Appeared in Marc Wallace, editor, *Proceedings of the Second International Conference on the Practical Applications of Constraint Technology (PACT'96): Scheduling, Resource Allocation, Planning, Configuration*, pages 211–220, London, UK, April 1996.

*FLIP++: a partial constraint satisfaction system for steelmaking scheduling

Ulrich Santa Wolfgang Slany*
Mazen Younes

Christian Doppler Laboratory for Expert Systems
E184/2, TU Wien, A-1040 Vienna, Austria, Europe
Phone: +43–1–58801–6123, Fax: +43–1–5055304
E-Mail: {santa|wsi|younes}@dbai.tuwien.ac.at
http://www.dbai.tuwien.ac.at/staff/slany.html#research

Abstract

Real world problems are often overconstrained, ill-defined, or tolerate good though suboptimal solutions as long as those are found before a deadline. Such problems are often called partial constraint satisfaction problems (PCSPs). The model for PCSPs was initially inspired by classic constraint satisfaction methods. Another path leading to comparable notions and originating from fuzzy decision making is introduced in the present paper, adding the following features to increase modeling capability: gradual satisfaction, priority, and compromises. Because of these additional degrees of freedom, fuzzy constraints must be fine-tuned for a certain task. However, such fine-tuning cannot be done for all possible instantiations of a problem, and thus is generally done on prototypical constraints. These templates are applied during run-time to a certain actual situation, i.e., dynamically adapted and applied only where necessary. This paper describes issues arising from the existence of such dynamic constraints and how they are handled in the *FLIP++ optimization libraries. Examples from a steelmaking scheduling application are given. Theoretical analysis and some initial experiments indicate that our method makes PCS ready to confront real world problems.

1 Introduction

In manufacturing industry such as steelmaking, the distinction between commercial viability and failure often lies in the ability to control the production process through efficient scheduling. Government as well as industry require practical approaches to a diverse set of complex scheduling and planning problems. While scheduling has been studied in isolation for many years, recent advances in artificial intelligence and operations research indicate a renewed interest in the area. In addition, the scheduling problem is being defined more generally, and work is beginning to consider the closed-loop use of scheduling systems in operational contexts. However, a primary source of difficulty in constructing good schedules stems from the conflicting nature of the objectives.

As with many real life decision-making situations, it is usually not possible to perfectly fulfill all objectives when building new schedules. This applies to class-room schedules, staff-roostering, as well as production schedules in manufacturing. Existing approaches to scheduling tend to reduce the complexity of the problem by considering only a *small subset* of objectives. In real world situations, it would often be more realistic to find *viable compromises* between the objectives. For many problems, it makes sense to partially satisfy objectives. The satisfaction degree can then be used to evaluate the achieved compromise. In addition, real objectives are often prioritized, therefore it is necessary to weight their satisfaction with importance factors. One especially straightforward way to achieve these two aspects of scheduling problems — to satisfy constraints to a certain degree, and to take into account relative importances — is the modeling of these constraints through fuzzy constraints. While fuzzy constraints are at least as powerful as other partial constraint satisfaction models [7], they are particularly well

*Please send all correspondence regarding this paper to Wolfgang Slany.

suited for modeling, since constraints can be written in a format easily understood by human experts, and because they feature a robust behavior which needs almost no tuning to yield reasonable control. In addition, the evaluation of their gradual satisfaction can be very efficiently used to guide heuristic search methods, in order to find approximate 'good' solutions while at the same time greatly simplifying the complexity of the scheduling problem. Real world descriptions naturally contain vaguely formulated relations, because more details are not known or would anyway not lead to better results as they would be canceled out through uncertain data. These uncertain data values can be well modeled through the use of possibility distributions, which are special interpretations of a part of fuzzy set theory.

1.1 An application example: Scheduling a steelmaking plant with *FLIP++

A detailed description of the steelmaking scheduling problem can only be found in German language (sorry) in Dorn et al. [4], the description of the employed search algorithms and their performance in Dorn et al. [3]. The main characteristics are shortly restated here to explain the scheduling context. Steel is produced on two continuous caster units, one being double stranded. Each day a cast sequence of about 35 charges (jobs) is scheduled for one caster over the next 24 hours. Hot pig iron is delivered from the blast furnace to the LD-converters. The steel is then poured into ladles, processed further in secondary metallurgy aggregates, and finally delivered to the casters. The casters produce continuously strands of steel that are cut into slabs of specific length. Although the steel is produced in different aggregates in a fixed flow, the casters are the bottleneck resource of the shop. The tundish, a part of the caster, has to be maintained after approximately 240 minutes. The maintenance takes approximately 100 minutes, therefore a second tundish is used while the first is maintained.

