Chapter 3

Design and Implementation Issues of DomFLIP++

Design. As I write a program, I should use a language that minimizes the distance between the problem-solving strategies that I have in my head and the program text I eventually write on paper.

—Jon Bentley, *Programming Pearls*

3.1 Introduction

DomFLIP++ is at the heart of the ∗FLIP++ libraries. It is responsible for the representation of domain knowledge. With DomFLIP++, the role of the knowledge engineer comes into play, too. He is the individual that forms the link between the knowledge of the domain expert and the knowledge of the system expert. With a perfect domain knowledge representation, the role of the system expert should fall together with the role of the domain expert since the additional knowledge about the computer system to implement a problem can be easily learned without the help of a specialist. However, this is not yet the case with the current status of DomFLIP++. Here, we rely on the interfacing role of the knowledge engineer who is capable of formulating domain specific problems in a computer language format and sometimes even entrusted with the implementation.

In this thesis, I tried to fulfill this task in the following way: At the beginning, I had to figure out a design of DomFLIP++ that works together in as seamless a way as possible with the other ∗FLIP++ components. Then, I proceeded with the implementation of DomFLIP++, which also included some more or less intensive adaptations of existing ∗FLIP++ modules. Finally, I tested the system with a real world problem, namely the design of good shift schedules, which will be dealt with in more detail in Chapter 5. Also, I had to acquire the intricacies of the shift scheduling domain (see Chapter 4 for more details on that). Only a good working knowledge of a domain is a guarantee for a successful representation of a problem domain. Besides, the thorough understanding of the domain in hand including advice both provided by a real domain expert are also almost a necessity.

Fortunately, some works and studies have been already completed about the fundamentals behind DomFLIP++. Firstly, I could rely on the insights and visions of Wolfgang Slany (see [20]), the initiator of the ∗FLIP++ project, concerning the general concept of DomFLIP++. Secondly, Witold Hendrysiak has already done some trial implementations of a DomFLIP++ like environment in connection with production scheduling in a steel production plant. He summarized his findings in [12]. With these two main references in hand, I had a sound basis to start with. Nonetheless, working through the problem domain revealed several new issues with the need to be addressed in the design of DomFLIP++.

The main tasks that have to be achieved by DomFLIP++ are the following:

- It should support the knowledge engineer with the definition of a new problem domain. This mainly concentrates on the provision of class and/or data structures in a generic way.

- The acquisition and administration of all domain relevant data. A problem instantiation has to be completely under control of DomFLIP++. All other parts of ∗FLIP++ do not depend in any way of domain specific knowledge.

- Connection of DomFLIP++ with other parts of ∗FLIP++, namely DynaFLIP++ and OptiFLIP++. This is a crucial point for the successful implementation of DomFLIP++. Since it is solely focused on domain knowledge representation, DomFLIP++ has to rely on the services of other parts to achieve a positive treatment of a problem.

  Firstly, an instantiation of a problem has to be evaluated according to certain, domain-dependent requirements, or in ∗FLIP++ terms constraints. This evaluation process is done by the DynaFLIP++ library, which has been specifically designed to handle partial constraint satisfaction problems allowing for antagonistic criteria.

  Secondly, to pursue the objective of optimization DomFLIP++ needs the support of the optimization library OptiFLIP++. It provides algorithms that work on a problem instantiation partly guided by its evaluation (provided by DynaFLIP++), partly by the use of domain specific knowledge (provided by DomFLIP++). All three ∗FLIP++ components together can be viewed as a decision support tool that tries to optimize problems in cooperation with and under the guidance of a knowledge engineer.

3.2 Design issues of DomFLIP++

In the following the different approaches that have been made so far to design and implement DomFLIP++ are examined in more detail. The first approach is to be considered as a basis and fruitful input for the second, which justifies its explanation within this thesis.

