Visopt ShopFloor: On the edge of planning and scheduling

Roman Barták*

Charles University in Prague, Faculty of Mathematics and Physics
Malostranské námestí 2/25
118 00 Praha 1, Czech Republic
bartak@kti.mff.cuni.cz

Abstract. Visopt ShopFloor is a complete system for solving real-life scheduling problems in complex industries. In particular, the system is intended to problem areas where traditional scheduling methods failed. In the paper we describe the heart of the Visopt system, a generic scheduling engine. This engine goes beyond traditional scheduling by offering some planning capabilities. We achieved this integrated behaviour by applying Constraint Logic Programming in a less standard way - the definition of a constraint model is dynamic and introduction of constraints interleaves with search.

1 Introduction

Scheduling is one of the strongest application areas of constraint programming [16]. The reason of such success can be found in a similar static character of both scheduling problems and constraint satisfaction problems. First, the problem structure is fully described in advance: in case of scheduling it means that all the activities and relations among them are known in advance, in case of standard constraint satisfaction problem (CSP): all the variables, their domains, and constraints are known in advance. The problem solving then consists merely of searching the space of alternative combinations of activity locations or variables' values respectively. Search is also a core technology in the close area of planning. However, the main difference from scheduling is the dynamic character of planning [13]. The structure of the plan is so variable that it can be hardly encoded in variables, domains, and constraints defined before the problem solving starts. Thus, planning typically uses ad-hoc algorithms even if there exist approaches based on general concepts like CSP. These approaches either try to fit a planning problem into a static concept of CSP [6,14] or they are based on generalisations of the CSP framework like Dynamic CSP [11] or Structured CSP [12].

To solve problems on the edge of planning and scheduling we propose to use an existing technology of Constraint Logic Programming (CLP) in the way this framework was originally defined [8]. Note that since CSP has been formalised, the

* The research is supported by the Grant Agency of the Czech Republic under the contract no. 201/01/0942.

same static approach is applied to solve problems in CLP as well, i.e., define the variables, domains, and constraints first and then do search [15]. In our opinion this is not a natural way of solving problems in (constraint) logic programming framework. Moreover, it leads to such strange formulations like calling ILOG Solver "a C++ implementation of constraint logic programming" (C++ is no more logic programming). Our basic idea is to return to the roots, i.e., to use constraints in CLP in the same way as unification is used [8]. It means that constraints are introduced during search and thus different sets of constraints are active in different branches of the search tree. The usual argument against this approach is weak constraint propagation. However, this is not a fully fair argument because different branches in the search tree typically represent disjunction and it is a known wisdom that propagation through disjunction is not good as well. Moreover, we can post the constraints in advance in CLP too, if we know them. Our proposal is to use CLP capabilities to reduce exponential grow of the number of constraints in the planning-like problems.

We believe that CLP approach is sufficient to model and solve mixed planning and scheduling problems. To demonstrate this attitude we have implemented a generic scheduling engine for the Visopt ShopFloor system. The main difference of our solver from other scheduling systems is a full integration of the planning component, i.e., the solver does both planning and scheduling tasks. In fact, there is no strict border between planning and scheduling in the Visopt system

The paper summarises our work on the scheduling engine for Visopt ShopFloor system. We first introduce the problem area and discuss its description (Section 2). Then we present some details of the internal architecture of the scheduler, in particular we describe a dynamic constraint model and a basic scheduling strategy (Section 3). We conclude with some results and lessons learned (Section 4).

2 Problem area and its description

Visopt ShopFloor is not another academic planner/scheduler but its design has been driven completely by a commercial area, i.e. by existing problems in real-live factories. The emphasis has been put to complex areas where the traditional scheduling techniques failed because they were not able to cover whole complexity of the problem. In particular, we concentrate on scheduling problems in food, chemical, and pharmaceutical industries. Nevertheless, the original goal was to design a generic scheduling engine applicable to other areas as well. So, from the beginning we accommodate the engine by rich modelling interface for problem description.

2.1 Order-driven production

Basically, the goal is to generate a plan/schedule of production for a given time period. It means that notions like makespan are not used there directly because the scheduled period is fixed.

The production is driven by *orders*, i.e., the user specifies a set of orders to be scheduled. Each order is described by a set of ordered items together with requested

quantities. It is also possible to describe alternative items that can be delivered if the ordered item is not available (provided that the total ordered quantity is fulfilled). Moreover, the ordered quantity can be relaxed as well so we can deliver more or less than the user requested (if the customer allows it). Thus the ordered quantity is not necessarily a constant number, it could be an interval with one number inside the interval indicating the ordered quantity.