The main criteria that constrain the sequence of charges are compatibility constraints between jobs. Three compatibility aspects must be considered in order to optimize production quality:

- The steel grade (the chemical analysis) of subsequent charges must be similar.
- The casting format may vary only within certain limits.
- The degassing procedure in the secondary metallurgy aggregates must be compatible between subsequent charges.

The rules are explicitly given in a crisp form, but human operators have been observed to 'break' or relax these a-priori hard rules and to implicitly make trade-offs in 20% of their decisions. Their behavior can be adequately modeled by using fuzzy constraints.

To optimize the production, the casters should process the steel without interruption. Various operations can help when hard barriers otherwise would be broken. These include inserting a steel plate into the strand to separate the different qualities such that they do not mix too much, or to prematurely exchange the tundish in use with a second one. However, both operations have associated costs. If the tundish-exchange caused by quality considerations can be combined with a tundish-exchange because of tundish maintenance considerations, these costs can be minimized. The most expensive operation is to stop and then setup from scratch the whole casting unit. This takes a lot of time, but is sometimes necessary to do maintenance on the caster, or to produce certain difficult jobs.

There are soft temporal release-date and due-date constraints for certain jobs that should be delivered still hot to a subsequent plant. However, since there are (limited) warm-holding places available, the exact observance of temporal constraints is not obligatory. Additionally, the temporal constraints can be renegotiated with subsequent plants, and sometimes it is possible to exchange the temporal constraints of two jobs.

Generally speaking, it is possible to find a feasible schedule. However, criteria such as produced quality, tundish usage time, quality separating operations, number of setups and maintenance intervals of machines, release-dates, due-dates, and even such aspects as work-load of staff should all be optimized. Additionally, rescheduling has to be done quite frequently when some production parameters change due to machine breakdowns. In the past, most human errors were made in these rescheduling situations since time to think is scarce and the situation worsens rapidly (e.g., forgetting for some time a waiting aggregate, resulting in longer waiting times or worse qualities for certain jobs) if no action is taken.

In a first expert system approach, Stohl et al. [8] applied a constructive domain heuristic to the problem. Although the system found good feasible solutions, Stohl et al. believe that the solutions could

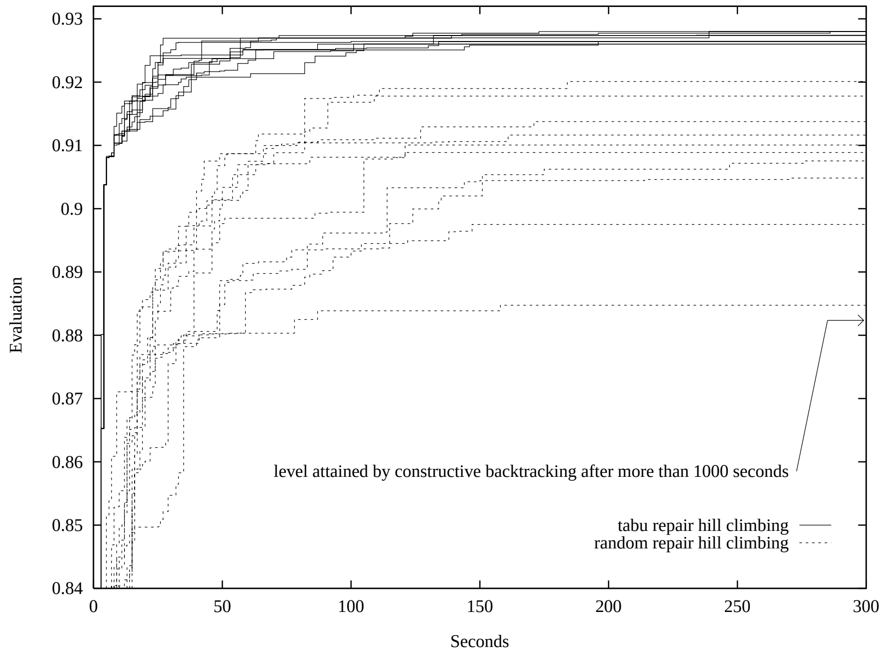

Figure 1: Tabu list min-conflicts repair based hill climbing heuristic versus min-conflicts repair based random search hill climbing heuristic, 10 samples each, with a fixed bad random initialization, versus constructive backtracking heuristic.

be further improved, especially since constraints could only be broken through explicit user-intervention, and because the relaxing of constraints was not evaluated.

