Composing constraint solvers
Zoeteweij, P.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 6

Job-Shop Scheduling in OpenSolver

As a case study, we demonstrate how OpenSolver can be configured as a solver for the job-shop scheduling problem (JSSP). For this purpose we will introduce a small number of dedicated plug-ins. Because we can rely on existing facilities for search and optimization, building this specialized solver involves only a modest implementation effort. Two of the new plug-ins are variable domain types, and this particular OpenSolver configuration demonstrates a technique that we refer to as constraining special-purpose domain types. We will conclude the chapter with a discussion of the pros and cons of this technique. Also, JSSP is considered to be a non-trivial problem that is representative for many scheduling problems that occur in practice. As such, this case study demonstrates that our approach leads to solvers that have a relevance beyond puzzle-type problems such as the ones we used in the previous chapters.

6.1 Introduction

The tools that are available to model a combinatorial problem as a CSP differ for various constraint solvers. The basic machinery typically includes:

- finite domain and interval representations for the domains of integer variables, and a floating-point interval representation for real numbers,
- arithmetic constraints on numerical variables,
- global constraints, such as the all\_different constraint.

Depending on the problem that we want to model, these facilities may or may not be fully adequate to construct a CSP.

In an open-ended constraint solver, we have the possibility to add new facilities. If a problem is hard to model, it may be easier to add a few special-purpose facilities, such as a new constraint. This may lead to a CSP that is much closer
to the original problem than a CSP that uses only the built-in primitives. In a library-based system, like ILOG Solver, the user can write new constraints and goals to guide the search by writing new subclasses of base classes provided by the library. In logic programming systems, like ECL/PS6, such facilities can be written in the host language, usually Prolog. Here we will be using the readily available facilities for search and optimization of OpenSolver, and complement these with plug-ins for special purpose domain types and reduction operators for the job-shop scheduling problem.

The chapter is structured as follows: In Section 6.2 we introduce the job-shop scheduling problem, and describe an algorithm for solving it. In Section 6.3 we detail the implementation of this algorithm in OpenSolver. We conclude in Section 6.4 with a discussion of our approach from a software engineering perspective. An evaluation of our implementation on a set of benchmark problems is postponed until the next chapter, where we compare it with an alternative implementation based on nested search.

6.2 The Job-Shop Scheduling Problem

A JSSP instance consists of a set of activities, and a number of machines (in general, resources). An activity is characterized by the machine that it must be processed on, and by a processing time, which specifies for how long the machine is needed. JSSP is a non-preemptive scheduling problem, which means that activities cannot be interrupted. They acquire the machines for their full processing time. Activities are grouped in jobs, where all activities of a job have to be executed in a specified order. The problem is to find for each activity an interval in which it can be executed on the specified machine, such that no two activities require the same machine simultaneously (the capacity constraint: for JSSP, the machines have a capacity of one activity), and such that the precedence constraints inside the jobs are respected. An optimal schedule minimizes the completion time of the activities that finish last.

The table in Figure 6.1(a) specifies an example JSSP consisting of three jobs, each having three activities that require three different machines. Each row of the table specifies the numbers of the machine needed for the three activities, and between parentheses the processing time of the activity. An optimal schedule for this JSSP instance is depicted in Figure 6.1(b). The three bars represent the machines, with the activities drawn on them. Black areas correspond to machines being idle.

Algorithm 6.1 is a basic JSSP solver, due to Baptiste, Le Pape, and Nuijten [BLPN01]. It is a branch-and-propagate algorithm, where branching determines the relative order of the activities, expanding a partial schedule until all activities have been ordered, and constraint propagation verifies that the current partial schedule does not violate any precedence or capacity constraints. Each activity
6.2. The Job-Shop Scheduling Problem

$A_i$ has the following data associated with it:

- its **release date**, or earliest possible starting time, denoted $r_i$,
- its **deadline**, or latest possible completion time, denoted $d_i$, and
- its processing time, denoted $p_i$.

