DIPLOMARBEIT

Implementation of a Shift Scheduling Application using the *FLIP++ Environment

ausgeführt am Institut für Informationssysteme
Abteilung für Datenbanken und Expertensysteme
der Technischen Universität Wien

unter Anleitung von
o.Univ.Prof. Dr. Georg Gottlob
und Univ.Ass. Dr. Wolfgang Slany
als verantwortlich mitwirkendem Universitätsassistenten

durch
Andreas Raggl
Trankhütte 238
A-6426 Roppen

Wien, im Januar 1997 ....

To my parents, for their invaluable love and continuous support.

Abstract

DomFLIP++ is the knowledge representation module of the $*$FLIP++ (pronounce: StarFlipPlusPlus) project. $*$FLIP++ is a tool that is focused on optimizing multiple criteria problems. It uses fuzzy constraints to model optimizing criteria and applies various iterative improvement techniques such as Tabu search, genetic algorithms, and iterative deepening to the problems. DomFLIP++ is implemented in the C++ programming language making intensive use of the Standard Template Library. It allows the definition of new optimization problems by aiding the domain engineer in the design of the structure of a new problem at hand. Generally, a division between domain-dependent and domain-independent methods and data structures characterizes the structure of DomFLIP++. While the domain-dependent data structures are specific to the problem, the domain-independent part is provided as a framework by the library. Moreover, there is a domain-independent interface to other $*$FLIP++ modules such as OptiFLIP++, DynaFLIP++, and CheckFLIP++. After each iteration in the optimization process, the considered instantiations of the problem are evaluated. Each evaluation produces a list of evaluated constraints and hence provides hints on violations of requirements. For each constraint, modification operators, so-called repair steps, are defined that can be used to increase the score of the constraint in further iterations of the optimization. A domain can be fine-tuned through modifying of constraints and their fuzzy representation, changing the choice of repair steps, and varying optimizing parameters. A well-tuned domain can be successfully applied for the optimization. The shift scheduling domain presented in this thesis is a fruitful area of investigating the power of DomFLIP++. This is due to its variety of constraints, inherent fuzziness of requirements, and large search space that recommends the application of heuristics.

Acknowledgments

Writing this dedication is more than a pleasure to me. It is one small way in which I can give a nod of both appreciation and admiration to the people who have helped me along the way.

First is my family. I would like to especially thank my parents, Irene and Engelbert Raggl, for their invaluable support and love. To my sister Alexandra and her family, Heinz, Arko, and Falk. Thank you all for supporting me through the years, especially during the time away from home.

Second is the colleagues and friends that I have made throughout my studies at the university. I hope you all will not take it too seriously that I locked myself away for three months to accomplish this. Special thanks to Christian Blaas, Martin Maierhofer, Andreas Rauber, and Sascha Schmid who joined my way right from the beginning. Thanks also go to my friends Gunter Aha, Oliver Gabriel, Heinz Habermuth, and Johannes Köll.

Third is my best online and internet friends with special thanks to Julie Albright for your invaluable electronic messages. Thank you.

Fourth is the ∗FLIP++ team. Special thanks to Wolfgang Slany who earned my respect as a great and enthusiastic person keeping my interest in the area more than just alive. Also, many thanks to you Wolfgang for the unique opportunity to take part in the IJCAI'95 and EUFIT'96 conferences which has been an unforgettable experience for me. To Markus Bonner, Georg Fischer, Johannes Hajek, Witold Hendrysiak, Stefan Mayer, Michael Prah, Ulrich Santa, Karl Schott, Christian Schwaiger, Simone Ulreich, Mazen Younes, and all the others that are currently following my footsteps for the patience to continue my work. Thank you all for being so enjoyable to work with.

Finally, I would like to thank the members and former co-workers of the Database and Expert Systems Group of the Department for Information Systems at the Vienna University of Technology. Special thanks to Gerald Ehritz, Thomas Eiter, Ilse Eppel, José Ezquerra, Heidi Milos, Gerhard Nesvadba, Katrin Seyr, Markus Stumptner, and Therese Zahradnik. I also want to thank Prof.Georg Gottlob, Head of the Department and the Christian Doppler Research Laboratory for Expert Systems, for his support and the technical equipment I could use as both a member of the department and a student. Also, many thanks to my new employer for allowing me to take my time off to complete this work and for preserving my position in the mean time.

To those not mentioned here, but who have also helped me in my pursuits—thank you.

Contents

List of Figures

Chapter 1

Introduction

Begin at the beginning . . . and go on till you come to the end: then stop.

—Alice's Adventures in Wonderland

1.1 Organization of the text

Although the topic of the thesis focuses on a quite specific part of the ∗FLIP++ project called DomFLIP++, throughout the evolution of this text the author felt that there were various other parts that needed to be covered. As a result, this thesis does not merely concentrate on its core part that is on DomFLIP++ related issues. Subsequently, various inputs were brought together to put the work into a wider context where the author tries to give a perspective of the full-scale project. Out of this objective the second chapter and parts of the third chapter were written. The rest of the thesis predominantly concentrates on the above mentioned central part, which comprises DomFLIP++ on the one side and an application residing in the shift scheduling domain and making use of DomFLIP++ on the other side.

To begin with, Chapter 2 is meant as an introductory chapter. It introduces the concepts and ideas that are behind the ∗FLIP++ project. It also emphasizes the role of DomFLIP++. Furthermore, it takes a closer look at the parts that are directly linked with DomFLIP++, namely DynaFLIP++ and OptiFLIP++.

Chapter 3 contains a detailed description of DomFLIP++. This includes both design issues and details on the implementation level. As an extension, a previous design approach is taken into special consideration. In this part the author tries to track down possible overlaps between the current DomFLIP++ structure and the first design approach. Furthermore, explanations of differences between the two design approaches are given.

In Chapter 4, the focus turns from the system, i.e. DomFLIP++, to the application domain. An extensive treatment of the shift scheduling domain is given, which the author considers as a necessary premise for a complete and thorough understanding of the following chapter.

Chapter 5 deals with the shift scheduling application itself. It shows how the various characteristics of the domain that have been pointed out in Chapter 4 can be transformed into the DomFLIP++ environment. Moreover, a discussion of the empirical evidence provided by the system leads to further insights into the possibilities as well as the boundaries of the system.

The conclusion summarizes the key points of the thesis and provides some hints on possible future extensions.

1.2 Production of the thesis

The entire thesis was typeset by the author on both a PC and a workstation using LaTeX, the wonderful typesetting system designed by Leslie Lamport. LaTeX gives the author complete control over the structure and appearance of a document; it is especially convenient for text involving mathematic formulae. Therefore, it is quite popular in the computer science community. It derives its origin from its equally prominent 'mother' product TeX.

For the coding part, I did my software work in parallel on two platforms. First, I used the *Microsoft™ Visual C++ Version 4.2* environment within the *Microsoft™ Windows 95* operating system. Second, I coded with *GNU C Compiler Version 2.7.2* on both a *Linux (Kernel 2.0.0)* platform on a PC and a *Sun™ Solaris 2.5.1* environment on the various *Sun™* workstations at the Department for Information Systems of the Vienna University of Technology. Having formulated cross-platform development as an outspoken aim of this work, it was an almost undeniable necessity to conform to the *ANSI C++ Standard* as rigorously as possible; sometimes for the sake of efficiency on the respective platform. Such nice things as the Standard Template Library did a great job in facilitating my work and in fulfilling this objective.

I also used standard unix tools like `emacs` for editing both source code and LaTeX files. Furthermore, a version control tool called `cvs` helped never to lose the thread and overview on the many versions of source code and LaTeX files I produced throughout the entire work.

Chapter 2

Description of *FLIP++

The following text gives an overview of the *FLIP++ project. It puts the system and application developed with this thesis into a context. After shortly touching upon the entire *FLIP++ project, I concentrate on those parts that are most relevant for the DomFLIP++ environment, namely FLIP++, DynaFLIP++, and OptiFLIP++. CheckFLIP++ which also relies on the functionality of DomFLIP++ and is still under construction at the time of writing is being discussed only briefly (a more detailed description will hopefully be available when you read this in [17]).

2.1 The project

2.1.1 Introduction

*FLIP++ is a library [20, 21] for real world decision-making. It is a tool for optimization under vague constraints of different importance using uncertain data. Through the use of fuzzy computations, compromises between antagonistic criteria can be modeled. Typical application areas include scheduling, design, configuration, planning, and classification.

Up to now all libraries have been implemented in the object-oriented language *C++*, which was the obvious choice at the start of the project. Today, with more appealing programming languages and object-oriented concepts having reached a more mature and stable level, there is a larger variety of options available as far as the implementation is concerned. Hence, the *JAVA* programming language with such convenient standard features like networking classes implying full Internet connectivity and a high degree of platform independence is the first choice for future *FLIP++ implementations. Nevertheless, *C++* is still a well-justified environment especially with such powerful extensions like the Standard Template Library, which will be discussed in more detail in Appendix B.

Although throughout the major part of the development time dedicated to *FLIP++ a production scheduling problem in a steel production plant has been the key area of application, the design of the library has never been explicitly biased towards a certain application problem. Due to this fact, an open system evolved that can treat a large variety of problems with shift scheduling being just one of them.

2.1.2 Overview

$*$FLIP++ is composed of the following layered sub-libraries:

- FLIP++: the basic fuzzy logic inference processor library.
- ConFLIP++: the static fuzzy constraint library which has been incorporated in DynaFLIP++ only recently.
- DynaFLIP++: the dynamic fuzzy constraint generation library.
- DomFLIP++: the domain knowledge representation library.
- OptiFLIP++: the 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 genetic algorithm heuristic.
- CheckFLIP++: the knowledge-change consistency checker library that also allows fine-tuning of the configuration parameters of a problem.
- InterFLIP++: the graphical user interface for all other libraries.
- DocuFLIP++: the on-line documentation available separately for end-users, knowledge-engineers, and programmers, and accessible via the World-Wide-Web as HTML documents.

  Furthermore, the following parts are currently under development or they are considered to be candidates for future development in the case of ReaFLIP++ and SimFLIP++:

- ReaFLIP++: the reactive optimizer as an extension of DomFLIP++.

- TestFLIP++: the version control and test environment for the complete library set.

- SimFLIP++: the simulation toolkit library.

- JavaFLIP++: a major reuse/redesign of the existing ∗FLIP++ libraries including an implementation in the *JAVA* programming language.

Figure 2.1 shows a view of the layered structure of ∗FLIP++.

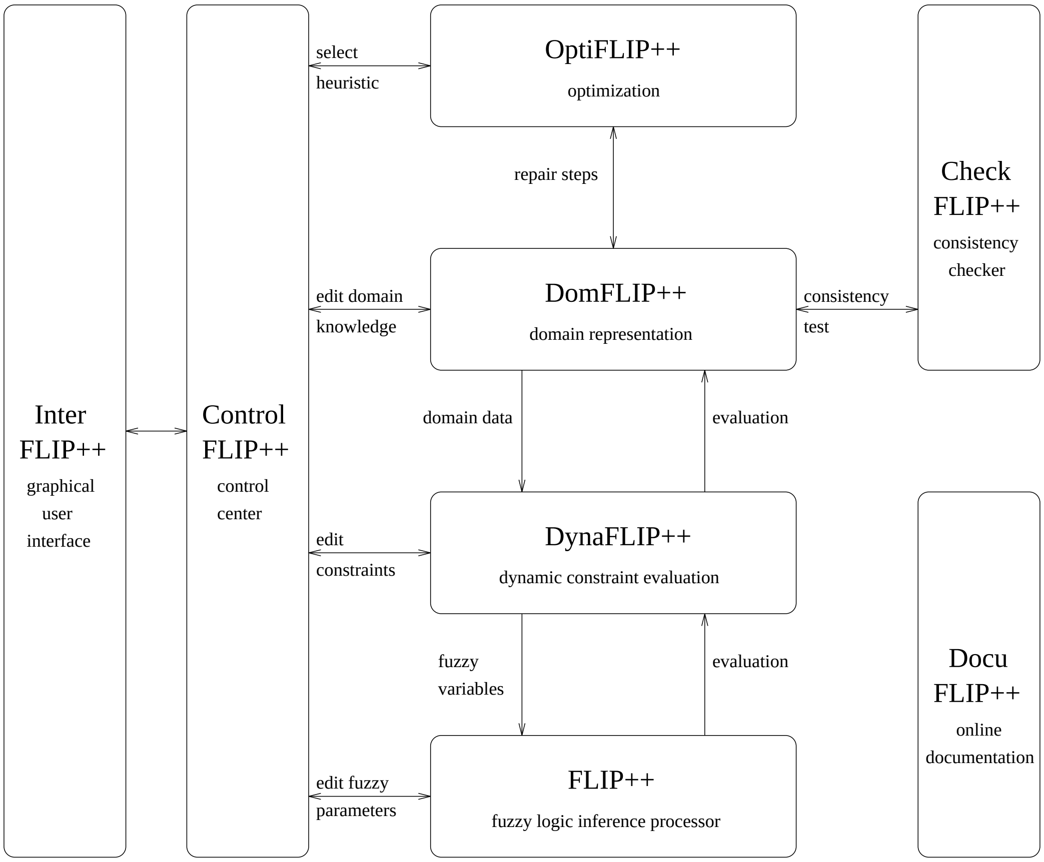

Figure 2.1: Structure of ∗FLIP++.

2.2 FLIP++

FLIP++ was the first part that has been developed in the ∗FLIP++ project. It is a general purpose fuzzy logic inference processor (hence the name FLIP). It is considered to be the basic level of the layered ∗FLIP++ hierarchy responsible for doing all the fuzzy computations. It provides functions for the representation of a fuzzy system. It allows to define linguistic variables and linguistic terms including their underlying mathematical representations called membership functions. Furthermore, various fuzzy operators are available to aggregate variables. A rule system using defined linguistic variables can be formulated. The fuzzy computations done by the inference machine are then based on the rule system. Results can be obtained in both fuzzified and defuzzified form.

Since fuzzy computations involve several operators for prioritizing rules, aggregating results, and also for the defuzzification, it is normally not very easy to follow the behavior of the system according to the manipulation of the various parameters without the aid of visualization. Therefore, a graphical representation is a very convenient way that allows the user to modify membership functions of linguistic variables, rule sets, and many other parameters in an intuitive manner. FLIP++ supplies this feature using its 'companion' InterFLIP++.

The way fuzzy rules are formulated is most easily illustrated with a short example given below:

```
IF temperature IS hot THEN cooling IS high WITH CERTAINTY 0.7
```

Here the input linguistic variable *temperature* is used to define the value of the output linguistic variable *cooling*. Of course, there have to exist many more rules of a similar pattern in order to define a specific output value for all possible input values. The certainty value introduces a weighting indicating how strongly the user believes in this rule. Furthermore, the shape of the membership functions of the various terms (e.g., *hot*) is critical in terms of how fuzzy the computation will be. All fuzzy computations on a higher level of ∗FLIP++ rely on the evaluation of a system of such rules. For example, the constraints that are used in DynaFLIP++ are evaluated by the generation of a system of corresponding FLIP++ rules.

The library has been designed in a very generic way so that add-on developments are straightforward. Adding more complex fuzzy operators is rather simple and does require only minor modifications. Currently, FLIP++ supports various fuzzy operators such as t-norms, t-conorms, gamma operators, and associative compensatory operators. Moreover, the inference machine does its computations hierarchically following the inherent dependencies of the formulated rules. Starting from the rules of the output variable the algorithm evaluates recursively rule by rule until it ends up with values for all input variables needed.

