Declarative Local-Search Neighbourhoods in MiniZinc

Gustav Björdal*, Pierre Flener*, Justin Pearson*, Peter J. Stuckey†, and Guido Tack†
**Department of Information Technology, Uppsala University*, Uppsala, Sweden
`firstname.lastname@it.uu.se`
†*Faculty of Information Technology, Monash University*, Melbourne, Victoria, Australia
`firstname.lastname@monash.edu`

***Abstract*—The aim of solver-independent modelling is to create a model of a satisfaction or optimisation problem independent of a particular technology. This avoids early commitment to a solving technology and allows easy comparison of technologies. MiniZinc is a solver-independent modelling language, supported by CP, MIP, SAT, SMT, and constraint-based local search (CBLS) backends. Some technologies, in particular CP and CBLS, require not only a model but also a search strategy. While backends for these technologies offer default search strategies, it is often beneficial to include in a model a user-specified search strategy for a particular technology, especially if the strategy can encapsulate knowledge about the problem structure. This is complex since a local-search strategy (comprising a neighbourhood, a heuristic, and a meta-heuristic) is often tightly tied to the model. Hence we wish to use the same language for specifying the model and the local search. We show how to extend MiniZinc so that one can attach a fully declarative neighbourhood specification to a model, while maintaining the solver-independence of the language. We explain how to integrate a model-specific declarative neighbourhood with an existing CBLS backend for MiniZinc.**

***Index Terms*—declarative neighbourhood, (constraint-based) local search, modelling**

I. Introduction

The constraint-based modelling language MiniZinc [1] for satisfaction and optimisation problems is independent of solving technologies: it has backends for *complete* solvers of diverse technologies, such as constraint programming (CP), mixed-integer programming (MIP), Boolean satisfiability (SAT), SAT modulo theories (SMT), and hybrids, such as CP with lazy clause generation (LCG), which injects SAT ideas into CP. When their default inference and search turn out to be inefficient, CP and LCG solvers offer elegant ways for the modeller to prescribe declaratively (i) which inference to perform (via a choice of propagators) when pruning the space of candidate solutions, and (ii) which search to perform (via a choice of variable and value selection strategies) when traversing the remaining space of candidate solutions. In MiniZinc, one can do this by providing model annotations, which can be ignored by any backend (in particular MIP, SAT, and SMT ones) but can be used by those that understand them.

The Sweden-based authors are supported by the Swedish Research Council (VR) through Project Grant 2015-04910. Gustav Björdal's visit in March 2017 to the Australia-based authors was supported by the Ericsson Research Foundation under grant FOSTIFT-17:075.

Recently, the first MiniZinc backends for *incomplete* solvers have emerged, such as those of constraint-based local search (CBLS) [2] technology. For example, fzn-oscar-cbls [3] calls the OscaR.cbls solver [4] after categorising the hard constraints of a given MiniZinc model into the three CBLS constraint categories (soft, one-way, and implicit) and adding a black-box search strategy.

Just like CP solvers, CBLS solvers are often used with programmed search strategies particular to the model in question. CBLS search strategies are usually tightly tied to the model. Therefore, in this paper we extend MiniZinc to enable stating part of a local-search strategy within a model.

The contributions and organisation of this paper are as follows. In Section II, we explain the required background on MiniZinc, (constraint-based) local search, and the targeted CBLS backend. In Section III, we introduce the new MiniZinc syntax for declaratively prescribing neighbourhoods as search annotations, as well as new FlatZinc concepts for supporting the underlying representation. In Section IV, we show how to use declarative neighbourhoods within the MiniZinc CBLS backend fzn-oscar-cbls. In Section V, we experimentally show how suitable neighbourhood annotations can accelerate the solving: this helps close the inevitable gap between ad-hoc local search and black-box local search. Finally, in Section VI, we conclude, discuss related work, and outline future work.

II. Background

A. MiniZinc and FlatZinc

The constraint-based modelling language MiniZinc [1] for satisfaction and optimisation problems is independent of solving technologies. Its open-source toolchain contains a *flattener*, which translates a model and instance data into a sub-language called FlatZinc, which is amenable to interpretation by a *backend* that calls the targeted solver (note that 'backend' and 'solver' are not synonyms). The flattener is parametrised by the native capabilities of the targeted technology and solver: the variables of non-native types are *encoded* by variables of native types and possibly consistency constraints thereon; and the constraints with non-native predicates are *decomposed* into sometimes complex or long conjunctions of constraints with native predicates, possibly introducing additional variables.

We now present our running example and a MiniZinc model.

```
 1 include "globals.mzn"; include "local-search.mzn";
 2 set of int: Orders;
 3 set of int: Slabs = Orders; % need max one slab per order
 4 set of int: Colors;
 5 set of int: Capacities;
 6 int: maxColors;
 7 int: maxCapa = max(capacity);
 8 array [Capacities] of int: capacity;
 9 array [Orders]     of int: size;
10 array [Orders]     of Colors: color;
11 array [0..maxCapa] of 0..maxCapa: slack=array1d(0..maxCapa,
12   [min([c | c in capacity++[0] where c >= x]) - x
13     | x in 0..maxCapa]);
14 array [Orders] of var Slabs: placedIn;
15 array [Slabs]  of var 0..maxCapa: load;
16 array [Slabs]  of var 0..maxColors: nColors;
17 constraint bin_packing_load(load, placedIn, size);
18 constraint forall(s in Slabs)(nColors[s]=sum(c in Colors)(
19   exists(o in Orders where color[o]=c)(placedIn[o]=s)));
20 var int: objective = sum(s in Slabs)(slack[load[s]]);
21 ann: search; % here unspecified search annotation
22 solve  ::search   minimize objective;
```

Listing 1. A MiniZinc model for steel-mill slab design.