From these follow:

- the **earliest possible completion time**, denoted $ect_i$, and
- the **latest possible starting time** of the activity, denoted $lst_i$.

The rule for step 1 of the algorithm is to select the machine that is the **critical resource**. Criticality is measured by comparing supply and demand for the resources. Supply is the time window given by the earliest release date, and the latest deadline among all activities that require the machine. Demand is their total processing time. A machine with the smallest difference between these two quantities, which is called the resource's **slack time**, is selected. The rule for step 2 is to select the activity with the earliest release date. The latest starting time is used for breaking ties.

Constraint propagation in step 3 of the algorithm narrows the time windows for the activities by increasing release dates and decreasing deadlines to enforce the precedence and capacity constraints. In [BLPN01] a number of propagation techniques are presented for these constraints. From this collection we used the following techniques:

- For two consecutive activities $A_i$ and $A_j$ of a job, we ensure that $d_i \leq lst_j$, and $ect_i \leq r_j$ to enforce the precedence constraint.

- The disjunctive constraint. For every pair of activities $A_i$ and $A_j$ that require the same machine we know that either $A_i$ precedes $A_j$, or $A_j$ precedes $A_i$. Therefore, if we find that $ect_j > lst_i$, we know that $A_j$ cannot precede $A_i$, and we can propagate the reverse constraint by enforcing $d_i \leq lst_j$, and $ect_i \leq r_j$, and similarly for the case that $ect_i > lst_j$.

- The **edge finding** algorithm, implementing further pruning for the capacity constraint by identifying activities that must execute first, or last, in a given set of activities. We implemented the variant of the algorithm described in [BLPN01]. Its time complexity is quadratic in the number of activities that require the same resource.

Edge finding was introduced by Carlier and Pinson [CP89], and provided a breakthrough in job-shop scheduling because it allowed that for the first time, the famous benchmark problem FT10 (also known as MT10) was solved.
1. Select a machine for which the activities are not fully ordered

2. Select an activity to execute first among the unordered activities of that machine. *Post the corresponding precedence constraints.* Keep the other activities as alternatives to be tried upon backtracking.


4. Iterate step 2 until all activities on the selected machine are ordered.

5. Iterate step 1 until all activities on all resources are ordered.

Algorithm 6.1: Basic algorithm for solving the job-shop scheduling problem
6.3  JSSP in OpenSolver

In this section we describe the plug-ins that were developed to implement algorithm 6.1 as an OpenSolver configuration. First, two special-purpose variable domain types were introduced.

Activity

These are data structures consisting of three integers, that hold the release date, deadline, and processing time of an activity. Branching on these variables will enumerate candidate starting times, or candidate completion times, similar to left/right-enumeration, which is illustrated in Figure 4.2(b) and (c) for finite domains. Algorithm 6.1 branches only on the order, and not on the actual timing of the activities, though, but this facility will be used in the solver of Section 7.5.4. The size reported by an Activity domain is one plus the width of its time window minus its processing time, which is the actual number of possibilities for scheduling the activity.

The command for introducing an activity $A_i$, with (initial) release date and deadline $r_i$ and $d_i$ and processing time $p_i$ is

\[
\text{AUX } A_i \text{ IS Activity } \{ r_i, p_i, d_i \};
\]

Here the keyword AUX is used instead of VARIABLE to mark an activity as an auxiliary variable.