3.2.1 A first design approach

The first approach proposed by Hendrysiak [12] did start from different premises than the one I will explain in this chapter. At that time, ∗FLIP++ has not been in a ready-to-use development stage. Especially, parts crucial for the smooth operation of DomFLIP++ like DynaFLIP++ had not been finished by the time of inception of DomFLIP++. Of course, this confronted the design team with some problems and did not facilitate the design of DomFLIP++ at all. The only reasonable solution was to work with well-defined interfaces between the different parts of ∗FLIP++. In particular, the interface between DomFLIP++ and DynaFLIP++ has been of a more virtual nature. And test runs had to be done with dummy evaluation functions as long as DynaFLIP++ was still undergoing its own implementation works. Nevertheless, the basic concepts remain the same with the approach taken by Hendrysiak and the one presented in this thesis.

Now, we take a closer look at the approach presented by Hendrysiak. Generally, he distinguishes between application-dependent and application-independent classes. While the former have to be designed anew according to the specific details of a problem and its domain, the latter show the same behavior for all domains merely changing their contents, i.e. the domain data they hold.

Hendrysiak also chooses a three layered model for domain knowledge representation. This model stems mainly from the findings drawn out of the main area of application which was production scheduling in the steel production domain. Later on, I will explain why my approach deviates from this particular aspect.

At the bottom level, the so-called *SchedulingObject* is a domain-dependent class. It is the representation of those elements in a domain that are in the main focus as far as modifications for optimization purposes are concerned. An optimizing algorithm performs its operations on exactly this level, for example, moving around *SchedulingObjects* in a list of several such. In the steel production application that has been used with this first version of DomFLIP++ each job that has to be scheduled for production represents one such *SchedulingObject*. This class contains all the attributes that are needed to describe the considered domain element. The attributes in turn are used as input for the constraints. These constraints are also part of the *SchedulingObject*. They formulate the various requirements a *SchedulingObject* has to fulfill. DynaFLIP++ uses the constraints to obtain an evaluation of the *SchedulingObject*.

The next level is the *Aggregate*. It is a conglomerate of several *SchedulingObjects*. The idea behind this level is the fact that most scheduling problems have different views. In the steel production application an *Aggregate* corresponds to a machine that processes a series of the above mentioned jobs. Another nice illustration of this is given in [13] where the different views of classroom scheduling in schools are mentioned. There, the problem can be viewed with the time table of a class, the time table of a teacher, or the allocation plan of a class room being in the focus of interest. Also, the *Aggregate* level allows to formulate other constraints than those in the *SchedulingObject* level. Since it is a container for several *SchedulingObjects*, it is also the appropriate level for holding constraints that refer to interrelationships of *SchedulingObjects*. For example, a constraint that depends on the processing time used for all jobs on a machine cannot be formulated within each job, i.e., within each *SchedulingObject*, because at that level it does not have access to other *SchedulingObjects*. On the *Aggregate* level such a constraint simply extracts the information it needs from all underlying *SchedulingObjects*.

At the top level comes the *Domain* class. This is not merely the aggregation of various *Aggregates*. *Domain* also provides all data structures necessary for the administration of domain data. Moreover, it supplies the interfaces to the other libraries of ∗FLIP++, more specifically, to OptiFLIP++, ConFLIP++, and DynaFLIP++. At the time when this approach was incepted ConFLIP++ was still an independent part of *FLIP++ and therefore required its own interface. Later on, ConFLIP++ and DynaFLIP++ have been merged together after a major redesign of DynaFLIP++ (see Chapter 2 for a more detailed explanation on this point). Apart from the list of *Aggregates*, the following structures and methods are contained in the *Domain* class:

- Domain constraints.

  Similar to the constraints on the *Aggregate* level, there are also constraints that are positioned in the *Domain* level. They refer to relations between several *Aggregates*.

- Constraint violations.

  This structure stores information about violated constraints. The decision whether and to what degree a constraint is violated is taken in the evaluation done by DynaFLIP++. A list of the most severely violated constraints is part of the result of each evaluation of a problem instantiation.

