AEON: Synthesizing Scheduling Algorithms from High-Level Models

Jean-Noël Monette, Yves Deville and Pascal Van Hentenryck

Abstract This paper describes the AEON system whose aim is to synthesize scheduling algorithms from high-level models. AEON, which is entirely written in COMET, receives as input a high-level model for a scheduling application which is then analyzed to generate a dedicated scheduling algorithm exploiting the structure of the model. AEON provides a variety of synthesizers for generating complete or heuristic algorithms. Moreover, synthesizers are compositional, making it possible to generate complex hybrid algorithms naturally. Preliminary experimental results indicate that this approach may be competitive with state-of-the-art search algorithms.

Key words: Scheduling, Constraints, Modelling, Local Search, Job-Shop

1 Introduction

Scheduling problems are ubiquitous in industrial applications and have been the topic of significant research over several decades. Effective algorithms are now available for various classes of problems and general systems are available for modeling and solving complex problems. One of the difficulties with existing tools, however, is that modelers not only need to understand their application domain, but also need to be well-versed in the algorithmic and combinatorial aspects of solving the application at hand: Indeed, two applications which look essentially similar for a modeling standpoint may require fundamentally different approaches to obtain high-quality solutions.

This work is a first step to address these limitations and to bridge the gap between high-level modeling and effective solving of scheduling applications. It presents the

Jean-Noël Monette, Yves Deville
INGI, UCLouvain, 1348 Louvain-la-Neuve, Belgium, e-mail: jean-noel.monette@uclouvain.be

Pascal Van Hentenryck
Brown University, Box 1910, Providence, RI 02912

AEON[1] system which allows high-level scheduling models to be synthesized into effective algorithms by exploiting the model structure. Models in AEON are written with traditional high-level abstractions and their structure is analyzed to synthesize scheduling algorithms tailored to the applications at hand. Users state the models and only specify the synthesizer which then generates a scheduling algorithm for a specific resolution framework (e.g. greedy search, constraint programming, or local search). Synthesizers in AEON are compositional, which makes it possible to specify hybrid algorithms naturally.

The system has a number of benefits. From a user standpoint, AEON allows modelers to focus on describing their applications at a high level of abstraction, relieving them from delving into the algorithmic aspects. Moreover, since models reveal the structure of the applications, the AEON synthesizers are in a position to exploit the wealth of scheduling research in order to derive effective algorithms. Finally, because the model is independent of the solving technology, AEON makes it possible to apply various paradigms and to develop hybridizations whose potential had been demonstrated in a variety of practical applications. From an implementation standpoint, AEON also features several innovations. First, the model analysis is extensible and allows new classes of scheduling problems to be described using standard XML formats. Second, novel synthesizers can be added naturally and compositionally. Finally, several novel abstractions simplify the tasks of writing synthesizers. In particular, AEON provides the concepts of *model view*, which enables access to a general model and its solution through an interface specialized to the model at hand.

This paper extends and generalizes the research initiated in Van Hentenryck and Michel (2007) which demonstrated how to synthesize local-search algorithms from COMET models. The synthesizers proposed herein apply to the scheduling domain, consider various paradigms to solve scheduling applications (greedy algorithms, local search, and constraint programming), and are extensible and compositional. The models are similar in style to those used in ILOG Scheduler, OPL, and COMET, and earlier systems in constraint-based scheduling. ILOG Concert also provides a modeling layer that can be extracted by various solvers, but there is no attempt to synthesize search algorithms or their hybridizations. It is also useful to contrast the research to recent work in constraint programming concerning the design of default search procedures: See, for instance, Laborie and Godard (2007) which describes a Self-Adapting Large Neighborhood Search for scheduling applications, and Refalo (2004) for the use of impacts in directing the search. While their goal is to find a search procedure robust across a variety of models, our objective is to exploit the model structure to derive an effective search procedure for the model at hand and the chosen methodology. We view these approaches as orthogonal since robust search procedures must also be available for various classes of problems. However, revealing and exploiting the model structure is one of the main contributions of constraint programming and the search algorithm may significantly benefit from a structural synthesis.

---

[1] Aeon is another name for the Greek god of time Chronos. It means forever, eternity.

```
1   range jobs = 1..nbjobs;
2   range machines = 0..nbmachines-1;
3   range tasks = 1..nbjobs*nbmachines;
4   int proc[tasks];
5   int mach[tasks];
6   int job[jobs,machines];
7
8   Schedule<Mod> s();
9   Job<Mod> J[i in jobs](s,IntToString(i));
10  Machine<Mod> M[i in machines](s,IntToString(i));
11  Activity<Mod> A[i in tasks](s,proc[i],IntToString(i));
12  forall(i in tasks)
13     A[i].requires(M[mach[i]]);
14  forall(i in jobs)
15     J[i].containsInSequence(all(j in machines)A[job[i,j]]);
16  s.minimizeObj(makespanOf(s));
17
18  GreedyTabuSynthesizer synth();
19  Solution<Mod> sol =  synth.solve(s);
20  sol.printSolution();
```

Fig. 1 A Job-Shop Model and Its Synthesizer.