Modularity has been a key design objective. Consequently, FLIP++ allows to evaluate several fuzzy systems independently. Afterwards, a more complex system can be built by just feeding the input values with the evaluations of various sub-systems. The advantage is that the user can decide which computations do not have to be redone after each modification and move them into a sub-system, thereby avoiding unnecessary lengthy evaluations. Not to mention that it is in most cases more convenient to divide a complex problem into various sub-problems. This point will be met again in the next section when we discuss the creation of a constraint tree which deals with modularity on the constraint level.

For a much more in-depth treatment on the theoretical background and the basic concepts of fuzzy decision making as well as the modeling of fuzzy constraints—both of them had a major impact on the design of FLIP++—refer to [20]. A more detailed description of the FLIP++ libraries can be found in [1].

2.3 DynaFLIP++

As the main purpose of ∗FLIP++ is to optimize problems with the use of vague fuzzy constraints, there is a strong need for a tool that aids the creation of such constraints. DynaFLIP++ fulfills this task by the provision of a constraint script interpreter. This interpreter renders it possible to write scripts of fuzzy constraints like usual programs. Although being a prototype, it comes with various features which are common in usual programming languages. Certainly, even if the interpreter was sufficient for the representation of our applications, there is no doubt that larger and more complex problems will still need the power of a full scale programming language.

2.3.1 ConFLIP++

Historically, before the development of DynaFLIP++ was taken into consideration another tool for constraint handling had been already in use. This module was called ConFLIP++. It mainly supported the modeling of static constraints, i.e., constraints with well defined parameters before runtime.

Constraints are based on the underlying structure provided by FLIP++. They use linguistic variables, sets of parameters, fuzzy operators, and fuzzy rule systems, which have been already mentioned in the description of the FLIP++ library (see Section 2.2). Basically, a constraint consists of a linguistic variable, a value, and a relation operator that connects the variable and the value. E.g.,

temperature_constraint:

```
temperature <= 20
```

is a constraint defined on the linguistic variable *temperature*. The fuzziness of the constraint is twofold. First, the linguistic variable has of course a fuzzy representation with its various terms and their respective membership functions. Second, the way the relation operator is used, is not in its strict mathematical sense. Depending on the particular definition of a fuzzy comparison table for the variable involved, the relation can be more or less violated. Hence, the modeling of soft constraints, i.e., requirements that have not be fulfilled in a binary 'yes/no' manner, is possible. This type of constraint is a comparison constraint. In order to aggregate the evaluations of several comparison constraints, ConFLIP++ provides a further type of constraint. The concatenation constraint allows the aggregation of two comparison constraints. With this constraint a hierarchy of constraints can be built in an easy way. When creating a hierarchy of constraints the choice of the aggregation operator should not be neglected. Only an associative operator provides the functionality to aggregate constraints in any order and still to arrive at the same overall evaluation results. An example for an aggregation constraint is

overall_constraint:
`temperature_constraint AND pressure_constraint`

where the two partial constraints are either also a concatenation constraint or the basic comparison constraint. Figure 2.2 shows the tree-like structure of a constraint hierarchy.

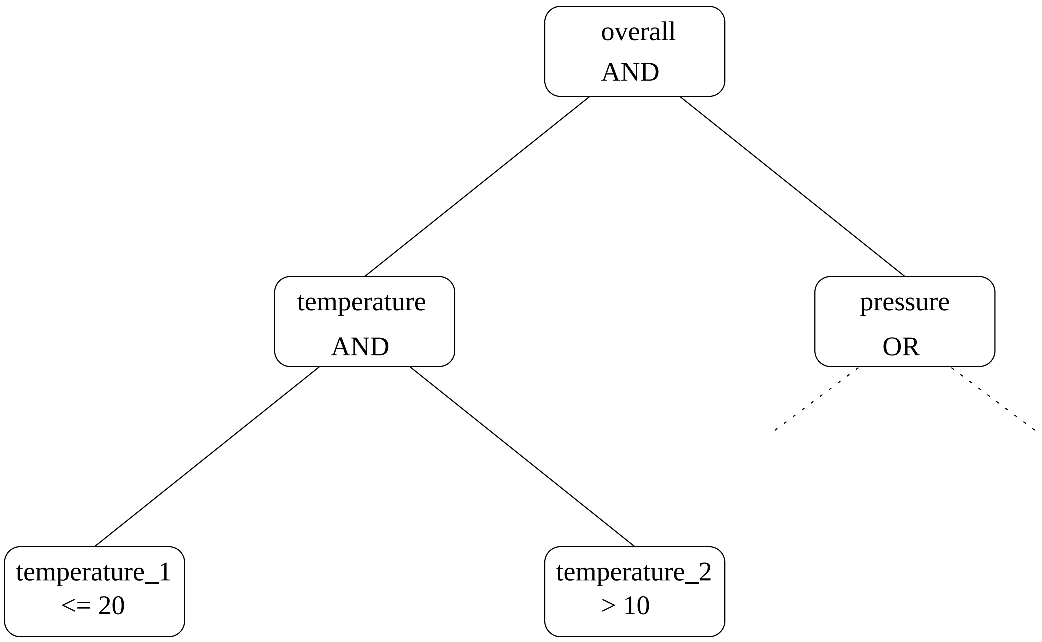

Figure 2.2: Structure of a constraint hierarchy in ConFLIP++.

To ease the creation of constraints, ConFLIP++ and FLIP++ provide standard definitions taking over the burden of defining each parameter by hand before the constraint is evaluated. Hence, the user normally starts with the standard definitions provided by the system (e.g., membership functions of triangular shape, standard comparison tables for concatenation constraints), then moves on to step-by-step fine-tuning of the application until attaining satisfying results.

2.3.2 What makes DynaFLIP++ dynamic

In the previous section, we dealt only with static constraints. With ConFLIP++ there is no possibility to let the system create constraints dynamically at runtime with varying parameters based on a template constraint. This is where the real dynamic part of DynaFLIP++ comes into play. Initially, the idea has been to enhance *FLIP++ with another level residing above ConFLIP++ in the hierarchy. Then, after the first steps have been taken towards that direction, a major redesign caused ConFLIP++ to be merged in DynaFLIP++.

The reason why static constraints are not sufficient comes from our experience with real world optimization problems. Santa (see [19]) finds that with different problems, namely steel production scheduling and classroom rostering, virtually the same disadvantage of ConFLIP++ due to its static nature was revealed. Although static constraints do a good job when modeling aspects associated with partial satisfaction of constraints, compromising constraints, and relative importance of constraints, they fail when it comes to dynamic areas. He also reasons that sometimes the actual number and the type of constraints is linked with the current instantiation of a problem. Since any modification during the optimization process is a change of such an instantiation, the structure of the constraint evaluation tree has to change, too. Consequently, DynaFLIP++ relies on the definition of template constraints that have to be formulated and tuned by the knowledge engineer. According to the requirements of the problem instantiation at hand, DynaFLIP++ then uses these templates to create specialized constraints.

Since the dynamic nature of constraints requires a lot of computations at runtime that are based on the instantiation of a problem, DynaFLIP++ needs powerful aggregation functions to achieve that. Furthermore, DynaFLIP++ is totally independent of the domain knowledge. Still, the generation of dynamic constraints is tightly coupled with the extraction of domain knowledge. Hence, the approach of how to come up with a solution that satisfies all these requirements has been a critical one. Finally, the decision was to implement a prototype interpreter. This interpreter should allow a knowledge engineer to do both: using all the functionality of DynaFLIP++ and applying domain knowledge where necessary to formulate dynamic constraints.

2.3.3 CSI

Accordingly, Santa [19] proceeded with the design of the prototype constraint script interpreter (or short CSI). Although being still considered to be a prototype, CSI evolves to be a quite powerful tool for the formulation of a problem using fuzzy constraints. Of course, the performance of the script processing is lacking a bit of performance power, which is not an uncommon effect when using interpreted languages. Furthermore, the multilevel architecture of ∗FLIP++ may also drag performance down to a lower level.

Conceptually, with the formulation of a problem in a constraint script, DynaFLIP++ handles such a script in the following way: All information about the problem domain that has to be considered in the run of an evaluation has to be put into the constraint script. This is more or less the same as an interface to DomFLIP++. It remains in the responsibility of how DomFLIP++ handles the knowledge representation. Only if it connects to DynaFLIP++ in order to do an evaluation of a problem instantiation it must feed all domain values including the problem instantiation into such a constraint script file. This is mainly done by the initialization of CSI variables. The definition of constraints is the core part of every constraint script file. It comprises everything from the specification of the linguistic variables encountered in FLIP++ to the formulation of dynamic constraints that depend on the characteristics of the problem instantiation at hand. Control structures and commands known from common programming languages like *Pascal* and *Basic* ease the programming of constraints. The constraint evaluation proceeds in the same manner as described in the ConFLIP++ section (see Section 2.3.1). A tree structure is built that finally delivers an overall evaluation score. This evaluation score is again at the interface between DomFLIP++ and DynaFLIP++. Furthermore, for the purposes of DomFLIP++ it is necessary to retrieve the constraint evaluations as they are needed for the optimization process, which will be explained later on.

What is still a major source for improvement is the way how DynaFLIP++ handles sequences of evaluations where only minor parts of a problem instantiation have been modified. We strongly believe that there is a great potential for performance improvement when parts of the constraint evaluation tree are preserved between subsequent evaluations, thus avoiding unnecessary recomputations. Although, this should not be at the center of this thesis, I will return to this subject in the performance evaluation of the system in Chapter 5. Also, the concepts of CSI will be met again when the constraint script file of the shift scheduling application is discussed in Chapter 5. For a thorough discussion of CSI including a tutorial chapter turn to [19].

2.4 OptiFLIP++

OptiFLIP++ provides the functionality that is needed for optimization of an application problem. It uses the evaluation capabilities of DynaFLIP++ to grade an instantiation of a problem. Moreover, it relies on the domain representation functions of DomFLIP++ to access and modify a problem instantiation. It has been designed on a highly generic level allowing it to connect to various applications very easily.

Due to complexity reasons of realistic scheduling applications, the ∗FLIP++ project has never striven for an optimization method that is based on classical enumeration techniques guaranteeing an optimal solution. Investigations in the field of complexity theory show that many problems, especially real world applications, represent intractable problems. Search spaces are almost always enormous pushing the computational capabilities of a state-of-the-art computer very soon to the limits if an enumerative approach is pursued. Hence, $*$FLIP++, or to be more specific OptiFLIP++, takes a more compromising approach as a problem solving method. The compromise is between the feasibility of a solution of the problem and the amount of time it takes to arrive at the solution. The method concentrates on the iterative improvement of a preliminary solution of a problem. Heuristics are tightly coupled with this approach. They decide which type of modification should be chosen based on the evaluation results of a solution. One appeal of an iterative improvement technique are the simple inexpensive modifications, which are called repair steps in the OptiFLIP++ terminology. As a second advantage, the quality of the preliminary initial solution is not a decisive factor when it comes to finding a good solution. Moreover, satisfying solutions are found in a relatively short time due to the 'anytime' feature. 'Anytime' means that the algorithm can be stopped after each iteration and it is up to the user to decide whether he wants to continue with the iterative improvement loop or the result is already a satisfying one.

Nonetheless, there remain still some questions with the above description that have to be addressed. First, one has to decide which strategy is followed for the modifications. Often a *MinConflicts heuristic* is suggested. It tries to start modifications at the position where the greatest conflict occurred. Minton et al. [16] showed that this technique leads to convincing results, especially with large and complex problems. Second, the possibility to get stuck in a local maximum using hill-climbing search is still there. This inconvenience can be overcome with the introduction of random techniques. They often show superior behavior because a more or less randomized jump within the search space can escape local maxima. Also, the inclusion of a random factor makes an algorithm generally simpler in terms of computation time. Another technique to prevent the optimizing algorithm from getting trapped in a local optimum is to make certain restrictions on the modifications applied, so that endless loops of identical sequences of modifications cannot be entered.

A more exhaustive discussion of the iterative improvement techniques including a comparison of four different approaches can be found in [5]. There, the authors used production scheduling in a steel production plant as a sample application. OptiFLIP++ has so far produced two different strategies. The first one is based on a tabu search hill-climbing strategy, whereas the second one is a genetic algorithm. During the implementation of the shift scheduling application, I introduced a third algorithm that is based on iterative deepening. It will be explained in more detail in Chapter 5.

To summarize, the optimization relies on the following parts:

- an evaluation function: this is the task of DynaFLIP++ via interpreting a CSI script.
- a set of modification (repair) operators: these operators are definitely dependent on the problem in hand. They have to be defined as part of the problem representation in DomFLIP++.
- a procedure to select a modification: again, this is mainly a question of how the knowledge engineer entrusted with the representation of the problem would apply the steps. Consequently, this is a part of DomFLIP++.
- the overall control algorithm: Meaning one of the various iterative improvement techniques realized within OptiFLIP++.

2.5 CheckFLIP++

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. All consistent changes of a problem instantiation are handled by CheckFLIP++ in a database.

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.

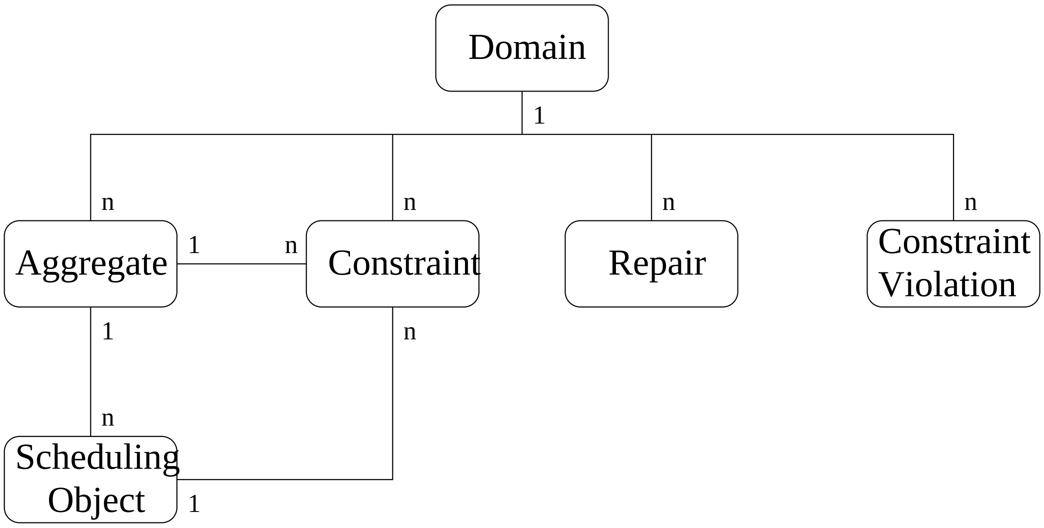

Figure 3.1: Class structure of the first approach of DomFLIP++

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

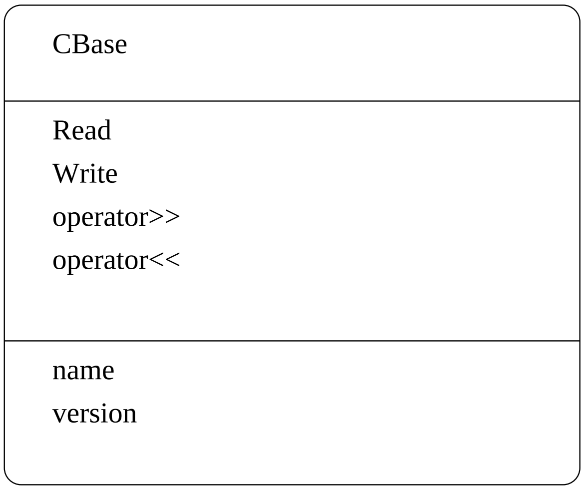

Figure 3.2: Class *CBase*.

