Modern production management has to face considerable organizational problems due to the shifting from mass production to more specialized small lot-size production. The production of smaller lot sizes and the requirements for fast reactions to market demands poses a manifold of constraints on the usage of available resources on the shop-floor level. To be competitive in the global market, companies must especially solve the efficient coordination in their production to avoid as much as possible of the so-called hidden costs.

The adequate coordination of all activities that have to be performed to fulfill a customer’s order are a key process to eliminate these costs. Due to the diversity of products as well as the diversity of required resources such as human resources, machines, energy, and others, it becomes impossible to realize such scheduling systems with standard software. Each production unit is unique and requires dedicated solutions and moreover, it will change due to new production technologies available and changing market demands.

A promising software engineering technique are thus application frameworks that offer reusable semi-complete applications that can be specialized to produce custom applications [Fayad and Schmidt 1997]. An application framework is based on a domain analysis that states the basic assumptions on a number of related applications. The input to such a domain analysis is due to [Prieto-Díaz 1990] several sources such as literature, existing implementations, customer surveys and more. For the domain of production scheduling there exists a huge body of theory, e.g., [Conway et al. 1963], [French 1982], and [Blazewicz et al. 1993] and a lot of dedicated journals. However, these theories have all an idealized view on scheduling applications. They abstract from real-world problems. It is expected that software developers reuse the design of such theories (mainly algorithms).
How production data is gathered, how information is displayed to a user, and how the user can participate in the solution is not treated in these approaches. Typically, these interfaces are the most tedious development work.

As a consequence, "high-level scheduling languages" were advised to improve the efficiency of developing scheduling systems. Systems such as ISIS [Fox 1994], OPIS [Smith et al. 1994], or SONIA [Le Pape 1994] offer a constraint-based language to the developer to express scheduling problems. The architecture of the system is pre-defined and the scheduling algorithm is more or less fixed, the user can only tune the applied heuristics. There have been also attempts to build application generators for scheduling systems as for example, the Protos-System [Sauer 1993], but the success of such generators in industrial practice was small. Today, several research groups see the limited reusability of both approaches. For example, the Robotics Institute of the Carnegie Mellon University, a very well-known group for intelligent scheduling systems has just started to design also a scheduling framework (OZONE/DITOPS Project, http://www.cs.cmu.edu/afs/cs/project/ozone/www/DITOPS/ditops.html).

The DÉJÀ VU Scheduling Class Library is such an application framework for modern production scheduling. Based on a broad domain analysis and the development of two industrial scheduling systems, we have started in 1995 to develop this framework. We will describe the basic assumptions that lead to the design of the scheduling framework and describe how dedicated applications can be produced with DÉJÀ VU.

## 21.1 Design Principles of DÉJÀ VU

DÉJÀ VU is a framework of C++ classes to support developers in the construction of scheduling systems for industrial production processes. The design of the framework was directed by the following criteria:

- the scheduler’s 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,
- the scheduler applies iterative improvement methods to optimize solutions, and
- the framework should be extendible and refinable.

The goal is to support the construction of dedicated scheduling systems for a specific application rather than developing a system capable of scheduling different applications.

### 21.1.1 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 as processing times or due dates and objectives such as minimization of the makespan or of the mean flow-time are
often insufficient to represent industrial problems. The DÉJÀ VU framework supports further constraints and objectives like compatibility constraints (chemical and format), idle time constraints, minimization of substitutable resources, restricted capacity, or equilibre load of sharable resources. These constraint types have been derived from several scheduling problems in the steel industry. Other constraint types for new domains can be generated by deriving constraint types from existing types with minimal effort due to the general approach to represent them.

From a software engineering point of view, we achieve this by defining a standard interface for constraints. Thus it is possible to call constraint methods in other components without knowing which constraints are defined for a certain application. If a new kind of constraint must be defined for an application, only the new constraint class must be defined. An application specific class can then call the constructor method without any further manipulations.

