Modelling and Solving Online Optimisation Problems*

Alexander Ek,[1,2] **Maria Garcia de la Banda,**[1] **Andreas Schutt,**[2,3]
Peter J. Stuckey,[1,2] **Guido Tack,**[1,2]
[1]Monash University, Australia
[2]CSIRO Data61, Australia
[3]The University of Melbourne, Australia
{alexander.ek, maria.garciadelabanda, peter.stuckey, guido.tack}@monash.edu,
andreas.schutt@data61.csiro.au

Abstract

Many optimisation problems are of an *online*—also called *dynamic*—nature, where new information is expected to arrive and the problem must be resolved in an ongoing fashion to (a) improve or revise previous decisions and (b) take new ones. Typically, building an online decision-making system requires substantial ad-hoc coding to ensure the *offline* version of the optimisation problem is continually adjusted and resolved. This paper defines a general framework for automatically solving online optimisation problems. This is achieved by extending a model of the offline optimisation problem, from which an online version is automatically constructed, thus requiring no further modelling effort. In doing so, it formalises many of the aspects that arise in online optimisation problems. The same framework can be applied for automatically creating sliding-window solving approaches for problems that have a large time horizon. Experiments show we can automatically create efficient online and sliding-window solutions to optimisation problems.

1 Introduction

Many important optimisation problems are *online*—also called *dynamic*—in nature (see, e.g., Jaillet and Wagner, 2012), that is, the information that defines the problem is not completely known and may not be finite. Rather, new information arrives either continuously or periodically, and must be incorporated into the problem in an ongoing fashion. Consider, for example, a traditional job-shop scheduling problem. If the complete set of jobs is known from the start, the problem can be solved offline to generate an optimal (or good enough) schedule. However, it is common to only know an initial set of jobs, with new ones arriving before all previous jobs have finished executing the generated schedule. It is also common for previous jobs to take longer/shorter than expected. While one could wait until all previous jobs have finished to schedule the new jobs, this will typically result in the underutilisation of the available machines. A higher quality solution may be found if the problem is *resolved* to find a new schedule for all jobs that have not started yet (be it old or new) taking any new information into account, i.e., if the problem is solved in its natural online format.

Despite the strong similarities between all online optimisation problems, current approaches to solving them are *problem-specific*. This is because every time new information arrives and a resolve is required, some previous decisions cannot be changed while others can. For example, in an online job-shop scheduling problem, tasks known to have already started cannot be rescheduled. Similarly, in a dynamic vehicle routing problem, we may not be able to add more customers to a vehicle's route after the vehicle has left the depot, but we may be able to change the order in which the customers are visited. Specifying exactly which decisions can be changed and which cannot, requires specifying **how time interacts** with the variables and constraints used to *model* the problem. This is usually done by the modeller, who understands this interaction and can implement an *update-model*, that is, a model that combines the results of previously executed decisions with the newly arriving data. An iterative algorithm can then be used to repeatedly instantiate and solve this update-model at each time point.

In addition, many large offline optimisation problems can be better solved by decomposing them into smaller, simpler problems along a timeline. This popular approach is called *sliding-window decomposition* (Marquant, Evins, and Carmeliet 2015; Belov et al. 2014), where the problem is decomposed into (usually overlapping) windows, each solved in increasing time order. In effect, this converts the offline problem into an online one, where each new window refers to some old parts of the problem (those overlapping with the previous window), and to some new (the rest in the new window). Thus, methods developed to solve online problems can be used to solve large optimisation problems, once one decides how to break up the data into windows.

This paper proposes a *more generic* approach to online optimisation that enables modellers to specify the online aspects of a problem in a declarative way, and automates the resolving process. To achieve this, modellers first create an offline model, i.e., a model of the offline version of the problem. Then, they add *annotations* to the offline model to specify how its data, variables and constraints interact with time.

*An earlier version of this paper has been presented at the ModRef'19 workshop and the doctoral program of CP'19.

Once this is provided, an update-model is constructed automatically from the annotated offline model, and used by an iterative algorithm to solve the online problem. The main contributions of this paper are as follows: (i) a framework for the declarative modelling of online and sliding-window optimisation problems that identifies common interactions of models with time; (ii) an automatic approach to transform an annotated offline model into the update-model needed to resolve the online problem, thus taking into account new data and the results of previously executed decisions; (iii) an implementation of the framework in the MiniZinc (Nethercote et al. 2007) system; and (iv) an experimental evaluation that shows the effectiveness of the framework for tackling online problems and sliding-window decompositions.

2 Background

A *constraint optimisation problem* (COP) $P = (V, D, C, o)$ consists of a set of variables V, an initial domain D mapping variables to (usually finite) sets of possible values, a set of constraints C defined over variables V, and a selected variable $o \in V$ to minimise (without loss of generality). In practice, constraint optimisation problems are specified by a *data-independent model* M written in a modelling language such as MiniZinc (Nethercote et al. 2007), Essence (Frisch et al. 2008), AMPL (Fourer, Gay, and Kernighan 2002), or OPL (Van Hentenryck et al. 1999). A model M of a problem can be *instantiated* with data D into a concrete COP instance $P = \text{instantiate}(M, D)$.

