Modelling with Option Types in MiniZinc*

Christopher Mears[3], Andreas Schutt[1,2], Peter J. Stuckey[1,2], Guido Tack[1,3], Kim Marriott[1,3], and Mark Wallace[1,3]

[1] National ICT Australia (NICTA)
[2] University of Melbourne, Victoria, Australia
[3] Faculty of IT, Monash University, Australia

Abstract. Option types are a powerful abstraction that allows the concise modelling of combinatorial problems where some decisions are relevant only if other decisions are made. They have a wide variety of uses: for example in modelling optional tasks in scheduling, or exceptions to a usual rule. Option types represent objects which may or may not exist in the constraint problem being modelled, and can take an ordinary value or a special value $\top$ indicating they are *absent*. The key property of variables of option types is that if they take the value $\top$ then the constraints they appear in should act as if the variable was not in the original definition. In this paper, we explore the different ways that basic constraints can be extended to handle option types, and we show that extensions of global constraints to option types cover existing and common variants of these global constraints. We demonstrate how we have added option types to the constraint modelling language MINIZINC. Constraints over variables of option types can either be handled by transformation into regular variables without extending the requirements on underlying solvers, or they can be passed directly to solvers that support them natively.

1 Introduction

A common feature of complex combinatorial models is that some decisions are only relevant if other decisions are made. Hence some part of the model may be irrelevant dependent on decisions in another part. This is typically modelled by requiring that the "irrelevant decisions" are fixed to some default value, and none of the constraints about the irrelevant decisions are enforced. While it is certainly possible to express such models in traditional modelling languages, it is neither concise nor straightforward.

Many programming languages, including ML, Haskell and Scala, provide an *option type*, to wrap an arbitrary type with an additional "None" value to indicate that the returned value is not meaningful. This paper introduces an extension to the MINIZINC modelling language [14] that uses a similar approach to indicate that a value is irrelevant. We add support for option types in MINIZINC, that is types extended with an additional value $\top$ indicating that a decision was irrelevant. The meaning of constraints

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council. The first author was sponsored by the Australian Research Council grant DP110102258.

and functions on normal types is lifted to option types to create the desired behaviour that variables taking the value $\top$ do not constrain the rest of the model.

Variables of option type, or *optional variables*, are used to model objects (or relationships) that may or may not exist in the problem being modelled. Such a variable may take a value like an ordinary variable, or have an *absent* value. The most common use of optional variables in constraint programming is in modelling the start times of *optional tasks*, tasks in a scheduling problem that may or may not occur.

Example 1. In a flexible job shop scheduling problem a task t can be performed on one of k machines, with a duration d_{tl} if it runs on machine l. A common model for this is to model a single task t with start time S_t and (variable) duration D_t, as well as k optional tasks with optional start times O_{tl} and (fixed) duration d_{tl}. If task t runs on machine l then $O_{tl} = S_t$, $D_t = d_{tl}$ and $O_{tj} = \top, 1 \leq j \neq l \leq k$.

Optional variables are also used to model “exceptional circumstances”.

Example 2. Consider an assignment problem where each of m workers should be assigned to one of n tasks, where there are too many workers for the tasks ($m > n$). A common way of modelling this in constraint programming is using the global constraint `alldifferent_except_0`$([w_1, \ldots, w_m])$ where worker assignments $w_i, 1 \leq i \leq m$ take values in the range $0..n$, where 0 represents that the worker is not assigned to any task. With optional variables this is modelled differently: each variable w_i is an optional variable with domain $1..n$ and $\top$. The $\top$ plays exactly the same role as 0 in the traditional approach, but the constraint is now simply `alldifferent`$([w_1, \ldots, w_m])$ applied to optional variables w_i. Lifting this constraint to work on optional variables will automatically provide the desired behaviour.

Some of the global constraints in the global constraint catalogue [1] in fact arise from the common occurrence of modelling with optional variables. We believe the global constraint catalogue could be simplified by the addition of optional variables and automatic lifting of constraints to optional variables. Optional variables can also make some forms of modelling easier even if they are not required.

Example 3. Consider an assignment problem with m workers and n tasks, but this time $n > m$. If we require that workers who work on tasks adjacent in the list are compatible, we can model this by

$$\forall 1 \leq j \leq n-1.\ \forall 1 \leq i, i' \leq m.\ w_i = j \wedge w_{i'} = j+1 \rightarrow \mathit{compatible}[i, i'] \ ,$$

but this creates a very large and complex set of constraints. Instead we can model this using an `inverse` global constraint, using optional variables. Let $t_j = i$ be the optional variable representing that task j is worked on by worker i, but if no one does the task then $t_j = \top$. We can then model the above constraint using

$$\texttt{inverse}(w, t) \wedge \forall 1 \leq j \leq n-1.\ \mathit{compatible}[t_j, t_{j+1}] \ .$$

If $t_j = \top$ or $t_{j+1} = \top$ then the constraint $\mathit{compatible}[t_j, t_{j+1}]$ will automatically hold.

In the remainder of this paper we formally define option types, and how we can automatically lift the meaning of constraints and functions on ordinary types to their option type extension. We show how with option types we can concisely model common modelling idioms, and extend the expressiveness of the modelling language. The contributions of this paper are:

– A theory for extending a constraint definition over standard types to option types (Section 2.1).
– A standard approach to extending functions from standard types to option types (Section 2.2).
– An extension to MINIZINC to support option types (Section 3).
– An extension of the comprehension syntax of MINIZINC to allow iteration over variable sets and variable conditions (`where` clauses) using option types (Section 3.2).
– An approach to automatically translating models with option types into models without option types (Section 4). This has the advantage of providing support for the new modelling features for all solvers supporting FLATZINC without changes.
– The ability for solvers to specify their support for constraints with option types, so that models using them can make use of efficient variable implementations and propagation algorithms, where the solvers support them (Section 4.4).