Many constraints of industrial production environments are soft and can be relaxed. Moreover, constraints may be contradictory and a trade-off between constraints must be found for a good solution. DÉJÀ VU meets these requirements as follows: A constraint is a relation between two or more scheduling objects. 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. A schedule is evaluated by a weighted aggregation of all satisfaction degrees. Further, for each constraint type a threshold can be specified to decide whether the constraint violation is hard. The constraint’s weight and threshold can be modified in a scheduling session to experiment with different settings.

### 21.1.2 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, new production techniques and objectives are developed regularly, and the demand for products changes. The software must therefore be adaptable and under 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 the described problem best. The user has always either the ability to let the system schedule automatically or to perform some scheduling tasks manually. The user should always be able to change the schedule that was constructed by the system, but the system should show new conflicts effected by this change to the user. Furthermore, the user should have the ability to “freeze” some part of the schedule and let the system improve the remaining part. DÉJÀ VU supports mixed-initiative scheduling by defining scheduling tasks for schedule alterations that provide a common interface with methods for undoing, redoing, evaluating, etc.
21.1.3 Iterative Improvement Methods

An iterative improvement method is an optimization technique which starts with an initial solution and tries to find better solutions by “local” modifications. The initial schedule can be constructed randomly, by some constructive method, or by a 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, for example, the exchange of two adjacent jobs, the move of an operation from one machine to another machine having the same capabilities. Scheduling tasks construct always legal schedules that do not violate hard constraints. 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 to perform the task leading to the best “neighbor”. To determine whether an improvement can be achieved by a task, the comparison of schedules by an evaluation function must be possible. The most efficient look-ahead is achieved when the schedule evaluation can be determined locally.

A simple hill-climbing algorithm would accept only schedules which evaluate better. Unfortunately, scheduling problems usually have many solutions that differ in their quality, and good solutions are not direct neighbors. Therefore, a search method based on local improvements can be easily trapped in a local optimum. An important feature of all iterative improvement methods is therefore the capability to escape from local optima. However, with this ability, it becomes more likely to search in cycles and some kind of control to avoid repetitions is needed.

DÉJÀ VU allows the user to select between different improvement methods (tabu search, simulated annealing, iterative deepening, and genetic algorithms) and to set different parameters of these algorithms individually. Furthermore, if a combination of techniques seems to be appropriate (e.g. tabu search with some stochastic technique) 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].

21.1.4 Reusability of the Scheduling Framework

The design criteria mentioned so far have lead to a software architecture that should be applicable to a wide range of scheduling applications. The main principle to support the reusability of the developed framework is the object-oriented design with the support of some well-known design patterns [Gamma et al. 1994] such as

- the abstract factory (e.g., create a domain dependent schedule object without specifying its concrete class),
- the factory method (e.g., create domain-dependent orders in the domain-independent order director class),
The Déjà Vu Scheduling Class Library

- chain of responsibility (giving the most specific or selected user interface element first the chance to react on a user action to pass it to its supervisor if the element does not know how to react),
- command (encapsulate a user action as an object that has an common interface for undoing or redoing the action),
- iterator (provide a way to access elements of an aggregate object such as a list of constraints sequentially without knowing its implementation),
- observer (when an object such as a schedule changes, all dependents should be notified without calling all these dependents by the scheduling object),

and more which will be partly explained later in detail.

However, the critical task in designing reusable software (or reusable classes) 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. Also the associations between these objects are obvious. Unfortunately, this theoretical work does not integrate user interaction with schedule optimization. As previously mentioned, the user of complex industrial applications should be capable of modifying a proposed solution of the system.

Our approach to support the reuse is as follows: The core of DÉJÀ VU is a framework of classes realizing basic scheduling theory. Furthermore, basic forms for the representation of constraints are realized by abstract classes. For example, many constraints such as due dates or release dates refer to attributes of a job. Thus, we have defined an abstract class for job constraints.