- Repair steps.

  Repair steps are modifications defined on the domain data. They mainly refer to the level of *SchedulingObjects*. For example, a possible repair step may be to swap two *SchedulingObjects* at defined positions in an *Aggregate*. The complexity of repair steps is mainly dependent on the structure of the problem. Still, with most problems the modifications can be reduced to quite simple exchange and move operations, especially when the domain data can be represented in a linear order. Repair steps provide the modification mechanism each optimizing algorithms needs in order to make its path through the search space. In combination with the constraint violations produced after each evaluation, one can devise a more or less sophisticated strategy that derives suggestions for repair steps. While the success of most algorithms relies positively on such recommendations, a random factor can be chosen as well. Randomization has the advantage of being easy to implement compared to a costly choice—in terms of performance—of repair steps. We will get back to this issue when we discuss the repair steps of the shift scheduling application in Chapter 5.

- Evaluation method (ConFLIP++/DynaFLIP++ interface).

  This method represents the interface to ConFLIP++ and DynaFLIP++. It is responsible for supplying DynaFLIP++ with an appropriate structure that contains all the domain data on a constraint basis. For the current problem instantiation all constraints are generated, using ConFLIP++, on each level of DomFLIP++ based on the various attributes of *SchedulingObjects*, *Aggregates*, and *Domain*. Afterwards, DynaFLIP++ evaluates these constraints. As a result, the most severely violated constraints and an overall evaluation score are returned.

- Optimization method (OptiFLIP++ interface).

  So far, a *Domain* can evaluate a problem instantiation. But to improve the instantiation as far as the satisfaction of constraints is concerned, one has to go one step further. An optimizing algorithms provides a strategy of how to work through the search space of the problem domain in order to arrive at a better solution. OptiFLIP++ provides several such strategies. They are based on the evaluation scores of a problem instantiation, the repair recommendations provided by the repair step choice strategy of the *Domain*, and, of course, the repair steps available.

- Several methods for the handling of domain data.

  These are fairly straightforward methods that store and retrieve domain data. Also, representing the problem instantiation on a user interface belongs to this area. Furthermore, if domain data have to be retrieved from other applications, a conversion has to take place within such methods.

The three-layered model described above is illustrated in Figure 3.1.