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

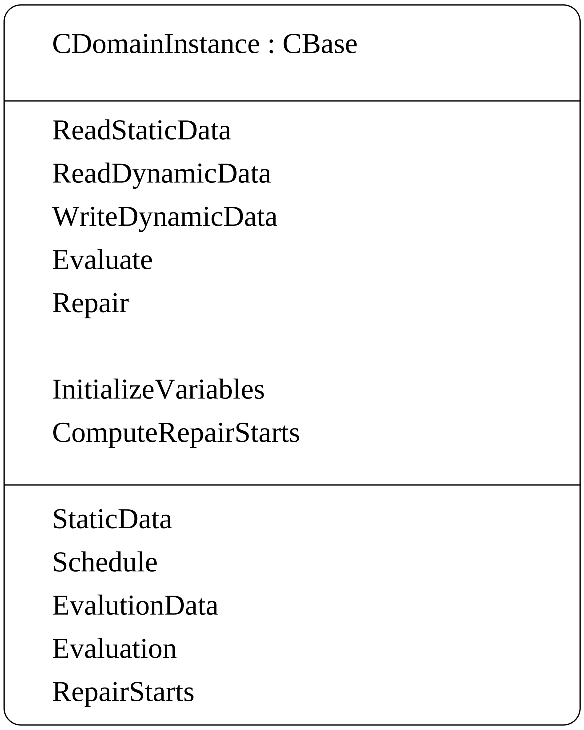

Figure 3.3: Class *CDomainInstance*.

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*.

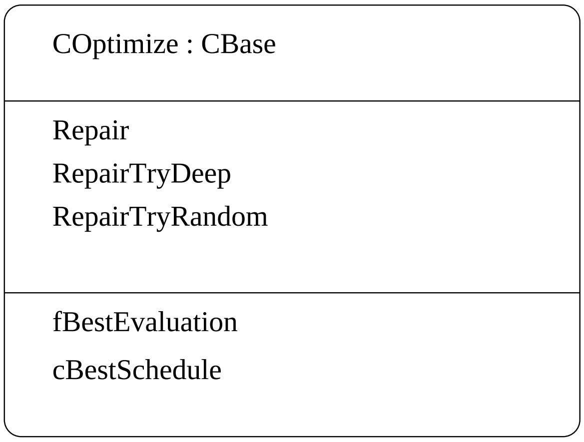

Figure 3.4: Class *COptimize*.

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 ad-hoc 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.

Chapter 4

Introduction to the Shift Scheduling Domain

"I like work . . . I can sit and watch it for hours."

—unknown

In this chapter an explanation of the domain of shift scheduling is given. The introduction shows why shift scheduling can be quite a problem worth being further examined. Next, several definitions concentrate on the various key terms of this specific domain that are used throughout this and the next chapter. Furthermore, an excursus to the field beyond the technical implications of the problem domain shows how fast the domain is growing, turning it into a 'foggy swamp' search space and making optimization more and more difficult. Nevertheless, the section on the requirements gives various hints on what to focus on when we try to find the constraints of our problem domain. The chapter concludes with an examination of the optimization issues in this problem domain.

4.1 Introduction

Right from its beginning the ∗FLIP++ project has always been strongly coupled with problems encountered in the process of steel production. This was partly due to the fact that the entire project has been initiated by a research cooperation with the Austrian steel production industry. A wide range of publications have been published on the steel production domain over the last couple of years out of this fruitful research cooperation. In [20] a more in-depth discussion of this domain is given in connection with fuzzy scheduling. Moreover, Dorn and Shams [4] discuss an expert system approach designed for this domain.

With versatility being a key objective of the ∗FLIP++ project, next choice was to move away from the original steel production domain. The advantages should be twofold: Firstly, we expect to gain further experience about the process of knowledge extraction and transformation into a ∗FLIP++ compatible format, which should lead us eventually to a system that is a quite generic tool as far as the representation of domain knowledge is concerned. Secondly, another problem domain may also give us numerous hints on weaknesses of the system. These weaknesses may be localized in the choice of optimization method, in the performance of the system, and in the representational power provided by the various fuzzy tools of the FLIP++

library.

The choice in favor of the domain of shift scheduling was a promising one out of various reasons. To begin with, there has been quite a lot of research in this area only recently. One major conclusion drawn out of these research efforts is the fact that one runs rather soon into major difficulties in this area when conventional optimization methods are applied, e.g. with simplex, enumeration or backtracking methods (for a more in-depth discussion on this particular aspect turn to [7]). This is always a good starting point for the introduction of a fuzzy constraint satisfaction system, one like ∗FLIP++. And indeed, as we will see later on, shift scheduling is a good ground for such an approach. According to Gärtner and Wahl [8] a high degree of fuzziness can be attributed to many requirements encountered with a shift scheduling problem. They also argue that due to the complexity of the problem and the lack of powerful optimization methods it is more important to move the focus from automation of design towards aiding design. The project team also believes that ∗FLIP++ should be a more supportive tool that allows to find an optimal schedule via the interaction with the knowledge engineer. Nevertheless, optimism is still there and ∗FLIP++ strives to eventually accomplish a system that comes close to an automation of design. Moreover, the flexibility offered by ∗FLIP++ when it comes to the definition of fuzzy variables, fuzzy constraints and aggregation operators should make it superior compared to classical optimization methods which often show a lack of representational power. Moreover, the repair-based optimization process that tries to tackle constraint violations with specifically defined repair steps seems to be more promising than classical approaches.

Furthermore, the problem of developing 'good' shift schedules is a highly practical application. It also has a lot of consequences on the people that are involved and concerned which makes it even more interesting. These sometimes more social than technical implications will be discussed in more detail in Section 4.3. Although my prototype solution does not include the social requirements in full scale, I believe that the excursus is well-justified. It investigates whether a future enhancement of the application towards this direction can be done efficiently; a further probation of the representational power of ∗FLIP++.

4.2 Definitions

In the following several important definitions are given. We concentrate on the key terms that are used throughout this and the next chapter. Some illustrations should try to ease the understanding of the various definitions.

Shift

A *shift* defines duties (e.g., morning shift from 6 a.m. to 2 p.m.). Shifts name a certain time interval at one or more specific days of the week where operation is required. Of course, it is possible to have different shifts for different days of the week. For example, many shift scheduling problems require particular shifts at the weekend. When taking a decision on the structure of a shift, one has to consider various aspects, such as length of operation time, breaks, overlapping.

Figure 4.1 shows a graphical definition of various shift types.

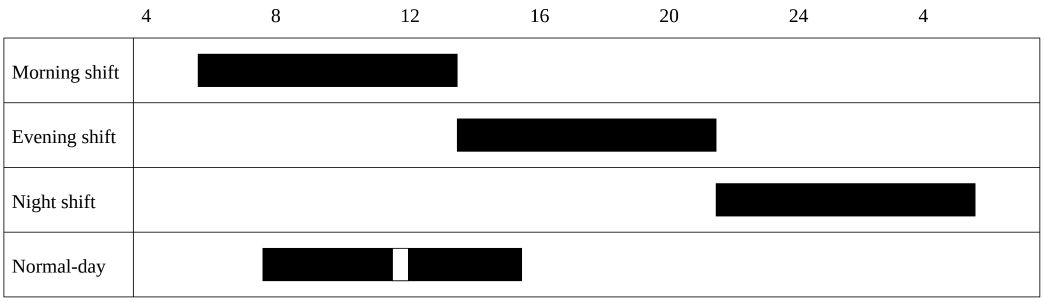

Figure 4.1: Shifts

Groups/Subgroups

It is very common, especially in the shift schedules of production plants to aggregate employees in functional and/or organizational units. These units are simply called *groups*. One can also think of a certain hierarchy of grouping: with groups at the top level and below *subgroups* as further partitions of these groups. At the bottom level comes a single employee. At each level it is known how many employees are needed to form the respective group. As will be shown in the next chapter, our problem uses two levels, namely group and subgroup. Of course, this particular restriction is only due to the specific application problem discussed in this thesis and is not at all a necessity.

Roster

A *roster* defines which employees have to work on which shift on a specific day. The requirements for the operation are the primary guideline for the design of a roster. If it is known on which days of the week and at what times of the day how many employees are required for operation, it is possible to allocate an appropriate shift and group at that time. For example, on Monday two groups in the morning shift have to be available in order to make sure a sufficient level of operation. Figure 4.2 is an example for a roster with a length of 50 days and 5 groups. It allocates the shifts in clusters of 6 shifts and 4 days off using a rotation pattern.

DAY	1	2	3	4	5	6	7	8	9	10
Group 1	M	M	E	E	N	N				
Group 2			M	M	E	E	N	N		
Group 3					M	M	E	E	N	N
Group 4	N	N					M	M	E	E
Group 5	E	E	N	N					M	M

Figure 4.2: Roster with the length 50 days (M. . . Morning shift, E. . . Evening shift, N. . . Night shift)

Generally, we can expect rosters of a more complex nature than the above one. Then, the description of the shift-sequence may not be as easy as with a rotation cycle. Furthermore, an increased number of both groups and group levels introduces a further degree of complexity. It is also quite common and sometimes necessary depending on the nature of the problem at hand to define rosters of a longer duration (e.g., 48 weeks).

Shift schedule

With the previous definitions, we can now take a further step and define the term *shift schedule* as an aggregated view. Shift schedules fix the distribution of working hours of employees. They are primarily defined by shifts and rosters.

Time horizon

The *time horizon* of the shift scheduling process consists normally of at least three layers. At the *day* level the position and internal structure of shifts have to be defined. This corresponds to the definition of a shift given above. The *week* level consists of rosters that fix the nominal work force per day in the period of a week. Over the period of a *year* the nominal and actual work force have to be balanced. Several sources can cause the differences between nominal and actual work force. Among them are higher levels of production, further education, holidays, sickness leave, etc. They are mainly handled by overtime or stand-by staff (included as an extra group/subgroup in the design process).

4.3 Beyond the technical issue

The social implications of shift work arrangements are at least as important as the more technical issue of finding an appropriate shift schedule that fits into the constraints model of operation.

The idea of replacing shift work by individual flexibilization of working time is challenged by various arguments. Especially in the service sector, where consumption and production normally coincide, shift work is an undeniable instrument for work arrangements (e.g., in hospitals). Also, the tendency towards longer opening hours in the commerce sector weakens the correspondence between company working hours and individual working hours, i.e., work can be shifted more into the evening hours or morning hours, or it can be even split into several shifts.

When designing a shift schedule, the organizational environment has to be analyzed first. It is a serious mistake to separate the pure scheduling problem of shift scheduling from its context. Successful shift schedule design can only proceed with a cooperative strategy where the computer system supports a flexible construction of a solution as well as its analysis and evaluation. Furthermore, the system should be based on requirements that represent an aggregate view of all groups and interests concerned.

The following list shows some of the possible effects on both employees and employers caused by the introduction of a new shift schedule:

- income and costs
- the position of leisure time
- physical stress (sleep, relaxation, etc.)
- mental stress (workload, restrictiveness of the shift schedule, etc.)
- social consequences concerning family life, friends, social, cultural, and sporting activities
- working hours of the company

Among the groups that are mainly concerned by shift work and its changes are:

- employees directly affected by the shift schedule
- personnel managers, shop-stewards
- craftsmen and foremen in responsibility of detailed work assignments
- employees entrusted with short-term staff assignments (holidays, illness, ...)
- departments indirectly affected by the shift schedule
  - maintenance and service

  - bus service, porters, medical service
  - controllers

- interest groups outside the company
  - major customers
  - public transport
  - employment offices
  - trade unions and employers' organizations

Of course, there is not only a one way impact of shift schedules on these groups. The influence comes also from the other direction and is a major factor in the design of a shift schedule. For example, in an industry where the various production stages are done in a vertical line by different companies, a just-in-time concept introduced by one powerful company can force the companies in previous stages, its suppliers, to adopt the same shift schedule system. Also, the role of trade unions and employers' organizations is in many industries a strong one based on a long tradition.

The large variety of different interests involved in the design process has a strong impact. Each violation encountered by a shift schedule has to be handled according to the strength of the interest and group involved. Even a shift schedule that may be of high quality in technical terms, may not be so favorable if it violates the social interests of an important group (e.g., a union). Besides, the introduction of a shift schedule cannot repeated again and again since switching from one schedule to another is quite costly and does not happen from one day to another. At the other end of the spectrum is the fact that time resources are limited. For those entrusted with the design and implementation of a new shift schedule it is fairly difficult to discuss extensively various schedules with both employees and employers as the normal day-to-day duties are an obstacle. It is simply not possible to take some time off and devote it exclusively to the planning process. Relying on the services of consultants may be too costly and increases the risk of failure due to communication problems between company staff and the consultants from outside.

The involvement of many groups of often diverging interests makes the design process of new shift schedules a sensitive task. Gärtner and Wahl [8] call the domain therefore a 'swamp'. Moreover, the inherent fuzziness of social requirements and the difficulty to grade and formalize them turns this swamp into a 'foggy' one. Eventually, the time resources allocated to shift schedule design are—as with most design problems—rather scarce. Using the metaphor once again, in order to arrive at a good shift schedule at the end of the design process in time one has to 'run in a foggy swamp'.

4.4 Requirements

The requirements that have to be taken into account during the design process of shift scheduling can be quite enormous. First, there are mainly technical ergonomic requirements which should be satisfied by a good shift schedule. Then, legal restrictions in combination with collective agreements put a shift schedule into a more or less restrictive framework. Finally, the social requirements discussed in the previous section have to be taken into consideration. Bearing in mind the various groups and interests mentioned earlier in connection with the non-technical requirements, it is fairly easy to imagine a rather high overall number of constraints encountered in shift scheduling. In [8] an estimation finds "200–300 or even more requirements" that have to be met under rather optimistic assumptions. Many of these requirements lead to conflicts making the initial scenario even worse. Although the high number of requirements reduces a fairly large solution space (to get a grasp of the amount of possible solutions, just imagine the possible combinations of shift allocations in a period of a year with several different shift types), they can lead to search spaces with few admissible solutions.

Whereas the more technical requirements, such as ergonomic and legal restrictions, can be modeled formally, even if sometimes in a tricky way, the social requirements do not lend themselves for being modeled in a formal language due to their individual and often subjective nature. Consequently, I will concentrate more on the technical requirements in the next chapter and neglect the issue of representing non-technical constraints. The interpretation of a computer evaluated shift schedule is hence up to the knowledge engineer or domain expert and takes place outside the computer system. Although a rather difficult task, the representation of non-technical, subjective constraints within the supportive computer system should be an area of future investigation. Especially with the arrival of more powerful systems and machines, it remains to be seen how soon this task can be completed.

discussion of possible constraints

4.5 Optimization

The aim of optimization is to find good schedules within the pool of possible solutions. When speaking of optimization, we also have to draw attention to the evaluation of a shift schedule. Without some sort of evaluation function a grading of shift schedules is not possible. Generally, evaluations can be:

- Global/Local

  Global evaluations consider the whole schedule and provide an overall score. A local evaluation regards only parts of a schedule and hence provides hints on problem areas and improvement.

- Overall/Features

  Overall evaluations provide an overall score for a shift schedule or part of it. A feature evaluation concentrates on a specific feature of a shift schedule (e.g., average night work per month). Both types can be global or local.

The part of ∗FLIP++ responsible for evaluation is DynaFLIP++. It evaluates constraints that form a hierarchy and provides, in principle, all of the above mentioned types of evaluation. The optimization in ∗FLIP++ is predominantly guided by an overall, global evaluation score. As hints on improvement strategies constraint violations are derived from both local and global feature evaluations.