2 The Logic of Option Types

Option types are defined using the type constructor `opt` which maps a type to an option type. If T is the set of possible values (excluding $\top$) then `opt` $T = T \cup \{\top\}$.

Modelling for constraint programming (CP) consists of defining relations over decision variables and functionally defined terms on these decision variables. To extend modelling to include option types, we must define how to interpret relations and functions on option types. To be useful for modelling, these definitions should have meanings that are natural to the modeller.

2.1 Lifting Constraints to Option Types

The key requirement for easy use of option types is for the modeller to clearly understand the meaning of constraints involving option types. Here we show how to automatically lift an existing constraint definition that uses standard types to be one defined on option types. The lifting is meant to reflect the implicit understanding that "*if an option type variable x takes the value $\top$ each constraint involving x should act as if x was not part of its definition.*"

Projection Interpretation Converting the implicit understanding into a formal definition, which we denote the *projection interpretation*, leads to the following:

Let $c(x_1,\ldots,x_n)$ be a constraint requiring a non-option variable of type T in the i^{th} position, then if x_i is of type `opt` T, the *extended version of the constraint*, $c^{\mathrm{e}:\{x_i\}}$ for the i^{th} argument is

$$c^{\mathrm{e}:\{x_i\}}(x_1,\ldots,x_n) \Leftrightarrow (x_i \neq \top \wedge c(x_1,\ldots,x_n)) \vee (x_i = \top \wedge \exists x_i . c(x_1,\ldots,x_n)) \ .$$

Thus either the variable x_i takes a non-$\top$ value and the constraint acts as usual, or the variable takes the $\top$ value and the constraint acts as if x_i was projected out.

We can extend this to lift arbitrary numbers of variables as follows. Define $c^{\mathrm{e}:S}$ where $S \subseteq vars(c)$ as c when $S = \emptyset$ and $(c^{\mathrm{e}:S'})^{\mathrm{e}:\{x\}}$ when $S = S' \cup \{x\}$ otherwise. The meaning of the lifting is independent of order; e.g.:

$$c^{\mathrm{e}:\{x\}^{\mathrm{e}:\{y\}}}(x,y,z) = c^{\mathrm{e}:\{y\}^{\mathrm{e}:\{x\}}}(x,y,z)$$

Note that while the lifting operation does distribute over disjunction, it does not distribute over conjunction.

Example 4. When $y = \top$ the constraint $c^{\mathrm{e}:\{y\}}(x,y,z)$ where $c(x,y,z) = x \le y \wedge y \le z$, is equivalent to $x \le z$, whereas $x \le^{\mathrm{e}:\{y\}} y \wedge y \le^{\mathrm{e}:\{y\}} z$ is equivalent to *true*.

Note that pushing negations into relations is not a valid transformation with option types.

Example 5. When $y = \top$ the expression $\neg(x =^{\mathrm{e}:\{y\}} y)$ is equivalent to *false*, while $c_2^{\mathrm{e}:\{y\}}(x,y)$ where $c_2(x,y) \equiv x \neq y$, is equivalent to *true*.

However, this does not further restrict MiniZinc: pushing negations is already invalid for MiniZinc given the relational semantics [6] treatment of partial functions.

Since we will usually lift all arguments of a constraint to option types, we define $c^{\mathrm{p}}(x_1,\ldots,x_n) = c^{\mathrm{e}:\{x_1,\ldots,x_n\}}(x_1,\ldots,x_n)$. The lifted version of equality $x =^{\mathrm{p}} y$ defines a *weak equality*, denoted $x =_w y$, which holds if x equals y or either x or y is $\top$. In order to give names to expressions of option type and support substitutivity we also need the *strong equality*, $x =_{\mathrm{s}} y$ (or just $x = y$), which holds if x and y are identical, e.g. it is false if x takes value $\top$ and y takes a value different from $\top$.

Example 6. Given a constraint $O_{ij} + d_{ij} \le S_j$ between an optional variable O_{ij} and standard variable S_j, then if $O_{ij} = \top$ the constraint is automatically satisfied. Note that it is important to disregard the possible values of the optional variable which are not $\top$ when determining the meaning. If O_{ij} had a domain of $0..10 \cup \{\top\}$ and duration $d_{ij} = 4$ then we do not want to have this constraint force $S_j \ge 4$ even when $O_{ij} = \top$. The projection interpretation does not constrain S_j since O_{ij} can take value $-\infty$ (or any value not larger than $l - 4$ where l is the lowest value in the domain of S_j).

Compression Interpretation An alternate intuition for extending constraints to option types exists for constraints with an n-ary array argument. In this alternate interpretation, the *compression interpretation*, we treat the constraint as if $\top$ values had been removed from the array argument.

Example 7. Given a constraint `alldifferent` on array of optional variables $W = \{w_i \mid 1 \le i \le m\}$, then under the assumption that $w_1 = w_2 = \cdots = w_k = \top, k < m$ then the compression interpretation of $\texttt{alldifferent}^{\mathrm{c}}([\top,\ldots,\top,w_{k+1}\ldots.,w_m])$ is the constraint $\texttt{alldifferent}([w_{k+1},\ldots,w_m])$. Note that for `alldifferent` the compression interpretation and the projection interpretation agree, since $\texttt{alldifferent}^{\mathrm{e}:W}(w)$ is equivalent to $\texttt{alldifferent}([w_{k+1},\ldots,w_m])$ when $w_1 = w_2 = \cdots = w_k = \top, k < m$.