To investigate real world PCSPs, we have designed and partly implemented a family of C++ libraries: *FLIP++ (pronounce: StarFlipPlusPlus; this refers to the fact that *FLIP++ stands as a regular expression for all names of the individual sub-libraries, and all of the latter are based on FLIP++, which is a C++ library). *FLIP++ is composed of the following layered sub-libraries: FLIP++: a basic fuzzy logic inference processor library. ConFLIP++: a static fuzzy constraint library. DynaFLIP++: a dynamic fuzzy constraint generation library. DomFLIP++: a domain knowledge representation library. OptiFLIP++: a heuristic optimizing library (several heuristics have so far been implemented and tested, namely: a tabu list min-conflicts repair based hill climbing heuristic, a min-conflicts repair based iterative deepening heuristic, a min-conflicts repair based random search hill climbing heuristic, and a min-conflicts repair based genetic algorithm heuristic). CheckFLIP++: a knowledge-change consistency checker library that also allows a fine-tuning of the configuration parameters of a problem. InterFLIP++: a userinterface for all other libraries. DocuFLIP++: an on-line documentation available separately for end-users, knowledge-engineers, and programmers, and accessible via the World-Wide-Web as HTML documents. In the present paper, we focus on the roles played by ConFLIP++ and DynaFLIP++. Of course, these libraries come without the domain-knowledge base. Therefore, during a first knowledge-acquisition phase, the knowledge engineer describes the products and the machinery with their respective attributes for the environment. In a second step, the functional relations among process variables have to be defined. A variety of mathematical description methods have been implemented for this goal. In a further step, constraints for these process variables can be entered to define restrictions in the value domains of these variables. When this knowledge-modelling step is finished, a given instantiation of a schedule can be constructed from actual process data and, after a schedule has been proposed, evaluated. After the evaluation of all constraints, changes on the schedule are usually done in order to find a more satisfying instantiation. What these changes are and how they look like is specified by the knowledge engineer in relation to the optimization methods supported by the OptiFLIP++ library. For instance, the genetic optimization algorithm uses special genetic operators such as the cross-over operator to perform changes on the schedule, which are useless in the tabu search type optimization. So for each optimization algorithm the knowledge engineer wants to apply to the problem at hand, it is possible to specify a range of corresponding operators that change something in the schedule.

Figure 1 shows some results for the tabu list min-conflicts repair based hill climbing heuristic versus the min-conflicts repair based random search hill climbing heuristic, for 10 samples each, with a fixed bad random initialization evaluating to 0.58139 for the chosen configuration, versus the constructive backtracking heuristic. The repair based timings were taken on a 386-PC and not on a Sparc workstation as for the constructive approach in order not to disturb the statistics by the UNIX multi-user virtual-memory management operating system.

Results using the other repair based algorithms (iterative deepening, genetic algorithm) implemented in *FLIP++ have so far yielded intermediate results, almost equal to those found by the tabu list hill climbing heuristic. As one sees in Figure 1, simple random search repair is easier trapped in suboptimal solutions. Starting from a better initial instantiation (for example from the solution found with the constructive approach) does not really influence results taken after three minutes run-time. More statistics and a detailed analysis of the different algorithms can be found in Dorn et al. [3]. All repair based heuristics are much faster and yield better results than the constructive approach that was evaluated using the same configuration parameters. However, the timing results are not really comparable because of different programming/run-time environments and slightly different rule sets used to search solutions in the constructive backtracking case, thus leading to slightly different optimal solutions.

Besides the application of *FLIP++ to the steelmaking plant LD3 of the VOEST Alpine Stahl AG in Linz, Austria, there are concrete plans to reuse as much as possible of the library when applying it to other problems in the future.