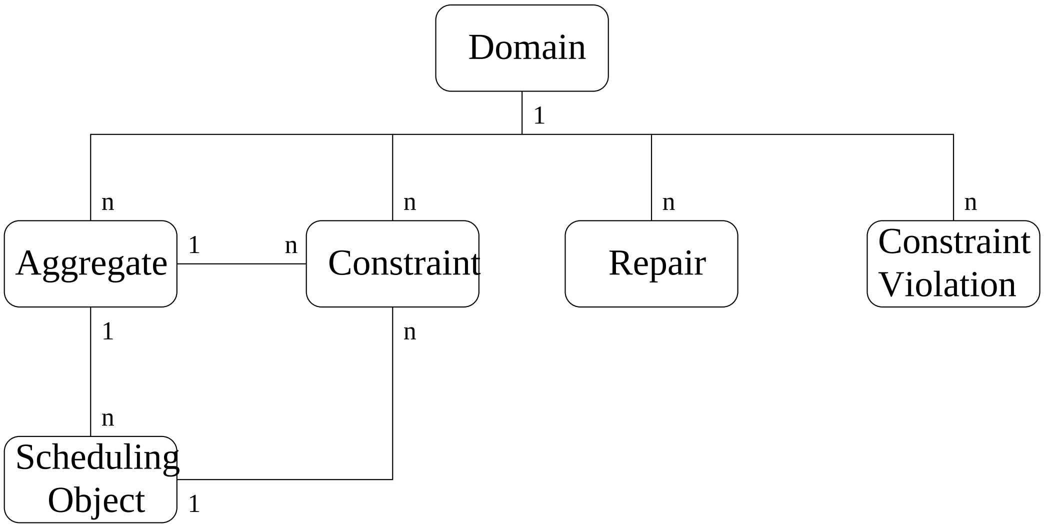

Figure 3.1: Class structure of the first approach of DomFLIP++

In summary, the approach taken by Hendrysiak is a sound basis to start with as far as the design is concerned. However, since the implementation of DynaFLIP++ has been put on hold later on in order to redesign it, some assumptions are not valid any more. With the current version of ∗FLIP++, DynaFLIP++ has swallowed all the functionality provided by ConFLIP++. Also, owing to the implementation of a constraints script interpreter (CSI) (see Chapter 2) the way DomFLIP++ interacts with DynaFLIP++ is somewhat different. Furthermore, bearing in mind the flexibility provided by the constraint script language, the design of a much more leaner DomFLIP++ is possible, for example, the constraint representation can be 'outsourced' to the CSI language.

3.2.2 A second design approach

Many hours of discussion led us to the conclusion that the payoff gained by implementing a generic framework for a problem domain—one like the three level structure discussed in the first design approach—is not justified in terms of the efforts. With most problems having very specific attributes, such frameworks would soon become too generic and based on loose assumptions, and thereby not really supportive. Hence, the second design approach emphasizes more on a clear definition of interfaces between DomFLIP++ and DynaFLIP++, and DomFLIP++ and OptiFLIP++ respectively.

The whole design of the new DomFLIP++ is centered around the class *CDomainInstance*. It has been designed out of a compromise between putting as many generic functions as possible into the class and still giving the knowledge engineer a high degree of freedom to implement domain specific methods and data structures. This is mainly achieved by the inheritance mechanism of the implementation language *C++*. It enables the implementation of a class that is not itself a full-functioning unit due to the abstract definition of certain methods that must be implemented according to the requirements of a domain in derived classes. The advantage is that all interfaces are predefined and other parts of the ∗FLIP++ library can rely on these interfaces. At the time of writing, OptiFLIP++ and CheckFLIP++ make use of the interface.

When an application for a specific domain is implemented, one simply creates a new class that inherits all the generic functionality of *CDomainInstance* and overwrites all abstractly defined methods. Furthermore, all domain specific data structures can be added to this derivative. Apart from this there are no further restrictions on the domain relevant design. E.g., some domains may be rather complex, so that they do not fit into a simple class and require a more complex class structure distributed among several classes. In Chapter 5 the domain dependent classes for the shift scheduling domain will be explained in more detail.

The class structure is kept quite simple consciously because this allows to keep interfaces to a minimum level. Besides, reuse is done easier with a lean class structure since it frees the programmer from the tedious traversing of a complex class hierarchy to understand the library. Moreover, generic parts should always be specialized according to the needs of the domain with the inheritance mechanism provided by the *C++* language. Of course, providing a more generic class structure has always its trade-offs. For example, how much freedom should be given to the programmer; or, how tightly defined should be interfaces. Due to the variety of domains that exist and the ∗FLIP++ concept should be open for, the more leeway the programmer has when it comes to the definition of the domain the better. Interfaces between various FLIP++ modules should be clearly defined, but still flexible. For example, after each evaluation done by DynaFLIP++ constraint evaluations are returned and can be treated by DomFLIP++ and OptiFLIP++, respectively. Although for most purposes of DomFLIP++ and OptiFLIP++ it would not make sense to use all evaluations during the optimization, it is up to these modules to decide whether to use just the most severely violated constraint evaluations, or any other subset. The interface is such that all constraint evaluations are provided in a data structure by DynaFLIP++.

3.2.3 Class structure

In the following I will give an overview of the classes of DomFLIP++. Due to the lean class structure, there are only two DomFLIP++ classes *CBase* and *CDomainInstance*. Since the optimization with existing OptiFLIP++ algorithms encountered several implementation problems (see Chapter 5 for more details) for the shift scheduling application, a separate approach was taken by the implementation of an other optimizing algorithm. This algorithm has been wrapped into the *COptimize* class, which—according to the ∗FLIP++ conventions—belongs to the OptiFLIP++ module, but is described here. The following description leaves out any unnecessary comments about trivial functions (e.g., get and set functions to initialize or retrieve attributes of a class).

