CHAPTER 4

Case-based reactive scheduling

Jürgen Dorn
Christian Doppler Laboratory for Expert Systems,
Vienna University of Technology,
Paniglgasse 16, A-1040 Vienna, Austria

Abstract

Scheduling of operations in a production process is usually seen as a combinatorial search for a feasible and perhaps optimal plan. In actual industrial practice however, constraints and optimization criteria can often not be given exactly because they are unknown or vague. Therefore a combinatorial approach often does not meet the actual requirements. Furthermore, the representation of all potential alternatives of the production process would lead to a combinatorial explosion that cannot be solved in a reasonable time frame.

In this paper case-based reasoning is applied to reactive scheduling to model the experience of skilled human operators in reacting to unforeseen events in the production process. Old solutions stored in a case base are used to solve new problems. Case-based reasoning is supported by fuzzy reasoning on a constraint-based representation to find and adapt old cases and to evaluate them. An iterative improvement method is further used to optimize solutions found by case-based reasoning. The approach is illustrated by means of an application in the steel industry.

Keyword Codes: I.2.1, I.2.6, I.2.8
Keywords: Applications and Expert Systems; Learning; Control Methods and Search

1. INTRODUCTION

Since the early fifties attempts have been made to support the scheduling activity in factories using computer systems. The advantage of such systems is the power to simulation production, to recognize bottlenecks and to search exhaustively for optimal solutions. However, in recent years it has become evident that *predictive scheduling* reaches its limit if it is applied to real factories. There are so many possibilities and uncertainties that cannot be considered during scheduling that it becomes very likely that a new schedule will only remain valid for a short period.

As a consequence, sometimes it is said that instead predictive scheduling some kind of intelligent *dispatching* is sufficient. The idea is to make no schedule and to allocate new jobs only to a machine when it becomes available. Several heuristic rules of how to dispatch such operations have been proposed (Pan-

R. Kerr et al. (eds.), *Artificial Intelligence in Reactive Scheduling*

walkar and Iskander 1977) or (Haupt 1989). These rules can be improved by some look-a-head as was shown by Kanet and Zhou (1992). However in both approaches, no dynamic bottlenecks are recognized and thus for highly specialized production environments it will be very likely that some jobs will block the treatment of other jobs. To recognize bottlenecks early, some kind of prediction seems to be necessary. However the accuracy of such a simulation should not be stressed too much.

Moreover, these heuristics consider only temporal constraints and objectives. If other constraints exist as for example in our application in the steel making industry, very specialized rules must be developed. The problem with such specialized rules is that they will remain valid only a limited period. If some aspects of production change (new products or production technologies), new strategies and rules become necessary and the scheduling system needs to be adjusted. Thus a predictive scheduling system has a higher generality than a simple dispatching system.

Reactive scheduling is the process of revising a given schedule due to unexpected events on the shop floor. This process is often seen as a problem of rescheduling a set of jobs under modified constraints. In this case, the problem can be solved with similar techniques to the original problem – a combinatorial search for a solution. Two possibilities are described in the literature:

- one can resolve the problem again from scratch or
- one tries to adapt the old schedule to the new situation.

For the first approach any scheduling algorithm can be used. For the second approach either a specialized repair mechanism must be implemented or iterative improvement methods are used for the first construction and the repair as for example in (Dorn *et al.* 93). The first approach (to reschedule from scratch) is often rejected in industrial practice because a new plan can differ considerably from the old one. This is not desirable since many other decisions like assignments of personnel to duties, delivery of raw materials, and further treatment in subsequent shops will be influenced by this "shop floor nervousness". Further, it is common practice of human schedulers to patch schedules that have become invalid.

Le Pape (1992) and Smith (1994) propose further a distribution of the knowledge for predictive scheduling and intelligent dispatching in separate agents. The advantage of such a dispatching agent is that it can always react very quickly to events without changing the whole schedule. However, in this architecture it is open as to how the scheduling agent reacts to modifications. If for example a machine breaks this has consequences for the whole schedule and perhaps new strategies may become necessary too as we will see later in an example.

Usually, reactive scheduling is illustrated by means of deviations of processing times of operations. A typical scenario goes as follows: One operation requires a longer processing time then expected. The starting time of a later job must therefore be shifted and this causes a violation of a due date. Another very simple scenario is that one machine breaks down and the operations allocated to this

machine must be performed on other machines which causes other jobs allocated to the alternative machines to become tardy. These kind of problems can be easily formulated and solved in the original problem space.

However, what is not seen in these approaches is that the revisions are often caused by faults in production that cannot be handled so easily. For example, if a product with high priority in an application is manufactured badly, one has to decide on a higher level of reasoning what to do. If the product is part of a series the treatment will be different then for a part with very distinguished features. Moreover, human schedulers often use heuristics that vary in a situation dependent way.

The argument is that for reactive scheduling it is important to know more about the manufacturing technology to decide whether certain constraints may be violated. Reasoning about times and resources and finding a solution that has the fewest constraint violations is insufficient. Case-based scheduling can be used to react to such manufacturing challenges. Techniques developed so far are still important in the proposed methodology. For example, constraint-based reasoning can be applied to verify whether solutions proposed by a case-based reasoner are valid. Furthermore, case-based reasoning is not the best way to optimizing a schedule. The proposed approach solves the reactive scheduling problem by first using cases, i.e. old experiences to find a solution and then an iterative improvement method like those described in (Dorn *et al.* 1993) to find a good schedule.

