Noname manuscript No.
(will be inserted by the editor)

Solving Constraint Satisfaction Problems with SAT Modulo Theories

Miquel Bofill · Miquel Palahí · Josep Suy · Mateu Villaret

the date of receipt and acceptance should be inserted later

Abstract Due to significant advances in SAT technology in the last years, its use for solving constraint satisfaction problems has been gaining wide acceptance. Solvers for satisfiability modulo theories (SMT) generalize SAT solving by adding the ability to handle arithmetic and other theories. Although there are results pointing out the adequacy of SMT solvers for solving CSPs, there are no available tools to extensively explore such adequacy. For this reason, in this paper we introduce a tool for translating FlatZinc (MiniZinc intermediate code) instances of CSPs to the standard SMT-LIB language. We provide extensive performance comparisons between state-of-the-art SMT solvers and most of the available FlatZinc solvers on standard FlatZinc problems. The obtained results suggest that state-of-the-art SMT solvers can be effectively used to solve CSPs.

1 Introduction

The Boolean satisfiability problem (SAT) is the problem of determining if there exists an assignment to the variables of a Boolean formula that makes it evaluate to true. Over the last decade there have been important advances in SAT solving techniques, to the point that SAT solvers have become a viable engine for solving constraint satisfaction problems (CSPs) [38,16,36].

On the other hand, 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 background theory (or combinations of them) in first order logic with equality [28,32]. SMT has its roots in the field of hardware and software verification and, although most SMT solvers

This is an extended version of a short paper [12] presented at the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT 2010) in Edinburgh, UK. Research partially supported by the Spanish MICINN under grant TIN2008-04547.

Miquel Bofill · Miquel Palahí · Josep Suy · Mateu Villaret
Departament d'Informàtica i Matemàtica Aplicada
Universitat de Girona, Spain
E-mail: {mbofill,mpalahi,suy,villaret}@ima.udg.edu

are restricted to decidable quantifier free fragments of their logics, this suffices for many applications. In fact, there are already promising results in the direction of adapting SMT techniques for solving CSPs, even in the case of combinatorial optimization [26]. Fundamental challenges for SMT for constraint programming and optimization are detailed in [27].

The Satisfiability Modulo Theories Library (SMT-LIB) [8] provides a common standard for the specification of benchmarks and of background theories for SMT. In the field of Constraint Programming (CP) MiniZinc [25] has been proposed as a standard CSP modelling language among many others [37, 22]. MiniZinc model and data files are compiled into model instances written in an intermediate language called FlatZinc. Those instances can then be fed into any existing CSP solver, provided that a FlatZinc front-end is available for it.

With the aim of bringing a bit more of SMT technology to CP, in this paper we introduce a tool for solving FlatZinc instances through SMT called `fzn2smt`.[1] Essentially, `fzn2smt` is based on a translator from the FlatZinc language to the standard SMT-LIB language v1.2 [30]. A suitable logic for solving each problem instance is determined automatically by `fzn2smt` during the translation, and the generated output can be fed into any SMT solver, which is used as a black box.

Our work is similar to that of FznTini [23], where FlatZinc instances are translated into SAT, and also to that of Simply [11], where a compiler from a declarative language to the standard SMT-LIB language was developed. Roughly speaking, the translator inside `fzn2smt` has the same input language as FznTini, and the same output language as Simply. Moreover, like FznTini, `fzn2smt` is able to solve optimization problems by means of successive calls to the decision procedure, performing either linear, binary or a kind of hybrid search. However, only binary search is supported by FznTini.

We provide extensive experimentation with `fzn2smt` (using different SMT solvers) and other FlatZinc solvers. We also compare distinct encodings and optimization methods for `fzn2smt`. The good results obtained by `fzn2smt` suggest that state-of-the-art SMT solvers can effectively be used to solve CSPs. We remark that scheduling problems are the ones on which we obtained the best results. Interestingly, these problems have a significant amount of disjunctions (non-unary clauses) and Boolean variables, which we believe allow the SMT solvers to profit from specialized built-in techniques such as unit propagation, learning and backjumping.

The rest of the paper is organized as follows. In Section 2 we briefly introduce MiniZinc and SMT. In Section 3 we describe `fzn2smt` and point out the main aspects of the translation from FlatZinc to the standard SMT-LIB language v1.2. Section 4 is devoted to performance comparisons between several SMT solvers within `fzn2smt` and, on the other hand, between several FlatZinc back-ends and `fzn2smt`. We provide some statistical tests on performance. Finally, in Section 5 we conclude and point out some possible future work.

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

2 Preliminaries

2.1 CSPs, MiniZinc and FlatZinc

A *Constraint Satisfaction Problem* (CSP) consists of a set of variables with associated domains plus a set of constraints (relations) on the variables. More formally, a CSP is a tuple $\langle X, D, C\rangle$ where $X = \{x_1, \ldots, x_n\}$ is a set of variables with domains $D = \{D_1, \ldots, D_n\}$ and C is a set of constraints. A constraint $c \in C$ is a relation $c \subseteq D_1 \times \cdots \times D_n$. A solution is any assignment of values to the variables within their domains such that satisfies all the constraints, i.e., given a CSP $\langle X, D, C\rangle$, a tuple $d \in D$ is one of its solutions if and only if $d \in c$ for all $c \in C$. When the problem is an optimization one, the solution must also optimize a given cost function.

Typically, CSPs are divided into two parts: the *model* (where a parametrized specification of the problem is given) and the *data* (where particular values are given). The model plus the data give us an *instance*.

MiniZinc [25] aims to be a standard language for specifying CSPs (with or without optimization) over Booleans, integers and real numbers. It is a mostly declarative constraint modelling language, although it provides some facilities such as *annotations* for specifying, e.g., search strategies, that can be taken into account by the underlying solver. One of the most appealing features of MiniZinc is that it can be easily mapped onto different existing solvers, by previously compiling its model and data files into FlatZinc instances. FlatZinc is a low-level solver input language, for which there exist front-ends for several solvers, such as Gecode [31], ECL^{i}PS^e [6], SICStus Prolog [4], JaCoP [1] and SCIP [3], apart from a number of solvers developed by the G12 research team.

Example 1 This is a MiniZinc toy instance of the well-known Job-shop problem. We use this example later on to illustrate the translation process of our tool.

```
size = 2; d = [| 2,5 | 3,4 |];
int: size; % size of problem
array [1..size,1..size] of int: d; % task durations
int: total = sum(i,j in 1..size) (d[i,j]); % total duration
array [1..size,1..size] of var 0..total: s; % start times
var 0..total: end; % total end time

predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
    s1 + d1 <= s2 \/ s2 + d2 <= s1;

constraint
  forall(i in 1..size) (
    forall(j in 1..size-1) (s[i,j] + d[i,j] <= s[i,j+1]) /\
    s[i,size] + d[i,size] <= end /\
    forall(j,k in 1..size where j < k) (
      no_overlap(s[j,i], d[j,i], s[k,i], d[k,i])
  )
);
solve minimize end;
```

FlatZinc instances are simply a list of variable declarations and flat constraints, plus (possibly) a variable to optimize. The MiniZinc to FlatZinc translation has two parts (see [25] for details):

- *Flattening*, where several reductions (built-ins evaluation, comprehension lists unrolling, fixed array accesses replacement, etc.) are applied until a fix-point is reached.
- *Post-flattening*, where some normalisation is done. For example, for Boolean expressions that are not top-level conjunctions, each sub-expression is replaced by a new Boolean variable, and constraints equating these new variables with the sub-expressions they replaced are added (and similarly for non-linear numeric expressions).

Example 2 This is the FlatZinc instance resulting from translating the MiniZinc instance of Example 1.

```
var bool: BOOL____1;
var bool: BOOL____2;
var bool: BOOL____3 = true;
var bool: BOOL____4;
var bool: BOOL____5;
var bool: BOOL____6 = true;
array [1..4] of int: d = [2, 5, 3, 4];
var 5..14: end;
array [1..4] of var 0..14: s;
constraint array_bool_or([BOOL____1, BOOL____2], BOOL____3);
constraint array_bool_or([BOOL____4, BOOL____5], BOOL____6);
constraint int_lin_le([-1, 1], [end, s[2]], -5);
constraint int_lin_le([-1, 1], [end, s[4]], -4);
constraint int_lin_le([1, -1], [s[1], s[2]], -2);
constraint int_lin_le([1, -1], [s[3], s[4]], -3);
constraint int_lin_le_reif([1, -1], [s[1], s[3]], -2, BOOL____1);
constraint int_lin_le_reif([-1, 1], [s[1], s[3]], -3, BOOL____2);
constraint int_lin_le_reif([1, -1], [s[2], s[4]], -5, BOOL____4);
constraint int_lin_le_reif([-1, 1], [s[2], s[4]], -4, BOOL____5);
solve minimize end;
```

Note that the arithmetic expressions have been encoded as linear constraints with the `int_lin_le` constraint.

The G12 MiniZinc distribution [2] is accompanied by a comprehensive set of benchmarks. Moreover, in the MiniZinc challenge [35], which is run every year since 2008 with the aim of comparing different solving technologies on common benchmarks, all entrants are encouraged to submit two models each with a suite of instances to be considered for inclusion in the challenge.

2.2 Satisfiability Modulo Theories (SMT)

An SMT instance is a generalization of a Boolean formula in which some propositional variables have been replaced by predicates with predefined interpretations

from background theories. For example, a formula can contain clauses like, e.g., $p \vee q \vee (x + 2 \leq y) \vee (x > y + z)$, where p and q are Boolean variables and x, y and z are integer variables. Predicates over non-Boolean variables, such as linear integer inequalities, are evaluated according to the rules of a background theory. Examples of theories include linear real or integer arithmetic, arrays, bit vectors, uninterpreted functions, etc., or combinations of them.

Formally speaking, a *theory* is a set of first-order formulas closed under logical consequence. A theory T is said to be *decidable* if there is an effective method for determining whether arbitrary formulas are included in T.

The SMT problem for a theory T is: given a first-order formula F, determine whether there is a model of $T \cup \{F\}$. Usually, T is restricted to be decidable and F is restricted to be quantifier-free so that, while providing a much richer modelling language than is possible with Boolean formulas, the problem is still decidable.