Two important aspects considering evaluations are:

- how well does the evaluation handle antagonistic behavior of contraints
- are different constraints treated differently (e.g., by the use of a relative importance scheme)

Due to the complex nature of shift scheduling environments with many conflicting and ill-defined constraints, it is rather difficult to rely solely on a computer-based evaluation. The evaluation by the user is still needed and even absolutely necessary if the computer system models only the technical requirements of the shift scheduling problem. A continuous fine-tuning and adaptation of requirements is necessary in the design cycle. Especially, bearing in mind the risk of overlooking requirements or using wrong, ill-defined constraints.

The choice of optimizing algorithm is mainly influenced by the size of the solution space. Classical optimization methods like simplex, enumeration, or backtracking techniques show deficiencies since they work their way through a huge search space without guidance. The application of heuristics and repair-based improvement techniques appears to be far more promising, particularly in connection with a strong evaluation tool like DynaFLIP++. With the application presented in the next chapter, I opted for an iterative deepening strategy using repair based improvement. Using a genetic algorithm would also be well-justified, but was neglected due to performance issues (the number of evaluations is considerably higher leading to unsatisfying computation times with the current implementation of DynaFLIP++).

Chapter 5

The Shift Scheduling Application

5.1 Introduction

This chapter gives an explanation of the shift scheduling application and how it is embedded in the ∗FLIP++ environment. A detailed description of the problem to solve makes the reader acquainted with the subject-matter. The discussion of the problem also includes some precomputations and premises necessary for the solution stage. Then, the approach taken towards a solution is explained step by step. First, the constraints that have been taken into consideration are discussed. Second, the repair steps representing the modification operators on the problem instantiation are described. Third, we take a look at the optimization method applied and how it connects to the problem instantiation. Finally, the whole application would be worthless without the revelation of new insights and results which are presented at the end of the chapter. Before, we proceed with the description of the problem some comments will be given on issues that will not be included or only slightly touched upon in this discussion. A side-effect of the shift scheduling application is that it should ultimately provide connectivity to a visual design tool for shift schedules that is already in use. The communication between these two systems follows a standard specification for shift schedules. The design of domain specific classes is prepared for the application of this standard. However, the discussion of this more technical detail should not be a part of this chapter. Moreover, the generation of an initial solution, which is an absolute necessity for the way ∗FLIP++ works on the problem, is an optimization problem of its own. Therefore I will only show in brief how such an optimization could take place, while the actual implementation relies on a manually produced initial problem instantiation. The initial instantiation does also satisfy a variety of hard constraints which are subsequently not taken into consideration in the formulation of ∗FLIP++ constraints. Furthermore, due to the implementation problems with DynaFLIP++ and OptiFLIP++, I deviated from the original intention of solely using the functionality of these modules, which resulted in some functions that per definitionem actually belong to DynaFLIP++ being moved to DomFLIP++ mainly out of performance reasons. This concentrates mainly on the interpreted constraint script file where some computations have been moved to DomFLIP++ in order to work around some slow interpreted functions of DynaFLIP++.

5.2 Problem description

Admittedly, the shift scheduling problem taken into consideration for this thesis has been cut short to some degree in order to arrive at a solution rather quickly. In particular, the number of represented constraints could be reduced by focusing on a problem instantiation with quite simple shift types. For example, the use of a night shift has been completely neglected. Consequently, all constraints referring to night shifts could be left out. Nevertheless, restricting ourselves to a narrower definition of the problem does not reveal limits of our system. Once a proper representation of a problem is found, enhancing the constraints of the problem does not cause any difficulty at all.

The objective of our problem is to find a shift schedule for twelve employees. These employees are aggregated in three groups with an even distribution, i.e., each group consists of four employees. The groups are further divided into subgroups of two employees each. Each of the subgroups is fully covering the requirements of operation, hence no interdependencies between the various subgroups have to be taken into consideration. Ruling out interdependencies is a further simplification that is rather rare in real world problems. Again, as mentioned above, such a simplification does not hurt the judgement of the basic functionality of ∗FLIP++ in connection with a shift scheduling problem. For ease of use, the groups are named *A*, *B*, and *C* with subgroups *A1*, *A2*, *B1*, *B2*, *C1*, and *C2*. The working hours are 38.5 hours on a weekly basis. Weekly working hours can vary over the length of the shift schedule, but on average the 38.5 hours are the limit. Shifts of 8–9 hours (variable) and 12 hours have breaks of 30 minutes, whereas shifts of 4 hours do not involve any breaks. The 30 minutes break of the variable shift is calculated as 20 minutes unpaid and 10 minutes paid work. The 30 minutes break of the 12 hours shift is considered to be wholly unpaid. There are three different types of shifts defined in Figure 5.1.

The most common shift type is the *day shift*, this can also be seen from the roster presented next. The length of this shift is also considered to be flexible, i.e., it can be anywhere between 8 and 9 hours for the whole shift schedule.

This provides some leeway concerning the working hours requirements.

name	length (in hours)	shortcut
day shift:	8–9	(TD)
day shift at weekends:	4	(TDWE)
shift substitution at weekends:	12	(SWWE)

Figure 5.1: Shift type definitions.

The roster is defined in Figure 5.2 and displays the requirements of operation. It can be easily detected that the shifts required for operation remain the same week by week with only one exception. On every third Saturday of the shift schedule a different requirement is given due to maintenance work. Because of this the cycle of the operation plan is set to three weeks.

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Week 1	5 SG TD	5 SG TD	5 SG TD	2 G TD	2 G TD	1 SG TDWE + 1 G SWWE	1 SG TDWE
Week 2	5 SG TD	5 SG TD	5 SG TD	2 G TD	2 G TD	1 SG TDWE + 1 G SWWE	1 SG TDWE
Week 3	5 SG TD	5 SG TD	5 SG TD	2 G TD	2 G TD	2 G SWWE	1 SG TDWE

SG...Subgroup, G...Group
TD...day shift, TDWE...day shift weekend, SWWE...shift substitution weekend

Figure 5.2: Operation plan.

So far, the various parameters of the problem have been given. Before we proceed with the description of the solution for this specific problem, some sort of precomputations has to be done. This is necessary due to the following reason: The optimization methods applied in the ∗FLIP++ environment are all dependent on an initial solution. It does not really matter whether the quality of the solution is good or bad, as several studies have already shown (e.g., in [5]). As the automatic generation of valid initial solutions has not been part of the objective of this thesis, the system relies on the provision of a preliminary problem instantiation (= initial solution) from outside. So it is up to the knowledge engineer to provide such a solution. This is where the precomputations come into play. They determine various parameters of the problem instantiation:

- length of the shift schedule cycle
- number of shifts allocated per employee/subgroup/group during the entire shift schedule cycle
- length of the various shift types

The first two parameters are needed to generate a valid initial solution. The length of the shift schedule cycle remains constant throughout the optimization. Including such a parameter as a variable in the scope of the optimization process would lead to an enormous search space. It determines the size of a problem instantiation. The length of the cycle is derived from the cycle of operation and the number of minimal units of operation. In our specific problem the cycle of operation is three weeks which can be verified in Figure 5.2. The minimal unit of operation is a subgroup, and there exist six of them. Hence it follows that the shift schedule cycle must have a length that is equal to 18 weeks, because this is the minimal multiple of 6 and 3 that allows to rotate the operation cycle for each subgroup. Once a shift schedule of this length is found, it can be applied in consecutive sequences and the working hours of all subgroups and the employees respectively are the same over the length of one cycle.

In order to generate an initial problem instantiation, it is necessary to define how many shifts of each shift type have to be allocated for a shift schedule of the above computed cycle length while still satisfying the 'hard' requirements of operating hours and average working hours per week. The computation proceeds as follows:

According to the definitions of the shifts the 12 hours shift corresponds to 11.5 hours of working time, the 4 hours shift corresponds to 4 hours working time, and the *day shift* is variable, 10 minutes of the break are counted as working time. The operation plan tells us what amount of shifts is required for an entire 18 week shift schedule and the subgroups, respectively. Figure 5.3 displays the shift requirements for our problem.

Actually, it is not necessary to break down the figures of required shifts on a per employee basis since the smallest unit of operation is a subgroup.

shift type	entire shift schedule	subgroup
TD	414	69
TDW	30	5
SWWE	48	8

Figure 5.3: Shift requirements.

Now that we have the number of shifts that has to be allocated for each shift type and operation unit over the length of the entire shift schedule cycle, we can determine the value for the variable *day shift* length.

Equation 5.1 simply equates the number of hours worked by adding up the products of shift length and number of shifts applied on the left side with the number working hours needed, 18 weeks times the average working hours per week (38.5). The variable x stands for the length of the variable *day shift*. After the unpaid break time of 20 minutes has been added, the result for the gross shift length is a time of approximately 8 hours 45 minutes.

$$69\times x+5\times 4+8\times 11.5 = 18\times 38.5 \Rightarrow x = 8.42+0.3\dot{3} = 8.75\dot{3} \approx 8\,h\,45\,m \quad (5.1)$$

The computation that has been done for the length of the *day shift* is interesting since the shift length is an important input for constraints on working hours.

In summary, the precomputations done here are in combination with the generation of an initial problem instantiation a nice example of a scheduling problem itself. The problem is to find an initial solution that satisfies the hard constraints of operating hours and average working hours per week. The requirement of operating hours is satisfied by proper modification operators (repair steps) that move each shift type only to well-defined positions of the shift schedule, e.g. a *day shift* is only possible from Monday to Friday with at most five subgroups allocated. The requirement of average working hours has been resolved by the above computations and is subsequently inherently satisfied by providing each shift type in restricted numbers. For each unit of operation there exists a pool that holds the numbers of shifts that are available for the unit. In our example this would require six identical pools each referring to a subgroup and its shift requirements (69 *day shifts*, 5 *day shifts at weekends*, and 8 *shift substitutions at weekends*). An optimizing algorithm could now start to allocate shift after shift from a pool of shifts to the shift schedule. A solution is reached when the pool of shifts empty, i.e. all shifts have been allocated to the shift schedule, and hence none of the requirements is violated.

5.3 Solution

First, I want to clarify that the solution of the above described problem is not solely focused on the actual, optimal problem instantiation. The notion is to be comprehended in a wider context. With solution, I mean as well the process of knowledge transformation that takes places when the problem is formulated in a computer readable format after an intensive consultation of a human domain expert, in this case using the various representation mechanisms provided modules of $*$FLIP++. This task is a crucial one, since most failures are made by misinterpretations or leaving out parts of domain knowledge. Experience shows that this activity is a more gradual one requiring permanently shifting back and forth between communication with the domain expert and the actual design phase; by the way, a practice that should be observed with any serious software development project. Actually, the representation results in constraints that are formulated in a constraint script file and several classes and data structures that hold the dynamic domain data (for a discussion of *dynamic domain data* and *static domain data* refer to Section 3.2.3). Both forms of representations will be shown in the following sections.

Coming back to the actual solution, repair steps and the method of optimization are the key aspects in the discussion of how to arrive at a 'better' solution. The term 'better' in this sense is a subjective one, it relies on the definition of the constraints. The constraints are more or less the guidelines that tell the optimization process through their evaluation scores what kind of problem instantiations is preferred. Hence, the entire notion of a solution is subjective; it depends on how constraints are defined by the human expert. If there are ill-defined constraints they may hinder the optimizing algorithm in making its path towards a 'better' solution. Also, two many requirements may lead to an 'over-constrained' problem that is intractable in terms of optimization.

5.3.1 Constraints

The constraints define certain requirements of the problem domain. Normally, constraints are of dynamic nature, i.e., their concrete instantiations depend on the instantiation of the problem. In our application this means that different shift schedules lead to different constraint instantiations of the same type of constraint. Hence it is necessary to define constraints in a language-like style that is based on the use of variables. Each evaluation run of DynaFLIP++ is based on variables whose values are fed by the problem instantiation of DomFLIP++. Initially, I started to compute all the dynamic variables of the constraints in the constraint script file where the constraints are also defined. Unfortunately, the constraint script interpreter did not prove to be very strong in performance. As a consequence, I moved all the computations that depend on the *dynamic domain data* into the *CShiftSchedulingDomain* class resulting in superior performance mainly due to the use of compiled *C++* methods. However, the constraint script file is still necessary because it uses the computed variables to initialize the corresponding fuzzy variables and fuzzy constraints, aggregates the constraints into hierarchy, and finally evaluates the constraints. In the following I will shortly explain some of the constraints that have been implemented.

Constraint WorkingHours

The average working hours per week are fixed for our problem due to the precomputations that restrict the allocation of shifts per subgroup and hence per employee. Still, it is of course possible that over the shift schedule cycle the weekly working hours deviate from the average working hours. A rule of thumb tells us that an employee should not work more than 50 hours per week. This can be easily formulated as a constraint. For each week and each subgroup of the schedule the amount of working hours in computed with a method of the *CShiftSchedulingDomain* class. Then, CSI (= constraint script interpreter) variables (actually a matrix for the entire shift schedule) are initialized before the evaluation takes place. There, a fuzzy variable is initialized with the value and subsequently used in a simple compare constraint. All the constraints of this type are then aggregated in the constraints hierarchy and finally evaluated. Figure 5.4 shows the

```
// definition of operator sets
...
// definition of parameter sets
...
// definition of fuzzy variables
...
// definition of constraint hierarchy
...

FOR i = 1 TO NumberOfGroups STEP 1
   FOR j = 1 TO LengthOfCycle STEP 1
      // initialize fuzzy variable with computed working hours
      SET workinghours = c_workinghours[i][j]
      // formulate constraint
      FUZZYCOMPARE c_workinghours IS workinghours < 50
      // evaluate constraint
      EVAL c_workinghours
      // put constraint into branch ‘workinghours’ of
      // constraint hierarchy
      AGGREGATE workinghours AND c_workinghours group[i].week[j]
   NEXT j
NEXT i
```

Figure 5.4: CSI section of constraint ‘workinghours’.

corresponding CSI section.

In the CSI section, the rather lengthy definitions for the fuzzy variables, the parameter sets, the fuzzy operators, the compare table for fuzzy comparisons, and the constraint hierarchy are not shown. They are of course an important instrument when it comes to fine-tuning of constraints. A large variety of parameters allows to influence the way a constraint is evaluated.

Constraint of even distribution of working hours

This constraint tries to guide the evaluation towards smooth shift schedules, meaning that deviations between weekly working hours should be relatively small. It computes the difference in working hours between consecutive weeks and detects those pairs of weeks where the difference is rather high. This should lead to a more evenly distributed sequence of shifts and hence working hours.

Constraint to avoid shift sequences

Shift scheduling research has shown that shift sequences with six and more shifts on consecutive days should be avoided. After several consecutive shifts rest periods get more and more important. If night shifts are also part of the operation plan, than the acceptable length of shift sequences is reduced further. During a scan of the shift schedule blocks of sequences and their lengths are detected. The constraint will then lead to bad evaluations for shift sequences of six and more shifts.

Constraint for weekends

Weekends are a crucial area with most shift scheduling problems. Often, a human expert has to pay especially attention to this area of the shift schedule. In our application, two different constraints are concerned with the evaluation of weekends. First, one constraint simply checks the number of free weekends for one subgroup and leads to better evaluations the more free weekends there are. Second, another constraint tries to avoid the use of shift allocations on consecutive weekends for the same subgroup, since this is another rule of thumb a shift scheduling expert would take into consideration.