2.1 Running Example: Job-Shop Scheduling

This paper uses MiniZinc to model problems. While most of MiniZinc's syntax is self-explanatory, some complex elements we make use of include (i) *array comprehension*, e.g., `[arr[n]-5|n in NODE]`, which builds an array of expressions with the arrival time for node `n` minus 5; (ii) *array concatenation*, e.g., `a ++ b`, which builds an array by concatenating array `b` at the end of array `a`; (iii) **`if`** `b` **`then`** `t` **`else`** `e` **`endif`**, which yields `t` if b is true and e otherwise, where `e =` **`true`** if the *else* part is omitted; (iv) `forall(i in S)(c[i])`, which holds if for each `i` in range `S` the constraint `c[i]` holds; (v) `exists(i in S)(c[i])`, which holds if for some `i` in range `S` the constraint `c[i]` holds; and (vi) *array slicing* `p[..,i]` returns the i^{th} column of the 2d array `p`. Let us use an actual problem model to clarify MiniZinc syntax further.

Consider a job-shop scheduling problem, where each job `j` has input data about its arrival (earliest start) time `a[j]`, the machine `m[j,k]` that processes each of its tasks `k`, and the processing time `p[j,i]` it requires on each machine `i`. The decisions to be made are the start times `s[j,k]` for each task `k` of each job `j`. A solution must satisfy the arrival times, task order (task i must finish before the start of task j for $i < j$), and machine usage constraints (each machine can only handle one task at a time), while minimising the total makespan. A model for the data and decisions of this problem is shown in Figure 1 (keywords in bold).

The first 9 lines declare the *parameters* of the problem, where the values for those declared in lines 1, 2 and 6–8 will

```
1  int: M;      % number of machines
2  int: J;      % number of jobs
3  set of int: MACH = 1..M;
4  set of int: TASK = 1..M;
5  set of int: JOB = 1..J;
6  array[JOB] of int: a;
7  array[JOB,TASK] of MACH: m;
8  array[JOB,MACH] of int: p;
9  int: h = max(a)+sum(p);
10 array[JOB,TASK] of var 0..h: s;
11 constraint forall (j in JOB)
12   (s[j,1] >= a[j]);
13 constraint forall (j in JOB,
14                     k in 1..M-1)
15   (s[j,k]+p[j,m[j,k]] <= s[j,k+1]);
16 constraint forall (i in MACH)
17   (disjunctive([s[j,t] | j in JOB,
18     t in TASK where m[j,t]=i],
19     p[..,i]));
20 solve minimize max (j in JOB)
21   (s[j,M]+p[j,m[j,M]]);
```

Figure 1: A MiniZinc model for job-shop scheduling.

be given in an input data file, while those in lines 3–5 and 9 are computed in the declaration. Lines 1 and 2 declare two integer parameters representing the number of machines and jobs in the problem, respectively. Lines 3 to 5 declare three sets of integer parameters, each computed from an integer *range* of the form $l..u$, representing the range from l to u inclusive if $l \leq u$, and the empty range otherwise. Lines 6–8 correspond to the `a`, `m` and `p` arrays of parameters introduced above. Line 9 declares the integer parameter `h`, which represents the latest possible completion time, and is computed as the sum of all processing times plus the latest arrival time. Line 10 declares the array `s` of decision variables (keyword **`var`**) introduced before, where the domain of the decision variables is the range `0..h`. The three constraints in lines 11–19 ensure no job starts before its arrival time, each task in a job finishes before the next one starts, and tasks on the same machine do not overlap in time (by using the predefined constraint `disjunctive`), respectively. Finally, the objective is defined in lines 20–21 as minimising the maximum finishing time of the tasks in any job.

2.2 Solving Online Problems by Iteration

As mentioned in the introduction, given an offline model of a problem and a solver, one can implement an iterative algorithm for the online version of the problem. To illustrate this approach in MiniZinc, we extend the job-shop model from Figure 1 with additional parameters to take previous solutions into account. To do this we assume new jobs can arrive at each time point, but the number of machines (and thus tasks) remains constant. The new parameters:

```
int: pJ; % # jobs in previous solution
array[1..pJ,TASK] of int: sol_s;
```

specify that `pJ` of the original `J` jobs are old (from a previous iteration), and the rest are new. For each old job `j` in `1..pJ`,

array cell `sol_s[j,t]` contains the current start time for previous solved task `t` of job `j`. This might be a past, current or future time.

We also add a parameter called `now` that is always set to the current time (in the model's view of time) for the current iteration of solving, and allows modellers to reason about whether an existing job has already started running or not. If it has, we constrain it to remain scheduled at the same time:

```
int: now;
constraint forall(j in 1..pJ, t in TASK)
  (if sol_s[j,t]<=now then
      s[j,t]=sol_s[j,t] endif);
```

We call the resulting, extended model the *update-model*. Once update-model M is defined, a simple iterative algorithm, such as the one in Figure 2, can be used to solve the online problem at each time point. This algorithm iterates, while there is new data, retrieving first the current view D of the problem data, that is, all current parameter values. For job-shop this means that, if new jobs have arrived, the value of `J` will change and arrays `a`, `m` and `p` become larger. The arrays' old values might also change if, say, the processing time of an old job has now increased. The algorithm then retrieves the *current solution* θ, that is, the solution from the previous iteration updated to reflect any changes required since then. For job-shop this might mean changing the start time decisions of jobs due to, for example, the break down of a machine. Then the constrain function adds the parameters that are used for restricting the time-dependent variables. In job-shop this means setting `pJ` and `sol_s` according to the solution θ, and updating `now`. Finally, the update-model is instantiated with the updated data set and solved, and the new solution θ' is used in the shop until the next solve.

online-solve(M):
  while (new data)
    D := get_current_data()
    θ := get_current_soln()
    D' := constrain(D, θ)
    θ' := **solve**(instantiate(M, D'))
    output θ'

Figure 2: Solving online problems iteratively.

