DÉJÀ VU – A reusable framework for the construction of intelligent interactive schedulers

Jürgen Dorn, Mario Girsch, and Nikos Vidakis
Institut für Informationssysteme, Technische Universität Wien
Paniglgasse 16, A-1040 Vienna, Austria
E-mail {dorn|girsch|vidakis}@dbai.tuwien.ac.at

Abstract
We describe the techniques of the DÉJÀ VU Scheduling Class Library to achieve a library of reusable and extendible classes for the construction of interactive scheduling systems. The constructed systems shall be efficient and user centered. We describe abstract scheduling objects, constraints between them, and potential user interactions with the system. A first scheduling system was developed for the steel plant of Böhler Kapfenberg. We demonstrate which extensions were necessary and show prototypical examples from the graphical user interface.

Keywords
Scheduling, Software Reuse, Soft Constraints, Steel manufacturing

1 INTRODUCTION

DÉJÀ VU is a framework of C++ classes supporting the construction of industrial scheduling systems. The design was directed by the following criteria:

- the evaluation of a schedule is based on the evaluation of individual constraints and their weighted aggregation,
- the user has the full control over the scheduling process with the ability to experiment with different settings,
- iterative improvement methods are applied to optimize solutions, and
- the framework should be extendible and refinable.

1.1 Constraint-based Representation of Schedules

Scheduling is an activity controlled by constraints and guided by several objective functions. Usually scheduling is described as a problem of satisfying temporal constraints. However, temporal constraints such as due dates and objectives such as minimization of the mean flow-time are often insufficient to represent industrial problems. DÉJÀ VU supports constraints and objectives like compatibility constraints, idle time constraints, minimization of substitutable resources, or equilibre load of sharable resources. These constraint types have been derived from several scheduling problems. New constraint types can be generated by deriving from existing types with minimal effort due to the general approach to represent them.

Many constraints of industrial production environments are soft and can be relaxed. Moreover, objectives may be contradictory and a trade-off between them must be found for a good solution. These requirements are reflected by the constraint model: A constraint is a relation between two or more scheduling objects and/or attributes. The relation is mapped on a satisfaction degree that evaluates how good this constraint is satisfied in the actual schedule. Different constraint types obtain a domain-dependent weight reflecting the constraint's importance for the domain. A schedule is evaluated by a weighted aggregation of all satisfaction degrees. Further, for each constraint type a threshold to decide whether the constraint violation is hard can be specified.

1.2 Interactive Scheduling

An automatic scheduler cannot consider all aspects relevant to the evaluation of a schedule because the environment of industrial scheduling systems is too complex and many quantities cannot be measured. The complexity also comes from the ever changing production environment: new machines are erected and new production techniques and objectives arise regularly. The software must therefore be adaptable, but under the full control of the user to overrule outdated system rules. Although production control and planning software shall support human personnel as far as possible, the responsibility should remain in human hands. Mixed-initiative scheduling is a paradigm that solves this problem best. Whenever the user has the ability to let the system schedule automatically or to perform some scheduling tasks manually. The user can always change the schedule constructed by the system, but the system should show new conflicts effected by this change to the user. Furthermore the user can "freeze" some part of the schedule and let the system improve the remaining part. DÉJÀ VU supports interactive scheduling by *scheduling tasks* for schedule alterations providing a common interface with methods for undoing, redoing, evaluating, etc.

1.3 Iterative Improvement Methods

Iterative improvement is a search method which starts with an initial solution and tries to improve it by "local" modifications. The initial schedule can be constructed randomly, by a constructive method, or by an heuristic method. It can also be created by a human or another computer process. To modify given schedules, scheduling tasks are used to transform a schedule into a new and similar schedule. A scheduling task can be e.g., the exchange of two adjacent jobs. If several tasks are applicable, a procedure must choose the task to be applied. This selection can be made randomly or with some look-ahead, allowing the selection of the best "neighbor". To determine whether an improvement can be achieved by a task, the evaluation of schedules is compared. The most efficient look-ahead is achieved when the new schedule can be evaluated locally.

A simple hill-climbing algorithm would accept only schedules having a better evaluation. Since scheduling problems tend to have many solutions with different qualities that are not direct neighbors, a search method based on local improvements can be trapped in a local optimum. An important feature of all iterative improvement methods is the capability to escape from local optima. However, with this ability the probability of searching in cycles raises and some kind of control to avoid repetitions is needed.

