Modelling and Solving Online Optimisation Problems*

Alexander Ek[1,2], Maria Garcia de la Banda[1], Andreas Schutt[2], Peter J. Stuckey[1,2], and Guido Tack[1,2]

[1] Monash University, Melbourne, Australia
{alexander.ek,maria.garciadelabanda,peter.stuckey,guido.tack}@monash.edu
[2] Data61, CSIRO, Australia Andreas.Schutt@data61.csiro.au

Abstract Many optimisation problems are of an online nature, where new information arrives and the problem must be resolved periodically in order to (a) improve previous decisions and (b) take the required new ones. Typically, building an online optimisation system requires substantial ad hoc coding, where the optimisation problem is continually adjusted and resolved, keeping track of which previous decisions may be committed and which new decisions need to be taken. In this paper we define a framework for automatically solving online decision problems. This is achieved by extending a model of the offline optimisation problem so that the online version is automatically constructed from this model, requiring no further implementation. In doing so, we formalise 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 extend over a large time domain. Experiments show that we can automatically create efficient online and sliding window solutions to problems.

1 Introduction

Many important optimisation problems are *online* in nature (see e.g., [17]), that is, the information that defines the problem may not be finite and is not completely known. Rather, new information arrives either continuously or periodically, and needs to 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. While one could wait until all previous jobs have finished to schedule these new jobs, this will typically result in the underutilisation of the available machines. A better 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), that is, if the problem is solved in its natural online format.

* A shorter version of this paper is part of the CP2019 doctoral program.

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 that requires a resolve, some previous decisions cannot be changed while others can. For example, in an online job-shop scheduling problem, tasks that have already started cannot be rescheduled. As another example, 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 of the problem model. This is usually done by the modeller, who understands this interaction and can implement an *update-model*, that is, a model of the problem that combines the previous solution 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* (see e.g., [19,1]), 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 the modeller to specify the online aspects of a problem in a declarative way, and automates the resolving process. To achieve this, the modeller adds *annotations* to an offline model of the optimisation problem. These annotations specify how the data, variables and constraints in the model interact with time. Once this is provided, an update-model can be constructed automatically from the annotated model, which can then be used in an iterative algorithm to solve the online problem. The main contributions of this paper are as follows:

- a framework for the declarative modelling of online and sliding window optimisation problems that identifies common interactions of models with time;
- an automatic approach to transform an online optimisation model into the update-model needed to resolve the problem that takes into account new data and previously fixed decisions;
- an implementation of the framework in the MiniZinc [20] system; and
- 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 *data-independent models* written in a modelling language such as MiniZinc [20], Essence [13], AMPL [11], or OPL [26]. 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, we will use an example to introduce some of its syntax.

Consider a job-shop scheduling problem where each job includes exactly one task on each machine. Each job has input data about its arrival time (earliest start time for that job), the machines that process each of its tasks, and the processing time it requires on each machine. The decisions to be made are the start times for each task of each job. 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 natural model for the data and decisions of this problem is shown in Figure 1.

The first 9 lines declare the *parameters* of the problem, where the values for those declared in lines 1, 2 and 6–8 will 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. Line 6 declares a one-dimensional array of integer parameters, where `a[j]` gives the arrival time of job `j`. Line 7 declares a two-dimensional array of machine parameters indexed by `JOB` and `TASK`, where `m[j,k]` gives the machine where task `k` of job `j` needs to be processed. Line 8 declares a two-dimensional array of integer parameters, where `p[j,m]` gives the processing time for the task of job `j` processed by machine `m`. Line 9 declares an integer parameter for the latest possible completion time, computed as the sum of all processing times plus the latest arrival time. Line 11 declares a two-dimensional array of variables with domain in the range `0..horiz`, where `s[j,k]` will be determined by the solver to be the start time for task `k` of job `j`. The three constraints express that no job can start before its arrival time, each task in a job finishes before the next one starts, and tasks on the same machine cannot overlap in time. The disjunctive constraint makes sure that intervals (defined by the pairs enumerated by array comprehension) do not overlap. Finally, the objective is defined as minimising the maximum finishing time of the last task in any job.