2 Implementation issues and results with ConFLIP++

The reusable C++ object library ConFLIP++ is a constraint handling extension to FLIP++, which itself is a general purpose fuzzy logic inference library. FLIP++ handles everything concerning fuzzification, membership functions, and linguistic variables. The user can choose between several different fuzzy inference methods, various priority schemes, different aggregation operators, and several defuzzification methods. FLIP++ also permits the graphical editing of membership functions and the easy manipulation of rule sets. Bonner et al. [1] describe for instance how to solve a fuzzy control problem using FLIP++ alone. The overall interaction model of all *FLIP++ libraries is analogous to the layers of an onion, where each layer interacts only with its immediate neighbors. The InterFLIP++ userinterface tool supports all functionality provided by ConFLIP++ and FLIP++. This includes creating, interactively editing, saving and reloading named sets of constraints including all parameters, and evaluating constraints. ConFLIP++ thus serves as a knowledge engineering tool in which domain knowledge can be stored, manipulated, and used for reasoning independently from the rest of the program.

2.1 Modeling fuzzy constraints with ConFLIP++

In ConFLIP++, the first step is creating simple constraints like

$$\text{aluminum-content} \leq 0.08$$

and named, e.g., 'aluminum-constraint', using the objects and methods defined in ConFLIP++. The aim is to catch vagueness in constraint-equations where the $\leq$ sign is not meant to be interpreted in its strict mathematical sense, but such that 'smaller' violations are acceptable. What these 'smaller' violations could be has to be defined explicitly (and precisely) through the membership functions associated to the 'terms' of the variable as defined below. Additionally, ConFLIP++ is able to handle uncertainty about the exact value of 'aluminum-content', which is possible by propagating possibility distributions instead of defuzzified values. The operators to infer values and to aggregate several constraints are then applied to fuzzy values, which can always be represented as membership functions. This capability to model *with accuracy* vague relations and uncertain data is the major contribution of fuzzy and possibilistic logics.

In our example of a simple constraint, 'aluminum content' is a so called *linguistic variable*, a generalization of the conventional concept of a variable. A linguistic variable has a finite set of terms, which are mapped to an interval of real numbers by a membership function. By a linguistic variable we mean a variable whose values are words or sentences in a natural or artificial language rather than numbers. In ConFLIP++ fuzzy sets like linguistic terms are represented by the *ParameterSet-object*. For example the fuzzy set *temperature* has the linguistic terms *cold, medium, warm*, which are mapped to intervals of real temperature values by the appropriate membership function. The object to model a simple constraint has the following structure:

ConstraintCompare(name, importance, dilatation, comment, variable, compare_operator, compare_value)

In the next step, several such constraints are logically combined, ie, they are aggregated by one of the aggregation operators, such as AND and OR, which for example could be implemented as *minimum* and *maximum* operators, to build more and more complex constraints with a hierarchical structure. The object to model such complex constraints has the following structure:

ConstraintConcat(name, importance, dilatation, comment, constraint, constraint, concat_operator)

where dilatation is the type of the constraint (either crisp, fuzzy, or mixed), and constraint is either a ConstraintCompare or a ConstraintConcat. ConFLIP++ then automatically creates a ruleset out of default or userdefined terms of sets for standard linguistic variables, standard rule set tables, standard membership functions for the term sets, default priority values, and various default operators using FLIP++. FLIP++ is called again and again later to evaluate the constraints for some instantiations of the free linguistic variables appearing in the constraint. Additionally, the system checks the scores of all constraints having a priority different from zero as well as of their constituent subconstraints before these constraints are aggregated to find out whether a hard constraint violation occurred (evaluation score equals zero) in order to invalidate instantiations that crossed the hard barrier of the corresponding constraint, which is not allowed.

The ruleset is built for instance such that, if the first linguistic variable is compared to its term 'positive_big', and the involved inequality is 'variable $\geq$ constant', and another linguistic variable is compared to 'zero', and the constraints corresponding to the two linguistic variables are concatenated by 'or', then the resulting term for the aggregated rule is 'very_good'. The latter term comes from the predefined standard term set {'very_good', 'good', 'zero', 'bad', 'very_bad'}.

The human expert will usually have to fine-tune the automatically created ruleset and the membership functions associated to the terms. However, it is possible to store user-defined standard sets of term sets and the associated set of semantic rules. Additionally, fuzzy methods are quite robust, such that the exact determination of the membership functions is not essential. The predefined triangular membership functions often perform well in a first approximation. Nevertheless, one reason that makes fine-tuning necessary is that ConFLIP++ has no a-priori domain knowledge. If the constraint is 'aluminum-content $\leq$ 0.08', some generated default rules are for instance:

IF aluminum-content is positive_small THEN aluminum-constraint is zero
IF aluminum-content is positive_medium THEN aluminum-constraint is bad
IF aluminum-content is positive_big THEN aluminum-constraint is very_bad

The really important object of ConFLIP++ is *SetOfConstraints*. Each SetOfConstraints has a name and a list of constraints. Furthermore, it needs a ruleset-object, tables for concatenating and comparing constraints with the appropriate operators defined in the *OperatorSet*, and a parameter set describing fuzzy linguistic variables.