The 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 we have implemented common specializations as a job-shop or a flow-shop schedule and several optimization algorithms. A further derivation layer consists of specific classes for subdomains. At the moment, there exist one subdomain for steelmaking applications and we are just designing another subdomain for computer scheduling.
21.2 The Scheduling Core

The main scheduling object is a schedule. It stores the temporal allocation of operations for all resources. There may be different instances of a schedule because the optimization algorithm must store intermediate schedules and the user shall be able to experiment with different settings, thus constructing different schedules. The user must be able to return to schedules produced earlier. A schedule object consists 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:

- the representation should be as flexible as possible to enable the representation of schedules of different applications with different resources and jobs,
- support of scheduling tasks initiated either by a user or an iterative improvement method,
- scheduling tasks must be very efficient to provide users an immediate feedback and to fasten the optimization algorithms, and
- a schedule should be an object that can be copied easily (especially optimization techniques such as Genetic Algorithms rely heavily on an efficient technique to produce new schedules).

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

The dynamic links between allocations support the algorithm that checks and enforces temporal consistency of all allocated operations. Each time an allocation is moved in the schedule, the start and end points of the adjacent allocations must be adjusted. The adjustment of another allocation will be propagated. This consistency mechanism is complete because only simple temporal algebra is used. The efficiency of the mechanism relies strongly on the linkage structure.

The sequence of the jobs in the list of all jobs also represents the sequence of jobs in the schedule. When this sequence is changed by a scheduling task, this change is further propagated to each resource on which this job is scheduled to move also the allocation accordingly. The following figure illustrates the core schedule representation.
An abstract root schedule class realizes already many methods sufficient for handling schedules. Its main service provided to other classes is the temporal consistency enforcing mechanism that ensures that specified temporal relations between allocations hold. However, a schedule is specialized to reflect certain characteristics of job shops, flow shops, and schedules consisting of only one machine. 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. For example, a schedule has a method that generates a set of scheduling tasks that can be applied to find a neighbor. The basic abstract schedule class defines this method not because the differences between a flow-shop and a job-shop are too great. The flow-shop schedule realizes a very general version which generates many tasks and the schedule class for a dedicated application may generate a smaller set by applying heuristics which results in a more efficient search through the neighborhood.

Methods dependent of the schedule type are the methods that realize different scheduling tasks. The efficiency of scheduling tasks is supported if inverse scheduling tasks can be defined. In this case not whole schedules have to be copied. 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 that can not be used for the other job. Moreover, two jobs may be scheduled simultaneously. 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 is performed if possible

### 21.2.1 Temporal Reasoning Component

Temporal reasoning is an inference that occurs in every scheduling application and should be reusable always. Different characteristics how temporal reasoning is applied in different applications should be selectable for the developer. Two different types of temporal knowledge must be supported:

- quantitative time (e.g. Monday, 10 pm) and
- qualitative temporal relations (e.g. operation 1 must start before operation 2).

A time point is the basic class to represent temporal knowledge. It is an abstract data type with overloaded operators and access methods. Currently, two temporal representations are supported: time can be represented by a simple long integer or by a week time consisting of the number of the week, a day, an hour, and minutes. The representation is selected by a simple class variable. The design allows also an easy extension for fuzzy time points.

Time points are used to define time intervals consisting of a starting and a finishing time point and a duration. Other classes as the allocation, the operation, the job, and the schedule are derived classes which means that each object of these classes has a starting and finishing time point and of course, they reuse the methods defined for intervals.

Temporal relations are hard temporal constraints on the sequence of intervals. Simple sequencing relations such as before or after occur in almost every application. We define furthermore a “meets”- relation and its inverse relation which constrains two intervals to be immediately after another without any slack time. These relations are defined as a class of its own to support different reasoning mechanisms. In the future, we want to support also the full expressiveness of Allen’s interval calculus [Allen 1983] with 13 different relations. Whether we can support the same propagation as Allen is still under investigation because its combinatorial complexity is intractable.

