Slot Models for Schedulers Enhanced by Planning Capabilities

Roman Barták[*]

Charles University, Faculty of Mathematics and Physics
Institute for Theoretical Computer Science
Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic
bartak@kti.mff.cuni.cz
http://kti.mff.cuni.cz/~bartak

Abstract. Scheduling is one of the most successful application areas of constraint programming and, recently, many scheduling problems were modelled and solved by means of constraints. Most of these models are confined to a conventional formulation of a constraint satisfaction problem that requires all the variables and all the constraints to be specified in advance. However, many application areas like complex process environments require a dynamic model where new activities are introduced during scheduling. In the paper we propose a framework for modelling dynamic scheduling problems where some activities can be introduced during scheduling. We also show how problems like scheduling alternatives, set-ups, and processing of by-products can be modelled in this framework.

1 Introduction

Planning and scheduling are closely related areas that attract a continuous attention of optimisation community for many years. Planning deals with finding plans to achieve some goal or, more precisely, with generating a sequence of activities that will transfer the initial world into one in which the goal description is true. Scheduling solves a bit different task of allocating known activities to available resources over time. Briefly speaking, planning helps answer the question, "What should I do?" while scheduling helps with the question "How should I do it?"

Due to a different character of planning and scheduling tasks, i.e., generating vs. allocating activities, it is a common practice that both tasks are solved separately using rather different methods. This is not intelligible to an industrial community where planning is just a form of long-horizon scheduling or, in other words, scheduling is a high-resolution planning. Consequently, there is a strong demand for Advanced Planning and Scheduling (APS) systems integrating both planning and scheduling tasks.

In [2] we presented a general framework for integrating planning and scheduling (see Section 3 for a survey) and in the present paper we show how to realise this framework in practice. In particular we deal with the scheduling problems that require

[*] Supported by the Grant Agency of the Czech Republic under the contract no. 201/99/D057.

introduction of activities in course of scheduling. Because generation of activities is a planning task, we are speaking about scheduling enhanced by planning capabilities. Nevertheless, it is possible to call this approach planning under resource constraints [8] as well.

We use constraint programming (CP) as the main technology behind the proposed scheduling engine simply because it provides rich modelling capabilities that cannot be reached by other approaches like operation research. In CP, the problem is formulated by stating constraints over problem variables and these constraints can be of mathematical (equations, comparisons, etc.), logical (conjunctions, implications, etc.), or symbolic (tabular relations…) nature. This expressive power is very important because many real-life problems can hardly be modelled using single type of constraints[1]. Moreover, the mechanism of constraint propagation provides a powerful way of reducing the search space and special scheduling techniques like edge finder can be described as constraint propagation [1].

The main difficulty of CP when applied to mixed planning and scheduling problems is the static formulation of constraint satisfaction problem (CSP)[2]. However, when planning is involved, it is impossible to predict which actions will be used in which combinations [10]. This is the main reason why CP is rarely used in planning; the works [7,10] are still white crows there. Recent development of Dynamic CSP [12] or Structural CSP [9] shows that the constraint community is aware of such dynamic real-world problems. In Section 2 we compare the static and dynamic formulation of scheduling problems and we give examples where dynamic formulation is necessary. This also justifies why scheduling enhanced by planning capabilities is a useful approach to solve real-life problems.

In the proposed approach we expect some activities to be introduced during scheduling (this is the planning task inside scheduling). The activities can be generated dynamically but we lose the power of constraint propagation then. Therefore we propose using slots to reserve space for activities - the slots are being filled by activities in course of scheduling. Because slots can contain (some) activity variables even if the activity in the slot is not known yet, we can post constraints among these variables earlier and thus we can exploit constraint propagation. This is the main advantage over dynamic introduction of activities because early constraint propagation can reduce the search space dramatically and thus it speeds up the scheduler. Note also, that our slot model is different from the traditional slot models used in timetabling because we allow the slots to slide over time. This is necessary to keep reasonable memory requirements when doing industrial scheduling [3]. The slot models are described in detail in Section 4. We introduce the slot chains and we classify the variables and constraints here. In Section 5 we sketch how to solve several problems that appear during scheduling with slot models.