While the projection and compression interpretation often agree, there are constraints where they do not.

Example 8. Consider the constraint $\texttt{sliding\_sum}(l,u,k,v)$ which requires that $l \leq v[i] + \ldots + v[i+k-1] \leq u, \forall 1 \leq i \leq n-k+1$ where n is the length of the array v. The projection interpretation allows a $\top$ value for $v[j]$ if there is some value that keeps all the sliding sums in the right range. For example $\texttt{sliding\_sum}^{\text{p}}(1,2,3,[1,1,\top,1,1])$ holds since it is satisfied for $v[3] = 0$, but $\texttt{sliding\_sum}^{\text{p}}(1,2,3,[1,1,\top,0,\top,1,1])$ does not hold (the only solution would replace each $\top$ by 0 but this fails on the middle 3). Compression eliminates the $\top$ values from the array, changing the values being summed, hence $\texttt{sliding\_sum}^{\text{c}}(1,2,3,[1,1,\top,1,1])$ equals $\texttt{sliding\_sum}(1,2,3,[1,1,1,1])$ and does not hold, while $\texttt{sliding\_sum}^{\text{c}}(1,2,3,[1,1,\top,0,\top,1,1])$ equals $\texttt{sliding\_sum}(1,2,3,[1,1,0,1,1])$, and does hold.

The compression interpretation must be modified when we have a tuple of arrays used to represent an array of tuples.

Example 9. Given a constraint $\texttt{cumulative}(s,d,r,L)$ where s are optional variables, and assuming that $s_1 = s_2 = \cdots = s_k = \top, k < n$, then the constraint under the projection interpretation is equivalent to $\texttt{cumulative}([s_{k+1},\ldots,s_n], [d_{k+1},\ldots,d_n], [r_{k+1},\ldots,r_n], L)$; that is, we treat the constraint as if the tasks whose start time is optional did not exist in the constraint. Note that this is equivalent to a compression interpretation which removes the "corresponding" values of other arrays. Under the assumption that the durations d and resources r are also optional, and assuming that $r_1 = \top, d_2 = \top, s_3 = \top, r_4 = \top, d_5 = \top, s_6 = \top$ then the projection interpretation of the constraint is equivalent to $\texttt{cumulative}([s_7,\ldots,s_n], [d_7,\ldots,d_n], [r_7,\ldots,r_n], L)$.

Given that the projection interpretation is clear for any constraint and usually agrees with the compression interpretation when that makes sense, for MINIZINC we define the projection interpretation as the meaning for lifting a constraint to option types.

2.2 Lifting Functions to Option Types

MINIZINC includes many built-in functions (and operators) and also allows users to define their own functions. Lifting existing functions from standard types to option types is also important, so that the modeller can concisely make use of functions in their models with option types. Again we want to have a clear policy so the modeller understands how functions interact with option types.

First note that mapping functions to relations and using the projection-based lifting does not give us what we want. Consider the constraint $plus(x,y,z)$ defined as $x+y=z$. Then $plus^{\text{p}}(i,\top,j)$ holds for all i and j (the $\top$ takes the value $j-i$), hence we have lost the *functional nature* of the expression $x+y$!

Absorption Lifting A straightforward extension of functions to option types is to treat the absent value as an absorbing element:

$$x \oplus^{\mathrm{a}} y \overset{def}{=} \begin{cases} \top & \text{if } x = \top \\ \top & \text{if } y = \top \\ x \oplus y & \text{otherwise} \end{cases}$$

The absent value can be viewed as "contagious". This definition can transform any function $\oplus : A \times B \rightarrow C$ to $\oplus^{\mathrm{a}} : \texttt{opt}\ A \times \texttt{opt}\ B \rightarrow \texttt{opt}\ C$.

Identity Lifting The intended meaning of the absent value is that it should be ignored wherever possible. With this in mind, binary operations of the form $\oplus : S \times S \rightarrow S$ can be lifted to $\oplus^{\mathrm{i}} : \texttt{opt}\ S \times \texttt{opt}\ S \rightarrow \texttt{opt}\ S$ by the definition:

$$x \oplus^{\mathrm{i}} y \overset{def}{=} \begin{cases} y & \text{if } x = \top \\ x & \text{if } y = \top \\ x \oplus y & \text{otherwise} \end{cases}$$

For these operations the absent value acts as the identity: for $+$ it is zero, for $\wedge$ it is *true*, and so on. When both values are absent, the result of the operation is absent. This definition is the natural conversion from the semigroup $(S, \oplus)$ to the monoid $(S \cup \{\top\}, \oplus^{\mathrm{i}})$. Note we can still use identity lifting for operations like *min* even though its identity element $(+\infty)$ is not in the usual domain of integer variables.

Arithmetic subtraction and division are not associative, and therefore do not form semigroups. The above definition does not make sense for these operators since they do not have left-identities. However these operators do have right-identities, and so identity-lifting can be extended to this case by defining:

$$x \oplus^{\mathrm{r}} y \overset{def}{=} \begin{cases} \top & \text{if } x = \top \\ x & \text{if } y = \top \\ x \oplus y & \text{otherwise} \end{cases}$$

Effectively when we lift a function then an absent value in a position where it has no identity element gives an absent result. For example $3 - \top = 3$ since $\top$ acts as 0, but $\top - 3 = \top$ since there is no identity in this position. Note that the result is absent if and only if x is absent.

