Universal Booleanization of Constraint Models

Jinbo Huang

National ICT Australia and Australian National University
jinbo.huang@nicta.com.au

Abstract. While the efficiency and scalability of modern SAT technology offers an intriguing alternative approach to constraint solving via translation to SAT, previous work has mostly focused on the translation of specific types of constraints, such as pseudo Boolean constraints, finite integer linear constraints, and constraints given as explicit listings of allowed tuples. By contrast, we present a translation of constraint models to SAT at language level, using the recently proposed constraint modeling language MiniZinc, such that any satisfaction or optimization problem written in the language (not involving floats) can be automatically Booleanized and solved by one or more calls to a SAT solver. We discuss the strengths and weaknesses of such a universal constraint solver, and report on a large-scale empirical evaluation of it against two existing solvers for MiniZinc: the finite domain solver distributed with MiniZinc and one based on the Gecode constraint programming platform. Our results indicate that Booleanization indeed offers a competitive alternative, exhibiting superior performance on some classes of problems involving large numbers of constraints and complex integer arithmetic, in addition to, naturally, problems that are already largely Boolean.

1 Introduction

General constraint satisfaction and optimization problems are often solved with finite domain (FD) or linear programming (LP) techniques. When variables are all Boolean, however, satisfiability (SAT) solvers offer a natural solution whose efficiency and scalability in practice remain largely unmatched to date.

Booleanization of constraints has thus remained an ongoing quest in the constraint programming (CP) community. For example, pseudo-Boolean constraints (integer linear constraints over the domain $\{0, 1\}$), including the special case of Boolean cardinality constraints, have come under continual scrutiny due to their resemblance to their Boolean cousins [1–5]. Restricted to finite domains, general integer linear constraints have also been Booleanized by transforming all constraints to *primitive comparisons*, of the form $x \leq c$, and encoding each of these by a distinct Boolean variable [6]. Set constraints over a finite universe have been Booleanized by creating a Boolean variable for each possible element of a set [7]. For other types of constraints over finite domain variables, particularly extensional constraints (those given as explicit listings of allowed tuples), it's sometimes effective to use the well-known technique of encoding each value of a variable's domain with a distinct Boolean variable.

```
0   int: z = 10;
1   array [0..z] of 0..z*z: sq = [x*x | x in 0..z];
2   array [0..z] of var 0..z: s;
3   var 0..z: k;
4   var 1..z: j;
5   constraint forall ( i in 1..z ) ( s[i] > 0 -> s[i - 1] > s[i] );
6   constraint s[0] < k;
7   constraint sum ( i in 0..z ) ( sq[s[i]] ) = sq[k];
8   constraint s[j] > 0;
9   solve maximize j;
```

Fig. 1. A "perfect square" problem in MiniZinc.

These previous lines of work have allowed CP to directly benefit from the great advances in SAT and related technologies over the past decade. In attempting to further push the envelope, however, one identifies two major limitations in this body of work. First, the types of constraints dealt with are limited; in particular, many nonlinear operations (such as multiplication/division, min/max, and the example given below involving array access) frequently required in modeling are left unsupported. Second, the techniques proposed, being specific to the respective types of constraints they target, are not necessarily compatible and are implemented for different problem specification formats, making the Booleanization of heterogeneous constraint models a difficult task both in theory and practice.

Both these limitations can be removed at once by a procedure that systematically Booleanizes a general constraint *language*, rather than special constraint types, which is the subject of the present paper. Although the techniques we shall present will apply as well to similar languages, our actual Booleanization procedure has been developed and implemented for MiniZinc [8], a recently proposed simple but expressive constraint modeling language. This choice of language is ideal for us as two existing solvers for MiniZinc are available for comparison, and the public distribution of MiniZinc [9] contains a large number of examples and benchmarks of different types, both of which facilitate the empirical evaluation of our new solver. An additional benefit of using MiniZinc, as we discuss in Section 2, is that it comes with a tool that "flattens" things into a more convenient subset of the language, without compromising its expressiveness.

Fig. 1 gives an example constraint model expressed in MiniZinc taken from the MiniZinc distribution [9], modeling the size-10 instance of the "perfect square" problem—finding a largest set of integers from $\{1, \ldots, z\}$ the sum of whose squares is itself a square. While this example should be largely self-explanatory,[1] a detailed explanation of the syntax of MiniZinc can be found in [8]. It's worth

[1] Line 0 declares a constant $z = 10$; line 1 declares an initialized array $sq = [0, 1, 2, 4, \ldots, z^2]$; line 2 declares an array s of integers from domain $\{0, \ldots, z\}$; lines 3 and 4 declare integers k and j from domains $\{0, \ldots, z\}$ and $\{1, \ldots, z\}$, respectively; line 5 asserts the logical condition $(s[i] > 0) \rightarrow (s[i-1] > s[i])$ for all i in $\{1, \ldots, z\}$; line 7 asserts $\sum_{i \in \{0,\ldots,z\}} sq[s[i]] = sq[k]$.

noting here, though, that this small model, written in a natural way, already contains a type of constraint not handled by previous Booleanization methods: On line 7, we are summing over elements of an array ($sq[\,]$) using indices ($s[i]$) that are themselves variables, something that cannot be readily turned into a linear constraint, nor an explicit listing of allowed tuples of a reasonable length.

