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

A Constraint-Based Local Search Backend for MiniZinc

Gustav Björdal · Jean-Noël Monette · Pierre Flener · Justin Pearson

the date of receipt and acceptance should be inserted later

Abstract MiniZinc is a modelling language for combinatorial problems, which can then be solved by a solver provided in a backend. There are many backends, based on technologies such as constraint programming, integer programming, or Boolean satisfiability solving. However, to the best of our knowledge, there is currently no constraint-based local search (CBLS) backend. We discuss the challenges to develop such a backend and give an overview of the design of a CBLS backend for MiniZinc. Experimental results show that for some MiniZinc models, our CBLS backend, based on the OscaR/CBLS solver, is able to give good-quality results in short time.

Keywords Constraint-Based Local Search · MiniZinc

1 Introduction

Solving combinatorial problems is a difficult task and no single solver can be universally better than all other solvers. Hence, when facing a problem, it is useful to be able to model it once and run several solvers to find the best one. MiniZinc [25] is a technology-independent modelling language for combinatorial problems, which can then be solved by a solver provided in a backend. There are many backends, based on technologies such as constraint programming (CP), integer programming, or Boolean satisfiability solving. However, to the best of our knowledge, there is currently no constraint-based local search (CBLS) backend. While most MiniZinc backends are just a parsing interface in front of the underlying solver, things are not as straightforward in the case of CBLS. We discuss the challenges to develop such a CBLS backend and give an overview of the design of a backend based on the OscaR/CBLS solver [9]. Our backend is hereafter called `fzn-oscar-cbls` and is publicly available from `https://bitbucket.org/oscarlib/oscar/src/?at=fzn-oscar`. A preliminary version of `fzn-oscar-cbls` has been developed by the first author [6].

The main *contributions* of this paper are:

Uppsala University, Department of Information Technology, 752 37 Uppsala, Sweden
E-mail: FirstName.LastName@it.uu.se

– a description of a CBLS backend for MiniZinc;
– a heuristic to discover the structure of a model that can be used by a black-box local search procedure;
– a black-box local search procedure making use of constraint-specific neighbourhoods;
– a description of how to adapt MiniZinc models to be more suitable to CBLS.

The paper is organised as follows. We start with a description of both MiniZinc and CBLS in Section 2. We give an overview of `fzn-oscar-cbls` in Section 3. We describe the creation of a good CBLS model from the MiniZinc model in Section 4 and our search procedure in Section 5. Experimental results in Section 6 show that, for some MiniZinc models, `fzn-oscar-cbls` is able to give good-quality results in short time. Then, in Section 7, we describe discrepancies between the MiniZinc and CBLS worlds, and give hints on how to write models suitable to CBLS. Related work is discussed in Section 8. We conclude and discuss future work in Section 9.

2 Background

We only present the aspects of both MiniZinc and CBLS that are relevant to our purpose. We first recall some basic notions in combinatorial optimisation.

2.1 Problems and Constraints

A *constrained optimisation problem* (COP) is a formal way to describe a combinatorial optimisation problem. A COP is comprised of a set of decision variables (the unknowns of the problem), a set of constraints, and an objective variable. Each variable has a *domain* in which it can take its value. A *solution* to a COP is an assignment of each variable to a value in its domain so that all constraints are satisfied. An *optimal solution* is a solution for which the value of the objective variable is optimal (minimal or maximal). A *constraint satisfaction problem* (CSP) is a COP where the objective variable is absent (i.e., all solutions are equally good). Without loss of generality, we only talk of COPs hereafter.

In this paper, we define *solving* a COP to be the finding of a best possible solution within a given amount of time. Solving a COP is performed by a *solver* that receives as input a *model* of the COP. Solving is gradual and relative: One can say that a solver A solves a COP better than a solver B if, for instance, A returns a better solution than B within the time limit, or if A returns an optimal solution faster than B.

A *global constraint* is a constraint representing a recurring substructure of COPs (there is no single definition of a global constraint; see, e.g., [4] for an overview). Beside yielding more succinct models, the use of global constraints usually improves the solving of a COP by enabling the use of good decompositions or specialised algorithms inside the solver.

2.2 MiniZinc and Its Tool Chain

In an attempt to have a standard modelling language for combinatorial problems, MiniZinc [25], a solver-independent medium-level modelling language, has been

```
1 include "globals.mzn";
2 int: n;                                  % number of queens
3 array[1..n] of var 1..n: c;             % the column of each queen
4 constraint all_different(c);            % no two queens on ... the same column
5 constraint all_different([c[i] + i | i in 1..n]); % ... the same up−diagonal
6 constraint all_different([c[i] - i | i in 1..n]); % ... the same down−diagonal
7 solve satisfy;
```

Fig. 1 A MiniZinc model for the n-queens problem.

designed. MiniZinc supports most of the usual modelling constructs, such as sets, arrays, user-defined constraints and functions, and decision variables of Boolean, integer, float, and integer-set types. A `solve` statement defines the type of problem (satisfaction, minimisation, or maximisation) and the objective variable (if any). MiniZinc comes with a library of solver-independent declarative decompositions of its global constraints and allows solver-specific decompositions of those global constraints. MiniZinc allows parametric models with instance data provided in separate data files. We will talk of a MiniZinc *instance* for the combination of a parametric model with an associated data file, as well as for a non-parametric model. MiniZinc supports annotations of variables, constraints, and `solve` statements. The annotations are not part of the model but give extra information that a solver may exploit or ignore. Predefined annotations are `search` on a `solve` statement to describe a branching strategy, and `defines_var(`x`)` on a constraint to inform that the constraint *functionally defines* the variable x.

Figure 1 presents a MiniZinc model for the classical n-queens problem. The goal is to place n queens on an $n \times n$ chessboard so that no queen can attack another queen. Variable `c[`i`]` denotes the column of the queen in row i. The model makes use of the AllDifferent global constraint, which states that all its argument expressions must take different values.

2.2.1 FlatZinc

MiniZinc is paired with the low-level language FlatZinc, which is a small subset of MiniZinc without complex expressions (e.g., loops or function calls). To solve a MiniZinc instance, it must first be transformed into a FlatZinc model by a process called *flattening* [24]. The resulting FlatZinc model is then presented to a *backend*, which encapsulates a solver. During flattening, new constraints and (functionally defined) variables can be introduced. Flattening produces solver-specific models. In particular, each global constraint is replaced by its decomposition, or is kept as is if it has no decomposition. In the latter case, we talk of a *native* global constraint.

Figure 2 presents excerpts of a FlatZinc model obtained by flattening, for $n = 8$, the MiniZinc model of Figure 1 for a backend in which AllDifferent is a native global constraint. The variables `X_01` to `X_16` in lines 1 to 16 are introduced during flattening to represent each instance of the expressions `c[i] + i` and `c[i] - i` in the MiniZinc model. The introduced variables are defined by the constraints in lines 21 to 36, as indicated by the `defines_var` annotations.

```
 1 var  0..7: X_01;
 2 var -1..6: X_02;
   [...]
15 var 8..15: X_15;
16 var 9..16: X_16;
17 array [1..8] of var 1..8: c;
18 constraint all_different_int([X_01, X_02, X_03, X_04, X_05, X_06,X_07,X_08]);
19 constraint all_different_int([X_09, X_10, X_11, X_12, X_13, X_14,X_15,X_16]);
20 constraint all_different_int(c);
21 constraint int_lin_eq([-1, 1], [X_01, c[1]],  1) :: defines_var(X_01);
22 constraint int_lin_eq([-1, 1], [X_02, c[2]],  2) :: defines_var(X_02);
   [...]
35 constraint int_lin_eq([-1, 1], [X_15, c[7]], -7) :: defines_var(X_15);
36 constraint int_lin_eq([-1, 1], [X_16, c[8]], -8) :: defines_var(X_16);
37 solve satisfy;
```

Fig. 2 Excerpts of a FlatZinc model resulting from flattening, for $n = 8$, the MiniZinc model of Figure 1. The INT_LIN_EQ$([a_1, a_2], [x_1, x_2], b)$ constraint holds if and only if $a_1 \cdot x_1 + a_2 \cdot x_2 = b$, where the a_i and b are constants and the x_i are variables.