Other components of MiniZinc that will be used in our models include:

- *array comprehensions* that build one-dimensional arrays, e.g., `[ arrival[n] - 5 | n in NODE ]` builds an array of expressions of the arrival time for node `n` minus 5;

```
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;            % arrival time
7 array[JOB,TASK] of MACH: m;      % machine for task
8 array[JOB,MACH] of int: p;       % processing time
9 int: horiz = max(a)+sum(p);      % latest possible time
10
11 array[JOB,TASK] of var 0..horiz: s; % start times
12
13 constraint forall (j in JOB) (s[j,1] >= a[j]);
14 constraint forall (j in JOB, k in 1..M-1)
15   (s[j,k]+p[j,m[j,k]] <= s[j,k+1]);
16 constraint forall (m in MACH)
17   (disjunctive ([s[j,t] | j in JOB, t in TASK where m[j,t]=m]
18                 , p[..,m]));
19 solve minimize max (j in JOB) (s[j,M]+p[j,m[j,M]]);
```

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

- *array concatenation*, e.g., `a ++ b` gives the result of concatenating array `b` on the end of array `a`;
- *if-then-else-endif* expressions, e.g., `if b then t else e endif` is equal to `t` if b is true and e otherwise. If the *else* part is omitted it is treated as if `e = true`;
- *forall* expressions, e.g., `forall(i in S)(c[i])` holds if for each `i` in range `S` the constraint `c[i]` holds;
- *exists* expressions, e.g., `exists(i in S)(c[i])` holds if for some `i` in range `S` the constraint `c[i]` holds;

2.2 Solving Online Problems by Iteration

As mentioned in the introduction, given a model and solver for an offline problem, one can implement an iterative algorithm for the online version of the problem. To illustrate this approach, we will extend the job-shop model from Figure 1 with additional parameters to take previous solutions into account:

```
int: sol_J; % number of jobs in previous solution
array[1..sol_J,TASK] of int: sol_s; % previous start times
```

The assumption is that at each time point, new jobs can be added to the problem, but the number of machines (and thus tasks) remains constant. The new parameters specify that `sol_J` of the `J` jobs are old (from a previous iteration), and the rest are new. For each old job `j` in `1..sol_J`, `sol_s[j,t]` contains the start time of task `t` of `j` in the previous iteration.

```
online-solve(M,D,θ):   while (new data δ)
                           D := append(D, δ)
                           D' := constrain(D, θ)
                           θ := solve(instantiate(M, D'))
```

Figure 2. An iterative algorithm for solving online problems.