5.3.2 Class structure

CShiftSchedulingDomain

The class *CShiftSchedulingDomain* (see Figure 5.5 for a sketchy description) encompasses the functionality that is necessary to represent the problem described above. Furthermore, it provides the means for optimization. The class is derived from the class *CDomainInstance* and hence inherits all the methods and attributes from this class. *CDomainInstance* has been already discussed in Chapter 3. *CShiftSchedulingDomain* specializes *CDomainInstance* so that the particulars of the shift scheduling domain can be represented.

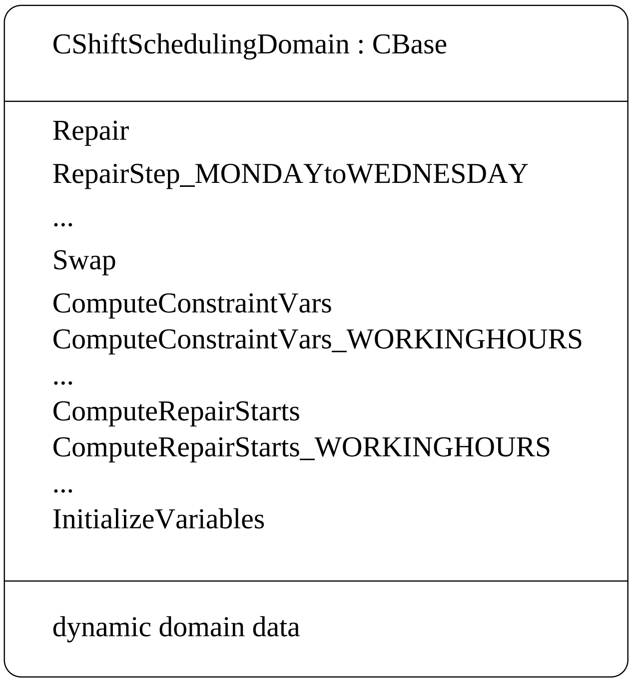

Figure 5.5: Class *CShiftSchedulingDomain*.

The class redefines the method *CShiftSchedulingDomain:Repair* that chooses, depending on the position in the schedule, a repair step. The repair steps are implemented in the methods *CShiftSchedulingDomain:RepairStepX*, where X stands for the number of the repair step. In our problem we have actually four different types of repair steps. They will be discussed in more detail in Section 5.3.4. Basically, all repair steps can be reduced to a simple swap operation on the schedule which is implemented in the private method *CShiftSchedulingDomain:Swap*.

The method *CShiftSchedulingDomain:ComputeConstraintVars* is responsible for the preparation of variables that are needed by the constraint script file. It computes for each constraint all the variables for the current instantiation of the problem. Due to the number of constraints, it is recommended to write a separate method for each Constraint. For example, the values of the variables of the constraint WORKINGHOURS are computed in *CShiftSchedulingDomain:ComputeConstraintVars_WORKINGHOURS*. These variables are then used in the method *CShiftSchedulingDomain:InitializeVariables*. This method transforms the variables into CSI (= constraint script interpreter) variables and initializes the interpreter properly before executing the constraint script for the evaluation of the problem instantiation.

The *CShiftSchedulingDomain:ComputeRepairStarts* method derives possible start positions for the repair based optimizing algorithms from the evaluations of the constraints. Since each constraint has its particular definition that leads to quite specific derivations of repair start recommendations, the method relies again on the services of private methods for each constraint. For example, the repair start recommendations for the constraint WORKINGHOURS are computed in the method *CShiftScheduling-Domain:ComputeRepairStarts_WORKINGHOURS*.

Furthermore, as with all classes there exist methods for input/output of the whole set of domain data which comprises *static domain data* and *dynamic domain data*; a concept already explained in Chapter 3.

Among the class attributes is the inherited *schedule*. A shift schedule can be easily represented as a matrix of shift allocations with the dimensions unit of operation (= subgroup in our example application) and day respectively. Therefore, a matrix of integers, which identify the allocated shift type, has been chosen for the implementation of the *CSchedule* class that determines the type of the *schedule*. In addition, all domain relevant attributes reside in the *CShiftSchedulingDomain* class; just to number a few of them, the length of the operation plan, the length of the shift schedule, amount of working hours per week, etc. Moreover, there are references to containers of classes that represent shifts, groups/subgroups, and the operation plan. These classes are briefly explained below.

CShift

The class *CShift* is a straightforward implementation of a shift type. It holds attributes like name and length of the shift (e.g., *day shift* of 8 hours). Besides, it contains information about how many shifts of this type are required for the entire shift schedule, which is determined in the already discussed precomputations of the problem. All shifts are aggregated in a sequence that is an attribute of *CShiftSchedulingDomain*.

CGroup

The units of operation, mainly groups and subgroups, are represented in the class *CGroup*. It stores attributes such as the group name (e.g., A1), the number of employees in the group, the level in the aggregation hierarchy (e.g., a group that contains subgroups). Similar to *CShift*, groups are aggregated in a sequence that is referred to by *CShiftSchedulingDomain*.

COperation

The class *COperation* provides the requirements of the roster, i.e., how many shifts of a certain shift type are required on a specific day (e.g., five subgroups with *day shifts* on Monday). An aggregation of such classes allows to represent the operation plan in a compact form. Such a sequence of *COperation* classes is again used by *CShiftSchedulingDomain*.

5.3.3 Data structure

The concept of most data structures that can be identified both in connection with the shift scheduling problem and DomFLIP++ in general lends itself for the application of sequence-like implementations. Deviating from the previous implementations of ∗FLIP++ modules, also different to first version of DomFLIP++, I waived the use of linear lists or similar structures. Subsequently, all data structures used throughout DomFLIP++ have been implemented by the use of sequence containers provided by STL (see Appendix B for more information about STL).

The class *CSchedule* that represents the shift schedule is represented as a matrix of integers, which is in STL terms a vector of vectors of integers. The integers identify the allocated shift type, the position in the matrix determines the concerned day and subgroup, respectively.

The repair starts, evaluation data, groups and shifts are all aggregated by the use of the vectors. The obvious advantage is that all these structures are treated consistently in the same way facilitating implementation tremendously. Furthermore, a plethora of algorithms is supplied by STL that makes the solution of many detail problems much more convenient allowing the programmer to concentrate on the real issues of the application.

5.3.4 Repair steps

Repair steps represent the modification operators of a problem. They allow to move from one valid instantiation of the problem to an other. Such modifications are the basis for every optimizing algorithm that tries to arrive at—in terms of evaluation—superior problem instantiations by modifying the actual problem instantiation.

Generally, it is always possible to identify certain types of repair steps for a problem. In our example, the definition of repair steps is almost entirely dictated by the plan of operation, i.e. the roster. With the use of the roster (see Figure 5.2) it is possible to define for each day what type of shift and how many of it must be allocated. Due to performance reasons, repair steps should be preferably kept simple since they are used quite heavily during the optimization process. Mostly, the modifications for a problem can be broken down to simple swap or move operations. This has also been shown for the application in the steel production domain in [12].

Repair steps represent modifications of the shift schedule. Without any guidance these modifications would take place randomly. To avoid that, the positions where constraint violations are identified are used as input for repair steps. This can be any position in the shift schedule. As we will see in the next paragraph, in our problem the position in the shift schedule will also determine which repair step is applied. In our application, repair steps take the position as an input and then try to find a possible modification while iterating through the shift schedule beginning from a start position. The indication of a start position allows some leeway in the application of the repair step. One is not restricted concerning the start of the search for a successful modification. The start position could be a random one, or the same as the position of the violation, or whatever seems to be appropriate for the problem at hand.

Taking a closer look at the roster of our shift scheduling problem shown in Figure 5.2, one can identify four main clusters where operations might be the same within that cluster. From Monday to Friday the operation is identical and there is only one shift type involved. Since Thursday and Friday do have a more complex operation plan, as they use different operation units, namely groups, we have to split this cluster into two: One modification operator will be defined for modifications based on Monday to Wednesday, another one will be defined for those based on Thursday to Friday. The third modification operator is focused on changes that involve the *day shift at weekends* on Saturday and Sunday. Finally, a repair step for the *shift substitutions at weekends* on Saturday is defined. In the following the four different types of repair steps will be explained.

Repair step 'Monday–Wednesday'

The operation plan shows that on the days from Monday to Wednesday there is only one shift type required, the *day shift*. If a violation of a constraint and its derived repair start position is within this range, there are two ways of applying a modification operator.

First, the position of the violation, which is characterized by the day and the subgroup (= smallest unit of operation in our problem) affected, can be swapped with another position of the range Monday to Wednesday. Actually, two swaps have to be made in order to preserve the requirements of the operation plan. The reason is straightforward: From Monday to Wednesday five subgroups are allocated with *day shifts* resulting in one subgroup without a shift. If the violation is on a position where a *day shift* is allocated, a swap is sought with a position in the same subgroup where no shift is allocated. Because the number of shifts allocated to subgroup is not affected by such a swap this hard constraint is preserved. Furthermore, by making a second opposite swap the hard constraint for the required shifts at one day is preserved, too. Both swaps involve an exchange of a free subgroup, i.e., one that has no shift allocated, with a *day shift* subgroup. The operation is illustrated in Figure 5.6.

Second, since the *day shift* can also be found on Thursday and Friday, it is also possible to seek for swaps destinations within this range. There is only one thing that has to be considered in addition to the above swap operation, on Thursday and Friday groups are required. Consequently, a swap operation must take care of this and preserve the group structure of

DAY	1	2	3	4	5	6	7	8	9	10
Subgroup A1	TD	TD	TD	TD				TD	TD	TD
Subgroup A2	TD	TD	TD	TD		TDWE			TD	TD
Subgroup B1	TD	TD	TD	TD	TD			TD	TD	
Subgroup B2				TD	TD		TDWE	TD		TD
Subgroup C1	TD	TD	TD		TD	SWWE		TD	TD	TD
Subgroup C2	TD	TD	TD		TD	SWWE		TD	TD	TD

Violation at (Subgroup A2, Day 1)

Swap 1: (Subgroup A2, Day 1) - (Subgroup A2, Day 8)

Swap 2: (Subgroup B2, Day 1) - (Subgroup B2, Day 8)

Figure 5.6: Repair step 1 for Monday to Wednesday.

the operation requirements on Thursday and Friday. This results in a more complex two-step modification described in Figure 5.7.

DAY	1	2	3	4	5	6	7	8	9	10
Subgroup A1	TD	TD	TD	TD				TD	TD	TD
Subgroup A2	TD	TD	TD	TD		TDWE			TD	TD
Subgroup B1	TD	TD	TD	TD	TD			TD	TD	
Subgroup B2				TD	TD		TDWE	TD		TD
Subgroup C1	TD	TD	TD		TD	SWWE		TD	TD	TD
Subgroup C2	TD	TD	TD		TD	SWWE		TD	TD	TD

Violation at (Subgroup A2, Day 1)

Swap 1: (Subgroup A2, Day 1) - (Subgroup A2, Day 5)

Swap 2: (Subgroup B2, Day 1) - (Subgroup B2, Day 5)

Swap 3: (Subgroup A1, Day 5) - (Subgroup A1, Day 10)

Swap 4: (Subgroup B1, Day 5) - (Subgroup B1, Day 10)

Figure 5.7: Repair step 2 for Monday to Wednesday.

Repair step 'Thursday/Friday'

The repair step of the Thursday to Friday range is similar to the one described above. Again, there are two variants of the modification operator.

First, a swap from the range Thursday to Friday to itself is made. Because the operation on these days works solely with groups, the swaps have to be effected on the group level to preserve this structure. Figure 5.8 illustrates the modification.

DAY	1	2	3	4	5	6	7	8	9	10
Subgroup A1	TD	TD	TD	TD				TD	TD	TD
Subgroup A2	TD	TD	TD	TD		TDWE			TD	TD
Subgroup B1	TD	TD	TD	TD	TD			TD	TD	
Subgroup B2				TD	TD		TDWE	TD		TD
Subgroup C1	TD	TD	TD		TD	SWWE		TD	TD	TD
Subgroup C2	TD	TD	TD		TD	SWWE		TD	TD	TD

Violation at (Subgroup A2, Day 4)

Swap 1: (Subgroup A1, Day 4) - (Subgroup A1, Day 5)

Swap 2: (Subgroup A2, Day 4) - (Subgroup A2, Day 5)

Swap 3: (Subgroup C1, Day 4) - (Subgroup C1, Day 5)

Swap 4: (Subgroup C2, Day 4) - (Subgroup C2, Day 5)

Figure 5.8: Repair step 1 for Thursday to Friday.

DAY	1	2	3	4	5	6	7	8	9	10
Subgroup A1	TD	TD	TD	TD				TD	TD	TD
Subgroup A2	TD	TD	TD	TD		TDWE			TD	TD
Subgroup B1	TD	TD	TD	TD	TD			TD	TD	
Subgroup B2				TD	TD		TDWE	TD		TD
Subgroup C1	TD	TD	TD		TD	SWWE		TD	TD	TD
Subgroup C2	TD	TD	TD		TD	SWWE		TD	TD	TD

Violation at (Subgroup B2, Day 4)

Swap 1: (Subgroup B1, Day 4) - (Subgroup B1, Day 10)

Swap 2: (Subgroup B2, Day 4) - (Subgroup B2, Day 9)

Swap 3: (Subgroup C1, Day 4) - (Subgroup C1, Day 10)

Swap 4: (Subgroup C2, Day 4) - (Subgroup C2, Day 9)

Figure 5.9: Repair step 2 for Thursday to Friday.

Second, similar to the second variant of the 'Monday–Wednesday' repair step, it is also possible to make a swap between Thursday or Friday and the range from Monday to Wednesday. Preserving the group structure requires a more complex modification that involves four swaps. The example in Figure 5.9 shows such a modification.

Repair step 'Saturday/Sunday'

This repair step is focused on the shift type *day shift weekend*. The operation plan allows this shift to be used on Sundays and with restrictions on Saturdays. The swap operation is defined rather straightforward. The shift is always applied in subgroups, so that no complex swap operation is required. Figure 5.10 shows an example of this type of repair step.

DAY	5	6	7	8	9	10	11	12	13	14
Subgroup A1				TD	TD	TD	TD		TDWE	
Subgroup A2		TDWE			TD	TD	TD			
Subgroup B1	TD			TD	TD		TD	TD	SWWE	
Subgroup B2	TD		TDWE	TD		TD	TD	TD	SWWE	
Subgroup C1	TD	SWWE		TD	TD	TD		TD		TDWE
Subgroup C2	TD	SWWE		TD	TD	TD		TD		

Violation at (Subgroup A2, Day 6)
Swap 1: (Subgroup A1, Day 6) - (Subgroup A1, Day 13)
Swap 2: (Subgroup A2, Day 6) - (Subgroup A2, Day 13)

Figure 5.10: Repair step for Saturday/Sunday *day shift weekend*.

Repair step 'Saturday'

Similar to the previous one, this repair step is also a straightforward implementation. It swaps shifts of the type *shift substitution weekend*. These shifts are always applied in groups. Therefore, a swap must always include groups thus preserving the requirements of the operation plan. Figure 5.11 illustrates this kind of repair step.

DAY	5	6	7	8	9	10	11	12	13	14
Subgroup A1				TD	TD	TD	TD		TDWE	
Subgroup A2		TDWE			TD	TD	TD			
Subgroup B1	TD			TD	TD		TD	TD	SWWE	
Subgroup B2	TD		TDWE	TD		TD	TD	TD	SWWE	
Subgroup C1	TD	SWWE		TD	TD	TD		TD		TDWE
Subgroup C2	TD	SWWE		TD	TD	TD		TD		

Violation at (Subgroup C1, Day 6)

Swap 1: (Subgroup B1, Day 6) - (Subgroup B1, Day 13)

Swap 2: (Subgroup B2, Day 6) - (Subgroup B2, Day 13)

Swap 3: (Subgroup C1, Day 6) - (Subgroup C1, Day 13)

Swap 4: (Subgroup C2, Day 6) - (Subgroup C2, Day 13)

Figure 5.11: Repair step for Saturday *shift substituion weekend*.