The next step is the evaluation of a constraint. The evaluation happens according to the ruleset of the constraint. First, the free linguistic variables have to be given values, the latter being either defuzzified real numbers or possibility distributions themselves. The evaluation function returns by default a defuzzified value that describes the degree of satisfaction of the constraint with the given values. The ConFLIP++ object that holds evaluated constraints for further operations is the *SetOfEvalConstraints*.

The human expert can influence the decision making behavior of ConFLIP++ in various ways. After a constraint knowledge base has been compiled, it can be copied and the copy can be edited. First, the human expert can select one of several aggregation, implication, and defuzzification operators. The weighting scheme can be chosen as well. Of course, the individual membership functions and priorities of the constraints can be graphically edited. For instance, it is easy to selectively edit the constraint responsible for the observation of due-dates. These changes will immediately take effect on the decision function. To ease configuration of a complete constraint knowledge base built up from scratch, the default values for all these parameters are prespecified in a way that seems to apply reasonably well to most cases. However, the human expert can later soften or harden all those constraints that have not yet been fine-tuned on an individual base. In such a case, ConFLIP++ searches the complete knowledge base for membership functions that make the decision making behavior of the constraint knowledge base fuzzier or crisper.

$$
\begin{array}{llll}
\text{Job}_i\text{.C.high} \overset{\text{soft}}{<} \text{alloy\_limit.C} & \overset{\text{soft}}{\wedge} & \text{Job}_{i+1}\text{.C.high} \overset{\text{soft}}{<} \text{alloy\_limit.C} & \overset{\text{soft}}{\vee} \\
\text{Job}_{i+1}\text{.C.high} \leq \text{Job}_i\text{.C.high} & \overset{\text{soft}}{\wedge} & \text{Job}_i\text{.C.low} \leq \text{Job}_{i+1}\text{.C.low} & \overset{\text{soft}}{\vee} \\
\text{Job}_i\text{.C.high} \leq \text{Job}_{i+1}\text{.C.high} & \overset{\text{soft}}{\wedge} & \text{Job}_{i+1}\text{.C.low} \leq \text{Job}_i\text{.C.low} & \overset{\text{soft}}{\vee} \\
\text{overlapping.C}\langle \text{Job}_i,\text{Job}_{i+1}\rangle \overset{\text{soft}}{\geq} \text{pair\_limit.C} & & &
\end{array}
$$

$$
\Big(\text{where} \quad \text{overlapping.C}\langle \text{Job}_i,\text{Job}_{i+1}\rangle := \\ \max(0, \min(\text{Job}_i\text{.C.high},\text{Job}_{i+1}\text{.C.high}) - \max(\text{Job}_i\text{.C.low},\text{Job}_{i+1}\text{.C.low}))\Big)
$$

$\vdots$ similar for other chemical elements.

Table 1: Sketch of macro definition for Chemical_Compatibility_CC$_3\langle \text{Job}_i,\text{Job}_{i+1}\rangle$, restricted to carbon.

3 Implementation issues and results with DynaFLIP++

Dechter and Dechter [2] introduced the term *dynamic constraint network* to deal with changes from one constraint network to another one, such that new facts about the environment can be modeled. While this issue is addressed in the CheckFLIP++ part of *FLIP++, we want to focus here on a different problem, namely on the *dynamic* generation of constraints at runtime, based on environment data and so called template or static constraints. Thus, our use of the term *dynamic constraint* is unrelated to the dynamic constraint networks defined in [2].

The DynaFLIP++ library interacts with ConFLIP++, the latter being needed to formulate static fuzzy constraints and then operate on them. DynaFLIP++ wraps additional functionality around these fuzzy constraints and reaches this functionality up to DomFLIP++ where the actual domain knowledge is processed and optimization is performed. This additional functionality has become necessary because of factors encountered in many different real-world optimization problems, of which we only became aware of after trying to handle two different problems, namely steel production scheduling and classroom roostering. Since special consideration is given to the reusability aspect of all libraries, we found that static fuzzy constraints are good to model the aspects associated with partial satisfaction, compromising, and relative importance of constraints. However, other aspects require dynamic generation of constraints from template constraints. For instance, the actual number and kind of constraints often depend upon the current instantiation of the problem that is to be optimized. With each search step in the optimization process, the structure of this instantiation may change. Therefore, the structure of the constraint evaluation tree has also to change. For example, it is possible that a job with an associated delivery date is exchanged with another job that has no delivery date, therefore this constraint must not be evaluated for the second schedule.