CBase

Figure 3.2 shows a sketchy class definition of the *CBase* class. The reason why this class has been included in the design of DomFLIP++ stems from the experience gained throughout the implementation of ∗FLIP++. *CBase* represents a 'mother' class for all other classes. All functionality that is needed for all other classes can be put into this one. With previous ∗FLIP++ modules, some difficulties were encountered due to inconsistent implementation of the same concepts in different classes making it a very tedious task to make conceptual changes. By the introduction of an overall class that provides both concrete and abstract methods and data structures

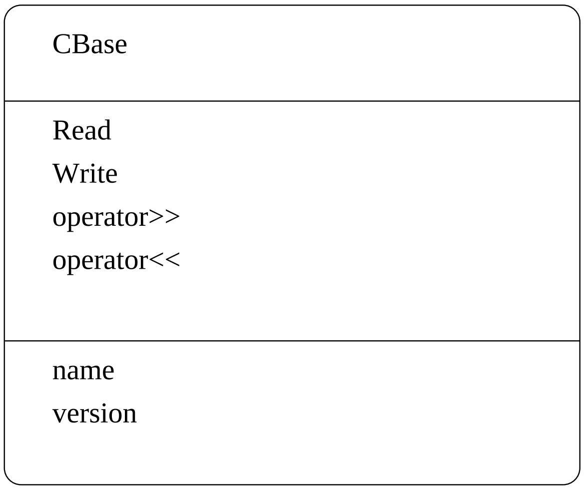

Figure 3.2: Class *CBase*.

for all classes, this weakness should be eliminated. At the moment, *CBase* is mainly focused on administrative chores like storing a name and version for every class. Furthermore, *CBase:Read* and *CBase:Write* and the corresponding stream operators *CBase:operator>>* and *CBase:operator<<* are provided by this class. These methods and operators allow to use the class for input/output operations just like conventional *C++* objects. Each class derived from *CBase* enhances this simple input/output functions according to its specific attributes.

CDomainInstance

The description in Figure 3.3 shows the methods and attributes of the class *CDomainInstance*. As already mentioned, this class provides the primary functions of DomFLIP++ that are not domain-dependent. This shows some similarity to the concept of the first design approach, where Hendrysiak distinguishes between domain-dependent and domain-independent classes. However, now these two categories are not completely separated by being represented in different classes. In contrast, the domain-independent class *CDomainInstance* is a 'mother class' of any domain-dependent class (e.g., of *CShiftSchedulingDomain* which will be explained in Chapter 5). Inheritance allows therefore to specialize domain-independent data structures, so that one arrives at domain-dependent data structures. Another difference to

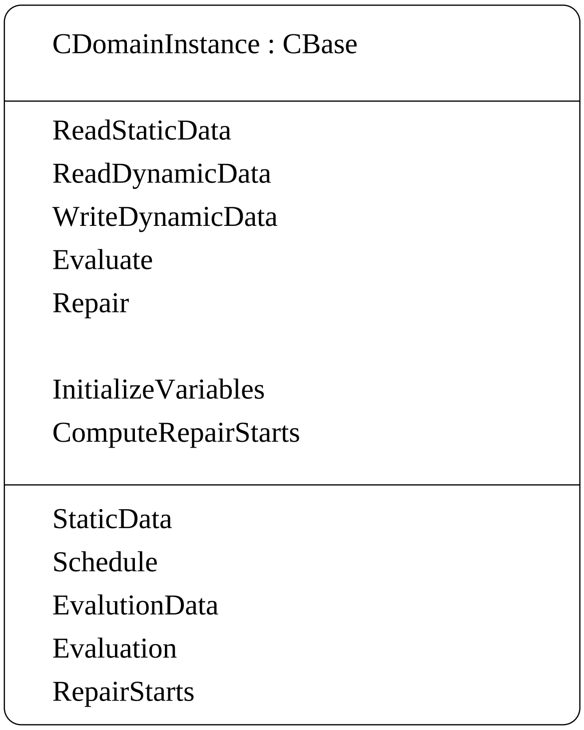

Figure 3.3: Class *CDomainInstance*.