### 21.2.2 Orders and Jobs

An order contains a description of one or more products to be produced and a process plan that describes how to produce these products. The process plan contains the required operations and their prescribed sequences. Furthermore, an order may have attributes such as a priority, and constraints as the release and the due date. A job describes the performance of an order in the shop floor. A job may be scheduled to produce several orders. However, the main conceptual difference is the specification of the planned starting and finishing times for the scheduled operations. In contrast, the order describes only the requirements.
In some domains (especially flow-shop domains), the order does not need to have an explicit process plan to describe the required operations and their temporal dependencies because the sequence is for all jobs the same. In this case, a job is generated from an order by following predefined rules of its application. 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 to enable computations dependent on the produced good that is not represented in the job object. A job points to its first and last allocation which are linked in the allocation network. For a simple job in a flow-shop, the chain of 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, for example, the last operation is moved, the tardiness constraint must be updated if it exists.

### 21.2.3 Capacitive Reasoning Component

Operations to be performed require prescribed resources. These resources have typically restricted capacity. There are resources that can handle only one operation at a time (e.g. a machine) and there are resources (e.g. a stock) which have restricted capacity but several operations may use it at the same time.

A resource object stores which operations are to be performed on it. For the set of all resources, an array of unlimited length is used because the schedule class shall not be restricted already to a certain application with a predefined number of resources. The resources and the array are generated from a description of the application in a derived class of the scheduler class. If an additional resource is to be considered in the schedule, no modification of the schedule class is necessary. Pointers to resources are stored instead of the resources themselves because the usage of different resources requiring different memory space shall be supported. Resources maintain their own list of resource constraints and they have a list of so-called working zones which are time intervals that describe time periods when the resource is usable. So if a machine is broken down or in maintenance, these periods are not included.

Different kinds of resources are distinguished: from the abstract class resource the classes non-sharable resource, sharable resource, and resource group are derived.

### Allocations

An allocation assigns an operation being part of a job to a resource. Simple allocations are used for resources that can perform only one operation at a time and thus cannot overlap. Allocations on a resource are linked forwards and backwards. For the basic type of allocation, this sequence means also a temporal sequence, but the derived capacitive allocation may overlap. An allocation is assigned a number, making it unique to a resource. This is necessary, because we store a pointer to an allocation in a scheduling task to exchange two allocations, for example. We need a unique identification, to point to the same allocation in two schedules. 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 for this allocation. Another derived class is defined for allocations having a sequence which is constrained by compatibility constraints.

**Non-Sharable Resources**

On a non-sharable resource operations are allocated that are required for a job. These allocations are stored in double-linked list whereby the sequence in the list reflects also the temporal sequence. The link structure is also more efficient for scheduling tasks such as swapping allocations or moving an allocation to another place. A non-sharable resource has a pointer to the first and the last allocation of the resource.

A non-sharable resource knows how to perform scheduling tasks such as allocating an operation, swapping an allocation, moving an allocation to another place on the resource, or deleting an allocation. It may have defined attributes such as, for example, minimal idle time or required set-ups. If the non-sharable resource allocates operations or modifies the allocations, it maintains the constraints derived from the idle time and the set-up attributes accordingly.

**Sharable Resources**

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. Another example are a group of workers that may handle several jobs simultaneously. Since such resources are limited 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 a soft constraint which could be considered is the equilibre load. 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 peaks of high energy consumption are often expensive.

**Resource Groups**

A resource group is used to represent 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, objectives such as, for example, minimizing the number of used resources and constraints on subsequent allocations may constrain the usage of the group. 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 individual resources. A special method of a resource group is the method to find the best resource for an allocation. The resource group will return the first resource that is free for the desired interval. Derived classes will overload this method with more sophisticated heuristics.
21.3 Schedule 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 and makespan 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 constraint types or measures can be defined. In a preference-setting dialog, the user can select which of the defined measures shall be evaluated for the next schedule construction process. These settings can be assigned to a schedule thus constructing schedules with different evaluation settings which is used to support what-if games in the sense that a user neglects some constraints to see whether this leads to a better schedule.