In practice, most update models require more complex relationships between the model variables and time, than those introduced above. Section 4 defines new modelling constructs that allow modellers to specify complex relationships concisely and declaratively.

2.3 Characteristics of our Online Problems

We assume our online problems have *complete recourse* (Dantzig 1955), i.e., at each loop iteration in Figure 2, the data D and the decisions θ cannot yield the problem unsatisfiable. This is typically achieved by adding penalties to the objective, e.g., for not scheduling jobs. In addition, external sources can use D to update previous data (e.g., new task durations) and θ to update optimisation decisions (e.g., new task start times) until they are *realised*, that is, until their actual value is observed, often indicating they refer to the past. We say an execution is *perfect* if neither is actually updated, making the realisations always follow what is given.

Note that our framework does not reason about whether (or how) the information may have changed. It simply assumes everything is static and then reacts to any changes that occur. This is similar to what (Brown and Miguel 2006) calls *pure reaction*, except that rather than using local repair methods, we resolve the model in its entirety in each iteration. Thus, no information about the previous iteration is used unless necessary (e.g., the assignment of a committed variable). Also, we do not use warm-start techniques (this remains future work). Note, however, that only future events are resolved; past events are fixed by the annotations and not resolved. As a result, this paper does not consider methods for specifying or utilising stochastic information (such as probabilities or distributions) during the online optimisation (Van Hentenryck and Bent 2009; Verfaillie and Jussien 2005; Bent and Van Hentenryck 2004), nor for predicting future changes or data based on historical data (Bent and Van Hentenryck 2005). Both remain future work.

3 Related Work

Online problems and solution methods have been well studied. The two main approaches are (a) using an off-the-shelf solver with an ad-hoc online algorithm wrapped around it, and (b) developing a problem-specific algorithm. This paper develops a new approach by extending a solver-independent modelling language to support online problems natively. This considerably reduces the implementation effort and allows modellers to experiment with different solvers.

Approach (a) requires the implementation of an iterative resolve algorithm that is wrapped around a particular solver, and uses a sliding-window approach where the new data arrives between resolves. Examples of this approach include that of Bertsimas, Jaillet, and Martin (2019) for solving an online vehicle routing problem. They update the problem by adding nodes and edges to a graph, and develop their own iterative online algorithm. See (Lim et al. 2016; Rahbar, Xu, and Zhang 2015; Clark and Clark 2000) for other examples. These wrapper algorithms are often problem-specific, and require the model to be formulated in a way that obfuscates the underlying problem.

Examples of approach (b) are more widespread, and include the algorithms for online vehicle routing given in survey (Jaillet and Wagner 2008), and the online scheduling algorithms described in (Pruhs, Sgall, and Torng 2004). In some cases, the same decisions have to be taken repeatedly (with some or total disregard to previous decisions) over time, in real-time. This case is often addressed by developing fast single-point algorithms or models that can be used to resolve with the latest data as desired, and then replacing the old decisions with the new ones (He et al. 2018).

We have not found any problem-independent framework (solver-independent or not) that enables the modelling of online problems for real-time applications or sliding-window decompositions. The closest work is that of modelling language AIMMS, which supports the modelling and use of sliding-window decomposition (referred to as "rolling horizon") of time-based offline problems (Roelofs and Bisschop

2019). This is done by first coding how all the parts of a model can be divided into multiple (possibly overlapping) windows, and then coding an iteration script that iterates through all these windows, solves them, and makes any necessary changes between the iterations. Hence, it is really an example of approach (a). The sliding-window feature of AIMMS have been used in several works (Marquant, Evins, and Carmeliet 2015; Beraldi et al. 2011).

Other work that can be seen as related includes that on dynamic constraint satisfaction problems (DCSP) (Dechter and Dechter 1988), which is often used to reason about (rather than solve) online and dynamic problems (Frank 2016). DCSP could thus be used to formalise our modelling framework. An interesting DCSP variation, called *constraint networks on timelines*, was introduced as a way to unify planning and scheduling (Pralet and Verfaillie 2008). Rather than an online problem, this approach allows specifying and solving problems where the number of steps required to find a solution must be determined during solving.

Dunke and Nickel (2016) presented a formal and mathematical way of reasoning about and analysing online problems. They consider a smaller class of online problems that are not real-time (meaning that *the time it takes* to solve an iteration does not impact the next iteration) and where once a decision is made, it cannot be revised. Their main contribution is their concept of lookahead in online optimisation. In our framework, we always have (a dynamic degree of) lookahead, unless all the variables in an iteration are committed in the next iteration.

Chien et al. (2000) presented an iterative repair method for planning and scheduling NASA robots in an online fashion. Our framework can be seen as using iterative repair to deal with online problems as well, and can be thus used to tackle these problems.

Note that this paper does not consider methods for specifying robustness and stability criteria (Climent et al. 2014). This remain as future work. Currently, specifying these criteria is only possible when manually defined by the modeller in the objective function, using the functionalities currently available in our framework.

4 Modelling Online Problems

This section introduces our extensions to the MiniZinc language to support the solver-independent modelling of online optimisation problems. Recall the iterative algorithm of Figure 2 for the online job-shop example from Section 2.2. Compared to a standard algorithm for solving the offline model M, this algorithm contained three additional components: the functions get_current_data and get_current_soln to retrieve the current data for the problem and the current solution, both of which may update previous information; and the constrain function which uses the current solution and current data to generate the actual time dependencies of the update-model.

Our extensions are based on *annotations* that modellers can add to a standard, offline MiniZinc model to capture these aspects in a concise and declarative way. The update-model, together with the constrain functions, is then generated automatically from this annotated model (the generation of the update-model is discussed in Section 5).