Hendrysiak's approach is the fact that the number of classes has been reduced by moving data structures and methods into the *CDomainInstance* class or its derivatives. This reduces interfaces within DomFLIP++ classes considerably and still allows flexible and abstract definition of the data structures owing to the use of STL (see Appendix B for more details). Such data structures and methods comprise the representations of constraint evaluations, repair steps, repair start positions, and domain schedules.

Generally, the data of *CDomainInstance* is split into *static domain data* and *dynamic domain data*. *Static domain data* is represented by the constraint script file. It is read by the method *CDomainInstance:ReadStaticData* and then stored in memory in *CDomainInstance:pStaticData*. Since the manipulation of the constraint script file is ruled out within DomFLIP++, no write method is necessary and manipulations of the file are made with a plain text editor. *Dynamic domain data* comprise all domain specific data structures, hence they have to be defined in the specific inherited classes of *CDomainInstance*. However, each problem does have a schedule independent of the domain considered. The schedule is a data structure that represents the core data for optimization, e.g. shift scheduling problem have shift schedules just as a number sorting problem has a list of numbers as schedule. The attribute *CDomainInstance:schedule* stores such a schedule, the implementation of the schedule depends on the class *CSchedule* which is specified according to the requirements of the actual problem at hand. The methods *CDomainInstance:ReadDynamicData* and *CDomainInstance:WriteDynamicData* allow input/output operations with *dynamic domain data*. Of course, they have to be enhanced as more attributes are added to the *dynamic domain data* in inherited classes.

Moreover, the method *CDomainInstance:Evaluate* is an interface both between DomFLIP++ and DynaFLIP++ and between DomFLIP++ and OptiFLIP++. First, it uses the private method *CDomainInstance:InitializeVariables* to intialize CSI (= constraint script interpreter) variables with *dynamic domain data*. This point will be addressed in more detail in the section about interfaces (see Section 3.2.4). Then, the interpretation is incepted by passing on the *static domain data* and calling the interpreter of DynaFLIP++. After the evaluation has taken place, the results of the evaluated constraints are returned by DynaFLIP++. The constrain evaluations are stored in *CDomainInstance:EvaluationData* that is a sorted container of all evaluated constraints including their name, type, and evaluation score. The overall evaluation score is stored in *CDomainInstance:evaluation*. Eventually, an evaluation run is completed with the computation of possible repair starts that are derived from the constraint evaluations. The private method *CDomainInstance:ComputeRepairStarts* computes and stores the repair starts in the data structure *CDomainInstance:repairstart*. The method *CDomainInstance:Repair* is responsible for the selection of repair steps depending on a repair start position.

Any domain specific class that is derived from *CDomainInstance* has to add and/or specialize various of the above methods to provide a full functional system. It has to add the attributes for further *dynamic domain data* apart from the schedule. It must provide repair steps that allow to modify the schedule. Moreover, it has to take care of a proper intialization of the constraint script interpreter with *dynamic domain data* variables. Additionally, the computation of repair starts has to be specialized depending on the constraints of the domain. In Chapter 5 a definition of such a specialization of *CDomainInstance* will be shown with the presentation of the *CShiftSchedulingDomain* class.

COptimize

The class *COptimize* (see Figure 3.4) is already a part of OptiFLIP++. It provides an iterative improvement technique as optimizing algorithm, which will be explained in more detail in Chapter 5.