Another aspect is that it is normally not useful to tune each constraint separately. Instead, a static constraint is tuned for a selected reference value, and DynaFLIP++ then uses this static constraint to generate a dynamic constraint adapted to the actual situation. Again, looking at the delivery date example, this implies that the template is a static constraint that is tuned around the value zero, with an appropriate fuzzy distribution around it. DynaFLIP++ then specializes this to an actual time in the scheduling horizon, for instance 10 a.m. on April 22, 1996. This dynamic adaptation is mostly harmless for simple constraints such as delivery dates, but becomes more tricky when complex constraints are involved. A more complex example, also taken from the steelmaking domain, would be the duration of the tundish life expectancy. The problem is that the attributes of a definite number of jobs must be aggregated, in this case by adding their durations, without knowing at the time when the static constraint is defined how many jobs will have to be eventually aggregated. Their number can only be determined dynamically at optimization time by checking certain criteria of the actual schedule instantiation. These aggregation operators are similar to those found in spreadsheet software that process a range of values. At this point we would like to observe concerning the use of the term *aggregate* that it is a synonym for large metallurgical equipment such as continuous casters or blast furnaces in the context of steelmaking. On the other hand, *aggregation* stands for the conjunction of constraints by soft AND and OR operators as we have seen before. Additionally, *aggregation* as in *aggregation operators* in the context of this section can also be found in the field of constraint databases as described by Kuper [5]. This is because

Figure 2: Outline of constraint tree constructed by DynaFLIP++.

CONDITION:
  $\langle \text{Job}_i, \text{Job}_{i+1} \rangle \in \text{CC}_3 \wedge$
  quality separation $\notin \langle \text{Job}_i, \text{Job}_{i+1} \rangle \wedge$
  tundish change $\notin \langle \text{Job}_i, \text{Job}_{i+1} \rangle \wedge$
  CC setup $\notin \langle \text{Job}_i, \text{Job}_{i+1} \rangle$
CONSTRAINT:
  Chemical_Compatibility_$\text{CC}_3 \langle \text{Job}_i, \text{Job}_{i+1} \rangle$

Table 2: A rule specifying when and how a template constraint is used.

the number of arguments in constraints evaluated by DynaFLIP++ depends on conditions that can be checked only at execution time.

Another difficulty arises because DynaFLIP++ contains absolutely no knowledge about the domain. Using DomFLIP++, optimization structures are defined together with their associated constraints, using the knowledge-engineering features of DynaFLIP++. Additionally, relations between attributes of items in these optimization structures and the static constraints must be defined using a special rule-based language that is interpreted at evaluation time by DynaFLIP++. A rule, slightly simplified but taken from the actual steelmaking domain, illustrates this mechanism as given in Table 2. Here, 'Chemical_Compatibility_$\text{CC}_3 \langle \text{Job}_i, \text{Job}_{i+1} \rangle$' stands as a macro that links several attribute values of Job_i and Job_{i+1} to linguistic variables defined in the pre-tuned static constraints. For the actual implementation, the whole definition of this macro must be specified. A sketch of the definition, where only the formulas involving the chemical element carbon are detailed, is presented in Table 1, thus giving an idea of what kind of dynamic adaptations must be computed. The actual compatibility encompasses 12 more chemical alloying elements, the degassing procedure in the secondary metallurgy aggregates, as well as the casting format between the jobs.

3.1 Evaluating dynamic constraints

To evaluate a given instantiation of a partial constraint satisfaction problem, a decision function aggregating all the constraints with their respective priorities, using an appropriate aggregation operator and a corresponding weighting scheme, must be established. Whereas the representation of template constraints is handled with the ConFLIP++ library, we present in this section the DynaFLIP++ library responsible for efficiently establishing a new global constraint representation for a specific instantiation of the problem. This global constraint will result in a highly structured constraint tree for the whole schedule. The constraint evaluation function will return the weighted global satisfaction score based on the current schedule and the constraint evaluation tree. Figure 2 outlines the general structure of this dynamically constructed tree, where the nodes are weighted aggregation operators (in the simplest case conjunctions) and the leafs are ConFLIP++ objects representing individually fine-tuned static constraints, again taken from the steelmaking domain. DynaFLIP++ is able to use most of the framework provided by ConFLIP++ to efficiently compute the evaluation scores for a new schedule.

