of any search strategy when dealing with other solvers, as no search control is possible within SMT solvers.

Fig. 7 shows the time in seconds spent by each solver in each problem, with a timeout of 1800 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. The column labeled Simply refers to the `Simply` compilation time. The following 5 columns contain the solving time spent by the different SMT solvers on the generated SMT instances. The rest of columns detail the times (including compilation and solving) spent by solvers of other nature. We can observe the following:

- For the *Queens* problem, G12 obtains the best results. The poor performance of SMT solvers on this problem is probably due to the fact that the modeling (see Fig. 1) results in a SMT instance with only unit clauses, while SMT is well-suited for problems whose satisfiability highly depends on the combination of the boolean structure and the background theory.

[2] SMT-COMP: The SAT Modulo Theories Competition (`http://www.smtcomp.org`).
[3] Since our naive modeling of the *Queens* problem falls into the QF_IDL fragment of QF_LIA, we have made this explicit in the generated instances. In this way, the SMT solvers with a specialized theory solver for QF_IDL can take profit of it.

- For the *BACP* problem, similar results are obtained by SMT solvers, G12 and SICStus Prolog. However, complicated instances around the phase transition are not solved by any of them. The SMT instances generated by `Simply` for this problem include a relevant boolean structure mixed with arithmetic expressions, mainly due to the `Count` and `Sum` global constraints (see Fig. 6). This is probably why some SMT solvers obtain extremely good results on this problem. It is also important to notice that SPEC2SAT fails to solve all the instances, since the generated SAT formula is too big for zChaff. Moreover, Comet is very unstable on this problem.
- For the *Schur's Lemma* problem, good results are obtained by most of the solvers. Surprisingly, G12 consumes all the time with no answer. With the chosen modeling, `Simply` is able to generate an SMT instance with no arithmetic at all, since all expressions can be evaluated at compile time.
- For the *Job-shop* problem, best results are obtained by some of the SMT solvers and by ECLiPSe. This is again a problem with a relevant boolean structure, and some arithmetic.

Globally, it seems that most of the SMT solvers are good in all the problems considered. This is especially relevant if we take into account that those solvers come from the verification arena and, therefore, have not been designed with those kind of constraint problems in mind. Moreover, they seem to scale up very well with the size of the problems. Let us remark that these problems are just the first ones at hand that we have considered, i.e., we have not artificially chosen them. For this reason, SMT can be expected to provide a nice compromise between expressivity and efficiency for solving CSPs in some contexts.

5 Conclusion and further work

We have presented `Simply`, a tool for easy CSP modeling and solving, whose main novelty is the generation of SMT problem instances in the standard SMT-LIB format as output. Our aim is to take advantage from the improvements that take place from year to year in SMT technology and methods, in order to solve CSPs. Our tool can also serve as a CSP benchmark generator for SMT solvers comparison. However, 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:

Features of the tool Probably the most important aspect of future work is to study the way of obtaining better SMT encodings from our modeling 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, (unidimensional) arrays of integers in `Simply` programs can be flattened into integer variables (as we currently do) or they can be directly translated into SMT array variables. Nevertheless, since SMT solvers highly differ in the treatment given to different theories and logics, more experimentation has to be done in

	Simply + SMT solver						Other tools				
	Simply (compilation)	Z3.2	MathSAT-4.2	CVC3-1.5	Barcelogic 1.3	Yices 1.0.10	G12 MiniZinc 0.9	mzn2fzn + ECLiPSe 6.0	mzn2fzn + SICStus 4.0.7	SPEC2SAT 1.1	Comet 1.2
Queens_50	0.22	t.o.	53.00	m.o.	11.72	29.47	0.22	2.04	6.98	248.01	t.o.
Queens_100	0.72	t.o.	t.o.	m.o.	389.04	19.22	0.84	t.o.	28.51	t.o.	t.o.
Queens_150	1.54	t.o.	t.o.	m.o.	995.94	t.o.	150.40	t.o.	256.18	t.o.	t.o.
Bacp_12_6	0.17	0.55	2.53	t.o.	56.98	0.19	0.84	t.o.	3.8	m.o.	268.56
Bacp_12_7	0.18	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	m.o.	t.o.
Bacp_12_8	0.22	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	m.o.	t.o.
Bacp_12_9	0.21	0.27	10.86	t.o.	314.91	0.64	0.94	t.o.	5.3	m.o.	0.51
Bacp_12_10	0.24	0.24	14.97	t.o.	190.10	0.79	1.44	t.o.	6.02	m.o.	0.60
Bacp_12_11	0.24	0.27	13.60	t.o.	237.50	1.24	1.70	t.o.	7.97	m.o.	19.56
Bacp_12_12	0.26	0.48	13.24	t.o.	338.46	1.32	40.59	t.o.	11.32	m.o.	t.o.
Schurl_12_3	0.06	0.01	0.08	7.91	0.03	0.02	t.o.	t.o.	0.24	0.38	0.39
Schurl_13_3	0.08	0.04	0.07	14.30	0.05	0.05	t.o.	t.o.	0.28	0.55	0.40
Schurl_14_3	0.09	0.23	0.50	18.24	0.12	0.16	t.o.	t.o.	0.32	0.50	0.40
Schurl_15_3	0.10	0.35	0.79	29.15	0.15	0.18	t.o.	t.o.	0.37	0.73	0.40
Jobshop_54	0.31	0.12	0.27	104.97	7.79	2.69	34.13	1.54	33.30	80.41	1.79
Jobshop_55	0.32	0.20	0.35	211.48	11.57	3.63	122.16	2.16	t.o.	80.03	11.65
Jobshop_56	0.30	0.12	0.46	358.53	12.08	4.37	396.03	3.13	t.o.	88.11	100.01
Jobshop_57	0.30	0.34	0.89	475.55	16.05	6.62	1115.09	1.13	t.o.	85.66	892.54
Jobshop_58	0.34	0.10	0.25	134.71	20.75	11.48	0.09	1.22	236.64	95.11	0.82

Fig. 7. Time in seconds spent on solving the problems detailed in Section 4. The first column refers to `Simply` compilation time. Time out (t.o.) was set to 1800 seconds. Memory out is denoted by m.o.

order to decide a suitable encoding for every construct. 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. This would make `Simply` be a more complete benchmark generator.