[1] For example, in scheduling applications we use an equation to bind start time, end time, and duration of the activity, a disjunction to express alternatives, and a tabular relation to describe transitions between activities.

[2] Operation research models are confined to a static formulation of the problem as well.

2 Static vs. Dynamic Problem Formulation

Conventional Constraint Satisfaction Problem (CSP) is defined statically, i.e., all the variables and all the constraints are expected to be known in advance. Therefore, most methods for solving CSP follow the schema:

1) introduce all the variables,
2) post all the constraints,
3) label all the variables respecting the constraints.

Naturally, this schema is reflected in constraint models of real-world problems, i.e., we expect all the entities (variables) and relations among the entities (constraints) to be specified before we start solving the problem. Because neither the variables nor the constraints are changing during the process of solving the problem, we are speaking about a *static problem formulation.*

A pure scheduling problem, i.e., the allocation of known activities to available resources over time, can be formulated statically in the above sense. Each activity is described by a set of parameters/variables (e.g., a start time, a resource etc.) and the scheduling task is to find values of these parameters, typically to find when and where the activity is processed. Before the scheduling starts, the parameters are bound by posting constraints restricting the possible combinations of parameters' values.

The static formulation has the advantage of exploiting fully the constraint propagation because all the constraints are posted in advance[3]. The following example shows how constraint propagation reduces domains of variables and, thus, it prunes the search space.

Example: Let all the resources be single capacity resources, i.e., a single activity can be processed at given time, and let activities *a1* and *a2* have duration 5 and 4 time units respectively. To express the relation that these two activities cannot overlap when scheduled to the same resource we can use the following constraint (dot notation identifies activity parameters):

```
a1.resource = a2.resource ⇒
     a1.start+5 ≤ a2.start ∨ a2.start+4 ≤ a1.start
```

Such constraint is specified for every pair of activities to express the resource capacity restriction[4].

Now, if both activities are processed at the same resource, *a1.start* is in the interval 3,..,5 and *a2.start* is in the interval 2,..,12 then the constraint propagation reduces the domain of *a2.start* to the interval 8,..,12 (*a2* cannot be completed before *a1* so it must start after *a1*).

The constraint propagation is not directional, for example when *a1.start* is in the interval 3,..,5 and *a2.start* is 7 or 8, the constraint propagation deduces that the following constraint holds:

```
a1.resource ≠ a2.resource
```

[3] Other solving methods like local search require the static problem formulation as well.

[4] Even better constraint propagation can be achieved when single global constraint over all the activities is used instead of the set of simple constraints [1].

Despite the advantages of the static problem formulation there exist real-life problems that cannot be modelled statically because of efficiency issues. Then we are speaking about a *dynamic problem formulation* where introduction of new constraints and new variables is allowed during variable labelling. In particular we allow adding new activities, i.e., chunks of variables and constraints connecting them, during scheduling. This approach should not be confused with the dynamic constraint satisfaction [12] that tries to revise a current variable assignment with given changes in the constraint graph. In the slot representation described below we are refining the given constraint graph by adding more variables describing new activities. From this point of view, our approach is closer to structural constraint satisfaction [9].

2.1 When the dynamic problem formulation is appropriate?

The dynamic formulation of scheduling problems, i.e., enhancing scheduling by planning, is motivated by requirements of complex process environments. In industries like chemical, pharmaceutical, or food factories, the appearance of (some) activities in the schedule depends on allocation of other activities. In static models, these process-dependent activities are usually modelled using dummy activities that can be omitted (deactivated) by setting the activity duration to zero (or using another technique like a Boolean indicator of usage). However, dummy activities add overhead which is intractable in large-scale scheduling problems so we propose to use slot representation instead of dummy activities to remove the overheads.

Typical examples of process-dependent activities can be identified in chemical, food, or steelmaking industries where it is necessary to insert *special set-up, cleaning, or re-heat activities* between production activities. In many current scheduling systems, set-ups are modelled by transition times between two consecutive activities [13]. Unfortunately, this model cannot be applied when set-up activities produce some low-quality products that must be stored or consumed by other resources[5]. In [11] another example of process-dependent activity is studied, in particular re-heat activity that is modelled using dummy activities.