Unary Operators and Functions Unary operators and functions $\oplus : S \rightarrow S$ can be lifted naturally to $\oplus^{\mathrm{a}} : \texttt{opt}\ S \rightarrow \texttt{opt}\ S$. Absorption lifting is the only thing that makes sense for a unary function.

Boolean Operations Objects of type `opt bool`, optional Booleans, are not common when modelling with optional variables, since usually optionality is captured within lifted constraints (which only take two-valued truth values). But we can directly create objects of type `opt bool`. For the Boolean operations $\wedge$ and $\vee$ we use identity lifting, and negation $(\neg)$ uses absorption lifting like any other unary operator. This defines a three-valued logic where an absent value has the effect that it neither contributes to satisfying a proposition nor hinders it. For example, both $A \wedge^{\mathrm{i}} \top$ and $A \vee^{\mathrm{i}} \top$ are simply equivalent to A.

This is not a standard three-valued logic, for example while De Morgan's laws still hold, distribution of $\wedge$ over $\vee$, and vice versa do not. Essentially the $\top$ value acts as a context sensitive default value.

Absorption lifting of logical operators corresponds to Kleene's weak (or Bochvar's internal) three-valued logic [10,2]. Unfortunately, absorption lifting does not accord with our intuition as it makes $\top \wedge^{\mathrm{a}} C$ equal to $\top$ rather than C. Note that both of the standard three-valued logics—Łukasiewicz's [18] and Kleene's (strong) three-valued logics—also conflict with our intuition, as they make $\top \wedge \mathit{true}$ equal to $\top$ rather than *true*.

N-ary Functions The identity-lifted functions permit the easy definition of many useful n-ary functions over optional variables. For example:

$$\begin{array}{rlrl}
\Sigma_j x_j &= 0 + x_1 + x_2 + \cdots + x_n & \Sigma^{\mathrm{i}}_j x_j &= 0 +^{\mathrm{i}} x_1 +^{\mathrm{i}} x_2 +^{\mathrm{i}} \cdots +^{\mathrm{i}} x_n \\
\forall j.b_j &= \mathit{true} \wedge b_1 \wedge b_2 \wedge \cdots \wedge b_n & \forall^{\mathrm{i}} j.b_j &= \mathit{true} \wedge^{\mathrm{i}} b_1 \wedge^{\mathrm{i}} b_2 \wedge^{\mathrm{i}} \cdots \wedge^{\mathrm{i}} b_n \\
\exists j.b_j &= \mathit{false} \vee b_1 \vee b_2 \vee \cdots \vee b_n & \exists^{\mathrm{i}} j.b_j &= \mathit{false} \vee^{\mathrm{i}} b_1 \vee^{\mathrm{i}} b_2 \vee^{\mathrm{i}} \cdots \vee^{\mathrm{i}} b_n
\end{array}$$

In these cases, if all variables x_j or b_j are absent then the result is the default zero, *true* or *false* value. That is, these functions have the type `array of opt` $T \rightarrow T$.

The meaning of an n-ary function where some of the arguments are $\top$ is the function with those arguments omitted, as in the compression interpretation for relations. For example, an absent value in a forall will not cause it to be *false*.

In some cases we may wish for an n-ary function of type `array of opt` $T \rightarrow$ `opt` T, where if all argument values are absent the result is also absent. This is the only choice for functions where there is no default value, such as *minimum*.

$$\mathit{minimum}(X) = \min(x_1, \min(x_2, \ldots \min(x_{n-1}, x_n) \ldots))$$

becomes

$$\mathit{minimum}^{\mathrm{i}}(X) = \min{}^{\mathrm{i}}(x_1, \min{}^{\mathrm{i}}(x_2, \ldots \min{}^{\mathrm{i}}(x_{n-1}, x_n) \ldots))$$

Thankfully most builtin functions in MiniZinc are either unary, binary, or n-ary functions resulting from folding binary functions. It is not necessarily obvious how to extend other functions to option types, and hence we do not propose any lifted meaning for these, although we could suggest absorption lifting as the default.

For some models it is convenient to make use of the absorption-lifted form of a function even if it has an identity-lifted form. This can make defining complex interactions of optional variables more natural.

Example 10. Consider the `span` constraint [11] on optional tasks: $\mathit{span}(s_0, d_0, [s_1, \ldots, s_n], [d_1, \ldots, d_n])$ where s_i are optional start times and d_i are durations. The `span` constraint ensures that task 0 starts at the earliest start time of any task $1 \leq i \leq n$ and ends at the last end time, and if none occurs then $s_0 = \top$. The start time constraint is captured by $s_0 =_s \mathit{minimum}([s_1, \ldots, s_n])$. If each of $s_1, \ldots, s_n$ are $\top$ then the *minimum* function forces $s_0 = \top$. In contrast the end time constraint is not captured by $s_0 +^{\mathrm{i}} d_0 =_s \mathit{maximum}([s_1 +^{\mathrm{i}} d_1, \ldots, s_n +^{\mathrm{i}} d_n])$ since even absent start times can contribute to the *max* if the corresponding duration is large. Instead we need $s_0 +^{\mathrm{a}} d_0 =_s \mathit{maximum}([s_1 +^{\mathrm{a}} d_1, \ldots, s_n +^{\mathrm{a}} d_n])$ where $+^{\mathrm{a}}$ is the absorption lifted version of the $+$ function. Only the tasks that occur will contribute to the *maximum*.

3 Using Option Types in MiniZinc

Option types are included in MINIZINC with the addition of a new type constructor `opt`, which maps a type to an option version of the type. An ordinary constraint lifted to option types uses the projection interpretation, and a function lifted to option types uses absorption lifting, except for the binary operators where identity lifting applies (and folds over these operators).