In addition, we will add a parameter called `now` that is set to the current time (in the model's view of time) for the current iteration of solving, and allows us to reason about whether a previously scheduled job has already started running or not. If it has, then we constrain it to remain scheduled at the same time:

```
int: now;
constraint forall (j in 1..sol_J, 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 it 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. The arguments of the online-solve function are the update-model M, the original data D, and the initial solution θ (which can be constructed using a heuristic or by solving offline with the original data D). As long as there is new data, the append function adds it to the current data. For our concrete example, this means appending the data for the new jobs to the `a`, `m` and `p` arrays, and adjusting `J`. The constrain function adds the parameters that are used for restricting the time-dependent variables (in our job-shop example, this means setting `sol_J` and `sol_s` according to the previous solution, and updating `now`). Then the update-model is instantiated with the updated data set and solved. The new solution θ will be used in the next iteration.

Section 4 introduces new modelling constructs that enables specifying relations between variables and time, such as the one above, much more concisely.

Note that for the remainder of the paper we will assume that the online problems we consider have *complete recourse* [9], that is, neither the previous solution nor the new data will ever make the problem unsatisfiable. This is typically achieved by adding penalties to the objective, e.g., for not scheduling jobs in the case of a job-shop problem.

3 Related Work

Online problems and online solution methods have been well studied. The two main approaches are (a) using an off-the-shelf solver with an ad hoc sliding window algorithm wrapped around it, and (b) developing a problem-specific algorithm. In this paper we develop a new approach by extending a solver-independent modelling language to support online problems natively.

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 et al. [4] 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 [18,23,7] for other examples. These wrapper algorithms are often problem-specific in nature, and require the model to be formulated in such a way it obfuscates the underlying problem.

Examples of approach (b) are more widespread, and include the algorithms for online vehicle routing given in survey [16], and the online scheduling algorithms described in [22]. 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 [14].

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 the modelling language AIMMS, which supports the modelling and use of sliding window decomposition (referred to as "rolling horizon") of time-based offline problems [25]. 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 [19,3].

Often in the literature, dynamic constraint satisfaction problems (DCSPs) are used to reason on online and dynamic problems [12]. DCSP is a potential formalisation of our proposed high-level modelling framework.

Note that, in this paper, we do not look into stochastic and advanced forms of dynamic online optimisation (see [27,28,2,5]) nor robustness and stability criteria (see [8]), all of which we consider future work. Some other interesting concepts include Constraint Networks on Timelines [21], Constraint Programming for Real-Time Allocation [15], and iterative repair techniques [6].

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, this algorithm contained two additional components: the functions append and constrain. The append function needs to know which arrays in the model to extend with new data. The constrain function requires the additional parameters for the previous solution to be present in the model, and relies on additional constraints in the model to implement the actual time dependencies.

We will now introduce *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 append and constrain functions, can then be generated automatically from this annotated model, as it will be discussed in Section 5.

The first new annotation, `:: online`, is used to indicate which parameters can be extended with new data at each time point. If a parameter annotated with `:: online` is used to define other parameters (e.g., it is part of an array index set), then those other parameters automatically become extendable with new data as well.

Example 1. 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 must start by annotating the parameter in line 2, obtaining the line `int: J :: online;`, thus indicating that parameter `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 key feature of online problems is that some decisions cannot be changed 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 have fixed a time variable to a value earlier than the *current time*, cannot be changed. Also, if such a variable is not yet fixed, then current decisions cannot fix it to a value that is earlier than the current time. To reflect all this, modellers can simply annotate such variables with `:: time;`.

Example 2. Continuing the job-shop example from Figure 1, the main variables of the model are the start times for each task of each job, that is, the array of variables defined in line 11. The domain of these variables is indeed time and, hence, the declaration must be annotated, resulting in the line `array[JOB,TASK] of var 0..horiz: s :: time;` □

While the domain of some variables is not time itself, it may nevertheless reflect decisions that cannot be changed after a certain point in time. We say that the 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.

Example 3. 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* a variable can take: 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 using the annotation `lock_val_time(`t`)`, where t is a one-dimensional array that corresponds to the declared domain of v.

Example 4. Consider a package delivery routing problem for C customers and V vehicles, where each customer must be visited for a delivery. 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 variables include, for each node n, the arrival time at n, the next node visited from n, and the vehicle that visits n. A partial model for an online version of this problem is as follows:

```
1  int: V :: online;  % number of vehicles
2  int: C :: online;  % number of customers
3  int: horiz :: online;  % scheduling horizon
4  int: N = C + 2*V;  % number 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..horiz: arrival :: time;
9  array[NODE] of var NODE: next
10       :: lock_var_time([ arrival[n] | n in NODE ]);
11 array[NODE] of var VEH: veh
12       :: lock_val_time([ arrival[C+v] | v in VEH ]);
```

In this model we may get new customers and new vehicles (a vehicle returning to the depot becomes available as a new vehicle). The time horizon for scheduling also changes as more customers arrive. The arrival time at each node is a `time` constrained variable (hence, line 8). The decision about where to go next from node n is locked at the time point where the vehicle arrives at n (hence, line 10). Also, since the packages must be loaded onto vehicles v at the depot, the decision of which customers v visits 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 `arrival[C+v]` (hence, line 12). □

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`.

Example 5. Consider again the problem of Example 4. When deciding which vehicle should visit each customer, it is unrealistic to add (or remove) a customer

to (or from) a vehicle if the vehicle 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, the following code (substituting that of lines 11 and 12) reflects the correct behaviour:

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

The annotations state that the decision of assigning a customer to vehicle v cannot be changed if v leaves in the next 15 minutes, and that a customer cannot be (newly) assigned to a vehicle that is leaving in the next 5 minutes. □

The above annotations define some of the most common time constraints that arise when solving an online problem. In certain problem-specific cases, where these annotations are not sufficient, the modeller is given access to the solution computed in the previous iteration via a generic function `sol(`x`)`, which returns, for each variable and parameter x, the value of x in the previous solution. This is analogous to the use of the function `sol()` in MiniSearch [24] and other extensions of MiniZinc [10] to refer to the previous solution to a problem. In addition, modellers can use the function `has_sol(`x`)` to test whether x actually existed in the previous solution, and they can make use of the `now` parameter in their time constraints.

Example 6. Consider an extension of the delivery problem in Example 4, 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. This can be expressed in the model as follows:

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

When used as an offline model, `has_sol` will return **`false`** for any argument, so the constraint will not be active. □

For simplicity, this paper assumes that execution will perfectly follow decisions, but our framework does work without this assumption. Using our job-shop example, a realistic scenario incorporating uncertainty in execution could be that a task, according to the past decisions, should have started at some time, but, according to the actual execution, the task started at another time (or perhaps not at all). To address this, `sol()` and `has_sol()` can be set, before each iteration, to reflect the execution instead of the past decisions, by passing the desired data to MiniZinc.

Finally, we discuss how the current time in the model's view of time, `now`, is defined. Since only modellers understand the relationship of `now` with time in the real world, they are the ones who must define `now` as a parameter computed using the system time calls already available in MiniZinc.

5 Transforming and Solving Online Problems

Once an online model M_O has been created, the next step is to instantiate the iterative algorithm from Figure 2, i.e., to call $instantiate(M, D)$. This involves *transforming* M_O into an update-model $M = \text{transform}(M_O)$ with additional parameters and constraints that implement the model's time-dependent aspects.

5.1 Transformation

The following describes how the annotations of an online model M_O are used by $\text{transform}(M_O)$ to generate the update-model M.

Online annotations The `online` annotation is used to generate the append function, which combines old and new data. In fact, we use MiniZinc itself to perform the append. For any parameter `p :: online`, 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, results in the following constraint:

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

which ensures the solution to x can only change if it was set to a point still in the future, and cannot change to a point in the past (in other words, 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 for the update-model:

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

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

Value annotations We show how commit and forbid annotations are transformed. (Recall that lock annotations simply combine 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 results in the following constraint:

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

This correctly ensures that if the 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 results in the following constraint:

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

5.2 Garbage Collection

The append function in the iterative algorithm from Figure 2 will construct larger and larger problems as time progresses: more and more data is added, which leads to more and more variables and constraints. In typical online problems, many variables will be fixed immediately due to their time dependencies. Even though they may therefore not have a big impact on solving time, for long running online problems this can lead to an increase in time for the instantiation phase (translating the MiniZinc model into the solver-level FlatZinc). In many cases, we can determine that some parts of the data can have no effect on the remaining solving and, hence, can be omitted. We call this *garbage collection* of old data.

For example, in the online job-shop problem of Figure 1, jobs that are completely finished by the current time can have no further effect on any jobs scheduled after the current time. Hence, they can be omitted from the data. Jobs that have started by the current time but not finished, however, can still affect new jobs, since they still use resources. Hence, they cannot be omitted.

This example shows that garbage collection is problem specific: the mere fact that the start time of a job is in the past does not mean that it does not affect the future schedule. We therefore need the modeller to express which parts of the data can be garbage collected. An interesting avenue for future work is whether this could be determined automatically from the model and its annotations in certain cases. For now, we require modellers to introduce into their models new parameters that identify which parts of the data have become obsolete. These parameters are annotated as `::online_gc` (for garbage collection).

Example 7. We can change our online job-shop model in Figure 1 slightly to enable garbage collection.