In the following the steel making shop LD3 of the VOEST Alpine Stahl in Linz is described as an example for a highly specialized production process. We recapitulate the proposed methodology based on a representation with fuzzy constraints and an iterative improvement method. Then we present some examples of cases that can used to find solutions for new problems that occur and how case-based reasoning is applied to solve new problems by adapting old cases.

2. DESCRIPTION OF THE APPLICATION

Steel production is a multi-stage process with different groups of processes involving different plants. Raw materials are first melted together in a blast furnace to produce pig iron. Liquid pig iron, tapped from the blast furnace, is transported to the steel making shop. Here, converters are used to refine the iron into steel of the desired composition by blowing oxygen into the hot metal to oxidize the impurities. Ladle refining further eliminates impurities and adds alloy ingredients to make steel of a certain grade. The molten steel is then transported to a continuous caster to form slabs. Ladles pour the molten steel into a reservoir called a *tundish* serving as a buffer between the ladle and the casting machine, and allowing uninterrupted casting from several ladles. Depending of the steel quality, slabs of steel can go directly from the caster to the hot rolling mill. If they are not delivered immediately to the mill they can be stored in a slab stock, but must be reheated before rolling at a later time. Some steel qualities however, must cool down before rolling. Some of the rolled material goes then to

the cold strip mill for cold reduction. Rolled strips of steel are finally transported to the finishing section where they may undergo a variety of operations such as pickling, tempering, galvanizing, heat treating, coating, etc.

Due to complexity issues the organization of the whole production process is separated in autonomous plants like the blast furnace, the steel making shop, and the rolling mills. For each of these units individual scheduling takes place. However, a scheduler for the steel making shop must also consider constraints of other shops.

Fig. 1. describes the organization of the steel making shop and its interfaces to other shops in more detail. The blast furnace (BF) delivers the pig iron either to the desulfurization unit (DS), to the mixer (MX), or directly to one of the converters (CV7, CV8, CV9). The mixer is a huge container with a capacity of 2000 t that is used as a stock between blast furnace and converters. Usually only two of the three converters are used (CV7, CV9). There are two typical production routes through the plant. The steel from CV7 is poured into a ladle and delivered to the ladle furnace (LF) and later to the single-stranded continuous caster CC3. The other route is from CV9 to the conditioning stand (CS) and to the two-stranded caster CC4. On both routes also the vacuum-degassing unit (VAC) can be involved. About 90% of the cast slabs are delivered either hot to the hot rolling mill (HRM) or stored in the slab stock (SS). The remaining jobs are produced for the forge and the foundry.

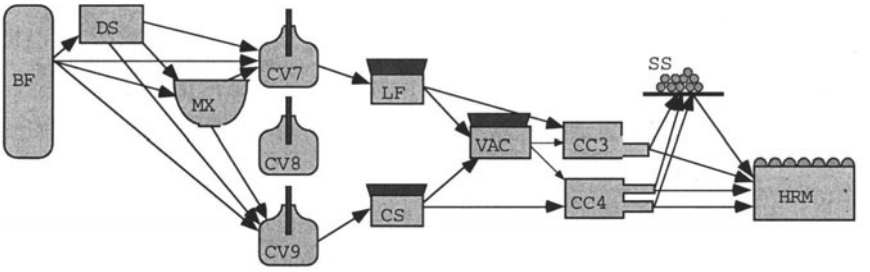

Figure 1. Organization of the LD3-plant

2.1. Constraints in the steel making shop

The main scheduling task is the sequencing of given jobs on the two production lines. Unfortunately both lines are not independent, because the vacuum treatment can be necessary in both production routes. Additionally, in the case of break downs or other failures in production the routes can be modified. Several constraints have to be considered when a new schedule is constructed or when an unexpected event occurs.

The tundish has to be maintained after approximately 240 minutes. A second tundish is used, because the maintenance operation takes about 100 minutes. Before one tundish is worn out the tundishes are exchanged to allow a continu-

ous operation of the caster. Experience show that at most about eight jobs can be cast on CC4 without change of tundish. The number is smaller for CC3.

The main criteria that constrains the sequence of jobs are compatibility constraints between jobs. There are three compatibility aspects that must be considered. The steel grade (the chemical analysis) of subsequent charges must be similar, the casting format may vary only in certain ranges and the degassing procedure in the secondary metallurgy must be compatible. A heuristic says that the format should be alternatively increasing and decreasing slowly whereby some orders have restrictions that only increasing or decreasing is possible.

Rules about which steel grades may be cast one after the other are given explicitly. Usually, there are hard constraints saying that two grades may not be sequenced, but there are also soft rules that say that it is not so good to sequence two grades. If no feasible sequence can be found, three operations may help:

- To separate two incompatible jobs, a plate may be inserted into the strand. This results in the production of a useless slab. These costs are quantifiable. However, this operation has also some compatibility constraints. If the difference between the subsequent steel grades is too great, no *quality separation* is possible.
- Also a *change of tundishes* may help. Such a change has higher costs than a quality separation. However, if the change must be performed anyway due to the required maintenance operation, the costs are less.
- The most expensive operation is a *set-up* of the caster. This means some hours of interruption without casting. Set-ups are also required when an interruption in the delivery occurs or the duration between two changes of tundishes is less than 100 minutes.

The three operations ensure that every sequence of jobs can be transformed into a feasible schedule. However, to maximize production the casters should process the steel continuously without interruptions. So one of the objectives is to have as few set-ups, change of tundishes, and quality separations as possible.