The rest of this paper presents an overview of the different parts of the system. Section 2 covers the use of the system and the available abstractions and solvers. In Section 3, the architecture is presented and some characteristics of the system are highlighted. Section 4 then shows how the system can be extended to deal with other families of problems or solvers. Section 5 presents and analyzes the experimental results.

2 Modelling and Solving Scheduling Problems

Figure 1 presents an AEON model for a Job-Shop Problem (JSP) and an associated synthesizer. The initialization of the input data in lines 1–6 is not shown. The model itself is given in lines 8–16. First, a schedule object is created. Then, the objects populating this schedule are created (lines 9–11), including activities, jobs, and machines. Next the constraints are stated: Machine requirements (lines 12–13) and precedences inside jobs (lines 14–15). Finally, the objective in line 16 minimizes the makespan.

The three last lines deal with the resolution of the model. Line 18 defines the synthesizer, which, in this case, synthesizes the hybridization of a greedy heuristic followed by a tabu search. It is easy to change the synthesizer: Simply replace `GreedyTabuSynthesizer` by `CPSynthesizer` to obtain a constraint-

programming search. Line 19 applies the synthesizer to solve the model. This induces the analysis and classification of the model, the generation of the appropriate variables, constraints, and objectives for the solvers, and the execution of a search algorithm dedicated to the model. The synthesizer application produces a solution which can be queried and used in various ways. Line 19 simply prints the obtained solution.

As shown in Fig. 1, the modelling and solving parts are clearly separated. AEON features a rich set of abstractions to model a broad range of scheduling problems and constraints and objectives are stated using functions and methods. The remainder of this section reviews the set of abstractions that are available to model problems and to solve them.

The modelling classes end with "`<Mod>`" to denote that they are used for modelling[2]. Inside the system, other classes with names post-fixed with "`<CP>`" or "`<LS>`" are used for the actual search algorithms. For instance, the purpose of the class `Activity<Mod>` is completely different from the class `Activity<CP>`. Although they are associated with the same concept (an activity), the first modeling version provides methods to perform the analysis of the problem, while the CP version encapsulates variables to represent the starting and finishing dates of an activity and a constraint linking them. To simplify reading, the post-fix "`<Mod>`" is omitted in this section when it is clear that we refer to modelling classes.

Table 1 presents the modelling classes available in AEON at this stage. They are explained in the following. The central modelling class is `Schedule` (i.e. `Schedule<Mod>`). It is passed to all the other created objects and is responsible for the internal consistency of the model. To represent activities, there are two classes. `Activity` and `MultiModeActivity` represent single- and multi-mode activities respectively. At creation time, an activity receives as input a schedule, a processing time and a name. The processing time is either fixed or defined by lower and upper bounds. A `MultiModeActivity` is given the `Schedule`, the number of modes, and a name. The processing time of the modes are given separately for each mode. The methods available on activities (single- and multi-mode) allow to specify preemption, the membership to a `Job`, the resource requirements, and the precedences between activities. The requirements are mode-dependent but the remaining constraints are common to every modes. Precedence constraints can involve the start and the end of activities and jobs. They can also define delays. The aforementioned `Job` class represents groups of activities logically related. The activities are not necessarily ordered but they cannot be executed at the same time. Jobs share some features with activities: They can be grouped into other jobs and their ends and starts can be constrained with precedences. Lastly, activities and jobs can be defined as optional, meaning that their execution is not required.

Resources are represented by four classes, depending on the type of resource under consideration. The `Machine` class represents unary resources. Two activities that require the same machine cannot overlap in time. The `Resource` class represents renewable resources. At every moment, the sum of the requests of the

---

[2] In spite of a syntax similar to C++, such classes are not templated classes.

Table 1 Summary of the classes available for modelling.

Description	Classes
Schedule	`Schedule`
Activities	`Activity`
	`MultiModeActivity`
Jobs	`Job`
Resources	`Resource`
	`Machine`
	`Reservoir`
	`StateResource`
Objectives	`ScheduleObjective`
	`TaskObjective`
	`CompletionTime`
	`Lateness`
	`Tardiness`
	`Earliness`
	`UnitCost`
	`PiecewiseLinearFunction`
	`AbsenceCost`
	`AlternativeCost`
	`ModifObjective`
	`MultObjective`
	`ShiftObjective`
	`AgregObjective`
	`SumObjective`
	`MaxObjective`

Table 2 Summary of the classes available to solve.