Example 1: The steel-mill slab design problem [5] has a set of orders, each with a size and colour, and a set of slabs, each taking one of a set of capacities. The principal decisions are to assign orders to slabs, such that at most `maxColors` colours are used in orders on any one slab, and all orders assigned fit on the slab. The aim is to minimise the total slack in the slabs (empty slabs have zero slack). A MiniZinc model corresponding to the one in [6] is in Listing 1. □

B. (Constraint-Based) Local Search

Local search (e.g., [7]) initialises and iteratively modifies the *current assignment*, which maps each variable to a value, called its *current value*, in its domain. The *initial assignment*, with the *initial values*, is built under some amount of randomisation. At every iteration, a two-step search *heuristic* is followed. First, several *candidate moves*, giving a few variables new values and thereby tentatively modifying the current assignment, are *probed* for how much closer they bring the current assignment to a feasible assignment and how much they improve the value of the objective function, if there is one. The set of probed candidate moves (or, equivalently, the set of tentative assignments they reach) is called the *neighbourhood*, which is said to be *explored*, and its elements are called *neighbours*. A candidate move can be non-*valid*, for example because it gives a variable a new value not in its domain. Second, among the valid candidate moves, the heuristic picks one under some amount of randomisation and actually *commits* to it, yielding the new current assignment. A *meta-heuristic* is used to escape local optima of the objective function. Together, the neighbourhood, heuristic, and meta-heuristic form a local-search *strategy*.

In *constraint-based local search* (CBLS) [2], [4], a CP-style declarative model is coupled with a user-defined local-search strategy. For each built-in constraint predicate, predefined *violation functions* can be used for probing candidate moves. A CBLS model has two categories of *explicit constraints*: *soft constraints* are treated as penalty functions and search will try to satisfy them by minimising the penalty, while *one-way constraints* (such as `p <== x * y` in OscaR.cbls syntax [4], and referred to as *invariants* in [2]) cannot be violated by candidate moves. In the example, the controlled variable `p` cannot undergo a move, since its value is maintained by the solver to be the product of the variables `x` and `y`, which can undergo moves. An *implicit constraint* in a CBLS model is satisfied by the initial assignment and preserved by all committed moves: this can be done by instantiating and using a *constraint-specific neighbourhood* [3]. For each constraint of a problem, a CBLS modeller must choose whether to make it soft, one-way, or implicit, thereby meshing search aspects into the model, making it unsuitable for complete solvers.

C. A Local-Search Backend to MiniZinc

The fzn-oscar-cbls [3] backend to MiniZinc calls the OscaR.cbls solver [4] after categorising the constraints of a given MiniZinc model into the three CBLS constraint categories (soft, one-way, and implicit) by using the following three-step *structure identification scheme* (for a full description of the scheme see [3]).

First, it identifies each constraint of the MiniZinc model that will be made one-way in the OscaR.cbls model that is being generated. Each one-way constraint is said to *control* the variable it functionally determines.

Second, it identifies each remaining constraint of the MiniZinc model that will be made implicit, because fzn-oscar-cbls provides a built-in constraint-specific neighbourhood for it. Each such neighbourhood is said to *control* its variables.

Third and finally, it makes all remaining constraints of the MiniZinc model soft in the OscaR.cbls model.

The *black-box local search* performed by fzn-oscar-cbls works as follows. Regarding the search heuristic, each implicit constraint corresponds to a constraint-specific neighbourhood and all uncontrolled variables are put into a built-in non-constraint-specific neighbourhood, whose candidate moves assign any of its controlled variables a new value within its domain. Note that an important design decision is that every variable that is not controlled by a one-way constraint participates in exactly one neighbourhood. At every iteration, a random best candidate move is picked from the union of all neighbourhoods. A tabu search [8] meta-heuristic is used.

Example 2: For the steel-mill slab design problem of Example 1, the black-box search categorises the constraints of lines 18–19 as one-way, controlling the `nColors` variables. The **`bin_packing_load`** constraint of line 17 is replaced by its standard decomposition in MiniZinc, as fzn-oscar-cbls does not support this global-constraint predicate directly, and also categorised as one-way, controlling the `load` variables. No implicit constraint is identified. The uncontrolled `placedIn` variables are put into a built-in non-constraint-specific neighbourhood with assignment candidate moves, which the black-box search explores. □

III. Declarative Neighbourhoods in MiniZinc

After describing in Section III-A the new MiniZinc syntax for declarative neighbourhoods (but not for heuristics or meta-heuristics) in order to get part of a local-search strategy, we discuss in Section III-B the FlatZinc implementation issues.

A. Extended MiniZinc Syntax for Declarative Neighbourhoods

In our extended MiniZinc syntax,[1] a candidate *simple move* is stated using the infix operator `:=` or `:=:` for assignment and swap respectively. The *assignment* `x := v` assigns `v` to variable `x`, where `v` can itself be a variable and must be of the same type as `x`. The *swap* `x :=: y` exchanges the current values of variables `x` and `y` of the same type. The candidate *compound move* μ_1 `/\` μ_2 performs the moves μ_1 and μ_2 in parallel. For example, `Xs[i] := Xs[Xs[i]] /\ Xs[j] := Xs[i] /\ Xs[Xs[i]] := Xs[j]` is a 3-exchange move (e.g., [7]) when `Xs[i]`$\neq$`j` and `Xs[Xs[i]]`$\neq$`j`.