We shall now begin our journey toward a Booleanization of MiniZinc, which will allow problems such as this one to be automatically translated and solved by a SAT solver.

2 Basis for Universal Booleanization

Booleanization of constraint models at language level naturally consists of two parts: Boolean encoding of variables and Boolean encoding of constraints.

MiniZinc provides three scalar types: Booleans, integers, and floats; and two compound types: sets and arrays. In this work we do not consider floats. Hence we need to handle the following variable types: (1) integers (both bounded and unbounded), (2) sets of integers (must be bounded, a requirement in MiniZinc), (3) arrays of Booleans, (4) arrays of integers (both bounded and unbounded), and (5) arrays of sets of integers. We need not directly handle multi-dimensional arrays as MiniZinc comes with a tool that automatically flattens those, among other things, so that the model is rephrased in a subset of MiniZinc called FlatZinc [8], where the above five categories are the only possible types.

The basis for our Booleanization of an integer is to use k Boolean variables to represent the bits of the number in binary. This ensures that the encoding will be adequate for all possible types of constraints—as long as k is sufficiently large (we address this later). Sets are Booleanized as in [7], by using a Boolean variable for each possible element of the set (the boundedness helps here). Arrays are decomposed into individual variables, one for each index, which is feasible as MiniZinc requires that array index ranges be fixed at compile time.

The second, more substantial part of the task is to Booleanize all types of constraints that can be written in MiniZinc. This is facilitated again by the tool that converts MiniZinc to FlatZinc, where all user defined predicates have been inlined, compound operators (**forall**, **sum**, and **product**, which range over elements of an array) unrolled, and all constraints normalized as necessary to conform to a pre-defined set of *operators* (i.e., constraint types). It then remains for us to provide a translation for each of these operators.

The next section presents our Booleanization procedure in more detail, assuming that MiniZinc models have been converted to FlatZinc in a preprocessing step (note that this makes our Booleanization applicable not only to MiniZinc, but potentially any language that can be translated to FlatZinc). A FlatZinc model consists of a list of variables declarations followed by a list of constraints and finally a specification of the nature of the problem (satisfaction/optimization) and the desired output—all of these are known as *items* of the model, and we will describe the translation of them in that order.

3 An Itemized Procedure

It's perhaps helpful to interpose a description of our "finished product" here, so that the goal will be clear when we go through our translation procedure. This is a program called FznTini, as it takes FlatZinc models as input and solves them using an interface to the SAT solver Tinisat [10]. It also has the option of printing a Boolean translation of the model (for a given k—the number of bits used to encode an integer) without solving it, in one of two formats: Boolean FlatZinc (the subset of FlatZinc where all variables are Boolean and all constraints are basic Boolean operations; this is to allow the integration of the Booleanization into the G12 platform [11] which understands FlatZinc) and the DIMACS CNF format widely accepted by SAT solvers. For optimization problems, the translation encodes all the constraints, and information on optimization (i.e., which Boolean variables correspond to the integer variable to be maximized/minimized, and its lowerbound and upperbound if known) is coded as an annotation item in the case of Boolean FlatZinc and a special comment line in the case of DIMACS, so that solvers accepting these translations can reconstruct the original optimization version of the problem and solve it accordingly.

Going from Boolean FlatZinc to DIMACS will be straightforward, as it involves simply converting basic Boolean operations to CNF; hence we will describe our translation using Boolean FlatZinc as the target language. Language syntax will be explained as we go along.

3.1 Booleanization of variable declarations

Declarations of Boolean variables are left untouched. Hence we have the following five cases to handle as explained earlier.

Integers An integer declaration in FlatZinc has the following form:

```
var int: x;
```

which translates into declarations of k Boolean variables:

```
var bool: x_1; ... ; var bool: x_k;
```

Note that the symbols `x_1` through `x_k` are meant to represent k fresh identifiers that are not used in the source model or the translation of previous items. This convention applies through the rest of the paper.