21.3.1 Constraint Evaluation

A constraint is a relation between two or more scheduling objects or attributes of scheduling objects 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 evaluates 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. A relaxed form where a too early completion is also considered to be good is realized by a lateness constraint.

The satisfaction of a tardiness constraint shall used as a prototype to illustrate in the next figure how the satisfaction of a constraint may be specified and how it is computed for a given relation.

If a job has a defined due date, and the measure tardiness was selected by the user, the job class creates a tardiness constraint in its constructor having two parameters “OptimalDeviation” and “LeastAcceptableDeviation”. If the deviation between due date and finishing time 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:

\[ \text{Satisfaction}(J_i) = (\text{LeastAcceptableDeviation} - | \text{DueDate}(J_i) - \text{FinishingTime}(J_i) |) / (\text{LeastAcceptableDeviation} - \text{OptimalDeviation}) \]

![Figure 21.2: Satisfaction Degree of a Tardiness Constraint](image)
21.3.2 Constraint Types

Below the abstract constraint class, four further abstract constraint classes are defined describing relations between different scheduling objects. An allocation constraint is a relation between an allocation and its predecessor on a resource. If this sequence or the distance to the predecessor is changed, the allocation updates this constraint. A job constraint is a relation over 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 this resource. The update is initiated by the scheduling object if all changes on this resource are finished. The fourth kind is a form for constraints 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 by defining 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 concrete classes. 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 .4, for example, means that the 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 should be elaborated upon is the compatibility constraint, with its two specializations chemical constraint and format constraint. A compatibility constraint is a relation between subsequent operations that assigns a value to this pair of operations, 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 small infiltrations. Thus, the threshold cannot be 0. Similar constraints exist for some machines as, for example, rolling mills, on the format or size which can be described by a format constraint. The compatibility constraints can be seen as a prototype of the manner new constraints can be integrated in the framework. For allocations having such a compatibility aspect, the compatible allocation class is derived from an allocation. When certain conditions hold, the allocation creates a compatibility constraint. Compatibility constraints and the way they are handled are explained in more detail in [Dorn and Slany 1994].
21.4 **User Interaction**

The design of DÉJÀ VU is user-oriented which means that the user always has the control over the scheduling activities. The system supports the user as far as possible: the user shouldn’t be unnecessarily restricted while senseless actions should be prevented. It should disable actions that violate hard constraints. For example, the system does not allow an operation to be moved to a resource that is not capable of performing this operation. On the other hand, the user should be capable of examining solutions that are evaluated poorly by the system. An inexperienced user should be supported by indicating the possible actions, an experienced user should be able to perform the manipulation of the schedule with as few keystrokes or mouse actions as possible.

### 21.4.1 The supervision hierarchy

User actions are context-dependent. The system must know which actions are allowed in a certain context, and which modifications must be made if an allowed action is performed. To determine this, one has to model which part of the user interface is active. Typically, the front-most window is active and only actions manipulating this window are enabled. A window itself may contain different panes from which one pane may be active (or selected). Then, only commands for this pane or commands that change the status (deactivate this pane and activate another) are allowed. However, if a pane of the window is active, there are still commands associated with the enclosing window or with the whole application. To model this behavior flexible, two control hierarchies are defined:

- the *supervision hierarchy* (if the pane has no command handler or its command handler does not handle the actual command, the command is given to its supervisor) and
- the *visual enclosure hierarchy* (if a mouse click cannot be handled by the pane, it is deferred to its enclosing view).