DÉJÀ VU allows the user to select between different improvement methods and to set different parameters of these algorithms individually. Furthermore, if another combination of techniques seem to be appropriate this can be easily realized by derived classes since the optimization algorithms are also designed as classes that can be inherited. Experimental comparisons of these algorithms with data from the VA Stahl Linz LD3 plant are described in (Dorn et al. 1996) and important design issues for iterative improvement methods in (Dorn 1995).

1.4 Reusability of Scheduling Classes

The main principle to support the reusability is the object-oriented design of the software. However, the critical task in designing reusable software (or reusable objects) is always to foresee the potential extensions and problems of new applications. A good practice is to implement existing theoretical frameworks because they are based on abstractions of many practical applications. Especially in scheduling, there is a large amount of theoretical work offering many forms for such a design. Objects like *order*, *job*, *operation*, *resource*, *allocation*, and *schedule* or synonyms exist in almost every theoretical investigation. Unfortunately, this theoretical work does not integrate user interaction with schedule optimization.

The core of DÉJÀ VU are abstract classes realizing the basic scheduling theory. Forms for the representation of constraints are also realized by abstract classes. This abstract core enables an application- and platform-independent definition of

- a schedule evaluation (all constraints stored in a constraint list are evaluated and aggregated),
- scheduling tasks (exchange of operations on a resource, exchange of jobs, ...)
- algorithms that apply and compare applicable scheduling tasks to find better schedules, and
- graphic entities like windows, panes, and text fields to represent scheduling objects on the user's desktop.

On top of this core common specializations such as a job-shop or a flow-shop schedule and several optimization algorithms are implemented. A further derivation layer consists of specific classes for steelmaking applications.

2 SCHEDULING OBJECTS

The main scheduling object is a schedule consisting of three conceptual parts:

- a list of resources with scheduled allocations,
- a list of jobs with their operations, and
- a list of constraints.

The main design criteria for a schedule are:

1. the representation should be as flexible as possible to enable the representation of schedules of different applications with different resources and jobs,
2. support of scheduling tasks initiated either by a user or iterative improvement,
3. scheduling tasks must be very efficient to provide users an immediate feedback and to fasten the optimization algorithms, and
4. a schedule should be an object that can be copied efficiently.

Flexibility and efficiency are two potential conflicting objectives for which a trade-off must be found. Thus, pointer arithmetic is used for the core schedule instead of pure object oriented representation. Lists are realized as pointer arrays based on the template mechanism of C++. They can be extended dynamically and store only pointers, because it is not know in advance how much storage is needed for the objects. A typical resource points to a double-linked list of allocations that store when operations are performed on the resource and a job points to a double-linked list of allocations describing allocations of a job.

The dynamic links between allocations support the algorithm that checks and enforces temporal consistency of all allocated operations. Each time an operation is moved in the schedule, the adjacent allocations of its resource and its job are adjusted temporally. An adjustment of another allocation will be propagated. This consistency mechanism is complete because only simple temporal algebra is used.

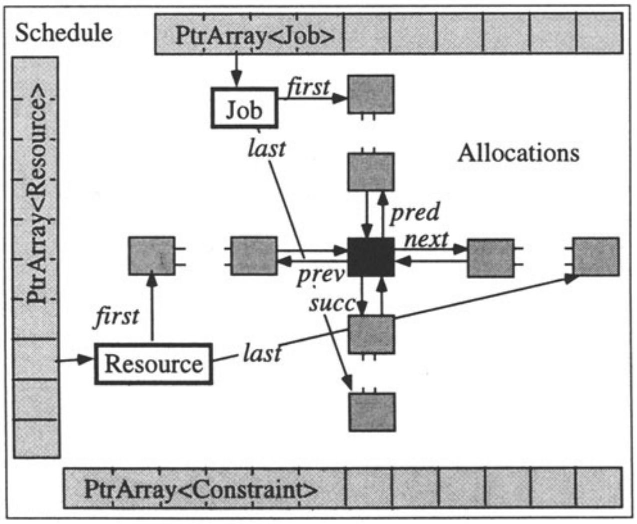


Figure 1 Structure of a Schedule

The sequence of the jobs in the list of all jobs represents also the sequence of jobs in the schedule. This sequence may be changed by certain scheduling tasks, but a move of a job is further propagated to each resource on which this job is scheduled to move also the allocation accordingly.