Description	Classes
Synthesizers	`ScheduleSynthesizer`
	`CPSynthesizer`
	`TSSynthesizer`
	`SASynthesizer`
	`GreedySynthesizer`
	`SequenceSynthesizer`
	`ScheduleAnimator`
Solutions	`Solution`

activities being executed cannot exceed the capacity of the resource. On the contrary, the `Reservoir` class is used for non-renewable resources whose capacity is decreased after the execution of each activity. A minimum capacity can be defined for both the `Resource` and `Reservoir` classes. For these two classes and the `Machine` class, it is possible to define (periodic) breaks, i.e. time intervals of unavailability. The last kind of resource is the `StateResource` that represents a state of the world. The resource can only be in one state at a time. Two activities that require different states cannot overlap in time. For all kind of resources, it is possible to define sequence-dependent setup times and costs. The set of requirements of an activity (or of a mode of a multi-mode activity) has the form of a tree whose internal nodes are either conjunctions or disjunctions of simpler requests. External nodes are

the basic requirements: a required machine, some required or provided amount of a resource, some consumed or produced amount of a reservoir, or a particular state of a state resource.

Objective functions are subclasses of `ScheduleObjective`. The subclasses are either simple or compound functions. Compound functions are obtained by summing or taking the maximum/minimum of other functions, or multiplying a function by a constant. Simple functions are the classical lateness, tardiness, earliness, and, more generally piecewise-defined linear functions based on the completion time of activities and jobs. The set of simple functions includes also cost functions associated with the modes of multi-mode activities, with the absence of optional activities or with the sequence-dependent setup of resources. The global objective function is passed to the `Schedule` object with a method that specifies if the function must be minimized or maximized. The function `makespanOf` found in Fig. 1 is a shortcut for the makespan, or sum of the completion times, which is a common and important objective.

This set of abstractions makes it possible to model problems as various as classical shop problems (Job Shop, Open Shop, Flexible Shop, Flow Shop, Group Shop, Cumulative Shop, Just-In-Time Job Shop), variations of the Resource-Constrained Project Scheduling Problem (RCPSP, MRCPSP, RCPSP/max, MRCPSP/max) (Kolisch and Sprecher (1997)), the trolley problem (Van Hentenryck et al (1999)) and the NCOS and NCGS classes of MaScLib (Nuijten et al (2004)). This covers problems with different kind of objectives and different properties (disjunctive or cumulative, single- or multi-mode).

Although the modelling abstractions are able to represent a large set of problems, the set of problems that can be solved depends on the search that can be synthesized. Table 2 presents the available synthesis classes in AEON at this point. Currently, there are three underlying engines: Constraint Programming, Local Search (Tabu search and Simulated Annealing) and Greedy Search.[3] Their capabilities define the capabilities of the whole system. Based on these basis solvers, more complex solvers can also be synthesized. In particular, hybrid and animated solvers can be made out of other solvers. For instance, an animated solver wraps any underlying solver into a visual environment that shows the succession of (improving) solutions. Hybrid solvers can be as simple as a sequence of solvers, or can provide decomposition schemes parametrized by solvers for the various phases. Synthesizers also accept parameters, for instance to bound the allocated time. Line 16 of Fig. 1 shows how to create a synthesizer that is a sequence of two search algorithms. The first one greedily finds a feasible solution that will serve as initial solution to the second algorithm that is a Tabu Search.
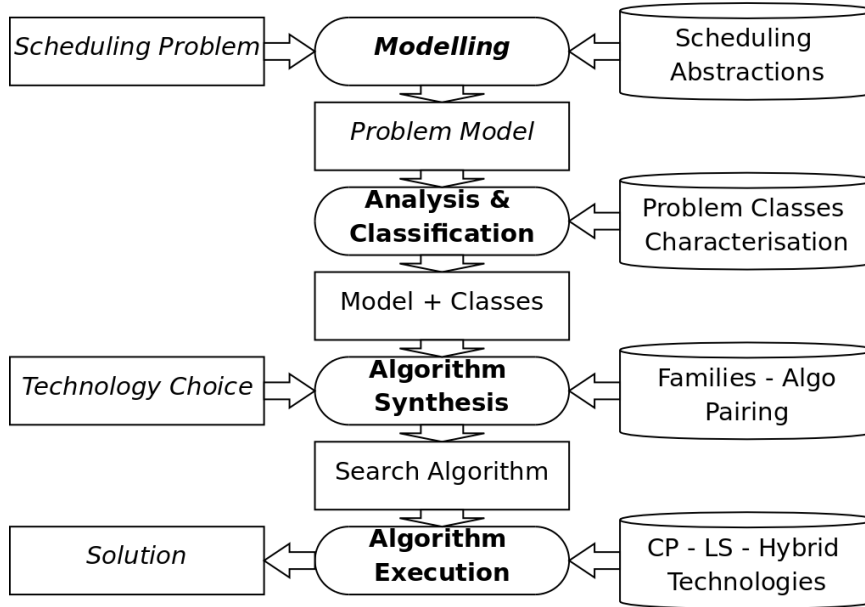
Fig. 2 Overview of AEON. Rounded boxes represent actions to solve the problem. Rectangles and Containers represent their inputs and outputs. The containers on the right are provided by the system, the rectangle on the left are inputs from, and output to, users and those between actions are transitional products. Italic text denotes user involvement.