2.2.1 The lazy SMT approach

Although an SMT instance can be solved by encoding it into an equisatisfiable SAT instance and feeding it to a satisfiability checker for propositional logic (a.k.a. SAT solver), currently most successful SMT solvers are based on the integration of a SAT solver and a T-solver, that is, a decision procedure for the given theory T. In this so-called lazy approach, while the SAT solver is in charge of the Boolean component of reasoning, the T-solver deals with sets of atomic constraints in T. The main idea is that the T-solver analyzes the partial model that the SAT solver is building, and warns it about conflicts with the theory T (T-inconsistency). This way, we are hopefully getting the best of both worlds: in particular, the efficiency of the SAT solver for the Boolean reasoning and the efficiency of special-purpose algorithms inside the T-solver for the theory reasoning.

Algorithm 1 shows a simplified version of an enumeration-based T-satisfiability procedure, borrowed from [14], where T-consistency is only checked for total Boolean assignments. We refer the reader to [32] for a survey on the lazy SMT approach.

Algorithm 1 Bool+T

Input: φ : SMT formula
Output: the satisfiability of φ

```
A^p := T2B(Atoms(φ));
φ^p := T2B(φ);
while Bool-satisfiable(φ^p) do
   μ^p := pick_total_assignment(A^p,φ^p);
   (ρ, π) := T-satisfiable(B2T(μ^p));
   if ρ = sat then
      return sat;
   else
      φ^p := φ^p ∧ ¬T2B(π);
   end if;
end while
return unsat;
```

The algorithm enumerates the Boolean models of the propositional abstraction of the SMT formula φ and checks for their satisfiability in the theory T.

- The function Atoms(φ) takes a quantifier-free SMT formula φ and returns the set of atoms which occur in φ.
- The function T2B maps propositional variables to themselves, and ground atoms into fresh propositional variables; φ^p is initialized to be the propositional abstraction of φ using T2B.
- The function B2T is the inverse of T2B.
- μ^p denotes a propositional assignment as a set (conjunction) of propositional literals.
- The function pick_total_assignment returns a total assignment to the propositional variables in φ^p. In particular, it assigns a truth value to all variables in A^p.
- The function T-satisfiable(μ) checks if the set of conjuncts μ is T-satisfiable, i.e., if there is a model for $T \cup \mu$, returning (`sat`,$\emptyset$) in the positive case and (`unsat`,π) otherwise, being $\pi \subseteq \mu$ a T-unsatisfiable set (the theory conflict set). Note that the negation of the propositional abstraction of π is added to φ^p in case of `unsat` (learning).

We illustrate Algorithm 1 with Example 3.

Example 3 Consider the following SMT formula, expressed as a set of clauses, where T is assumed to be the theory of linear integer arithmetic:

$$\begin{aligned}\varphi = \; & \neg(x > 0) \vee a \vee b \\ & \neg a \vee \neg b \\ & \neg(x + 1 < 0) \vee a \\ & \neg b \vee \neg(y = 1)\end{aligned}$$

Then $\{x > 0, a, b, x + 1 < 0, y = 1\}$ is its set of atoms and

$$A^p = \{p_{(x>0)}, a, b, p_{(x+1<0)}, p_{(y=1)}\}$$

is the Booleanization of this set, where $p_{(x>0)}, p_{(x+1<0)}$ and $p_{(y=1)}$ are three fresh propositional variables corresponding to the arithmetic atoms $x > 0, x + 1 < 0$ and $y = 1$, respectively. The propositional abstraction of φ is then the following Boolean formula:

$$\begin{aligned}\varphi^p = \; & \neg p_{(x>0)} \vee a \vee b \\ & \neg a \vee \neg b \\ & \neg p_{(x+1<0)} \vee a \\ & \neg b \vee \neg(p_{(y=1)})\end{aligned}$$

It is not hard to see that φ^p is Bool-satisfiable. Suppose that the function pick_total_assignment(A^p,φ^p) returns us the following Boolean model of φ^p:

$$\mu^p = \{p_{(x>0)}, a, \neg b, p_{(x+1<0)}, \neg(p_{(y=1)})\}$$

Now we need to check T-satisfiability of B2T(μ^p). Since we are interested in checking the consistency of the current Boolean assignment with theory T, here we only need to take into account the literals corresponding to the theory, i.e., we have to check the T-satisfiability of $\{x > 0, x + 1 < 0, \neg(y = 1)\}$. Obviously this is not T-satisfiable, so we get a subset of T-inconsistent literals from the T-solver, e.g., $\pi = \{x > 0, x + 1 < 0\}$, and we extend φ^p with the learned clause, namely $\neg p_{(x>0)} \vee \neg p_{(x+1<0)}$. Then the search starts again.

In practice, the enumeration of Boolean models is carried out by means of efficient implementations of the DPLL algorithm [39], where the μ^p are partial assignments built incrementally. These systems inherit the spectacular progress in performance from SAT solvers in the last decade, achieved thanks to better implementation techniques and conceptual enhancements. Adaptations of SAT techniques to the SMT framework have been described in [33]. *Unit propagation* is used extensively to perform all the assignments which derive deterministically from the current μ^p. This allows the system to prune the search space and to backtrack as high as possible (backjumping). Another important improvement consists on checking the T-satisfiability of intermediate assignments, in order to anticipate possible prunings (*early pruning*). *Theory deduction* can be used to reduce the search space by explicitly returning truth values for unassigned literals, as well as constructing and learning implications. The deduction capability is a very important aspect of theory solvers, since getting short explanations (conflict sets) from the theory solver is essential in order to keep the learned lemmas as short as possible. Apart from saving memory space, shorter lemmas will allow for more pruning in general. Finally, in order to avoid getting stuck into hard portions of the search space, most systems *restart* from scratch in a controlled manner with the hope of exploring easier successful branches.

Although we use SMT solvers as black boxes, we conjecture that it is precisely this strong cooperation between the Boolean reasoning and the arithmetic reasoning what provides us with a robust performance in problems having a significant Boolean component. In Section 4 we provide some insights supporting this conjecture.

2.2.2 The Theories

The Satisfiability Modulo Theories Library (SMT-LIB) [8] has the goal to establish a library of benchmarks for SMT, as well as to establish a common standard for the specification of benchmarks and of background theories. The Satisfiability Modulo Theories Competition (SMT-COMP) is an associated yearly competition for SMT solvers. Among the theories considered in [8] we are interested in *Ints* (integer numbers) and *Reals* (real numbers). We consider the following logics:

- QF_LIA: quantifier-free *linear integer arithmetic*. Closed quantifier-free formulas with Boolean combinations of inequations between linear polynomials over integer variables, e.g. $(3x + 4y \geq 7) \rightarrow (z = 3)$ where x, y and z are integer variables.
- QF_IDL: quantifier-free *difference logic* over the integers. It is a fragment of the QF_LIA logic where arithmetic atoms are restricted to have the form $x - y < k$, where x and y are integer variables and k is an integer constant.
- QF_LRA: quantifier-free *linear real arithmetic*. Closed quantifier-free formulas with Boolean combinations of inequations between linear polynomials over real variables (similar to QF_LIA but with real variables).
- QF_NIA: quantifier-free *integer arithmetic* with no linearity restrictions, e.g. $(3xy > 2 + z^2) \vee (3xy = 9)$ where x, y and z are integer variables.

The expressivity of each of these logics has its corresponding computational price. Checking consistency of a set of IDL constraints has polynomial time complexity whereas checking consistency of a set of LIA constraints is NP-complete.

The non-linear case is in general (with no bounds on the domain of variables) undecidable.

In addition, we have also considered the previous logics combined with *uninterpreted functions* (e.g., QF_UFLIA). The theory of uninterpreted function symbols (a.k.a. the empty theory) is just the theory of equality with no additional equational axioms. As we show in Subsection 3.2.1, the possibility of using uninterpreted function symbols allows us to trivially codify array accesses, since $x = y \rightarrow f(x) = f(y)$ is an axiom of the theory of equality.

3 The Tool

This section is devoted to the description of our tool, `fzn2smt`. First of all we show the architecture of the tool, and afterwards describe its translation and optimization processes.

3.1 Architecture of the Tool

Our tool is diagrammatically represented in Fig. 1, through the process of compiling and solving FlatZinc instances. Shaded boxes (connected by dashed lines) denote inputs and outputs, rounded corner boxes denote actions and diamond boxes denote conditions.

The input of the compiler is a FlatZinc instance which we assume to come from the translation of a MiniZinc one. Hence we are dealing with "safe" FlatZinc instances, e.g., we don't care about out of bounds array accesses. We are also assuming that all global constraints have been reformulated into FlatZinc constraints with the default encoding provided by the MiniZinc distribution.

The input FlatZinc instance is translated into an SMT one (in the standard SMT-LIB format v1.2) and fed into an SMT solver. As a by-product, `fzn2smt` generates the corresponding SMT instance as an output file. 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 currently being Yices 2 Prototype).[2]

The FlatZinc language has three solving options: `solve satisfy`, `solve minimize` *obj* and `solve maximize` *obj*, where *obj* is either the name of a variable v or a subscripted array variable $v[i]$, where i is an integer literal. Since optimization is supported neither in the SMT-LIB language nor by most SMT solvers, we have naively implemented it by means of iterative calls successively restricting the domain of the variable to be optimized (as explained in detail in Subsection 3.2.3). Notice from the diagram of Fig. 1 that when, after restricting the domain, the SMT solver finds that the problem is not satisfiable anymore, the last previously saved (and hence optimal) solution is recovered. Moreover, since there is no standard output format currently supported by SMT solvers,[3] we need a specialized *recovery module* for each solver in order to translate its output to the FlatZinc output

[2] `http://yices.csl.sri.com/download-yices2.shtml`

[3] There are even solvers that only return `sat`, `unsat` or `unknown`. A proposal of a standard format for solutions has been recently proposed in the SMT-LIB Standard v2.0 [9].

format. Currently, `fzn2smt` can recover the output from Yices [20], Barcelogic [10], Z3 [19] and MathSat [15] SMT solvers.

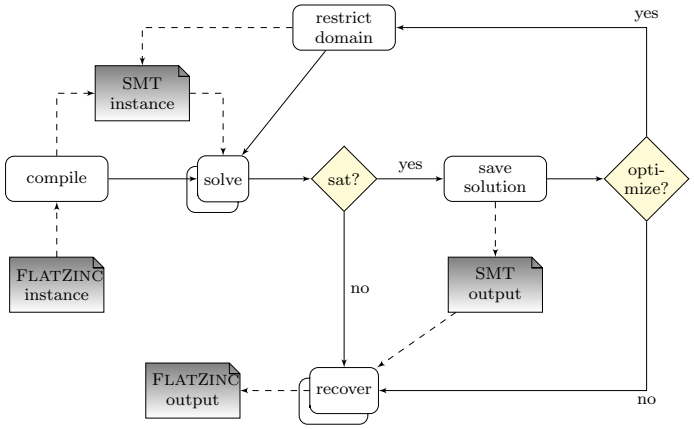

Fig. 1 The compiling and solving process of `fzn2smt`.