The first annotation, `:: extends`, identifies parameters that can be extended with new data at each time point. If a parameter annotated with `:: extends` is used to define other parameters (e.g., it is part of an array index set), those other parameters automatically become extendable with new data as well. Consider the offline job-shop problem introduced in Figure 1, and assume that in its online version new jobs can arrive as time progresses. To transform the offline model for this problem into an online one, we start by annotating the parameter `J` in line 2, obtaining **`int`**`: J :: extends;`, thus indicating that `J` might increase with time. Since `J` is used to define the set `JOB`, this also indicates that the amount of data in each of the arrays `a`, `m`, and `p`, might similarly increase with time.

The second annotation, `:: time`, identifies decisions that cannot be changed by the solver after a certain time point. The most obvious of these affect *time variables*, that is, variables whose domain is time itself. In particular, past decisions that fixed a time variable to a value earlier than the *current time*, cannot be changed by the solver. Also, if such a variable is not yet fixed, the solver cannot fix it to a value that is earlier than the current time. Continuing the job-shop example, the only decision variables of the model are the start times for each task of each job, that is, the array of variables defined in line 10. The domain of these variables is indeed time and, hence, the declaration must be annotated, yielding **`array`**`[JOB,TASK]` **`of var`** `0..h: s :: time;`

While the domain of some variables is not time itself, it may nevertheless reflect decisions that cannot be changed by the solver after a certain point in time. We say such decision must be *locked*. To achieve this, modellers can annotate such a variable v with `:: lock_var_time(`t`)`, where t is a variable whose domain is time and whose value is the time point after which a decision for v cannot be changed. Note that when annotating an array d of variables, t must also be an array of variables with the same dimensions as d.

Consider an open-shop scheduling problem similar to that of Figure 1, except that the allocation of tasks to machines is not fixed, i.e., the array of parameters in line 7 is now declared as an array of variables. This means the solver now needs to decide the order of the tasks of a job by allocating each task to a machine, as this is no longer provided by the input data. Clearly, once a task has started to be processed, the machine that processes it cannot change. Thus, the online model for this problem has in line 7 the declaration **`array`**`[JOB,TASK]` **`of var`** `MACH: m :: lock_var_time(s);`. Note that the dimensions of arrays `s` and `m` are the same. This annotation ensures that if the start time `s[j,k]` for task `k` of job `j` is less than or equal to the current time, then the machine `m[j,k]` chosen for this task cannot be changed.

A more complex form of time constraint common in online problems, involves checking the *values* of a variable: while some of these values might need to be locked once selected, others might become unavailable as time progresses. To achieve this, modellers can annotate such a variable v with `lock_val_time(`t`)`, where t is a one-dimensional ar-

ray that corresponds to the declared domain of v.

To illustrate this, consider a package delivery routing problem for C customers and V vehicles, where all customers must be visited once. Assume the problem is modelled using a graph with $N = C + 2V$ nodes, where there is one node for each customer and two nodes for each vehicle v, representing the time when v leaves from and returns to the depot. The decision variables for each node n, are the arrival time at n, the next node visited from n, and the vehicle that visits n. The following partial model for an online version of this problem:

```
1 int: V :: extends;  % # of vehicles
2 int: C :: extends;  % # of customers
3 int: h :: extends;  % scheduling horiz.
4 int: N = C + 2*V;   % # of nodes
5 set of int: NODE = 1..N;
6 set of int: CUST = 1..C;
7 set of int: VEH  = 1..V;
8 array[NODE] of var 0..h: arr :: time;
9 array[NODE] of var NODE: next
10 ::lock_var_time([arr[n]|n in NODE]);
11 array[NODE] of var VEH: veh
12 ::lock_val_time([arr[C+v]|v in VEH]);
```

indicates new customers and new vehicles might appear (a vehicle returning to the depot becomes available as a new vehicle), and the time horizon for scheduling can also change (as more customers arrive). The models also indicates that the arrival time at each node is a `time` constrained variable (line 8), and the decision about where to go next from node n is locked at the time point where the vehicle arrives at n (line 10). Also, since the packages must be loaded onto vehicles v at the depot, the decision of which customers are visited by vehicle v is locked at the time point where v leaves the depot. This is recorded as the arrival time at the vehicle's start time node `arr[C+v]` (line 12).

Note that the `lock_val_time` annotation introduced above, conflates two different kinds of restrictions: *commit* and *forbid*. These indicate, respectively, that a decision cannot be changed or is no longer available as time progresses. We thus define two annotations `commit_val_time` and `forbid_val_time`, to separate the two parts conflated by `lock_val_time`.

Consider again the package delivery problem. When deciding which vehicle should visit each customer, it is unrealistic to add (or remove) a customer to (or from) vehicle v if v is about to leave the depot, since it takes time to load (or unload) the package. Assuming we need 5 minutes to pack a new delivery, and 15 minutes to find and remove a packed delivery, we can substitute lines 11 and 12 with code:

```
array[NODE] of var VEH: veh
::forbid_val_time([arr[C+v]-5 | v in VEH])
::commit_val_time([arr[C+v]-15 | v in VEH]);
```

to reflect the correct behaviour. The annotations state that the decision of assigning a customer to vehicle v cannot be changed by the solver if in the current solution v leaves in the next 15 minutes, and a customer cannot be (newly) assigned to a vehicle leaving in the next 5 minutes.