5.3.5 Optimization

Although, OptiFLIP++ (see Section 2.4) does already provide two algorithms for optimization, I decided to take a new approach for the application of shift scheduling. First, the interface of OptiFLIP++ has not been defined in a sufficient generic way to be appropriate for the purposes of the shift scheduling application. The two algorithms of OptiFLIP, a tabu search repair based hill climbing algorithm and a genetic algorithm, have been designed tightly coupled with the steel production application of ∗FLIP++. This produced quite satisfying results within this environment, but it did not prove to be an easy task to adapt the structure of OptiFLIP++, in particular the interface, to the shift scheduling application. For example, the steel production domain with its job scheduling has always schedules that can be represented with linear sequences, whereas the shift schedules have a the format of a matrix; which did not correspond to the OptiFLIP++ interface. Hence, a new approach has been taken by the implementation of another optimizing algorithm. The key objective was to come up with a simple and quick solution that does not sacrifice too much of the performance that has already been pushed to the limits by the use of DynaFLIP++ and its interpreter.

The new algorithm that has been implemented is based on *iterative improvement techniques*. Such an algorithm tries to improve an initial preliminary schedule iteratively. The modifications, or repair steps, are the operations to move from one schedule to another. The problem of getting trapped in a local optimum, can be overcome in several ways.

Our approach, which is called *iterative deepening*, works as follows: In a first try, a search to depth 1 is made. This means that starting from the initial schedule one modification is executed. The execution of one modification does imply that all possible modifications of that type are tried on the violated position. If no improvement is found, a depth-first backtracking search to depth 2 is performed in a second attempt. This means that the modification operator is applied twice which also allows to accept a worse intermediate schedule in terms of evaluation as long as the schedule at the end of the sequential application of modification operators is better. If still no improvement is found, the depth is increased iteratively until either a successful repair step is achieved or a limit is reached causing the algorithm to abort. Figure 5.12 shows how the nodes of the search tree are visited by the algorithm.

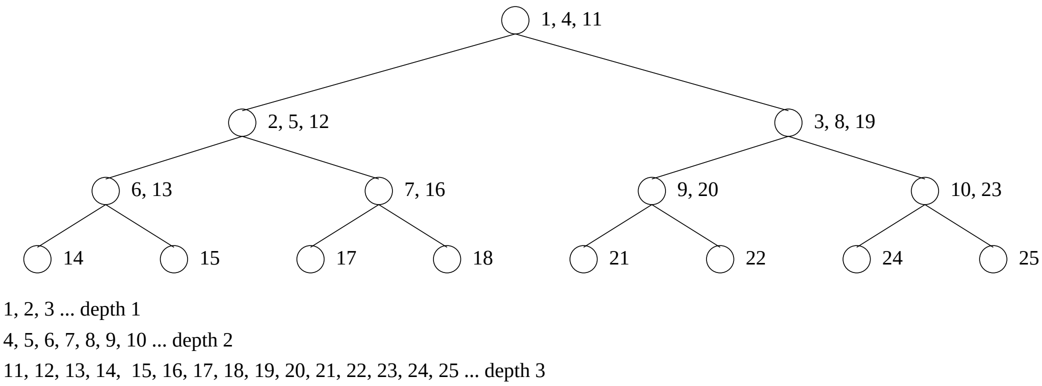

Figure 5.12: How the iterative deepening algorithm traverses the search tree.

A convenient feature of an iterative improvement technique is the anytime feature, i.e., the algorithm can be stopped at any time and always provides the best solution reached so far.

Of course, there is a great portion of leeway in how to define such an algorithm specifically. In this application, I added some additional features.

First, the algorithm starts with an initial schedule that has been evaluated with DynaFLIP++. The evaluation produces a list of evaluated constraints that is sorted by the severity of the violations. From each violated constraint we can derive possible candidates for modification, i.e., start positions, according to the type of the constraint. To increase the number of possible modifications tried in each depth level of the search, the algorithm tries to do all possible modifications by traversing the list of violated constraints and for each constraint applying modification operators on all start positions. For example, the algorithm starts with a depth level equal to 1, takes the most severely violated constraint, and starts with the first recommended possible start candidate. Then it applies the modification operator to this position until it finds an improvement or all possible operations with this specific start candidate have been tried. When the search for one start candidate is exhausted, the next one is tried, and so. If the list of start candidates has been traversed, the next violated constraint is selected. If all these tries fail, the depth level will be increased and the entire process starts all over again. The description makes clear that the possible steps tried by the algorithm are numerous. The success of the search is strongly coupled with some parameters like the number of violated constraints and possible modification start candidates that are taken into consideration. Figure 5.13 shows a sketch implementation of iterative deepening.

```
RepairDeep()
{  int Depth = 1, MaxDepth = 5;
   int Tries = 0;

   Evaluate();
   BestEvaluation = GetEvaluation();
   BestSchedule = GetSchedule();
   while ( Depth < MaxDepth )
   {  // RepairTry is a recursive functions that tries a
      // repair step by applying a modification operator
      // Depth times.
      bool Improved = RepairTry( Depth );
      if ( Improved )
      {  BestEvaluation = GetEvaluation();
         BestSchedule = GetSchedule();
         Depth = 1;
      }
      else
         Depth++;
      if ( BreakConditionFulfilled() ) break;
   }
};
```

Figure 5.13: A sketch of the iterative deepening optimizing algorithm.

5.4 Results

In the following I briefly summarize the results that have been attained with the problem so far. There is no doubt that there is still a lot of ground to cover as far as the fine-tuning of constraints is concerned. Nevertheless, the optimizations that can be done to this time are very promising and show the strength of the concept of the ∗FLIP++ system. Admittedly, the performance of the system has been pushed to its outmost limits by the implementation of the shift scheduling application and hence the focus of future developments should be on improvements of this crucial ingredient of the system.

The problems that have been encountered with the performance of the system during the optimization runs are mainly caused by the somewhat inefficient implementation of intensively used FLIP++ and DynaFLIP++ routines. Consequently, I took appropriate measures to simplify the problem. These measures predominantly concentrated on freeing the system from the burden of sophisticated and intensive computations. Randomizing various features of the optimizing algorithm in the way discussed in the previous section proved to be one of the more successful measures. These modifications resulted in an optimizing algorithm that behaves like a greedy hill-climbing optimization method, very similar to the well-known GSAT and similar iterative improvement optimization algorithms (turn to [16] for more details). Initial results are depicted in Figure 5.14. As one sees, results improve for approximately 1000 iterations and then stabilize around an overall satisfaction degree of 0.69. Several runs are depicted to show the variability of results obtained with the randomized search.

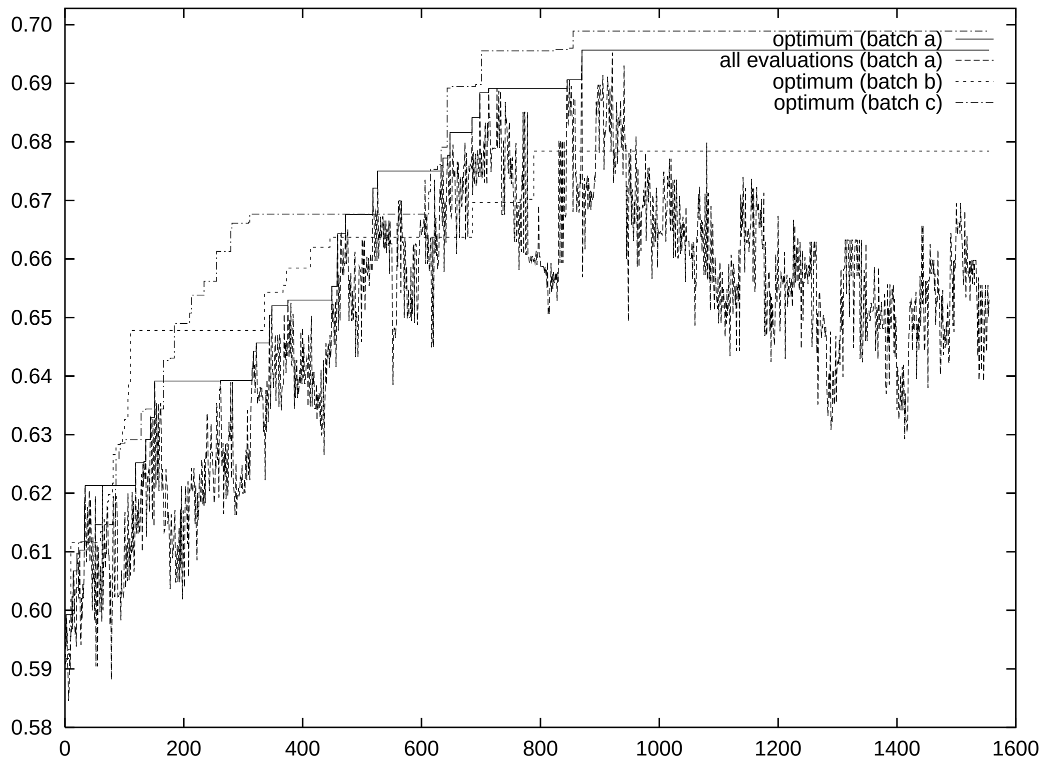

Figure 5.14: Optimization results.

Chapter 6

Conclusion

Don't worry about the world coming to an end today.
It's already tomorrow in Australia.

—Charles Schultz

The key area of this thesis involved the implementation of a domain knowledge representation module including a real world application within the ∗FLIP++ environment. The principal premise of the design and implementation has been an adaptation of the existing modules to ensure the provision of sound interfaces. This preparatory work has also resulted in intensive performance tuning and code revisions of ∗FLIP++. Subsequently, the knowledge representation module DomFLIP++ has been reimplemented. Simultaneously, we have considered the particular requirements of the shift scheduling application. Another key point has been the documentation of the transition of expert knowledge from the problem domain to the ∗FLIP++ system. As a result, both aims, providing a generic knowledge representation tool on the one side and a specialized application based on this tool on the other side, could be attained. However, the author is well aware of the plethora of variations of the application problem, not to mention the many other domains, that still remain and should be followed by further investigation to prove the capabilities of the system.

In the near future the entire ∗FLIP++ project will be ported and reimplemented in the *Java* programming language. This major transition is likely to offer various new possibilities and will make the library even more versatile. The *Java* language offers such convenient features as garbage collection and just-in-time compilation. Furthermore, the strong emphasis on portability, the already built-in networking capabilities, and a consistent object-oriented concept are features that make *Java* definitely superior in comparison to *C++*. Nevertheless, the current implementation that relies strongly on the Standard Template Library does underscore the strengths of *C++*.

Considerations regarding the reuse of current ∗FLIP++ implementations are closely associated with the upcoming transition. Basically, there exist three different aspects of reuse. Class hierarchy reuse is focused on the reuse of design; this requires reimplementation but with close observation of existing code. By contrast, direct component reuse is centered around wrapping existing code into an interface allowing to reuse it in a new program. Finally, conceptual reuse merely concentrates on the concepts and ideas and tries to embed them in the new environment mainly requiring both a redesign and reimplementation.

Bibliography

[1] Markus Bonner, Stefan Mayer, Andreas Raggl, and Wolfgang 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, August 1995. URL: ftp://ftp.dbai.-tuwien.ac.at/pub/papers/slany/ijcai95wsflai1.ps.gz.

[2] Markus Bonner. Learning Fuzzy Rules with Genetic Algorithm and Neural Nets. Diplomarbeit ($\simeq$ master's thesis), Technical University of Vienna, 1996.

[3] Per Cederqvist et al. Version Management with CVS. Signum Support AB, Linkoping, Sweden, October 1993.

[4] Jürgen Dorn and Reza Shams, An Expert System for Scheduling in a Steelmaking Plant. In *Proceedings of the 1st World Congress on Expert Systems*, pages 396–404, Pergamon Press, Orlando, USA, December 1991.

[5] Jürgen Dorn, Mario Girsch, Günther Skele, and Wolfgang 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.

[6] Margaret A. Ellis, Bjarne Stroustrup. The Annotated C++ Reference Manual. Addison-Wesley Publishing Company, 1990.

[7] Johannes Gärtner, Thomas Grechenig. Computergestützte, arbeitnehmerorientierte Arbeitszeitgestaltung — Möglichkeiten, Anforderun-

gen, Grenzen. In *Computergestützte Gruppenarbeit*, Proceedings, German Chapter of the ACM, Bremen, 1991.

[8] Johannes Gärtner, Sabine Wahl. Computer-aided Design of Shift Schedules — Running in a Foggy Swamp. Technical University of Vienna, 1995.

[9] Johannes Gärtner, Andrea Birbaumer. Enabling Group Work — Facilitation and CSCW tightly Knitted. Technical University of Vienna, 1995.

[10] Mario Girsch. Optimierung von Schedules mit Genetischen Algorithmen und Iterativer Vertiefung. Diplomarbeit ($\simeq$ master's thesis), Technical University of Vienna, 1994. In German.

[11] Neill Graham. Learning C++. McGraw-Hill, 1991.

[12] Witold Hendrysiak. Ein objektorientierter Entwurf von wiederverwendbarer Scheduling Software. Diplomarbeit ($\simeq$ master's thesis), Technical University of Vienna, 1995. In German.

[13] Witold Hendrysiak, Andreas Raggl. DomFLIP++. In *Proceedings of the FLINS conference*, Mol, Belgium, September 1996.

[14] Leslie Lamport. LaTeX: A Document Preparation System. User's Guide & Reference Manual. Addison-Wesley, 1986.

[15] Stanley B. Lippman. A C++ Primer. Addison-Wesley Publishing Company, 1989.

[16] Steven Minton, Mark Johnston, Andrew Philips, and Philip Laird. Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method. In *Proceedings of the 8^{th} National Conference on Artificial Intelligence*, pages 17–24, Menlo Park, USA, 1989.

[17] Michael Prah. CheckFLIP++. Diplomarbeit ($\simeq$ master's thesis), Technical University of Vienna, 1997.

[18] Ulrich Santa, Wolfgang Slany and Mazen Younes. $*$FLIP++: a partial constraint satisfaction system for steelmaking scheduling. In *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.

[19] Ulrich Santa. Dynamically Adapted Constraints for Fuzzy Multiple Criteria Optimization. Diplomarbeit ($\simeq$ master's thesis), Technical University of Vienna, 1996.

[20] Wolfgang Slany. Fuzzy scheduling. CD-Technical Report 94/66, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, 1994. URL: ftp://ftp.dbai.tuwien.ac.at/pub/papers/slany/cd-tr9466.ps.gz.

[21] Wolfgang Slany. Scheduling as a fuzzy multiple criteria optimization problem. *Fuzzy Sets and Systems*, 78:197–222, March 1996. Issue 2. Special Issue on Fuzzy Multiple Criteria Decision Making; URL: ftp://ftp.dbai.tuwien.ac.at/pub/papers/slany/fss96.ps.gz.

[22] Alexander Stepanov, Meng Lee. The Standard Template Library. Hewlett Packard Company, July 1995.

[23] Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2nd Edition, 1991.

[24] Standard Template Library Reference. Rensselaer Polytechnic Institute, 1994.

Appendix A

Concurrent Versions Control

The following part is a mix of experience the author gained with this piece of software throughout his work in the $*$FLIP++ project and the material he has gathered about `cvs`.