3.1 Basic Modelling with Option Types

Two versions of equality are provided for option types: a strong equality $=_s$, denoted `=`, which ensures that both sides are identical, and a weak equality $=_w$, denoted `~=`, which also holds if either side is $\top$. The actual value $\top$ is represented in models using _ and has polymorphic type `opt $T`. But we suggest avoiding using it directly, and instead provide two (polymorphic) primitive constraints defined on option types:

```
predicate occurs(var opt $T: x) = not absent(x);
predicate absent(var opt $T: x) = (x = _);
```

where *occurs*(x) iff $x \neq \top$ and *absent*(x) iff $x = \top$. Clearly *occurs* is the negation of *absent*. Both are provided for clarity of modelling.

Example 11. A MINIZINC model for flexible job shop scheduling as discussed in Example 1 is shown below. Note that the optional variables are used to give start times for the version of the tasks starting on each machine. The `disjunctive` constraint is the version lifted for option types. The `alternative` constraints are directly defined on option types; they enforce the `span` constraint (see Example 10) as well as ensuring at most one optional task actually occurs. The redundant `cumulative` constraint ensures no more than k tasks run at one time. The aim is to minimize the latest end time.

```
int: horizon;               % time horizon
set of int: Time = 0..horizon;
int: n;                     % number of tasks
set of int: Task = 1..n;
int: k;                     % number of machines
set of int: Machine = 1..k;
array[Task,Machine] of int: d;
int: maxd = max([ d[t,m] | t in Task, m in Machine ]);
int: mind = min([ d[t,m] | t in Task, m in Machine ]);
array[Task] of var Time: S;
array[Task] of var mind..maxd: D;
array[Task,Machine] of var opt Time: O;
constraint forall(m in Machine)
          (disjunctive([O[t,m]|t in Task],[d[t,m]|t in Task]);
constraint forall(t in Task)(alternative(S[t],D[t],
             [O[t,m]|m in Machine],[d[t,m]|m in Machine]));
constraint cumulative(S,D,[1|i in Task],k);
solve minimize max(t in Task)(S[t] + D[t]);
```

Example 12. A MINIZINC model for the problem of assigning m workers to n tasks discussed in Example 2 is shown below. The constraint is simply an `alldifferent` over optional variables.

```
int: n;            % number of tasks
int: m;            % number of workers
array[1..m] of var opt 1..n: w; % task for each worker
constraint alldifferent(w);
solve satisfy;
```

Example 13. Similarly a MINIZINC model for the compatible worker assignment problem discussed in Example 3 is shown below.

```
int: n;            % number of tasks
int: m;            % number of workers
array[1..m,1..m] of bool: compatible;
array[1..m] of var 1..n: w;     % task for each worker
array[1..n] of var opt 1..m: t; % worker for each task
constraint inverse(w,t);
constraint forall(j in 1..n-1)(compatible[t[j],t[j+1]]);
solve satisfy;
```

Example 14. The `span` constraint of Example 10 can be expressed as shown below, where ~+ is a new operator added to MINIZINC to encode absorption lifted addition.

```
predicate span(var opt Time:s0, var int: d0,
            array[int] of var opt Time:s, array[int] of int:d) =
  s0 = min(s) /\
  (absent(s0) -> d0 = 0) /\
  s0 ~+ d0 = max([s[i] ~+ d[i] | i in index_set(s)]);
```

The first line makes use of *strong* equality to ensure that s_0 is absent if each of the tasks in s is absent. The second line of the definition ensures that if the spanning task is absent, then its duration is fixed (to zero). The third line use absorption lifted + (written ~+) to constrain the end time. We can define `alternative` using `span`.

3.2 Extending Comprehension Syntax using Option Types

A significant advantage of the addition of option types for MINIZINC is that it allows us to increase the expressiveness of comprehensions in the language. At present an array comprehension expression of the form $[f(i) \mid i \text{ in } S \text{ where } c(i)]$ requires that S is a fixed set and $c(i)$ is a condition that does not depend on decision variables (has type/mode `par bool`). Once we include option types this can be relaxed.

Suppose $f(i)$ is of type T. We allow $c(i)$ to be dependent on decisions (type/mode `var bool`) by modifying the array comprehension to output an array of `opt` T rather than an array of T, with the interpretation that if $c(i) = \mathit{false}$ then $f(i)$ is replaced by $\top$.

Hence we rewrite $[f(i) \mid i$ `in` S `where` $c(i)]$ as[1]

$$[\ \texttt{if}\ c(i)\ \texttt{then}\ f(i)\ \texttt{else}\ \top\ \texttt{endif} \mid i\ \texttt{in}\ S\ ]$$

Once we have extended the `where` clause to be decision dependent it is straightforward to also support comprehensions where the set S is itself a decision variable. The comprehension $[f(i) \mid i$ `in` S `where` $c(i)]$ where S is a *set decision variable* is equivalent to $[f(i) \mid i$ `in` `ub`(S) `where` $i \in S \wedge c(i)]$ where `ub`(S) returns an upper bound on the set S. Note that all set variables in MINIZINC are guaranteed to have a known finite upper bound. Generating comprehensions over variable sets can be highly convenient for modelling. The constraint language ESSENCE [7] has a similar feature but only for specific functions like *sum*. In MINIZINC since comprehensions generate arrays that can be then used as arguments to any predicate or function, this is impossible without option types.