3 Architecture

Figure 2 presents an overview of AEON centered around the resolution of a scheduling problem. Rounded boxes are the successive steps toward a solution. Only the first one, the modelling, involves the user. Subsequently, the model is analyzed and categorized into some classes of problems. An algorithm is then synthesized and run, yielding a solution to the problem. The containers on the right hand side of Fig. 2 represent what is provided by the system to perform each step. The remainder of this section explains in more details how each step is performed. Section 4 describes how it is possible to enlarge those containers.

3.1 Modelling

Section 2 presented the modelling from a user point of view. Under the cover, when the model is executed, an internal representation of the problem is built. Most of

---

[3] Future work will also consider MIP-based solvers for various classes of scheduling problems.

the information is recorded in the modelling objects that were presented. For instance, the `Activity` class contains an attribute recording whether preemption is allowed or not. In addition, the `Schedule` object keeps a reference to all objects that were created. The information is recorded using graph structures: precedence relations into a digraph, objective functions and resource requests into rooted trees. The precedence constraints are labeled arcs of a graph whose nodes represent the start and end of activities, jobs and the schedule. In addition to the arcs explicitly added by the user, arcs are created to link the start of a job with the start of contained activities and the end of the job with the end of contained activities. There are also arcs to ensure that all activities and jobs are executed between the start and the end of the schedule. The trees representing objective functions and resource requests are necessary to represent combinations of simple functions or requirements. Such combinations are sums, products, and minimums/maximums in the case of objectives. They are conjunctions and disjunctions for the requirements.

3.2 Analysis and Classification

The goal of the second step is to categorize the model into one of the "known" classes. This classification is based on problem characteristics. Each class of problems is defined by a combination of pairs (characteristic,value). Table 3 presents a subset of the considered characteristics. Additionally, the last two columns specify the values for two well-known problem classes. A dash means that the value can be anything for this class of problems. This table presents a simplified version of the definition of classes. In fact, it is not a simple conjunction of pairs characteristic-value but rather a Boolean formula, with negation, disjunction and conjunction of simpler formulas. Atoms correspond to the pairs whose truth value is determined by the analysis of the model. If the value returned by the analysis is equal to the expected one, the atomic formula receives the value true. A model belongs to a class of problem if the valuation of the formula defining this class is true.

Moreover, recurring sub-formulas are defined as higher-level characteristics or more general models from which other models may inherit. For instance, the JSP with Makespan is a special case of the JSP having the characteristic Makespan. The JSP is in turn a case of Disjunctive Problem. The hierarchy of categories forms a directed acyclic graph (DAG). This means that a problem categorized into some class is also member of all its ancestor classes. The output of the classification is thus a sequence of classes rather than a single class. This sequence represents a total order on the classes of the problem compatible with the DAG of classes. This means that, if a class inherits from another, it must appear before the ancestor in the sequence but the order of unrelated classes is arbitrarily fixed.

The analysis of characteristics in itself is achieved by a set of functions that gather information from the internal representation presented previously. Prior to the analysis, a normalization step is performed on the internal representation. In particular, the precedence graph is simplified to its transitive reduction in order to

Table 3 Partial Listing of Characteristics. The first column gives the characteristic, the second one defines the type of values. The third and fourth columns illustrate possible values for the Job-Shop (JSP) and RCPSP problems.

Characteristic	Type	JSP	RCPSP
Unit Processing Time	boolean	–	–
Fixed Processing Time	boolean	true	true
Preemption Allowed	enum	never	never
Common Release Dates	boolean	true	true
Common Deadlines	boolean	–	–
Deadlines Exist	boolean	false	false
Form of the Precedence Graph	enum	chains	DAG
Delay between Activities	boolean	false	false
No wait between Activities	boolean	false	false
Jobs inside Jobs	boolean	false	–
Number Of State Resources	integer	0	0
Maximum Capacity	integer	1	–
All Capacities are Equal	boolean	true	–
Reservoir Consumption	boolean	false	false
Reservoir Production	boolean	false	false
Setup Times	boolean	false	false
Disjunctive Requirements	boolean	false	false
All Activities in Jobs	boolean	true	false
Nb of Multi-Mode Activities	integer	0	0
Sum Of Requirements	integer	1	–
Objective Type	enum	minimize	minimize
Objective Form	enum	maximum	total
Objective Components	enum	completion time	lateness
Objective Scope	enum	all activities	one activity
All Due-Dates are equal	enum	–	–

remove useless constraints. The trees for the requirements and the objective function are also simplified. For instance, a sum of sums is replaced by a single sum. Finally, useless objects (unused machines, empty jobs, for instance) are marked as such.

To be useful, the analysis must be robust with respect to modelling variations. AEON compiles models into a canonical form and the analysis is performed on the canonical form. For instance, the code of Fig. 3 shows an alternative model of a JSP. There are several differences (multi-mode activities, reservoirs, no jobs, explicit objective function) compared to the code in Fig. 1. However, AEON correctly categorizes it as a JSP, which is a highly desirable feature in practice. Indeed, it is the semantic of the model which is significant, not the syntax of how the meaning is described.