Ranking

OpenSolver does not directly support posting and retracting constraints, as specified in step 2 of Algorithm 6.1. Instead, we introduced a domain type Ranking for exploring alternative assignments of a machine to activities. A value for a variable of type Ranking is essentially a permutation of the numbers $0 \ldots n_a - 1$ that specifies a particular order in which $n_a$ activities that require the same machine are executed. The domain of such a variable is a set of permutations. It is implemented as a data structure consisting of

- an array of length $n_a$, containing (initially in that order) the different indices $0$ through $n_a - 1$.
- an integer $n_o$, indicating that the first $n_o$ entries of the array have been ordered. The remaining $n_a - n_o$ entries are considered to be unordered.
- an index $i \in [0..n_a - n_o - 1]$, indicating a specific entry in the unordered part of the array. This is the next candidate for expanding the ordered part.
Chapter 6. Job-Shop Scheduling in OpenSolver

The search tree is expanded by splitting variables of domain type `Ranking`. This is depicted in Figure 6.2. In the left branch, the element indicated by the index $i$ is added to the ordered set, and $i$ is set to point to the first element of the new unordered set. In the right branch, the ordered set is unchanged, and $i$ points to the next candidate in the unordered set. In other words, the domain in the left branch corresponds to all permutations where the number pointed to by $i$ is the next element, and the right branch corresponds to the set of all permutations where this number is not the next element. If $i > n_a - n_o - 2$ then the `Size()` method of a `Ranking` domain will return 1, indicating that the permutation is fixed. Otherwise a value greater than 1 is returned (detailed below).

The code for introducing a `Ranking` variable for machine $m_j$ is the following.

```plaintext
VARIABLE m_j IS Ranking { n_o };
```

A number of reduction operators implement the interaction between logical variables of domain types `Activity` and `Ranking`:

**Precedes**

This propagation operator enforces the precedence constraint on two activities, as described in the previous section. The operation is similar to enforcing bounds consistency for the linear inequality constraint on integer intervals (see Chapter 5), but now operating on `Activity` domains. This operator is applied to all consecutive pairs of activities $A_i$, $A_j$ of the same job:

```plaintext
DRF Precedes { A_i, A_j };
```
RankActivities

This operator applies to a variable of domain type Ranking, plus \( n_a \) Activity variables. It is imposed on the activities that require the same machine. On the activities whose indices are in the ordered part of the Ranking data structure, the corresponding precedence constraints are enforced. A precedence constraint is also enforced on all possible combinations of the last ordered activity and an unordered activity.

The syntax for introducing the DRF that enforces the constraint that activities \( A_{i_1}, \ldots, A_{i_{na}} \) are processed on the machine with Ranking variable \( m_j \) is the following.

\[
\text{DRF} \quad \text{RankActivities} \{ m_j, A_{i_1}, \ldots, A_{i_{na}} \};
\]

We would like to remark that RankActivities, and some of the other reduction operators used in this chapter, have all the properties of a global constraint, as described in Section 4.2. Indeed, as we already discussed there, global constraints could be implemented in this way, and special-purpose domain types can be used to store any information that needs to be maintained during constraint propagation or search.

The Precedes and RankActivities operators already provide the necessary ingredients for a JSSP solver that is sound and complete. From this point of view, the other plug-ins are an optimization:

Disjunctive

The Disjunctive plug-in implements the constraint that two activities that require the same machine cannot overlap in time. It is applied to all pairs of activities \( A_i, A_j \) that require the same machine:

\[
\text{DRF} \quad \text{Disjunctive} \{ A_i, A_j \};
\]

EdgeFinding

This plug-in implements the edge finding algorithm. Its specifier string lists the \( n_a \) Activity variables that require the same machine:

\[
\text{DRF} \quad \text{EdgeFinding} \{ A_{i_1}, \ldots, A_{i_{na}} \};
\]

DecorateRanking

This is a branching operator that serves only to decorate a variable of domain type Ranking with the information that is needed to implement the variable selection strategy described in Section 6.2. Because no subdomains are created by the branching operator, we call it a pseudo branching operator. Like RankActivities, it applies to a variable of domain type Ranking, and a sequence
of \( n_a \) activities requiring the same machine. For the unordered activities, it calculates the difference between the size of the available time window, and the total processing time, as described in the previous section. Ranking variables that have not been subject to branching report this difference plus one as the size of the domain. In combination with a regular fail-first variable selection strategy, this ensures that activities on critical resources are tried first in branching.