If a candidate move is not valid, then it cannot be committed: a *valid* candidate move neither accesses an out-of-bounds array index, nor gives a variable a value outside its domain, nor gives all variables their current values, nor transforms the current assignment so that it violates a *post-condition*, which is any constraint satisfaction problem (CSP) over the variables of the MiniZinc model, stated by **`ensuring`**`(`*PostCondition*`)`, and is attached to the candidate move using the infix `/\` operator. For example, `Xs[i] := y /\` **`ensuring`**`(`**`all_different`**`(Xs))` is only valid if index `i` is within the index set of array `Xs`, (the current value of) `y` is different from the current value of `Xs[i]` but in its domain, and all values in `Xs` are distinct after assigning (the current value of) `y` to `Xs[i]`.

A *neighbourhood* is a set of candidate moves, stated by **`moves`**`(`*Generators* **`where`** *PreCondition*`)(`*CandidateMove* `/\` **`ensuring`**`(`*PostCondition*`))`, similar to the MiniZinc syntax for generator calls, e.g. **`forall`**`(`*Generators* **`where`** *Condition*`)(`*Constraint*`)`. Here, the **`where`** clause can be any CSP over the variables of the MiniZinc model and states a *pre-condition* that must be satisfied by the current assignment for a move to be valid. For example, **`moves`**`(i` **`in`** **`index_set`**`(Xs), v` **`in`** `V` **`where`** `v > Xs[i])(Xs[i] := v)` denotes the candidate moves where a variable `Xs[i]` is assigned a value in set `V` that is larger than its current value. The *compound neighbourhood* ν_1 **`union`** ν_2 denotes the union of neighbourhoods ν_1 and ν_2.

By default, a solver picks the initial values of the variables. However, an initialisation post-condition **`initially`**`(`*PostCondition*`)`, which can be any CSP, can be attached to the neighbourhood using the infix `/\` operator. An initial assignment must satisfy the initialisation post-condition, but can be infeasible with regards to the actual MiniZinc model. This is crucial, for example, when using a constraint-specific neighbourhood, whose candidate moves (i) require the underlying constraint to be satisfied by the current assignment and (ii) preserve its satisfaction: that constraint is both a pre-condition and a post-condition on the candidate moves, but it then need not be stated (using **`where`** and **`ensuring`** respectively) and hence not be dynamically checked; it is up to the neighbourhood designer to ensure that all candidate moves preserve its satisfaction.

[1]New keywords are typeset in purple, for those viewing the paper in colour.

```
function ann:
  all_different_neighborhood(array [int] of var int: Xs)
    ::neighborhood_definition =
  moves(i, j in index_set(Xs) where i < j)
       ( Xs[i] :=: Xs[j] )
  union
  moves(i in index_set(Xs), v in dom_array(Xs)
    where not member(Xs, v))
    ( Xs[i] := v ) /\
  initially(all_different(Xs));
```

Listing 2. Constraint-specific neighbourhood for **`all_different`**`(Xs)`.

```
function ann: hard_steelmill() ::neighborhood_definition =
  moves(i in Orders, s in Slabs
    where slack[load[placedIn[i]]] > 0 /\
          size[i]+load[s] <= maxCapa)
    ( placedIn[i] := s /\
      ensuring(nColors[s] <= maxColors) ) /\
  initially(forall(o in Orders)(placedIn[o] = o));
```

Listing 3. Hard neighbourhood for steel-mill slab design.

Example 3: Consider a constraint-specific neighbourhood for **`all_different`**`(Xs)`: if all variables of the array `Xs` are initialised to distinct values, then the candidate moves are either to swap the values of any two variables in `Xs`, or to assign any variable in `Xs` any value not assigned to any variable in `Xs`, as shown in Listing 2. □

For a constraint-specific neighbourhood, the initialisation post-condition is always the constraint itself; a conjunctive initialisation post-condition is given in the following example, where we continue from Example 1.

Example 4: Two neighbourhoods for the steel-mill slab design problem are described in [6]: a hard neighbourhood that initialises to a feasible assignment and only has candidate moves to feasible assignments, and a soft neighbourhood that has candidate moves to infeasible assignments.

More specifically, the hard neighbourhood constructs an initial feasible assignment by placing one order per slab and has candidate moves to "all the assignments where all orders but one remain unchanged and the remaining order is placed in another slab without violating the capacity and color constraints of this slab" [6]. They use a semi-greedy search heuristic for the hard neighbourhood: at every even-numbered iteration, an order is removed from a slab with a positive slack; and at every odd-numbered iteration, an order is removed from a slab with the highest slack. While we argue that the search heuristic is orthogonal to the neighbourhood, we can include part of this heuristic in a neighbourhood by stating, as a pre-condition, that only orders from slabs with a positive slack can be removed. This revised neighbourhood can be stated in MiniZinc as shown in Listing 3.

```
function ann: soft_steelmill() ::neighborhood_definition =
  moves(i, j in Orders
    where (size[i] != size[j] \/ color[i] != color[j])
          /\ i < j)
    ( placedIn[i] :=: placedIn[j] )
  union
  moves(i in Orders, s in Slabs
    where load[s] > 0 /\ slack[load[placedIn[i]]] > 0)
    ( placedIn[i] := s ) /\
  initially(global_cardinality(placedIn,
               [s | s in Slabs where s <= nItems div 2],
               [maxColors | o in Orders]));
```

Listing 4. Soft neighbourhood for steel-mill slab design.