3.3 Algorithm Synthesis

The classes responsible for the synthesis are `ScheduleSynthesizer` and its subclasses (see Table 2). As reflected in Fig. 2, the input of the synthesis step is com-

```
1   range machines;
2   range tasks;
3   int proc[tasks];
4   int mach[tasks];
5
6   Schedule<Mod> s();
7   Reservoir<Mod> M[i in machines](s,0,5,5,IntToString(i));
8   MultiModeActivity<Mod> A[i in tasks](s,1,IntToString(i));
9   forall(i in tasks) {
10      A[i].setProcTime(1,proc[i],proc[i]);
11      A[i].requires(1,M[mach[i]],3);
12  }
13  forall(i in tasks:i%nbmachines!=0)
14      A[i].precedes(A[i+1]);
15  s.minimizeObj(maxOf(all(i in tasks)completionTimeOf(A[i])));
16
17  GreedyTabuSynthesizer synth();
18  Solution<Mod> sol =  synth.solve(s);
19  sol.printSolution();
```

Fig. 3 Alternative Model for the Job-Shop Problem

posed of three parts: the user model, its classification, and a choice made by the user for a particular solving technology. The chosen subclass of `ScheduleSynthe-sizer` defines the solving technology (for instance Constraint Programming for `CPSynthesizer`) and the `solve` method takes the model as an argument.

Based on the classification output, the synthesizer chooses the appropriate solving strategy. A strategy is a search algorithm specific to a class of problems that will be instantiated to a particular instance. Each synthesizer associates different strategies with the classes of problems. For instance, the `TSSynthesizer` class associates the Tabu Search of Dell'Amico and Trubian (1993) with the class Job-Shop Problem with Makespan. Each synthesizer might not define strategies for each class of problems but it is possible that it defines a strategy for a more general problems. As the output of the classification is a sequence of problem classes, the synthesizer will look for a strategy for the first class. If it does not exist, a strategy for the next class will be looked up. The sequence is visited while there is no matching strategy. In the worst case, the problem is recognized as a "general scheduling problem" for which there is a basic default search.

Once the strategy is chosen, it must be instantiated to the problem being solved. The synthesizer delegates this work to a subclass of the `Strategy` class. Roughly speaking, there is such a subclass for each existing pair of problem class and solving technology. Each `Strategy` subclass is responsible for setting up and running a search algorithm for the problem being solved. The difficulty is that although the class of the problem is known, it may be hard to find the suitable information to instantiate the search. To facilitate this step, AEON features a set of classes called

```
1   Solution<Mod> solve(Schedule<Mod> sched){
2     JobShopView view(sched);
3     range Activities = view.getActivities();
4     range Jobs = view.getJobs();
5     range Machines = view.getMachines();
6     int[] duration =
7             all(i in Activities) view.getProcessingTime(i);
8     int[] machine = all(i in Activities) view.getMachine(i);
9     int[][] jobAct =
10            all(j in Jobs) view.getOrderedActivitiesOfJob(j);
11
12    JobshopAlgorithm  ls(LocalSolver(),Activities, Jobs,
13                     Machines, duration, machine, jobAct);
14    ls.solve();
15    SolutionView sol(view);
16    ls.saveSolution(sol);
17    return sol.getModelSolution();
18  }
```

Fig. 4 Solving a Job-Shop Problem

views. The views are used to present the schedule and its components in a unified way, no matter how they were introduced. Different views correspond to different conceptions of the problem. The most general (`ScheduleView`) is a generic way to access the information, while specific views give direct access to the subset of useful information for some classes of problems. For instance, the `JobShopView` gives information for JSPs. It is meant to give the same interface no matter how the problem was modeled by the user (whether it was specified as in Fig. 1 or as in Fig. 3).

3.4 Algorithm Execution

The algorithm being actually run is different for each strategy. However they have in common that a solution is returned. Objects of the class `Solution` assign a value to each decision variable of the problem. This assignment is expressed in terms of the model objects. For instance, the method `getStartingTime(Activity<Mod> act)` returns the starting time of an activity. Beside the starting time, other decision variables of activities are the completion time, the set of resources effectively used, the mode (for multi-mode activities) and the presence or absence (for optional activities). The solution records also the value of the objective function under this assignment. The main benefit of solution objects is that the model stays independent. It can thus have several solutions that can be compared. Moreover, solutions serve to communicate between cooperating strategies.

They can be used to perform an initial assignment, to provide an upper bound, or to guide heuristics.

Solutions are expressed in terms of the model but strategies deal with views. They need a `SolutionView` to express the solution in terms of the view. A `SolutionView` object is created from a view and the values for the decision variables are given in terms of this view. The underlying solution in terms of the model can then be retrieved from the solution view. Figure 4 shows the body of the `solve` method of the `DellAmico` class. It features the `JobShopView` and `SolutionView` classes. Line 2-10 shows the creation and the use of the view for Job-Shop problems. The actual search is delegated to another class named `JobshopAlgorithm` (lines 12-14). Line 15 creates the view for the solution from the view for the problem. This view is then fulfilled in line 16 and the actual solution is returned in line 17.