2.2.2 The MiniZinc Challenge

Since 2008, solvers can compete in the annual MiniZinc Challenge [34,35]. For each challenge, a collection of 100 MiniZinc instances is gathered and used to compare solvers. After each challenge, the results and the instances are published and can in turn be used to further benchmark new solvers and solving technologies.

2.2.3 Existing Backends

There are many backends for MiniZinc and they use various solving technologies. Some of them are based on Gecode [16] (CP), OR-Tools [30] (CP), Opturion CPX [29] (CP/lazy-clause generation), SCIP [1] (mixed integer programming), fzn2smt [7] (SAT modulo theories), iZplus [15] (a hybrid of CP and local search), and sunny-cp [3] (a portfolio of solvers). With the exception of sunny-cp, most backends are parsers translating FlatZinc models into constructs of the underlying solver in a straightforward manner. However, sunny-cp extracts features of the model to decide which solver(s) to use.

To the best of our knowledge, iZplus [15] is the only MiniZinc backend using some form of local search: solutions found by its CP solver are modified by some form of local search to find neighbouring solutions. Unfortunately, very few details are available on this local search procedure.

2.3 Constraint-Based Local Search

Local search (see, e.g., [21]) is a family of search procedures in which all variables are assigned from the start and the assignment is iteratively modified until some stopping criterion is met, such as reaching a time or iteration limit. The modification from one assignment to the next one usually involves only a few variables and is called a *move*. A move is usually picked among a set of candidate moves, called a *neighbourhood*. Some well-known local search procedures are called tabu

search, simulated annealing, and variable neighbourhood search (see [21] for an overview).

Local search is incomplete, meaning that it is unable to prove unsatisfiability or optimality, or to conclude having found all solutions to a problem. Instead, local search methods are often able to find good-quality solutions in a short amount of time, including for very large problem instances that complete methods cannot handle in useful time. Local search often uses randomisation, hence several executions of the same search procedure on the same instance can produce different results.

Constraint-based local search (see [40]) adapts ideas of CP to local search: a declarative modelling language allows the programmer to define a problem in terms of variables, invariants, constraints, and an objective variable. An *invariant* maintains the value of one or more variables (the *output* of the invariant) to be equal to some function of the values of other variables (the *input*). Invariants are stated declaratively and the solver takes care of maintaining them incrementally upon modification of the input variables, say upon a move. The variables appearing in an invariant need not be decision variables of a COP. They can also be introduced by the solver to maintain auxiliary information.

As in CP, one can use the constraints to guide the CBLS procedure towards a good solution. There are however a number of key differences with CP. First, as in local search, variables are assigned a value at all time. Second, constraints can be handled in three different ways:

- **Implicit constraints** are always satisfied during search. An implicit constraint is maintained satisfied through the use of a neighbourhood that only contains moves that keep the constraint satisfied. The initial assignment is also created to satisfy the implicit constraint.
  For example, an AllDifferent(x) constraint can be made implicit by initially assigning all variables in x to different values, and, during search, performing moves on x that either swap the values of two variables or assign a variable to an unused value. Other neighbourhoods are given in Section 5.1.
- **One-way constraints** are also always satisfied during search. A one-way constraint is maintained satisfied by setting one or more of its variables to be the output of an invariant whose inputs are its other variables. We say that the output variables of the invariant are *defined* by the one-way constraint.
  For example, the constraint $a + b = c$ can be made a one-way constraint in any one of three ways. Either c is maintained to be $a + b$, or a is maintained to be $c - b$, or b is maintained to be $c - a$.
- **Soft constraints** are constraints that do not have to be satisfied during search. Instead, the *violation* of a soft constraint is a constraint-specific measure of how violated it is. The violation is zero when the constraint is satisfied and positive otherwise. The violation of a constraint is maintained by an invariant defining an introduced variable. All soft constraints must be satisfied in any solution.
  For example, the violation of the constraint $a \leq b$ can be defined to be $\max(0, b - a)$: the violation is 0 if $a \leq b$ and $b - a$ otherwise.

All constraints can be soft but only the ones representing a functional dependency can be one-way. All constraints can theoretically be made implicit but in practice only a few implicit constraints can be handled, because designing a neighbourhood of only solutions would be equivalent to solving the problem.

```
1  val n = 8
2  val init = RandomPermutation(1..n)
3  var c = [Var(1..n,init.next()) | i in 1..n]
4  var cpi = [Invariant(c[i] + i) | i in 1..n]
5  var cmi = [Invariant(c[i] - i) | i in 1..n]
6  AllDifferent(cpi)
7  AllDifferent(cmi)
8  while(violation > 0){
9      val i1 = selectOneOf(1..n)
10     val i2 = selectOneOf(1..n)
11     swapValues(c[i1],c[i2])
12 }
```

Fig. 3 Pseudo-code of a CBLS model for the 8-queens problem.

In addition to the model, CBLS solvers usually require the modeller to write a search procedure (e.g., a tabu search). To this end, CBLS solvers provide ways to query the values of variables and the violations associated with constraints and variables, in order to help drive the search towards (good) solutions.

Figure 3 shows pseudo-code of a CBLS model to solve the n-queens problem with $n = 8$. Line 3 creates an array of variables: the constructor `Var(`d`,`v`)` creates a variable with domain d and initial value v. Lines 4 and 5 introduce sixteen one-way constraints to define the `cpi` and `cmi` variables through invariants (corresponding to the expressions inside the constraints on lines 5 and 6 of Figure 1). The constraints on lines 6 and 7 are soft constraints. The constraint (of line 4 of Figure 1) stating that all `c[`i`]` should take a different value is here implicit: the initial assignment is a permutation of the values 1 to n (lines 2 and 3), and the moves are defined to swap the values of two variables `c[`i`]` (line 11). A basic search procedure is given on lines 8 to 12: while the sum of the violations of the two soft constraints is larger than zero (line 8), two rows are selected randomly (lines 9 and 10) and the columns of the queens in these rows are swapped (line 11). When given the FlatZinc model of Figure 2, our backend creates the same CBLS model as the one in Figure 3, but with a different search procedure.

3 Designing a CBLS Backend

The implementation of a CBLS backend for MiniZinc presents several challenges. Some arise because CBLS is a more recent field than, say, constraint programming or integer programming. Others are caused by implicit assumptions made in the MiniZinc world (be it by the designers of the language or by its users). In this work, we focus on presenting a backend that works with existing MiniZinc models without modification. In Section 7, we will briefly discuss how one can modify MiniZinc models or add annotations that would help a CBLS backend.