We define one abstract root schedule class which realizes many methods sufficient for handling schedules. However, we specialize a schedule to reflect certain characteristics of job shops and flow shops. For a certain application, we may further specialize to represent in this class application specific information and to overload general methods by more efficient domain-dependent strategies.

Methods dependent of the schedule type are the methods that realize different scheduling tasks. For the efficiency of scheduling tasks it is better if inverse scheduling tasks can be defined instead of copying whole schedules. Moving a job from one position to another in a flow-shop is more efficient, because its operations are in the same sequence for both jobs and its inverse task can be defined easily by storing the old position. In a job-shop, it is not clear what an exchange of two jobs means. The jobs may be allocated on different resources which cannot be used for the other job and two jobs may be scheduled overlapping. We can define the move, but for the inverse task we must return to the old schedule by copying the old schedule. For a flow-shop, the move of single operations is not useful. Each schedule type has its own method for deciding which scheduling tasks are applicable and how it can be performed, if possible.

2.1 Allocations

An *allocation* is an assignment of an operation being part of a job to a resource. The allocation is a temporal interval consisting of start, duration, and end. Simple allocations are used for resources that can perform only one operation at a time and thus cannot overlap temporally. Allocations on a resource are linked forwards and backwards. For the basic type of allocation, a temporal sequence can be concluded, but the derived *capacitive allocation* may overlap. To find the job and the resource object to which the allocation belongs, two pointers to these objects are stored. Further pointers to the next and to the previous allocation of the job exist. If a predecessor allocation exists on the resource, one or more *allocation constraints* may be stored.

2.2 Resources

A resource stores the operations of different jobs to be performed on it. Resources are generated from a description of the *scheduler* class. Resources maintain their own list of *resource constraints.* The class *resource* is an abstract class from which no objects can be generated.

Most typical is the *non-sharable resource* on which operations required by a job are allocated. These allocations are stored in double-linked list whereby the sequence in the list reflects also the temporal sequence. The link structure is efficient for scheduling tasks such as swapping allocations. The non-sharable resource has a pointer to the first and to the last allocation. A non-sharable resource knows how to perform scheduling tasks as allocating, swapping, moving, or deleting an operation. Attributes may be defined such as minimal idle time or required set-ups. If it allocates operations or modifies the allocations, it maintains the constraints derived from the idle time and the set-up attributes accordingly.

A *sharable resource* can be used by several jobs simultaneously. An example is the space to store products. This space is often limited but several products produced in different jobs may be stored at the same time. Since such resources have a limited capacity and different operations may require different amounts of the resource, a *capacitive allocation* is used to incorporate additional attributes for size and amount. Scheduling tasks, such as moving or shifting an allocation must be realized for sharable resources. The maximal capacity of a sharable resource is a hard constraint, but the equilibre load may be a soft constraint. A typical example is energy consumption which has an upper limit. For a cheap production however, it is important to distribute the energy consumption as much as possible over the whole production period because peak's in the energy consumption are often expensive.

A *resource group* represents a group of almost identical resources. For the production process, it makes no difference which of them is used because all have the same capabilities. Yet, an objective such as to minimize the number of used resources and constraints on subsequent allocations may constrain the usage of resources. The only scheduling task a resource group must support is the move of an operation from one of its resources to another. Other tasks are deferred to the resources. A special method of a resource group is the method of finding the best resource for an allocation. Generally, this will be the first available resource. Derived classes will overload this method with more sophisticated heuristics.

2.3 Orders and Jobs

An *order* describes the product to be produced, the required operations and their required sequences, its priority, and such constraints as the release and the due date. The operations and their temporal relations are described in a *process plan.* A *job* describes the performance of an order in the shop floor. It may be scheduled to produce several orders. The main conceptual difference is the specification of planned starting and finishing times for the scheduled operations, whereas the order describes only the requirements.

In some domains the order does not need to have a process plan to describe the required operations and their temporal dependencies because the sequence is for all jobs mainly the same. In this case a job is generated from an order by following predefined rules of its domain. A process plan is then constructed for the job.