Besides the problem that the vacuum-degassing unit is used in both production lines it must be maintained after about four jobs which takes several hours. Usually, there are not so many jobs that have to be processed on this unit. Therefore, this constraint must be considered but it is not a hard problem.

In the hot rolling mill another scheduler searches for good sequences which are often different to the best sequences in the steel making shop since other sequencing constraints exist. If slabs are delivered hot directly from the casters to the mill no stock is required and slabs must not be reheated. Since this means a shorter flow time of jobs and less energy consumption, direct charging is optimal for overall production. However, due to the different sequencing constraints this is not possible. For about 20–30% of jobs due dates and fixed sequences are posted by the hot rolling mill. These temporal constraints should also be considered by the steel making shop scheduler.

2.2. Reaction strategies

The main optimizing criterion in the LD3 plant is throughput. Therefore the casters should produce without any set-ups. Due to compatibility problems this is not always possible. A possible heuristic in this case is to set priority to the caster CC4 because it has a higher throughput. Thus the aim is to produce with the two-stranded caster as long sequences as possible. Jobs that require set-ups due to chemical problems are scheduled on the single-stranded caster, because its capacity is smaller than that of the two-stranded caster.

The supply of pig iron from the blast furnace is not always in time. If the supply of pig iron is low it will be likely that one or both casters must stop eventually. In this case the objective to produce long sequences without set-ups can be relaxed. If not enough pig iron exists to supply the two-stranded caster continuously but there is enough for the single stranded caster, it makes sense to assign the difficult jobs now to the two-stranded caster. More generally one may say, that in the case of the interruption of the casting sequence on one caster, it becomes more important that the other caster does not stop.

Sometimes the produced steel has not that quality that is demanded. This means that the job must be scheduled again. However, if the damaged quality can be used to make another steel quality that is also ordered, jobs may be exchanged. Of course this conversion has compatibility constraints. Further, the scheduler may look at a larger pool of orders to see whether the damaged steel grade can be used for some order that is not yet scheduled.

If the actual processing time of some jobs between two tundish changes are shorter than predicted another job could be processed before the next change. Usually such an optimization is not done because of changes in subsequent processes. However, if some very important new job or a damaged job must be rescheduled, this possibility will be chosen.

The capacity of the casters is so small that it is sufficient to operate only two converters. Usually only the converters CV7 and CV8 are used because converters must be maintained and a maintenance of three converters is a greater effort than the maintenance of two. However, in the case of a break down of a converter the third may be used.

3. THE COMBINATORIAL SOLUTION

Our first approach to solve the scheduling problem in the LD3 plant is a constraint-based representation with tabu search (Dorn *et al.* 1993). We represent the scheduling domain by objects like machines, orders, jobs, operations, and schedules. The violation of constraints is represented by fuzzy sets and a given schedule is evaluated by a weighted aggregation of all constraint violations. By repairing iteratively the greatest constraint violation an initial schedule is improved until some threshold value is reached. Tabu search controls this improvement process. We explain this methodology briefly because it will be used later also for several steps in the case-based approach.

3.1. Fuzzy constraints and the evaluation of schedules

The satisfaction of constraints is represented by fuzzy sets. For example, the satisfaction of a due date can be mapped on the linguistic fuzzy values *very early*, *early*, *in time*, *late*, or *very late*.

We distinguish between hard and soft constraints in the application. Due dates are soft constraints and the maximal lifetime of a tundish (240 minutes) is a hard constraint. The duration of the casting of one job depends on the size of a heat and the casting velocity. Since these parameters cannot be determined exactly we use fuzzy numbers to represent the duration. To decide whether a tundish change must be inserted at a certain place in the sequence of jobs on a caster, the fuzzy durations of all jobs since the last change are summed. This estimation of the duration of the casting of a group of jobs till the next tundish change is mapped on linguistic values like *very short*, *short*, *medium*, *long*, *very long*, and *too long* to represent how likely it is that the jobs can be cast without tundish change. The last symbol represents a hard constraint violation and is not allowed. Both constraints are represented by fuzzy sets and we construct only schedules without violations of hard constraints. Soft constraint violations are subject to optimization and their satisfaction is considered in the evaluation function.

Due date constraints and an objective like minimizing the number of set-ups have of course different degrees of importance. In the application due dates are not so important. Although the number of set-ups is never a hard constraint violation, minimization of set-ups is more important than due date violations. Therefore different kinds of constraint violations get different weights. Although compatibility constraints are the strictest constraints in the application, their consistency is achieved through insertion of a quality separation, a tundish change, or a set-up. The following table shows the considered constraints and their weights.

Table 1. Considered constraints and their weights

c_i	type	weight
c_1	minimization of set-ups	1.0
c_2	minimization of number of tundish changes	0.9
c_3	minimization of number of quality separation	0.8
c_4	maximization of the length of tundishes	0.7
c_5	sum of all format changes	0.7
c_6	due dates	0.6
c_7	compatibility	0.6
c_8	negative format changes	0.2

Usually there are more jobs to be scheduled than actually can be scheduled. Therefore the most important jobs should be scheduled first. To achieve a balance between important jobs and the satisfaction of constraints we also model the importance of jobs by fuzzy sets. Moreover, to control the scheduling process,

we assign criticality values to jobs that seem to be difficult to schedule. The construction of a first schedule is then controlled by these criticality values.