In addition to this, the unordered part of the array of Figure 6.2 is sorted according to increasing release date and latest starting time. When splitting a Ranking domain, this results in the value section strategy for step 2 of Algorithm 6.1. The specifier string for the DecorateRanking plug-in is identical to that for the RankActivities plug-in:

\[
\text{DRF DecorateRanking}\{ m_j, A_{i_1}, \ldots, A_{i_{n_a}} \};
\]

Algorithm 6.1 computes feasible schedules instead of minimal schedules. Optimization can be realized by adding an extra step 6, which backtracks after a solution, and constrains subsequent solutions to have a shorter schedule length than the current solution. The last solution found is then the optimal schedule. The resulting branch-and-bound search is implemented by introducing an activity makespan having processing time 0, which is scheduled to start after the last activity of each job. This approach is described in [VHPP00]. Via another special-purpose operator BoundActivity, the domain of an integer interval variable is constrained to range from the release date to the deadline of this activity. The Optimize operator, introduced in Section 5.9.2, constrains the length of the schedule to decrease for subsequent solutions.

Program 6.1 shows an example configuration file for a JSSP instance. Such files are generated from JSSP specifications by a small preprocessor program. The initial release date for the makespan activity is set to the maximum processing time among all jobs and all sets of activities that are assigned to the same machine. Its initial deadline is the sum of all activity processing times. All variables except the instances of Ranking are auxiliary variables. This means that for any feasible ordering that is found, the time windows for the individual activities may be wider than their processing times. To avoid having to search actively for the minimum makespan of a given ordering, which can easily be achieved by letting all activities start on their release dates, we use an additional operator FixMakespan. For all (suboptimal) solutions encountered during the search, this pseudo branching operator collapses the time window for the makespan activity such that it can only be scheduled at its release date. The minimal schedule for a feasible permutation of activities follows. We implemented a variant of the Precedence and Disjunctive operators that apply to any number of activities. We found that this gives slightly better performance than the binary operators described above. A schedule of reduction operators, in the language of Figure 4.1, is generated to coordinate constraint propagation such that application of the expensive edge finding operators is postponed until a fixed point of the other
6.4 Discussion

Constraining Special-Purpose Domain Types

The scheduling facilities discussed here were developed primarily for the purpose of testing OpenSolver on benchmarks of arguable relevance, but we believe that the technique of constraining special-purpose data structures, like the Ranking and Activity variables, is interesting in itself. On the one hand, it illustrates the use of OpenSolver as an abstract branch-and-prune tree search engine, that can be configured in different ways for different tasks. Again, the effort of developing these plug-ins is modest compared to developing a JSSP solver from scratch. While these plug-ins have little relevance outside the specific application of scheduling, the framework allows for a seamless integration with existing facilities, notably for search, optimization, and parallel processing.

On the other hand, scheduling is an example of a combinatorial problem for which an efficient translation to regular constraint programming primitives is not straightforward. For this reason, other platforms have built-in facilities (for example, the OPL Resource data type [VH99]) for scheduling as well (implementing permutation-based JSSP solving on top of standard modeling facilities is explored in [Zho97]). In general, these "heavy-weight" application-specific domain types allow us to write constraint programs that are very close to the original problem, and are hence more easily verified to be correct. A specific advantage of our approach is that these facilities are not hard-wired in the system. We expect that the technique can be applied to other combinatorial (optimization) problems, for which a direct translation to regular constraints is not straightforward.