When a job is generated from an order, some attributes like the release and the due date are copied into the job. However, a job also has a pointer to its order because some computations are dependent on the produced good that is not represented in the job object. A job points at its first and its last allocation and these are linked in the "allocation network". For a simple flow-shop job, the chain of job allocations describes a sequence of operations. If a more complex temporal dependency must be described, interval relations are used (Dorn 1995b). Jobs have a unique identifier to enable pointing to the same job in two schedules. Furthermore, a job maintains its own list of *job constraints*. If certain operations of a job are modified, the job updates these constraints accordingly. If the last operation is moved, the tardiness constraint is updated.

2.4 Scheduling Tasks

Scheduling tasks are a paradigm for the coupling of automatic scheduling with user actions and is derived from concepts in model-based knowledge acquisition (e.g. Bylander and Chandrasekaran 1988). In principle, we model each action that can be performed by the user as a scheduling task. A scheduling task is described by a class that provides all types of tasks a uniform interface. If a new task is to be defined, all methods of this interface must be realized. If a task is initiated by the user, all necessary data are stored to undo or redo it. The definition of an inverse task also supports the iterative improvement methods. With such a search method, we apply a scheduling task to check whether a task leads to an improvement. To evaluate the schedule, operations and jobs of the schedule have to be adjusted. If other alternatives are tried, we must return to the old schedule. For complex applications it is more effective to have a task that makes the last change undone than copying a whole schedule. In cases in which no reverse task can be specified, we store the whole schedule before performing the task. Additionally, for tabu search, the inverse tasks are used as a tabu criterion, thus forbidding cycles during search.

The realization of scheduling tasks is dependent of the schedule type. So the performance in a job-shop and in a flow-shop can differentiate and some tasks are not applicable in all schedule types. For example, the move of an operation in a flow-shop and the exchange of a job in a job-shop are not allowed.

Following scheduling tasks are defined: to allocate a job as early as possible, to allocate a job after another job, to allocate a job at a certain time, to remove a job (back into the list of orders), to exchange two adjacent jobs, to move a job to another position, to exchange an operation with an adjacent operation, to move an operation to another place on the resource, to move an operation to another resource, and to shift an operation. This set of operations can be extended easily if other tasks are becoming necessary for an application.

3 CONSTRAINT EVALUATION

The evaluation of schedules in DÉJÀ VU is based on the evaluation of individual *constraints*. Constraint types are differentiated and we define, for example, tardiness constraints. If all tardiness constraints of a schedule are evaluated, the tardiness of jobs is a *measure* of the schedule. For a certain application, different measures can be defined. In a preference-setting dialog, the user can select which of these defined measures shall be evaluated for the next schedule construction process. The settings can be assigned to a schedule thus constructing schedules with different evaluations.

3.1 Constraint Evaluation

A constraint is a relation between two or more scheduling objects or attributes of them mapped on a satisfaction degree which evaluates how well the constraint is satisfied in the actual schedule. A typical example of such a relation is the tardiness of a job. A due date indicates when a certain job should be completed, which is related to the scheduled finishing time. The relation is mapped on a satisfaction degree that indicates how good this constraint is satisfied. If the finishing time equals the due date, the satisfaction of this constraint is considered to be very good. Otherwise it is considered to be poor. This exact meeting of a due date is modeled by a *tardiness constraint.* The relaxed form where a too early completion is also good is realized by a *lateness constraint.* A tardiness constraint shall illustrate how a constraint is specified and how its satisfaction is computed.

If a job has a due date (DD), the job creates a tardiness constraint having two parameters "OptimalDeviation" (OD) and "LeastAcceptableDeviation" (LAD). If the deviation between due date and finishing time (FT) is smaller than the optimal deviation, the constraint evaluates to 1.0. If it is larger than the least acceptable deviation, it evaluates to 0. Otherwise, it is computed as follows:

$$Satisfaction(J_i) = LAD - \frac{|DD(J_i) - FT(J_i)|}{LAD - OD}$$

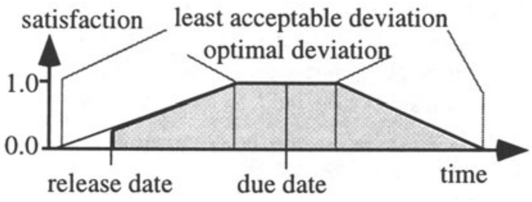


Figure 2 Satisfaction Degree of a Tardiness Constraint

3.2 Constraint Types

Below the abstract root *constraint* class, four abstract constraint classes are defined describing relations between different scheduling objects. An *allocation constraint* relates an allocation with its predecessor. If this sequence is changed, the resource updates this constraint. A *job constraint* relates different attributes of a job. If one of these attributes is changed, the constraint is updated by the job. A *resource constraint* describes a relation between different objects and attributes of a resource. The update is initiated by the scheduling object if all changes on this resource are finished. The fourth kind is a form relating objects of the whole schedule. The *schedule constraint* is maintained by the schedule. The four described abstract classes support the construction of new constraint types because they define a common interface and a predefined mechanism to create and update them. The scheduling objects only know this interface, and the allocation can update a constraint without knowing which actual constraint type it is. If a new allocation constraint type is defined, a derived class of an allocation has to insert this constraint, but no further changes need to be made.

All constraints defined below the four classes are no longer abstract. These constraint types describe actual relations between scheduling objects. After being updated, they will have a satisfaction degree which is used to evaluate a schedule. To reflect that different constraint types have different importance for the application, constraint types are associated with a weight factor between 0 and 1. The sum for all types is defined as "1". If several constraint types are defined, a weight of e.g. .4 means that this constraint type has a great influence on the evaluation function. Another attribute describes a threshold to differentiate soft and hard constraint violations. A constraint satisfaction below this threshold indicates that the constraint must be repaired to get a legal schedule. If the threshold is set to 0, no repair will be necessary.

A special constraint which is elaborated upon is the *compatibility constraint.* It is a relation between subsequent operations assigning a value to this pair, reflecting how optimal it is to schedule both after each other. In the process industry, resources are often infiltrated with residuals of the produced good which may spill subsequent products. This infiltration can either be accepted (if small enough), or some cleaning operation must be scheduled as well. A compatibility constraint can represent the cost of a cleaning operation or the quality-loss due to the infiltration. For some processes, such as steel making, cleaning operations are either not possible, or too expensive. It is therefore important to find sequences that incorporate only a small infiltration. Thus, the threshold cannot be "0". Compatibility constraints can be seen as a prototype of the way new constraints can be integrated in the framework. For allocations having such a compatibility aspect the *compatible allocation* class was derived from an allocation. It creates a compatibility constraint if certain conditions hold. Compatibility constraints and the way they are handled are explained in more detail in (Dorn and Slany 1994).

4 REUSABILITY OF DÉJÀ VU

With the DÉJÀ VU Class Library we have implemented a scheduler for the Böhler company in Kapfenberg (Austria) to schedule heats in a steelmaking plant. This application described in detail in (Dorn and Shams 1995) is a prototype for industrial applications, characterized by a lot of domain-dependent data that users want to see on their computer desktop. Moreover, many of the preferences in solving subproblems must be applied. These very domain-dependent features are realized by new derived classes. For example, the existing order class with 10 attributes

must be specialized to read more attributes (about 30). Nevertheless, techniques as presenting an order graphically, deleting an order, or to schedule an order need not be re-implemented. These modifications have no effect on the interaction classes or on the algorithms that require the evaluation and scheduling tasks. Another refinement that must be performed for the application is the design of new windows. This can again be achieved by deriving from existing window classes.
We estimate that only about 10% new code has been developed for the application. A second application for a different steel plant has been used as a further test-bed which shows that approximately the same effort is required here.

The following graphics show two views of the schedule whereby the first is generic and the second specific for the application. The main schedule window shows the whole schedule in a scroll pane. Resources are shown below each other. For example, the first resource in the figure is an electric arc furnace (EAF), then one sees a group of ladles. The allocations on these resources are depicted by small panes. The last two resources are sharable resources that describe the space requirements in the teeming bay and the load of the workers in the teeming bay. The bottom window shows the total evaluation and the mean chemical compatibility. With a popup menu the user can also select other measures.

