6. The *FLIP++ library

6.1 Introduction

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: atabu 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 user-interface 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 this thesis, we focus on the roles played by ConFLIP++ and DynaFLIP++. For the functionality of the other libraries we will give only a short overview. 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. The *reusability* of the libraries was a fundamental aim while designing and implementing the single modules.

Figure 6.1 shows the module hierarchy of the *FLIP++ library:

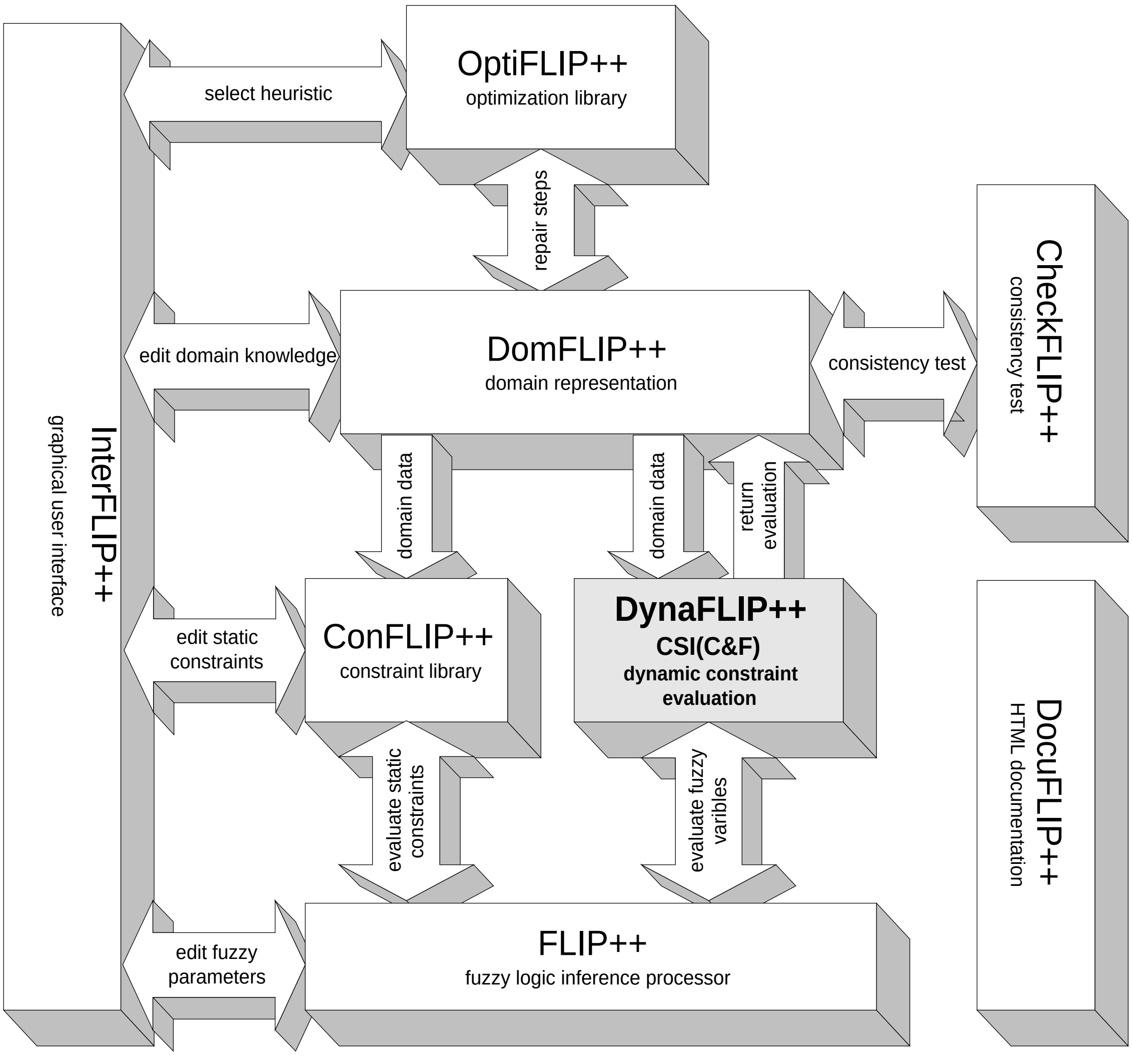


Figure 6.1: interaction structure of the *FLIP++ libraries

6.2 The FLIP++ library

FLIP++ 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,
- ☐ several **defuzzification** methods.

FLIP++ also permits the *graphical editing* of membership functions and the easy manipulation of rulesets. Bonner et al.[BONNER95] describe for instance how to solve a fuzzy control problem using FLIP++ alone. In short terms, the main purpose of this module is to evaluate fuzzy inference rules that determine the behavior of a system, such as e.g.

```
IF temperature IS warm THEN cooling IS medium WITH CERTAINTY 0.7
```

These fuzzy inference rules express several fuzzy judgments about relations between different system variables. These judgments are in the form of membership functions and can be simplified to the linguistic term to which the judgment mainly pertains. The resulting fuzzy values can be all combined by computing a weighted mean of the membership functions for each component to get one overall value. The result of an evaluation of a rule can be used as crisp- (float) value for controlling or as a fuzzy value to be used in further rules. A resulting linguistic term can be defuzzified into a crisp value by calculating the center of gravity of the surface and then taking its x-coordinate as the result.

Control:	result is 150V for controlling the air-condition
	or
Rule:	IF cooling is a little_bit THEN open_ventile is half.

A detailed description of this library and the implementation issues with FLIP++ can be found in [BONNER95]. In chapter 7 we will see something about the interface of FLIP++ and CSI(C&F). Most of the ‘fuzzy’ functionality of DynaFLIP++ is achieved by mapping the CSI(C&F) fuzzy constructions to FLIP++ objects and methods. For the theoretical background and the basic concepts on fuzzy decision making, linguistic variables and the use of membership functions please refer to Chapter 3.