The soft neighbourhood initialises by placing two random orders on every slab (leaving some empty) and has candidate moves "assigning an order to a new slab and swapping two orders" [6]. Unlike in [6], where candidate moves "assigning a singleton order to an empty slab and swapping two singleton orders" are not considered, we here exclude moves that either swap two orders of the same size and colour, or remove an order from a slab with zero slack, or move an order to an empty slab. Note that this last pre-condition will decrease the number of used slabs over time. Our experiments showed that these conditions were more appropriate under the black-box (meta-)heuristic of fzn-oscar-cbls. This revised neighbourhood is shown in Listing 4. □

As seen in Listings 2 to 4, a neighbourhood is defined as a function that is of return type **ann** and is annotated as a **neighborhood_definition** using the infix `::` operator. A neighbourhood can be made model-independent by taking variables and parameters of the model as arguments, as seen in Listing 2. However, if a neighbourhood is written as part of a model, then it can refer directly to variables and parameters of the model, as seen in Listings 3 and 4.

The neighbourhood to explore at each iteration is stated in an annotation **use_neighborhood**(*Neighbourhood*) to the **solve** keyword of a model.

Example 5: To invoke the neighbourhood of Listing 3, we add `search =` **use_neighborhood**`(hard_steelmill())` to Listing 1. □

B. Extended FlatZinc Syntax

A MiniZinc model and instance data are compiled into FlatZinc in order to communicate a model instance to a backend. FlatZinc is a low-level subset of MiniZinc, comprised of only parameter, variable, and constraint declarations. All expressions in a MiniZinc model are flattened such that each sub-expression is defined by a constraint. Note that the structure identification scheme of fzn-oscar-cbls essentially un-flattens the FlatZinc instance during its analysis.

In order for a solver to use a declarative neighbourhood, we need, for the first time, to be able to transmit code via FlatZinc to the solver so that the latter can execute it during solve time. To achieve this, we extend FlatZinc with the notion of flat functions. A *flat function* takes the form:

```
function ann: FunctionName(Arguments) [Annotations] =
let {
  FlatZinc constraint and variable declarations
} in Annotation;
```

A flat function is a legitimate function in MiniZinc [9], so we preserve that FlatZinc is a subset of MiniZinc. The body of a flat function is defined using existing FlatZinc and its result is an annotation that can take any number of arguments of any type. The semantics of a flat function is defined through its own annotations and the context it is called in. The main advantage of this approach is that since the body of a **let** expression is defined in terms of FlatZinc declarations, existing backends can already parse the body of a flat function.

Normally, during flattening, generator-call expressions are unrolled. For example, **sum**`(i` **in** `Idx)(Xs[i])` yields `Xs[`**min**`(Idx)] + ⋯ + Xs[`**max**`(Idx)]`, which is then further flattened. However, we do not wish to unroll **moves** expressions since this would defeat the purpose of leaving the neighbourhood exploration to the solver. Therefore, **moves** expressions are instead treated as a special kind of MiniZinc expression. Specifically, we extract the components of a **moves** expression so that they can be passed to the backend by using flat functions.

We translate a neighbourhood **moves**(*Generators* **where** *PreCondition*)(*CandidateMove*/\ensuring(*PostCondition*)) in five steps into two flat functions: one, here called `fun_moves`, encodes *Generators*, *PreCondition*, *CandidateMove*, and a call to the other, here called `fun_ensuring`, which encodes *PostCondition*.

First, the generator variables are extracted from *Generators*: a *generator variable* is declared like a decision variable but annotated with **defines_generator**. For example, the generator `i` **in** `S` is compiled into **var** `S: i ::`**defines_generator**. Generator variables are different from normal FlatZinc decision variables: they are iterated over by the solver during neighbourhood exploration.

Second, *PreCondition* is flattened as usual and added to the body of `fun_moves`, together with the generator variables. Note that generator variables are distinguished by their annotation from the decision variables introduced when flattening *PreCondition*. Also note that generator variables are treated as decision variables, rather than parameters, when flattening *PreCondition*, hence the resulting constraints can be checked given any assignment of the generator variables.

Third, *CandidateMove* is flattened. Each candidate simple move is translated into an annotation, denoted by M_i below, namely **assign**`(x,v)` in FlatZinc for `x := v` in MiniZinc, but **assign_array**`(x,i,v)` for the common case `x[i] := v`, and **swap**`(x,y)` for `x :=: y`, but **swap_array**`(x,i,y,j)` for `x[i] :=: y[j]`. Recall that *CandidateMove* can be compound; either way, it is encoded as an array of annotations for its n candidate simple moves, which becomes part of the returned annotation of `fun_moves`. Any sub-expressions of *CandidateMove* are flattened as usual and added to the body of `fun_moves`.

Fourth, *PostCondition*, if it exists, is flattened as usual and becomes the body of `fun_ensuring`, whose returned annotation is **void**, meaning that it will not be used by anyone. The `fun_ensuring` function may need to take generator variables as arguments. In our prototype implementation we do not yet support arguments to FlatZinc functions, so we break the scoping rules to access any generator variables (defined in `fun_moves`) that may be used in `fun_ensuring`.

Fifth, `fun_ensuring` is used in the second argument of the returned annotation of `fun_moves`:
neighborhood_and`(`$[M_1, \ldots, M_n]$`,`
ensuring`(fun_ensuring(`*Generators*`)))`.