A bounded integer is declared as follows:

```
var <g>..<h>: x;
```

where `<g>` and `<h>` are two integer constants giving the domain (e.g., 0..10). This translates into the same k Boolean variables as above, but we also add the following two (less-than-or-equal-to) integer comparison constraints, which are then Booleanized along with other constraints in the source model, as will be described in Section 3.2 (note that depending on the values of g and h, these constraints may fix some of the bits of `x` to constants, effectively reducing the actual number of bits required to encode a bounded integer):

```
int_le(<g>, x); int_le(x, <h>);
```

Sets of integers A set of (bounded) integers is declared as follows:

```
var set of <g>..<h>: S;
```

which means that the value of `S` must be a set whose elements are from the universe $\{g, \ldots, h\}$. This translates into declarations of $h - g + 1$ Boolean variables:

```
var bool: S_g; ...; var bool: S_h;
```

where each `S_i` encodes the proposition "`<i>` $\in$ `S`."

Arrays of Booleans Arrays in FlatZinc are always 0-indexed.[2] A declaration of an array of Booleans has the following form:

```
array[0..<m>] of var bool: X;
```

which translates into declarations of $m + 1$ Boolean variables:

```
var bool: X_0; ...; var bool: X_m;
```

Arrays of integers Each declaration of an array of integers

```
array[0..<m>] of var int: X;
```

is first decomposed (conceptually) into declarations of of $m+1$ integer variables:

```
var int: X_0; ...; var int: X_m;
```

which are then Booleanized the same way as other integers in the source model (in practice, of course, the intermediate step need not take place explicitly).

Each declaration of an array of bounded integers

```
array[0..<m>] of var <g>..<h>: X;
```

translates into the same Boolean variables as in the unbounded case, plus the translation of additional integer comparison constraints as in the case of bounded scalar integers.

[2] Subsequent to the completion of this work, the Zinc family of languages and the MiniZinc benchmarks and examples have been modified to use 1-indexed arrays.

Arrays of sets of integers Each declaration of an array of sets of integers:

```
array[0..<m>] of var set of <g>..<h>: X;
```

is first decomposed (conceptually) into declarations of $m + 1$ set variables:

```
var set of <g>..<h>: X_0; ...; var set of <g>..<h>: X_m;
```

which are then Booleanized the same way as other set variables in the source model (again, the intermediate step need not actually take place).

Finally, we note that these five types of variable declarations can all take an (initialization) assignment, in which case the "variables" effectively become constants and the `var` keyword in the declaration can be omitted (as in lines 0–1 of Fig. 1). These cases can be handled in one of two ways: (1) We can simply skip the declaration, storing the value of the variable in a look-up table, and plug in the value when later translating a constraint involving that variable. This method can be easily implemented for Booleans, integers, arrays of Booleans, and arrays of integers. (2) We can Booleanize the variables the same way as if they weren't initialized, and then add constraints equating the resulting Boolean variables to appropriate Boolean constants corresponding to the assignment. This method has the advantage of uniformity, in that when we later translate the constraints, initialized and uninitialized variables need not be distinguished. This makes it easier to implement for the more complex variables types of sets and arrays of sets. This is a relatively unimportant choice, after all, and our implementation uses a combination of both methods.

3.2 Booleanization of constraints

Booleanization of constraints involving integers will depend on how an integer is represented by the k bits. We assume the *two's complement* representation commonly used in computers, where a positive number has the usual representation and flipping all its bits and adding 1 gives its negation. For example, if $k = 4$, then 3 would be 0011 and -3 would be 1101. We also assume that `x_k` represents the most significant, and `x_1` the least significant bit of `x`

Constraints in FlatZinc are instances of a pre-defined set of operators, grouped into several categories. Below we will use these as headings to present translations of constraints. In FlatZinc, arguments to a constraint can be either a variable, array access, or constant. For simplicity, we will represent arguments as variables in all operators (except in linear constraints where the coefficients must be constants). The other two cases can be handled in the following straightforward way: (1) Any argument in the form of `X[<i>]` (array access with a constant index) is replaced with `X_<i>` before applying the relevant translation procedure (recall that these individual variables have been invented in translating the array declaration); note that array access with a variable index would not appear as such in FlatZinc, but will have been normalized during flattening into *array element operators*, which we will cover below. (2) Constants can be handled by (conceptually) inventing a temporary variable in its place and plugging in appropriate Boolean constants at the end.

```
Algorithm 1 Comparison (≤) of two k-bit integers (in two's complement)
int_le($x, y, k$): assuming $x_k \dots x_1$ ($y_k \dots y_1$) are the bits of $x$ ($y$)
    1: return $(x_k > y_k) \lor ((x_k = y_k) \land$ unsigned_int_le$(x, y, k-1))$

unsigned_int_le($x, y, k$)
    2: if $k = 1$ return $x_1 \le y_1$
    3: return $(x_k < y_k) \lor ((x_k = y_k) \land$ unsigned_int_le$(x, y, k-1))$
```

```
bool_gt_reif(x_k, y_k, a_1);     % a_1 = (x_k > y_k)
bool_eq_reif(x_k, y_k, a_2);     % a_2 = (x_k = y_k)

bool_lt_reif(x_k-1, y_k-1, b_1) % b_1 = (x_k-1 < y_k-1)
bool_eq_reif(x_k-1, y_k-1, b_2) % b_2 = (x_k-1 = y_k-1)
...                               ...
bool_lt_reif(x_2, y_2, b_2k-5)  % b_2k-5 = (x_2 < y_2)
bool_eq_reif(x_2, y_2, b_2k-4)  % b_2k-4 = (x_2 = y_2)

bool_le_reif(x_1, y_1, b_2k-3)  % b_2k-3 = (x_1 <= y_1)

bool_or(a_1, c_1, true);        % true = (a_1 or c_1)
bool_and(a_2, c_2, c_1);        % c_1 = (a_2 and c_2)
                                % rest is recursive definition of c_i
bool_or(b_1, c_3, c_2);
bool_and(b_2, c_4, c_3);
...
bool_or(b_2k-5, c_2k-3, c_2k-4);
bool_and(b_2k-4, c_2k-2, c_2k-3);

bool_eq(c_2k-2, b_2k-3);        % base case for recursion
```