In order to manipulate on the domain data, COptimize expects a reference to *CDomainInstance* or any derivation at its inception. The methods *COptimize:Repair* and *COptimize:RepairTryDeep* provide the functionality of the algorithm. *COptimize:Repair* tries to repair a schedule after repeatedly calling the method *COptimize:RepairTryDeep*, which is the implementation of a recursive iterative deepening algorithm. *COptimize:Repair* uses intensively the methods of the *CDomainInstance* class when it starts an evaluation (*CDomainInstance:Evaluate*), calls repair steps (*CDomainInstance:Repair*), gets repair start positions (*CDomainInstance:GetRepairPositions*), and manipulates the schedule (*CDomainInstance:GetSchedule* and *CDomainInstance:SetSchedule*). Furthermore, the best score of evaluation and best evaluation schedule are stored in *COptimize:fBestEvaluation* and *COptimize:cBestSchedule*.

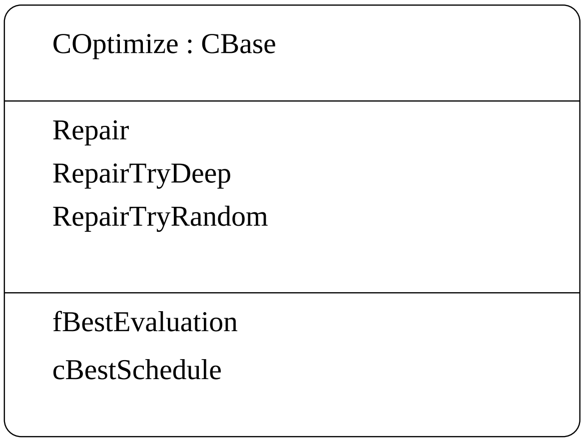

Figure 3.4: Class *COptimize*.

3.2.4 Interfaces

The interface between DomFLIP++ and DynaFLIP++ is a crucial one. The domain dependent data are provided by several data structures of DomFLIP++. To communicate with DynaFLIP++, DomFLIP++ has to extract and prepare domain data from such data structures so that DynaFLIP++ can handle it. After DynaFLIP++ has completed an evaluation of the problem instantiation, it delivers results of the constraint evaluations to DomFLIP++.

As has already been mentioned earlier, DynaFLIP++ has been designed as an interpreter that evaluates input files formulated in a constraint script language. What a domain has to deliver is twofold: On the one hand, *static domain data* are provided with a constraint script. This includes the definition of the constraints and their parameters, the aggregation of constraints, to make it short, all the operations and data that are independent of a concrete problem instantiation. On the other hand, *dynamic domain data*, those which depend on the actual instantiation of a problem (e.g., the shift schedule in shift scheduling), must also be provided properly by DomFLIP++. Both types of domain data are parameters for the DynaFLIP++ interpreter. The *static domain data* is in the actual implementation a string stream that holds the constraint script file, whereas the *dynamic domain data* are converted from the various DomFLIP++ structures into a list of CSI (Constraint Script Interpreter) variables. The variables scope of the interpreter is initialized with these variables before the constraint script of the domain (= *static domain data*) is executed. The reason why the variables are handed over directly to the interpreter are due to performance considerations. Of course, an alternative would have been to let DomFLIP++ add *dynamic domain data* to *static domain data*, and then incept the interpretation with both types of data in one aggregate parameter (e.g., a stream). This option did not prove to be successful in terms of performance. Besides, the precomputations of a problem being discussed in Chapter 5 are also an argument against that option. Furthermore, the clear separation between *static domain data* and *dynamic domain data* would have been blurred.

After the domain data have been evaluated, the interpreter delivers a data structure of constraint evaluations. Actually, this structure comprises the entire constraint hierarchy that has been constructed by DynaFLIP++ during the evaluation. DomFLIP++ uses this structure to find the most severely violated constraints. These violated constraints are then the starting points for optimization methods. However, the strategy of how to derive constraint violations from the constraint evaluations is only in the interest of DomFLIP++, DynaFLIP++ merely provides the evaluations of the constraints (including their name, type, and score).