```
1 int: J  :: online;      % number of jobs
2 int: LJ :: online_gc(  % last job that may affect new jobs
3   arg_max( [ exists(i in TASK)
4                     (sol(s[j,i]) + p[j,m[j,i]] > now)
5            | j in JOB ] ++ [true]));
6
7 set of int: JOB = LJ..J;      % meaningful jobs
8 array[JOB] of int: a;         % arrival time
9 array[JOB,TASK] of MACH: m;   % machine used
10 array[JOB,MACH] of int: p;   % processing time
11 array[JOB,TASK] of var 0..horiz: s :: time;
```

Initially, we set `LJ = 1`. After instantiating the update-model, the expression inside the `online_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: discarding all irrelevant data of jobs up to (but not including) `LJ`. □

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 weakness occurs with very long-running jobs. A long-running job keeps all the jobs appearing after it in the job list alive until it has finished, even if they finished much earlier. Still the simplicity of the approach is very appealing, and if the jobs are reasonably uniform in duration, this will not be a problem.

6 Experiments and Examples

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).

6.1 Online Job-Shop Scheduling Problem

Our first experiment uses the online job-shop scheduling problem described throughout the paper. The instances used are formed from offline job-shop scheduling problems of the MiniZinc benchmarks.[3] We first take an offline instance as the initial problem. The online data is then constructed by repeatedly adding copies of the jobs from the offline instance into an endless queue.

[3] `https://github.com/MiniZinc/minizinc-benchmarks/tree/master/jobshop`

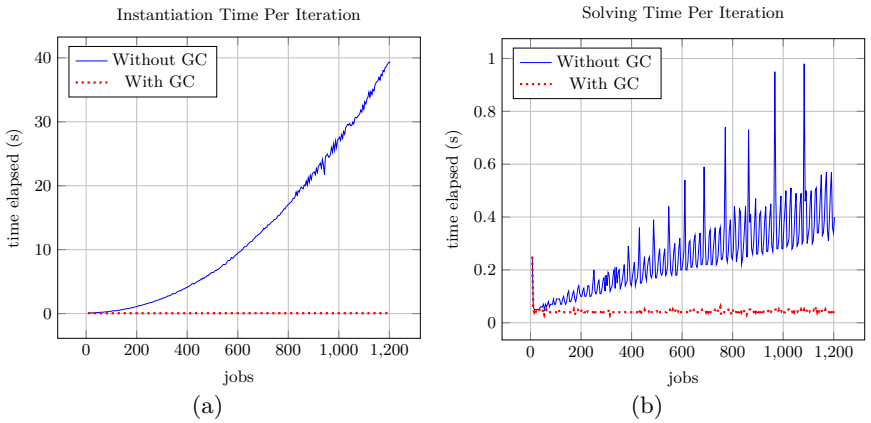

Figure 3. (a) Instantiation time and (b) solving time per iteration, with and without garbage collection (GC), for online job-shop using instance `abz5`.

Data Driven Online Problem The main focus of this experiment is to illustrate the effect of garbage collection. To achieve this, we define `now` to ensure that the solving time per iteration is significantly less than the time between two iterations. We 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. 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.

Figure 3 shows the instantiation time (a) and solving time (b) for each iteration and Table 1 (left) 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. Given these results, from now on we always use garbage collection in all experiments.

Solver Driven Online Problem This second experiment shows how our framework can deal with online problems where the duration of one iteration is in fact constrained by the time it takes to solve the update-model. Note that in this setting `now` is defined as the time at the end of the solving time limit.

Suppose a set number of jobs arrive every second, then the more time we allow the solver to spend, the more new jobs will arrive for each iteration. Since solving is now time constrained, we may not be able to obtain an optimal solution within each iteration, but simply use the best solution found within the time limit.

Table 1. Left: Accumulated instantiation and solving time with and without garbage collection (GC). Right: Time-constrained online job-shop scheduling (instance `ft20`).

	With GC		Without GC	
instance	solving (s)	instn. (s)	solving (s)	instn. (s)
`abz5`	13.01	19.00	75.64	3913.48
`ft06`	14.56	18.91	56.63	1363.91
`ft10`	17.04	19.86	121.59	3882.49
`ft20`	1467.56	18.84	1490.86	941.64

	Makespan	
time/iteration	4 jobs/s	8 jobs/s
0.25 s	**117809**	65266
0.50 s	117840	65270
1.00 s	117973	**65260**
2.00 s	118170	65493
4.00 s	118749	66272
8.00 s	119581	67904
16.00 s	121352	71168

Table 1 (right) shows the results for scheduling 1162 jobs in total (`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 second real time 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 (4 per second), then a fast resolving time of 0.25 per iteration yields the best overall makespan. With a higher data rate (8 jobs per second), a time out of 1 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.