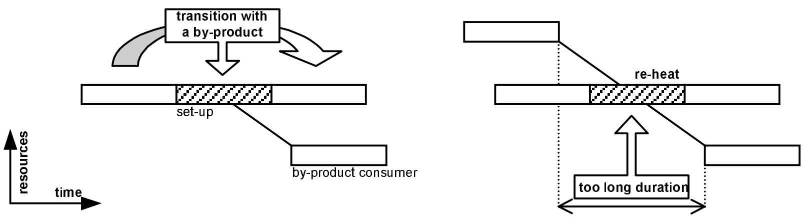

Fig. 1. Set-up (left) and re-heating (right). In both cases a special activity (striped rectangle) is necessary because this activity is connected to activities in other resources.

[5] Typically, these so-called by-products are not assumed in scheduling systems, which may cause problems with storing if a big quantity of by-products appear. Moreover, the by-products can be used in further production as raw material that may decrease the production cost.

Existence of many *alternative activity chains* is another feature of complex process environments. Usually, single activity chain is chosen during planning so the scheduler gets a “fixed’ sequence of activities to be allocated. However, as noted in [2] this approach requires the planner to have information about possible activity allocation because otherwise, there could be conflicts among chosen activity chains. These conflicts are detected during scheduling but resolved during planning by selecting another alternative. Thus, backtracking from scheduling to planning is required. There is an attempt to postpone choosing the alternative activities to the scheduling stage. In [4], a method of scheduling alternative activities is proposed. This method requires all the alternative activities to be generated in the form of a process plan. Unfortunately, if the number of alternatives is large, like in complex-process environments, then the process plan is huge and, thus, intractable. But, if the activities are introduced on demand during scheduling then there are no memory overheads.

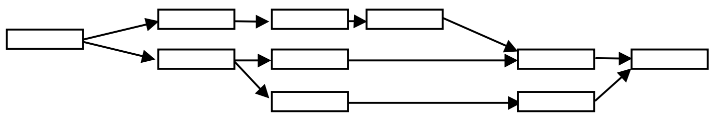

Fig. 2. A process plan with alternatives (each rectangle corresponds to an activity). Exactly one path from the leftmost to the rightmost activity will be selected during scheduling.

Last but not least there is a *non-ordered production*. In complex-process environments there exist plants where the ratio of ordered production is very low, say less than 20%, and the remaining production is driven by a marketing forecast. It is possible to introduce virtual orders describing the marketing forecast so the planner generates activities according to these orders. However, there is a danger that the marketing plan is too tighten and it cannot be scheduled together with given set of orders or the production cost is too high (e.g. due to many expensive set-ups). Therefore it seems to be more appropriate to postpone decision about non-ordered production until the scheduling stage when we may choose from several alternative marketing process plans. For example we may schedule continuation of a production of some item even if there is no demand for the item because it is less expensive to continue in the production than stopping the machine.

3 Integrated Planning and Scheduling

In the above paragraphs, we gave several examples where appearance of the activity in the final schedule depends on allocation of other activities. We also argued here for a dynamic introduction of activities during scheduling (instead of using many dummy activities) and for postponing decisions about alternative activities from planning to scheduling (a thorough discussion can be found in [3]). From the global point of view, we argue here for systems doing planning task (introduction of activities) and scheduling task (allocation of activities) together. The architecture of such mixed planning and scheduling systems was proposed in [2]. The functional decomposition to the module for generating activities (planner) and to the module for allocating activities (scheduler) is preserved here but both modules co-operate more tightly. In particular, as soon as the generator introduces a new activity, this activity is passed in the allocator that composes the new activity into the current (partial) schedule. Information about allocation of already introduced activities can be used when deciding about next activities. This is a big advantage because the planing module can exploit information from the scheduling module even if this information is partial only[6]. Moreover, the scheduler may ask explicitly the planner to introduce new activities.