3.2 The Translation

Since we are translating FlatZinc instances into SMT, we have to keep in mind two important considerations: on the one hand we have a much richer language than plain SAT, thanks to the theories, and this will allow for more direct translations. On the other hand, in order to take advantage of the SMT solving mechanisms, the more logical structure the SMT formula has, the better. In particular, it is better to introduce clauses instead of expressing disjunctions arithmetically.

A FlatZinc file consists of

1. a list of constant and (finite domain) variable declarations,
2. a list of flat constraints, and
3. a solve goal.

Here we describe the translation of these three basic ingredients.

3.2.1 Constant and Variable Declarations

FlatZinc has two categories of data: constants (also called parameters) and variables (typically with an associated finite domain). Data can be of any of three scalar types: Booleans, integers and floats, or of any of two compound types: sets of integers and one-dimensional $(1..n)$-indexed arrays (multi-dimensional arrays are flattened to arrays of one dimension in the translation from MiniZinc to FlatZinc). Scalar type domains are usually specified by a range or a list of possible values. Our translation of FlatZinc constants and variables goes as follows:

– Scalar type constant names are always replaced by their corresponding value.

– Scalar type variable declarations are translated into their equivalent variable declaration in SMT, plus some constraints on the possible values of the SMT variable in order to fix the domain. For example, `var 0..14: x` is translated into the SMT declaration of the integer variable x plus the constraints $x \geq 0$, $x \leq 14$, whereas `var {1,3,7}: x` is translated into the SMT declaration of the integer variable x plus the constraint $x = 1 \vee x = 3 \vee x = 7$.
Although SMT solvers are able to deal with arbitrarily large integers (as well as with arbitrary precision real numbers), for unrestricted domain integer variables we assume the G12 FlatZinc interpreter default domain range of `-10000000..10000000`, i.e., we add the constraints $x \geq -10000000$, $x \leq 10000000$ for every unrestricted integer variable x. This way, the results obtained by our system are consistent with the ones obtained by other tools.
– The domain of a FlatZinc set of integers is specified either by a range or by a list of integers. For this reason, we simply use an occurrence representation by introducing a Boolean variable for every possible element, which indicates whether the element belongs to the set or not. This allows for a simple translation into SMT of most of the constraints involving sets (see Subsection 3.2.2 below).
However, in order to be able to translate the set cardinality constraint, which implies counting the number of elements in a set, a 0/1 partner variable is needed for each introduced Boolean variable. For example, given the declaration `var set of {2,5,6}:s`, we introduce three Boolean variables s_2, s_5 and s_6, three corresponding integer variables s_{i_2}, s_{i_5} and s_{i_6}, the constraints restricting the domain of the integer variables

$$0 \leq s_{i_2},\ s_{i_2} \leq 1$$
$$0 \leq s_{i_5},\ s_{i_5} \leq 1$$
$$0 \leq s_{i_6},\ s_{i_6} \leq 1$$

and the constraints linking the Boolean variables with their integer counterpart

$$s_2 \rightarrow s_{i_2} = 1,\ \neg s_2 \rightarrow s_{i_2} = 0$$
$$s_5 \rightarrow s_{i_5} = 1,\ \neg s_5 \rightarrow s_{i_5} = 0$$
$$s_6 \rightarrow s_{i_6} = 1,\ \neg s_6 \rightarrow s_{i_6} = 0.$$

Hence, the number of SMT variables increases linearly with the size of the domain of the set. Note also that all introduced clauses are either unary or binary, and hence facilitate Boolean unit propagation.
For the case of constant sets no variables are introduced at all. Instead, the element values of the constant set are directly used in the operations involving it, in order to obtain a simpler encoding.
– For the translation of arrays, we provide two options (which can be chosen by a command line option):[4]
  – Using uninterpreted functions: each array is translated into an uninterpreted function of the same name. For example `array[1..3] of var 1..5:a` is translated into $a : int \mapsto int$. The domain of the elements of the array

[4] Due to our encoding of the operations on sets, arrays of sets are always decomposed into a number of sets.

is constrained as in the scalar case, that is, $1 \leq a(1)$, $a(1) \leq 5$, $1 \leq a(2)$, $a(2) \leq 5$, $1 \leq a(3)$, $a(3) \leq 5$.
- Decomposing the array into as many base type variables as array elements. For example, the previous array `a` would be decomposed into three integer variables a_1, a_2 and a_3, with the domain constrained analogously to before.

In the case of constant arrays, equality is used (instead of two inequalities restricting the domain) to state the value of each element. If, moreover, an access to a constant array uses a constant index, we can simply replace that access with its corresponding value. And, if this is the case for all the accesses to an array, then there is no need for introducing an uninterpreted function or a set of base type variables to represent the array.
Regarding the two possible encodings, the use of uninterpreted functions seems to be more natural, and allows for more compact encodings of array constraints. For example, to express that some element of the previous array a is equal to 1, we simply write

$$1 \leq i, i \leq 3$$
$$a(i) = 1$$

where i is an integer variable, whereas in the decomposition approach the same statement should be expressed as

$$1 \leq i, i \leq 3$$
$$i = 1 \rightarrow a_1 = 1$$
$$i = 2 \rightarrow a_2 = 1$$
$$i = 3 \rightarrow a_3 = 1.$$

On the other hand, we have ruled out using the SMT theory of arrays. This theory involves *read* and *write* operations and, hence, is intended to be used for modelling state change of imperative programs with arrays. But, since it makes no sense thinking of *write* operations on arrays in the setting of CP, it suffices to translate every expression of the form $read(a, i)$ into $a(i)$, where a is an uninterpreted function. Moreover, deciding satisfiability of sets of atomic constraints involving uninterpreted functions is far cheaper than using the arrays theory.
However, the uninterpreted functions approach still has the drawback of using more than one theory, namely, uninterpreted functions (UF) and linear integer arithmetic (LIA), and suffers from a non-negligible computational overhead due to theory combination. In Section 4 a performance comparison of the two possible encodings for arrays is given.

3.2.2 Constraints

The second and main block of a FlatZinc file is the set of constraints that a solution must satisfy. The arguments of these flat constraints can be literal values, constant or variable names, or subscripted array constants or variables $v[i]$ where i in an integer literal. A literal value can be either a value of scalar type, an explicit set (e.g., `{2,3,5}`) or an explicit array `[`$y_1, \ldots, y_k$`]`, where each array element y_i is either a non-array literal, the name of a non-array constant or variable, or

a subscripted array constant or variable $v[i]$, where i in an integer literal (e.g., `[x,a[1],3]`).

We perform the following translation of constraint arguments prior to translating the constraints. Constant names are replaced by their value. Scalar type variables are replaced by their SMT counterpart. Finally, array accesses are translated depending on the chosen array treatment (see the previous subsection): when using uninterpreted functions, an array access $v[i]$ is translated into $v(i)$, where v is an uninterpreted function; when decomposing the array into base type variables, $v[i]$ is translated into v_i (the corresponding variable for that position of the array). We remark that, in all array accesses $v[i]$, i can be assumed to be an integer literal, i.e., i cannot be a variable, since all variable subscripted array expressions are replaced by `array_element` constraints during the translation from MiniZinc to FlatZinc. Moreover, we don't need to perform array bounds checks, because this is already done by the MiniZinc to FlatZinc compiler.

In the following we describe the translation of the constraints, that we have categorized into *Boolean constraints*, *Integer constraints*, *Float constraints*, *Array constraints* and *Set constraints*.

- *Boolean constraints* are built with the common binary Boolean operators (*and*, *or*, *implies*, ...) and the relational operators ($<$, $\leq$, $=$, ...) over Booleans. All of them have their counterpart in the SMT-LIB language, and hence have a direct translation.
  There is also the `bool2int(a,n)` constraint, which maps a Boolean variable into a 0/1 integer. We translate it into $(a \rightarrow n = 1) \wedge (\neg a \rightarrow n = 0)$.
- *Integer constraints* are built with the common relational constraints over integer expressions (hence they are straightforwardly translated into SMT). They also include some named constraints. Here we give the translation of some representative ones.
  The constraint
  $$\texttt{int\_lin\_eq}([\texttt{c}_1, \ldots, \texttt{c}_\texttt{n}], [\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}], \texttt{r})$$
  where $\texttt{c}_1, \ldots, \texttt{c}_\texttt{n}$ are integer constants and $\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}$ are integer variables or constants, means, and is translated as
  $$\sum_{i \in 1..n} c_i v_i = r.$$
  The minimum constraint `int_min(a,b,c)`, meaning $min(a, b) = c$, is translated as
  $$(a > b \rightarrow c = b) \wedge (a \leq b \rightarrow c = a).$$
  The absolute value constraint `int_abs(a,b)`, meaning $|a| = b$, is translated as
  $$(a = b \vee -a = b) \wedge b \geq 0.$$
  The constraint `int_times(a,b,c)`, that states $a \cdot b = c$, can be translated into a set of linear arithmetic constraints under certain circumstances: if either a or b are (uninstantiated) finite domain variables, we linearize this constraint by conditionally instantiating the variable with the smallest domain, e.g.,
  $$\bigwedge_{i \in \mathcal{D}om(a)} (i = a \rightarrow i \cdot b = c).$$

In fact, we do it better (i.e., we do not necessarily expand the expression for all the values of $\mathcal{D}om(a)$) by considering the domain bounds of b and c, and narrowing accordingly the domain of a.
Since it is better to use the simplest logic at hand, we use linear integer arithmetic for the translation if possible, and non-linear integer arithmetic otherwise. Hence, only in the case that a and b are unrestricted domain variables we translate the previous constraint as $a \cdot b = c$ and label the SMT instance to require QF_NIA (*non-linear integer arithmetic logic*).