A.1 Introduction

`cvs` stands for concurrent versions control. It is used to record the history of source files.

It is an inherent characteristic of the process of software development that one makes mistakes. Sometimes it is quite annoying to remove a buggy code section after it has been tracked down. Even worse, not to seldom a programmer gets locked in a situation where returning to the last working piece of software is more than a routine task. The reasons why one can run into all these difficulties are obvious: Storing a separate set of source code files for every change is considered to be a major disturbance for the developer. Not to mention the fact that resources are almost always scarce, and, with a large project, storing many instances of the source code is a serious waste of disk space. This is a typical scenario where version control comes into play.

`cvs` does not only store several file versions in an efficient manner, i.e., in one single file using just the differences between the various versions. This tool also provides mechanisms that allow multiple users to work on the same source code sections. However, it is not a guaranteed safeguard against manipulations of the same code section by more than one user. Such conflicts always need some sort of communication between the developers on a meta-level (e.g., a review). Still, such conflicts are rare events that are most often avoided by both a clear partition of work and well-defined interfaces among the software developers. With these two strategies being a requirement for every major software project, the feature of parallel source code manipulation is well-justified.

A.2 History

The development of `cvs` was initiated by Dick Grune in 1986 [3]. At that time it was developed as a set of shell-scripts. Later on, a major redesign by Brian Berliner moved the tool away from the mere shell-script status. The system relies strongly on the unix tool `diff`, for tracking the file history, and `rcs`, for doing the version control. `cvs` is today available at various ftp sites, for instance `prep.ai.mit.edu` in '`pub/gnu`'. It is still public domain and is distributed under the *GNU Public License Agreement*[1], what makes `cvs` even more attractive for academic purposes. Nevertheless, `cvs` found its way into several professional software development institutions according to various reports on Internet newsgroups.

Like most public domain tools, `cvs` is still work in progress. Since its first major release in 1989, several refinements produced new versions. Both

[1]GNU is a software foundation with the objective to distribute software that is up to its standards for free.

users and developers think that there is still room for further improvement indicating that further releases are to be expected.

A.3 The limitations of CVS

To clarify the position of `cvs` as a software product, it is necessary to draw some borderlines. What `cvs` is really about is source control. It is by no means a full-scale configuration management tool and can therefore not be measured by such standards.

`cvs` is not a build system. This means that `cvs` does not store any information about the interrelationship of the source files. It is up to the user how to build the system. With ∗FLIP++, we relied on the popular unix tool `makefile` to accomplish that. Also, the organization of the source files is totally independent of the system's build structure. But, organizing your files in various modules that correspond to the structure of the system in hand is a convenient method. When we introduced `cvs` in the ∗FLIP++ project, the modules were organized exactly in the same as the entire project (for more information about the ∗FLIP++ structure turn to Chapter 2). However, the amount of work that gets tied up in the design of a proper build system is often underestimated and can be on the critical path of a project. ∗FLIP++ still has some ground to cover as far as the build system is considered.

Moreover, nothing could characterize the following argument better than the saying: *"A fool with a tool remains still a fool."* It points out that even if `cvs` is used as a project tool, it does not prevent its user from making serious mistakes. ∗FLIP++ has also encountered these problems and until recently some of them have not been addressed properly. Introducing a tool like `cvs` without instantaneously specifying rules and standards on how to use it, leaves both the project leader and the users in an unsatisfying position. Also, the experience with the tool proved that a step-by-step introduction seems to be more promising. Overwhelming the user with a every little hairy detail of the tool is not the way to introduce it. Furthermore, many features are not relevant at the time of inception or their implications cannot be estimated properly. Hence, the capabilities of `cvs` are still not applied in full in ∗FLIP++. The primary objective was to get a fast grip on source-code control, which has been achieved by the use of the very basic features of `cvs`.

`cvs` uses a quite powerful tool to resolve conflicts that arise when a file is manipulated simultaneously by more than one user. It is the `diff` command that is quite common among `unix` users. Still, the resolution of conflicts is limited to the textual level. If there are any conflicts on the logical level, it is again up to the developers to resolve that conflict via direct communication. This is of course not a weakness of `cvs`. No serious software developer would expect a tool to accomplish such a conflict resolution on the logical level. Besides, the importance of communication and clearly defined interfaces is once more emphasized. The ∗FLIP++ team tries to push that idea by regular meetings for all the participants. Apart from the need of intensive communication—a necessity for almost any software project—the ∗FLIP++ project has to cope with another problem: Since the project has been initiated at the university and the implementation is done almost entirely by students, the fluctuation of project members is very high. This provides a large field for coordination bearing in mind the possibility of reinventing the wheel again and again as new people join the project. Well-structured and up-to-date documentation is considered to be a cornerstone of coordination making it easier to grasp the concept of previously developed parts. A major advantage of the high fluctuation in the project is the fact that new, fresh ideas provide new input worth being further investigated.

A.4 Basic concepts

All the files that should be version controlled by `cvs` have to be checked in to a centralized repository. This is just a normal `unix` directory that contains a hierarchy of all history files that are necessary to track down the development path of each file. In order to handle the source files systematically `cvs` allows to group several files and directories into modules. This makes it very easy to organize logical units of a projects, e.g., more or less independent subprojects.

A new repository can be either an empty one or via the use of an import one that is already populated with a file structure. Of course, it is a wise step to rethink the structure of project before the version control is started so that the appropriate structure can be taken into consideration right from the beginning. Changes to the file structure are still possible after a repository has been generated, however, they can sometimes be a quite tedious task.

With easy-to-use command line options—as with most `unix` tools the whole functionality of `cvs` is wrapped into just one powerful command—the user can check in and out the files of the repository. Or, if new files have to be added or old files removed, one may change the repository. The files can be checked out to any directory. The retrieved files form a working set which is ready for modifications. This means that individual users are free to perform activities on the files as they have their own working set. Only when the working is checked in the changes will be made in the repository and are visible for other users. After the modification of a file and a check in a change log message is customary and the respective file receives a new revision number. The fact to comment on every change via the change log messages is very convenient. It gives everyone who participates in the repository the possibility to track down all the changes and to retrieve further information about them (like who, when, why, ...). It is also possible to modify the log mechanism in a manner that allows to notify other users via email if a change of a certain file has been made. This feature does not have any restrictions introduced by `cvs` and it is up to the creativity of the user and the capability of the `unix` operating system to find out new forms of applications. The ∗FLIP++ experience shows that the notification option is of real value when you want to monitor the development of source code that is manipulated by more than one user.

`cvs` also distinguishes between versions of a single file and versions of an entire piece of software (or a module in `cvs` terms). The former is called revisions and the latter named releases. While the revisions are tagged automatically by the system, the numbers and symbolic names for the releases are chosen by the user. A release should not be produced arbitrarily, but after reaching a strategic point in the development life-cycle of the project. What makes source control really a gift is the twofold advantage of moving along the revision/release tree. On the one hand, it is always possible to take a step in backward direction meaning to work on an older revision of a file or an older release of a complete module. On the other hand, the revision/release tree can be split up into various branches. This is an extremely convenient way to handle a situation where the development path of a software project is ambiguous and one decides to take more than one approach. Also, bug fixes for older releases can be introduced while the development of the current, often unstable, code is still under construction on a separate branch of the revision/release tree. If the modifications done on a side branch have proven to be stable, `cvs` allows to merge them with the main stream of development.

Another interesting feature of `cvs` is the possibility to maintain a remote repository. By using the `unix` networking capabilities the system is extended in a straightforward manner to behave as a client/server system enabling developers distributed all over the globe to work with the same repository. The drawback of this feature is the lack of security unless tools like `kerberos` are used.

As already mentioned in the introductory section, `cvs` handles conflicts solely on a textual level. However, the conflict management is well-structured and easy to understand. It is always possible to check to status of the files that are currently in the working set. The status reports whether a file is the same as its counterpart in the repository or whether it has been modified locally by another user or both. Unless a change is done at the same or nearly the same file position, it is always possible to merge simultaneous changes. When a working set is updated a detailed report is given on the detected changes and performed merges. Another related mechanism that can prevent unintended changes to files is the watch feature. It allows to make files only readable when they are checked out. A special command has to be issued in order to perform changes to a file which in turn releases a notification. This notification can be sent to any users who are specified on the watch list of the file.

Finally, a simple keyword substitution mechanism allows to automatically put information about file modifications directly into the file. This releases the programmer from the sometimes annoying task to keep comments (e.g., those in the file headers) up-to-date. The content of the information includes author, date, file name, log messages, revision number, etc. Taking for granted that the formulation of log messages is handled seriously, this allows all files to come with their complete history. Unnecessary to mention that the the keywords for the information can be placed at any position of the file so that it can even be used as the content of a program variable.

A.5 Concluding remarks

Having used `cvs` for over one year on a regular basis, it appears to be a practical and straightforward tool for version control. Certainly, the value one gets out of it is tightly coupled with the policy applied. Doing commits after every little fix may both overwhelm other users with log message and maybe leave a system that is almost never ready for compilation and release. On the other hand, making only rare updates may increase the risk of conflicts arising in the mean time. The best strategy to follow would be a test suite that has to be run before a new change is committed to the repository. `cvs` allows for such a strategy through its open design that is more or less only limited by the creativity of the user and the operating system used.

Appendix B

The Standard Template Library

Sometimes a programmer is called upon to be more a craftsman than an artist; and a craftsman's work is quite enjoyable when good tools and materials are present.

—Donald E. Knuth, *Literate Programming*

B.1 Introduction

The Standard Template Library (STL) is a C++ library that provides a set of well structured generic C++ components that work together in a seamless way. It is mainly focused on container classes and generic algorithms that are based on template functions and classes. STL has been mainly developed at the Hewlett Packard Laboratories by Alexander Stepanov, Meng Lee, and David Musser. Years of intense research activity in the area of generic programming and generic software libraries has been a major input for STL. Furthermore, many experts and experienced programmers participated in the revision of the proposal that has been made for STL to the ANSI/ISO C++ Standards Committee. In 1994, it was decided to adopt the Standard Template Library as part of the standard C++ library. Since then, more and more programs make use of the powerful functionality provided by STL. Nonetheless, it is still up to the compiler vendors to fully support the library according to its standard specifications. But this seems to be only a matter of time. During the work on this thesis, I used two different compilers that interpreted the STL more or less identically. The minor differences could be eliminated through the use of workarounds (e.g., compiler-dependent compilation via preprocessor switches).

STL is not exclusively based on object-oriented concepts, which will be further discussed in the section about algorithms (see Section B.4). But this is not a very uncommon feature in C++ that has been struggling with various relics of its ancestor C. However, leaving the grounds of object-oriented concepts is justified in the eyes of the designers of STL as long as performance is the paramount objective. Hence, one design guideline of STL was to achieve maximum performance. STL also enforces the concept of "generic programming", i.e., through the use of templates its algorithms and containers are independent of the underlying types used. Apart from algorithms and containers, there are several other components in STL. All together, they provide the basis for a highly generic concept with a high level of abstraction. The following gives a short overview of the STL components, afterwards the various components are discussed in more detail.

- Containers.

  Containers are provided to hold a set of objects of a specific type. There exists a variety of different container types; each having its advantages and disadvantages as far as time and space complexity are concerned. The basic container types are vectors, lists, deques, sets, and maps. Certainly, it is possible as with any C++ component to derive user-defined containers from these basic container classes.

- Iterators.

  Iterators enable the navigation on a set of objects, mainly referring to objects stored in a container. They can be compared with the conventional C/C++ pointers with the difference of being more generic in their application. The advantage of iterators is that they behave identically for each type of container. This is possible because each container provides the appropriate information about how an iterator is applied on it.

- Algorithms.

  They are used to make various manipulations on objects. They can be applied either on a single object or on an entire set of objects, i.e. on a container. Common algorithms are sort, find, delete, modify. Algorithms use iterators to work on containers. Subsequently, iterators represent the interfaces between algorithms and containers.

- Adaptors.

  Adaptors are used to modify STL according to individual preferences. They provide interfaces with other non-STL components, e.g., with streams.

B.2 Containers

From the very implementation view containers are responsible for managing a set of memory locations. In a more abstract way, containers hold a set of objects of a specific type. They are type-independent and hence require always the specification of a type.

Generally speaking, two types of containers can be distinguished:

B.2.1 Sequence containers

Sequence containers organize a set of objects, all of the same type, into a strictly linear arrangement. The order of the objects in the container is defined by the way they are inserted. STL provides three basic sequence containers:

- Vector.

A vector is a dynamic, self-growing array. It supports random access of elements. Delete and insert operations performed at the end of the vector are optimal and suboptimal (taking linear time) elsewhere.

- Deque.

  Deque is a short form for double ended queue. In contrast to a vector, a deque is a bidirectional dynamic array. This allows delete and insert operations at both the begin and the end with optimal performance. Similar to the vector, deletes and inserts in the middle of the container are suboptimal (taking linear time).

- List.

  List is a double linked list. It does not support random access. Being double linked means that a list enables iteration in both directions in constant time. Besides, delete and insert operations can be performed at any position with same time complexity.

Furthermore, by the use of adaptors it is possible to enhance basic sequence containers, e.g. to construct stacks or queues. Also, as with all STL components the user can derive and define its own sequence containers.

B.2.2 Associative containers

Associative containers are containers which hold pairs of data and keys. They are mainly used as structures for fast retrieval of data based on keys. The order of the objects in an associative container is defined by a sort criterion on the keys. Associative containers show better performance than sequence operators for sort and find algorithms. They are implemented as balanced binary trees. The basic types are sets, multisets, maps, and multimaps.

- Set/Multiset.

  A set sorts elements automatically according to their value. It supports only unique values. By contrast, the multiset allows the use of equal values. Both structures provide fast retrieval mechanisms.

- Map/Multimap.

  A map sorts elements automatically according to a key. It always stores pairs of key and value. A multimap supports equal key values. Both structures provide fast retrieval mechanisms for the values that are attached to the keys.

B.3 Iterators

Iterators can be compared to conventional C/C++ pointers iterating on an array. In STL they are generalized pointers that allow the programmer to use different structures, mainly containers, in a uniform manner. Each container type comes with its own iterator. The most important iterators used are the *begin* and *end* iterator. The *begin* iterator locates the position of the first element of a container. The *end* iterator marks the position at the end of the container, it is not referencing the last element of the container, i.e., the last element of the container is referenced by the last iterator before the *end* iterator.

Figure B.1 illustrates the use of the *begin* and *end* iterator in connection with a *vector* sequence container.

```
vector < int > v;
...
vector < int >::iterator i;

\\ show elements
for ( i = v.begin(); i != v.end(); i++ )
   cout << *i << ' ';
```

Figure B.1: Example of iterators.

At the beginning, a sequence operator is declared. It is a vector of integer objects. The specification of the object type is given, as with all template arguments, in angle brackets. Of course, any more complex type, e.g. a class, instead of integer could be used. Next, an iterator has to be declared that knows how to move on the vector. The iterator for a vector is contained in the vector class, which can be seen from the qualifier. This emphasizes once again that each container type has its own iterator, but all iterators behave in the same way. Finally, all elements of the container are shown on the standard output. The *begin* iterator refers to the start of the container and initializes the loop variable, whereas the *end* iterator is used for the exit condition of the loop. Since iterators are based on pointers, a object referenced by an iterator is accessed via the dereference operator. In summary, the loop shows three important operations: comparing iterators (i.e., the objects they are pointing at, `i != v.end()`), incrementing an iterator (`i++`), and dereferencing an iterator (`*i`).

The advantage of the generic concept of iterators is that a switch to an other container requires only the change of container and iterator declarations.