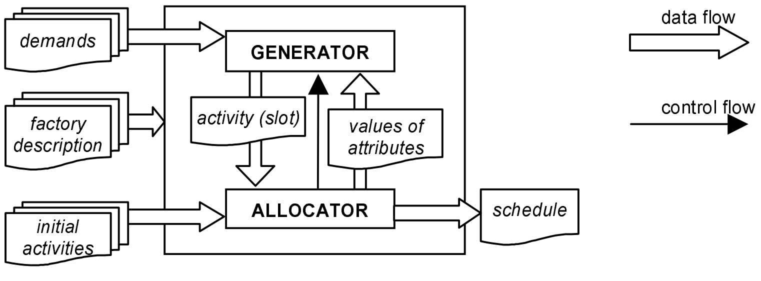

Fig. 3. The structure of mixed planning and scheduling system

The decomposition into the generator and the allocator is functional only, in the real system both functions can be governed using the same mechanism. In the slot representation, described in the next section, we expect to use slots as a shell for an activity. Introducing a new activity means deciding which activity will be filled in the slot. Using a special *Activity variable* describing which activities can be filled in the given slot does the job. Making this variable a singleton corresponds to generating the activity so no special activity generator is used. Both activity allocation (deciding about times and resources) and activity generation (deciding about the value of the Activity variable) is done using the constraint satisfaction technology.

4 Slot Models for Dynamic Activities

Slots are used in timetabling and rostering applications to represent a field where an activity can be placed. Usually, the definition of slots is part of the problem formulation - the slots are assigned to a given resource and they are fixed in time. For example, in school timetabling, there are time slots for each classroom representing time intervals when lectures can be given. If a lecture (=activity) is allocated to a given slot then the lecture occupies the whole slot. It is not possible to have one

[6] The allocation is done gradually by restricting domains of activity parameters (start time etc.). Until the domain is a singleton, only partial information about activity allocation is available (e.g. interval for the start time).

lecture completing in the middle of the time slot and another lecture starting there. If this is the case then the slot is divided into two or more slots and a single lecture can occupy more consecutive time slots if necessary.

Inspired by the slots in the traditional timetabling model (which is static in its nature) we generalise the notion of slot to work better with activity based models [3]. Still, slot is a field where an activity can be placed. The new thing here is that the slots can flow in time or between resources. In fact, the slots behave like activities so they are allocated to resources over time during scheduling. What is the difference from the activity then? In Section 2 we gave several examples where an activity is not known until we allocate some other activities. Still, we know that there will be some activity in the schedule but we do not know which one yet. So, instead of waiting until we identify which activity should be introduced we generate a slot to reserve the time and space for the activity. Because the slot can use some activity parameters like start time and duration it can participate in constraint propagation and vice versa, constraint propagation can restrict further which activities can be filled in the slot. Consequently, even if the activities are introduced dynamically during scheduling, we can exploit the power of constraint propagation.

4.1 Slot Chains

In a typical scheduling problem the activities are grouped in a sequence, for example a process plan describes the sequence of activities necessary to produce an ordered item (or a set of alternative sequences is used). We may group the slots in the same way so we are working with slot chains instead of individual slots that are not connected.

In [5] two different groupings of activities are identified, grouping per task (task-centric model) and grouping per resource (resource-centric model). The slot representation proposed in the present paper can be mapped to both groupings but its advantage is more evident in the resource-centric models where the number of alternatives is very large and using deactivable activities is intractable here. Moreover, the resource-centric model is more appropriate to represent problems with complicated resource constraints including set-ups, cleaning activities etc. as we showed in [3] where we describe the criteria for choosing the right grouping for a particular problem using the classification of constraints.

The slot chain corresponding to the group of activities can be generated before we start scheduling. In such a case, we need to estimate its length or more precisely we need to compute the upper bound for the chain length. In case of the task-centric model, we introduce a single chain per task and the length of the chain is equal to the maximum number of activities in all alternative production plans for the task. In the resource-centric model we use single slot chain per resource and the length of the chain is computed by dividing the schedule duration by the duration of the shortest activity that can be processed in this resource[7]. Because upper estimate for the number of slots is used it may happen that some slots remain empty after scheduling. To

[7] It is possible to use better upper estimate that takes in account the admissible sequences of activities. Still, this estimate is not precise, especially if the difference between the shortest and the longest activity is large.

prevent empty slots we can fill them by a dummy action (this is not required but it simplifies handling slots - after scheduling, every slot is filled by an activity).