The interface between DomFLIP++ and OptiFLIP++ is only a one-way interface. DomFLIP++ has to provide methods for OptiFLIP++ that allow to access the schedule, to evaluate a schedule, to apply repair steps, to get repair positions and constraint evaluations.

The schedule can be accessed and modified by the methods *CDomainInstance:GetSchedule* and *CDomainInstance:SetSchedule*. The method *CDomainInstance:Evaluate* allows OptiFLIP++ to evaluate a problem instantiation. Hence, the optimizing algorithm of OptiFLIP++ can use the evaluation of DynaFLIP++ via DomFLIP++. OptiFLIP++ applies repair steps via the method *CDomainInstance:Repair*, the decision which specific repair step is applied is up to this routine and depends on the position parameter. As guidance for the optimizing algorithm, DomFLIP++ provides possible repair positions for each constraint evaluation. The algorithm then has to decide which repair positions of which constraints will be used for a modification. The constraint evaluations are sorted in order of their evaluation scores with the lowest scoring constraints at the top of the list. In the same way the list of repair positions is organized providing one or multiple possible repair positions for each constraint. The method *CDomainInstance:GetRepairPositions* allows to retrieve such a position.

3.3 Implementation issues of DomFLIP++

A major contribution to the considerations concerning the implementation of DomFLIP++ have been the failures encountered with the first approach of DomFLIP++ as well as with ConFLIP++. Especially, the implementation of a library for the handling of sequence-like data structures did not prove to be advantageous. Its partly too restrictive design and buggy implementation due to a lack of testing was considered an obstacle by both the developers of DynaFLIP++ and DomFLIP++. This library has also been a factor in favor of the redesign of DynaFLIP++. Still, DynaFLIP++ does not make use of standardized tools instead.

So, when the implementation work on the current approach of DomFLIP++ has been initiated, I soon ran into similar difficulties as DynaFLIP++ had with ConFLIP++. The handling of data structures by DynaFLIP++—essential for the proper operation of DomFLIP++—evolved to be too buggy to rely on mainly due to the use of non-standardized adhoc implementations of linear lists and other types of sequence structures. In the urgent need of a hopefully bug-free, standardized tool in this area, I came across—after doing some more or less successful testing with various public domain libraries—the Standard Template Library. It unifies all the requirements DomFLIP++ needs for a proper and sound implementation: a standardized library (which is now even part of the *ANSI C++* standard), an almost bug-free tool due to its growing wide-spread popularity, and an excellent concept ideal for reuse because of its generic design. For a more detailed treatment of STL turn to Appendix B.

STL structures have been used throughout the complete DomFLIP++ library. The sequence containers of STL proved to be especially convenient because the design of many data structures of DomFLIP++, also of most ∗FLIP++ structures, lends itself for such an implementation. For example, the list of constraint evaluations is implemented as an STL vector. Also the schedule, which represents the core data structure of a problem instantiation (e.g., the shift schedule), is a matrix implemented by a two-dimensional STL vector for the shift scheduling application. Virtually every list-like data structure has been implemented in STL.

3.4 ControlFLIP++

ControlFLIP++ resides at the top end of the ∗FLIP++ hierarchy. It is designed to be the control center of all modules. ControlFLIP++ has been devised only of late due to the growing number of different modules and the strong need for coordination among them. It does not add in particular to the functionality of ∗FLIP++ by providing a new module. ControlFLIP++ is merely focused on the interaction of the various components by providing an interface for the user.

While striving for a more sophisticated interface, where the user can rely on such convenient features as graphical representation of domain schedules and optimization process, the current version of ControlFLIP++ provides a pure text-based user interface. It allows to read, write, and display domain data including both schedule and constraints. Furthermore, the optimization is the main function. Several parameters considering the optimization can be specified. Among the optimizing algorithm parameters are for example, maximum time dedicated for optimization, maximum depth for iterative deepening algorithm, number of constraint violations considered, etc.