B.4 Algorithms

Algorithms are very useful since they facilitate various manipulations of containers. Among the STL operations available are find, swap, sort, copy, add, and modify functions. Algorithms are not member functions of container classes, but global functions. They use container classes via iterators. The advantage is that each algorithm has one implementation which works for all containers. Additionally, different container types can be mixed when using one algorithm. The drawback is that the object-oriented concept is violated because the link between data and operations is broken.

The specification of STL makes clear statements about the time and space complexity of all algorithms used. All algorithms work on a range of elements. The range is specified by iterators. It is up to the user to ensure that all ranges are valid (e.g., iterators are of the same container, end iterator does not come before the begin iterator). There are also algorithms that use more than one range. For example the copy algorithm copies the elements of a specified range to another range. The destination range is only specified by its start iterator since its length is derived from the source range. Still, one has to make sure that the destination range is prepared to take new elements.

Figure B.2 shows an example of how to use STL algorithms on a *vector* container.

```
vector < int > v;
...
vector < int >::iterator i;

\\ find minimum
i = min_element( v.begin(), v.end() );
cout << "min: " << *i << endl;

\\ find maximum
i = max_element( v.begin(), v.end() );
cout << "max: " << *i << endl;

\\ sort elements
sort( v.begin(), v.end() );

\\ show sorted elements
for ( i = v.begin(); i != v.end(); i++ )
   cout << *i << ' ';
```

Figure B.2: Example of algorithms.

After declaring both a *vector* container and an iterator i, three different algorithms are applied in the example: min_element, max_element, and sort. The first two algorithms find the element in the container that is the minimum or maximum respectively. The sort algorithms sorts the elements of the container. All three algorithms work on all elements of the container since the range is specified by the *begin* and *end* iterator. Finally, the sorted container is shown on the standard output.

Of course, these algorithms work also with containers holding more complex objects. One could think of a class where a comparison is defined through the use of a class attribute (or several class attributes if even more complex). Using the comparison operator $<$ alone is sufficient for all three algorithms.

B.5 Adaptors

Adaptors can be quite useful instruments to enhance the capabilities of algorithms. They can adapt various objects to iterators which are then used as arguments for algorithms. In this sense, they provide interface mappings for either containers or iterators.

There exist two types of adaptors, container adaptors and iterator adaptors:

A container adaptor is useful when it is necessary to provide a restricted interface for a container. STL has several predefined container iterators that can be used in connection with sequence containers. Among them are *stack* which can be used with *vector*, *list*, and *deque*; *queue* which supports *list* and *deque*; and finally, *priority queue* as interface for *vector* or *deque*. A *priority queue* is also better known as heap.

In the same manner as container adaptors provide a new interface for a container, an iterator adaptor provides a new interface for an iterator. Various such adaptors exist: For example, a reverse iterator allows to iterate through a data structure in opposite direction, this works of course only for bidirectional or random access iterators. Another important iterator adaptor is the insert iterator. It allows to use the dereferencing mechanism in a way that inserts an element into the underlying container instead of overwriting it. The following example in Figure B.3 illustrates the idea of such an iterator adaptor. It also shows another form of iterators that allows to use other non-STL structures likes streams as algorithm arguments.

As with the previous examples, a *vector* container is declared. Then the *copy* algorithm copies elements straight from standard input into the

```
vector < int > v;

\\ copy elements from cin to container
copy( istream_iterator< int >( cin ),
      istream_iterator< int >(),
      back_inserter( v ) );

\\ sort elements
sort( v.begin(), v.end() );

\\ copy elements from container to cout
copy( v.begin(), v.end(),
      ostream_iterator< int >( cout, "\n" ) );
```

Figure B.3: Example of adaptors.

container. This is possible due to the use of so-called stream iterator adaptors that specify the source range of the *copy* algorithm. Also, the destination uses an iterator adaptor. The *back_inserter* used as iterator adaptor for the *vector* container inserts the elements into the container. This is particularly convenient because the elements are now put into the container safely via its insert operation so that the programmer does not have to take any further measures to ensure the container is prepared to take the elements. Next, the *vector* container is sorted and finally the sorted container is written to standard output, again with the use of a stream iterator adaptor.

Taking a short look at the example, one can easily grasp the compactness of STL. Although it may take some time to adapt to the style of STL, it soon proves to be a powerful tool.

B.6 Error handling

So far, the standard STL does not include an error handling. Most functions just show an undefined behavior in case something occurs unexpectedly. Among the most common mistakes STL users make are the following:

- undefined or uninitialized iterators,
- ranges where end iterator is not accessible from begin iterator,
- and destination ranges that are not large enough to take elements.

The reason why the designers did not include error prevention techniques in STL are once again linked with the goal of attaining maximum performance. However, in the mean time several enhancements of STL have been contributed by various programmers. One of these contributions is the so-called *safeSTL* that combines both standard STL and an error handling concept based on the C++ exceptions mechanism. Still, the problem remains that *safeSTL* is non-standard and has been ported only to a limited number of platforms.

B.7 Concluding remarks

To summarize, taking a first glance at STL shows that all of its elements are written in as general a way as possible. It has been developed as a product of many years of research in the area of generic programming. Furthermore, the design of STL has been embedded very well into the C++ programming language. It fully supports all built-in C++ data structures as well as user defined components and extends basic C++ paradigms in a consistent way. This has been mainly achieved by a strong cooperation between the developers of STL and the developers of C++ (among them such renowned figures of the 'C++ world' as Bjarne Stroustrup and Andrew Koenig).

Through its component concept—mainly based on the four components container, iterator, algorithm, and adaptor—STL is able to achieve both a reduction of component space used and high efficiency without violating the principle of being as generic as possible. A strong argument in favor of STL is the following: The designers view software components mainly as three-dimensional arrays, where the first dimension represents data types (e.g., int), the second dimension represents containers (e.g., vector), and the third dimension represents algorithms (e.g., sort). If each dimension has the size of i, j, and k, respectively, then $i * j * k$ versions of code have to be designed to satisfy all possible combinations. If a parameterization is possible, like the template mechanism of C++, this can be reduced to $j * k$ versions. More significantly, with algorithms working on different containers there is only a need for $j + k$ versions of code. This is exactly what STL provides. Moreover, the easy way to specialize components of STL just by the usual object oriented mechanism of inheritance makes it even more attractive for a serious programmer.

Still, one must not forget that STL does not properly address the important and crucial issue of error-handling. Future redesigns and enhancements will show whether STL will overcome this obstacle.

Appendix C

Source code

C.1 DomFLIP++ header files

C.1.1 base.h

```
// What   : Domain Knowledge Representation with *FLIP++
// Project: DomFLIP++
// Author : Andreas Raggl
// Start  : 05.12.1996
// Last   : 31.01.1997
// File   : base.h


#if !defined( BASE_H )
#define BASE_H

#if defined ( __GNUC__ ) // GNU C++ 2.7.2
#include <iostream.h>
#endif
#if defined ( _MSC_VER ) // VISUAL C++ 4.2
#include <iostream>
#endif
#include <vector>
typedef char namestr[255];

class CBase
{
public:
CBase ( char* s = "CBase" , long l = 1 );
virtual ~CBase ();

virtual void Read ( istream& is );
virtual void Write ( ostream& os ) const;

friend ostream& operator<< ( ostream& os, CBase& c );
friend istream& operator>> ( istream& is, CBase& c );
protected:
```

```
namestr classname;
long classversion;
};

typedef vector <int
#if defined ( _MSC_VER )
, allocator<int>
#endif
> CIntVector;

typedef vector <CIntVector
#if defined ( _MSC_VER )
, allocator<CIntVector>
#endif
> CIntMatrix;

#endif // !defined( BASE_H )
```

C.1.2 domflip.h

```
// What   : Domain Knowledge Representation with *FLIP++
// Project: DomFLIP++
// Author : Andreas Raggl
// Start  : 11.12.1996
// Last   : 31.01.1997
// File   : domflip.h


#if !defined( DOMFLIP_H )
#define DOMFLIP_H

#if defined ( __GNUC__ )
#include <fstream.h>
#include <String.h>
#include <algo.h>
#endif
#if defined ( _MSC_VER )
#include <fstream>
#include <algorithm>
#endif

#include <time.h>
#include "base.h"
#include "inter.h"


#define MAX_FILE_LENGTH 65535

class CShift : public CBase
{
public:
CShift ( char *s = "NONAME", int i = 0, int j = 0 );
virtual ~CShift ();

void SetLength ( float f );
```

```
float GetLength () const;
void SetName ( char *s );
char *GetName () const;
void SetShiftsRequired ( int i );
int GetShiftsRequired () const;

void Read ( istream& is );
void Write ( ostream& os ) const;
friend ostream& operator<< ( ostream& os, CShift& c );
friend istream& operator>> ( istream& is, CShift& c );
private:
namestr name; // identifier
float length; // length in hours
int shiftsrequired; // number of shifts required
};


class CGroup : public CBase
{
public:
CGroup ( char *s = "NONAME", int i = 0 );
virtual ~CGroup ();

void SetMembers ( int i );
int GetMembers () const;
void SetName ( char *s );
char *GetName () const;

void Read ( istream& is );
void Write ( ostream& os ) const;
friend ostream& operator<< ( ostream& os, CGroup& c );
friend istream& operator>> ( istream& is, CGroup& c );
private:
namestr name; // identifier
int members; // number of group members
};
```

```
class COperation : public CBase
{
public:
COperation ();
virtual ~COperation();

void SetShift ( int i );
int GetShift () const;
void SetLevel ( int i );
int GetLevel () const;
void SetNumber ( int i);
int GetNumber () const;

void Read ( istream& is );
void Write ( ostream& os ) const;
friend ostream& operator<< ( ostream& os, COperation& c );
friend istream& operator>> ( istream& is, COperation& c );
private:
int shift;
int level;
int number;
};

typedef CIntMatrix CSchedule;

struct CShiftBlock
{
int start;
int size;
};

typedef vector <CShiftBlock
#if defined ( _MSC_VER )
, allocator<CShiftBlock>
#endif
> CShiftBlockVector;
```

```
typedef vector <CShiftBlockVector
#if defined ( _MSC_VER )
, allocator<CShiftBlockVector>
#endif
> CShiftBlockMatrix;

struct CPosition
{
int group;
int day;
};

typedef vector <CPosition
#if defined ( _MSC_VER )
, allocator<CPosition>
#endif
> CPositionVector;

typedef vector <CPositionVector
#if defined ( _MSC_VER )
, allocator<CPositionVector>
#endif
> CPositionMatrix;

typedef vector<float
#if defined ( _MSC_VER )
, allocator<float>
#endif
> CFloatVector;

typedef vector <CFloatVector
#if defined ( _MSC_VER )
, allocator<CFloatVector>
#endif
> CFloatMatrix;
```

```
typedef vector <COperation
#if defined ( _MSC_VER )
, allocator<COperation>
#endif
> COperationVector;

typedef vector <COperationVector
#if defined ( _MSC_VER )
, allocator<COperationVector>
#endif
> COperationMatrix;

typedef vector <CGroup
#if defined ( _MSC_VER )
, allocator<CGroup>
#endif
> CGroupVector;

typedef vector <CShift
#if defined ( _MSC_VER )
, allocator<CShift>
#endif
> CShiftVector;

class CDomainInstance : public CBase
{
public:
CDomainInstance();
virtual ~CDomainInstance ();

virtual float Evaluate ();
virtual bool Repair( int group, int day, int &start, int &start2 ) = 0;

bool Optimized ();

ConstraintEvaluations& GetEvalData();
float GetEvaluation ();
```

```
CPositionMatrix& GetRepairPositions();
void SetSchedule( CSchedule cs );
CSchedule& GetSchedule();
void SetStaticData ( char *c );
char* GetStaticData ();
void SetStaticDataFileName ( char *s );
char *GetStaticDataFileName ();
bool ReadStaticData ();

void Read ( istream& is );
void Write ( ostream& os ) const;
friend ostream& operator<< ( ostream& os, CDomainInstance& c );
friend istream& operator>> ( istream& is, CDomainInstance& c );

protected:
virtual void InitializeVariables( TVarlist *vars ) = 0;
virtual void ComputeRepairStarts() = 0;


CSchedule schedule;
CPositionMatrix repairstarts;
ConstraintEvaluations EvalData;
char *pStaticData;
namestr StaticDataFileName;
bool optimized;
float evaluation;
};

class CShiftSchedulingDomain : public CDomainInstance
{
public:
CShiftSchedulingDomain ();
virtual ~CShiftSchedulingDomain();

bool Repair( int group, int day, int &start, int &start2 );

void SetLengthOfCycle( int i );
```

```
int GetLengthOfCycle() const;

void WriteVariables ( char *filename );
void WriteVariables ( ostream &output );
void Write ( char *filename ) const;
void Read ( char *filename );
void Write ( ostream &output ) const;
void Read ( istream &input );
friend ostream& operator<< ( ostream& os, CShiftSchedulingDomain& c );
friend istream& operator>> ( istream& is, CShiftSchedulingDomain& c );

protected:
bool CheckPlan () const;
bool RepairStep1( int group, int day, int &start );
bool RepairStep2( int group, int day, int &start );
bool RepairStep3( int group, int day, int &start, int &start2 );
bool RepairStep4( int group, int day, int &start, int &start2 );
void Swap ( int group1, int day1, int group2, int day2);

void ComputeConstraintVars();
void ComputeConstraintVars_DIFFHOURS();
void ComputeConstraintVars_WORKINGHOURS();
void ComputeConstraintVars_SHIFTBLOCKS();
void ComputeConstraintVars_FREEWEEKENDS();
void ComputeConstraintVars_BADWEEKENDS();

void ComputeRepairStarts();
void ComputeRepairStart_DIFFHOURS( char *s );
void ComputeRepairStart_WORKINGHOURS( char *s );
void ComputeRepairStart_SHIFTBLOCKS( char *s );
void ComputeRepairStart_FREEWEEKENDS( char *s );
void ComputeRepairStart_BADWEEKENDS( char *s );

void InitializeVariables ( TVarlist *vars );

COperationMatrix *operation;
CShiftVector *shifts;
```

```
CGroupVector *groups;

int LengthOfCycle;
int LengthOfOperationCycle;
int HoursPerWeek;

CFloatMatrix c_diff;
CFloatMatrix c_work;
CShiftBlockMatrix c_shiftblocks;
CFloatVector c_freeweekends;
CIntMatrix c_badweekends;
};

#endif // !defined( DOMFLIP_H )
```

C.1.3 optimize.h

```
// What   : Domain Knowledge Representation with *FLIP++
// Project: DomFLIP++
// Author : Andreas Raggl
// Start  : 11.12.1996
// Last   : 31.01.1997
// File   : optimize.h


#if !defined( OPTIMIZE_H )
#define OPTIMIZE_H

#if defined ( __GNUC__ )
#include <fstream.h>
#endif
#if defined ( _MSC_VER )
#include <fstream>
#endif

//#include "base.h"
#include "domflip.h"

class COptimize : public CBase
{
public:
COptimize ( CShiftSchedulingDomain *c );
virtual ~COptimize ();

void RepairDeep ();
bool RepairTry( int depth );

void RepairRandom ();
void RepairTry2( int depth );

void Read ( istream& is );
void Write ( ostream& os ) const;
```

```
friend ostream& operator<< ( ostream& os, COptimize& c );
friend istream& operator>> ( istream& is, COptimize& c );
private:
CShiftSchedulingDomain *cDomain;
float fBestEvaluation;
CSchedule cBestSchedule;
};

#endif // !defined( OPTIMIZE_H )
```