4 Adding Classes of Problems and Strategies

As the main concern of this work is to simplify the use of scheduling algorithms, it is also important to provide simple mechanisms to extend the system. In particular, the AEON architecture allows implementors to easily add classes of problems, synthesizers, and solving strategies. The extension of the modelling abstractions is not covered as it may need deeper modifications into the system. New classes of problems can be specified using XML files. Synthesizers and strategies are defined by extending existing classes.

4.1 Adding Classes of Problems

All classes of problems and high-level characteristics are defined in XML files. Each class is defined by its unique name and a structure of constraints that the problems of this class must respect. This structure is recursively made of the following elements:

- SimpleConstraint: The characteristic must have a given value.
- And: All constraints must be respected.
- Or: At least one constraint must be respected.
- Not: The constraint cannot be respected.
- IsA: The constraints of another given model must be respected.

The root element is called "Constraints" and corresponds to an "And". To add a new class, it is necessary to write an XML file that defines the constraints to satisfy. It is simple to reuse previous model thanks to *IsA* inheritance construct. For instance, Fig. 5 shows the file for the particular case of a JSP with two jobs that can be solved in polynomial time (Akers and Friedman (1955)). It is a conjunction of constraints,

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE Model SYSTEM "models.dtd">
3  <Model ID="JobShopWithMakespanWithTwoJobs">
4    <Constraints>
5      <IsA Name="Makespan"/>
6      <IsA Name="JobShop"/>
7      <Constraint Name="nbJobs" Value="2"/>
8    </Constraints>
9  </Model>
```

Fig. 5 XML Definition of the Job-Shop with 2 jobs.

namely that it is a Job-Shop problem, that the objective function is the makespan, and that the number of jobs is two.

4.2 Adding Strategies

A new search strategy is created by extending class `Strategy`, which requires specifying two methods: `solve(Schedule<Mod> s)` and `solve(Schedule <Mod> sched, Solution<Mod> initSol)`. The first method implements the resolution of the problem from scratch and the second one solves the problem, starting from an initial solution. This initial solution may be discarded, for instance in the case of a greedy solver. The body of these methods should use views. This is illustrated in Fig. 4, which shows the implementation of the first method. The implementation of the second one is similar. The only modification is the replacement of line 14 by the instruction `ls.solve(new SolutionView(view,initSol))` where a view of the initial solution is forwarded to the search algorithm.

A newly created strategy must then be linked to a class of problem by mean of a synthesizer. This pairing is done by the method `registerStrategy(string name, Strategy strategy)` defined in the `Synthesizer` class. This method associates a class (defined by its name) to a strategy. If another strategy was already associated with a synthesizer, the new one replaces the old one. This method is typically called inside the constructor of a new class of synthesizer. Figure 6 shows such a case, where a new synthesizer is defined as a subclass of `TSSynthesizer`. This means that a JSP with two jobs will be solved using the ad-hoc polynomial algorithm and all other problems will be solved with Tabu Search.

From this example, it is clear that the user choice for a particular search technology (in Fig. 2) can also be removed, allowing a completely black-box search. It suffices to create a default synthesizer with each class of problems, choosing the best possible strategy for each problem subclass. However, the "best" strategy is not necessarily unique even for a subclass: it may depend on the time constraints, the

```
1  class MySynthesizer extends TSSynthesizer{
2    MySynthesizer():TSSynthesizer(){
3      registerStrategy("JobShopWithMakespanWithTwoJobs",
4                  new AkersAndFriedmanAlgorithm());
5    }
6  }
```

Fig. 6 Adding a new strategy to the TSSynthesizer

need to obtain lower and upper bounds, the desire for optimality, and the characteristics of the instances at hand. So providing the synthesizers increase the flexibility and effectiveness of the system.

4.3 Building New Strategies Compositionally

New strategies can also be built from simpler strategies. The architecture of AEON allows implementors to build composite searches by specialization or composition. The first possibility is to create a new strategy for a specialized class as shown in the previous subsection. At a more general level, a synthesizer can systematically create compound strategies from other synthesizers. Figure 7 presents the two possibilities for a simple compound: a Tabu Search followed by a CP search. Lines 1–14 illustrate a compound strategy for the JSP and lines 15-26 show the code of a synthesizer chaining TS and CP. The methods `classify` and `solve` with several argument are defined in `ScheduleSynthesizer` and represent the different steps under the responsibility of the synthesizer: the classification and the resolution (with and without initial solution). It is interesting to see how the code of the compound synthesizer mimics the code of the compound strategy.

5 Experiments

The goal of this section is to show that the genericity of the system is compatible with effective and efficient solving of scheduling problems. To assess this, we chose to perform experiments on a few classical benchmark, the Job-Shop Problem with Makespan minimization (JSP), the Open-Shop Problem with Makespan minimization (OSP) and the Job-Shop Problem with total weighted tardiness minimization (JSTWT). For each benchmark, three synthesized search algorithms will be considered: A Local Search (LS), a Constraint Progamming approach (CP) and a compound where the Tabu Search gives an upper bound to the CP part (LS+CP). They will be compared with the COMET implementation (Van Hentenryck and Michel