The above seven annotations capture the most common time constraints that arise when solving an online problem. For problem-specific cases, where these annotations are not sufficient, we extend MiniZinc to give modellers access to the current solution via a generic function `sol(`x`)`, which returns, for each variable and parameter x, the value of x in the current solution. This is analogous to the use of the function `sol()` in MiniSearch (Rendl et al. 2015) and other extensions of MiniZinc (Dekker et al. 2018) to refer to the previous solution to a problem. In addition, we give modellers access to function `has_sol(`x`)` to test whether x actually exists in the current solution, and to the `now` parameter for use in their time constraints.

Consider a variation of the package delivery problem, where we inform customers of the expected arrival times within the next 24 hours. Consider also that, after informing clients about their arrival time, we want to ensure that later solutions do not delay these arrival times by more than an hour. Modellers can express this in the model as follows:

```
constraint forall(c in CUST where
                  has_sol(arr[c]))
  (if sol(arr[c]) <= now + 24*60
   then arr[c] <= sol(arr[c]) + 60 endif);
```

Note that, since `has_sol` always returns **`false`** if used offline, the constraint will not be active then. Also, since only modellers know how `now` relates to time in the real world, they are the ones who must define `now` as a parameter computed using the system time calls available in MiniZinc.

5 Transformation

Once modellers have created an online model M_O by annotating the offline model, the next step is to transform M_O into an update-model M. This is achieved by automatically (i) adding a declaration for `now` and for the `sol` counterpart (e.g., `sol_s`, in our job-shop model) of every parameter and variable, as shown in Section 2.2, and (ii) transforming each annotation as shown in the rest of this section.

Extends This annotation is used to generate the current form of the data by appending old and new data. We use MiniZinc itself to perform the append as follows. For any parameter `p :: extends`, we generate two versions in the update-model: `p_old` and `p_new`. The actual parameter `p` is then computed as `p = p_old+p_new` for numeric parameters, and `p = p_old ++ p_new` for array parameters. As mentioned in Section 4, if an online parameter `p` is used to define the index set of another array parameter `q`, then `q` is also considered to be an online parameter.

Time annotations The `time` annotation:

```
var D: x :: time;
```

where x is a variable over domain D, is transformed into the following constraint:

```
if has_sol(x) /\ now >= sol(x)
then x=sol(x) else x>=now endif;
```

This ensures a previous decision for x can only be changed if its value is still in the future, and new decisions for x cannot be set to the past (i.e., the past cannot be changed).

Variable annotations An annotation of the form:

```
var D: x :: lock_var_time(t);
```

where x and t are variables over D and time, respectively, is transformed into the following constraint:

```
if has_sol(x) /\ now >= sol(t)
then x=sol(x) endif;
```

This ensures a previous decision for x is not changed once the time point given by the value of t in the current solution has arrived. The extension to an annotation for an array of variables (rather than over a single one x) is straightforward.

Value annotations We show how commit and forbid annotations are transformed. (Recall that lock annotations simply combines the two.) A commit annotation is of the form:

```
var D: x :: commit_val_time(t);
```

where x is a variable over D, and t is an array of variables indexed by D. It is transformed into:

```
if has_sol(x) /\ now >= sol(t[sol(x)])
then x = sol(x) endif;
```

This ensures that if the current time associated with the value taken by x is in the past, then the decision is fixed. A forbid annotation is of the form:

```
var D: x :: forbid_val_time(t);
```

where x is a variable over domain D, and t is an array of variables indexed by D. It is transformed into:

```
forall(d in D
       where has_sol(x) /\ sol(x) != d)
  (if has_sol(t[d]) /\ now >= sol(t[d])
   then x != d endif);
```

This ensures that, for each value $d \in D$ of x (except its current value), if the associated time point of d (given by `t[d]`) has passed, then x cannot be newly assigned to d.

6 Garbage Collection

As defined, the iterative algorithm from Figure 2 will construct larger problems as time progresses: more data is added, which also leads to more variables and constraints. In typical online problems, many variables will be fixed immediately due to their time dependencies. While this may not have a big impact on solving time, for long running online problems it can lead to an increase in the time taken by he instantiation phase (translating the MiniZinc model into the solver-level language). If we can identify parts of the data that can have no effect on the remaining solving, such data can be omitted. We call this *garbage collection* of old data.

For example, in the online job-shop problem based on Figure 1, jobs that are completely finished can have no further effect on any jobs scheduled after `now`. Hence, they can be omitted from the data. Jobs that have started but not yet finished, however, can still affect new jobs, since they still use resources, and must be kept.

Currently, the framework requires the modeller to identify the parts of the data that can be garbage collected, by adding to their model new parameters that are annotated as `::gc` (for garbage collection). An interesting avenue for future work is to explore how (much of) this could be determined automatically from the model and its annotations.

For example, modellers can change the online version of the job-shop model to enable garbage collection as follows:

```
1 int: LJ :: gc(arg_max( [ exists(i in TASK)
2       (sol(s[j,i]) + p[j,m[j,i]] > now)
3             | j in JOB ] ++ [true]));
4 set of int: JOB = LJ..J; % meaningful jobs
```

Initially, `LJ` is set to `1`. After instantiating the update-model, the expression inside the `gc` annotation will be evaluated to the index of the first job with a task that is still running (`sol(s[j,i]) + p[j,m[j,i]] > now`). Note that because of the **`true`** entry in the array, `arg_max` returns the first element where the Boolean condition evaluates to true. If this expression determines, for example, that job number 5 is the first such job, then `LJ` will be set to 5. For the next iteration, the data for all parameter arrays that use `LJ` in their index sets (`a`, `m`, and `p`) will be garbage collected accordingly.