- *Float constraints* are essentially the same as the integer ones, but involving float data. Hence, the translation goes in the same way as for the integers, except that the inferred logic is QF_LRA (*linear real arithmetic*).
- *Array constraints.* The main constraint dealing with arrays is `element`, which restricts an element of the array to be equal to some variable. As an example, `array_var_int_element(i, a, e)` states $i \in 1..n \wedge a[i] = e$, where n is the size of a. The translation varies depending on the representation chosen for arrays (see the previous subsection):
  - In the uninterpreted functions approach, the translation is

$$1 \leq i \wedge i \leq n \wedge a(i) = e,$$

    where a is the uninterpreted function symbol representing the array `a`.
  - In the decomposition approach, the translation is

$$1 \leq i \wedge i \leq n \wedge \left(\bigwedge_{j \in 1..n} i = j \rightarrow a_j = e \right).$$

  Constraints such as `array_bool_and`, `array_bool_or` or `bool_clause`, dealing with arrays of Booleans, are straightforwardly translated into SMT.
- *Set constraints.* These are the usual constraints over sets. We give the translation of some of them.
  The constraint `set_card(s,k)`, stating $|s| = k$, is translated by using the 0/1 variables introduced for each element (see the previous subsection) as

$$\sum_{j \in \mathcal{D}om(s)} s_{i_j} = k.$$

  The constraint `set_in(e,s)`, stating $e \in s$, is translated depending on whether e is instantiated or not. If e is instantiated then `set_in(e,s)` is translated as s_e if e is in the domain of s (recall that we are introducing a Boolean variable s_e for each element e in the domain of s), and as *false* otherwise. If `e` is not instantiated, then we translate the constraint as

$$\bigvee_{j \in \mathcal{D}om(s)} (e = j) \wedge s_j.$$

  For constraints involving more than one set, one of the difficulties in their translation is that the involved sets can have distinct domains. For example,

the constraint `set_eq(a,b)`, stating $a = b$, is translated as

$$\left(\bigwedge_{j \in \mathcal{D}om(a) \cap \mathcal{D}om(b)} a_j = b_j\right) \wedge \left(\bigwedge_{j \in \mathcal{D}om(a) \setminus \mathcal{D}om(b)} \neg a_j\right) \wedge \left(\bigwedge_{j \in \mathcal{D}om(b) \setminus \mathcal{D}om(a)} \neg b_j\right).$$

And the constraint `set_diff(a,b,c)`, which states $a \setminus b = c$, is translated as

$$\left(\bigwedge_{j \in (\mathcal{D}om(a) \setminus \mathcal{D}om(b)) \cap \mathcal{D}om(c)} a_j = c_j\right) \wedge \left(\bigwedge_{j \in (\mathcal{D}om(a) \setminus \mathcal{D}om(b)) \setminus \mathcal{D}om(c)} \neg a_j\right) \wedge \left(\bigwedge_{j \in \mathcal{D}om(c) \setminus \mathcal{D}om(a)} \neg c_j\right) \wedge \left(\bigwedge_{j \in \mathcal{D}om(a) \cap \mathcal{D}om(b) \cap \mathcal{D}om(c)} \neg c_j\right) \wedge \left(\bigwedge_{j \in (\mathcal{D}om(a) \cap \mathcal{D}om(b)) \setminus \mathcal{D}om(c)} a_j \rightarrow b_j\right).$$

Although the translation of the set constraints seem to be convoluted, note that we are mainly introducing unit and binary clauses. We remark that when the sets are already instantiated at compilation time, some simplifications are actually made. Note also that the size of the SMT formula increases linearly in the size of the domains of the sets.

Finally, let us mention that almost all FlatZinc constraints have a reified counterpart. For example, in addition to the constraint `int_le(a,b)`, stating $a \leq b$, there is a constraint `int_le_reif(a,b,r)`, stating $a \leq b \leftrightarrow r$, where a and b are integer variables and r is a Boolean variable. In all these cases, given the translation of a constraint, the translation of its reified version into SMT is direct.

3.2.3 Solve Goal

A FlatZinc file must end with a solve goal, which can be of one of the following forms: `solve satisfy`, for checking satisfiability and providing a solution if possible, or `solve minimize` *obj* or `solve maximize` *obj*, for looking for a solution that minimizes or maximizes, respectively, the value of *obj*, where *obj* is either a variable v or a subscripted array variable $v[i]$, where i is an integer literal. Although search annotations can be used in the solve goal, they are currently ignored in our tool.

When the `satisfy` option is used, we just need to feed the selected SMT solver with the SMT file resulting from the translation explained in the previous subsections (see Example 4 to see a complete SMT-LIB instance generated by `fzn2smt`). Thereafter the recovery module will translate the output of the SMT solver to the FlatZinc format (as explained in Subsection 3.1).

Since most SMT solvers do not provide optimization facilities,[5] we have implemented an ad hoc search procedure in order to deal with the `minimize` and `maximize` options. This procedure successively calls the SMT solver with different problem instances, by restricting the domain of the variable to be optimized with the addition of constraints. We have implemented three possible bounding strategies: linear, dichotomic and hybrid. The linear bounding strategy approaches the optimum from the satisfiable side,[6] while the dichotomic strategy simply consists of binary search optimization. Finally, the hybrid strategy makes a preliminary approach to the optimum by means of binary search and, when a (user definable) threshold on the possible domain of the variable is reached, it turns into the linear approach, again from the satisfiable side. Both the bounding strategy and the threshold for the hybrid case can be specified by the user from the command line.

When possible, it would be interesting to keep the learnt clauses from one iteration of the SMT solver to the next (for example in the linear strategy, where we are approaching to the optimum from the satisfiable side). This is not possible with the basic version of `fzn2smt`, which is designed to communicate with an SMT solver using external files, as shown in Figure 1. However, we have tested the approach of keeping the learnt clauses by using the Yices API without obtaining significantly better results.

Example 4 Continuing Example 2, here follows the SMT-LIB instance produced by `fzn2smt`.

```
(benchmark jobshopp.fzn.smt
  :source { Generated by fzn2smt }
  :logic QF_IDL
  :extrapreds ((BOOL____4) (BOOL____2) (BOOL____1) (BOOL____5))
  :extrafuns ((s_1_ Int) (s_2_ Int) (s_3_ Int) (s_4_ Int) (end Int))
  :formula (and
    (>= end 5)
    (<= end 14)
    (>= s_1_ 0)
    (<= s_1_ 14)
    (>= s_2_ 0)
    (<= s_2_ 14)
    (>= s_3_ 0)
    (<= s_3_ 14)
    (>= s_4_ 0)
    (<= s_4_ 14)
    (= (or  BOOL____1 BOOL____2) true)
    (= (or  BOOL____4 BOOL____5) true)
    (<= (+ (~ end) s_2_) (~ 5))
    (<= (+ (~ end) s_4_) (~ 4))
    (<= (+ s_1_ (~ s_2_)) (~ 2))
```

[5] There are however some solvers, such as Yices and Z3, that already provide Max-SMT facilities. On the other hand, the problem of optimization modulo theories has been recently addressed in [17], by introducing a theory of costs.

[6] This allows us to eventually jump several values when we find a new solution. On the contrary, approaching from the unsatisfiable side is only possible by modifying the value of the variable to optimize in one unit at each step.

```
    (<= (+ s_3_ (~ s_4_)) (~ 3))
    (= (<= (+ s_1_ (~ s_3_)) (~ 2)) BOOL____1)
    (= (<= (+ (~ s_1_) s_3_) (~ 3)) BOOL____2)
    (= (<= (+ s_2_ (~ s_4_)) (~ 5)) BOOL____4)
    (= (<= (+ (~ s_2_) s_4_) (~ 4)) BOOL____5)
  )
)
```

4 Benchmarking

In this section we compare the performance of `fzn2smt` and that of several existing FlatZinc solvers on FlatZinc instances, and provide some possible explanations about the `fzn2smt` behaviour. We first compare several SMT solvers within `fzn2smt` and, then, use the one with the best results to compare against other existing FlatZinc solvers.

We perform the comparisons on the benchmarks of the three MiniZinc challenge competitions (2008, 2009 and 2010), consisting of a total of 294 instances from 32 problems. These benchmarks consist of a mixture of puzzles, planning, scheduling and graph problems. Half of the problems are optimization problems, whilst the other half are satisfiability ones.

We present several tables that, for each solver and problem, report the accumulated time for the solved instances and the number of solved instances (within parenthesis). The times are the sum of the translation time, 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 executed on an Intel Core i5 CPU at 2.66 GHz, with 6GB of RAM, running 64-bit openSUSE 11.2 (kernel 2.6.31), with a time limit of 15 minutes per instance (the same as in the competition).

4.1 `fzn2smt` with State-of-the-Art SMT Solvers

Here we compare the performance of several SMT solvers which are SMT-LIB 1.2 compliant, working in cooperation with `fzn2smt` v2.0.1, on the MiniZinc challenge benchmarks. We have selected the solvers that historically have had good performance in the QF_LIA division of the annual SMT competition. These are Barcelogic 1.3 [10], MathSAT 5 (successor of MathSAT 4 [15]; still work in progress), Yices 1.0.28 [20], Yices 2 Prototype (still work in progress), and Z3.2 [19]. Linux binaries of most of these solvers can be downloaded from `http://www.smtcomp.org`.

In the executions of this subsection we have used the default options for `fzn2smt`: array expansion (see Subsection 3.2.1) and binary search for optimization (see Subsection 3.2.3).

In Table 1 we can observe that Yices 1.0.28 and Yices 2 are able to solve, respectively, 213 and 212 (out of 294) instances. We consider performance of Yices 2 the best of those considered because it had the best performance on 19 of the problems, far more than any of the other solvers.

Table 1 Performance of state-of-the-art SMT solvers in cooperation with `fzn2smt` on the MINIZINC challenge benchmarks. Type 's' stands for satisfaction and 'o' for optimization. # stands for the number of instances.