When scheduling, it is often advisable to introduce an additional measure into the decision function dependent upon whether the current schedule (= instantiation of the constraint satisfaction problem) contains certain difficult jobs. If the scheduling of these jobs is not introduced as a bonus into the decision function, these jobs might never be considered for actual scheduling. There usually exists a non-empty pool of waiting jobs, and only a subset of jobs from the pool can be scheduled immediately. Therefore, the danger is that some difficult jobs will remain in the pool forever unless additional measures are taken. It is clear that this 'difficulty' or 'importance' of a job must increase over the time for which it is still reasonable to 'produce' it, to favorize its eventual scheduling. The easiest way to introduce this 'difficulty' is to formulate a corresponding constraint with an associated priority that will represent these difficult jobs and which will therefore be represented by another branch of a certain constraint type as in Figure 2. Thus, the 'difficulty' of jobs will be one criteria considered when the partial constraint satisfaction problem is optimized. The same applies equally to other partial constraint satisfaction problems such as those encountered in design or planning.

3.2 Implementation issues

The DynaFLIP++ library is located between ConFLIP++ and the domain knowledge representation library DomFLIP++, which is a description tool for the environment that has to be optimized. The structure of a domain holds a list of aggregates, which themselves hold a schedule of objects with their respective attributes and variables. Additionally, on each level one or more SetOfConstraints can be specified in order to describe relations and restrictions on the process variables. DomFLIP++ is also responsible for repair steps on a badly evaluated schedule, using a list of violations and badly evaluated variables, which are computed at runtime by DynaFLIP++, and the optimization algorithms supported by OptiFLIP++.

To build up the evaluation tree, DynaFLIP++ has to consider the domain structure with its aggregates and scheduling objects, so that the evaluating tree is built up analogous to the hierarchical structure of the modeled application. The structure of the scheduling objects depends on the application they are designed for. Considering production scheduling in industrial environments as a special CSP, we meet different types of imprecision, stemming from constraints that are blurred in definition and include vagueness and uncertainty. We can imagine that an operation on the schedule may start a 'little' earlier and that 'small' deviations of optimal values may be acceptable. The scheduling object 'Order' then has attributes such as *plant, plantmark, throughput, weight, format, thickness, speed, slabgroup, chemical_elements, delivery_date, ...* and associated constraints like

$$\text{domain constraint: out\_date} \leq \text{delivery\_date}$$

In this case, an aggregate could be the *continuous caster CC-4*, with constraints concerning *tundish durations, average throughput, or setup-restrictions.* Of course there would exist a variety of other constraints, such as compatibility constraints, capacity constraints, or temporal constraints. To evaluate a schedule for one aggregate, DynaFLIP++ has to first create and instantiate all SetOfConstraints for the domain objects which are designed by the knowledge engineer. In a second step, all the variables that have restrictions in form of constraints have to be computed. This is done by evaluating the computation clauses designed parallel to the related constraints. Of course all variables have to be computed before they are compared to constraints. The next step is the adaptation of template constraints to the actual situation on the schedule. We can imagine that a template compare value has a fuzzy distribution around zero, but the real compare value may be an aggregation of other processes variables and would have another value.

$$\text{template constraint: fuzzy\_var\_foo1} \leq 0$$
$$\text{specialized constraint: out\_date - delivery\_date} \leq 0$$

On the evaluation of each constraint, DynaFLIP++ invokes the evaluation mechanisms of ConFLIP++. The evaluated constraints are put into the SetOfEvalConstraints, where they can be aggregated as described in Table 2. When all relevant constraints determined by the rule-based language are evaluated, the overall evaluation score is returned to DomFLIP++. Further, we are interested in the biggest violations on the schedule, so we select bad evaluations at runtime and put them into an appropriate data structure, which is later used by DomFLIP++ to make changes on the schedule in order to avoid these violations. The repair steps depend on the optimization algorithm used by DomFLIP++. If a local change on the schedule has occurred, DynaFLIP++ has to check where the changes took place, in order not to build up a complete new evaluation tree, but only to recompute those parts of the schedule that have been changed. This reuse of already computed data structures will, similar to a caching mechanism, influence the runtime behavior of the evaluation process.