We developed our CBLS backend, `fzn-oscar-cbls`, on top of the CBLS part of the OscaR library [9]. OscaR [31] is a collection of open-source software, written in Scala, for operations research. It includes, among others, a CP solver, a wrapper to linear programming solvers, and a CBLS solver here referred to as OscaR/CBLS. Other CBLS solvers exist, such as Comet [40], Kangaroo [26], and the Adaptive Search library [8]. Oscar/CBLS is, to the best of our knowledge, the only publicly

available and actively maintained CBLS solver that is complete enough for our purposes. While our code is specific to OscaR/CBLS, the ideas developed in this paper are generally applicable to other CBLS solvers.

When given a FlatZinc model, parsed using a parser generated with Antlr [32], `fzn-oscar-cbls` performs the following tasks:

1. The FlatZinc model is simplified and variable domains are tightened.
2. Each constraint is classified as soft, implicit, or one-way.
3. The actual CBLS model is created.
4. Search is performed on the CBLS model and solutions are printed as they are found.

These tasks are explained in Sections 4 (tasks 1 to 3) and 5 (task 4).

Our backend is accompanied by a library of global constraint decompositions to be used during flattening to FlatZinc. In particular, a number of global constraints are handled natively: AllDifferent, (Sub)Circuit, counting constraints (e.g., GlobalCardinality, Among), Inverse, Maximum, Minimum, and Cumulative. For the other global constraints, we use the solver-independent decompositions of MiniZinc, sometimes because they are good enough for our purpose, but mainly due to their absence in OscaR/CBLS. We are currently working on extending OscaR/CBLS to improve the global constraint support. In its current state, our backend only supports integer and Boolean variables and constraints. It would be straightforward to extend it to set variables as OscaR/CBLS supports them.

The source code of `fzn-oscar-cbls` is available from `https://bitbucket.org/oscarlib/oscar/src/?at=fzn-oscar`. The executable is currently available from `http://www.it.uu.se/research/group/astra/software` and will be part of the next release of OscaR.

4 Structuring the CBLS Model

We now describe how the FlatZinc model is transformed and analysed to obtain a suitable CBLS model (tasks 1 to 3 of Section 3).

4.1 Simplification

In the first task, unary constraints are propagated, to reduce the domains of their variables, and removed from the model. Some simple binary constraints are also propagated. This process can reduce the domains of some variables to singletons, turning them into constants. Hence, more constraints can become unary and the propagation is repeated until no more de-facto unary constraints can be removed. This task is necessary to avoid unbounded domains that may appear in naïve models. In the current version of the backend, this propagation is implemented in an ad-hoc way, hence we only implemented propagation for some constraints that appear often. In future versions of `fzn-oscar-cbls`, we plan to use the CP part of OscaR to perform propagation of all constraints until fix-point as this might drastically reduce the size of the search space.

Figure 4 gives an example a of FlatZinc model where propagation largely reduces the domains. In this model, the domains of the variables are initially unbounded, as denoted by the keyword `int` on lines 1485 to 1487. Using the unary

```
    [...]
1485 array [1..192] of var int: bout;
1486 array [1..192] of var int: buf;
1487 array [1..192] of var int: c;
    [...]
2990 constraint int_le(0, bout[3]);
    [...]
3181 constraint int_le(0, buf[2]);
3182 constraint int_le(0, buf[3]);
    [...]
3375 constraint int_le(bout[3], 2500);
    [...]
3749 constraint int_le(buf[2], 10000);
3750 constraint int_le(buf[3], 10000);
    [...]
4133 constraint int_lin_eq([1, -1, 1, 1], [bout[3], buf[2], buf[3], c[3]], 4000);
    [...]
```

Fig. 4 Excerpts of the FlatZinc model resulting from flattening the `wwtpp.mzn` model with the `ex02000_2400_100.dzn` data file from the 2010 MiniZinc Challenge.

constraints on lines 2990 to 3750, our current constraint propagation reduces the domains of the `bout` variables to 0..2500 and the domains of the `buf` variables to 0..10000. By propagating all constraints until fix-point, it is in this case also possible to reduce the size of the domain of the `c` variables by propagating constraints such as the `int_lin_eq` constraint on line 4133. Here the domain of `c[3]` can be reduced to $-8500..14000$.

4.2 Constraint Classification

The second task is concerned with the main challenge in implementing a CBLS backend, namely to decide how to translate each constraint of the FlatZinc model into a CBLS constraint. Indeed, there are up to three possibilities for each constraint. In theory, one could use only soft constraints, but this generally leads to very poor performance. On the contrary, we argue that it is important to have as much structure as possible by using one-way and implicit constraints. However, there are some limitations of one-way and implicit constraints: A variable can only be defined by one one-way constraint; and an implicit constraint cannot be defined on variables that are defined by a one-way constraint or that appear in another implicit constraint. When alternatives exist, it is not clear how to choose which constraints are made implicit or one-way in order to solve best the FlatZinc model. We describe here our heuristic approach to this challenge.

First, we try to greedily discover as many one-way constraints as possible. We execute the following steps in order:

1. All constraints annotated with `defines_var` in the FlatZinc model are made one-way constraints.
2. Starting from the objective variable, if any, we explore the model to turn as many of the constraints as possible into one-way constraints, as described in Algorithm 1.

Algorithm 1 FindOneWayConstraintsFromObjective

```
1: Set all variables to non-visited
2: Let Q be a FIFO queue containing initially only the objective variable
3: while Q is not empty do
4:     Let v be the first variable in Q and dequeue it
5:     if v is not defined and v can be defined by some constraint c then
6:         Set c to be a one-way constraint defining v
7:     if v is defined by some constraint c then
8:         Enqueue all non-visited input variables of c into Q and set them visited
```

3. For each variable v that is not yet defined by a one-way constraint, by order of decreasing domain size, we pick a constraint c on v in order to make c a one-way constraint defining v. The constraint c must functionally define a unique variable, namely v. For instance, the constraint $\max(x, y) = z$ only functionally defines z. But the constraint $x + y = z$ functionally defines each of x, y, and z from the other two variables, and would not be considered in this step. A variable for which there exists no such constraint is skipped in this step.
4. For each variable v that is not yet defined by a one-way constraint, by order of decreasing domain size, we pick a constraint c on v in order to make c a one-way constraint defining v. The constraint c must functionally define v but, in this step, it is not required to functionally define only v. The constraint c must not have been made one-way yet. A variable for which there exists no such constraint is skipped in this step.
5. The one-way constraints are topologically sorted based on the following digraph: each one-way constraint is a node; there is an edge from a constraint a to another constraint b if the variable defined by a appears in b. For each cycle, one constraint is made non-one-way. We pick a constraint whose defined variable has the smallest domain, ties being broken by taking the constraint with the largest number of input variables being defined by another invariant. This step ensures that no cycle is formed by the invariants. While cycles can be accepted in the static graph of invariants as long as they are absent from the dynamic graph (see [40, page 96] for details), we forbid cycles altogether to keep things simple. Finding the minimum number of one-way constraints to remove to make the graph acyclic is known as the minimum feedback arc set problem and is known to be NP-hard [22]. Hence we do not try and find the smallest number or even approximate it.

In the current version of `fzn-oscar-cbls`, we only consider one-way constraints that define one variable. However, there are constraints that can have several defined variables. This is for instance the case of the Sort and GlobalCardinality (with variable cardinalities) constraints. Trying to make those constraints one-way requires changes to our heuristic and is left as future work.