The evaluation function of a schedule is the mean of the weighted satisfaction degree of all constraints plus the weighted sum of the criticality values. The value of this evaluation function is always a fuzzy value between 0 and 1. The optimal value is theoretically 1, but when different steel grades are involved, this value can never be reached. Good evaluations for the data that we have used in experiments range between .91 and .93 (Dorn *et al.* 1994).

3.2. Repair-based improvement of schedules

Iterative improvement methods (Dorn 1994) are a class of problem solving methods that suit best for complex optimization problems and can also be applied for simple reactive scheduling problems. One subclass of iterative improvement methods is tabu search (Glover 1989, 1990).

An iterative improvement method is a problem solving method that starts with an initial schedule that may be created randomly and thus still violates some constraints. Through local modifications it is tried to improve the schedule until some good schedule is found with a satisfactory evaluation. These modifications can be an exchange of jobs in the schedule, a shift of jobs or an exchange of groups of jobs. Especially, if the exchange of groups is considered, there will be too many such operators applicable to examine all possible neighbors. As a consequence, in most other approaches with tabu search only the exchange of single jobs is considered. However, good results in this application have been obtained with the movement of groups of jobs. This is tractable in our case because we apply the repair-heuristic to guide our search.

The repair-heuristic is to try always to repair the greatest constraint violation. Since we achieve feasible schedules by introducing operations like quality separation, tundish change, or set-up, no hard constraint violations must be repaired. Thus first repair steps to minimize the number of set-ups will be tried by the algorithm.

With tabu search (Glover 1989, 1990), the search for a near-optimal schedule is controlled. From a set of considered modifications that modification is taken that results in a schedule with the best evaluation. For this schedule again, a set of modifications are examined and the best neighbor is chosen. For problems having many good solutions that are not immediate neighbors it is very likely that such an iterative improvement method will become stuck in a local optimum. Therefore, during search schedules with a worse evaluation are also accepted as new candidates. Tabu search allows the construction of worse schedules, if all better schedules that can be reached are *tabu*. A schedule is made tabu when it becomes a candidate, to avoid cycles that would be possible otherwise. The tabu schedules are stored in a ring buffer of length n, thus allowing cycles of a length greater than n. Experiments have shown that tabu search finds good solutions for our application in about 30 seconds on a 386 PC (Dorn *et al.* 1994).

As was shown in (Dorn *et al.* 1993) tabu search combined with the repair heuristic can be applied easily to simple reactive scheduling problems. If the

processing time of an operation takes longer than expected, the due date of some job may be violated more strongly than before. If this violation exceeds some threshold value, the repair-based method is applied again to repair this violation. Since the search is focused on this violation we get repaired schedules that are only slightly modified.

This approach performs so well, because we have reduced the complexity by using some heuristic decisions. We have not represented the third converter CV8 because it is usually not used. Of course we could easily represent it in our approach but then we have to model also further constraints that result in a worsen evaluation if all three converters are used. If all other possibilities in the production would be considered in such a combinatorial framework the performance would significantly degrade.

4. CASE-BASED REASONING

Many researchers propose heuristics to reduce the inherent complexity of scheduling problems. These are mainly rules of thumb that can be represented easily in the form of production rules. However, the problem with these heuristics is that usually no generally accepted domain theory or formal model exists. Human experts learn them by practicing and often they are not aware of any formal model of their knowledge. In expert system projects it is the most difficult problem to elicit this knowledge. Additionally, the knowledge elicitation process will never be complete and therefore expert systems will react anomanously on the boundaries of their knowledge. Maintenance of these rule-based systems is still a considerable problem.

Deep modeling of domains is proposed elsewhere to overcome these problems. The knowledge in these models is generic and in principle every task may be solved using a causal domain theory. With object-oriented representation the modeling of jobs, resources, and their characteristics is supported and constraint-based reasoning can be used to represent and process constraints between jobs and resources. These techniques are good in describing physical knowledge about an application and the restrictions constraining a solution. Due to its pure descriptive nature a deep model is easy to maintain. However, the complexity of the search for a schedule is intractable for realistic applications. Heuristics are still necessary to guide the search to a solution.

Case-Based Reasoning (Riesbeck and Schank 1989, Kolodner 1993) combines advantages from heuristic and deep modeling. Usually, a case contains an implicit heuristic for a given problem. Further it promises a solution for the knowledge elicitation and maintenance problems, because it adapts old cases to solve actual problems. Cases contain at least a description of a problem and its solution. One of the main issues in case-based reasoning is the possibility to use analogous cases to find a solution for a problem that was never solved before by the system. In order to achieve such a functionality, old cases are stored in a structured way in a case base. Cases are attributed with their characteristics and links between cases are established to represent similarities between them. To

solve a problem, the case-based system looks for the most promising old case. Usually, some features of this case must be adapted to the actual problem. Then the system tries to solve the problem with this adapted case. In contrast to approaches with deep modeling the evaluation component must not search exhaustively for a solution, but applies the causal domain theory only to perform a simulation. In this simulation the new schedule is evaluated and if this evaluation is poor, the system explains the failure and repairs the plan by performing in the worst case an exhaustive search in the causal model. The repaired plan and the explanation of the failure are stored in the case base. The latter is used to anticipate failures in the future.