Example 15. Consider the Balanced Academic Curriculum problem [4] (problem 30 in CSPlib (`www.csplib.org`)). The computation of the total load $T[s]$ of a student s, taking courses $C[s]$ where course c has load $L[c]$, is usually defined in MINIZINC as

```
array[Student] of var set of Course: C;
constraint forall(s in Student)
           (T[s] = sum(c in Course)(bool2int(c in C[s])*L[c]));
```

but a more natural formulation is simply

```
array[Student] of var set of Course: C;
constraint forall(s in Student)(T[s] = sum(c in C[s])(L[c]));
```

which can be transformed automatically to

```
array[Student] of var set of Course: C;
constraint forall(s in Student)
           (T[s] = sum(c in Course)
                   (if c in C[s] then L[c] else _ endif));
```

The *sum* is now over optional integers. We can use the same syntax even for a user defined function, e.g. if we are interested in calculating average course load $A[s]$

```
constraint forall(s in Student)(A[s] = average(c in C[s])(L[c]));
function var float: average(array[int] of var opt int:x) =
  int2float(sum(x)) /
  sum([1.0 | i in index_set(x) where occurs(x[i])]);
```

This also allows us to use option types to express *open global constraints* in a closed world [9], that is global constraints that act on a set of variables which is part of the decisions made, but is bounded. For example if the set decision variable S holds the set of indices of variables x to be made all different, we can express this as

[1] To support this we need to extend `if-then-else-endif` in MINIZINC to allow non-parametric tests, or use the alternate translation $[\top, f(i)][\texttt{bool2int}(c(i))+1]$ (that is, an array lookup returning $\top$ if $c(i)$ is false and $f(i)$ otherwise).

```
constraint alldifferent([ x[i] | i in S ]);
```

which becomes

```
constraint alldifferent([ if i in S then x[i] else _ endif |
                          i in ub(S) ]);
```

creating a call to lifted `alldifferent` with exactly the right behaviour (variables whose index is not in S are ignored).

4 Implementing Option Types in MiniZinc

Changes to MINIZINC to support option types are surprisingly small. The major change is the addition of the `opt` type constructor to the language and an additional automatic coercion from type T to `opt` T, and the extension of type inference to handle the new type constructor and coercion.

The remainder of the changes are simply adding a library of lifted definitions of predicates and functions to the system. In this library a decomposition is defined for each of the FLATZINC predicates lifted to option types. Similarly the global constraints of MINIZINC are defined in a library lifted to option types.

4.1 Rewriting Option Type Variables

Critically the option type library *translates* away option types so that underlying solvers do not need to support (and indeed never see) option types. The key to translation is replacing a variable x of type `opt` T by two variables: a Boolean o_x encoding $occurs(x)$, and v_x of type T encoding the value of x if it is not $\top$. If the optional variable x does not occur then $o_x = false$ and the value v_x is left unconstrained.

These are encoded using special functions $v_x = deopt(x)$ and $o_x = occurs(x)$. These two functions are the “primitive” operations on option variables. They are used as terms to represent the encoding, and rely on common subexpression elimination to ensure that there is a unique representation for each optional variable x. These functions are extended to arrays of variables in the natural way. Only in the last stage of translation to FLATZINC are these expressions replaced by new FLATZINC variables. Finally, variables on option types are removed if they do not appear in any constraints in the resulting FLATZINC model.

4.2 Lifting Constraints

Given this encoding we can lift constraints automatically according to the projection interpretation. Existential quantification corresponds to introducing fresh variables in a `let` expression. For example, given a predicate `p(array[int] of var int: x, var int: y)`, the MINIZINC compiler could automatically generate the lifted version

```
predicate p(array[int] of var opt int: x, var opt int: y) =
  let { array[index_set(x)] of var int: xx; constraint xx ~= x;
        var int: yy; constraint  yy ~= y; } in p(xx,yy);
```

Although this automatic lifting would be correct, we can often define better versions of lifted constraints that avoid introducing all those temporary variables.

Example 16. The lifted version on integer disequality `int_ne` can be defined without introducing additional variables as

```
predicate int_ne(var opt int: x, var opt int: y) =
  (occurs(x) /\ occurs(y)) -> (deopt(x) != deopt(y));
```

It simply enforces the disequality if both optional variables occur. Similar definitions exist for all primitive constraints.

4.3 Global Constraints over Option Types

Using the usual MINIZINC rewriting capabilities we can define default decompositions for global constraints over option types. In many cases these can be the same as the regular decomposition.

Example 17. The standard decomposition for `alldifferent` is given below rewritten for option types by adding `opt` to the type.

```
predicate alldifferent(array[int] of var opt int: x) =
          forall(i,j in index_set(x) where i < j)(x[i] != x[j]);
```

This gives a correct implementation since the calls to `!=` will be replaced by `int_ne` over option types, as defined above in Example 16.

For certain global constraints we may be able to do better than simply reusing a standard decomposition.

Example 18. If the underlying solver supports `alldifferent_except_0` we can use that to translate `alldifferent` on optional variables. In the new array *ox*, we shift the values of the original array *x* to be above 0 and map $\top$ for *x* variables to 0.

```
predicate alldifferent(array[int] of var opt int: x) =
  let { int: l = lb_array(x); int: u = ub_array(x);
        array[index_set(x)] of var 0..u-l+1: ox; } in
  alldifferent_except_0(ox) /\
  forall(i in index_set(x))
        ((absent(x[i]) -> ox[i] = 0) /\
         (occurs(x[i]) -> ox[i] = deopt(x[i])-l+1));
```

Example 19. A solver supporting the `cumulative` constraint could implement the disjunctive constraint by modelling absent tasks as using zero resources:

```
predicate disjunctive(array[int] of var opt int: s,
                      array[int] of int: d) =
    cumulative(deopt(s),d,
               [bool2int(occurs(s[i])) | i in index_set(s)],1);
```

4.4 Native Support for Option Types

To achieve better performance, solvers that natively support some option types can define predicates directly (without decomposition) and they will be passed directly to the FLATZINC model used by the solver. In order to mix the usage of option types where some functions and predicates are natively supported and some are not, we require solvers that natively support option types to natively support the primitive functions $deopt(x)$ and $occurs(x)$.