Next, we try and find implicit constraints. We currently only support a small number of implicit constraints, namely AllDifferent, GlobalCardinality with non-variable cardinalities, LinearEquality (called `int_lin_eq` in FlatZinc, see Figure 2) with unit coefficients, Circuit, and Subcircuit. For each such constraint c, by decreasing order of its number of variables, if c is not defined over any variable that is defined by a one-way constraint or that is in another implicit constraint, then c is made implicit. All remaining constraints are classified as soft.

Our heuristic approach to the classification of constraints gives priority to finding one-way constraints rather than implicit constraints. Our rationale is that it seems better to reduce the search space (by defining variables with invariants) than to structure the search space with specific neighbourhoods.

This greedy heuristic can have negative effects. For instance, considering LinearEquality constraints over Boolean variables as one-way proved to hinder the solving of some problems (e.g., some *vrp* instances of the MiniZinc Challenge 2011). Also, in general, having a greedy approach has the drawback that some bad decisions might remove good choices later on. However, it has the advantage of being fast and seems to give overall satisfactory results. We plan to study further how other heuristics might have an impact on the efficiency of `fzn-oscar-cbls`.

4.3 CBLS Model Creation

In the third task, the CBLS model is created. It comprises the following elements, which will be used in the search procedure (see Section 5):

- A collection of neighbourhoods for the implicit constraints.
- A collection of invariants that maintain one-way constraints.
- The *global violation*, which is the sum of the violations of all soft constraints.
- A collection of *independent variables*, which are the variables that do not appear in an implicit constraint and are not defined by a one-way constraint.
- Optionally, the objective variable.

In OscaR/CBLS, an invariant does not enforce that its output variable must take a value in its domain. Hence, when creating the CBLS model, we must add a soft unary constraint on each variable defined by a one-way constraint to enforce that it takes a value in its domain.

5 Black-Box Local Search

Another challenge for our CBLS backend is to design a suitable black-box search procedure. One cannot get the help of the often used `search` annotation of MiniZinc, as it is specific to the description of CP branching heuristics. Hence it is necessary to design a general-purpose and autonomous search procedure, as OscaR/CBLS does not provide one.

Our search procedure (task 4 of Section 3) is based on the elements of the CBLS model presented in Section 4.3. In Section 5.1, we describe the used neighbourhoods. In Section 5.2, we explain our search procedure.

5.1 Constraint-Based Neighbourhoods

Our search procedure makes use of several neighbourhoods: There are two general-purpose neighbourhoods involving only independent variables, and one specific neighbourhood for each implicit constraint. By design choice, each variable can be involved in at most one neighbourhood. The only variables that do not appear in a neighbourhood are the variables defined by one-way constraints.

Each neighbourhood has two responsibilities: defining possible moves given a current assignment; and creating a randomised initial assignment. Each constraint-specific neighbourhood is defined such that the associated constraint is satisfied by the initial assignment and is maintained satisfied by the moves. In our implementation, neighbourhoods do not return all possible moves to the search procedure but are queried for a (random) best move (see [6] for details). We now review the moves defined by each neighbourhood.

The first general-purpose neighbourhood defines the moves for all independent integer variables. Each move of this neighbourhood is the reassignment of a independent integer variable to another value in its domain. The initial assignment is created by picking a random value in the domain of each independent integer variable.

The second general-purpose neighbourhood defines the moves for all independent Boolean variables. It defines two kinds of moves. Moves of the first kind change the value of one independent Boolean variable; moves of the second kind swap the values of two independent Boolean variables. The initial assignment is created by picking a random value in the domain of each independent Boolean variable.

The two general-purpose neighbourhoods are not mutually exclusive: Both can be used simultaneously during search if the model features both independent integer variables and independent Boolean variables, as is often the case. Those neighbourhoods are also used together with the neighbourhoods specific to any constraints that have been made implicit. We now describe those constraint-specific neighbourhoods.

The neighbourhood for AllDifferent defines two kinds of moves. The first one is a swap between the values of two variables; the second one is a reassignment of a variable to an unused value. Those two kinds of moves ensure that the constraint stays satisfied if it is satisfied initially. The initial assignment is created by picking a random value for each variable within its domain until all values are different.

The neighbourhood for GlobalCardinality with upper and lower bounds on the cardinalities defines two kinds of moves: A swap between the values of two variables, and a reassignment of a variable so that all cardinalities are satisfied. Any swap keeps the constraint satisfied. A reassignment is part of the neighbourhood only if it violates neither the lower bound for the old value of the reassigned variable nor the upper bound for the new value of the reassigned variable. The initial assignment is created by randomly assigning variables so that all lower and upper bounds are respected.

Each move defined by the neighbourhood for Circuit corresponds to the removal of one vertex from the circuit and its insertion at some other point. The neighbourhood for Subcircuit extends this by also allowing removals without corresponding insertion as well as insertions of previously removed vertices. The initial assignment is created by setting the successor of each vertex i to vertex $i+1$ modulo the number of vertices.

The neighbourhood for LinearEquality with unit coefficients defines moves involving two variables: the value of one variable is decreased by some amount and the value of the other variable is increased by the same amount in order to keep the sum constant. The initial assignment is created by first setting each variable to the minimal value of its domain, then randomly increasing some of them until their weighted sum is equal to the required sum.

5.2 Search Procedure

Our search procedure works in two or three consecutive phases: a greedy local search that improves the initial assignment (Section 5.2.1); a first tabu search to find a solution (Section 5.2.2); if there is an objective variable, a second tabu search to find better solutions (Section 5.2.3).

The values of all numerical parameters used below have been chosen through preliminary tests on the MiniZinc Challenge 2010 (see [6] for details). Although our search procedure is sensitive to the parameter values, it is likely that it is even more sensitive to other factors (e.g., which constraints are handled natively). As our backend is in its early stages of design, we consider it to be premature to make a proper sensitivity analysis. In the future, we plan to use automated algorithm configuration tools (e.g., [20]) to tune properly the parameters of the search procedure.

5.2.1 Greedy Search

The greedy search tries to minimise the global violation. It cycles over all the independent variables and for each variable reassigns it to a possibly random value that leads to the smallest global violation. This search stops when either a solution is found, or the overall time limit is exceeded, or three cycles over all the independent variables did not decrease the global violation.

This greedy search cannot change the values of variables in implicit constraints and can be trapped in a local minimum. It nevertheless proved very efficient to decrease initially the global violation, as the initial assignment is randomly created and usually has a huge global violation.

5.2.2 Tabu Search for Satisfaction

The first tabu search tries to find a solution disregarding the objective, if there is one. In tabu search (see, e.g., [17]), each variable that is involved in a move becomes tabu for a number of iterations, called the *tabu tenure*. At each iteration, the search procedure queries each neighbourhood for a random best acceptable move. A move is *acceptable* if it involves at least one non-tabu variable or leads to an assignment with a smaller global violation than the best one encountered so far. A random best move among the ones returned by the neighbourhoods is then performed. In order to have a single tabu tenure while accounting for different kinds of moves, we accept moves involving tabu variables as long as one of them is non-tabu. Otherwise, moves involving several variables would need a shorter tabu tenure than moves involving one variable.