6.3 The ConFLIP++ library

The reusable C++ object library ConFLIP++ is an extension to FLIP++ for the handling of *static constraints*. 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. ConFLIP++ can be used for constraint evaluation, where the related variables of the constraints have well defined values.

6.3.1 Modeling fuzzy constraints with ConFLIP++

In ConFLIP++, the first step is creating simple constraints like aluminum-content ≤ 0.08 and named, e.g., „aluminum-constraint-5.47“, using the objects and methods defined in ConFLIP++. The aim is to catch vagueness in constraint-equations where the ≤ 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 logic. 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 functions. Of course, before evaluating a constraint we have first to create it. Basically we distinguish two kinds of constraints, the C++ object *ConstraintCompare* to model a simple constraint has the following structure:

Parameter	Description
name	unique name of a constraint
importance	a real number ∈ [0,1] denoting the relative importance
dilatation	denotes the type of the constraint: fuzzy, crisp or mixed
comment	a user - editable comment for the constraint
variable	unique name of the variable this constraint is designed for
compare_operator	an operator out of the operator set { =, ≠, <, >}
compare_value	the value to which the variable is compared

Table 6.2: paramteters of the ConstraintCompare - object

In the next step, several such constraints are logically combined, i.e., 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 C++ object *ConstraintConcat* to model such complex constraints has the following structure:

Parameter	Description
name	unique name of a constraint
importance	a real number ∈ [0,1] denoting the relative importance
dilatation	denotes the type of the constraint: fuzzy, crisp or mixed
comment	a user - editable comment for the constraint
constraint_1	name of the first constraint for the new constraint
constraint_2	name of the second constraint for the new constraint
concat_operator	a conjunction operator out of the operator set {and, or}

Table 6.3: paramteters of the ConstraintConcat - object

where dilatation is the type of the constraint (either crisp, fuzzy, or mixed), and constraint_x is either a ConstraintCompare or a ConstraintConcat.