There exists a more dynamic approach to slot introduction that is useful if the upper estimate for the number of slots is large[8]. We can use lower bound for the number of activities to introduce a minimal number of slots corresponding to the minimal number of activities to be scheduled. If we find later during scheduling that more activities are necessary then we can generate more slots dynamically. This is the approach that we currently use in the VisOpt system for scheduling complex-process environments [14]. Note that no dummy activities are necessary there because we generate slots on demand. Each time a slot is introduced we know there must be an activity scheduled in the slot.

Slots vs. deactivable activities. While the traditional approach with deactivable activities [4,11] is eager in generating activities, our approach with slots is more modest. When deactivable activities are used to represent alternatives or process-dependent activities, all the activities must be generated and some (most) of them will be deactivated during scheduling. In case of process plan from Figure 2, we need to introduce ten activities even if we know that maximally six activities will be included in the final schedule. However, if the slot model is used to represent the same problem we can introduce six slots only (or five in case of dynamically introduced slots). The advantage of the slot model becomes even more evident when the number of deactivable activities is very large.

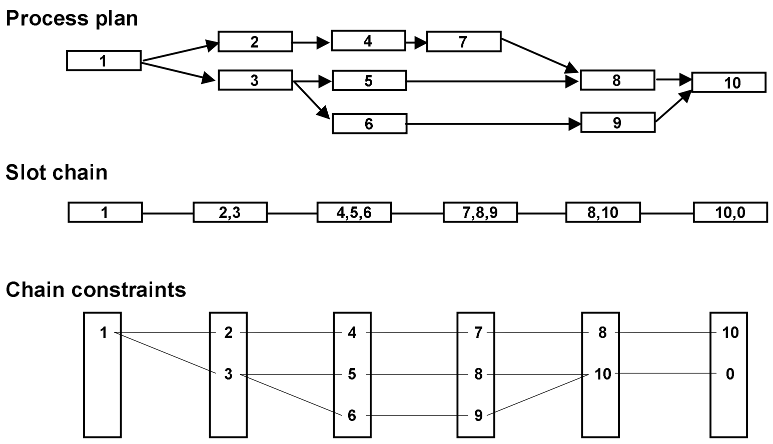

Fig. 4. The process plan with ten activities can be represented using six connected slots (the numbers in the slots indicate which activities can be filled in given slot, 0 is a dummy activity). There is a constraint between each pair of consequent slots that binds the activity attributes (e.g., if we choose activity 3 for the second slot, we can deduce - using constraint propagation - that the fifth slot contains activity 10 and the sixth slot is filled by a dummy activity 0).

[8] This is usually the case of resource-centric models, in the task-centric model the number of slots corresponds to the number of activities used.

4.2 Slot Parameters

One of the main reasons why we propose to work with slots instead of with activities directly is that we can move some activity parameters to the slot. This is very important because in the dynamic problem formulation we do not know the activities in advance so we cannot post constraints among the activity parameters until the activity is introduced. However, when (some of) these parameters are moved to the slot then we can post the constraints as soon as the slot is introduced and this is earlier than the activity is known. A typical example of such *static slot parameters* is start time and duration of the activity. More generally, all the parameters that are common for the activities, which can be filled in the given slot, can be moved to the slot.

As we already mentioned, there is also a special *Activity variable* in every slot whose domain describes which activities can be filled in the slot. By using this variable we naturally include planning task within scheduling and no special planning mechanism is required. The Activity variable participates in constraint propagation like other variables, making the domain for the Activity variable singleton corresponds to introducing the activity into the schedule.

Finally, there is a group of *dynamic slot parameters* that are specific for particular activity. For example, different activities can produce different items so the variables describing item quantities are specific for the activity[9]. These variables cannot be introduced before we know the activity in the slot thus we call them dynamic slot parameters. As soon as the Activity variable becomes ground (this condition may serve as a trigger), we can generate dynamic slot parameters.