The tabu tenure is adaptive according to the following scheme. Let t be the current tabu tenure, allowed to vary between 2 and some maximum m equal to the number of variables times 0.6. When the search makes no progress for some number of iterations (see [6] for details), t is incremented by $m/10$. It is decremented by 1 when an assignment with a smaller global violation is found. It is reset to its minimum when it has reached m. After reaching m five times, the search is restarted by creating a new random initial assignment. In addition, we add some randomisation by making each modified variable tabu for a random number of iterations between t and $t + m/10$.

The first tabu search stops when a solution is found or the overall time limit is exceeded.

5.2.3 Tabu Search for Optimisation

The second tabu search tries to find solutions with a better objective value. It works very much like the first tabu search, except that it does not try and minimise the global violation, but an aggregate of the global violation and the objective variable. If v is the global violation and o the objective variable, it tries to minimise $\alpha \cdot v \pm \beta \cdot o$, where $\pm$ is $+$ if o is to be minimised and $-$ if o is to be maximised, while α and β are positive integer coefficients. Initially, α and β are both set to 1. They evolve during search such that α is increased if the global violation is positive (i.e., there remain unsatisfied constraints) for a large number of iterations, and β is increased if the global violation is zero (i.e., all constraints are satisfied) but no better solution is found for a large number of iterations (see [6] for details).

The adaptation of the tabu tenure is slightly adjusted from the first tabu search: in addition to the scheme above, the tenure is divided by 2 whenever a better solution is found, and the search is restarted when no better solution is found during a quarter of the overall time limit.

The second tabu search stops when the overall time limit is exceeded or a solution can be proven optimal. This last case happens when the value of the objective variable reaches its relevant bound as given in the model or tightened by task 1, but this is very unusual in practice.

6 Experimental Evaluation

Our aim is to show that a CBLS backend is a viable alternative to solve problems modelled in MiniZinc. We also want to study the effect of using one-way and implicit constraints instead of only using soft constraints. We claim neither that CBLS is better suited than other technologies to solve problems modelled in MiniZinc, nor that `fzn-oscar-cbls` can win a MiniZinc Challenge, but rather that it is a good complement to other backends.

As discussed in Section 5.2, the parameters of our search procedure have not been set in a systematic way. Although it is expected that our search procedure is sensitive to the values given to these parameters, we believe that it is premature to study their effect on the current state of our backend. On the other hand, using one-way and implicit constraints is an important feature of our approach, and the effects of these features are studied here.

We discuss the results of running `fzn-oscar-cbls` on all models of the MiniZinc Challenges from 2010 to 2014. This benchmark is made of 500 instances for 59 parametric models. We ran `fzn-oscar-cbls` with a time-out of 3 minutes per instance, excluding flattening time, repeating each run 5 times. We also ran `fzn-oscar-cbls` without using one-way constraints, without using implicit constraints, and with neither one-way nor implicit constraints. Each of those three additional combinations was run once with a time-out of 3 minutes per instance, excluding flattening time. Experiments were carried out inside a VirtualBox virtual machine with access to one core of a 64-bit Intel Core i7 at 3GHz and 2 GB of RAM.

To give a perspective on the results, we compare the results of `fzn-oscar-cbls` with the results reported after the five MiniZinc Challenges over all categories. This comparison is only indicative as the challenges were run on different hardware and with a time limit of 15 minutes per instance, for only one run per instance.

6.1 Results

Table 1 summarises the results for optimisation problems. This table has the following columns:

- model: The name of the model.
- I: The number of instances of the model used in the MiniZinc Challenges 2010 to 2014. When the same instance has been used twice, it is counted as two separate instances.
- soft+implicit+1way: The results of `fzn-oscar-cbls` with soft, implicit, and one-way constraints, with the following sub-columns:
  - sat: The percentage of runs (over $5 \cdot I$ runs) in which `fzn-oscar-cbls` found a solution.
  - opt: The percentage of runs (over $5 \cdot I$ runs) in which `fzn-oscar-cbls` found the best solution known from the MiniZinc Challenges (often a proven optimal one).
  - $>$: The percentage of backends participating in the corresponding MiniZinc Challenge that performed worse than `fzn-oscar-cbls`, averaged over all $5 \cdot I$ runs. We say that a backend X performs *worse* than another backend Y when Y finds a solution but not X or when both find a solution but Y finds a solution with a better objective value than the one found by X. We do not take into account time and we do not consider that a backend finding an optimal solution without proving its optimality is worse than a backend proving that its solution is optimal.
  - $\geq$: The percentage of backends participating in the corresponding MiniZinc Challenge with respect to which `fzn-oscar-cbls` did not perform worse, averaged over all $5 \cdot I$ runs.
- soft+1way: The results of `fzn-oscar-cbls` without implicit constraints, hence using only soft and one-way constraints. The absence of numbers for some lines in these columns indicates that the full version of `fzn-oscar-cbls` did not use any implicit constraints, so that the results are the same.
- soft+impl: The results of `fzn-oscar-cbls` without one-way constraints, hence using only soft and implicit constraints.
- soft only: The results of `fzn-oscar-cbls` with neither one-way nor implicit constraints, hence using only soft constraints. The absence of numbers for some lines in this column indicates that `fzn-oscar-cbls` without one-way constraints did not use any implicit constraints, so that the results are the same.

Table 2 summarises the results for satisfaction problems, with the same columns as Table 1, minus the *opt* ones, which have no meaning for satisfaction problems.

For the following 10 optimisation problems, `fzn-oscar-cbls` produces good results and is often able to find the optimal or best-known solution: *bacp*, *depot-placement*, *fast-food*, *grid-colouring*, *mario*, *on-call-rostering*, *open-stacks*, *road-cons*, *roster*, and *sugiyama*. In the case of *on-call-rostering* and *road-cons*, it finds the

Table 1 Results for optimisation problems. All values except I are given in percent.

		soft+implicit+1way				soft+1way		soft+impl		soft only	
model	I	sat	opt	$>$	$\geq$	sat	opt	sat	opt	sat	opt
road-cons	5	100	80	72	97	-	-	0	0	0	0
on-call-rostering	5	100	80	48	97	-	-	0	0	0	0
roster	5	100	80	41	93	-	-	0	0	0	0
depot-placement	20	100	79	31	89	100	80	0	0	0	0
open-stacks	5	100	76	48	88	100	40	0	0	0	0
fast-food	10	100	72	13	76	-	-	0	0	0	0
mario	10	100	62	40	74	0	0	0	0	0	0
grid-colouring	10	100	60	66	94	-	-	100	20	-	-
sugiyama	5	100	40	15	49	100	80	0	0	0	0
mspsp	6	100	23	23	42	-	-	100	0	-	-
filters	20	100	5	34	38	-	-	5	0	15	0
celar	5	100	0	69	69	-	-	0	0	-	-
vrp	15	100	0	50	52	-	-	0	0	0	0
mqueens	5	100	0	17	19	-	-	0	0	0	0
stochastic-fjsp	5	100	0	0	1	-	-	20	0	-	-
bacp	20	99	56	20	64	-	-	0	0	0	0
project-planning	6	93	0	17	18	-	-	0	0	0	0
table-layout	5	92	8	14	21	-	-	0	0	0	0
tpp	7	88	2	26	32	0	0	0	0	0	0
league	11	81	0	28	36	-	-	0	0	0	0
smelt	5	80	0	25	33	-	-	0	0	40	0
radiation	10	72	0	14	31	-	-	0	0	0	0
openshop	5	68	0	12	20	-	-	40	0	-	-
ship-schedule	15	68	0	8	10	-	-	0	0	0	0
pattern-set-mining	15	64	0	12	20	-	-	0	0	0	0
still-life-wastage	5	60	0	13	21	-	-	0	0	0	0
prize-collecting	5	60	0	8	21	-	-	0	0	0	0
rcpsp	5	60	0	6	15	-	-	60	0	-	-
fjsp	5	56	0	17	43	-	-	60	0	-	-
ghoulomb	15	22	0	9	56	-	-	0	0	0	0
parity-learning	7	20	17	5	49	-	-	0	0	0	0
liner-sf-reposition.	5	20	0	1	28	20	0	0	0	0	0
train	11	18	0	3	26	-	-	0	0	0	0
cyclic-rcpsp	10	18	0	1	8	-	-	0	0	0	0
rcpsp-max	10	8	0	3	68	-	-	0	0	-	-
carpet-cutting	10	8	0	4	65	-	-	0	0	0	0
spot5	5	4	0	2	5	-	-	0	0	0	0
proteindesign12	5	0	0	0	100	-	-	0	0	-	-
elitserien	5	0	0	0	62	0	0	0	0	0	0
cargo	5	0	0	0	57	-	-	0	0	0	0
l2p	5	0	0	0	25	0	0	0	0	0	0
javarouting	5	0	0	0	15	0	0	0	0	0	0
traveling-tppv	5	0	0	0	9	0	0	0	0	0	0
jp-encoding	5	0	0	0	4	-	-	0	0	-	-
stochastic-vrp	5	0	0	0	2	0	0	0	0	0	0