Note that this method always lags behind by one iteration: the model computes which data would have been irrelevant for the time point of the current iteration, and that data is then excluded at next the iteration. Another weaknesses is that if there are some very long running jobs that arrive early, they will keep all the jobs after them alive until they finally are finished. Still, the simplicity of the approach is very appealing, and if the jobs are reasonably uniform in duration, this will not be a problem.

Currently, it is up to the modeller to make sure that the constraints and the objective function stays consistent after the garbage has been removed. Automating this process is also an interesting avenue for future work.

7 Experiments

All experiments were run on a 2.2 GHz Intel Core i7 processor with 16 GB RAM, using the lazy clause generation solver Chuffed (version bundled with the MiniZinc IDE Version 2.2.3). The MiniZinc interface for Python was used to simulate data changes between iterations, and the garbage collection mechanism described above was added to the model to improve the combination of old and new data.

7.1 Using our Framework[1]

We have extended the MiniZinc-Python[2] interface to automatically transform an online model and run it iteratively. Modellers can use this interface to define how time passes and how their problem evolves over time. They can also use it to to define how decisions are executed and data is realised. For example, modellers can overwrite the `get_execution` interface method to connect it to an external simulation or monitoring system, reflecting some real-world execution. They can also overwrite the `get_new_data` interface method to define how and from

[1] Code available at https://gitlab.com/minizinc/online-minizinc

[2] https://gitlab.com/minizinc/minizinc-python (21-Nov-2019)

	with GC		without GC	
instance	solving (s)	instn. (s)	solving (s)	instn. (s)
abz5	**13.01**	**19.00**	75.64	3,913.48
ft06	**14.56**	**18.91**	56.63	1,363.91
ft10	**17.04**	**19.86**	121.59	3,882.49
ft20	**1,467.56**	**18.84**	1,490.86	941.64

Table 1: Accumulated solving and instantiation time with and without garbage collection (GC).

where the new data is fetched, before solving is performed in each iteration. Other functionalities allows modellers to, e.g., define how time passes (e.g., real-time or stage-based) and how to translate wall-clock (or simulation) time to the time units used in their model.

7.2 The Effect of Garbage Collection

Our first experiment uses the online job-shop scheduling problem described throughout the paper. The online data used is constructed from the offline data file `abz5` of the MiniZinc benchmarks[3] by repeatedly adding copies of the jobs from `abz5` into an endless queue. We then iterate through the endless queue as follows. The first i jobs in the queue form the initial iteration, where i is the greatest integer such that the first iteration can be solved to optimality within 5 seconds under perfect execution. Consecutively, the next $i/2$ jobs from the queue form the next iteration, for as many iterations as desired. The `now` parameter for each iteration increases by p each time, where p is the lower bound makespan for scheduling the new jobs, assuming all tasks have the average processing time of the given instance.

The main focus of this experiment is to illustrate the effect of garbage collection. We define `now` to be independent of the solving time, so that we can compare identical iterations that use/not use garbage collection.

Figure 3 shows the instantiation time (a) and solving time (b) for each iteration, while Table 1 shows the accumulated solving and instantiation time for multiple instances, each running for 300 iterations. Without garbage collection, instantiation time quickly starts dominating, which highlights the importance of garbage collection for long running online problems. Solving time also increases without garbage collection, which is partly due to the fact that a much larger initial model has to be parsed, and the additional propagation for parts of the model that are fixed needs to be performed. Note that we are using perfect execution in this experiment.

Given these results, all remaining experiments use garbage collection.

7.3 Real-Time Online Job-Shop Scheduling

Our second experiment also uses the online job-shop scheduling problem. The online data used is constructed as before, but using the offline data file `ft20`.

[3]https://github.com/MiniZinc/minizinc-benchmarks/tree/master/jobshop (21-Nov-2019)

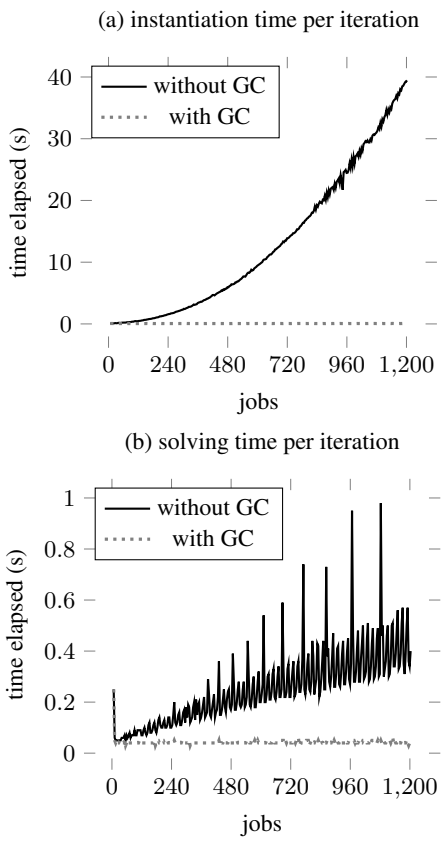

Figure 3: How (a) instantiation time and (b) solving time per iteration changes over time, with and without garbage collection (GC), using `abz5`.

This experiment shows how our framework can deal with real-time online problems with uncertain task durations. Because of the real-time aspect, `now` is defined as the time at the end of the solving time limit. Each task duration is given as an expected value of an unknown distribution, which means that its realisation may differ from its given value.