The **initially** expression of a declarative neighbourhood is translated separately, analogously to the **ensuring** expression, into a flat function `fun_init`, which is connected with the m flattened **moves** expressions using the annotation

```
function ann: fun_hard_steelmill_moves() =
let {
  var Orders: i ::defines_generator;
  var Slabs:  s ::defines_generator;
  var Slabs: pi;
  constraint int_element_var(i, placedIn, pi)
     ::defines_var(pi);
  var int: li;
  constraint int_element(pi, load, li) ::defines_var(li);
  var int: sl;
  constraint int_element(li, slack, sl) ::defines_var(sl);
  constraint int_le(1, sl);
  var int: si;
  constraint int_element(i, size, si) ::defines_var(si);
  var int: ls;
  constraint int_element_var(s, load, ls)
     ::defines_var(ls);
  constraint int_lin_le([1,1,-1], [si,ls,maxCapa], 0);
} in neighborhood_and([assign_array(placedIn, i, s)],
      ensuring(fun_hard_steelmill_ensuring(s)));
function ann: fun_hard_steelmill_ensuring(var Slabs: s) =
let {
  var int: ns;
  constraint int_element_var(s, nColors, ns)
     ::defines_var(ns);
  constraint int_le(ns, maxColors);
} in void;
function ann: fun_hard_steelmill_initially() =
let {
  constraint int_element(1, placedIn, 1);
  constraint int_element(2, placedIn, 2);
  ...
} in void;
...
search = use_neighborhood(neighborhood_declaration(
     [fun_hard_steelmill_moves()],
     initially(fun_hard_steelmill_initially())));
```
Listing 5. FlatZinc instance of the neighbourhood & model in Listings 3 & 1.

neighborhood_declaration([fun_moves$_1$(), ..., fun_moves$_m$()], **initially**(fun_init())).

Example 6: A FlatZinc instance of the neighbourhood in Listing 3 and the model in Listing 1 is shown in Listing 5. Note that a constraint that functionally defines a variable `x` may be annotated during flattening with `::`**`defines_var`**`(x)` in order to communicate this to the backend. □

IV. Using Declarative Neighbourhoods in CBLS

When using a declarative neighbourhood in a MiniZinc CBLS backend, the backend must choose initial values for the variables in the **`initially`** expression (Section IV-A) as well as probe candidate moves and evaluate their pre- and post-conditions (Section IV-B). Furthermore, declarative neighbourhoods should also integrate with the rest of the backend, such as its (meta-)heuristic and structure identification scheme (Section IV-C and IV-D). We now show how we have implemented this in fzn-oscar-cbls [3]. Of course, other backends may use different implementations and offer more or less flexibility.

A. Initialisation for Declarative Neighbourhoods

Initial values for the variables in a declarative neighbourhood are chosen using OscaR.cp, the CP solver of OscaR [10], thereby exploiting the felicitous co-existence of CP and CBLS solvers within the OscaR toolkit. If a declarative neighbourhood has an **`initially`** expression, with a conjunction of constraints and possibly introduced variables, then the corresponding CP model is created. At the start of the CBLS search, and possibly upon restarts, the CP solver is run for such a neighbourhood, with the conflict ordering [11] variable and value selection strategy plus randomised tie breaking for value selection, in order to get random feasible initial values for the variables in the **`initially`** expression. The advantage of using a CP solver for initialisation is that the MiniZinc modeller can really give *any* CSP as the initialisation post-condition, provided it is solvable in reasonable time. Variables not occurring in the **`initially`** expression are initialised by the default initialisation process of fzn-oscar-cbls.

B. Probing of Declarative Neighbourhoods

A neighbourhood is explored by iterating over all value tuples of the generator: if a tuple satisfies the pre-condition, then the candidate move is probed and checked for validity (recall that validity includes satisfaction of the post-condition).

The constraints of a pre- (or post-)condition are clustered in a constraint system [2] that the CBLS backend can query, separately from the constraint system of the actual model, for violation before (or while) probing candidate moves. By using the constraint syntax of OscaR.cbls to express the pre- and post-conditions, we really support *any* CSP as a pre- or post-condition. Also, since the violation of constraints and constraint systems is incrementally maintained by OscaR.cbls, the pre- and post-conditions can be checked very efficiently and will benefit from future improvements to OscaR.cbls.

C. Declarative Neighbourhoods in Black-Box Search

Declarative neighbourhoods are implemented in fzn-oscar-cbls by extending its base class of neighbourhoods and implementing the non-optional methods of its interface, namely initialisation and exploration. This allows a declarative neighbourhood to be used interchangeably with a built-in one.

Any variable that is controlled by neither a declarative neighbourhood nor a one-way constraint is put into built-in neighbourhoods of fzn-oscar-cbls, as determined by the latter's structure identification scheme: this means that built-in neighbourhoods can be used together with declarative ones.

We integrate a declarative neighbourhood more tightly with the black-box search of fzn-oscar-cbls by using two extra features that are specific to this backend but not necessary for implementing a declarative neighbourhood: we enable a declarative neighbourhood to better work together with (i) the heuristics of fzn-oscar-cbls, by enabling the neighbourhood to be explored in either a first-improving fashion or exhaustively, and (ii) the tabu meta-heuristic of fzn-oscar-cbls, by extending the validity check of a move to account for tabu variables.

D. The Controlled Variables of Declarative Neighbourhoods

As explained in Section II-B, a CBLS backend must decide for each variable in a MiniZinc model if it is functionally defined (and thus controlled) by a one-way constraint, or if it is controlled by a neighbourhood. Also, as noted in Section II-C, a design decision of fzn-oscar-cbls is that a variable can only be controlled by one built-in neighbourhood. This design decision is essential for the correct interaction of the built-in neighbourhoods and the rules of the structure identification scheme [3]. Therefore, a variable is now to be controlled by either a one-way constraint, or a built-in neighbourhood, or a declarative neighbourhood. In the structure identification scheme, when determining how each variable is controlled, a declarative neighbourhood takes precedence over a one-way-constraint, which in turn takes precedence over a built-in neighbourhood. *Hence the generated soft, one-way, and implicit constraints of the OscaR.cbls model can actually change if a declarative neighbourhood is present.*

V. Experimental Evaluation

We evaluate declarative neighbourhoods and the implementation discussed in Section IV by two sets of experiments. First, we return to our running example, steel-mill slab design, and show that a declarative neighbourhood overall improves the performance (Section V-A). Second, we augment models of the MiniZinc Benchmark[2] with simple problem-specific declarative neighbourhoods (Section V-B). We do not expect to surpass or even reach the best-known objective values for any of the selected problems, as handcrafted methods tend to outperform general methods, such as our declarative neighbourhoods. Instead, our aim is to show the usability and expressive power of declarative neighbourhoods, as well as their benefit over black-box search.