Fig. 2. Comparison ($\le$) of two k-bit integers in Boolean FlatZinc.

Comparison operators There are three sets of comparison operators in FlatZinc, for Booleans, integers, and sets, respectively. Each set contains six operators corresponding to $=, \neq, \le, <, \ge$, and $>$. The Boolean operators need not be touched. Hence we consider the Booleanization of integer and set comparisons. For both of these, we use the $\le$ as a representative as the cases of $<, \ge$, and $>$ are similar and the cases of $=$ and $\neq$ are simpler. All these operators have reified versions, which we also omit as the modifications required are straightforward.

The $\le$ comparison of two integer variables in FlatZinc, `int_le(x, y)`, can be translated by simulating Algorithm 1, which performs the comparison of two integers at bit level. Recall that in two's complement $x_k = 1$ signifies a negative number and $x_k = 0$ signifies a nonnegative number. Hence line 1 of Algorithm 1 says that $x \le y$ if $x < 0$ and $y \ge 0$, or if x and y have the same sign bit and the remaining bits of x are $\le$ those of y, both taken as unsigned numbers. The comparison of unsigned numbers is then implemented as a recursive function on lines 2–3, where the logic should be straightforward.

Now to convert this into a set of Boolean constraints, we unroll the recursion, introduce auxiliary variable a_1, a_2, and b_i for $i \in \{1, \dots, 2k-3\}$ to encode the various bit comparisons on lines 1 and 3 of the algorithm, and introduce auxiliary variables c_{2i} to encode the value of unsigned_int_le(x, y, i) and c_{2i-1} to encode

Algorithm 2 Comparison ($\leq$) of two sets of integers over the universe $\{g, \ldots, h\}$

```
set_le($X, Y, g, h$): assuming $X_g \ldots X_h$ ($Y_g \ldots Y_h$) are the Boolean variables created in
translating the declaration of set $X$ ($Y$) as described in Section 3.1
  1: if $g = h$ return $X_g \leq Y_g$
  2: return $((X_g < Y_g) \wedge (\bigwedge_{i=g+1}^{h} \neg X_i))$
  3:        $\vee((X_g > Y_g) \wedge (\bigvee_{i=g+1}^{h} X_i))$
  4:        $\vee((X_g = Y_g) \wedge$ set_le$(X, Y, g+1, h)$
```

the conjunction on line 3, for each $i \in \{1, \ldots, k-1\}$. Fig. 2 shows the resulting translation in Boolean FlatZinc, where comments have been added (after the % signs) to explain the meanings of the operators (the `constraint` keyword has been omitted from the beginning of each line).

The conversion of Algorithm 1 into the Boolean FlatZinc in Fig. 2 illustrates how such conversions may be done in general. Hence in the rest of the section we will refrain from giving actual Boolean FlatZinc code (which would be space-consuming) and focus on describing the bit-level algorithms or logical formulas, and sometimes just the general ideas, behind the Booleanization.

We now turn to the $\leq$ comparison of sets, which in FlatZinc is defined lexicographically as if a set is a "string" made up of its elements (in increasing order). For example: $\{1, 2, 3\} \leq \{2\} \leq \{2, 3\}$. Note that this differs from the lexicographical comparison of the characteristic bit strings of the sets (which would be 111, 010, and 011 in this example, assuming a common universe of $\{1, 2, 3\}$), or the "unsigned_int_le" function from Algorithm 1 would have sufficed.

Algorithm 2 gives a recursive bit-level implementation of the $\leq$ comparison of sets over the common universe $\{g, \ldots, h\}$, assuming that the set variables have themselves been Booleanized as described in Section 3.1. Line 1 is the straightforward base case, where there is only one possible element in both sets. Otherwise we examine the first pair of characteristic bits (X_g and Y_g), and there are three cases. If $X_g < Y_g$ (line 2), then $g \notin X$ and $g \in Y$, and hence $X \leq Y$ iff X is empty (the second conjunct on line 2), because any other element will be greater than g and thus make $X > Y$. The second case (line 3) can be similarly analyzed and finally if the pair are equal then we recurse (line 4).