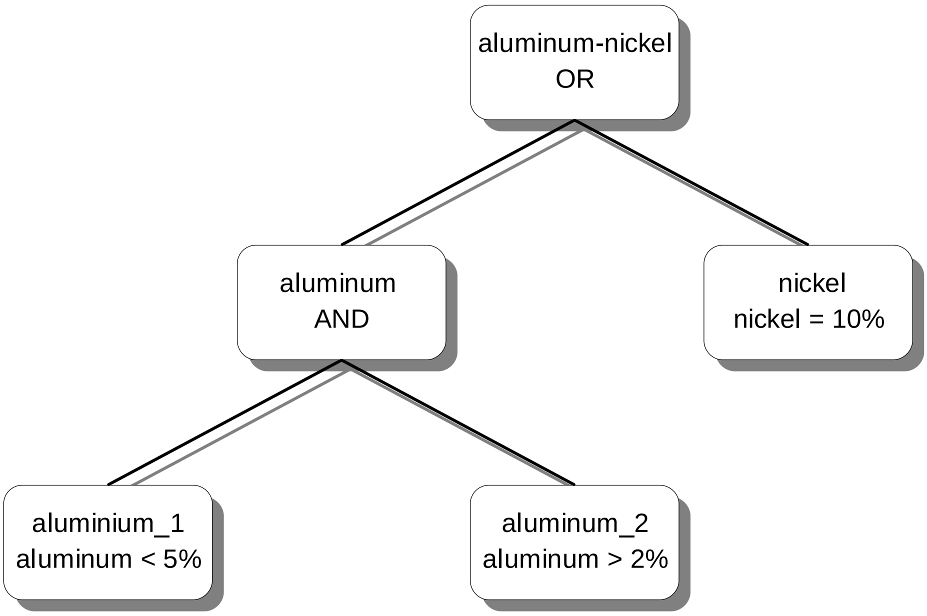


Figure 6.4: Example for a hierarchical constraint

ConFLIP++ then automatically creates a ruleset out of default or user-defined 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 ≥ 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’} or from a user defined one.

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 ≤ 0.08“, some generated default rules are for instance:

IF aluminum-content is positive_small THEN aluminum-constraint-5.47 is zero
IF aluminum-content is positive_medium THEN aluminum-constraint-5.47 is bad
IF aluminum-content is positive_big THEN aluminum-constraint-5.47 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 aggregation 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.

6.4 The DynaFLIP++ library

Dechter and Dechter [Dechter88] 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 [Dechter88]. In this section we will discuss the conceptual design and the functionality of DynaFLIP++.

In this thesis we can find two different terms for one subject: *DynaFLIP++* and *CSI(C&F)*. At this point we would like to remark that DynaFLIP++ stands for all conceptual issues stated in this thesis. CSI(C&F) is a *constraint script interpreter* on *crisp and fuzzy sets*, i.e., the implementation of all objectives and functionality of DynaFLIP++. This linguistic distinction may seem not to make a lot of sense, but it is based on a 'historical' motivation. Several different approaches of implementing DynaFLIP++ have been done, and, finally, CSI(C&F) seemed to be the best solution. Earlier implementations adapting the static constraint library ConFLIP++ did not perform as expected, and for this CSI(C&F) was developed.

DynaFLIP++ interacts with FLIP++, the latter being a general purpose fuzzy inference processor library, needed for all computation concerning fuzzification, defuzzification, membership functions, linguistic variables, distributions, etc. DynaFLIP++ wraps additional functionality around these objects 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 described in Chapter 5, 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.