Finally, for each order the user describes the delivery time and the latest delivery time. Both times can be equal, then the ordered items must be delivered at given time. The other extreme is that there is no restriction about the latest delivery time. Then we can postpone production for such order if there is not enough capacity to produce for this order. The latest delivery time is typically used to distinguished between the real orders from the customers (the latest delivery time is restricted) and the forecast orders (no restriction about the latest delivery time).

Using intervals instead of constant numbers for item quantities and delivery times is a way of introducing soft constraints into the model. Naturally, the users prefer that given quantities are delivered at given times. We use a penalty mechanism to describe how the user requirements are satisfied in the schedule. The penalty can be put to quantity, to alternative items, and to delivery time. Basically, the penalty is increasing (linearly) if we deliver quantity different from the ordered quantity. For the alternative items, the ordered quantity is zero so if we deliver alternative items instead of the ordered items then we pay the penalty. Thus no special mechanism to distinguish between the ordered items and the alternative items is necessary. For the delivery time, we pay the penalty if we exceed the acceptable delivery time. Then the penalty is increasing linearly (Figure 1). We spoke about optimisation issues later in the paper.

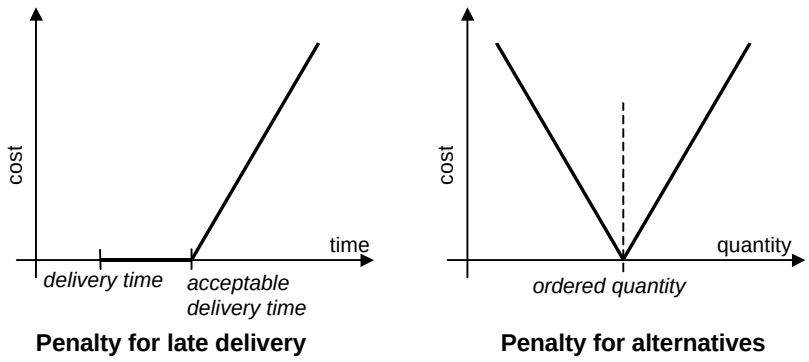

Fig. 1. Penalty mechanism for orders

Note that we do not allow delivering before the delivery time. We use a different mechanism to describe earliness of the deliveries. In reality, if the item is ready (produced) before the delivery time then it must be stored somewhere. So instead of the penalty for earliness we use the storing cost which makes the model closer to reality.

2.2 Complex resources

A typical feature of our problem area is using *resources (machines) with complex behaviour*. This behaviour is described using *states* and *transitions* between the states. At each time, the resource can be at one state only or the resource is in transition between two states (we allow transition time to be assigned to each transition). The transition scheme is typically described by a directed graph (Figure 2) or by a transition table. Note that using this general concept we can model set-ups, changeovers etc. either using transition times or using states. The set-up states are useful, if they are connected to other resources. For example, if some by-product is produced during the set-up or if a worker (another resources) is required to do the set-up. Still, at the modelling level set-up states are handled like all other states. The number of states is not limited in resources, some resources have just one state, in other resources the number of states can be rather large (tens to hundreds).

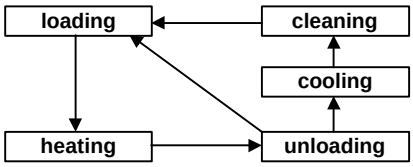

Fig. 2. A simple transition scheme for the resource.

Currently, we concentrate on batch production primarily so the schedule of the resource is described by a sequence of non-overlapping batches[1]. Each batch belongs to one of the resource states. The user may also restrict the length of the batch sequence in given state. For example a minimal number of five batches and maximal number of ten batches of some state S can be in sequence. It means that we cannot change the state S of the resource until at least five batches of this state are processed and we have to change the state S to another state (following the transition scheme) if ten batches of the state S have been processed. In addition to this min batch/max batch constraint and the transition scheme, the user may specify other sequencing constraint that we call a *counter*. The counter says that after a given number of batches there must a batch of given state, e.g. after ten processing batches there must a cleaning batch. The counter can be also state specific, i.e., only batches of given states are counted and batches of other states are ignored (Figure 3).

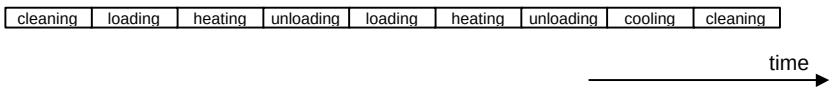

Fig. 3. Batch counters. After two "heating" batches a "cleaning" batch must be inserted (we use a transition scheme from Figure 2; "loading", "unloading", and "cooling" are not counted).

