Modelling and Solving Online Optimisation Problems[⋆]

Alexander Ek[1,2⋆⋆], Maria Garcia de la Banda[1,4], Andreas Schutt[2,4], 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.

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 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, then 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.

Motivation Despite the strong similarities between all online optimisation problems, current approaches to solving them are *problem-specific*. This is because it requires specifying **how time interacts** with the variables and constraints. This is usually done by a modeller, who can implement an *update-model*,

[⋆] A longer version of this paper (including experimental results) is part of the the ModRef 2019 workshop.
[⋆⋆] The lead author is the student. The other authors are supervisors of the student.

that is, a model of the problem that combines the previous solution with the newly arriving data.

Our goal is to create a high-level, problem- and solver-independent, framework that allows a modeller to express and solve models for online problems with the same richness and ease as currently possible for offline problems. This will relieve programmers and modellers from the time-consuming task of reinventing the wheel and writing error-prone online algorithms.

Contribution This paper proposes a *more generic* approach to online optimisation that enables a 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:

– a framework for the declarative modelling of online 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; and
– an implementation of the framework in the MiniZinc [20] system.

Our framework can also be used for sliding window decompositions of offline problems (see e.g., [19,1]). We also developed a garbage collection mechanism that allows removal of old data that has become irrelevant.

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 [27]. A model M of a problem can be *instantiated* with data D into a concrete COP instance $P = \text{instantiate}(M, D)$.

Running Example: Job-Shop Scheduling This paper uses MiniZinc to model problems. We assume familiarity with MiniZinc, and refer to the MiniZinc web page for a detailed syntax description: `https://www.minizinc.org/` [26].

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

```
int: M;      % number of machines
int: J;      % number of jobs
set of int: MACH = 1..M;
set of int: TASK = 1..M;
set of int: JOB = 1..J;
array[JOB] of int: a;           % arrival time
array[JOB,TASK] of MACH: m;     % machine for task
array[JOB,MACH] of int: p;      % processing time
int: horiz = max(a)+sum(p);     % latest possible time

array[JOB,TASK] of var 0..horiz: s; % start times

constraint forall (j in JOB) (s[j,1] >= a[j]);
constraint forall (j in JOB, k in 1..M-1)
  (s[j,k]+p[j,m[j,k]] <= s[j,k+1]);
constraint forall (m in MACH)
  (disjunctive ([s[j,t] | j in JOB, t in TASK where m[j,t]=m]
               , p[..,m]));
solve minimize max (j in JOB) (s[j,M]+p[j,m[j,M]]);
```

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

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.

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. 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. 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
int: now; % current time (in the model's view of time)
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);
```

Here, `sol_J` out of `J` jobs are old. The `now` parameter allows us to reason about whether a previously scheduled job has already started running or not. If it has, then the constraint constrain it to remain scheduled at the same time.

```
online-solve($M$,$D$,$\theta$):   while (new data $\delta$)
                                     $D := \text{append}(D, \delta)$
                                     $D' := \text{constrain}(D, \theta)$
                                     $\theta := \textbf{solve}(\text{instantiate}(M, D'))$
```

Figure 2. An iterative algorithm for solving online problems.

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.

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.

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 potential a 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 [28,29,2,5]) 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 online job-shop example from Section 2.

Below, we introduce *annotations* that modellers can add to a standard, offline MiniZinc model to capture online aspects in a concise and declarative way. The update-model, together with the append and constrain functions (from the iterative algorithm of Figure 2), can then be generated automatically from this annotated model. The new annotations define some of the most common time constraints that arise when solving an online problem. To address special cases not covered by these annotations, the modeller is given direct access a generic function `sol(`x`)`, which returns, for each variable and parameter x, the value of x in the previous solution.[1] 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.

For simplicity, this paper assumes that execution will perfectly follow decisions, but our framework does work either way. Using our job-shop example, a realistic scenario would 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, we just have to set `sol()` and `has_sol()` to reflect the execution instead of the past decisions.

[1] 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.

Modelling Time and Change Since only modellers understand the relationship of `now`—the current time in the model's view of time—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.

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 expendable 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 of 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. □

Modelling Decisions Regarding Time In online problems, 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 start times (line 11) are indeed time variables; hence, the declaration must be annotated, resulting in `array[JOB,TASK] of var 0..horiz: s :: time;` □

In general, the annotation of line 1 is transformed into the constraint of line 2:

```
1 var D: x :: time;
2 if has_sol(x) /\ now>=sol(x) then x=sol(x) else x>=now endif;
```

where x is a variable over domain D.

Modelling Variables Affected by 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 is *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. When annotating an array d of variables, t must 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 of 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 of line 7 the declaration `array[JOB,TASK] of var MACH: m :: lock_var_time(s);`. □

In general, the annotation of line 1 is transformed into the constraint of line 2:

```
1 var D: x :: lock_var_time(t);
2 if has_sol(x) /\ now>=sol(t) then x=sol(x) endif;
```

where x and t are variables over D and time, respectively.

Modelling Domains Affected by Time 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`, where `lock_val_time` is defined as the conjunction of them.

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. □

In general, the annotation of line 1 is transformed into the constraint of line 2:

```
1 var D: x :: commit_val_time(t);
2 if has_sol(x) /\ now >= sol(t[sol(x)]) then x = sol(x) endif;
```

where x is a variable over D, and t is an array of variables indexed by D. And, in general, a the annotation of line 1 is transformed into the constraint of lines 2–3:

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

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

5 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.

Future work includes, among other things, extending the framework to address dynamic and stochastic online problems (e.g., 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, and looking at stability and robustness criteria.

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. The MiniZinc Team. MiniZinc. `https://www.minizinc.org/`. [Online; accessed 2019-July-15].
27. 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.
28. Pascal Van Hentenryck and Russell Bent. *Online Stochastic Combinatorial Optimization.* The MIT Press, 2009.
29. Gérard Verfaillie and Narendra Jussien. Constraint solving in uncertain and dynamic environments: A survey. *Constraints*, 10(3):253–281, Jul 2005.