6.2 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.

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. [1]. 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.

We modified the model in the following ways: the number of vessels becomes an online 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 2 shows results for all instances from the MiniZinc benchmarks with at least 16 vessels. We compare the offline solving approach (as a baseline) with a

[4] `https://github.com/MiniZinc/minizinc-benchmarks/tree/master/cargo`

Table 2. Cargo Assembly Planning Problems solved as a single optimisation problem and using sliding window decomposition. The symbol — under runtime indicates a time out (120s) and under obj. val. indicates no solution was found.

	Original		Sliding Window	
instance	runtime	obj. val.	runtime	obj. val.
07_1s_133	—	10563	**40.35**	**5868**
08_222f_3475	—	70068	**90.38**	**66085**
09_1s_18_OPT	—	**7117**	**0.41**	22958
10_15966f_2060	—	38119	**90.34**	**36347**
16_10720f_4243	—	—	**80.71**	**88862**
19_31058f_2548	—	141119	**77.30**	**129748**

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.

7 Conclusion and Future Work

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

Experiments using our implementation of the framework for MiniZinc show the usefulness of the approach for both online problems and sliding window decomposition. Furthermore, the experiments highlight the importance of garbage collection, i.e., removing old data that is no longer relevant.

Future work includes, among other things, extending the framework to address dynamic and stochastic online problems (i.e., where known parameters can change over time and where disruptions can occur, with or without a priori distributions or probabilities), incorporating predictions of future data while solving, automatic detection of simple garbage collection rules via model analysis, looking at stability and robustness criteria, and using dynamically sized windows and multiple passes with sliding window decompositions.

References

1. Gleb Belov, Natashia Boland, Martin W. P. Savelsbergh, and Peter J. Stuckey. Local search for a cargo assembly planning problem. In Helmut Simonis, editor,

Integration of AI and OR Techniques in Constraint Programming, volume 8451 of *LCNS*, pages 159–175. Springer International Publishing, 2014.
2. Russell W. Bent and Pascal Van Hentenryck. Scenario-based planning for partially dynamic vehicle routing with stochastic customers. *Operations Research*, 52(6):977–987, 2004.
3. Patrizia Beraldi, Antonio Violi, Nadia Scordino, and Nicola Sorrentino. Short-term electricity procurement: A rolling horizon stochastic programming approach. *Applied Mathematical Modelling*, 35(8):3980–3990, 2011.
4. Dimitris Bertsimas, Patrick Jaillet, and Sébastien Martin. Online vehicle routing: The edge of optimization in large-scale applications. *Operations Research*, 67(1):143–162, January 2019.
5. Kenneth N. Brown and Ian Miguel. Uncertainty and change. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, chapter 21, pages 731–760. Elsevier, 2006.
6. Steve Chien, Russell Knight, Andre Stechert, Rob Sherwood, and Gregg Rabideau. Using iterative repair to improve the responsiveness of planning and scheduling. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, AIPS'00, pages 300–307. AAAI Press, 2000.
7. Alistair R. Clark and Simon J. Clark. Rolling-horizon lot-sizing when set-up times are sequence-dependent. *International Journal of Production Research*, 38(10):2287–2307, July 2000.
8. Laura Climent, Richard J. Wallace, Miguel A. Salido, and Frederico Barber. Robustness and stability in constraint programming under dynamism and uncertainty. *Journal of Artificial Intelligence Research*, 49:49–78, January 2014.