For each instance of each experiment we made ten independent runs with a 600 second timeout each and compute for each instance of each model the average over the ten runs.[3]

A. Published Neighbourhoods for Steel-Mill Slab Design

Three technologies — CP, large-neighbourhood search, and CBLS — are compared in [6] for solving the steel-mill slab design problem of Example 1: they conclude that CBLS is the most suitable and that their so-called hard neighbourhood is better than their soft neighbourhood, both presented in Example 4. As only one (real-world) instance was originally available,[4] but easily solvable to optimality using the models of [6], new instances were derived by changing its number of available slab capacities.[5] The new instances with at most seven capacities appear to be considerably harder than those with more capacities: the minimal total slack is not reached (efficiently) for most of these instances by any of the technologies tried in [6]. However, using integer programming, all of these instances are efficiently solved to optimality in [5].

We ran the model in Listing 1 with the neighbourhood of either Listing 3 or Listing 4. We selected the derived `bench_3_`x instances, with $x \in 0..19$, each with three available slab capacities and thus among the hardest instances tried in [6], [5]. Based on the geometric mean, using the soft neighbourhood improves the total slack by 14% over black-box search, while using the hard neighbourhood improves by 5% over black-box search. Using the soft neighbourhood strictly improves the total slack on 15 of 20 instances, while using the hard neighbourhood strictly improves on 12.

[2]https://github.com/MiniZinc/minizinc-benchmarks

[3]The neighbourhood-annotated models and data collected during the experiments are at https://github.com/GustavBjordal/neighbourhood-experiments.

[4]http://www.csplib.org/Problems/prob038/data

[5]http://becool.info.ucl.ac.be/steelmillslab

```
function ann: gbac_neighborhood()
  ::neighborhood_definition =
  moves(c in courses, p in periods
    where not isUndesirable[c,p])
    ( period_of[c] := p )
  union moves(c1, c2 in courses where c1 < c2)
    ( period_of[c1] :=: period_of[c2] ) /\
  initially(
    forall(c in curricula)(
      global_cardinality_low_up_closed(
        [period_of[i] | i in courses_of[c]],
        [i | i in periods],
        [min_courses | i in periods],
        [max_courses | i in periods])
    ) /\ forall(i in precedences)(
      period_of[precedes[i,1]] < period_of[precedes[i,2]]
    ));
```

Listing 6. Neighbourhood for generalised balanced academic curriculum design.

However, our best reached total slacks are about 1.8 times the best ones found in [6], which uses a *handcrafted* (meta-) heuristic, whereas our grey-box search strategy *generates* a (meta-)heuristic. Also, the best reached total slacks in [6] are about 1.3 times above the proven optima reported in [5].

B. Published MiniZinc Models

We selected published models that we know to be well-suited for local search.

Example 7: The generalised balanced academic curriculum design problem [12] has a set of periods, a set of courses, each belonging to a set of curricula and having a set of prerequisite courses, a course-load interval for each period and curriculum, and preferences on when it is desired not to hold a course. The problem is to decide for each course which period it is taught in, such that all prerequisites and course loads are satisfied, while the workload imbalance of each curriculum and the number of unsatisfied preferences are minimised.

We add the derived array `isUndesirable` to the published model in order to make it easier to look up if teaching a given course in a given period is undesirable. We declare in Listing 6 a neighbourhood that requires an initial assignment satisfying all prerequisite and course-load constraints, and has candidate moves that either assign a non-undesirable period to a course or swap the periods of two courses.

We use all the real-world instances of [12], called `UD1` to `UD10`. Based on the geometric mean, using the neighbourhood improves the objective value by about 13% over black-box search. On 7 of the 10 instances, using the neighbourhood strictly improves the average objective value; on `UD7`, using the neighbourhood backfires and the average objective value is 157% of the value reached using black-box search. □

Example 8: The car sequencing problem [13] has a set of car classes, each class with a set of used options (such as a radio or GPS), each option with an upper bound on how many cars of that option can be produced in a subsequence of given size, and an order stating how many cars of each class to produce. The problem is to find a production sequence for all ordered cars, such that all restrictions on options are satisfied.

```
function ann: car_neighborhood()
  ::neighborhood_definition =
  moves(i, j in steps where i < j)
    ( step_class[i] :=: step_class[j] ) /\
  initially(
    forall(c in classes)(
      count(step_class, c, cars_in_class[c])
    ));
```

Listing 7. Neighbourhood for car sequencing.

The published MiniZinc model contains a variable for each position, called a *step*, in the production sequence, denoting its produced car class. We declare in Listing 7 a neighbourhood that requires an initial assignment where all ordered cars are produced, so that candidate moves swapping two steps keep that initialisation post-condition satisfied. Note that the initial assignment can be a non-solution: it is only required to satisfy the initialisation post-condition.

We use the 73 satisfiable instances of the 79 instances in the MiniZinc Benchmark. Using black-box search on the published model, a solution is found in all but 114 of the $73 \cdot 10 = 730$ runs, and at least one solution was found for 67 instances. When using the neighbourhood of Listing 7, a solution is found in all but 17 runs, and at least one solution was found for 72 instances. On the 67 instances where both the black-box search and the neighbourhood lead to at least one solution, using the neighbourhood finds a solution about 5 times faster.