[1] Continuous process can be decomposed into a sequence of batches.

For each state the user specifies (process) *duration* of batches of this state. Duration can be either constant or variable. If the duration is variable then the solver may choose appropriate duration for each batch from a specified domain. The location of batch in time can be further restricted by using *time windows*, i.e., the batch must start and complete in specified time intervals. Again, the time windows are common for all the batches of given state, i.e., the user specifies the time windows for states rather than for particular batches. Batches of some states are not interruptible, i.e., they must run completely within some time window. We also allow interruptible batches, i.e., the batch may start in one time window and complete in another time window. Then we distinguish between processing duration, i.e., time spent in time windows, and idle duration, i.e., time spent out of time windows. The total duration of the batch equals to the sum of process duration and idle duration. Note also that the batch stays in the resource from the beginning to the end of processing so even if the batch is in idle time no other batch can be processed by the same resource (Figure 4).

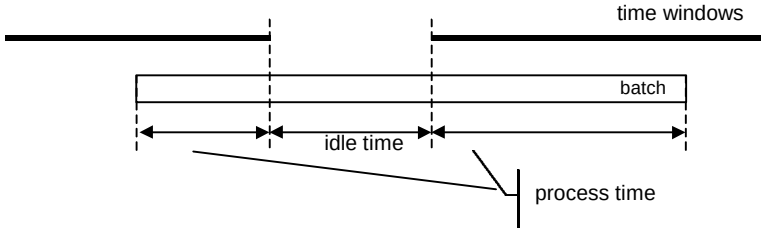

Fig. 4. Interruptible batch occupies the resource out of time windows too.

To make the concept of interruptible/non-interruptible batches more general, we allow the user to express the maximal ratio between the idle duration and the process duration. If this maximal ratio is zero, then the batch is non-interruptible. If maximal ratio is one than the batch is interruptible but the idle duration cannot be longer than the process duration etc. Thus, the user may describe a full scale of interruptibility.

Note finally, that the concept of interruptibility is different from pre-emptiness used in traditional scheduling. A pre-emptive activity may be stopped in the middle, another activity (activities) is processed and then the original activity is re-started (perhaps on different resource). In our concept, the pre-emptive activities can simply be modelled by a set of batches (see Figure 7 for example).

As we already mentioned, batches of the same state may have variable duration. We also allow having variable *capacity* of the batch: for example a single heating batch may be used to heat two to five tons of the item. Typically, the user describes some minimal capacity processed by the batch (e.g., two tons), maximal capacity processed by the batch (e.g., five tons), and the increment in capacity (e.g., one ton so two, three, four, or five tons of the item can be processed in the batch). By using batches with variable capacity we can model parallel processing as well (Figure 5), e.g., stores, but this way of modelling is not very efficient (concerning solving) as it is very similar to a timetabling approach [2].

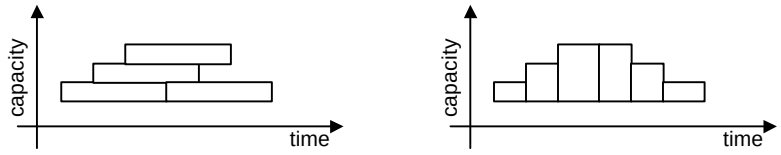

Fig. 5. Modelling parallel processing (left) using batches with variable capacity (right).

2.3 Resource dependencies

Typically, there are more resources involved in the problem and these resources interact in a predefined way. We describe this interaction by *supplier-consumer dependencies.* It means that each batch has some input items that are consumed and some output items that are produced (in some batches, there are only the input items or only the output items). There must be a supplier for the input items, i.e., there must exist some batches that produce the item, and there must be a consumer for the output items. For each item that appears in the model, the user describes a connection between the supplying resource and the consuming resource. Because this connection describes moving of the item between two resources, it is possible to specify the moved quantum as well as the delay between the end of the supplying batch and the start of the consuming batch (Figure 6).

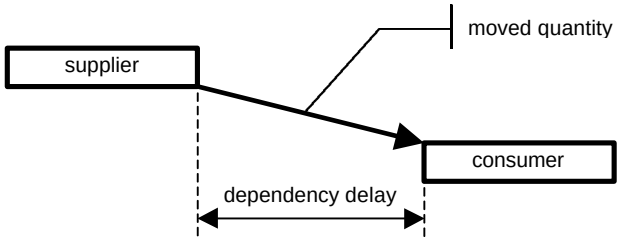

Fig. 6. Batches are connected via dependencies.