The case of set variables defined over different universes can be handled using the trick of inventing temporary variables. Specifically, we need only let $\{g, \ldots, h\}$ be the smallest range containing the universes of both sets, and continue to use Algorithm 2, with all X_i and Y_j replaced with false for all i outside X's universe and j outside Y's.

Arithmetic operators FlatZinc provides the following arithmetic operators: negation, addition, subtraction, multiplication, division, modulo, absolute value, min, and max. These can all be Booleanized using standard algorithms that perform the corresponding operations on binary numbers. We will not go into their details, but let us use the space here to mention the more critical issue of overflow protection.

In computers, arithmetic overflow can lead to an incorrect result; in the Booleanization of a constraint model, it can lead to spurious solutions if not guarded against. Our solution is analogous to what's implemented in computer hardware. For addition, both operands as well as the sum are extended by one bit (at the high end) and a constraint is added requiring the leading two bits of the sum to be identical. For multiplication, the product will temporarily have $2k$ bits, and we add constraints to ensure the leading $k+1$ bits are identical (the lower k bits then correctly represent the result). The other cases can be similarly handled where necessary.

Linear equality and inequalities An integer linear equality constraint comes in the following form in FlatZinc:

```
int_lin_eq([C1, ..., Cn], [X1, ..., Xn], rhs);
```

where `C1, ..., Cn` and `rhs` are integer constants and `X1, ..., Xn` are integer variables (can also be constants or array accesses; see paragraph 2 of Section 3.2). This encodes the equality $\sum_i CiXi = rhs$.

Inequality constraints have the same form, with the "eq" in the name of the operator replaced by "ne, le, lt, ge, gt" respectively for $\neq, \leq, <, \geq, >$. All of them have a reified version taking a Boolean as the fourth argument.

We handle these linear constraints by breaking them down to individual multiplications and additions, followed by an integer comparison, all of which we already know how to Booleanize (introducing auxiliary variables to encode intermediate results).

Set operators FlatZinc provides the following set operators: membership, cardinality, subset, superset, union, intersection, difference, and symmetric difference. Again, for simplicity we assume that sets involved in the same operator are defined over the same universe $\{g, \ldots, h\}$ (otherwise the trick of inventing temporary variables can be applied as described earlier).

The set membership operator, `set_in(x, Y)`, which encodes $x \in Y$, translates into $\bigvee_{i=g}^{h}((x = i) \wedge Y_i)$. Note that the comparison $x = i$ is on integers, but we already know how to Booleanize those and can easily expand the expression into a purely Boolean one. Set cardinality, `set_card(X, y)`, can be translated by adding up the Booleans X_i as integers, and equating the sum to y, both of which we know how to Booleanize (if `y` is a constant, then a unary representation of integers can offer more propagation power in this case [1]). Set inclusion operators, `set_subset(X, Y)` and `set_superset(X, Y)`, are the simplest case, translating into $\bigwedge_{i=g}^{h}(X_i \leq Y_i)$ and $\bigwedge_{i=g}^{h}(X_i \geq Y_i)$, respectively.

The remaining operators all take three arguments, with the final one holding the result of the operation. The union operator, `set_union(X, Y, Z)`, for example, encodes $X \cup Y = Z$, and translates into $\bigwedge_{i=g}^{h}((X_i \vee Y_i) = Z_i)$. Replacing the disjunction ($\vee$) in this formula with $\wedge$, $>$, and $\neq$, respectively, gives a translation for the other operators, intersection, difference, and symmetric difference.

Array element operators Array access with a variable index is expressed in FlatZinc indirectly, by first equating the array element with a new variable, via *array element operators*, and then using that variable in other constraints where the array access is required. The following operator applies to an array of Booleans, encoding `Y[x] = z`:

```
array_var_bool_element(x, Y, z);
```

Let m be the highest index of the array. This operation then translates into $\bigvee_{i=0}^{m}((x = i) \wedge (Y_i = z))$. Again, the integer comparisons $x = i$ are meant to be further Booleanized and the results plugged in.

The corresponding operators for arrays of integers and arrays of sets translate into the same formula as above, except that $Y_i = z$ will be an integer comparison, and set comparison, respectively, which we already know how to Booleanize.

Global constraints In addition to the above groups of operators, a FlatZinc model can also use global constraints. At the moment, `all_different` (over an integer array) is the only type of global constraints that is supported by MiniZinc but not implemented by the official MiniZinc-to-FlatZinc translator. We Booleanize an `all_different` constraint over n elements by turning it into $n(n-1)/2$ inequalities and Booleanizing them as we do comparison operators.