The most important object of DynaFLIP++ is the evaluation tree which contains the specialized constraints, the evaluated constraints, and the aggregation of the latter. The variables with their actual values and the violations are stored in a separate structure.

3.3 Dynamic constraint satisfaction algorithms

To guide the search of the OptiFLIP++ algorithms as discussed in [6], it is necessary to identify the constraint with the worst weighted evaluation, ie, the severest conflict which can be attacked to minimize conflicts. This can be considered as a side product of evaluating the current instantiation. It corresponds to computing the evaluation using the minimum operator, and more importantly, to remember the constraint involved in the minimal weighted evaluation. This constraint represents the largest conflict for the current instantiation. Often the constraint corresponds to a general feature of the instantiation and cannot be attributed to a specific part of the instantiation. Depending on the repair operators available to the repair based constraint satisfaction algorithms, it can be helpful to find additionally the second largest and third largest conflict. Generally, the search should return the largest conflict being of a type that can be handled by an available repair operator. When DynaFLIP++ has to generate a new dynamic constraint representation for a given instantiation, it computes the individual 'leaf' constraints by calling ConFLIP++ again and again with new variable instantiations on one of the stored reference constraints, and stores the results in an intermediate form that can be used by ConFLIP++ for further aggregation. This ensures a relatively efficient processing of the constraints since the sometimes very large data structure of a static constraint can be reused for all dynamic constraints of its type. At the same time, DynaFLIP++ sorts all the computed intermediate evaluation scores, together with type information, for later selection of 'good' repair operators.

4 Conclusion

In this paper, we presented problems as well as solutions associated with fuzzy constraints in real-world partial constraint satisfaction problems. We presented the knowledge engineering tool ConFLIP++ for modeling fuzzy constraints that can be aggregated to complex, hierarchical constraint structures. We showed its practical application in a steelmaking plant using fine-tuning and specializing concepts. We presented the DynaFLIP++ library which, based on ConFLIP++, is responsible for the evaluation of an instantiated PCSP. We also gave an overview of the heuristics and repair based algorithms integrated in *FLIP++.

References

[1] M. Bonner, S. Mayer, A. Raggl, and W. Slany. FLIP++: A fuzzy logic inference processor library. In *Workshop Notes of the IJCAI'95 Workshop on Fuzzy Logic in Artificial Intelligence*, pages 83–92, Montreal, Canada, Aug. 1995. URL: ftp://mira.dbai.tuwien.ac.at/pub/slany/ijcai95wsflai1.ps.gz.

[2] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings AAAI-88 Seventh National Conference on Artificial Intelligence*, pages 37–42, St. Paul, Minnesota, USA, Aug. 1988. Published by the American Association of Artificial Intelligence.

[3] J. Dorn, M. Girsch, G. Skele, and W. Slany. Comparison of iterative improvement techniques for schedule optimization. CD-Technical Report 94/61, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, 1994. A revised version has been accepted for publication in the European Journal of Operational Research. URL: ftp://mira.dbai.tuwien.ac.at/pub/slany/cd-tr9461.ps.

[4] J. Dorn, W. Slany, W. Snopek, C. Stary, W. Steindl, and K. Stohl. Aufgabenanalyse der Dispatcher im Stahlwerk LD3. CD-Studie 93/11, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, 1993. In German.

[5] G. M. Kuper. Aggregation in constraint databases. Technical report ECRC 93–13, ECRC, 1993. URL: ftp://ftp.ecrc.de/pub/ECRC_tech_reports/reports/ECRC-93-13.ps.Z.

[6] W. Slany. Scheduling as a fuzzy multiple criteria optimization problem. CD-Technical Report 94/62, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, 1994. A revised version has been accepted for publication in Fuzzy Sets and Systems. URL: ftp://mira.dbai.tuwien.ac.at/pub/slany/cd-tr9462.ps.Z.

[7] W. Slany. Comparing partial constraint satisfaction models. In *Workshop Notes of the CP'95 Workshop on Over-Constrained Systems*, pages 151–159, Cassis, France, Sept. 1995. URL: ftp://ftp.cs.city.ac.uk/pub/constraints/papers/ocs-workshop.tar.gz.

[8] K. Stohl, W. Snopek, T. Weigert, and T. Moritz. Development of a scheduling expert system for a steelplant. In *Proceedings of the IFIP Conference on Experts Systems in Mineral and Metallurgy*, Finland, 1991.