We allow the supplier-consumer dependency to be very flexible. It is possible to specify several supplying resources per item and per consumer and vice versa. Also, the dependency delay might be variable so one consuming batch can be connected to several supplying batches of a single resource (Figure 7). The number of connections is driven by the quantity of the item. For example, if the batch consumes ten tons of some item then we have to find enough supplying batches for this quantity. This concept of dependencies is very general and the users may describe an arbitrary structure of the factory including many-to-many relations between the resources or even (re-)cycling. This is usually very hard or even impossible in the existing scheduling systems where typically only the precedence relations can be defined.

Supplier-consumer dependencies may also mimic behaviour of some resources like movers and stores. If there are no other constraints on the mover (like limited capacity) then the mover can be fully modelled using the dependencies. Otherwise a

mover should be described as a resource. Also, it is possible to model some stores like buffers using dependencies with variable dependency delay. The variability of the delay describes the minimal and maximal storing time for the item.

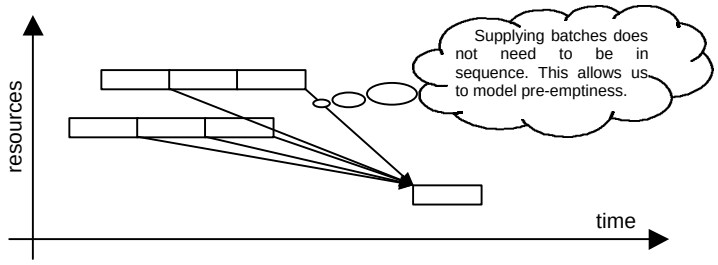

Fig. 7. One consuming batch can be connected to several supplying batches spread over more resources (and vice versa). The only restriction is that the requested quantity must be supplied and the time distance between the supplying batches and the consuming batch satisfies the dependency delay.

We can use orders in the definition of supplier-consumer dependencies as the consuming resource. Using this technique, the user describes which items can be ordered and what resources supply these items. In a similar way, the user can define purchase as the supplier for some items. It means that some items may be purchased from external suppliers. Thanks to generality of the dependency mechanism there is no problem to define items which can be both produced on site or which can be purchased. It is up to the solver to decide how the item is acquired.

2.4 Objectives

As we already mentioned, the basic task is to generate a feasible schedule for the fixed period of time. Some users prefer to minimise the number of set-ups, others like to minimise storing time or to maximise satisfaction of customers (to minimise the sum of penalties in orders). Quite often *multi-criteria optimisation* is required.

Our opinion about optimisation in scheduling is that all the optimisation criteria substitute what the user really needs - to maximise the profit. If the set-ups are expensive then the user asks to minimise the number of set-ups. But what he or she really wants is to minimise the cost of production. The same holds about minimisation of makespan. If the production is faster then the user can satisfy more customers and thus to earn more money. So it seems to us that most (all?) optimisation criteria used in traditional scheduling are in fact instances of minimisation of cost that is equivalent to maximisation of profit. These criteria differ in the way where the cost/penalty is put. Consequently, we believe that a unified cost model can cover most real-life optimisation criteria. Moreover, because there is a single optimisation parameter - the cost - there are no problems with multi-criteria optimisation. It means that the optimisation criterion is shifted from the solving level to the modelling level (to the definition of costs/penalties).

In addition to penalties discussed in previous sections, the user may put cost to batches (dependent on duration and processed quantity), to transitions, and to dependencies. The optimisation task is then to minimise the total sum of costs and penalties in the schedule.

Note finally, that a typical user does not require finding an optimal solution - a good enough solution is accepted as well. Cost optimisation is used to get good schedules but it is the user who decides if the solution is "good enough". Typically, the quality of the schedule is measured by savings achieved when the schedule is applied. In many cases, a small improvement of the schedule over the existing schedule is assumed to be a "good enough" solution. To summarise discussion about the optimisation: the main point is that we are not required to prove optimality or the distance from the optimal solution - we should simply produce a good schedule.

3 Realisation of the solver

Resources, dependencies, and orders are described formally in a Prolog-like modelling language. In fact, we use a set of Prolog facts to describe fully the problem and we call this set a *factory model.* The factory model makes an interface between the Visopt user interface and the solver. Using the factory model is the first major difference of our solver from existing schedulers. Note that the factory model describes declaratively the plant and the demands. Basically, it is a list of attributes of resources, dependencies, and orders. There are no constraints defined in the factory model and there are no activities to be scheduled. The constraints are hidden in the semantics of the factory model and we build automatically a constraint model from the factory model. Concerning activities, in our terminology we speak about batches, they are introduced dynamically during problem solving according to demands. Thus we solve a planning problem (introduction of activities) mixed with a scheduling problem (allocation of activities). Still, we can solve problems in the size close to pure scheduling and order of magnitude larger than pure planning problems (see the last section). And we have the flexibility of integrating both concepts so the user just describes the problem using a factory model and a generic solver finds a schedule. At least, so far we were able to solve all the problems coming from our pilot projects using a single generic engine tuned by few parameters.