3.3 Booleanization of `solve` and `output` items

FlatZinc provides three types of `solve` items: `solve satisfy`, `solve minimize x`, and `solve maximize x`, where `x` is an integer variable. Optimization of an expression is provided for indirectly: One introduces constraints equating the expression with a new variable, and optimizes that variable instead.

In Booleanization a `solve satisfy` item is left untouched. In the case of `solve minimize x` and `solve maximize x`, it's not possible for a single (ordinary) Boolean translation to encode the original optimization problem. Our solution is to turn them both into a `solve satisfy`, and, as mentioned in the beginning of the section, use an annotation or comment depending on the target format (Boolean FlatZinc or DIMACS CNF) to encode the information necessary for the optimization version of the problem to be reconstructed. Such translations will be complete and solvers accepting them are then free to decide how to handle the optimization.

FznTini solves optimization problems by an uninformed binary search, reducing the domain of the objective variable by at least half at each step. Since integers are encoded using k bits, any optimization problem can be solved this way by at most $k + 1$ calls to the SAT solver. Note that these successive calls will involve the original CNF formula plus different sets of new clauses encoding the bounds that are being tested. SAT solvers that support incremental addition and removal of clauses will hence be particularly suitable for these tasks.

For both satisfaction and optimization problems, FznTini starts with a k sufficient to encode all constants in the problem, and automatically increases it until either a solution is found, or k reaches the size of a C++ int (typically 32) on the machine on which it's run (this exceeds the capacity of G12/FD, which is fixed at 22 bits). Note that since overflow protection is in place, a solution found under any k is guaranteed to correspond to a correct solution for the original problem, while an "unsatisfiable" answer could just mean that k is not large enough. By the same token, for optimization problems the guanrantee of the optimality of a solution returned by FznTini is conditioned on the k used. In many problems, however, bounds on the objective variable are given either implicitly or explicitly, in which case the guarantee of optimality provided by FznTini will be absolute.

Lastly, output items are also encoded as annotations or comments so solvers can print output back in ordinary decimal.

3.4 Complexity of the encoding

The size of the Boolean encoding described in this section is quadratic in k, the number of bits used for an integer, for the multiplication, division, and modulo operators, and linear in k for other arithmetic operators and integer comparison operators. Where arrays or sets are involved, it's also linear in the size of the array or the size of the universe of the set, or both in the case of arrays of sets. For linear constraints, it's again quadratic in k as multiplications are involved.

In practice the size of the Booleanization is usually large compared with that of the MiniZinc or FlatZinc model (as one would expect), although not necessarily so from a SAT solver's point of view. For example, the little "perfect square" problem in Fig. 1 grows into 37 variables and 46 constraints in FlatZinc, and after Booleanization has 5441 variables and 5364 constraints in Boolean FlatZinc, or 5441 variables and 14576 clauses in CNF. However, it was solved in just 0.05 second by FznTini including translation and SAT solving time.

4 Weaknesses and Strengths

Needless to say, the goal of our universal Booleanization is *not* to solve CP at one fell swoop. Rather we are interested to see how far the SAT-based approach can be pushed, and on what types of problems it might suffer, or excel, so that one can be better informed when designing hybridizations of different techniques. In this light it's perhaps fitting to reflect for a moment on some possible weaknesses and strengths that might be inherent in this approach, before examining concrete experimental results.

The single most apparent weakness of such a solver is its remarkable "blindness." All explicit domain knowledge and structure is lost when everything boils down to Booleans. This is particularly acerbated in cases where the MiniZinc models contain annotations written by the user giving hints on the nature of the variables and constraints, what techniques might suit which constraints, what variable orderings might work best, etc., which the G12/FD solver is designed to read and make use of. Information contained in these annotations is all lost (in fact, ignored) through Booleanization, and all we can rely on is the generic heuristics of whichever SAT solver we use.

The binary search used by FznTini to optimize an integer variable is also blind in that it cannot directly benefit from techniques that may make an informed search efficient in the original search space. One type of heuristic may still be possible though: When the SAT solver is looking for a solution during one of the steps of the binary search, it can be helpful to have the solver prefer "larger" solutions in the case of maximization and the opposite in the case of minimization, where "larger" generally means false instead of true for the sign bit of the objective integer variable, and the opposite for its other bits. It's clear that an intermediate solution closer to the goal helps cut the domain of the objective variable faster, reducing the number of steps required in the binary search. However, such a heuristic can interfere with those of the SAT solver's, and combining them to the best advantage is nontrivial. Our preliminary experiments have not identified a way to achieve a consistent improvement; hence FznTini doe not currently use such a heuristic.

On the other hand, universal Booleanization offers an efficient way to seamlessly combine the propagators of all constraints, through the unit propagation of a SAT solver (an analysis of the propagation power of our Booleanization of the constraints is beyond the scope of this paper). In particular, queueing of propagators becomes irrelevant as all constraints are always propagated at once and the propagation iterated to saturation. This feature is especially interesting when one considers that the types of constraints in a model can be quite varied and an advantageous integration of their respective propagators may otherwise have been nontrivial.