```
1   class TS_CPJSP extends Strategy{
2     Strategy _s1;
3     Strategy _s2;
4     TS_CPJSP():Strategy(){
5       _s1 = new DellAmico();
6       _s2 = new CPJobShop();
7     }
8     Solution<Mod> solve(Schedule<Mod> s){
9       return _s2.solve(s,_s1.solve(s));
10    }
11    Solution<Mod> solve(Schedule<Mod> s,Solution<Mod> initSol){
12      return _s2.solve(s,_s1.solve(s, initSol));
13    }
14  }
15  class TS_CPSynthesizer extends ScheduleSynthesizer{
16    ScheduleSynthesizer _s1;
17    ScheduleSynthesizer _s2;
18    TS_CPSynthesizer():ScheduleSynthesizer(){
19      _s1 = new TSSynthesizer();
20      _s2 = new CPSynthesizer();
21    }
22    Solution<Mod> solve(Schedule<Mod> s){
23      string[] models = classify(s);
24      return _s2.solve(s,models,_s1.solve(s, models));
25    }
26  }
```

Fig. 7 Two implementations for a TS+CP strategy

(2005)) of respectively the Tabu Search of Dell'Amico and Trubian (1993) for JSP, the Tabu Search of Liaw (1999) for OSP and a Metropolis algorithm presented in Van Hentenryck and Michel (2004) for JSPTW.

The LS algorithms are counterparts of the original algorithms and have the same limits : 12,000 iterations for JSP and OSP and 600,000 iterations for JSPWT. The CP search is limited in time to $\max(300, 3 * \#\text{activities})$ seconds, that is 25 minutes for the largest instances.

For local search algorithms, 20 runs for each instance were performed. The algorithms involving CP were only run once as they are much less variable. All runs were performed on a Intel Core 2 Duo, 1.66Ghz with 1 Gb of RAM.

Table 4 presents a summary of the results for classical benchmark instances. More detailed results are available online[4]. For each algorithm, the mean relative error (MRE) is given. The MRE is equal to $100 * (UB - LB)/LB$ where UB is the value of the (average) makespan found by the algorithm and LB is the best lower bound known for the instance (taken from Zhang et al (2008, 2007); Van Hentenryck and Michel (2005); Laborie and Godard (2007)).

---

[4] http://becool.info.ucl.ac.be/aeon

Table 4 Mean Relative Error and running time (in seconds) for 4 algorithms. Ref. stands for the references algorithms, LS for Local Searches embedded in AEON, CP for Constraint Programming embedded in AEON and LS+CP for a compound of LS and CP. For CP and LS+CP, the number in parenthesis is the number of instances for which the search was complete and for which the running time is counted. For OSP, the column CP counts 2 values. The second one is a Large Neighborhood Search that can also be generated in AEON.

Problem	#Inst.	Average MRE				Average running time to best solution			
		Ref.	LS	CP/LNS	LS+CP	Ref.	LS	CP/LNS	LS+CP
JSP	78	2.08	2.09	54.40	2.03	2.6	3.1	4.4(30)	3.4(52)
OSP	80	1.68	1.70	1.58/0.01	0.85	24.1	25.0	8.0(49)/ <120	30.2(50)
JSPTW	22	4.28	3.87	97.88	4.14	24.4	24.3	-(0)	-(0)

To show another hybrid approach, we also generated a Large Neighborhood Search (LNS) for the OSP. This search is particularly efficient and solved all but one of the 80 instances in less than 2 minutes, yielding a MRE of 0.01 as shown in column CP/LNS of Table 4.

This table shows that there is no significant difference between the search generated by AEON and a search that is written apart. Of course the CP approach is not always usable for larger instances but it is not restricted to AEON. On the contrary, the use of CP in conjunction with the Local Search permits to prove optimality of heuristically found solutions. In terms of running time, the CP approach is not competitive for large problems but the local search (LS) is competitive with the COMET implementation of the reference algorithms.

The cost of the use of AEON is illustrated on Fig. 8 where the total time used by the setup, analysis, classification and generation operations is reported in function of the number of activities in the problem. The plot exhibits a quadratic progression with a soft slope. For instances containing 500 activities, the classification time is less than 1.5 second.

6 Conclusion