3.1 Constraint Representation

As we mentioned above, the factory model describes just the attributes and the constraints are hidden in the semantics of the model. In this section we describe how the model is realised in terms of constraint satisfaction, i.e. using domain variables and constraints.

There exist approaches trying to represent dynamic problems in a static way using dummy variables [6,14]. The difficulty of our problem area is that the ratio between the total number of variables (including dummies) and the number of variables participating in the solution is very large. Thus a fully static representation is inefficient. In fact, the static realisation cannot be realised for large-scale problems

due to huge memory consumption. Thus we decided to use a representation where some variables and constraints are introduced dynamically during search [3]. We still use some dummy variables to help the decision process via constraint propagation, but the number of dummy variables is limited. Basically, if there is a planning branching, i.e., a decision about which activities should be introduced, then we introduce all of them and via constraint propagation we can eliminate some of such activities. In some sense, this is a realisation of the idea of active decision postponement [9]. If we have to decide which activity is used, we introduce all the candidate activities and we postpone the decision until we get more information about the candidate activities.

Slots. Basically, the Visopt solver uses a resource-centric model [2,7], i.e., the activities are grouped per resources rather than per tasks. The reason for choosing this model is large complexity of the resource constraints in comparison with the dependency constraints. The resource centric model is realised via *slots.* Slot is a shell filled by a batch during scheduling. For each resource we have a chain of slots and during scheduling these slots are being filled by batches. The difference from slots in timetabling is time location of slots. In timetabling, the slots represent fixed time intervals. In the Visopt solver, the slots may slide in time. Still, the order of slots is fixed but the slots may be shifted in time, e.g., if the slot is moved to later time then all the successive slots must be moved as well (Figure 8).

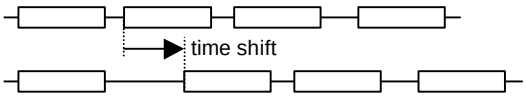

Fig. 8. Slots can move in time provided that the ordering of slots is preserved

Because slots are not fixed in time, there are two finite domain (FD) variables in each slot representing the *start time* and the *end time* of the slot. As described in section 2.2, we distinguish between the *process and idle duration* so there are two more FD variables representing these durations. Remind that decomposition of duration into process and idle parts depends on the time windows (Figure 4), i.e., on the state of the resource. So the next FD variable describes what is the *state* of the resource in the slot. Together, the semantic of the constraint that connects all these variables can be described in the following way:

```
start_time in TimeWindows(state)
end_time in TimeWindows(state)
process_duration = ProcessDuration(state,start_time,end_time)
idle_duration = IdleDuration(state,start_time,end_time)
start_time + process_duration + idle_duration = end_time
```

The first two constraints above are realised using tabular constraints [4] and the last three constraints are realised using a dedicated global constraint with a special filtering algorithm working with time windows.

Transitions. Note that the ordering of slots is fixed so the next slot must start after the end of the previous slot. The exact distance between the slots can be derived from the transition scheme defined for the resource. Basically, this distance depends on the states filled in these two slots so the following formula describes the transition time constraint:

$$end\_time_i + TransitionTime(state_i, state_{i+1}) = start\_time_{i+1}$$

where the index indicates the ordering number of the slot.

To complete the description of the transition scheme, we need to specify the relation between the states in two consecutive slots. Remind that for each state the user describes a minimal and maximal number of batches. Thus, for each slot we should know how many slots right before have the same state. Thus we introduce a new FD variable called a *serial number* that indicates a relative position of the slot in the longest continuous sequence of the slots with the same state (Figure 9).

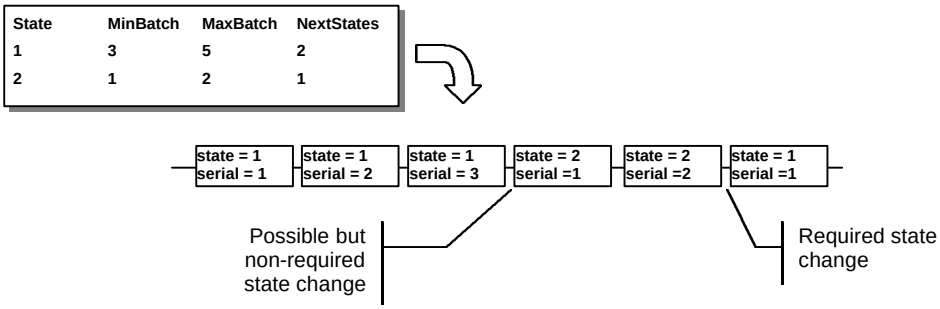

Fig. 9. Serial numbers in slots indicate the position of the batch (slot) in the sequence of slots of the same state.