A disadvantage of our approach is that heavy-weight domain types like Ranking do not lend themselves naturally for domain reduction by means of constraint propagation. For implementing Algorithm 6.1 this is not problematic, but it impedes the implementation of several possible improvements of this algorithm. For example, if the disjunctive constraint implies that one activity always precedes another in a particular branch of the search tree, we cannot reduce the domain of a Ranking variable accordingly. Permutations that violate this deduced constraint will be generated over and over again (trashing), and have to be refuted by propagation of the disjunctive constraint. This may lead to substantially larger search trees.
AUX J0A0 IS Activity \{0,5,29\};
AUX J0A1 IS Activity \{0,2,29\};
AUX J0A2 IS Activity \{0,3,29\};
AUX J1A0 IS Activity \{0,4,29\};
AUX J1A1 IS Activity \{0,4,29\};
AUX J1A2 IS Activity \{0,1,29\};
AUX J2A0 IS Activity \{0,3,29\};
AUX J2A1 IS Activity \{0,3,29\};
AUX J2A2 IS Activity \{0,4,29\};
AUX makespan IS Activity \{13,0,29\};
AUX imakespan IS IntegerInterval \{\}\;

VARIABLE M0 IS Ranking \{3\};
VARIABLE M1 IS Ranking \{3\};
VARIABLE M2 IS Ranking \{3\};

DRF RankActivities \{M0, J0A1, J1A2, J2A0\};
DRF RankActivities \{M1, J0A0, J1A0, J2A2\};
DRF RankActivities \{M2, J0A2, J1A1, J2A1\};
DRF Precedes \{J0A0, J0A1, J0A2, makespan\};
DRF Precedes \{J1A0, J1A1, J1A2, makespan\};
DRF Precedes \{J2A0, J2A1, J2A2, makespan\};

DRF Disjunctive \{J0A1, J1A2, J2A0\};
DRF Disjunctive \{J0A0, J1A0, J2A2\};
DRF Disjunctive \{J0A2, J1A1, J2A1\};

DRF EdgeFinding\{J0A1, J1A2, J2A0\};
DRF EdgeFinding\{J0A0, J1A0, J2A2\};
DRF EdgeFinding\{J0A2, J1A1, J2A1\};

DRF DecorateRanking \{M0, J0A1, J1A2, J2A0\};
DRF DecorateRanking \{M1, J0A0, J1A0, J2A2\};

DRF DecorateRanking \{M2, J0A2, J1A1, J2A1\};

DRF BoundActivity \{makespan, imakespan\};

DRF FixMakespan \{imakespan, imakespan\};

DRF Optimize \{-imakespan\};

DRF FailFirst \{0, M0, M1, M2\};

SCHEDULER ChangeScheduler \{schedule =
    \{(0, 1, 2, 3, 4, 5, 6, 7, 8, 12, 13, 14, 15, 16, 17, 18),
     9, 10, 11\}\};

Program 6.1: OpenSolver configuration for the JSSP instance of Figure 6.1
6.4. Discussion

One way to avoid this problem would be to "virtually" reduce the domain of Ranking variables by recording this kind of information in the domain data structures, and preventing that the erroneous permutations are generated. This way, more and more constraint solving functionality is pushed into the basic value selection strategy offered by the domain type. This is all the more interesting from a solver cooperation perspective, but other CSP formulations are better suited for this kind of propagation. One option would be to use a Boolean variable for every pair of activities on the same machine, and to use so-called reified constraints to enforce the precedence relations that these variables encode. A reified constraint is a constraint of the form $b \leftrightarrow C$, with $b$ a Boolean variable and $C$ a constraint [SS02]. It reads "$b$ if and only if $C$," which entails that if $b$ is false, the negation of $C$ is enforced, and if $C$ becomes redundant or falsified, the domain of $b$ is reduced accordingly. Reified constraints are typically used to express logical connectives between constraints, such as $C \leftrightarrow C'$.