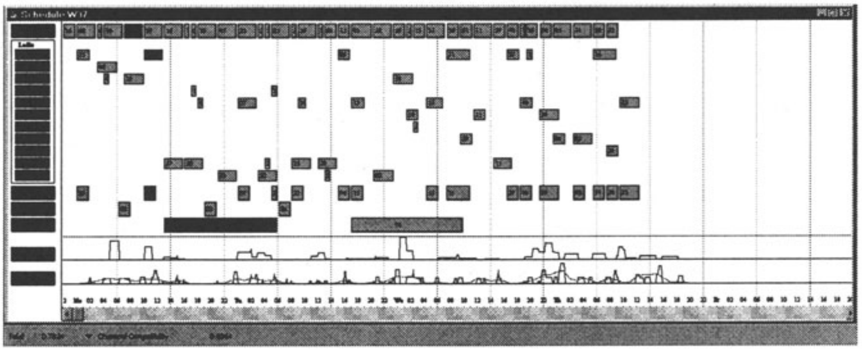
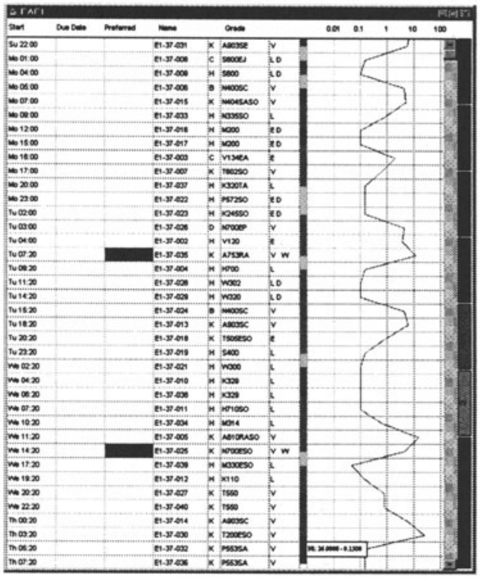


Figure 3 Graphical Representation of a Schedule

By clicking on the panes in the window, the user can select operations and jobs to move them to other places in the schedule. If an operation or a job is selected, menu commands can also be applied to the selected object.

The information shown in the schedule window is not always sufficient. With a double-click on the resource name's pane, the system opens a window specific for a resource. The following figure shows a window for the electric arc furnace. On the right side one can see a logarithmic diagram that visualizes the chemical content of subsequent orders on the furnace.



Start	Due Date	Preferred	Name		Grade	
Su 22:00			E1-37-031	K	A803SE	V
Mo 01:00			E1-37-008	C	S800EJ	L D
Mo 04:00			E1-37-009	H	S800	L D
Mo 05:00			E1-37-006	B	M400SC	V
Mo 07:00			E1-37-015	K	M404SASO	V
Mo 09:00			E1-37-033	H	M335SO	L
Mo 12:00			E1-37-016	H	M200	E D
Mo 15:00			E1-37-017	H	M200	E D
Mo 16:00			E1-37-003	C	V134EA	E
Mo 17:00			E1-37-007	K	T802SO	V
Mo 20:00			E1-37-037	H	K320TA	L
Mo 23:00			E1-37-022	H	P572SO	E D
Tu 02:00			E1-37-023	H	K245SO	E D
Tu 03:00			E1-37-026	D	M700EP	V
Tu 04:00			E1-37-002	H	V120	E
Tu 07:20		■	E1-37-035	K	A753RA	V W
Tu 08:20			E1-37-004	H	H700	L
Tu 11:20			E1-37-028	H	W302	L D
Tu 14:20			E1-37-029	H	W320	L D
Tu 15:20			E1-37-024	B	M400SC	V
Tu 18:20			E1-37-013	K	A803SC	V
Tu 20:20			E1-37-018	K	T506ESO	E
Tu 23:20			E1-37-019	H	S400	L
We 02:20			E1-37-021	H	W300	L
We 04:20			E1-37-010	H	K329	L
We 06:20			E1-37-038	H	K329	L
We 07:20			E1-37-011	H	H710SO	L
We 10:20			E1-37-034	H	M314	L
We 11:20			E1-37-005	K	A810RASO	V
We 14:20		■	E1-37-025	K	M700ESO	V W
We 17:20			E1-37-039	H	M330ESO	L
We 19:20			E1-37-012	H	K110	L
We 20:20			E1-37-027	K	T550	V
We 22:20			E1-37-040	K	T550	V
Th 00:20			E1-37-014	K	A803SC	V
Th 03:20			E1-37-030	K	T200ESO	V
Th 06:20			E1-37-032	K	P553SA	V
Th 07:20			E1-37-036	K	P553SA	V

Figure 4 Resource Window

5 CONCLUSIONS

To improve the reusability of DÉJÀ VU for new applications, it seems important to define an order description language from which the system can automatically generate the order class. Although the logic behind this class is simple the construction is error-prone.