It should be noted that we could represent the dynamic slot parameters statically too, i.e., all the parameters are introduced in the slot and if the activity in the slot does not use them then these parameters are ignored. The discussion here is similar to using deactivable activities - if the set of dynamic parameters is large, e.g. there are thousand of items processed by several activities in different combinations, then their static representation is intractable due to big memory consumption.

4.3 Constraints

When the variables in slots are defined we may introduce the constraints among them. We propose to classify the constraints in three groups according to the function of a particular constraint in the slot model. This classification helps us to map particular problem to the slot model and it simplifies understanding the differences between posting the constraints.

Slot constraints (intra-slot constraints). First, there are constraints binding variables in a single slot. Usually, these constraints take care of filling the slot by an activity so they bind the Activity variable with remaining variables in the slot. Typically,

[9] Because single activity can produce given item in different quantities we use a variable to capture item quantity. By using item quantity variables with constraints among them we can naturally model capacity of the resource (e.g. 10 tons of item 1 and 2 can be processed simultaneously and this quantity can be arbitrary distributed among the items).

different activities have different duration and different time windows when they can be processed. This can be expressed by two binary constraints or by a single ternary constraint in the form of table. Note that the propagation through the constraint is non-directional so the same constraint can be used to decide about the activity in the slot as well as about the duration of the slot. For example, if we know that the start time of the slot is greater than 30 then we can deduce that only action 4 can be filled in the slot which further sets the duration to be 1 (see table below).

Activity	1	2	3	4
Start time	10..20	15..30	10..30	25..35
Duration	5	10	3..5	1

Naturally, the slot constraint may also describe relations that do not involve the Activity variable like the following example shows:

```
Start + Duration = End
```

Chain constraints (inter-slot constraints or intra-chain constraints). The second group of constraints corresponds to relations between variables from different slots of a single slot chain. Typically, these constraints bind two successive slots in the slot chain and they describe the precedence relations:

```
∀i slot(i).Start + slot(i).Duration ≤ slot(i+1).Start
```

or the transition patterns/process plans (Figure 4 gives a particular example of such constraint describing alternative process plans):

```
slot(1).Activity = 1 ⇒ slot(2).Activity in {2,3}
```

Again, the propagation is bi-directional so information from given slot is propagated to both earlier and later slots. This is a nice feature because we can fill the slots in arbitrary order and the propagation guarantees that we can still fill the remaining slots. Finally note that we do not restrict the chain constraints to bind successive slots in the chain only but arbitrary slots may be connected. This allows us to define more complicated chain structures (not only the linear structure).

Inter-chain constraints. The last group contains the constraints binding slots from different chains. These constraints express the dependencies between different slot chains, i.e. between different tasks if the slot chain represents a task, or between different resources if the slot chain corresponds to a resource. For example, in the resource-centric model we can express the following synchronisation constraint:

```
∀k (chain(i).slot(k).Activity = 1 ⇒
      ∃l (chain(j).slot(l).Activity = 2 &
      chain(i).slot(k).Start = chain(j).slot(l).Start))
```

which says: if the resource *i* processes the activity 1 then the resource *j* must process the activity 2 starting at the same time[10].

The existential quantifier in the constraint makes difficulty when posting such constraint because we must find which slots are connected. This is more visible if we rewrite the constraint using disjunction:

```
∀k (chain(i).slot(k).Activity = 1 ⇒
      ∨l=1,..,n (chain(j).slot(l).Activity = 2 &
      chain(i).slot(k).Start = chain(j).slot(l).Start))
```

where *n* is a number of slots in the resource *j*. We discuss methods for posting inter-chain constraints in the next section.

In [3] we proposed a general grouping of constraints for scheduling problems, in particular we distinguished between:

- *resource constraints* describing limitations of single resource in given time, e.g. restricted capacity,
- *transition constraints* describing relations between states of single resource in different time points, e.g. transition patterns,
- *dependency constraints* describing relations between different resources, e.g. supplier-consumer relation.