known optimal solutions on all runs, except for the largest instance of each model. Only 4 backends (out of respectively 28 and 30 entries to the MiniZinc Challenges 2013 and 2014) achieved better results than `fzn-oscar-cbls` on those two problems: `iZplus-free`, `gurobi-free`, `mistral-free`, and `chuffed-free` for *on-call-rostering*; and `iZplus-free`, `iZplus-par`, `Choco-par`, and `OR-Tools-par` for *road-cons*. On *grid-colouring*, `fzn-oscar-cbls` finds an optimal solution on 3 instances out of 5 on all runs, including on the difficult *12_13* instance, which only `chuffed_par`,

Table 2 Results for satisfaction problems. All values except I are given in percent.

		soft+impl+1way			soft+1way	soft+impl	soft only
model	I	sat	$>$	$\geq$	sat	sat	sat
multi-knapsack	5	60	13	83	-	60	-
costas-array	10	50	31	88	40	0	0
fillomino	10	12	1	39	-	0	-
wwtpp-random	5	0	0	100	-	0	0
wwtpp-real	10	0	0	81	-	0	0
amaze2	6	0	0	72	-	0	0
solbat	30	0	0	67	-	0	0
pentominoes	10	0	0	51	-	0	-
nmseq	5	0	0	47	-	0	0
nonogram	15	0	0	44	-	0	-
black-hole	10	0	0	37	-	0	0
rubik	5	0	0	37	-	0	-
amaze	11	0	0	30	-	0	0
rectangle-packing	5	0	0	30	-	0	-

`bumblebee-free`, `g12_lazyfd-free`, and `smt-free` could solve to optimality in the MiniZinc Challenges 2010 and 2011. Interestingly, on *mario*, `fzn-oscar-cbls` finds and proves, in less than 2 seconds on all runs, optimal solutions to instances annotated as hard, while only finding suboptimal solutions for the medium instances.

For 6 other optimisation problems, `fzn-oscar-cbls` is able to find a solution in all runs, but usually a suboptimal one. This is the case for *celar*, *filters*, *mqueens*, *mspsp*, *vrp*, and *stochastic-fjsp*. In the case of *vrp*, it is interesting to note that although the solutions found by `fzn-oscar-cbls` are suboptimal, only integer programming backends, `iZplus-free`, `mistral-free`, and `chuffed-free` are able to achieve better results on most instances. In the case of *filters*, other backends than `fzn-oscar-cbls` either find an optimal solution or do not find any solution.

Regarding satisfaction problems, `fzn-oscar-cbls` finds solutions to *costas-array* instances of size up to 16, 3 *multi-knapsack* instances out of 5, and one *fillomino* instance out of 10. It is unable to find a solution to any of the other satisfaction models within the time-out of 3 minutes.

For the 40 other models, `fzn-oscar-cbls` is unable to find any solution or any good solution within the 3 minute time-out. For four of them, namely *league*, *radiation*, *smelt*, and *tpp*, despite finding suboptimal solutions, `fzn-oscar-cbls` finds a solution in most runs within the 3 minute time-out on some instances that many other backends are unable to solve at all.

Without using one-way constraints, `fzn-oscar-cbls` is unable to find any solution for 51 of the 59 models, hence it is clear that using one-way constraints is important. Regarding implicit constraints, using them clearly improves the results for *mario* and *tpp*, but they have little effect for most other models.

The median time to parse each instance is about 0.6 seconds and the median time to analyse the instance and create the CBLS model (i.e., tasks 1 to 3 from Section 3) is slightly less than 0.6 seconds. A few instances could not be parsed or analysed at all within the 3 minutes, mainly due to memory problems (e.g., the largest *nmseq* instances).

6.2 Analysis

Looking closer at the models for which `fzn-oscar-cbls` does not find any solution or any good solution, we observe that they usually exhibit one or more of the following features:

- global constraints that are not handled natively by `fzn-oscar-cbls` or OscaR/CBLS (e.g., Regular, Diffn);
- very large domains of some decision variables;
- complex logic expressions including disjunctions or implications.

It is not surprising that models using global constraints that are not native cannot be solved well. The obvious solution to this problem is to implement more global constraints in OscaR/CBLS and `fzn-oscar-cbls`. Beside implementing the soft versions of such constraints, it is also useful to implement them as one-way constraints and implicit constraints. The good results observed on the *mario* model are mainly due to the presence in the model of a Subcircuit constraint, for which an appropriate neighbourhood exists.

The fact that models with very large domains are not solved well comes from our search procedure. As each iteration requires evaluating all possible moves to find a best one, very few iterations can be performed within the time limit. To address this, two aspects have to be improved. First, using a CP solver to tighten initially the domains of the variables, as mentioned in Section 4, might actually substantially speed up the search. Second, it might be necessary, upon large domains, to switch to a search procedure that does not require the evaluation of all possible moves at each iteration.

Complex logic expressions in MiniZinc models are translated into reified constraints at the FlatZinc level. A *reified constraint* is a constraint of the form $b \equiv C$, where C is a constraint and b a Boolean decision variable indicating whether C is satisfied or not. Unfortunately, reified constraints are not well-handled in CBLS. Indeed CBLS solvers use the violation of soft constraints to drive the search towards (good) solutions but reified constraints only have a violation of 0 or 1, leading to a very poor discrimination between assignments. A way to overcome this limitation would be to replace, for each reified constraint $b \equiv C$, the indicator variable b by a variable v equal to the violation of C, and to replace all constraints on the indicator variables by appropriate constraints on the violation variables. For instance, a disjunction $b_1 \vee b_2$ would be replaced by the constraint $\min(v_1, v_2) = 0$; see [41] for more details.

Nevertheless, we consider the obtained results very promising. Indeed, while `fzn-oscar-cbls` is unable to find (good) solutions for many instances, it is able to find in a short time optimal solutions for instances that are seemingly hard for most other backends (e.g., the hard instances of *mario*, as well as some *grid-colouring* instances), showing that a CBLS backend for MiniZinc has strengths complementary to other backends.