As a result, the realisation of a task's start time is also uncertain, since it may have to be pushed forward during execution, due to a predecessor task running slower than expected. We simulate the execution step in this experiment as follows. The execution is based on the solution found by solving the model. Then, each task that will be finished in the next iteration (using the given task duration) becomes its realised task duration, generated from a normal distribution with the mean parameter as the original task duration (which is what was given to the model initially) and the standard deviation as one tenth of the average task duration in the instance file (`ft20`). If a task runs slower (or faster) than expected, then all other tasks that depend on it are set to their earliest possible start time respecting the constraints, and this propagates throughout all tasks. See online schedule generation (Sprecher, Kolisch, and Drexl 1995). The realised data and decisions are then read into the model and accessed as normal via the `sol()` command.

	makespan	
time/iteration	4 jobs/s	8 jobs/s
0.25 s	—	—
0.50 s	**117,845**	—
1.00 s	117,973	70,663
2.00 s	118,170	68,549
4.00 s	118,749	**67,892**
8.00 s	119,581	68,632
16.00 s	121,352	71,687

Table 2: Real-time online job-shop scheduling (file `ft20`). Showing average over 5 runs. The symbol — denotes a time-out of an iteration along the way.

	original		sliding-window	
instance	time	delay	time	delay
07_1s_133	—	10,563	**40.35**	**5,868**
08_222f_3475	—	70,068	**90.38**	**66,085**
09_1s_18_OPT	—	**7,117**	**0.41**	22,958
10_15966f_2060	—	38,119	**90.34**	**36,347**
16_10720f_4243	—	—	**80.71**	**88,862**
19_31058f_2548	—	141,119	**77.30**	**129,748**

Table 3: Cargo Assembly Planning Problems solved as a single optimisation problem and using sliding-window decomposition. The symbol — under time indicates a time out (120s) and under delay indicates no solution was found.

There is a trade-off for the amount of solver time allowed between iterations. Assuming new jobs arrive every second, the more time we allow the solver to spend and find solutions, the further in the future these jobs will have to be scheduled (to account for the solving time) and the more new jobs will arrive for the next iteration to consider. However, if we give the solver too little time, it may not be able to make the optimal decisions within each iteration, and may thus need to simply use the best solution found within the time limit (or in the worst case, find no solution at all).

Table 2 shows the results for scheduling 1162 jobs in total (from `ft20`), with different time limits per iteration, and different rates of new jobs per second. For the purpose of this experiment, we assume that 1 wall-clock second corresponds to 408 time units (based on the average task duration in the chosen benchmark).

As we can see, if the rate of new jobs is low (e.g., 4 jobs per second), then using short iterations (e.g., of 0.5 seconds per iteration) yields the best overall makespan. However, too short an iteration (e.g., of 0.25 seconds per iteration) can result in a timeout with no solution. With a higher data rate (e.g., 8 jobs per second), a timeout of 4 second per iteration yields the best result. Being able to quickly experiment with these settings, by translating the external time automatically into model-specific time units, is a significant advantage of our approach over manual approaches.

7.4 Sliding-Window Decomposition: Cargo Assembly Planning

A common approach for large offline optimisation problems is *sliding-window decomposition*, which decomposes the problem into a series of subproblems restricted to a small time window that slides forward during the process. All decisions before the window are fixed and all decisions after the window are not considered. We can use our framework to directly implement and solve sliding-window decompositions, simply by appropriately splitting the data.

To illustrate this, we use the cargo assembly planning problem (CAPP) from the MiniZinc benchmark suite,[4] a simplified version of the problem described by Belov et al. (2014). In CAPP, vessels arrive at different times. Every vessel has a set of cargoes that has to be assembled in a stockyard, at an unoccupied part on a stacking pad, into a set of stockpiles. Each stockpile is then loaded onto its vessel. The aim is to minimize the total delay.

We modified the model in the following ways: the number of vessels becomes an `extends` parameter, the stacking and loading start times for each stockpile are `::time` annotated, and the position of each stockpile is `::lock_var_time` annotated with the assembly start time of that stockpile.

Table 3 shows results for all instances from the MiniZinc benchmarks with at least 16 vessels. We compare the offline solving approach (original), as a baseline, with a sliding-window of 10 non-arrived vessels, adding 5 new vessels at each iteration. A total timeout of 120 seconds (only instantiation and solving time) for each method was used, divided equally amongst the sliding-window iterations.

Clearly, the sliding-window decomposition is usually better than solving the original problem, the exception is for the easiest of the problems, where the global viewpoint allows the solver to find a better solution.

8 Conclusion and Future Work

This paper presents a systematic approach that simplifies the modelling and solving of online optimisation problems significantly, making it more efficient for experienced modellers and more accessible for novices. We introduce several annotations that enable modellers to describe online aspects of their problem, i.e., how the decisions in their models are related to time, in a high-level way.

Experiments show the usefulness of the framework for both online problems and a sliding-window decomposition, as well as the importance of garbage collection. This paper mostly focuses on the job-shop scheduling problem, because of its brevity and clarity to explain online concepts. However, the framework can also be used for many other kinds of online problems, including online vehicle routing problems, where customers/packages are to be delivered (released over time) with, e.g., uncertain travel times; online stage-based assignment problems, where we need to (re-)assign (potentially new/different) agents to (potentially new/different) targets; and other forms of scheduling problems.

Future work includes extending the framework to be able

[4] https://github.com/MiniZinc/minizinc-benchmarks/tree/master/cargo (21-Nov-2019)

to specify in the model the stochastic aspects of online problems (i.e., where parameters have *a priori* distributions or probabilities), incorporating predictions of future data while solving, developing an improved garbage collection mechanism, looking at stability and robustness criteria, using warm-starts and local repair methods, and using dynamically sized windows and multiple passes with sliding-window decompositions.