However, the published model can be reformulated by replacing the constraint conjunction

```
50 constraint forall(c in classes)(
       count(step_class, c, cars_in_class[c]));
```

by the single constraint

```
constraint global_cardinality_low_up_closed(step_class,
    classes, cars_in_class, cars_in_class);
```

whose even higher-level predicate is flattened in MiniZinc in the same standard way as that conjunction, so that such a globalising [14] reformulation normally cannot backfire under any solving technology. Under black-box search, the built-in neighbourhood for `global_cardinality_low_up_closed` is now picked by the structure identification scheme of fzn-oscar-cbls: this neighbourhood performs essentially the same moves as the declarative neighbourhood.

Using the globalised model lead to a solution in all but 90 runs using black-box search, and at least one solution was found for 72 instances.

For these 3 models, a comparison of the average number of solved instances within a given runtime is in Figure 1. □

So we could here improve the performance by just raising the level of the model: this gives more evidence for the importance of high-level modelling. However, low-level modelling can often be compensated for by search annotations, such as our declarative neighbourhoods.

Example 9: Community detection [15] is about determining communities of a network, that is groups of nodes that are more connected to each other than those outside. The semi-supervised version includes constraints that force nodes to be in the same or different communities, and bounds on the size of communities. The aim is to maximise modularity, a measure of "communitiness". Ganji *et al.* [15] demonstrate that CBLS can be much faster than CP on large instances of this problem.

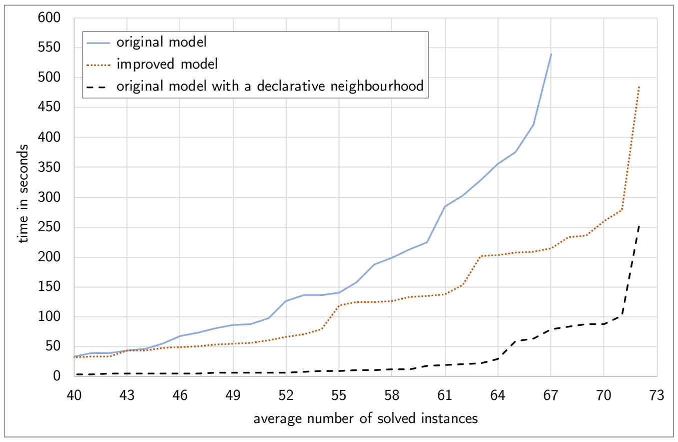

Figure 1. The average number of solved instances (x-axis) for the car sequencing problem within a given runtime (y-axis).

```
function ann: community_neighborhood()
  ::neighborhood_definition =
  moves(i in 1..n, c in 1..k)
    ( x[i] := c ) /\
  initially(
    forall(m in must)(x[ML[m,1]] = x[ML[m,2]]) /\
    forall(c in cannot)(x[CL[c,1]] != x[CL[c,2]]) /\
    global_cardinality_low_up(x, [i | i in 1..k],
      [0 | i in 1..k], [maxsize | i in 1..k]));
```

Listing 8. Neighbourhood for semi-supervised community detection.

The published model [15] has size $O(n^2)$, where n is the number of nodes. We use a more scalable model that is $O(e)$, where e is the number of edges. It uses suitably high-level global-constraint predicates, but (unlike in our globalised car-sequencing model in Example 8) these here cause the structure identification scheme of fzn-oscar-cbls to identify (for black-box search) an inappropriate built-in neighbourhood, namely the one for `global_cardinality_low_up`, which occurs once in the model, but is not constraining. Any declarative neighbourhood takes precedence in the structure identification scheme, and can thus prevent this built-in neighbourhood from being identified: we declare in Listing 8 a neighbourhood that requires an initial assignment that satisfies all constraints of the model and has candidate moves that assign a community to a node.

We select 12 instances of the problem, varying in size from 100 to 1000 nodes. Based on the geometic mean, using the neighbourhood leads to an objective value that is about 7.07 times higher (with a peak at 137.62) than for black-box search, and the reached objective value strictly improves for all instances. However, if we instead replace the mentioned global constraint by its standard decomposition in MiniZinc, then this also prevents the built-in neighbourhood for `global_cardinality_low_up` from being identified, and a neighbourhood similar to our declarative neighbourhood is identified under black-box search: this even gives an overall better performance than we have achieved by adding a declarative neighbourhood to either model. However, we propose an elegant way to bypass the need for de-globalising in the future-work part of Section VI. Indeed, black-box search using the de-globalised model reaches an objective value that is about 1.8 times higher than by the declarative neighbourhood on the original model, and 12.75 times higher than by black-box search using the original model. □

So, interestingly, we here improved performance by either adding a declarative neighbourhood or de-globalising the model. However, our insight for trying the de-globalising reformulation in the first place heavily relies on an intimate knowledge of the structure identification scheme of fzn-oscar-cbls, so that we can force which built-in neighbourhood is picked, thereby avoiding the need for a declarative neighbourhood in this case. Our declarative neighbourhoods give this power to the modeller, without requiring this deep knowledge, turning black-box search into a grey-box search.

VI. Conclusion, Related Work, and Future Work

We have extended MiniZinc with support for declaring a local-search neighbourhood together with a model, thus supporting rapid experimentation with various local-search strategies, and started closing the inevitable gap between ad-hoc local search and black-box local search. The extension broadens the interface between MiniZinc and solvers in order to allow code to be passed. This is relatively straightforward for CBLS solvers, which essentially are efficient incremental expression evaluators.