This work presents AEON, a system to model and solve scheduling problems. Given a scheduling model specified in a high-level modeling language, AEON recognizes and classifies its structure, and synthesizes an appropriate search algorithm. The synthesized algorithm is specialized to a particular paradigm such as local search or constraint programming. The approach makes it possible to exploit structural information from the models to derive scheduling algorithms dedicated to classes of problems.

AEON has a number of fundamental features: First the model classification does not depend on the syntax or on the modeling choices. Models are transformed into a canonical form on which the analysis is performed, increasing the robustness of the modeling process. Second, AEON is an open and extensible system: New problem classes can be specified in standard XML format and new solving strategies can be
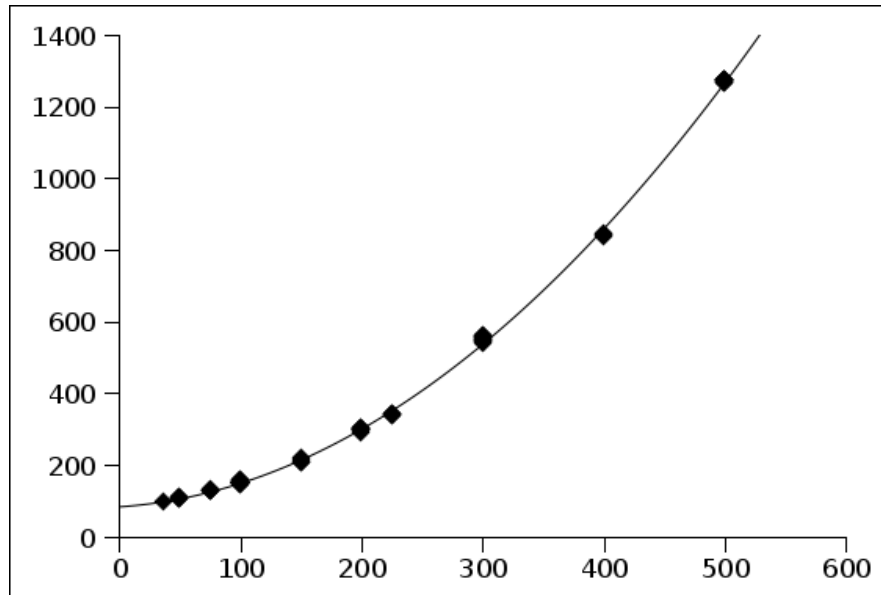
Fig. 8 Time (in milliseconds) to analyze a problem and generate the search in function of the number of activities.

added for all problem classes. Moreover, new synthesizers can be built from existing ones compositionally, building sequences of solvers or specializing decomposition algorithms (e.g., logical Benders decomposition) with different algorithms.

The experimental results demonstrated the feasibility of the approach. They show that the overhead of using AEON compared to dedicated algorithm is small and that the analysis cost is perfectly acceptable and grows quadratically with the problem size, taking about 1.5 seconds for 500 activities.

Our current work aims at defining a wide variety of scheduling algorithms for many problem classes. The inclusion of new algorithms, including large neighborhood search and parallel version of existing algorithms, is also under way and experimental results should be available soon. Long-term research will focus on two main directions. First, the automatic linearization of models will allow us to solve them using MIP technology or to obtain linear relaxations to provide lower bounds or guide the search. Second, robust default search should be built for various general classes of problems (e.g., disjunctive scheduling) when idiosyncratic constraints are present.

Acknowledgments

References

Akers S, Friedman J (1955) A non-numerical approach to production scheduling problems. Operations Research 3:429–442

Dell'Amico M, Trubian M (1993) Applying tabu search to the job-shop scheduling problem. Annals of Operations Research 41:231–252

Kolisch R, Sprecher A (1997) Psplib — a project scheduling problem library. European Journal of Operational Research 96:205–216, URL citeseer.ist.psu.edu/kolisch96psplib.html

Laborie P, Godard D (2007) Self-adapting large neighborhood search: Application to single-mode scheduling problems. In Proceedings MISTA-07, Paris

Liaw CF (1999) A tabu search algorithm for the open shop scheduling problem. Computers and Operations Research 26:109–126

Nuijten W, Bousonville T, Focacci F, Godard D, Le Pape C (2004) Towards an industrial manufacturing scheduling problem and test bed. PMS URL http://www2.ilog.com/masclib/

Refalo P (2004) Impact-based search strategies for constraint programming. In: CP 2004, Toronto (Canada), pp 557–571

Van Hentenryck P, Michel L (2004) Scheduling abstractions for local search. In: CP-AI-OR'04, Nice, pp 319–334

Van Hentenryck P, Michel L (2005) Constraint-Based Local Search. The MIT Press

Van Hentenryck P, Michel L (2007) Synthesis of constraint-based local search algorithms from high-level models. AAAI'07, Vancouver, British Columbia

Van Hentenryck P, Michel L, Laborie P, Nuijten W, Rogerie J (1999) Combinatorial optimization in OPL studio. In: Portuguese Conference on Artificial Intelligence, pp 1–15, URL citeseer.ist.psu.edu/article/vanhentenryck99combinatorial.html

Zhang CY, Li P, Rao Y, Guan Z (2007) A tabu search algorithm with a new neighborhood structure for the job shop scheduling problem. Computers & Operations Research 34:3229–3242

Zhang CY, Li P, Rao Y, Guan Z (2008) A very fast ts/sa algorithm for the job shop scheduling problem. Computers & Operations Research 35:282–294