An object that can handle a user action is called a bureaucrat. Each bureaucrat has a supervisor which is again a bureaucrat. If the bureaucrat cannot handle the command itself, it is deferred to its supervisor. The top-most supervisor, having no other supervisor, is the scheduler itself. The scheduler defines also the principal course of a scheduling session by internal states that decide which commands are to be enabled. So the user can first open a list of orders which are visualized in an order window. Then these orders may be scheduled and visualized either by a gantt chart in the schedule window or by a simple list chart in a resource window. Then the user may change the schedule interactively or s/he may start an optimization search which can be stopped anytime.

A static variable of the scheduler, *theExpert*, always points to the most specialized bureaucrat that is in the focus of the last action. If the user clicks for example into an allocation pane which visualizes an allocation in the gantt chart, then commands manipulating single allocations are enabled. If the user clicks in the menu bar, the actual supervision hierarchy from the expert up to the application object is responsible for deciding which menus and which menu items are enabled. Each bureaucrat informs the menu bar which commands it allows.
If the user selects (e.g. by a mouse click) some visual object in a window, this object becomes the expert (e.g. the allocation pane). If a new window is created, either its supervisor – a director – may become the expert, or some subview may become the expert. This expert is the first in the supervision hierarchy to react on a user command and the last one in the chain of this hierarchy to enable commands.

User actions can be classified into five basic categories:

- Selecting a visual object (with a mouse-click or a special key, such as an arrow key)
- Typing characters with the keyboard,
- Calling a menu command either by mouse or a command sequence,
- Moving the mouse, or
- Tracking an object with the mouse.

The basic techniques to perform these user actions are a mouse click, a key stroke, and moving the mouse. For the first type, the system decides to which part of the display the mouse points. Three main parts are differentiated with their associated classes: the menu bar, a window, or the desktop. A message to process the mouse-click is sent to the appropriate class object, either one of the objects TheMenubar, TheDesktop, or an instance of the window class.

If the user clicks in the menu bar, a menu shows all possible commands. A command item can be dimmed to reflect that the command is disabled in the actual context. To establish the menu accordingly to the actual context, all command items are first disabled, then an update method is sent to the expert to enable all the commands which are allowed in the context of this bureaucrat. Before the bureaucrat enables commands, it allows its supervisor to enable commands by calling the update method of its supervisor. Thus, the expert can also disable a command which was enabled by its supervisor and can overwrite the text of a command.

If a window is not active, it is activated by a mouse-click. If it is already active, the window class differentiates which part of the window the mouse points to. If the mouse-click is inside the window, the subview of the window in which the click was performed is determined. The click can be processed by a method of the visual object (also a bureaucrat). If the object does not define this method, the system looks upwards in the visual hierarchy whether another object can process the click. The visual hierarchy is a hierarchy of enclosing views. If a new visual object is defined such as a pane, its constructor is called with an argument representing its enclosing view. This reference is also used to define the position of the new subview by specifying relative coordinates. The enclosing view appends the new view in its list of subviews.

### 21.4.2 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, each action a user can perform is modeled 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 enable a undo or a redo. The definition of an inverse task also supports the iterative improvement methods. In such a search method, a scheduling task is applied to check whether a task leads to an improvement. To evaluate the schedule, we must usually adjust the operations and the jobs of the schedule. If we want to check for other alternatives, we must return to the old schedule. For complex applications, it is more efficient to have an inverse task that undoes the last change than copying a whole schedule. In cases in which no inverse task can be specified, the whole schedule must be stored 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. The performance in a job-shop and in a flow-shop can thus 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. The following scheduling tasks are defined:

- allocate a job as early as possible
- allocate a job after another job
- allocate a job at a certain time
- remove a job (back into the list of orders)
- exchange two adjacent jobs
- move a job to another position
- exchange an operation with an adjacent operation
- move an operation to another place on the resource
- move an operation to another resource
- shift an operation

This set can be extended easily if other tasks become necessary for an application.