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 specifying when and how a template constraint is used. Such a rule, slightly simplified but taken from the actual steelmaking domain, illustrates this mechanism as given in Table 6.5:

CONDITION:
$\langle Job_i, Job_{i+1}\rangle \in CC3 \wedge$
quality separation $\notin \langle Job_i,Job_{i+1}\rangle \wedge$
tundish change $\notin \langle Job_i,Job_{i+1}\rangle \wedge$
CC3 setup $\notin \langle Job_i,Job_{i+1}\rangle \wedge$
CONSTRAINT:
Chemical_compatibility_CC3$\langle Job_i,Job_{i+1}\rangle$

Table 6.5: A rule specifying when and how a template constraint is used

Here, „Chemical_compatibility_CC3$\langle Job_i,Job_{i+1}\rangle$“ 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. 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.

6.4.1 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. DynaFLIP++ is 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 6.6 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 evaluations of individually fine-tuned constraints, again taken from the steelmaking domain. DynaFLIP++ is able to use most of the framework provided by FLIP++ to efficiently compute the evaluation scores for a new schedule.

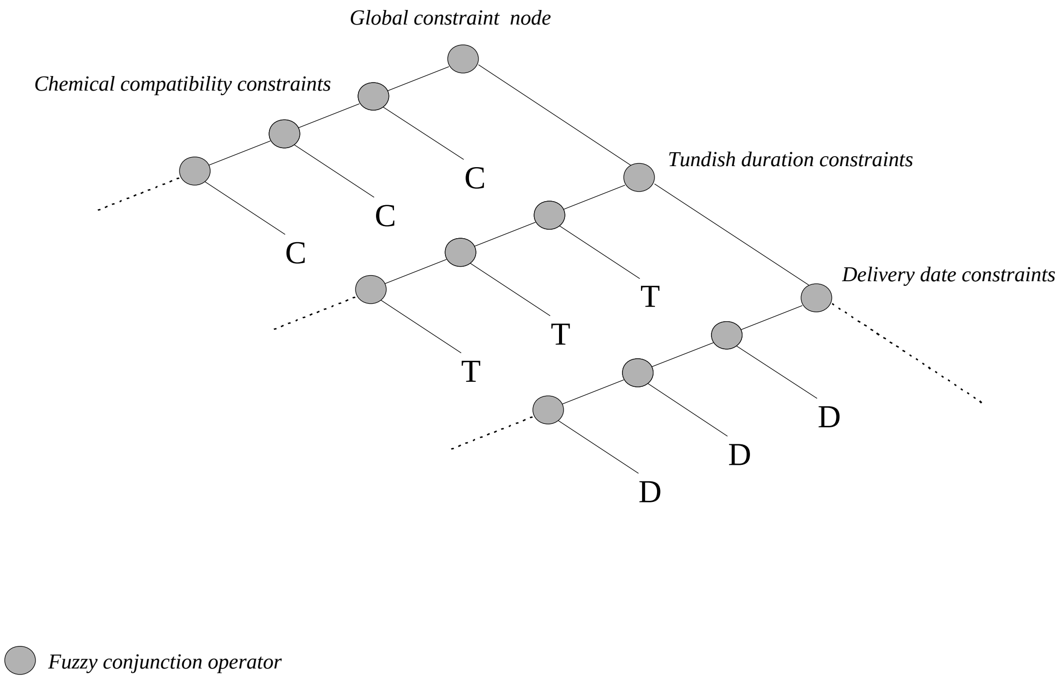


Figure 6.6: Outline of the constraint tree constructed by DynaFLIP++

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 schedule it, to favourize its eventual scheduling.

The easiest way to introduce this ‘difficulty’-measure 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 6.6. 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.

6.4.2 Conceptual issues with DynaFLIP++

The DynaFLIP++ library is located between FLIP++ and the domain knowledge representation library DomFLIP++, which is a description tool for the environment that has to be optimized. DynaFLIP++ has no restrictions about the complexity of a domain structure, all domain data is imported by the use of CSI(C&F) variables. A description of all CSI(C&F) structures can be found in chapter 7. In the steelmaking environment a domain holds a list of aggregates, which themselves hold a schedule of objects with their respective attributes and variables. For all variables used within a CSI(C&F)-script one or more constraints 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 byOptiFLIP++.