References

Belov, G.; Boland, N.; Savelsbergh, M. W. P.; and Stuckey, P. J. 2014. Local search for a cargo assembly planning problem. In *CPAIOR'14*, LCNS 8451, 159–175. Springer.

Bent, R., and Van Hentenryck, P. 2004. Scenario-based planning for partially dynamic vehicle routing with stochastic customers. *Operations Research* 52(6):977–987.

Bent, R., and Van Hentenryck, P. 2005. Online stochastic optimization without distributions. In *ICAPS'05*, 171–180.

Beraldi, P.; Violi, A.; Scordino, N.; and Sorrentino, N. 2011. Short-term electricity procurement: A rolling horizon stochastic programming approach. *Applied Mathematical Modelling* 35(8):3980–3990.

Bertsimas, D.; Jaillet, P.; and Martin, S. 2019. Online vehicle routing: The edge of optimization in large-scale applications. *Operations Research* 67(1):143–162.

Brown, K. N., and Miguel, I. 2006. Uncertainty and change. In *Handbook of Constraint Programming*. Elsevier. chapter 21, 731–760.

Chien, S.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve the responsiveness of planning and scheduling. In *AIPS'00*, 300–307. AAAI Press.

Clark, A. R., and Clark, S. J. 2000. Rolling-horizon lot-sizing when set-up times are sequence-dependent. *International Journal of Production Research* 38(10):2287–2307.

Climent, L.; Wallace, R. J.; Salido, M. A.; and Barber, F. 2014. Robustness and stability in constraint programming under dynamism and uncertainty. *Journal of Artificial Intelligence Research* 49:49–78.

Dantzig, G. B. 1955. Linear programming under uncertainty. *Management Science* 1(3/4):197–206.

Dechter, R., and Dechter, A. 1988. Belief maintenance in dynamic constraint networks. In *AAAI'88*, 37–42. AAAI Press.

Dekker, J. J.; Garcia de la Banda, M.; Schutt, A.; Stuckey, P. J.; and Tack, G. 2018. Solver-independent large neighbourhood search. In *CP'18*, LNCS 11008, 81–98.

Dunke, F., and Nickel, S. 2016. A general modeling approach to online optimization with lookahead. *Omega* 63:134–153.

Fourer, R.; Gay, D. M.; and Kernighan, B. W. 2002. *AMPL: A Modeling Language for Mathematical Programming*. Cengage Learning, 2nd edition.

Frank, J. 2016. Revisiting dynamic constraint satisfaction for model-based planning. *The Knowledge Engineering Review* 31(5):429–439.

Frisch, A. M.; Harvey, W.; Jefferson, C.; Martínez-Hernández, B.; and Miguel, I. 2008. Essence: A constraint language for specifying combinatorial problems. *Constraints* 13(3):268–306.

He, S.; Wallace, M.; Gange, G.; Liebman, A.; and Wilson, C. 2018. A fast and scalable algorithm for scheduling large numbers of devices under real-time pricing. In *CP'18*, LNCS 11008, 649–666. Springer.

Jaillet, P., and Wagner, M. R. 2008. Online vehicle routing problems: A survey. In *The Vehicle Routing Problem: Latest Advances and New Challenges*, volume 43 of *OR/CS Interfaces*. Springer. 221–237.

Jaillet, P., and Wagner, M. R. 2012. *Online Optimization*. Springer.

Lim, B.; Hijazi, H.; Thiébaux, S.; and van den Briel, M. 2016. Online HVAC-aware occupancy scheduling with adaptive temperature control. In *CP'16*, LNCS 9892, 683–700. Springer.

Marquant, J. F.; Evins, R.; and Carmeliet, J. 2015. Reducing computation time with a rolling horizon approach applied to a MILP formulation of multiple urban energy hub system. *Procedia Computer Science* 51:2137–2146.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. MiniZinc: Towards a standard CP modelling language. In *CP'07*, LNCS 4741, 529–543. Springer.

Pralet, C., and Verfaillie, G. 2008. Using constraint networks on timelines to model and solve planning and scheduling problems. In *ICAPS'08*, 272–279. AAAI Press.

Pruhs, K.; Sgall, J.; and Torng, E. 2004. Online scheduling. In *Handbook of Scheduling - Algorithms, Models, and Performance Analysis.* Chapman and Hall/CRC.

Rahbar, K.; Xu, J.; and Zhang, R. 2015. Real-time energy storage management for renewable integration in microgrid: An off-line optimization approach. *IEEE Transactions on Smart Grid* 6(1):124–134.

Rendl, A.; Guns, T.; Stuckey, P. J.; and Tack, G. 2015. MiniSearch: A solver-independent meta-search language for MiniZinc. In *CP'15*, LNCS 9255, 376–392. Springer.

Roelofs, M., and Bisschop, J. 2019. *AIMMS: The Language Reference*, May 2, 2019 edition. chapter 33, Time-Based Modelling. Available at: www.aimms.com.

Sprecher, A.; Kolisch, R.; and Drexl, A. 1995. Semi-active, active and non-delay schedules for the resource-constrained project scheduling problem. *European Journal of Operational Research* 80:94–102.

Van Hentenryck, P., and Bent, R. 2009. *Online Stochastic Combinatorial Optimization*. MIT Press.

Van Hentenryck, P.; Michel, L.; Perron, L.; and Régin, J. C. 1999. Constraint programming in OPL. In *PPDP'99*, LNCS 1702, 98–116. Springer.

Verfaillie, G., and Jussien, N. 2005. Constraint solving in uncertain and dynamic environments: A survey. *Constraints* 10(3):253–281.