Example 20. For a solver that natively supports optional tasks we can add this declaration to its globals library to pass the `disjunctive` constraint directly to the solver:

```
predicate disjunctive(array[int] of var opt int: s,
                      array[int] of int: d);
```

4.5 Different Encodings

An alternative encoding of option integer type variables (see [17] for details) is to replace each optional integer variable x ranging over $b..e$ by a Boolean $o_x = occurs(x)$ (as before) and two integer variables $l_x = lower(x)$ ranging over $b..e+1$ encoding the lower bound on x if it occurs, and $u_x = upper(x)$ ranging over $b-1..e$ encoding the upper bound if it occurs. These are related by $o_x \leftrightarrow l_x = u_x$, $\neg o_x \rightarrow l_x = e+1$, $\neg o_x \rightarrow u_x = b-1$. Propagators are extended to enforce lower bounds on x as lower bounds on l_x, and never enforce a lower bound greater than e. Similarly upper bounds on x are enforced as upper bounds on u_x and never less than $b-1$. The advantage of this representation is that in a CP solver with explanation [15] there are literals encoding optional lower bounds $[l_x \geq d]$ and optional upper bounds $[u_x \leq d]$.

Example 21. Using this alternate encoding we can define the disjunctive constraint to make use of a builtin disjunctive on these variables `opt3_disjunctive`

```
predicate disjunctive(array[int] of var opt int: s,
                      array[int] of int: d) =
    opt3_disjunctive([occurs(s[i])  | i in index_set(s)],
                     [lower(s[i])   | i in index_set(s)],
                     [upper(s[i])   | i in index_set(s)], d);
```

5 Experiments

In this section, we show that models involving optional variables yield efficient solver-level models, comparable in performance to a manual encoding of optionality. We have extended our MINIZINC compiler to handle option types as described in Sect. 4.

As an experiment, we model the flexible job shop scheduling problem as in Example 11 and solve it using a lazy clause generation solver, which is the state of the art for flexible job shop problems as described in [17]. The `alternative` constraints are implemented using a decomposition into `element` constraints. For the `disjunctive` constraints, we use the three-variable encoding from Sect. 4.5 and compare it with the following simple decomposition:

Table 1. Experimental results

Instance	decomposition			global			hand-written [17]		
	objective	*runtime*	#CP	*objective*	*runtime*	#CP	*objective*	*runtime*	#CP
fattahi/mfjs1	468*	1.60s	8250	468*	1.01s	387	468*	0.97s	388
fattahi/mfjs2	446*	1.53s	10133	446*	1.00s	327	446*	1.01s	330
fattahi/mfjs3	466*	4.49s	24506	466*	3.10s	681	466*	3.37s	697
fattahi/mfjs4	554*	8.84s	24546	554*	7.24s	1024	554*	6.98s	964
fattahi/mfjs5	514*	8.67s	23399	514*	7.30s	881	514*	7.20s	900
fattahi/mfjs6	634*	41.85s	37025	634*	33.70s	3666	634*	32.30s	3474
fattahi/mfjs7	959	—	—	909	—	—	909	—	—
fattahi/mfjs8	1095	—	—	889	—	—	889	—	—
fattahi/mfjs9	1466	—	—	1123	—	—	1123	—	—
fattahi/mfjs10	1609	—	—	1395	—	—	1395	—	—

```
predicate disjunctive(array[int] of var opt int: s,
                      array[int] of int: d) =
  forall (i,j in index_set(s) where i<j)
         ( s[i] ~+ d[i] <= s[j] \/ s[j] ~+ d[j] <= s[i] );
```

Table 1 compares the results of running the `lazyfd` G12 solver [5] on the *same* MiniZinc model using option types with two alternate definitions for `disjunctive`: the decomposition and a global using `opt3_disjunctive` from Section 4.5. The mapping to FlatZinc without option types is managed completely automatically. We compare against the hand-written MiniZinc model (hand-written) used in [17], which uses the same global constraint but with a manual encoding of the option types. Unlike [17], we use a fixed search in order to test the difference between the models. We compare the best objective value found (a * indicates the optimal value was proven), the runtime (or — for $> 600s$), and the number of choice points (#CP). Comparing the results shows that the automatic three-variable `disjunctive` decomposition matches the performance of the hand-written models from [17], the only difference being a slight variation in the explored search tree. The models using the simple decomposition, which can be used with any solver even if it does not support the global disjunctive constraint with optional tasks, scale quite well at least for the smaller examples.

6 Related Work and Conclusion

Modelling dependent decisions by hand is common in constraint programming, and arguably accounts for many models using implication to control whether constraints are active or not. Optional tasks [11,12] were explicitly added to the modelling language OPL to handle the common case of scheduling with optional variables. They correspond to option tuple types (start, duration, end) rather than optional start times we use here. When MiniZinc is extended to handle tuple types, we will be able to model optional tasks in exactly the same manner.