Problem	Type	#	Barcelogic 1.3	MathSAT 5	Yices 1.0.28	Yices 2 proto	Z3.2
debruijn-binary	s	11	0.60 (1)	0.56 (1)	**0.47 (1)**	0.50 (1)	0.57 (1)
nmseq	s	10	0.00 (0)	568.35 (2)	778. 07 (5)	118.74 (2)	**173.84 (5)**
pentominoes	s	7	0.00 (0)	291.60 (1)	**50.47 (2)**	168.47 (2)	314.48 (1)
quasigroup7	s	10	34.16 (2)	191.83 (5)	54.94 (5)	**20.03 (5)**	691.32 (4)
radiation	o	9	342.34 (1)	2202.92 (9)	**373.24 (9)**	1047.03 (9)	2473.27 (9)
rcpsp	o	10	0.00 (0)	315.46 (2)	920.08 (8)	**993.74 (9)**	1791.79 (8)
search-stress	s	3	0.84 (2)	1.05 (2)	0.86 (2)	**0.74 (2)**	0.84 (2)
shortest-path	o	10	**109.35 (9)**	484.73 (8)	658.58 (9)	1419.67 (7)	790.54 (8)
slow-convergence	s	10	405.25 (7)	426.79 (7)	269.66 (7)	**247.06 (7)**	291.99 (7)
trucking	o	10	254.11 (5)	31.70 (4)	**9. 50 (5)**	47.77 (5)	1084.97 (4)
black-hole	s	10	511.13 (1)	29.24 (1)	**3857.51 (9)**	892.46 (8)	765.09 (1)
fillomino	s	10	93.87 (10)	30.28 (10)	20.48 (10)	**19.99 (10)**	21.13 (10)
nonogram	s	10	0.00 (0)	0.00 (0)	**1656.54 (10)**	1546.56 (7)	0.00 (0)
open-stacks	o	10	1772.39 (5)	0.00 (0)	702.09 (6)	**707.25 (7)**	776.61 (6)
p1f	o	10	875.54 (8)	**86.90 (9)**	167.84 (9)	126.01 (9)	184.22 (9)
prop-stress	s	10	315.80 (7)	330.31 (7)	**266.11 (7)**	274.07 (7)	289.23 (7)
rect-packing	s	10	559.50 (5)	679.62 (10)	**104.82 (10)**	106.66 (10)	122.28 (10)
roster-model	o	10	98.41 (10)	51.03 (10)	53.89 (10)	**50.38 (10)**	56.04 (10)
search-stress2	s	10	23.19 (10)	14.63 (10)	9.43 (10)	**7.90 (10)**	10.55 (10)
still-life	o	4	30.64 (3)	132.51 (4)	128.71 (4)	**62.18 (4)**	173.82 (4)
vrp	o	10	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)
costas-array	s	5	0.00 (0)	0.00 (0)	675.23 (1)	**664.13 (2)**	628.83 (1)
depot-placement	o	15	2480.74 (5)	2496.69 (10)	613.53 (15)	**295.78 (15)**	2073.93 (15)
filter	o	10	24.02 (6)	38.01 (6)	24.28 (6)	**17.88 (6)**	22.42 (6)
ghoulomb	o	10	0.00 (0)	0.00 (0)	75.87 (1)	**2545.02 (6)**	508.19 (2)
gridColoring	o	5	4.82 (2)	202.18 (3)	857.74 (3)	**38.15 (3)**	51.94 (3)
rcpsp-max	o	10	85.04 (1)	238.34 (2)	533.87 (4)	475.14 (4)	**383.73 (4)**
solbat	s	15	0.00 (0)	1721.73 (15)	341.04 (15)	**141.48 (15)**	1339.75 (15)
sugiyama2	o	5	79.38 (5)	21.31 (5)	10.26 (5)	**8.64 (5)**	9.56 (5)
wwtp-random	s	5	61.08 (5)	29.72 (5)	17.71 (5)	**15.21 (5)**	16.56 (5)
wwtp-real	s	5	107.57 (5)	39.77 (5)	17.97 (5)	**14.47 (5)**	17.09 (5)
bacp	o	15	1753.26 (14)	232.22 (15)	75.16 (15)	**64.17 (15)**	97.51 (15)
Total		294	10023 (129)	10889 (168)	**13325 (213)**	12137 (212)	15162 (192)

4.1.1 Array encodings

In Table 2 we compare the performance of using array decomposition versus uninterpreted functions as array encodings. We only give the results for Yices 2 proto, which is the solver with the best performance in the executions of Table 1 (where array decomposition was used as default option).

As shown in Table 2, the decomposition approach clearly outperforms the uninterpreted functions approach on FLATZINC instances from the MINIZINC distribution. We have also tested other SMT solvers than Yices, obtaining similar results. This apparently strange behaviour is better understood when looking at how SMT solvers deal with uninterpreted functions and, in particular, how this behaves on the instances generated by our tool. Hence, first of all, let us see some possible treatments to uninterpreted functions in the context of SMT. The reader can refer to [14] for a deeper discussion on this issue.

When dealing with two or more theories, a standard approach is to handle the integration of the different theories by performing some sort of search on the equalities between their shared (or *interface*) variables. First of all, formulas are purified by replacing terms with fresh variables, so that each literal only contains symbols belonging to one theory. For example,

$$a(1) = x + 2$$

Table 2 Performance with Yices 2 using array decomposition vs uninterpreted functions (UF).

Problem	Type	#	Decomposition		UF	
debruijn-binary	s	11	**0.50**	**(1)**	0.74	(1)
nmseq	s	10	118.74	(2)	**83.51**	**(2)**
pentominoes	s	7	**168.47**	**(2)**	197.22	(1)
quasigroup7	s	10	**20.03**	**(5)**	336.59	(5)
radiation	o	9	**1047.03**	**(9)**	2360.85	(9)
rcpsp	o	10	**993.74**	**(9)**	695.84	(8)
search-stress	s	3	**0.74**	**(2)**	0.84	(2)
shortest-path	o	10	**1419.67**	**(7)**	1068.03	(4)
slow-convergence	s	10	**247.06**	**(7)**	522.42	(3)
trucking	o	10	47.77	(5)	**39.67**	**(5)**
black-hole	s	10	**892.46**	**(8)**	0.00	(0)
fillomino	s	10	19.99	(10)	**19.21**	**(10)**
nonogram	s	10	**1546.56**	**(7)**	0.00	(0)
open-stacks	o	10	**707.25**	**(7)**	1729.37	(5)
p1f	o	10	**126.01**	**(9)**	25.49	(8)
prop-stress	s	10	274.07	(7)	**262.95**	**(7)**
rect-packing	s	10	**106.66**	**(10)**	112.10	(10)
roster-model	o	10	**50.38**	**(10)**	51.01	(10)
search-stress2	s	10	**7.90**	**(10)**	9.74	(10)
still-life	o	4	**62.18**	**(4)**	97.39	(4)
vrp	o	10	0.00	(0)	0.00	(0)
costasArray	s	5	**664.13**	**(2)**	151.60	(1)
depot-placement	o	15	**295.78**	**(15)**	2651.71	(10)
filter	o	10	**17.88**	**(6)**	23.12	(6)
ghoulomb	o	10	**2545.02**	**(6)**	707.74	(2)
gridColoring	o	5	**38.15**	**(3)**	335.10	(3)
rcpsp-max	o	10	**475.14**	**(4)**	498.89	(4)
solbat	s	15	**141.48**	**(15)**	567.69	(15)
sugiyama2	o	5	**8.64**	**(5)**	9.04	(5)
wwtp-random	s	5	**15.21**	**(5)**	34.67	(5)
wwtp-real	s	5	**14.47**	**(5)**	28.82	(5)
bacp	o	15	**64.17**	**(15)**	77.85	(15)
Total		294	**12137**	**(212)**	12699	(175)

is translated into

$$\begin{aligned} a(v_1) &= v_2 \\ v_1 &= 1 \\ v_2 &= x + 2 \end{aligned}$$

where the first literal belongs to UF, and the rest belong to LIA. The variables v_1, v_2 are then called *interface variables*, as they appear in literals belonging to different theories. An *interface equality* is an equality between two interface variables. All theory combination schemata, e.g., Nelson-Oppen [24], Shostak [34], or Delayed Theory Combination (DTC) [13], rely to some point on checking equality between interface variables, in order to ensure mutual consistency between theories. This may imply to assign a truth value to up to all the interface equalities. Since the number of interface equalities is given by $|\mathcal{V}| \cdot (|\mathcal{V}| - 1)/2$, where $|\mathcal{V}|$ is the number of interface variables, the search space may be enlarged in a quadratic factor in the number of interface variables.

In the case of combining UF with another theory T, an alternative approach is to eliminate the uninterpreted function symbols by means of Ackermann's reduction [5], and then solve the resulting SMT problem with only theory T. In Ackermann's reduction, each application $f(a)$ is replaced by a variable f_a, and for each pair of applications $f(a)$, $f(b)$ the formula $a = b \rightarrow f_a = f_b$ is added, i.e., the single theory axiom $x = y \rightarrow f(x) = f(y)$ of the UF theory becomes instantiated as necessary. This is the approach taken by most state-of-the-art SMT solvers. However, this has the same disadvantage as theory combination in that the number of additional literals is quadratic in the size of the input and, in fact, as shown in [14], there is no clear winner between DTC and Ackermannization.

It is worth noting that current SMT solvers have been designed mainly to deal with verification problems, where there are few parameters and almost all variable values are undefined. In such problems, uninterpreted functions are typically used to abstract pieces of code and, hence, their arguments are variables (or expressions using variables). Moreover, the number of such abstractions is limited. This makes Ackermannization feasible in practice. On the contrary, in the instances we are considering, we have the opposite situation: a lot of parameters and a few decision variables. In particular, most arrays are parameters containing data. For example, in a scheduling problem, a FlatZinc array containing durations of tasks, such as

```
array[1..100] of int: d = [2,5,...,4];
```

could be expressed using an SMT uninterpreted function as follows:

$$\begin{gathered} d(1) = 2 \\ d(2) = 5 \\ \dots \\ d(100) = 4. \end{gathered}$$

Similarly, for an undefined array containing, e.g., starting times, such as

```
array[1..100] of var 0..3600: s;
```

we could use an uninterpreted function, and state its domain as follows:

$$\begin{gathered} 0 \leq s(1),\ s(1) \leq 3600 \\ 0 \leq s(2),\ s(2) \leq 3600 \\ \dots \\ 0 \leq s(100),\ s(100) \leq 3600. \end{gathered}$$

In any case, lots of distinct uninterpreted function applications appear, and Ackermannization results in a quadratic number of formulas like $1 = 2 \rightarrow f_1 = f_2$, which are trivially true since the antecedent is false. Although difficult to determine because we are using each SMT solver as a black box, we conjecture that this is not checked in the Ackermannization process since, as said before, uninterpreted functions are expected to have variables in the arguments.

Finally, although the decomposition approach exhibits better performance than the uninterpreted functions approach, we have decided to maintain both options in our system. The main reason is that the uninterpreted functions approach allows for more compact and natural representations and, hopefully, can lead to better results in the future if Ackermannization is adapted accordingly.

4.1.2 Bounding Strategy

Here we test the performance of Yices 2 with the different bounding strategies described in Subsection 3.2.3 for optimization problems. For the hybrid strategy we have used the default threshold of 10 units for switching from the binary to the linear approximation strategy. Experiments with a larger threshold have not yielded better results.

Table 3 Performance with Yices 2 using different optimization search strategies.