The second major strength of universal Booleanization, which also applies to previous work that translated specific types of constraint problems to SAT, lies in the general efficiency and scalability of modern SAT solvers. In our case a clause learning SAT solver has been used, as clause learning is currently known as the best technique for SAT problems arising from real-world applications [12]. Particularly of relevance here is the fact that clause learning is known to be more general and potentially more powerful than traditional nogood learning in constraint solvers (the basic reason is that learned clauses can involve any variables, while traditional nogoods involve decision variables only) [13]. Also, while SAT solvers cannot directly take advantage of domain knowledge and the original problem structure, their heuristics are often good at exploiting the hidden structure of the CNF formula and quickly focusing the search toward solutions or toward early detection of unsatisfiability.

5 Experimental Results

We now present an empirical evaluation of FznTini. To obtain a comprehensive picture, we enlist the entire set of benchmarks and examples distributed with MiniZinc [9]. This amounts to the 21 groups of problems listed in Table 1. The "perfsq" (perfect square) and "warehouses" problems in the examples group are scalable, and we have created additional, progressively larger instances and placed these two problems in their own groups of 10 instances each. Also, "2DBinPacking" contains 500 instances divided into 10 classes, and "nsp" (nurse scheduling problem) contains 400 instances divided into 4 classes; due to limited computing resources, we only use the first class of each.

Apart from "examples," which are a mixture of various types of problems, 12 of the groups are satisfaction problems, and 8 are optimization problems. Many of these problems involve a large number of constraints and complex integer arithmetic. For example, the largest "curriculum" instance involves 66 courses with various numbers of credits to be assigned over 12 periods "in a way 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," and the maximum academic load for all periods is minimized (this instance was solved by FznTini in 10.98 seconds, and could not be solved by the other solvers in 4 hours; see below).

The two solvers used for comparison are G12/FD, the FD solver developed by the G12 project [11] and distributed with MiniZinc, and a solver based on translating FlatZinc models to Gecode programs, described in [8] and available for download at the Gecode Web site [14]. Like FznTini, both these solvers assume that MiniZinc models have been converted to FlatZinc. We also note that G12/FD determines its search strategy based on annotations given by the user in the model files, and uses its default strategy (first-fail) in their absence. Experiments were run on a computer cluster featuring two types of CPUs, Intel Core Duo and AMD Athlon 64 X2 Dual Core Processor 4600+, both running Linux at 2.4GHz with 4GB of RAM. Each run of a solver on an instance was given a 4-hour time limit.

The overall results are shown in Table 1. For each solver and benchmark group we report the number of instances solved and the time spent on the solved instances. Time for converting MiniZinc to FlatZinc is common to all solvers, and not included (it ranges from a split second to a few seconds per instance).

It's clear that FznTini solves significantly more instances than the other two solvers, 263 vs. 103 and 178 out of a total of 488, indicating the robustness and versatility of universal Booleanization. It's also interesting to note that each solver appears to have its own "specialties": FznTini was good at curriculum design (curriculum), nurse scheduling (nsp-1-14), warehouse planning (warehouses), and some mathematical puzzles (bibd, perfsq), and relatively good at job shop scheduling (jobshop); G12/FD was good at linear equations under an all-different constraint (alpha) and the n-queens problem (queens); Gecode was good at car sequencing (carseq),[3] linear equations (eq), truck scheduling (trucking), and some other mathematical puzzles (golomb, magicseq).

Overall, these results suggest that universal Booleanization offers a relatively well-rounded, competitive solution to constraint solving, and is a viable alternative where other techniques might fail. The question we are interested to

[3] Some of the solutions generated by Gecode for carseq contain all zeros, which appear to be possibly incorrect, but we have not been able to formally verify it.

Table 1. Performance of FznTini vs. G12/FD and Gecode (4-hour time limit; times on solved instances only, in seconds except in bottom row).

Problem	No. of Inst.	FznTini		G12/FD		Gecode	
		Solved	Time	Solved	Time	Solved	Time
2DBinPacking-1	50	9	2843.46	7	2447.65	7	44.13
alpha	1	1	1.65	1	0.10	1	239.20
areas	4	4	0.69	4	0.71	4	0.04
bibd	9	8	16.17	8	757.72	6	197.48
cars	79	1	3.34	1	0.15	1	0.01
carseq	82	44	99403.70	2	3.58	82	66.65
curriculum	3	3	12.76	2	13.17	0	—
eq	1	1	49.92	1	0.18	1	0.00
examples	18	18	2076.74	18	1557.62	18	2.87
golfers	9	3	6278.30	4	12.88	6	1297.26
golomb	5	4	2030.23	5	323.54	5	10.35
jobshop	73	19	50294.40	2	1764.65	2	31.6
kakuro	6	6	0.17	6	1.10	6	0.01
knights	4	4	0.78	4	390.79	4	1.01
magicseq	7	4	9939.32	7	172.12	7	9.19
nsp-1-14	100	99	1800.36	1	3.97	0	—
perfsq	10	10	548.41	4	4350.19	5	2024.85
photo	1	1	0.08	1	0.20	1	0.00
queens	6	5	4168.79	6	90.68	3	0.33
trucking	10	9	14747.10	10	196.48	10	86.52
warehouses	10	10	671.71	9	2266.44	9	221.83
Total	488	263	54.14 hrs	103	3.99 hrs	178	1.18 hrs