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

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

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 (see Chapter 5). To evaluate a schedule for one aggregate, DynaFLIP++ has to first create and instanciate all constraints 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. 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.

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

On the evaluation of each constraint, DynaFLIP++ invokes the evaluation mechanisms of FLIP++. The evaluated constraints are put into the EVALTREE, where they can be aggregated as described in Figure 6.6. When all relevant constraints determined by the constraint script interpreter CSI(C&F) 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 EVALTREE, which contains the specialized and evaluated constraints, and the aggregation of the latter. The variables with their actual values and the violations are stored in a separate structure.

6.4.3 Dynamic constraint satisfaction algorithms

To guide the search of the OptiFLIP++ algorithms as discussed in [Slany94a], it is necessary to identify the constraint with the worst weighted evaluation, i.e., 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 FLIP++ 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 DynaFLIP++ 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. All DynaFLIP++ functionality is implemented by CSI(C&F), an interpreter language for dynamic constraint adaptation and evaluation. In most cases, the values of variables are the result of mathematical or logical expressions, or aggregations of other variables. In this sense CSI(C&F) has to provide a bit more functionality than simple constraint stating and evaluating, thus CSI(C&F) comes quite close to traditional programming languages.

6.5 The DomFLIP++ library

DomFLIP++ is a domain knowledge representation library which allows the knowledge engineer to model the environment that has to be optimized. In a first step the objects that have to be scheduled and their respective attributes are defined. Each domain can hold a list of different aggregates. These aggregates can be seen as ‘schedules’. Supported by DynaFLIP++, constraints for each level in the domain can be designed. So we are able to distinguish constraints on the domain-level, the aggregate-level, and on the item-level, i.e., constraints on the single scheduling objects. In a further step a optimizing method, implemented in the OptiFLIP++ library can be chosen. The final task of DomFLIP++ is to establish a link between the optimizing algorithms and the dynamic constraint evaluation provided by DynaFLIP++. For each instantiation of a schedule on an aggregate DynaFLIP++ is called again and again, until a sufficient constraint satisfaction degree is reached.

6.6 The OptiFLIP++ library

OptiFLIP++ is 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,
- a min-conflicts repair based genetic algorithm heuristic.