Problem	#	Binary		Hybrid		Linear	
radiation	9	**1047.03**	**(9)**	1304.59	(9)	2008.77	(9)
rcpsp	10	**993.74**	**(9)**	710.77	(8)	1180.11	(5)
shortest-path	10	1419.67	(7)	1381.92	(7)	**1034.22**	**(8)**
trucking	10	47.77	(5)	41.86	(5)	**34.66**	**(5)**
open-stacks	10	707.25	(7)	**650.27**	**(7)**	691.46	(7)
p1f	10	126.01	(9)	**125.44**	**(9)**	188.14	(9)
roster-model	10	50.38	(10)	50.29	(10)	**50.16**	**(10)**
still-life	4	**62.18**	**(4)**	118.54	(4)	119.39	(4)
vrp	10	0.00	(0)	0.00	(0)	0.00	(0)
depot-placement	15	295.78	(15)	**248.19**	**(15)**	263.61	(15)
filter	10	17.88	(6)	**17.37**	**(6)**	18.34	(6)
ghoulomb	10	**2545.02**	**(6)**	2825.16	(6)	1255. 42	(3)
gridColoring	5	38.15	(3)	**17.43**	**(3)**	17.81	(3)
rcpsp-max	10	475.14	(4)	**460.08**	**(4)**	1035.02	(4)
sugiyama2	5	8.64	(5)	**8.37**	**(5)**	9.25	(5)
bacp	15	64.17	(15)	**58.03**	**(15)**	64.98	(15)
Total	153	**7898**	**(113)**	8018	(113)	7971	(108)

Table 3 shows that the binary and hybrid strategies perform better than the linear one in general. Both the binary and hybrid strategies are able to solve the same number of instances, but the first one spends less time globally. Nevertheless, the hybrid strategy is faster than the binary in most of the problems. And, curiously, the linear strategy is better in three problems.

We want to remark that the linear strategy approaches the optimum from the satisfiable side. We also tried the linear strategy approaching from the unsatisfiable side but the results where a bit worse globally. This is probably due to the fact that this last strategy can only make approximation steps of size one whilst the former can make bigger steps when a solution is found. Moreover, the formula resulting from the translation of many MiniZinc benchmarks has very simple Boolean structure (the formula is often trivially satisfiable at the Boolean level), and hence it is likely that the SMT solver cannot substantially profit from conflict-driven lemma learning on unsatisfiable instances. In fact, there exist unsatisfiable instances that result in a few or no conflicts at all, and most of the work is hence done by the theory solver.

4.2 FlatZinc Solvers

In this section we compare the performance of `fzn2smt` (using Yices 2) and the following available FlatZinc solvers: Gecode (winner of all MiniZinc challenges),

Table 4 Performance comparison between `fzn2smt` and some available FLATZINC solvers.

n	Problem	Type	#	Gecode		FZNTini		G12		G12 lazy_fd		`fzn2smt`	
1	debruijn-binary	s	11	**4.46**	**(6)**	0.06	(1)	31.30	(6)	0.00	(0)	0.50	(1)
2	nmseq	s	10	**535.62**	**(8)**	2.64	(1)	927.42	(7)	0.00	(0)	118.74	(2)
3	pentominoes	s	7	**601.82**	**(7)**	89.68	(1)	848.17	(4)	466.57	(5)	168.47	(2)
4	quasigroup7	s	10	**278.73**	**(6)**	773.28	(4)	1.72	(5)	2.85	(5)	20.30	(5)
5	radiation	o	9	1112.14	(9)	4260.37	(7)	1302.79	(9)	**3.60**	**(9)**	1047.03	(9)
6	rcpsp	o	10	12.07	(5)	0.00	(0)	97.27	(5)	82.60	(8)	**993.74**	**(9)**
7	search-stress	s	3	11.30	(2)	391.71	(3)	14.66	(2)	**0.40**	**(3)**	0.74	(2)
8	shortest-path	o	10	442.49	(10)	0.00	(0)	4.27	(4)	**127.77**	**(10)**	1419.67	(7)
9	slow-convergence	s	10	**8.41**	**(10)**	62.62	(4)	95.42	(10)	154.83	(10)	247.06	(7)
10	trucking	o	10	**1.01**	**(5)**	593.23	(4)	4.12	(5)	159.84	(5)	47.77	(5)
11	black-hole	s	10	69.68	(7)	0.00	(0)	2423.48	(6)	97.69	(7)	**892.46**	**(8)**
12	fillomino	s	10	118.62	(10)	4.36	(10)	332.56	(10)	**2.59**	**(10)**	19.99	(10)
13	nonogram	s	10	1353.63	(8)	48.13	(7)	336.93	(2)	**1533.07**	**(9)**	1546.56	(7)
14	open-stacks	o	10	**169.74**	**(8)**	1325.98	(4)	299.55	(8)	0.00	(0)	707.25	(7)
15	p1f	o	10	2.57	(8)	315.65	(9)	2.40	(8)	0.00	(0)	**126.01**	**(9)**
16	prop-stress	s	10	600.80	(4)	223.52	(2)	883.08	(3)	**221.95**	**(9)**	274.07	(7)
17	rect-packing	s	10	134.36	(6)	569.57	(3)	339.80	(6)	165.58	(6)	**106.66**	**(10)**
18	roster-model	o	10	**1.04**	**(10)**	0.00	(0)	4.46	(10)	18.80	(7)	50.38	(10)
19	search-stress2	s	10	296.16	(9)	9.10	(10)	381.82	(8)	**0.08**	**(10)**	7.90	(10)
20	still-life	o	4	1.01	(3)	35.50	(3)	2.56	(3)	18.55	(3)	**62.18**	**(4)**
21	vrp	o	10	0.00	(0)	0.00	(0)	0.00	(0)	0.00	(0)	0.00	(0)
22	costas-array	s	5	**943.75**	**(4)**	405.28	(2)	423.09	(3)	145.54	(2)	664.13	(2)
23	depot-placement	o	15	1205.75	(12)	1820.57	(8)	522.81	(8)	613.51	(12)	**295.78**	**(15)**
24	filter	o	10	30.95	(1)	62.16	(7)	278.72	(1)	**2.65**	**(7)**	17.88	(6)
25	ghoulomb	o	10	0.00	(0)	0.00	(0)	246.67	(1)	**2512.92**	**(8)**	2545.02	(6)
26	gridColoring	o	5	0.48	(1)	152.71	(3)	0.51	(1)	0.10	(1)	**38.15**	**(3)**
27	rcpsp-max	o	10	42.11	(2)	0.00	(0)	30.00	(1)	596.41	(4)	**475.14**	**(4)**
28	solbat	s	15	746.31	(10)	1300.54	(14)	1540.71	(10)	311.92	(11)	**141.48**	**(15)**
29	sugiyama2	o	5	308.31	(5)	37.00	(5)	510.72	(5)	520.88	(5)	**8.64**	**(5)**
30	wwtp-random	s	5	0.03	(1)	322.21	(3)	2.79	(2)	0.00	(0)	**15.21**	**(5)**
31	wwtp-real	s	5	0.08	(3)	1239.45	(4)	0.31	(3)	71.83	(4)	**14.47**	**(5)**
32	bacp	o	15	847.78	(10)	1170.15	(5)	976.35	(10)	**28.54**	**(15)**	64.17	(15)
	Total		294	9881	(190)	15215	(124)	12866	(166)	7859	(185)	**12137**	**(212)**

G12 and G12 lazy_fd (the solvers distributed with MINIZINC) and FZNTini (a SAT based solver).

Let us remark that `fzn2smt` with Yices 2 obtained (*ex aequo* with Gecode) the golden medal in the *par* division and the silver medal in the *free* division of the MINIZINC challenge 2010, and the silver medal in the same divisions of the MINIZINC challenge 2011. It is also fair to notice that the solver with the best performance in the MINIZINC challenges 2010 and 2011 (in all categories) was Chuffed, implemented by the MINIZINC team and not eligible for prizes.[7]

Table 4 shows the results of this comparison without using solver specific global constraints, which means that global constraints are decomposed into conjunctions of simpler constraints. However, search strategy annotations are enabled in all experiments and, while `fzn2smt` ignores them, the other systems can profit from these annotations.

We can observe that `fzn2smt` is the solver which is able to solve the largest number of instances, closely followed by G12 lazy_fd and Gecode. Looking at the problems separately, `fzn2smt` offers better performance in 12 cases, followed by G12 lazy_fd in 10 cases and Gecode in 9 cases.

We remark that G12 lazy_fd does not support instances with unbounded integer variables: the ones of debruijn-binary, nmseq, open-stacks, p1f, wwtp-random. We have tried to solve these instances by bounding those variables with the MINIZINC

[7] See `http://www.g12.csse.unimelb.edu.au/minizinc/challenge2011/results2011.html` for details.

standard default limits, i.e., by setting `var -10000000..10000000 : x;` for every unrestricted integer variable x (similarly as we have done for `fzn2smt`), but G12 lazy_fd runs out of memory. This is probably due to its use of Boolean encodings for the domains of the integer variables, as these encodings imply introducing a new Boolean variable for each element of the domain (see [29]).

The plot of Figure 2 shows the elapsed times, in logarithmic scale, for the solved instances of Table 4. The instances have been ordered by its execution time in each system. The overall best system (in terms of number of solved instances) is `fzn2smt`. However `fzn2smt` is the worst system (in terms of execution time) within the first 50 solved instances and, moreover, Gecode is better along the 160 first instances, closely followed by G12 lazy_fd. It must be taken into account that the `fzn2smt` compiler is written in Java, and it generates an SMT file for each decision problem that is fed into the chosen SMT solver. Hence this can cause an overhead in the runtime that can be more sensible for the easier instances. Finally, note also that `fzn2smt` scales very well from the 50 to the 150 first solved instances. This exhibits the SMT solvers robustness.

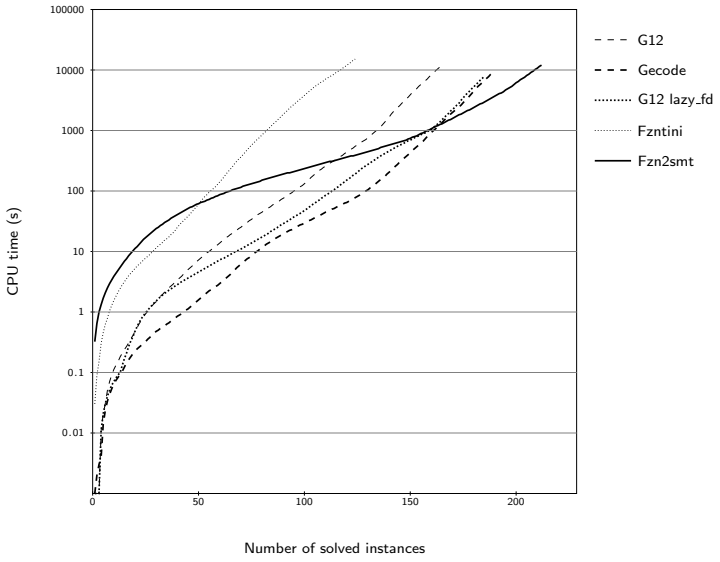

Fig. 2 Number of solved instances and elapsed times (referred to Table 4).