Search

Another issue is the flexibility of our search strategy. Languages like OPL and SALSA [LC02] offer richer facilities for specifying search procedures. To illustrate the limitations of the current set of plug-ins, consider the variable selection strategy of Algorithm 6.1. Once a machine is selected, all tasks that require it are scheduled before another machine is considered. This was implemented by manipulating the size of the variable domain that is reported by Ranking instances. When a domain of type Ranking is split, as depicted in Figure 6.2, the resulting subdomains will always report size 2. Domains that have never been split report a size greater than 2 that reflects the total slack time of the resource. As a result, a fail-first variable selection strategy will prefer variables that have been split before, which leads to the required strategy.

A more elegant implementation of the variable selection strategy would be to compose it from two basic strategies. Suppose that in addition to fail-first (FF) we have at our disposal a strategy $R$ that always selects the variable that has been selected most recently. A SALSA expression for our variable selection strategy would then be

$$(R^* \diamond FF)^* \diamond \text{cont}$$

This specifies that we keep applying $R$ until no subdomains are generated by splitting the most recently split variable. This happens at the beginning of the search. In that case we apply fail-first. This composite procedure is repeated until a leaf of the search tree is reached, where instead of terminating the search, we continue the exploration.

A similar composition can be achieved in OpenSolver through the adapter mechanism. We can implement a branching operator CompositeBranching that accommodates two, or any number of branching operators. When one of these internal branching operators does not yield any subdomains, it applies the next.
The following code would then result in the variable selection strategy for job-shop scheduling:

```plaintext
DRF CompositeBranching {
    DRF RepeatBranching { 0, MO, M1, M2 };
    DRF AnnotateVariableSelection { FailFirst { 0, MO, M1, M2 } };
};
```

where `RepeatBranching` branches on the most recently selected variable. The required variable index can be maintained using an annotation, like the `RoundRobin` plug-in does (see Section 4.1.2). Here we assume that `FailFirst` is modified to maintain this annotation through an adapter `AnnotateVariableSelection`.

Using this composition, we do not have to set the size of `Ranking` subdomains to two. If, as an experiment, we would like to reconsider the choice of the resource each time we extend the partial schedule with a new activity, we would just have to replace the above composite branching strategy with `FailFirst`. Currently we would have to modify and recompile the `Ranking` plug-in for this experiment. In OPL, Algorithm 6.1 is realized by nesting the value selection in a `while` statement, that prevents a new `Resource` to be selected while the current resource has not been completely ranked. For our experiment we would simply have to remove this `while` statement (see [VHPP00]).

For computing the slack times, and sorting the unordered part of the arrays of Figure 6.2, there are no good alternatives for using pseudo-branching operators. If we want to make this the responsibility of the branching operator, it would need to be aware of the relation between the ranking variables and the activities. This is possible, but would result in special-purpose branching operators, while currently, the problem-specific details are hidden in the pseudo branching operator `DecorateRanking`. Also we could consider to implement facilities for programmable value selection strategies, as supported by SALSA `Choice` specifications. However, it should be realized that the OpenSolver configuration language is a lower-level language than SALSA and OPL. As we discussed in Section 4.1.1, it should be seen as an assembly language, and consequently we have many options for realizing specific techniques. For the present problem, the pseudo branching operators seem a good solution. However, for a coherent set of plug-ins, languages that are closer to OPL and SALSA could be implemented on top of OpenSolver, as a compiler that generates configuration specifications.

### 6.5 Concluding Remarks

In this chapter we have demonstrated how OpenSolver can be configured as a basic solver for the job shop scheduling problem. This involves a technique that we refer to as constraining special-purpose domain types, which entails that new domain types and reduction operator plug-ins are added when problems cannot
be modeled efficiently with the facilities that are readily available. Through its open-ended nature, OpenSolver is particularly suited for this technique, and it demonstrates its use as an abstract branch-and-propagate tree search engine.

The job-shop scheduling problem is an interesting test case because it is computationally expensive, and because it is considered to be representative for many scheduling problems that occur in practice. An evaluation of the efficiency of the solver that we described here is postponed until Section 7.5.1 in the next chapter, where we compare it with an alternative optimization scheme on a set of benchmark problems.