7 Constraint Modelling Abstractions and Annotations

We now argue that the design of abstractions and annotations of constraint modelling languages has so far been very much geared towards backends performing

systematic search, so that extensions will have to be made to give better support to backends performing local search.

Search Annotations. The most notable MiniZinc annotation, `search`, can be made in an actually formalised language for prescribing how to perform systematic search in the CP style, namely as a combination of a variable selection heuristic with a value selection heuristic, so as to prescribe a sequence of branching guesses to perform under backtracking search. A CBLS backend must ignore such search annotations, as they are incompatible with local search.

Future work consists of defining another formalised annotation language, namely for prescribing how to perform local search, using abstractions like the ones of Comet/CBLS [37].

Symmetry-Breaking Constraints. A common approach when fine-tuning a constraint model is to identify and exploit some if not all of the symmetries of the model, namely by adding constraints that prevent the finding of solutions that are symmetric to the kind of solutions to which the search is geared. This practice is very useful in systematic CP-style search, often giving orders of magnitude of speed-up, but has often been shown counterproductive in local search [10,33], as the presence of symmetric solutions increases the chances of finding solutions.

It would be helpful if symmetry-breaking constraints were marked as such by model annotations, so that a CBLS backend (for MiniZinc) can consider discarding these constraints.

The tool chain for the Essence constraint modelling language [14], namely solver-independent model selection with Conjure [2] followed by solver-specific model flattening with Savile Row [27], features similarities with the MiniZinc tool chain, and Savile Row can also produce MiniZinc models. This tool chain is also currently biased towards systematic CP-style search. Since Conjure can add symmetry-breaking constraints to a model [2], a natural step is to annotate them as such (when producing a MiniZinc model), so that a CBLS backend (for MiniZinc) can consider discarding these constraints.

Implied Constraints. Another common approach when fine-tuning a constraint model is to infer constraints that are logically implied by those of the model but trigger additional propagation under systematic CP-style search, so that solutions are found faster. The impact of implied (or: redundant) constraints on local search performance has not been thoroughly studied yet (to the best of our knowledge), but it has been noted that they can help guide the search more effectively by increasing some violations, thus for instance distinguishing variables that would otherwise be equally violated [40, page 73].

It would be helpful if implied constraints were marked as such by model annotations, so that a CBLS backend (for MiniZinc) can consider discarding these constraints.

Propagation Reduction. Yet another common way to improve constraint models for systematic CP-style search is to replace equality constraints by non-strict inequality constraints or equivalence by implication (see [12]), when it is safe to do so. However, in local search, inequality constraints and implications cannot be made

into one-way constraints. Hence such model transformations usually increase artificially the size of the CBLS search space.

Although one cannot expect to use the same model when targeting different underlying technologies, such model transformations could rather be performed automatically during flattening.

Scheduling Abstractions. There are scheduling constraints that can also be used for non-scheduling problems. For instance, the CUMULATIVE constraint can also be used for packing problems. While this distinction does not matter for systematic CP-style search, one solves scheduling problems in local search by reasoning at the abstraction level of *tasks* and *resources*, which is usually much more efficient than reasoning at the level of constraints and variables (see, e.g., [28]).

Future work consists of adding scheduling abstractions, as in OPL [36] and Comet/CBLS [38], to MiniZinc, so that a CBLS backend for MiniZinc can more easily detect what kind of search to use.

The MiniZinc Challenge. As the MiniZinc Challenge is an important driver in the MiniZinc community, we believe it should be adapted to accommodate CBLS solvers better. Besides the points already raised in the previous paragraphs, there are two other kinds of bias specific to the challenge.

First, the scoring mechanism awards more points to backends *proving* optimality and unsatisfiability. As pointed out in Section 2, CBLS is an incomplete search method and is in general unable to prove optimality or unsatisfiability. If more incomplete solvers enter the challenge, it would be more fair to adapt the scoring mechanism or to create a separate category for incomplete solvers.

Second, local search procedures are almost always defined using randomisation, as that helps increase the chances of finding solutions [21]. Actually, CP-style systematic search is nowadays also increasingly relying on non-deterministic heuristics. However, the MiniZinc Challenge scores each solver on each instance for only *one* run (probably for time reasons), hence a particular solver may score very differently on an instance across different runs. It would be more fair if the scoring mechanism was based on aggregate statistics over at least ten, say, runs per instance.

8 Related Work

In [39], a way is proposed to exploit the semantics of a Comet/CBLS model to generate appropriate neighbourhoods. The modeller must explicitly define invariants for one-way constraints and annotate the other constraints as hard (i.e., implicit) or soft, and can give weights to soft constraints to guide the search better. As we start from FlatZinc models without such annotations, we are bound to guess how constraints can be handled (and we use unit weights for all soft constraints). It is also described how to maintain satisfied some implicit constraints.

Our use of neighbourhoods to maintain implicit constraints satisfied is very close to the notion of a *solution neighbourhood*, which only contains assignments satisfying an associated constraint [19]. The main difference with our work is that the approach in [19] does not require the constraint to be always satisfied, but a move will satisfy the constraint associated with the used solution neighbourhood.

LocalSolver [5] uses a generic black-box local search procedure to solve problems modelled using only Boolean decision variables. Its modelling language also features invariants, whose output can be an integer variable. The user of LocalSolver must explicitly state invariants and constraints, while no such distinction exists in a FlatZinc model.

Using the semantics of a model to generate an appropriate search procedure has not been limited to CBLS. In constraint programming (CP), CP-AS [11] proposes a way to synthesise a search heuristic from the constraints of the model. For scheduling problems, AEON [23] recognises the class of the problem to generate an appropriate search procedure (the modeller specifies the choice of a CP or CBLS search). Several works also integrate several technologies and base the integration on the structure of the model, e.g., [42, 13].

9 Conclusion and Future Work

We have discussed the challenges in designing a CBLS backend for MiniZinc and presented our approach implemented as `fzn-oscar-cbls`. We have shown that such a backend gives good-quality results in a short time for some MiniZinc models.

Experimental results also show that there is room for improvement in our black-box search procedure. The most important task will be to implement additional global constraints. Our approach presents very basic features of autonomous search (see, e.g., [18]). For instance, our tabu tenure is a simplistic example of online tuning and the use of constraint-specific neighbourhoods is related to case-based reasoning. However autonomous solvers go much further in their adaptation to a class of problems or to a particular problem and we plan to integrate some of those methods in `fzn-oscar-cbls`.

This work opens interesting research questions at the modelling level. As discussed in Section 7, existing solver-independent constraint-based modelling languages such as MiniZinc are very much geared towards CP-style solvers. Hence it might be necessary to rethink or extend such languages in order to use the full potential of other technologies (not only CBLS) without resorting to procedural languages.

We plan to enter `fzn-oscar-cbls` in the next MiniZinc Challenge. We hope that this paper will foster research on other MiniZinc backends based on optimisation technologies not currently available for MiniZinc, such as, e.g., genetic algorithms or ant colony optimisation.

Acknowledgements This work is supported by grants 2011-6133 and 2012-4908 of the Swedish Research Council. We thank R. De Landtsheer for his support on OscaR/CBLS. We thank the anonymous reviewers for their constructive and insightful comments.