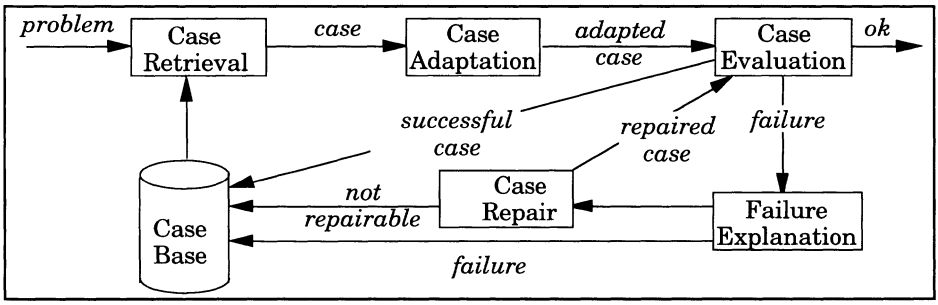

Figure 2. The general structure of a case-based system

4.1. Cases in reactive scheduling

Typically, the first cases that will be introduced in the case base will be obtained from domain experts. An expert will usually explain his reaction to certain events with very few concepts. More concepts may be necessary to make the decisions, but if he explains a reaction to a novice many details will be omitted due to four reasons:

- Many decision basics are in the background of his reasoning process and he is not aware of it. This is a crucial problem in traditional knowledge elicitation.
- Many assumptions of the experts are defaults and he thinks that they are accepted by everyone and it is not necessary to talk about them.
- On the other side the human expert sometimes simplifies the problem because he thinks that the knowledge engineer as a novice does not understand the problem.
- Finally, the expert has no complete model of the dependencies of his knowledge.

These points require that for the development of traditional expert systems knowledge elicitation phases are repeated to refine the knowledge. With a case-based system we start with simple cases that will be refined later. Thus, in this

paper some cases with very few attributes are first defined. Later the case-based system will find problems where further concepts are introduced. Failure to solve a problem will generate the need for a more detailed description and the case base will be refined by a more fine grained cases.

To support this kind of learning the domain theory must be very expressive, since the case-based system will not learn new concepts on its own but new problem solutions. Only a human user can introduce new concepts (as for example a new constraint type) into the domain theory. The domain theory must already contain all existing machines with their attributes, the operations that can be performed on the machines, and the relations between machines. Thus for the steel making shop the fact that all converters are of the same type must be represented. A steel grade, a job and similar other concepts must also be defined. A further important part of the domain theory are constraints on the use of objects. These constraints can be fuzzy, but it is also very important to give sharp boundaries of certain possibilities.

To decide now what to do in case of an unexpected event, we must store three features of an old problem solution in a case:

- a characterization of the problem that was solved,
- a description of the relevant situation on the shop floor and the characteristics of the schedule, and
- the problem solution that was used to solve the problem in this situation.

The first naive approach to design cases for scheduling decisions would include each scheduled job. However this is too expensive and furthermore the retrieval of similar cases would be more difficult. We store instead those characterizations that also decide the value of schedules. For example, if there are many set-ups on the two-stranded caster, the schedule will get a bad evaluation. This number of set-ups is relevant for a problem solution, but usually the sequence of individual jobs are not important. Further the status of relevant objects (such as the fact that the converter is broken down) is stored.

One of the main features must be a characterization of the actual problem. This can be a delay of an operation, a new job to be scheduled, a break down of a machine or more specific events in the steel making application such as the shortage of pig iron supply, a break of the cast strand, or a wrong chemical analysis of a cast slab. These problem characterizations will be again modeled by fuzzy sets since an event such as the shortage of pig iron is not crisply defined. Furthermore, the duration of a problematic event like the break down of a machine could be given as a fuzzy number. For the pig iron supply also some estimation of when new iron will be supplied should be stored.

For the problem solution we need further a characterization of the actual situation on the shop floor and a characterization of the jobs that are scheduled. We base the characterization on the represented objects and constraints. The fuzzy attributes are the main features that are compared when searching through old cases for a similar case. Whereas crisp features of a case must match the fuzzy attributes enable an explicit representation of similarity.

The problem solution will be in the most simplest case a direct manipulation of a schedule as will be pointed out in the first described case. More intelligent manipulations change the weight of a constraint type or relax certain constraints.

We describe now four very simple cases that exist in our exemplary case base. The first case describes how the problem of a break down of converter CV7 was solved. The problem was solved very easily by exchanging the broken converter CV7 with the converter CV8 in the schedule. Of course an evaluation is made to find any inconsistencies. Theoretically, it could be that the capacity of CV8 is not sufficient. The domain theory must be able to check such constraints.

case c1:	
problem:	break down of **CV7** for *some* hours
situation:	*many* **jobs** on **CV7**,
	free **CV8**
solution:	exchange **CV7** with **CV8** in **schedule**

The second problem is more crucial and the actual situation is described in more detail. Different reactions to a shortage of the pig iron are possible. In the old case it was decided to find a better sequence of jobs on the single-stranded caster and give the two-stranded caster a lower priority. The problem solution proposed does not immediately effect the schedule, but by changing the weights of constraints and the priorities of the casters the optimizing process results in another schedule.

case c2:	
problem:	*very few* **pig iron**
situation:	*medium* **set-ups** on **CC3**,
	few **set-ups** on **CC4**,
	many **quality separations** on **CC3**,
	medium ***quality*** **separations** on **CC4**,
	few **tundish changes** on **CC3**,
	medium **tundish changes** on **CC4**,
	some important **jobs** on **CC4**
solution:	increase weight set-ups **CC3**,
	increase weight important jobs,
	strong decrease weight set-ups **CC4**,
	optimize schedule