4.3 FlatZinc Solvers with Global Constraints

Some FlatZinc solvers provide specific algorithms for certain global constraints (such as `alldifferent`, `cumulative`, etc.). Thus, the user can choose not to decompose some global constraints during the translation from MiniZinc to FlatZinc, in order to profit from specific algorithms provided by the solvers.

Table 5 shows the performance of two FlatZinc solvers with support for global constraints compared to the performance of themselves, and that of `fzn2smt`, with-

Table 5 Performance comparison of `fzn2smt` vs available FLATZINC solvers with global constraints. +gc stands for using global constraints, and −gc stands for not using global constraints.

			Gecode		G12		`fzn2smt`
Problem	Type	#	+gc	−gc	+gc	−gc	
debruijn-binary	s	11	**4.14 (7)**	4.46 (6)	35.93 (7)	31.30 (6)	0.50 (1)
pentominoes	s	7	**65.82 (7)**	601.82 (7)	847.11 (4)	848.17 (4)	168.47 (2)
quasigroup7	s	10	**250.49 (6)**	278.73 (6)	1.55 (5)	1.72 (5)	20.03 (5)
rcpsp	o	10	10.56 (5)	12.07 (5)	0.51 (4)	97. 27 (5)	**993.74 (9)**
black-hole	s	10	20.88 (7)	69.68 (7)	2405.38 (6)	2423.48 (6)	**892.46 (8)**
nonogram	s	10	**493.61 (8)**	1353.63 (8)	351.35 (2)	336.93 (2)	1546.56 (7)
open-stacks	o	10	**168.56 (8)**	169.74 (8)	283.52 (8)	299.55 (8)	707.25 (7)
p1f	o	10	**730.60 (10)**	2.57 (8)	1.87 (8)	2.04 (8)	126.01 (9)
rect-packing	s	10	132.44 (6)	134.36 (6)	7.71 (5)	339.80 (6)	**106.66 (10)**
roster-model	o	10	**0.88 (10)**	1.04 (10)	4. 49 (10)	4.46 (10)	50.38 (10)
costasArray	s	5	**615.51 (4)**	943.75 (4)	411.82 (3)	423.09 (3)	664.13 (2)
depot-placement	o	15	1035.72 (12)	1205.75 (12)	519.64 (8)	522.81 (8)	**295.78 (15)**
filter	o	10	31.15 (1)	30.95 (1)	280.57 (1)	278. 72 (1)	**17.88 (6)**
ghoulomb	o	10	**1044.25 (10)**	0.00 (0)	598.54 (3)	246.67 (1)	2545.02 (6)
rcpsp-max	o	10	5.04 (2)	42.11 (2)	116.40 (2)	30. 00 (1)	**475.14 (4)**
sugiyama2	o	5	310.94 (5)	308.31 (5)	510.84 (5)	510. 72 (5)	**8.64 (5)**
bacp	o	15	848.88 (10)	847.78 (10)	979.85 (10)	976.35 (10)	**64.17 (15)**
Total		168	5769 (118)	6006 (105)	7357 (91)	7373 (89)	**8682 (121)**

out using that support, on problems where global constraints do occur. Note that `fzn2smt` does not provide any specific support for global constraints. We have not included the results for G12 lazy_fd with global constraints, since it exhibited very similar performance.

Again we can see that `fzn2smt` overall offers a bit better performance than Gecode and G12, even when they are using global constraints. This is even more significant if we take into account that most of these problems are optimization ones, and we have naively implemented a search procedure to supply the lack of support for optimization of SMT solvers (see Subsection 3.2.3). However, Gecode is best in 9 problems, whereas `fzn2smt` is best only in 8. We believe that unit propagation and conflict-driven lemma learning at Boolean level, partially compensate for the lack of specialized algorithms for global constraints in SMT solvers.

4.4 Impact of the Boolean component of the instances in the performance of `fzn2smt`

In this section we statistically compare the performance of `fzn2smt` with the best of the other available FLATZINC solvers, that is Gecode. We compare the number of solved instances by Gecode and `fzn2smt`, taking into account their Boolean component. In particular, we consider the number of Boolean variables and the number of non-unary clauses of the SMT instances resulting from the translation of each FLATZINC instance. We first look at this relation graphically (figures 3 and 4) and propose the following hypothesis: the more Boolean components the problem has, the better the performance of `fzn2smt` is with respect to that of Gecode. This hypothesis seems quite reasonable, because having a greater Boolean component, the SMT solver can better profit from built-in techniques such as unit propagation, learning and backjumping. We provide statistical tests to support this hypothesis.

First of all, we define the *normalized difference of solved instances* of each problem

$$dif = \frac{\#\texttt{fzn2smt}\ solved\ instances - \#Gecode\ solved\ instances}{\#instances}.$$

This difference ranges from -1 to 1, where -1 means that Gecode has solved all the instances and `fzn2smt` none, and 1 means the inverse.

We define the *Boolean variables ratio* r_v of each problem as the average of the number of Boolean variables divided by the number of variables of each SMT instance.

Similarly, we define the *disjunctions ratio* r_d of each problem as the average of the number of non-unary clauses divided by the number of constraints of each SMT instance.

In figure 3 we plot the differences with respect to the Boolean variables ratio, and in figure 4 with respect to the disjunctions ratio. These figures show that, the more Boolean variables and disjunctions the problem has, the better performance `fzn2smt` has, compared to Gecode. In particular, when the Boolean variables ratio r_v is above 0.2, `fzn2smt` is able to solve more instances than Gecode (i.e., the difference is positive). Only in two of those problems Gecode is able to solve more instances than `fzn2smt`, namely in **nmseq** (problem #2) and **open-stacks** (problem #14). In these problems, Boolean variables are mainly used in `bool2int()` constraints, hence these variables provide little Boolean structure and the SMT solver cannot profit from their contribution. When considering the disjunctions ratio, `fzn2smt` outperforms Gecode only when r_d is above 0.4. An exception to this fact is again on **nmseq**, where most disjunctions come from `bool2int()`.

Note that `fzn2smt` is able to solve more instances than Gecode in **propagation stress** (problem #16), which has neither Boolean variables nor disjunctions. This is probably due to the fact that the linear constraints of the problem can be solved by the *Integer Difference Logic* (IDL), a subset of LIA which is very efficiently implemented by Yices [21].

We use a paired t-test in order to show that our method (`fzn2smt` with Yices) solves significantly more instances than Gecode. For each problem $i \in 1..n$, being n the number of considered problems, we take X_i as the normalized number of instances solved by `fzn2smt` and Y_i as the normalized number of instances solved by Gecode, and define $D_i = X_i - Y_i$ with null hypothesis

$$H_0 : \mu_D = \mu_x - \mu_y = 0$$

i.e., $H_0 : \mu_x = \mu_y$. Then we calculate the t-value as

$$t = \frac{\overline{D}_n}{S^*_D/\sqrt{n}}$$

where $\overline{D}_n$ is the sample mean of the D_i and S^*_D is the sample standard deviation of the D_i. This statistic follows a Student's-t distribution with $n-1$ degrees of freedom.

Table 6 shows that in the general case with all 32 problems there is no significant difference between the means of the two solvers (the probability of the null hypothesis is $p = 0.2354$). Therefore we cannot say that `fzn2smt` is statistically better than Gecode. But, already for problems with Boolean variables ratio $r_v \geq 0.1$ we observe a significant difference (i.e., with $p < 0.05$ in all tests) in favor of `fzn2smt`. This confirms our hypothesis: the higher the Boolean variables ratio is, the better the performance of `fzn2smt` is with respect to that of Gecode. We also note that if we use the disjunctions ratio r_d for comparing the means, the

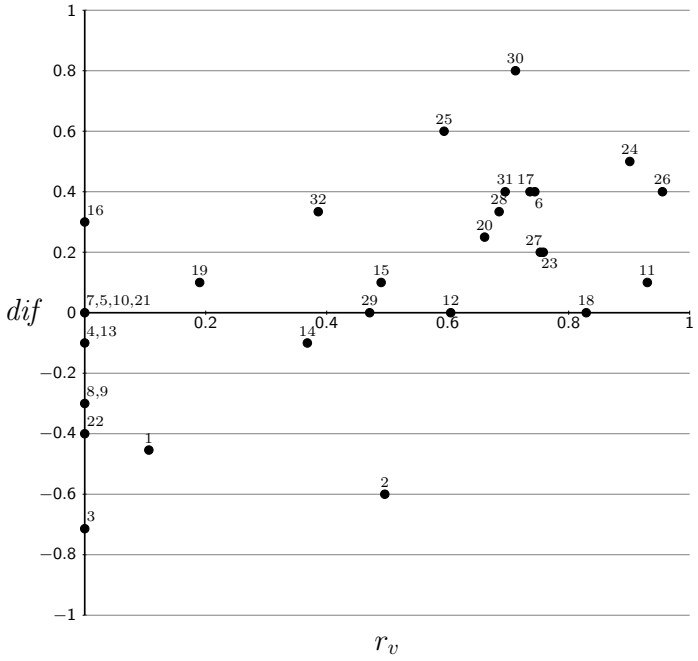

Fig. 3 Normalized difference of solved instances between `fzn2smt` and Gecode with respect to the ratio of Boolean variables. The numbers next to the points denote instance numbers (see Table 4).

results are similar: with $r_d >= 0.4$ the difference of means is significant in favor of `fzn2smt`.

Table 7 shows that the difference of means of `fzn2smt` and G12 lazy_fd is less significant. We have not taken into account the problems not supported by G12 lazy_fd due to unbounded integer variables. These results suggest that the two approaches work similarly well for the same kind of problems.