9. George B. Dantzig. Linear programming under uncertainty. *Management Science*, 1(3/4):197–206, 1955.
10. Jip J. Dekker, Maria Garcia De La Banda, Andreas Schutt, Peter J. Stuckey, and Guido Tack. Solver-independent large neighbourhood search. In John Hooker, editor, *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming*, volume 11008 of *LNCS*, pages 81–98, 2018.
11. Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Cengage Learning, 2nd edition, 2002.
12. Jeremy Frank. Revisiting dynamic constraint satisfaction for model-based planning. *The Knowledge Engineering Review*, 31(5):429–439, Nov 2016.
13. Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, Sept 2008.
14. Shan He, Mark Wallace, Graeme Gange, Ariel Liebman, and Campbell Wilson. A fast and scalable algorithm for scheduling large numbers of devices under real-time pricing. In John Hooker, editor, *Principles and Practice of Constraint Programming*, volume 11008 of *LNCS*, pages 649–666. Springer International Publishing, 2018.
15. Pierre-Emmanuel Hladik, Hadrien Cambazard, Anne-Marie Déplanche, and Narendra Jussien. Solving a real-time allocation problem with constraint programming. *Journal of Systems and Software*, 81(1):132–149, January 2008.
16. Patrick Jaillet and Michael R. Wagner. Online vehicle routing problems: A survey. In Bruce Golden, S. Raghavan, and Edward Wasil, editors, *The Vehicle Routing Problem: Latest Advances and New Challenges*, volume 43 of *Operations Research/-Computer Science Interfaces*, pages 221–237. Springer US, Boston, MA, 2008.
17. Patrick Jaillet and Michael R. Wagner. *Online Optimization*. Springer, 2012.

18. BoonPing Lim, Hassan Hijazi, Sylvie Thiébaux, and Menkes van den Briel. Online HVAC-aware occupancy scheduling with adaptive temperature control. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, volume 9892 of *LNCS*, pages 683–700. Springer International Publishing, 2016.
19. Julien F Marquant, Ralph Evins, and Jan Carmeliet. 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, 2015.
20. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, volume 4741 of *LNCS*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
21. Cédric Pralet and Gérard Verfaillie. Using constraint networks on timelines to model and solve planning and scheduling problems. In Jussi Rintanen, editor, *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, ICAPS'08, pages 272–279, Menlo Park, California, USA, 2008. AAAI Press.
22. Kirk Pruhs, Jirí Sgall, and Eric Torng. Online scheduling. In *Handbook of Scheduling - Algorithms, Models, and Performance Analysis.* Chapman and Hall/CRC, 2004.
23. Katayoun Rahbar, Jie Xu, and Rui Zhang. Real-time energy storage management for renewable integration in microgrid: An off-line optimization approach. *IEEE Transactions on Smart Grid*, 6(1):124–134, January 2015.
24. Andrea Rendl, Tias Guns, Peter J. Stuckey, and Guido Tack. MiniSearch: A solver-independent meta-search language for MiniZinc. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming*, volume 9255 of *LNCS*, pages 376–392. Springer International Publishing, 2015.
25. Marcel Roelofs and Johannes Bisschop. *AIMMS: The Language Reference*, May 2, 2019 edition, 2019. Chapter 33, Time-Based Modelling. Available at: `www.aimms.com`.
26. P. Van Hentenryck, L. Michel, L. Perron, and J. C. Régin. Constraint programming in OPL. In Gopalan Nadathur, editor, *Principles and Practice of Declarative Programming*, volume 1702 of *LNCS*, pages 98–116, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
27. Pascal Van Hentenryck and Russell Bent. *Online Stochastic Combinatorial Optimization.* The MIT Press, 2009.
28. Gérard Verfaillie and Narendra Jussien. Constraint solving in uncertain and dynamic environments: A survey. *Constraints*, 10(3):253–281, Jul 2005.