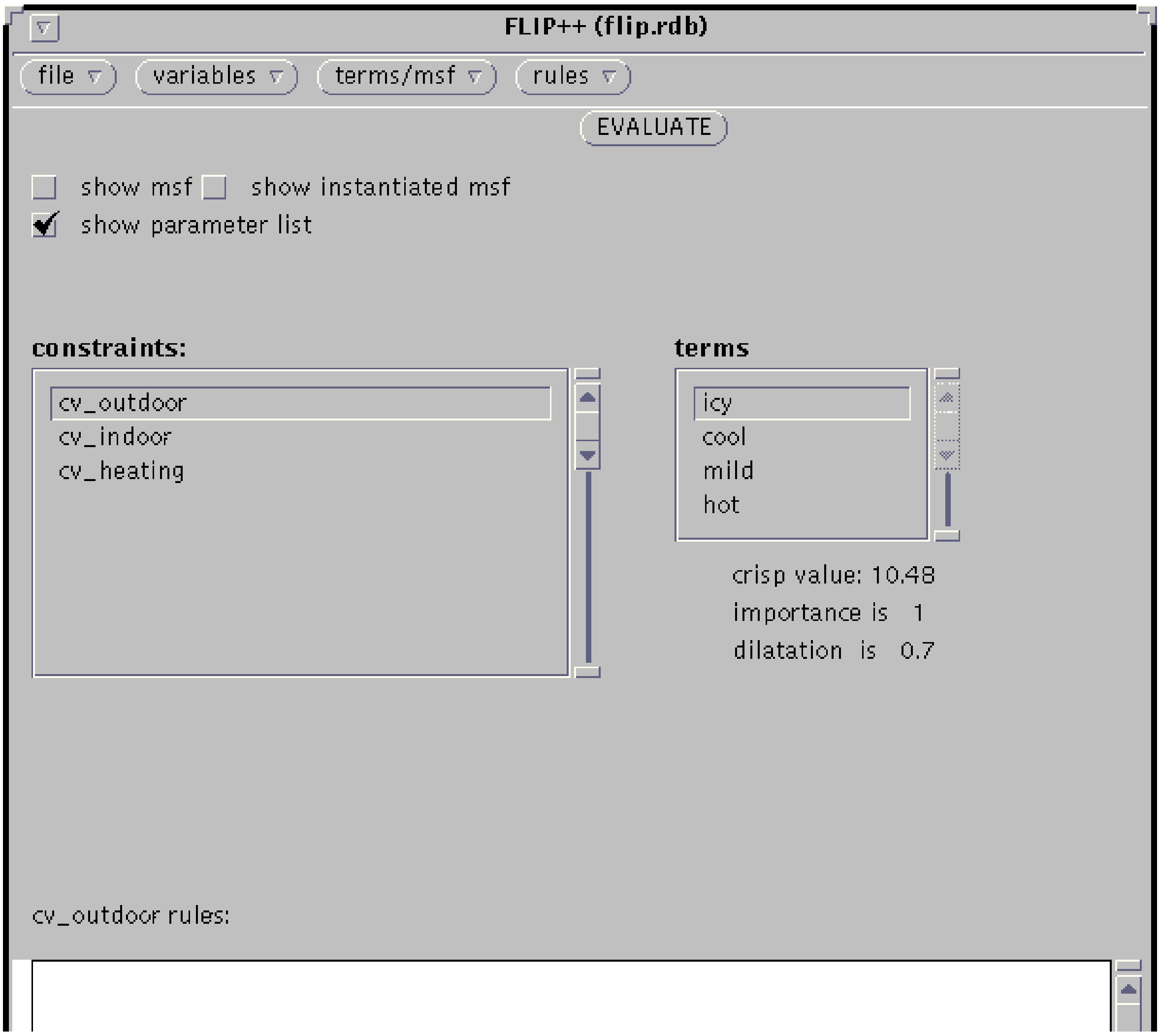
Results using this repair based algorithms implemented in *FLIP++ have so far yielded intermediate results, almost equal in performance. Simple random search repair is easier trapped in suboptimal solutions. Starting from a better initial instantiation (for example from the solution found with the constructive approach) does not really influence the results. More statistics and a detailed analysis of the different algorithms can be found in Dorn et al. [Dorn94]. All repair based heuristics are much faster and yield better results than the constructive approach that was evaluated using the same configuration parameters.

6.7 The other *FLIP++ libraries

CheckFLIP++ *is* a knowledge-change consistency checker library that also allows a fine-tuning of the configuration parameters of a problem. Configuration changes could be changes in the priorities between constraints, adapting a new aggregation operator, changing hard barriers, changing membership functions, or changing the logical structure of constraints. Thus it is necessary to test such configuration changes on consistency.

DocuFLIP++ *is* an on-line documentation available separately for end-users, knowledge-engineers, and programmers, and accessible via the World-Wide-Web as HTML documents.

InterFLIP++ *is* a user-interface for all other libraries. InterFLIP++ permits for instance the graphical editing of membership functions and the easy manipulation of rule sets, the creation of constraints, linguistic variables with their terms, etc.



☐ show msf ☐ show instantiated msf

☑ show parameter list

cv_outdoor
cv_indoor
cv_heating

icy
cool
mild
hot

crisp value: 10.48
importance is 1
dilatation is 0.7