### 21.4.3 Graphical Schedule Visualization

Fig. 21.3. shows the different available views on a schedule of the Böhler application. The main schedule window in the background (lower part on the left side) shows the whole schedule in a gantt chart that can be scrolled. Resources and its allocations are shown below each other. The top-most resource in the window is an electric arc furnace (EAF), then a group of ladles followed by a horizontal continuous caster (HCC) and a BEST-unit. The allocations on these resources are depicted by small boxes (allocation panes). The last two resources are sharable resources describing the space requirements in the teeming bay and the load of the workers in the teeming bay.

Two panes in the bottom window show the total evaluation and in this case the mean chemical compatibility. With a popup menu also other measures can be selected.

By clicking the panes in the window, the user can select allocations and jobs in order to move them to other places in the schedule. If an allocation or a job is selected, menu commands can also be applied to the selected object.
The information shown in the schedule window is sometimes insufficient. With a double-click on the resource name’s pane, a resource window that shows the sequence of its allocations as a list is created. The 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.

Figure 21.3: Graphical User Interface of the Böhler Application

21.5 Reusability of DÉJÀ VU

The general DÉJÀ VU Class Library consists of about 150 classes from which one third implements the user interface. On top of this framework we have build about 20 classes (user interface and scheduling theory) for steelmaking applications that realize especially the problem of compatibilities. The top of our application for the steelmaking plant of Böhler Kapfenberg are 10 domain specific classes.
The scheduler for Böhler company in Kapfenberg (Austria) schedules 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, a lot of preferences and heuristics how several subproblems have to be solved must be applied. These 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). However, techniques as presenting an order graphically, deleting it, or to schedule it must not be re-implemented. Further we had to define one new scheduling task and two new constraint types. However, these modifications have no effect on the interaction classes or on the algorithms that need the evaluation and scheduling tasks. We have defined a derived schedule class in order to realize improved domain heuristics for scheduling jobs. All together we estimate that only about 10% new code has been developed for the application. Of course, during the development of the framework we had the application in mind. A second application for a different steel plant (VAS) has been used as a further test-bed that shows that about the same effort is required here. A simplified application, the SteelDemo application which can be also obtained from our web server needs only 5 extra classes because of its simplification.

At the moment a totally different application is developed. This scheduler has to schedule operations on a distributed computer hardware. Although, we identify again objects such as orders, jobs, and resources, an own subdomain for computer scheduling has to be defined. And there are several other application types such as job-shop scheduling or roster scheduling that are not yet sufficient supported by our framework. Therefore, it is very likely that changes also in the architecture may become necessary in the future.

### 21.6 Future Extensions

To improve the reusability of DÉJÀ VU for new applications it seems important to define an order description language from which the system can generate automatically the order class and the class that is used to display an order. Although the logic behind this class is simple the construction is error-prone.
The framework is available at the moment for Windows NT and MacOS. The transformation to other platforms should be relative simple because the differences of the operating systems are already isolated. Thus, in the near future a transformation to UNIX will be expected.

At the moment only a very simple temporal logic is used to describe how operations of one job have to be performed. Since we use only before and after-relations we cannot express any possible constellation of operations. The temporal consistency mechanism incorporated is based on this simple model. To use the full expressiveness of Allen's interval algebra [Allen 1983] for the consistency mechanism would be computational too expensive (NP-complete), but it seems possible to use it to describe 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 as well as for most industrial problems is the daily work with the adaptation of the schedule to react on unexpected events like new rush orders, machine break-downs, destroyed products, or others. Based on [Dorn, Kerr, and Thalhammer 1995] we have already built a reactive scheduling prototype for Böhler which shall be integrated now in DÉJÀ VU. Since this prototype has worked in a simulation model we must now test it in a real domain.

21.7 References


A documentation of the scheduling class library as well as further infos and the SteelDemo application is publicly available at: http://www.dbai.tuwien.ac.at/proj/DejaVu/Docu.htm