We can now define the mapping between the classification in resource-centric and task-centric representation and the constraint groups in the slot models.

Table 1. Mapping between the slot model and resource-centric and task-centric representations

Slot models	**Resource-centric representation**	**Task-centric representation**
Slot constraints	Resource constraints	Resource constraints
Chain constraints	Transition constraints	Dependencies
Inter-chain constraints	Dependencies	Transition constraints

5 Scheduling with the slot representation

Having a declarative model of the real-life problem is the first step to solve the problem; we also need an efficient solver. The underlying solving mechanism is based on constraint satisfaction, in particular we are using constraint propagation for pruning the domains of the variables and labelling to assign values to the variables. In the rest of the paper we concentrate on a new aspect of the proposed representation, i.e. the dynamic character - new variables and constraints are introduced in course of solving.

[10] Constraints of this type express synchronisation between activities, e.g. cleaning activities must be processed in parallel in two resources or an activity in one resource must produce an item at given time that is consumed by another activity in another resource.

Generating dynamic variables: Because the set of dynamic slot variables may differ from activity to activity we decided to introduce them as soon as the Activity variable in the slot becomes ground (its domain becomes singleton). It is natural to use event-based programming to implement introduction of dynamic variables using the same mechanism as constraint propagation is implemented (typically, the propagation is waked-up when the domain of some constrained variable is changed).

Posting the constraints: To exploit constraint propagation, that reduces the domains of variables and prunes the search space, we prefer to post the constraints as soon as possible. Naturally, we can post all the constraints binding the static variables in advance (if we know which variables are bounded). Still, we can identify two main reasons leading to postponing the constraint introduction:

- First, the constraint includes some dynamic variables. Such constraint must be postponed until the dynamic variables are introduced - we can post this constraint immediately after creating the dynamic variables.

- Second, it is not clear which variables are bound by the constraint. This is the case of inter-chain constraints where we do not know which slots should be connected by the constraint. There are two extreme methods for posting such constraints (and many variants in-between): eager and lazy method.

  ✓ *Eager method* corresponds to the static formulation of the disjunctive constraint, i.e., we connect all the slots using the disjunctive constraint as Figure 5 shows. It is a known wisdom that propagation through a disjunctive constraint is weak but on the other side there exist special propagation methods designed for scheduling problems that can be used there [1]. The problem with large disjunctive constraints is memory overhead necessary to maintain consistency which could be intractable it large-scale scheduling problems.

  ✓ *Lazy method* waits until we know which slots should be connected, i.e., until we have enough information about the values of variables. The advantage of the lazy method is that it generates necessary constraints only (it needs no disjunction), unfortunately, the constraints are generated too late (in reality we must wait until most of the variables are ground) and, thus, they do not contribute to pruning the search space. In fact, the constraints posted by the lazy method behave like a test only.

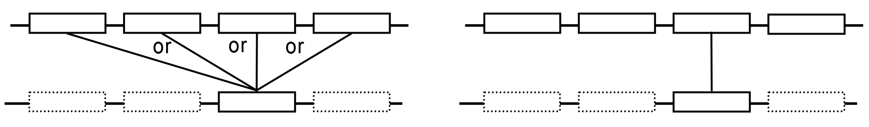

Fig. 5. The eager method (left) posts the large disjunctive constraint, i.e., it connects all potentially dependent slots, while the lazy method (right) waits until we know which slots are dependent and then post the constraint (it selects one constraint from the disjunction).

In practice, the mixture of both eager and lazy method seems useful to reduce memory overhead of the eager method and to improve the propagation of the lazy method. It means that we can generate part of the disjunction and if we find later that there are no dependent slots there then we can generate next part of the disjunction and so on. The Figure 6 shows this method.

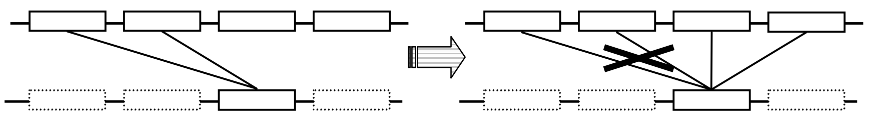

Fig. 6. We can post part of the disjunctive constraint first and if we find that dependent slots are not present there then we post the next part etc. This is useful if left to right scheduling is used (left slots are labelled first).