The semantics of the transition constraint can then be described using the following formulas:

$$\forall i \; serial\_number_i \text{ in } 1..MaxBatches(state_i)$$
$$state_{i+1} \text{ in } \{state_i\} \cup NextStates(state_i)$$
$$state_i = state_{i+1} \Rightarrow serial\_number_{i+1} = serial\_number_i + 1$$
$$state_i \neq state_{i+1} \Rightarrow serial\_number_{i+1} = 1$$
$$serial\_number_i < MinBatches(state_i) \Rightarrow state_i = state_{i+1}$$

It is possible to implement the transition constraints exactly as specified above but to get better pruning we use a special global constraint describing the transition. Note also, that the transition constraint can be seen as a special version of the counters (section 2.2) so counters are implemented in a very similar way.

Introducing the slots. There are two ways how to introduce slots to the system. First, it is possible to estimate a maximal number of slots using the transition scheme, duration of batches etc. and to generate all necessary slots in advance. The disadvantage of this approach is that it may introduce a huge number of dummy slots that will not be used in the final schedule. The only advantage could be better propagation because the slot variables are known and the constraints among them are

posted before we start labelling. However, we do not need propagation in far future slots that will not be used in the final schedule (moreover, the propagation is weak there and it slows down the system). Therefore we generate the slots dynamically on demand. It means that if we find that some batch could be allocated to the resource then we generate a slot for it. So slots are added due to a transition scheme (restricted transition time) or due to a demand from other resources (asking for supplier/consumer). Note also that even if we introduce the slot it does not mean that it will be filled by the batch that caused this introduction. Perhaps some other batch overhauls it or the slot stays empty. Still, the ordering of slots is fixed so it is not possible to introduce a new slot in-between two existing slots (because the transition constraints has already been posted). Thus, deciding to which slot the batch is allocated corresponds to the decision about absolute ordering of batches in the resource.

Filling the slots. There is another difference between slots and batches. The batch describes also input and output quantities of processed items so for each item there is a FD variable describing its quantity. If the number of items is large, it is not efficient to include such variables into the slot until the batch (state) in the slot is known. Thus, these variables (and corresponding constraints, e.g., the capacity limit) will be introduced dynamically when the state in the slot is known, i.e., when the domain of the state variable becomes singleton. In our constraint model, such introduction is done automatically using event-driven programming.

Dependencies. Dependencies form the most dynamic part of the model. Remind that the dependency describes a supplier-consumer relation between two batches so the dependency will connect two slots filled by respective batches. Because the dependency is closely related to the item we cannot introduce the dependency until we know the item and its quantity. As described in the previous paragraph, the variable specifying the item quantity is introduced as soon as we know the batch - the state - in the slot. At the same time we can start dependencies from the given slot.

Assume that we have an input item defined for the batch in the slot. Dependencies should connect this batch with all the supplying batches. It is possible to post dependency to every slot that can be filled by a supplying batch. However, this eager method has huge memory consumption when applied to large-scale problems with hundreds or thousands of slots. Thus, we use a more lazy method that posts a minimal number of dependencies covering the required quantity. Typically, these dependencies go to the first possible slot of the supplying resources. If we find later that these slots cannot be filled by a supplying batch then we move the dependency to the next slot and so on. If there is no slot found in the resource, the dependency is made empty. Other dependencies can be introduced as soon as we find that the dependencies generated so far are not enough (e.g. because some of them have been made empty). For each slot, the system maintains links to all non-empty dependencies going to this slot. These links are then used during scheduling when deciding what batch will be filled in the slot (see section 3.2).

Dependency constraints. Dependency connects the supplying batch with the consuming batch and it posts a constraint between the end time of the supplying batch and the start time of the consuming batch (Figure 6). Moreover, there is some quantity of the item going though the dependency so the sum of all such quantities per item must be equal to the processed quantity in the batch. These constraints define soundness of the dependency but we can post more constraints that simplify scheduling.

First, it should be said that the dependency could be started both from the supplying batch and from the consuming batch. To remove this symmetry we can use a first-come-first-serve constraint so only a single dependency will connect two batches (the dependency in the reverse direction will be made empty).