Table 6 Paired t-test, with probability p of the null hypothesis, for the difference in mean of the number of solved instances by `fzn2smt` and Gecode, for problems with different ratios r_v and r_d.

r_v	#problems	p
≥ 0.0	32	**0.2354**
≥ 0.1	21	**0.0150**
≥ 0.2	19	**0.0043**
≥ 0.3	19	**0.0043**
≥ 0.4	17	**0.0057**
≥ 0.5	14	**0.0001**
≥ 0.6	13	**0.0003**

r_d	#problems	p
≥ 0.0	32	**0.2354**
≥ 0.1	24	**0.0472**
≥ 0.2	24	**0.0472**
≥ 0.3	23	**0.0540**
≥ 0.4	17	**0.0060**
≥ 0.5	16	**0.0056**
≥ 0.6	11	**0.0006**

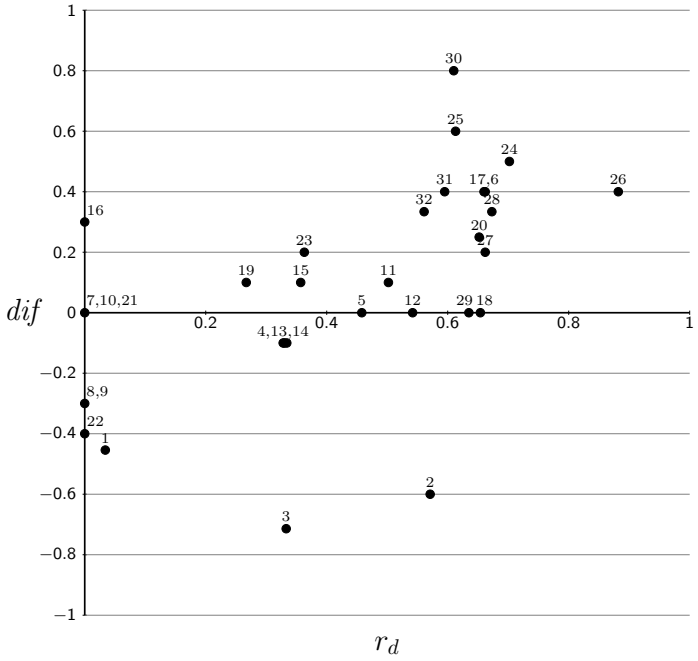

Fig. 4 Normalized difference of solved instances between `fzn2smt` and Gecode with respect to the ratio of disjunctions. The numbers next to the points denote instance numbers (see Table 4).

Table 7 Paired t-test, with probability p of the null hypothesis, for the difference in mean of the number of solved instances by `fzn2smt` and G12 lazy_fd, for problems with different ratios r_v and r_d.

r_v	#problems	p
≥ 0.0	27	**0.8929**
≥ 0.1	16	**0.0157**
≥ 0.2	15	**0.0152**
≥ 0.3	15	**0.0152**
≥ 0.4	14	**0.0147**
≥ 0.5	13	**0.0140**
≥ 0.6	12	**0.0028**

r_d	#problems	p
≥ 0.0	27	**0.8929**
≥ 0.1	20	**0.1853**
≥ 0.2	20	**0.1853**
≥ 0.3	19	**0.1856**
≥ 0.4	15	**0.0279**
≥ 0.5	14	**0.0274**
≥ 0.6	10	**0.0633**

5 Conclusion

In this paper we have presented `fzn2smt`, a tool for translating FlatZinc instances to the standard SMT-LIB v1.2 language, and solving them through a pluggable SMT solver used as a black box. Our tool is able to solve not only decision problems, but optimization ones. In spite of the lack of support for optimization of most SMT solvers, surprisingly good results have been obtained on many optimization problems by means of successive calls to the decision procedure, performing either linear, binary or hybrid (binary and linear) search. The `fzn2smt` system with Yices 2 has obtained remarkably good results in the MiniZinc challenges 2010 and 2011.

We have performed exhaustive experimentation on nearly 300 MiniZinc instances using four distinct FlatZinc solvers, and using `fzn2smt` with five distinct SMT solvers (using various translation options), lasting more than 12 days of CPU. The good results obtained by `fzn2smt` on the MiniZinc benchmarks suggest that the SMT technology can be effectively used for solving CSPs in a broad sense. We think that `fzn2smt` can help getting a picture of the suitability of SMT solvers for solving CSPs, as well as to compare the performance of state-of-the-art SMT solvers outside the SMT competition.

Table 4 evidences that in scheduling problems (rcpsp, depot-placement, rcpsp-max, wwtp and bacp) the performance of `fzn2smt` with Yices 2 is much better than that of other FlatZinc solvers. These problems have a rich Boolean component (Boolean variables and disjunctions). We have proposed the hypothesis that the more Boolean component the problem has, the better the performance of `fzn2smt` is with respect to Gecode.[8] This hypothesis seems quite reasonable, because the greater the Boolean component is, the better the SMT solver is supposed to profit from built-in techniques such as unit propagation, learning and backjumping. We have also provided statistical tests that support this hypothesis.

We recall that SMT solvers have been developed for solving verification problems that typically are small but hard. As pointed out by de Moura in [18], the efficiency of a solver is strongly dependent on its predefined strategies. Hence changing these heuristics could dramatically affect the SMT solver performance in other problem domains. We believe that there is much room for improvement in solving CSPs with SMT. For instance, apart from the possibility of controlling the solver strategies, we think that developing theory solvers for global constraints is also a promising research line. In fact, there exist already some promising results in this direction, for instance, for the `alldifferent` theory [7]. On the other hand, we think that better results could be obtained if directly translating from the MiniZinc language to SMT and avoiding some of the flattening. In doing so, most clever translations could be possible and probably less variables could be generated. For instance, MiniZinc disjunctions of arithmetic constraints are translated into FlatZinc constraints by reifying them with auxiliary Boolean variables. In our approach this step is not needed since it is already done by the SMT solver.

A better approach to optimization in SMT is also a pending issue. This has been, in a sense, recently addressed in [17] by introducing a theory of costs.

In conclusion, we think that the potential of the SAT and SMT ideas and technology has still not been sufficiently considered by the CP community. A relevant work in this direction is that of [29], where finite domain propagation is mimicked by (lazily) mapping propagators into clauses in a SAT solver. This way, crucial elements of modern SAT technology (such as two watched literals, 1UIP nogoods and conflict directed backjumping) are indirectly incorporated into a CSP solver, providing very good results in hard scheduling problems without the need of complex search strategies. Related to this let us mention that, although not eligible for prizes, the true winner of the MiniZinc challenges 2010 and 2011 (in all categories) was Chuffed, a solver developed by the MiniZinc group with lazy clause generation in mind. This solver reached orders of magnitude in search reduction on appropriate problems, compared to when running itself with lazy clause generation disabled.

[8] Gecode is the second best system considering the total number of solved instances.

Hence, it is apparent that more cross-fertilization between the CP and the SAT and SMT communities is necessary in order to tackle real-world problems properly.

Acknowledgements We thank Santiago Thió-Henestrosa for helping us with the statistical tests.

References

1. JaCoP Java Constraint Programming Solver. `http://jacop.osolpro.com`, 2010.
2. Minizinc + Flatzinc. `http://www.g12.csse.unimelb.edu.au/minizinc/`, 2010.
3. SCIP, Solving constraint integer programs. `http://scip.zib.de/scip.shtml`, 2010.
4. SICStus Prolog. `http://www.sics.se/sisctus`, 2010.
5. W. Ackermann. *Solvable cases of the decision problem*. Studies in logic and the foundations of mathematics. North-Holland, 1968.
6. K. R. Apt and M. Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.
7. M. Banković and F. Marić. An Alldifferent Constraint Solver in SMT. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.
8. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.SMT-LIB.org`, 2010.
9. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.
10. M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. The Barcelogic SMT Solver. In *CAV*, volume 5123 of *LNCS*, pages 294–298. Springer, 2008.
11. M. Bofill, M. Palahí, J. Suy, and M. Villaret. SIMPLY: a Compiler from a CSP Modeling Language to the SMT-LIB Format. In *Proceedings of the Eigth International Workshop on Constraint Modelling and Reformulation*, pages 30–44, 2009.
12. M. Bofill, J. Suy, and M. Villaret. A System for Solving Constraint Satisfaction Problems with SMT. In *SAT*, volume 6175 of *LNCS*, pages 300–305. Springer, 2010.
13. M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient Theory Combination via Boolean Search. *Information and Computation*, 204(10):1493–1525, 2006.
14. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, A. Santuari, and R. Sebastiani. To Ackermann-ize or Not to Ackermann-ize? On Efficiently Handling Uninterpreted Function Symbols in $SMT(\mathcal{EUF} \cup \mathcal{T})$. In *LPAR*, volume 4246 of *LNCS*, pages 557–571. Springer, 2006.
15. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT Solver. In *CAV*, volume 5123 of *LNCS*, pages 299–303. Springer, 2008.
16. M. Cadoli, T. Mancini, and F. Patrizi. SAT as an Effective Solving Technology for Constraint Problems. In *ISMIS*, volume 4203 of *LNCS*, pages 540–549. Springer, 2006.
17. A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability Modulo the Theory of Costs: Foundations and Applications. In *TACAS*, volume 6015 of *LNCS*, pages 99–113. Springer, 2010.
18. L. M. de Moura. Orchestrating Satisfiability Engines. In *CP*, volume 6876 of *LNCS*, page 1. Springer, 2011.
19. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
20. B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`, August 2006.
21. B. Dutertre and L. M. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.
22. A. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, and I. Miguel. Essence: A Constraint Language for Specifying Combinatorial Problems. *Constraints*, 13(3):268–306, 2008.
23. J. Huang. Universal Booleanization of Constraint Models. In *CP*, volume 5202 of *LNCS*, pages 144–158. Springer, 2008.

24. G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 1(2):245–257, 1979.
25. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In *CP*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
26. R. Nieuwenhuis and A. Oliveras. On SAT Modulo Theories and Optimization Problems. In *SAT*, volume 4121 of *LNCS*, pages 156–169. Springer, 2006.
27. R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Challenges in Satisfiability Modulo Theories. In *RTA*, volume 4533 of *LNCS*, pages 2–18. Springer, 2007.
28. R. Nieuwenhuis, 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(6):937–977, 2006.
29. O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via Lazy Clause Generation. *Constraints*, 14(3):357–391, 2009.
30. S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Dept. of Comp. Science, University of Iowa, 2006. Available at `www.SMT-LIB.org`.
31. C. Schulte, M. Lagerkvist, and G. Tack. Gecode. `http://www.gecode.org`, 2010.
32. R. Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3(3-4):141–224, 2007.
33. H. M. Sheini and K. A. Sakallah. From Propositional Satisfiability to Satisfiability Modulo Theories. In *SAT*, volume 4121 of *LNCS*, pages 1–9. Springer, 2006.
34. R. E. Shostak. Deciding Combinations of Theories. *J. ACM*, 31(1):1–12, 1984.
35. P. J. Stuckey, R. Becket, and J. Fischer. Philosophy of the MiniZinc Challenge. *Constraints*, 15:307–316, July 2010.
36. N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling Finite Linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.
37. P. Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, 1999.
38. T. Walsh. SAT v CSP. In *CP*, volume 1894 of *LNCS*, pages 441–456. Springer, 2000.
39. L. Zhang and S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. In *CAV*, volume 2404 of *LNCS*, pages 17–36. Springer, 2002.