The following case handles the problem of a badly produced job. A very specialized job with high quality constraints (453-89-13) violates these constraints. A subsequent job on the same converter has the same contents but the quality restrictions are not so strong. The first job is a specialization of the second one. Thus the conversion of both jobs makes no problem. Of course some constraints

must be checked by the case evaluation component (for example the compatibility constraints to neighbor jobs and also the casting format).

case c3:	
problem:	*bad* **quality** of **job** 453-89-13 on **CV7**
situation:	**job** 672-89-1 after **job** 453-89-13
solution:	convert **job** 453-89-13 to **job** 672-89-1
	convert **job** 672-89-1 to **job** 453-89-13

The fourth case describes how a new important job was introduced and exchanged with a not so important job.

case c4:	
problem:	new *important* job 884-89-27
situation:	**job** 424-89-29 on **CC3**,
	similar **job** 884-89-27 and **job** 424-89-29
	not important **job** 424-89-29
solution:	exchange **job** 884-89-27 with **job** 424-89-29 in **schedule**
	evaluate **schedule**

4.2. Case retrieval and the case base structure

The retrieval of cases is one of the main steps in case-based reasoning. The goal is to find a case that matches best the actual situation and solves the new problem. Schedules are usually very complex and to model all details of the schedule or the production environment in a case is not appropriate. Instead some characterizing surface knowledge is represented in a case. This abstraction of the proper problem also supports the application of the case for a new problem. For retrieval of old cases it is critical to define some measure of similarity between cases and the actual problem. Some features should be equal and others can be slightly different. With fuzzy sets we already use a representational mean to express such a similarity.

Assume the system is confronted with the search for a new reaction. The actual problem description contains a set of events E and a set of propositions P describing the actual situation. The system shall now decide how good a case C_i of the case base fits to the actual problem. The case C_i contains the set of events E_i and the set of propositions P_i. For the events we distinguish three subsets: those that are contained in both sets ($E \cap E_i$) and those that are only in one of both sets. The most important factor for the definition of the similarity seems to be that the case handles the same events. If there are events in the actual problem that are not handled by the case ($E - E_i \neq \emptyset$) the case does not fit so well. This will be reflected in the similarity function defined below by a constant χ. If there are events defined in the old case that are missing in the actual problem ($E_i - E \neq \emptyset$) this has a lower priority for the matching as will be reflected by constant β.

The cardinality of P will be always greater than P_i since we can always determine a feature of the actual situation. If a feature p_j is in the case C_i then the

function $\delta(f_j, C_i)$gives us the strength of this feature. For a crisp proposition as „free CV8" this will take a value of 1 and in other cases a fuzzy value. We compare the instantiation of two features by a fuzzy subtraction (Θ) and aggregate all differences. We define the similarity of case C_i as:

$$S(C_i) = \alpha \sum_{\forall e_j \in (E \cap E_i)} \left(1 - \left| \delta(e_j, E_i) \ \Theta \ \delta(e_j, E) \right|\right) \oplus \beta \sum_{\forall e_j \in (E_i - E)} \left(1 - \delta(e_j, E)\right) \oplus$$

$$\chi \sum_{\forall e_j \in (E - E_i)} \left(1 - \delta(e_j, E_i)\right) \oplus \gamma \sum_{\forall p_j \in P_i} \left(1 - \left| \delta(p_j, P_i) \ \Theta \ \delta(p_j, P) \right|\right)$$

Pragmatically, we choose the constants as follows: $\alpha = 1$, $\beta = -0.5$, $\chi = 0.5$ and $\gamma = -1$.

The structure of a case base is critical for the retrieval of cases. First, we can speed up the retrieval by imposing a structure that directs the retrieval component to the most similar case. Second, there may be different cases that have the same similarity measure, but one case is the more problem relevant. This could of course also be solved by modifying the weights for different features, but the structure method seems to be clearer. Finally the structure should support the generation of groups of cases. Such groups can be used to generate alternatives if a case adaptation fails.

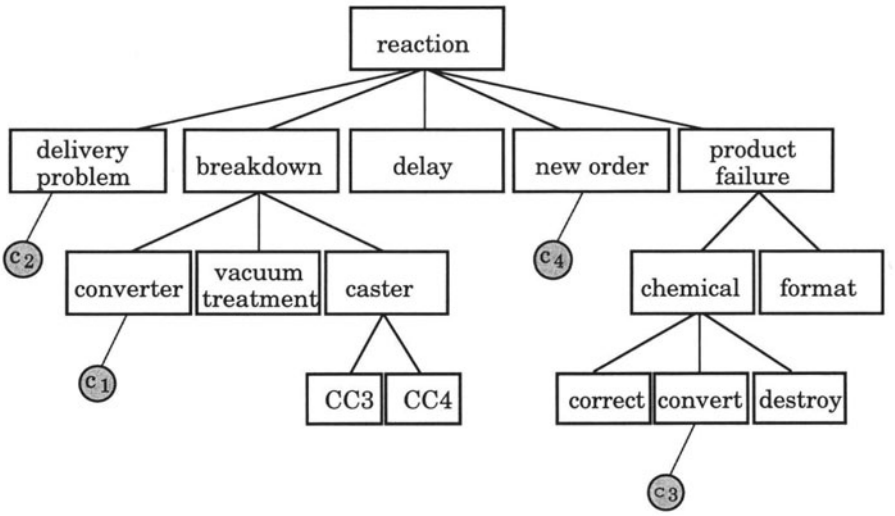

Figure 3. Hierarchical structure of the case base

We store cases in a hierarchical structure with the general "reaction" in the root. In Fig. 3 we distinguish five problems that must be handled. These represent classes of cases and they contain no knowledge of real cases. They are only used to find an appropriate case. If a new problem is classified as a product failure the structure leads the retrieval component to the class of product failures. Actual cases depicted as small circles in the figure must be attached to a certain class. If a new case is to be stored that cannot be classified in terms of one of the defined reaction classes, the case is attached to the general reaction class.