Note also that dependencies make demands for batches in the resource. Thus, if we decide about ordering of dependencies going to a particular resource then, in fact, we determine the ordering of batches in the resource. There exist filtering algorithms doing such decisions using information about time, like edge-finding [1]. However, the global constraints representing these constraints are usually static in the sense that they can be defined only over a known set of demands. Because dependencies are introduced dynamically we need a dynamic version of such global constraints [5]. In the Visopt solver we use a simple dynamic version of the edge-finding algorithm. Unfortunately, the edge-finding like methods are less effective there because domains of time variables are not very restricted. Methods based on ordering of batches, e.g. [10], are more appropriate there. We use a global constraint that orders the dependencies going to the resource using information about the ordering of batches in the resource where from the dependencies have been started. A detail description of this constraint is out of scope of this paper; Figure 10 shows the basic idea.

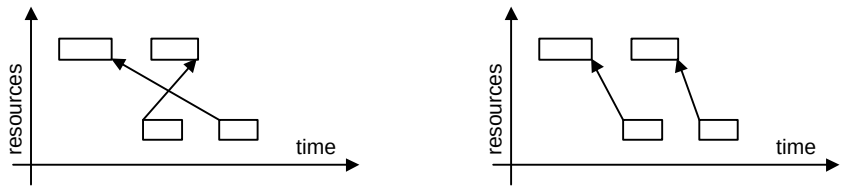

Fig. 10. A global constraint for ordering the dependencies may forbid the order of dependencies on the left and may force the ordering on the right depending on the dependency delay and the distance of consuming batches (the arrow shows a direction from the batch that posted the constraint).

3.2 Labelling strategy

The constraint model is dynamic but autonomous. It means that variables and constraints are introduced automatically when the system finds out that they are necessary. Thus, if the labelling procedure follows some rules about the ordering of variables, then it knows nothing about the dynamic character of the constraint model because the variables are already present there for labelling. For example, the state variable in the slot should be labelled before we can label the variables describing

item quantities in the batch (because as soon as the state variable becomes ground, the variables for item quantities are introduced automatically to the model).

We use standard backtracking-based search driven by heuristic. At the beginning, there are just orders in the system, dependencies going from the orders to some resources, and some (usually empty) slots in the resources. The scheduling strategy works in steps going from left to right (past to future), the size of the step is defined by the user. The step is represented by a border line called *frontier* that is moved from left to right. At each step, only the batches (slots) that must start before the frontier are scheduled. The consequence of this strategy is that only required production is scheduled.

The labelling within the step goes in the order-to-purchase direction, i.e., the slots in the resources that supply directly to the orders are being closed first, then the slots in the resources supplying these suppliers of orders and so on. In the resource the slots are being filled from left to right. For each slot, we first explore the dependencies going to the slot. Some of these dependencies are selected and thus connected to this slot (earlier dependencies are tried first). This decision further constraints the state in the slot so usually, the state variable becomes singleton. Otherwise, this variable is labelled. As soon as the state is known, the labelling proceeds to the variables describing the item quantities in the batch (these variables have been introduced when the state variable becomes singleton). At the end of each scheduling step, the time variables are labelled in the closed slots (earlier time is preferred).

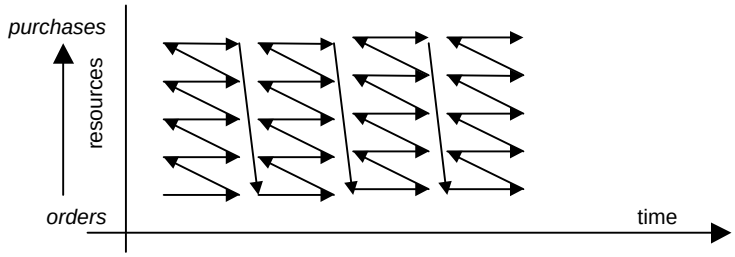

Fig. 11. Ordering used in the labelling strategy.

The key decision in the above labelling procedure is about which dependencies will go to a given slot. In fact this decision is about ordering of batches in the resources. In the standard labelling procedure, the earliest dependencies are preferred. For optimisation, we use a special variant of the labelling procedure where the decisions are done according to the cost. It means that the values are tentatively assigned and the value leading to the smallest cost is selected.

4 Results and conclusions

Visopt ShopFloor system has been tested in several pilot projects in one of the biggest chemical enterprises in Europe, one of the biggest and famous candy producers in The Netherlands, and one of the biggest dairies in Israel among others. The feedback from the companies is very positive - Visopt ShopFloor is the only system, among the

systems that they tested, that can fully cover the complexity of production in these enterprises via offering rich modelling capabilities supported by a new solving technology.