At the moment, only a very simple temporal logic is used to describe the temporal constraints between operations of one job. Since we use only before and after-relations, we cannot express any possible constellation of operations. The temporal consistency mechanism incorporated into the system is based on this simple model. Using the full expressiveness of Allen's interval algebra (Allen 1983) for the consistency mechanism would be computationally too expensive (NP-complete), but it seems tractable to apply it in describing the temporal relations of process plans.

The most important extension however, will be the introduction of reactive scheduling. The main problem for the application at Böhler is the daily work with the adaptation of the schedule to react on unexpected events such as new orders, machine break-downs, destroyed products, etc. Based on (Dorn, Kerr, and Thalhammer 1995), we have already built a reactive scheduling prototype for Böhler which shall be integrated into DÉJÀ VU. Since this prototype has worked in a simulation model we must now test it in a real domain.

A documentation of the scheduling class library is publically available at: http://www.dbai.tuwien.ac.at/proj/DejaVu/Document.

6 REFERENCES

Allen, J.F. (1983) Maintaining Knowledge about Temporal Intervals, *CACM* **26** (11), pp. 823-843.

Bylander, T. and Chandrasekaran, B. (1988) Generic Tasks in Knowledge-based Reasoning: The 'right' Level of Abstraction for Knowledge Acquisition, in *Knowledge Acquisition for Knowledge-Based Systems*, B. Gaines & J. Boose (eds), pp. 65-77, London: Academic Process.

Dorn J. and Slany, W. (1994) A Flow Shop with Compatibility Constraints in a Steel making Plant in Zweben and Fox(eds) *Intelligent Scheduling*, Morgan Kaufmann, pp. 629–654.

Dorn, J. and Girsch, M. (1994) Genetic Operators Based on Constraint Repair, *ECAI'94 Workshop on Applied Genetic and other Evolutionary Algorithms*, Amsterdam, August 9.

Dorn, J. (1995a) Iterative Improvement Methods for Knowledge-based Scheduling *AICOM Journal*, pp. 20–34 March.

Dorn, J. (1995b) Case-based reactive scheduling in Roger Kerr and Elisabeth Szelke (eds) *Artificial Intelligence in Reactive Scheduling*, London: Chapman & Hall, pp. 32-50.

Dorn, J. Kerr, R. M. and Thalhammer, G. (1995) Reactive Scheduling – improving the robustness of schedules and restricting the effects of shop floor disturbances by fuzzy reasoning, *International Journal on Human-Computer Studies* **42**, pp. 687–704.

Dorn, J. and Shams, R. (1996) Scheduling High-grade Steel Making *IEEE Expert* February, pp. 28-35.

Dorn, J., Girsch, M., Skele, G., and Slany, W. (1996) Comparison of Iterative Improvement Techniques for Schedule Optimization, *European Journal on Operational Research*.

7 BIOGRAPHY

Jürgen Dorn received his M.S and Ph.D. degrees in computer science from Technische Universität Berlin, Germany. From 1989 to 1996 he has headed the Knowledge-based Scheduling group of the Christian Doppler Laboratory for Expert Systems in Vienna. Currently he works as Ass. Prof. at Technische Universität Wien, Austria. His research interests include real-time planning, knowledge-based scheduling, case-based reasoning, and software engineering.

Mario Girsch received his M.S. in computer science from Technische Universität Wien. He has worked since 1995 for the Christian Doppler Laboratory for Expert Systems in Vienna. His interests are in knowledge-based scheduling, case-based reasoning, and genetic algorithms.

Nikos Vidakis received his B.Sc.. (Hons) degree from University of Northumbria at Newcastle (England) and the Ph.D. degree from Technische Universität Wien. He has worked from 1994 to 1996 for the Christian Doppler Laboratory for Expert Systems in Vienna. His interests are in tools for developing user interfaces.