explore—on what types of problems SAT might excel—now has an empirical answer in these results, but we look forward to a continued exploration, where an analytical answer may be sought in detailed analyses of the structure of the problems and their actual constraints.

Finally, it's worth mentioning that the examples group contains a purely Boolean instance, "wolf-goat-cabbage," where the results confirm that FznTini does retain the advantage of SAT on problems of a Boolean nature: This instance was solved by FznTini in 0.02 second, G12/FD in 1553.18 seconds, and Gecode in 2.47 seconds.

6 Related Work and Conclusion

A different approach to homogeneous treatment of constraints has been recently explored, also with MiniZinc as the source language but with linear programs as the target language for translation [15]. Interestingly, this approach attempts to do almost the opposite of Booleanization: The Boolean variables are turned into integers (with a domain of $\{0, 1\}$) and Boolean constraints, along with non-linear integer constraints, all into integer linear constraints. In the experiments presented, the result of this linearization is saved in the input format of the CPLEX solver, and solved by CPLEX. Unfortunately, we have learned from the authors of [15] that their linearization program is currently unavailable due to recent changes in the language (Cadmium) in which it was written.

In conclusion, we have presented the first translation of a constraint modeling language to SAT, and using a large set of benchmarks have shown that it can outperform traditional constraint solvers. Our Booleanization uses a fixed, somewhat basic encoding largely based on a binary representation of integers. We expect our results to motivate the study of other encoding methods that are suitable for Booleanization of heterogeneous constraint models, as well as hybridizations of different techniques for constraint solving, part of which we shall undertake as our own future work.

Acknowledgements

This work has been carried out with the support of members of the G12 project at NICTA. NICTA is funded by the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

References

1. Bailleux, O., Boufkhad, Y.: Efficient CNF Encoding of Boolean cardinality constraints. In Rossi, F., ed.: CP. Volume 2833 of Lecture Notes in Computer Science., Springer (2003) 108–122
2. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. [16] 827–831
3. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation **2** (2006) 1–26
4. Bailleux, O., Boufkhad, Y., Roussel, O.: A translation of pseudo Boolean constraints to SAT. Journal on Satisfiability, Boolean Modeling and Computation **2** (2006) 191–200
5. Silva, J.P.M., Lynce, I.: Towards robust CNF encodings of cardinality constraints. [17] 483–497
6. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. In Benhamou, F., ed.: CP. Volume 4204 of Lecture Notes in Computer Science., Springer (2006) 590–603
7. Hawkins, P., Lagoon, V., Stuckey, P.J.: Solving set constraint satisfaction problems using ROBDDs. Journal of Artificial Intelligence Research **24** (2005) 109–156
8. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard CP modelling language. [17] 529–543
9. G12: MiniZinc distribution, version rotd-2008-03-03. http://www.g12.csse.unimelb.edu.au/minizinc/.
10. Huang, J.: A case for simple SAT solvers. [17] 839–846
11. Stuckey, P.J., de la Banda, M.J.G., Maher, M.J., Marriott, K., Slaney, J.K., Somogyi, Z., Wallace, M., Walsh, T.: The G12 project: Mapping solver independent models to efficient solutions. [16] 13–16
12. SAT: The Annual SAT Competitions. http://www.satcompetition.org/.
13. Katsirelos, G., Bacchus, F.: Generalized nogoods in CSPs. In Veloso, M.M., Kambhampati, S., eds.: AAAI, AAAI Press / The MIT Press (2005) 390–396
14. Schulte, C., Lagerkvist, M., Tack, G.: Gecode. http://www.gecode.org/.
15. Brand, S., Duck, G.J., Puchinger, J., Stuckey, P.J.: Flexible, rule-based constraint model linearisation. In Hudak, P., Warren, D.S., eds.: PADL. Volume 4902 of Lecture Notes in Computer Science., Springer (2008) 68–83
16. van Beek, P., ed.: Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings. In van Beek, P., ed.: CP. Volume 3709 of Lecture Notes in Computer Science., Springer (2005)
17. Bessiere, C., ed.: Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings. In Bessiere, C., ed.: CP. Volume 4741 of Lecture Notes in Computer Science., Springer (2007)