Conditional constraint satisfaction problems (CCSPs) [13,8] are strongly related to option types. A CCSP is a CSP with control on which variables are *active*; i.e., participate in a solution. It splits the variables V into an *initial* always active set V_I, and a *possibly* active set V_P, and splits constraints into regular *compatibility constraints* and *activity constraints*, which control which variables need to take a value. Activity constraints can be of the form $c \xrightarrow{incl} v$ meaning if c holds v is active, and $c \xrightarrow{excl} v$ meaning if c holds v is inactive. A compatibility constraint is *relevant* if all of its variables are active; irrelevant constraints are not imposed. We can straightforwardly model CCSPs using option types. Variables in V_P are lifted to option types. A compatibility constraint c is *relevant* if $rel(c) \equiv \wedge_{v \in vars(c) \cap V_P}\ occurs(v)$, and is modelled as $rel(c) \rightarrow c$. Activity constraints are modelled directly as $rel(c) \wedge c \rightarrow occurs(v)$ for inclusion and $rel(c) \wedge c \rightarrow absent(v)$ for exclusion. Since conditional CSPs build on a traditional CSP framework they don't consider global constraints and complex Boolean and integer expressions and hence they in effect use a very simple form of constraint lifting, *relevance*, which is analogous to absorption lifting. This lifting does not give the behaviour we want for optional tasks for example.

Previous work [16] has attempted to reformulate CCSPs into ordinary CSPs by adding a "null" value to the domains of possibly active variables. This approach was necessary because the constraint solvers used in that work only supported extensional table constraints. Our approach generates implications involving arbitrary constraints, which builds on the support for reification in MINIZINC.

Recently Caballero *et al* [3] built a transformation that allows the user to extend base types of MINIZINC (`float`, `int` and `bool`) with additional values and map them to MINIZINC. This could be used to implement option base types, but the meaning of extended constraints and functions is left to the user to define. The work is very similar in flavor; there the type extension is more general, and the meaning defined by the user, while the translation to base MINIZINC is fixed. Here the extension is fixed—and a strong contribution is to define the meaning of the extension—while the translation to MINIZINC can be redefined.

Option types share some similarities to the treatment of partial functions in the relational semantics [6] adopted by MINIZINC. Partial functions introduce an undefined element $\perp$ which in the relational semantics percolates up to the nearest enclosing Boolean context, where it is treated as *false*. Option types are more complex, the extra value $\top$ percolates up to where there is an identity element, then acts like identity. A four-valued treatment of the semantics of MINIZINC would be ideal, but would not match the reality of the underlying two-valued solving technology.

Option types are a simple yet powerful addition to a modelling language. They allow concise and natural expression of circumstances where some decisions are irrelevant if other decisions are not made. Adding option types to MINIZINC turns out to be surprisingly easy, and also allows us to extend the comprehension syntax. We can use option types to recreate the state-of-the-art solution to flexible job shop scheduling problems.

References

1. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalogue: Past, present and future. Constraints 12(1), 21–62 (2007)

2. Bochvar, D., Bergmann, M.: On a three-valued logical calculus and its application to the analysis of the paradoxes of the classical extended functional calculus. History and Philosophy of Logic 2, 87–112 (1981)
3. Caballero, R., Stuckey, P.J., Tenoria-Fornes, A.: Finite type extensions in constraint programming. In: Schrijvers, T. (ed.) PPDP 2013. pp. 217–228. ACM Press (2013)
4. Castro, C., Manzano, S.: Variable and value ordering when solving balanced academic curriculum problems. http://arxiv.org/abs/cs/0110007 (2001)
5. Feydy, T., Stuckey, P.: Lazy clause generation reengineered. In: Gent, I. (ed.) CP 2009. LNCS, vol. 5732, pp. 352–366. Springer (2009)
6. Frisch, A., Stuckey, P.: The proper treatment of undefinedness in constraint languages. In: Gent, I. (ed.) Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 5732, pp. 367–382. Springer (2009)
7. Frisch, A.M., Harvey, W., Jefferson, C., Hernández, B.M., Miguel, I.: Essence : A constraint language for specifying combinatorial problems. Constraints 13(3), 268–306 (2008)
8. Geller, F., Veksler, M.: Assumption-based pruning in conditional CSP. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 241–255. Springer (2005)
9. van Hoeve, W.J., Régin, J.C.: Open constraints in a closed world. In: Beck, J.C., Smith, B.M. (eds.) CPAIOR 2006. LNCS, vol. 3990, pp. 244–257. Springer (2006)
10. Kleene, S.C.: Introduction to Metamathematics. North Holland (1952)
11. Laborie, P., Rogerie, J.: Reasoning with conditional time-intervals. In: Wilson, D.C., Lane, H.C. (eds.) FLAIRS 2008. pp. 555–560. AAAI Press (2008)
12. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: Reasoning with conditional time-intervals part II: An algebraical model for resources. In: Lane, H.C., Guesgen, H.W. (eds.) FLAIRS 2009. pp. 201–206. AAAI Press (2009)
13. Mittal, S., Falkenhainer, B.: Dynamic constraint satisfaction problems. In: Proceedings of the National Conference on Artificial Intelligence (AAAI). pp. 25–32 (1990)
14. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessiere, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer (2007)
15. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints 14(3), 357–391 (2009)
16. Sabin, M., Freuder, E., Wallace, R.: Greater efficiency for conditional constraint satisfaction. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 649–663. Springer (2003)
17. Schutt, A., Feydy, T., Stuckey, P.J.: Scheduling optional tasks with explanation. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 628–644. Springer (2013)
18. Łukasiewicz, J.: On three-valued logic. In: Borkowski, L. (ed.) Selected works by Jan Łukasiewicz, pp. 87–88. North Holland (1970)