In the following tables we summarise some numerical results of one of the pilot projects where the goal is to generate a detail schedule for a week production.

Number of resources	34
Total number of resource states	991
Size of scheduled period	1 week (10 080 minutes)
Time resolution	1 minute
Number of items	294
Number of orders	45
Total quantity in orders	88.5 tons (88 485 kg)
Quantity resolution	1 kilogram

Table 1. Problem size

The size of the factory model describing fully the problem is almost 1.4 MB. The structure of the problem consists of several groups of alternative resources including secondary resources (workers) and it contains both production and packing of the final items. The resources in the groups of alternative resources are not fully identical so the number of alternative production lines is very large. The item dependent set-up times are included in most production resources and time windows are defined for all the resources. The resolution of scheduling is another interesting point there: we are scheduling a week production with a one minute resolution, i.e., duration of the schedule is over ten thousand time units. Also there are over 88 tons of ordered items and we are scheduling with one kilogram resolution.

Number of batches	5938
Number of dependencies	9496
Runtime	65 minutes (Pentium 4/1700 MHz)

Table 2. Solution size

The complexity of this problem is hidden in the huge number of alternative production routes. So basically, we are solving a planning problem under time and resource constraints. Remind that the input to the engine consists of the plant model and the set of orders. All the activities/batches are introduced (planned) during the problem solving and allocated to resources (scheduling).

Design of the scheduling engine for Visopt brings some novelty problems and shows some solutions both to constraint programming and to scheduling. Despite the original disbelief of the constraint community we showed that the dynamic models are applicable to solving large-scale real-life problems. Our concept of dynamic global constraints [5] could be applied to other areas as well, especially when constraint logic programming is used and when introduction of constraints interleaves with search. For the scheduling area, we showed that resource-centric representation is superior when the number of alternative production routes is huge and when the resource description is complex including transition scheme etc. On the other hand, the memory consumption and weak constraint propagation through production routes are basic weaknesses of this approach so our next steps include integration of

resource-centric and task-centric representations. Finally, our experiments show that the traditional scheduling global constraints like edge finder are less effective when planning is involved so new global constraints are required there.

References

1. Baptiste, P. and Le Pape, C.: Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling, in *Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group* (1996).
2. Barták, R.: Conceptual Models for Combined Planning and Scheduling. *Electronic Notes in Discrete Mathematics*, Volume 4, Elsevier (1999).
3. Barták, R.: Dynamic Constraint Models for Planning and Scheduling Problems. *Proceedings of the ERCIM/CompulogNet Workshop on Constraint Programming*, LNAI Series, Springer Verlag (2000).
4. Barták, R.: Filtering Algorithms for Tabular Constraints, in *Proceedings of CP2001 Workshop CICLOPS*, Paphos, Cyprus (2001), 168-182.
5. Barták, R.: Dynamic Global Constraints in Backtring Based Environments, in *Annals of Operations Research*, Kluwer (2002), to appear.
6. Beck, J.Ch. and Fox, M.S.: Scheduling Alternative Activities. *Proceedings of AAAI-99*, USA (1999), 680-687.
7. 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).
8. Gallaire, H.: Logic Programming: Further Developments, in: IEEE Symposium on Logic Programming, Boston, IEEE (1985).
9. Joslin, D. and Pollack M.E.: Passive and Active Decision Postponement in Plan Generation. *Proceedings of the Third European Conference on Planning* (1995).
10. Laborie P.: Algorithms for Propagating Resource Constraints in AI Planning and Scheduling: Existing Approaches and New Results. In *Proceedings of 6th European Conference on Planning*, Toledo, Spain (2001), 205-216
11. Mittal, S. and Falkenhainer, B.: Dynamic Constraint Satisfaction Problems. *Proceedings of AAAI-90*, USA (1990), 25-32.
12. Nareyek, A.: Structural Constraint Satisfaction. *Proceedings of AAAI-99 Workshop on Configuration* (1999).
13. Nareyek, A.: AI Planning in a Constraint Programming Framework. *Proceedings of the Third International Workshop on Communication-Based Systems* (2000).
14. Pegman, M.: Short Term Liquid Metal Scheduling. *Proceedings of PAPPACT98 Conference*, London (1998), 91-99.
15. van Hentenryck, P.: *Constraint Satisfaction in Logic Programming*, The MIT Press, Cambridge, Mass. (1989).
16. Wallace, M.: Applying Constraints for Scheduling, in: *Constraint Programming*, Mayoh B. and Penjaak J. (eds.), NATO ASI Series, Springer Verlag (1994)