4.3. Case adaptation

The usual way to adapt a case is based on defined object classes. For example, we have defined the class "converter" as a subclass of "steelmaking aggregate" which is itself a specialization of a machine. If we have a problem concerning one certain aggregate and the retrieval component finds an old case with a different aggregate, the old case is adapted and instead of the old aggregate the new one is inserted.

A very easy adaptation will be the process of using case c1 to solve the problem that converter CV9 breaks. Since both aggregates are of the same type it will be very likely that the adaptation is sufficient to solve the problem. As was pointed out before an attribute like the capacity of the converter could be a reason as to why an adaptation fails.

case c1-1:	
problem:	break down of **CV9** for *some* hours
situation:	*many* **jobs** on **CV9**,
	free **CV8**
solution:	exchange **CV9** and **CV8** in **schedule**

Since the adapted case is very similar to the old case and its reconstruction from the old one is very easy, an adapted case is not stored in the case base.

4.4. Case repair

The case adaptation means that an instantiation of a "variable" is changed. This of course has effects on the proposed solution if the variable is also used in the description of the old solution. This adaptation may fail but there is also the possibility that the problem solution fails. Whereas an adaptation failure is recognized by local constraints the failing of a problem solution is detected by the evaluation function. If a hard constraint cannot be repaired or the evaluation function is very low and the optimizing method does not find an improvement, the solution proposed by the case-based system has failed.

A repair of this case consists in the refinement of the situation and an improved solution. Consider the case where another job is of bad quality. The case-based system finds case c3 as most promising one and adapts it with the actual job and its successor. The evaluation now delivers a bad value because the successor has a different steel grade and so the compatibility is bad. The insertion of a quality separation between both jobs drops the evaluation significantly.

Two possibilities exist to solve this problem. The system can either re-optimize the schedule or search for another job with a similar steel grade. The first alternative is easier to implement but may lead to a greater disturbance of the schedule. Therefore the second alternative is better. However, to avoid making the same repair later again, the system should adjust the situation in the old case and in the new repaired case. In case c3 an insertion of the constraint "job 672-89-1 similar job 453-89-13" helps to avoid taking this case if the successor cannot be converted. Case c5 will also be stored in the case base. This new case makes no assumptions about the situation which mans it is a more general case than c3. With the similarity function achieved, this case is only retrieved if the more specialized case cannot be adapted.

case c5:	
problem:	*bad* **quality** of **job** 864-17-22 on **CV9**
situation:	
solution:	find convertable_job J
	convert **job** 864-17-22 to **job** J
	convert **job** J to **job** 864-17-22
	evaluate **schedule**

5. CONCLUSIONS

In an experimental setting based on a real application we have explained a new approach to reactive scheduling. The approach is characterized by its openness since new reactions can easily be integrated. In the simplest case such new reactions can be learned from a human expert. Miyashita and Sycara (1994) store decisions of human experts as cases and reuse these cases if the same problem occurs again. However, they only consider temporal constraints and objectives in their approach. Although they report on improvement, we think that our approach is capable of solving a wider range of problems. Since we combine case-based reasoning with an iterative improvement method we are also able to produce new solutions if no old case is found that would help in solving the problem. This new solution and the path to the solution may now be stored in a new case so that later exhaustive search will be not needed.

Learned methods are the exchange of resources or the modification of constraint weights. Further we can imagine to learn a situation in which it is possible to choose the best optimization algorithm. Experiments have not so far extended to the point that we can identify clearly what algorithm should be taken for which kind of scheduling problem. We have examined Tabu Search, Iterative Deepening, Random Search, and Genetic Algorithms (Dorn *et al.* 1994). Simulated Annealing is another algorithm that looks very promising for this task. Prosser *et al.* (1994) have differentiated strongly constrained problems from not so strongly constrained. It seems that some easy problems can be solved faster with a simple technique like Simulated Annealing and other problems require a more sophisticated algorithm. A strategy that could be learned in our

framework is to recognize bottlenecks. In actual practice the casters are the bottlenecks and the scheduler tries to find first a sequence on the caster. If two converters would break down, the remaining converter would be the bottleneck and sequencing on the converter would be preferable.

In some applications reactions in time are very critical. Reactive schedulers should therefore be able to plan reactions in some predefined time limit. One could distribute the responsiveness to different agents as implemented by Le Pape (1992). Another attempt that we will examine is to reason with sequences of cases. The basic idea behind this approach is that problems occur usually not isolated. With the occurrence of one case we can predict with some probability a new problem that will arise. If we are aware of this possibility we can of course react faster.