6 Conclusions and results

In real-life industrial scheduling there exists problems that require adding new activities during scheduling, i.e. enhancing scheduling by planning capabilities. Because the conventional static constraint models are not able to handle such problems we proposed slot models for scheduling problems enhanced by planning capabilities. This enables us to formulate and solve wider range of real-life problems using the constraint programming technology.

The proposed slot representation of the resource-centric model is used in the generic scheduling engine of Visopt system [14]. This system is designed to solve mixed planning and scheduling problems in highly complex process industries and it is the first system that can handle the complexity of such industries including full modeling of set-ups, cleaning, alternative resources etc. The first pilot project, where the system is tested, is a complex cottage production line in a diary. The model consists of 58 resources with more then 600 alternative process lines. A complete schedule/plan for 9 days with the resolution of 10 minutes consists of about 1 500 objects (majority of them is introduced-planned during scheduling) and it takes about three minutes to find it on Celeron 500 with 196 MB memory.

Acknowledgements

Author's work is supported by the Grant Agency of the Czech Republic under the contract number 201/99/D057 and by Visopt Tech Ltd. I would like to thank Yossi Rissin and the team of Visopt Tech for introducing me to the problem and for interesting and encouraging discussions concerning real-life problems of industrial planning and scheduling. I am also grateful to Helmut Simonis from Cosytec for useful discussions concerning constraint scheduling.

References

1. Baptiste, P. and Le Pape, C.: Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems. *Proceedings of the Principles and Practice of Constraint Programming Conference-CP97*, Linz, Austria, LNCS 1330, Springer Verlag (1997)
2. Barták, R.: On the Boundary of Planning and Scheduling: A Study. *Proceedings of the Eighteenth Workshop of the UK Planning and Scheduling Special Interest Group*, Manchester, UK (1999) 28-39
3. Barták, R.: Dynamic Constraint Models for Planning and Scheduling Problems. *New Trends in Constraints*, LNAI 1865, Springer Verlag (2000)
4. Beck, J.Ch. and Fox, M.S.: Scheduling Alternative Activities. *Proceedings of AAAI'99*, USA (1999) 680-687
5. Brusoni, V., Console, L., Lamma. E., Mello, P., Milano, M., Terenziani, P.: Resource-based vs. Task-based Approaches for Scheduling Problems. *Proceedings of the 9th ISMIS96*, LNCS Series, Springer Verlag (1996)
6. Caseau, Y., Laburthe, F.: A Constraint based approach to the RCPSP. *Proceedings of the CP97 Workshop on Industrial Constraint-Directed Scheduling*, Schloss Hagenberg, Austria (1997)
7. Joslin, D. and Pollack M.E.: Passive and Active Decision Postponement in Plan Generation. *Proceedings of the Third European Conference on Planning* (1995)
8. Koehler, J.: Planning under Resource Constraints. *Proceedings of 13th European Conference on Artificial Intelligence*, Brighton, UK (1998) 489-493
9. Nareyek, A.: Structural Constraint Satisfaction. *Proceedings of AAAI-99 Workshop on Configuration* (1999)
10. Nareyek, A.: AI Planning in a Constraint Programming Framework. *Proceedings of the Third International Workshop on Communication-Based Systems* (2000), to appear
11. Pegman, M.: Short Term Liquid Metal Scheduling. *Proceedings of PAPPACT98 Conference*, London (1998) 91-99
12. Verfaillie, G. and Schiex, T.: Solution Reuse in Dynamic Constraint Satisfaction Problems. *Proceedings of the 12th National Conference on Artificial Intelligence AAAI-94*, USA (1994), 307-312
13. Wallace, M.: Applying Constraints for Scheduling. *Constraint Programming*, Mayoh B. and Penjaak J. (Eds.), NATO ASI Series, Springer Verlag (1994)
14. VisOpt web site, http://www.visopt.com