References

1. Achterberg, T.: SCIP: Solving constraint integer programs. Mathematical Programming Computation **1**(1), 1–41 (2009)
2. Akgün, O., Frisch, A.M., Gent, I.P., Hussain, B.S., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in CONJURE. In: C. Schulte (ed.) CP 2013, *LNCS*, vol. 8124, pp. 107–116. Springer (2013)
3. Amadini, R., Gabbrielli, M., Mauro, J.: Sunny: a lazy portfolio approach for constraint solving. Theory and Practice of Logic Programming **14**, 509–524 (2014)
4. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalogue: Past, present, and future. Constraints **12**(1), 21–62 (2007). The catalogue is at `http://sofdem.github.io/gccat`
5. Benoist, T., Estellon, B., Gardi, F., Megel, R., Nouioua, K.: LocalSolver 1.x: a black-box local-search solver for 0-1 programming. 4OR, A Quarterly Journal of Operations Research **9**(3), 299–316 (2011)
6. Björdal, G.: The first constraint-based local search backend for MiniZinc. Bachelor Thesis in Computer Science, Report IT 14 066, Faculty of Science and Technology, Uppsala University, Sweden (2014). `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-234847`
7. Bofill, M., Palahí, M., Suy, J., Villaret, M.: fzn2smt, a compiler from the FlatZinc language to the standard SMT-LIB language. `http://ima.udg.edu/Recerca/lap/fzn2smt/`
8. Codognet, P., Diaz, D.: Yet another local search method for constraint solving. In: K. Steinhöfel (ed.) SAGA 2001, First International Symposium on Stochastic Algorithms: Foundations and Applications, *LNCS*, vol. 2264, pp. 73–90. Springer (2001)
9. De Landtsheer, R.: Oscar.cbls: a constraint-based local search engine (2012). `https://bitbucket.org/oscarlib/oscar/downloads/Oscar.cbls.pdf`
10. Dotú, I., Van Hentenryck, P.: Scheduling social golfers locally. In: R. Barták, M. Milano (eds.) CP-AI-OR 2005, *LNCS*, vol. 3524, pp. 155–167. Springer (2005)
11. Elsayed, S.A.M., Michel, L.: Synthesis of search algorithms from high-level CP models. In: J. Lee (ed.) CP 2011, *LNCS*, vol. 6876, pp. 256–270. Springer (2011)
12. Feydy, T., Somogyi, Z., Stuckey, P.: Half-reification and flattening. In: J. Lee (ed.) CP 2011, *LNCS*, vol. 6876, pp. 286–301. Springer (2011)
13. Fontaine, D., Michel, L., Van Hentenryck, P.: Model combinators for hybrid optimization. In: C. Schulte (ed.) CP 2013, *LNCS*, vol. 8124, pp. 299–314. Springer (2013)
14. Frisch, A.M., Grum, M., Jefferson, C., Martinez Hernandez, B., Miguel, I.: The design of ESSENCE: A constraint language for specifying combinatorial problems. In: IJCAI 2007 (2007)
15. Fujiwara, T.: iZ based solver for MiniZinc Challenge 2014. `http://www.minizinc.org/challenge2014/description_izplus.txt`
16. Gecode Team: Gecode/FlatZinc. `http://www.gecode.org/flatzinc.html`
17. Glover, F.: Tabu Search Part I. ORSA Journal on Computing **1**(3), 190–206 (1989). DOI 10.1287/ijoc.1.3.190
18. Hamadi, Y., Monfroy, E., Saubion, F. (eds.): Autonomous Search. Springer (2012)
19. He, J., Flener, P., Pearson, J.: Solution neighbourhoods for constraint-directed local search. In: S. Bistarelli, E. Monfroy, B. O'Sullivan (eds.) SAC/CSP 2012, pp. 74–79. ACM Press (2012)
20. Hoos, H.H.: Automated algorithm configuration and parameter tuning. In: Y. Hamadi, E. Monfroy, F. Saubion (eds.) Autonomous Search, pp. 37–71. Springer (2012)
21. Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations & Applications. Elsevier / Morgan Kaufmann (2004)
22. Karp, R.M.: Reducibility among combinatorial problems. In: R.E. Miller, J.W. Thatcher (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press (1972)
23. Monette, J.N., Deville, Y., Van Hentenryck, P.: Aeon: Synthesizing scheduling algorithms from high-level models. In: J.W. Chinneck, B. Kristjansson, M.J. Saltzman (eds.) Operations Research and Cyber-Infrastructure, *Operations Research/Computer Science Interfaces*, vol. 47, pp. 43–59. Springer (2009)
24. Nethercote, N.: Converting MiniZinc to FlatZinc. `http://www.minizinc.org/downloads/doc-1.6/mzn2fzn.pdf`
25. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: C. Bessière (ed.) CP 2007, *LNCS*, vol. 4741, pp. 529–543. Springer (2007). `http://www.minizinc.org/`

26. Newton, M.H., Pham, D.N., Sattar, A., Maher, M.: Kangaroo: An efficient constraint-based local search system using lazy propagation. In: J. Lee (ed.) CP 2011, *LNCS*, vol. 6876, pp. 645–659. Springer (2011)
27. Nightingale, P., Akgün, O., Gent, I.P., Jefferson, C., Miguel, I.: Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In: B. O'Sullivan (ed.) CP 2014, *LNCS*, vol. 8656, pp. 590–605. Springer (2014)
28. Nowicki, E., Smutnicki, C.: A fast taboo search algorithm for the job shop problem. Management Science **42**(6), 797–813 (1996)
29. Opturion Pty Ltd.: Opturion CPX. `http://www.opturion.com/cpx`
30. OR Team at Google: OR-Tools. `https://code.google.com/p/or-tools/`
31. OscaR Team: OscaR: Scala in OR (2012). `https://bitbucket.org/oscarlib/oscar`
32. Parr, T.J.: The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Bookshelf (2007)
33. Prestwich, S.D.: Supersymmetric modeling for local search. In: P. Flener, J. Pearson (eds.) SymCon 2002 (2002). `http://www.it.uu.se/research/group/astra/SymCon02`
34. Stuckey, P.J., Becket, R., Fischer, J.: Philosophy of the MiniZinc challenge. Constraints **15**(3), 307–316 (2010)
35. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The MiniZinc challenge 2008–2013. AI Magazine **35**(2), 55–60 (2014)
36. Van Hentenryck, P.: The OPL Optimization Programming Language. The MIT Press (1999)
37. Van Hentenryck, P., Michel, L.: Control abstractions for local search. In: F. Rossi (ed.) CP 2003, *LNCS*, vol. 2833, pp. 65–80. Springer (2003)
38. Van Hentenryck, P., Michel, L.: Scheduling abstractions for local search. In: J.C. Régin, M. Rueher (eds.) CP-AI-OR 2004, *LNCS*, vol. 3011, pp. 319–334. Springer (2004)
39. Van Hentenryck, P., Michel, L.: Synthesis of constraint-based local search algorithms from high-level models. In: A. Howe, R.C. Holte (eds.) AAAI 2007, pp. 273–278. AAAI Press (2007)
40. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. The MIT Press (2009)
41. Van Hentenryck, P., Michel, L., Liu, L.: Constraint-based combinators for local search. In: M. Wallace (ed.) CP 2004, *LNCS*, vol. 3258, pp. 47–61. Springer (2004)
42. Yunes, T.H., Aron, I.D., Hooker, J.N.: An integrated solver for optimization problems. Operations Research **58**(2), 342–356 (2010)