We have only thus far built a small prototype to verify our ideas. Unfortunately the real effectiveness cannot be shown by means of such an academic prototype. The crucial thing will be the anticipation of new problems that cannot be foreseen. Situations and problems that we have encountered were already known and the whole structure was designed to solve these problems. Nevertheless we feel that the proposed system is more flexible than other existing in handling unanticipated problems.

6. REFERENCES

Dorn, J, Kerr, R. and Thalhammer, G. (1993) Reactive Scheduling in a Fuzzy-Temporal Framework, in R. M. Kerr and E. Szelke (eds.) *Knowledge-based Reactive Scheduling*, IFIP Transactions, North Holland, pp. 39–56.

Dorn, J. (1994) Iterative Improvement Methods for Knowledge-based Scheduling, Technical Report CD-TR 94/74.

Dorn, J., Girsch, M. Skele, G. and Slany, W. (1994) Comparison of Iterative Improvement Techniques for Schedule Optimization, *Proceedings of the 13th UK Planning Special Interest Group*, Glasgow (also available as Technical Report CD-TR 94-61).

Glover, F. (1989) Tabu Search–Part I, *ORSA Journal on Computing* **1** (3), pp. 190–206.

Glover, F. (1990) Tabu Search–Part II, *ORSA Journal on Computing* **2** (1), pp. 4–32.

Haupt, R. (1989) A Survey of Priority Rule-Based Scheduling, *OR Spektrum* **11**.

Kanet, J. J. and Zhou, Z. (1992) A Decision Theory Approach to Priority Dispatching for Job Shop Scheduling, *POM-1992 Conference* , Orlando Fla, Oct pp. 19–21.

Kolodner, J. (1993) *Case-Based Reasoning*, Morgan Kaufmann.

Miyashita, K. and Sycara, K. (1994) Adaptive Case-Based Control of Schedule Revision, in Zweben and Fox (eds) *Intelligent Scheduling*, Morgan Kaufmann, pp. 291–308.

Panwalkar, S. S. and Iskander, W. (1977) A Survey of Scheduling Rule, *Operations Research* **25** (1), pp. 45–61.

Prosser, P., Muller, C. and Brind, C. (1994) Personal communication on a project of the University of Strathclyde with British Telecom.

Smith, S.F. (1994) OPIS: A Methodology and Architecture for Reactive Scheduling, in Zweben and Fox (eds) *Intelligent Scheduling*, Morgan Kaufmann, pp. pp. 29–66.

APPENDIX

Formal specification of the Case Description Language in BNF

```
⟨case⟩            ::= case ⟨name⟩:
                      problem: ⟨events⟩
                      situation: ⟨propositions⟩
                      solution: ⟨actions⟩

⟨events⟩          ::= ⟨event⟩ ⟨duration⟩ |
                      ⟨event⟩ ⟨duration⟩, ⟨events⟩

⟨event⟩           ::= ⟨machine_state⟩ ⟨machine⟩ |
                      ⟨timeliness⟩ ⟨job⟩ |
                      ⟨quality⟩ of ⟨job⟩ |
                      new ⟨priority⟩ ⟨job⟩ |
                      ⟨amount⟩ ⟨material⟩

⟨propositions⟩    ::= ⟨proposition⟩ |
                      ⟨proposition⟩, ⟨propositions⟩

⟨proposition⟩     ::= ⟨event⟩ |
                      ⟨constraint⟩ |
                      ⟨constraint⟩ on ⟨machine⟩

⟨actions⟩         ::= ⟨action⟩ |
                      ⟨action⟩, ⟨actions⟩

⟨action⟩          ::= increase_weight ⟨constraint⟩ |
                      decrease_weight ⟨constraint⟩ |
                      exchange ⟨object⟩ with ⟨object⟩ in ⟨domain⟩ |
                      convert ⟨job⟩ to ⟨job⟩ in ⟨domain⟩ |
                      optimize schedule

⟨machine_state⟩   ::= ⟨occupation⟩,
                      break_down of,
                      maintainance on,
                      ⟨unknown⟩

⟨occupation⟩      ::= free,
                      ⟨fuzzy_number⟩ jobs,
                      ⟨fuzzy_number⟩ ⟨priority⟩ jobs,
```

```
⟨constraint⟩      ::= ⟨minim_number⟩ ⟨set_ups⟩,
                      ⟨minim_number⟩ ⟨tundish_changes⟩,
                      ⟨minim_number⟩ ⟨quality_separations⟩,
                      ⟨length⟩ ⟨tundish_length⟩,
                      ⟨minim_number⟩ ⟨format_changes⟩,
                      ⟨minim_number⟩ ⟨due_dates⟩,
                      ⟨quality⟩ ⟨compatibility⟩,
                      ⟨minim_number⟩ ⟨negative_format_changes⟩,
                      ⟨job⟩ after ⟨job⟩,
                      ⟨job⟩ similar ⟨job⟩
⟨duration⟩        ::= for ⟨fuzzy_number⟩ ⟨time_units⟩
⟨time_units⟩      ::= minutes |
                      hours

                      /*          fuzzy sets  */

⟨timeliness⟩      ::= very_early | early | in_time | late | very_late
⟨priority⟩        ::= very_important | important | normal | not_important
⟨quality⟩         ::= very_high | high | normal | bad | very_bad
⟨amount⟩          ::= much | sufficient | few | very_few
⟨fuzzy_number⟩    ::= many | some | medium | few
⟨minim_number⟩    ::= too_many | many | medium | few | very_few | zero
⟨length⟩          ::= very_short | short | medium | long | very_long |
                      too_long
```