Also, as said in Section 3, 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 of the original `Simply` program variables. This module will not be unique since

there is no standard language for SMT solver solution answers[4]. Nevertheless, the definition of a standard language for models of satisfiable benchmarks is planned in the SMT-LIB definition document [13].

The compiler can also be improved in order to obtain SMT formulas without unevaluated subexpressions that could be evaluated at compilation time. For instance `(+ 3 (+ a 10))` should be `(+ a 13)`.

Input language The input language can 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. The language could also be enriched so that users can define their own predicate constraints. Formula operators could be extended by allowing lists of formulas as parameters. This would allow comprehension lists to generate formulas too instead of just arithmetic expressions.

Concerning the declarativeness of the language, more global constraints such as, for instance, *cumulative*, *circuit* and *element* could be added. It could also be interesting to add *set* variables and therefore to provide global constraints on them. We believe that the use of the *bit-vectors* theory for encoding those constraints would result in compact SMT instances that could be solved efficiently.

The addition of optimization capabilities to `Simply` heavily depends on having those capabilities already implemented in SMT solvers. However, optimization is not the main interest of the SMT community and so it is only supported by some SMT solvers, such as Yices [4]. For SMT solvers not having optimization capabilities, one possibility 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. This is roughly what is done in [10] although, in this case, it is done internally by the solver. However, doing this from outside the solver would have the drawback of having to start from scratch at each search.

Finally, a complementary and interesting direction of research can be the development of a compiler from the MiniZinc or the FlatZinc languages to the SMT-LIB format, or even to the native language of some concrete SMT solver (in order to exploit some particular capabilities of it).

Benchmarking We have done some benchmarking between SMT solvers and some solvers of different nature, on a few classic constraint satisfaction problems. In most of them, the performance of the SMT solvers has turned out to be similar or even better than of others (without strategies). However, 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 example, less naive formulations (e.g., with symmetry breaking) of classic problems need to be tested. We also plan to do some experiments with industrial problems and compare the performance of state-of-the-art SMT solvers with other CSP solvers on the same problems.

Linux and Windows binaries of `Simply`, as well as documentation and some benchmarks, can be found at: `http://ima.udg.edu/~villaret/simply`.

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

References

1. Marco Cadoli, Giovambattista Ianni, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. NP-SPEC: an executable specification language for solving all problems in NP. *Computer Languages*, 26(2–4):165–195, July 2000.
2. Marco Cadoli, Toni Mancini, and Fabio Patrizi. SAT as an effective solving technology for constraint problems. In *Foundations of Intelligent Systems, 16th Intl. Symposium, ISMIS'06*, volume 4203 of *LNCS*, pages 540–549. Springer, 2006.
3. Marco Cadoli and Andrea Schaerf. Compiling problem specifications into SAT. *Artificial Intelligence*, 162(1–2):89–120, 2005.
4. B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper available at http://yices.csl.sri.com/tool-paper.pdf, August 2006.
5. Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.
6. Ian P. Gent and Toby Walsh. CSP_{LIB}: A benchmark library for constraints. In *CP*, volume 1713 of *LNCS*, pages 480–481. Springer, 1999.
7. Henry A. Kautz. Deconstructing planning as satisfiability. In *Proceedings of the Twenty-first Conference on Artificial Intelligence, AAAI'06*, pages 1524–1526. AAAI Press, 2006.
8. Patrick Mills, Edward Tsang, Richard Williams, John Ford, James Borrett, and Wivenhoe Park. Eacl 1.5: An easy constraint optimisation programming language. Technical Report CSM-324, University of Essex, Colchester, U.K., 1999.
9. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming, 13th International Conference, CP'07*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
10. R. Nieuwenhuis and A. Oliveras. On SAT Modulo Theories and Optimization Problems. In *Theory and Applications of Satisfiability Testing, 9th International Conference, SAT'06*, volume 4121 of *LNCS*, pages 156–169. Springer, 2006.
11. Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Challenges in Satisfiability Modulo Theories. In *18th International Conference on Rewriting Techniques and Applications, RTA'07*, volume 4533 of *LNCS*, pages 2–18. Springer, 2007.
12. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 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.
13. Silvio Ranise and Cesare 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`.
14. Roberto Sebastiani. Lazy satisability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3(3-4):141–224, 2007.
15. Hossein M. Sheini and Karem A. Sakallah. From propositional satisfiability to satisfiability modulo theories. In *Theory and Applications of Satisfiability Testing, 9th Intl. Conference, SAT'06*, volume 4121 of *LNCS*, pages 1–9. Springer, 2006.
16. Toby Walsh. SAT vs CSP. In *Principles and Practice of Constraint Programming, 6th International Conference, CP'00*, volume 1894 of *LNCS*, pages 441–456. Springer, 2000.
17. Hantao Zhang, Dapeng Li, and Haiou Shen. A SAT based scheduler for tournament schedules. In *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT'04, Online Proceedings*, pages 191–196, 2004.