In Comet [2], the entire search strategy has to be expressed in *procedural* code for each problem; by using closures [16], one can express not only the union of heterogeneous sub-neighbourhoods, similarly to our *declarative* neighbourhoods, but also their exploration. In OscaR.cbls [4], there are a few built-in general-purpose neighbourhoods and a combinator language [17] that allows one to define how neighbourhoods are combined (including our **`union`**) and explored; furthermore, one can define *procedurally* a new neighbourhood by using a common interface. However, our neighbourhoods *declaratively* and separately express only the pre-condition, post-condition, and candidate moves themselves, as opposed to the neighbour(hood) constructs in [16], [2] and [4], [17], where these are interleaved with each other in *procedural* code. Our approach currently cannot express (meta-) heuristics. The *declarative* formalisation of neighbourhoods is also outlined in [18], using predicate logic; however, to the best of our knowledge, there is currently no information on the expressiveness and performance of this approach.

Future work includes creating a MiniZinc library of standard declarative definitions of common neighbourhoods, such as in Listing 2. Then, similarly to global constraint predicates, a backend can provide a native implementation of such a neighbourhood, which would be used instead of the standard definition. Indeed, in Example 9, a declarative neighbourhood was outperformed by a similar built-in neighbourhood, due to overhead. Likewise, a MiniZinc library of annotations for predefined (meta-)heuristics, say to express intensification and diversification (e.g., [7]), will be created.

One drawback of our neighbourhoods is that they cannot have candidate moves on a run-time-dependent set of variables, such as "assign value `v` to *all* variables of current value `w`", which in a bin-packing problem could mean emptying bin `w` into bin `v`. Instead, such a candidate move must be simplified to "assign value `v` to *a* variable of current value `w`". This can be worked around by supporting further simple moves using new annotations, such as `assign_all(Xs,w,v)`.

Finally, we plan on using the same FlatZinc function mechanism to support more complex programmed search for CP solvers, where the CP solver needs to evaluate expressions to determine what branching decision to make next, by accessing views on the current solver state.

Acknowledgements

Many thanks to Kevin Leo for helping with the internals of the MiniZinc compiler, and to the referees for their feedback.

References

[1] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a standard CP modelling language," in *CP 2007*, ser. LNCS, vol. 4741. Springer, 2007, pp. 529–543.
[2] P. Van Hentenryck and L. Michel, *Constraint-Based Local Search*. The MIT Press, 2005.
[3] G. Björdal, J.-N. Monette, P. Flener, and J. Pearson, "A constraint-based local search backend for MiniZinc," *Constraints*, vol. 20, no. 3, pp. 325–345, 2015, `fzn-oscar-cbls` at http://astra.research.it.uu.se/software.
[4] R. De Landtsheer and C. Ponsard, "OscaR.cbls: An open source framework for constraint-based local search," in *ORBEL-27*, 2013, available as http://www.orbel.be/orbel27/pdf/abstract293.pdf.
[5] S. Heinz, T. Schlechte, R. Stephan, and M. Winkler, "Solving steel mill slab design problems," *Constraints*, vol. 17, no. 1, pp. 39–50, 2012.
[6] P. Schaus, P. Van Hentenryck, J.-N. Monette, C. Coffrin, L. Michel, and Y. Deville, "Solving steel mill slab problems with constraint-based techniques: CP, LNS, and CBLS," *Constraints*, vol. 16, no. 2, pp. 125–147, 2011.
[7] H. H. Hoos and T. Stützle, *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.
[8] F. Glover and M. Laguna, "Tabu search," in *Modern Heuristic Techniques for Combinatorial Problems*. Wiley, 1993, pp. 70–150.
[9] P. J. Stuckey and G. Tack, "MiniZinc with functions," in *CPAIOR 2013*, ser. LNCS, vol. 7874. Springer, 2013, pp. 268–283.
[10] OscaR Team, "OscaR: Scala in OR," 2012, http://www.oscarlib.org.
[11] S. Gay, R. Hartert, C. Lecoutre, and P. Schaus, "Conflict ordering search for scheduling problems," in *CP 2015*, ser. LNCS, vol. 9255. Springer, 2015, pp. 140–148.
[12] M. Chiarandini, L. Di Gaspero, S. Gualandi, and A. Schaerf, "The balanced academic curriculum problem revisited," *J of Heuristics*, vol. 18, no. 1, pp. 119–148, 2012, http://satt.diegm.uniud.it/projects/gbac.
[13] M. Dincbas, H. Simonis, and P. Van Hentenryck, "Solving the car-sequencing problem in constraint logic programming," in *ECAI 1988*. Pitman, 1988, pp. 290–295.
[14] K. Leo, C. Mears, G. Tack, and M. Garcia de la Banda, "Globalizing constraint models," in *CP 2013*, ser. LNCS, vol. 8124. Springer, 2013, pp. 432–447.
[15] M. Ganji, J. Bailey, and P. J. Stuckey, "A declarative approach to constrained community detection," in *CP 2017*, ser. LNCS, vol. 10416. Springer, 2017, pp. 477–494.
[16] P. Van Hentenryck and L. Michel, "Control abstractions for local search," in *CP 2003*, ser. LNCS, vol. 2833. Springer, 2003, pp. 65–80.
[17] R. De Landtsheer, Y. Guyot, G. Ospina, and C. Ponsard, "Combining neighborhoods into local search strategies," in *Recent Developments in Metaheuristics*, ser. ORCS, vol. 62. Springer, 2018, pp. 43–57.
[18] S. T. Pham, J. Devriendt, and P. De Causmaecker, "Formalize neighborhoods for local search using predicate logic," in *Data Science meets Optimization*, 2017, https://ds-o.org/images/Workshop_papers/Pham.pdf.