DomFLIP++

Witold Hendrysiak, Andreas Raggl, and Wolfgang Slany
E184/2, TU Wien, A-1040 Vienna, Austria, Europe
E-Mail:{hendry|raggl|wsi}@dbai.tuwien.ac.at
http://www.dbai.tuwien.ac.at/staff/slany.html#sflip

DomFLIP++ is the knowledge engineering module of the *FLIP++ (pronounce: StarFlipPlusPlus) project. *FLIP++ is a tool for optimizing multiple criteria problems. It uses fuzzy constraints to model optimizing criteria and applies algorithms such as Tabu search or genetic algorithms to the problems. DomFLIP++ is a C++ library. It allows the definition of new optimization problems. It helps a domain engineer to design the structure of a new problem. However, there is a domain independent interface to other *FLIP++ modules such as OptiFLIP++, DynaFLIP++, and InterFLIP++. After each iteration in the optimization process, the considered instantiations of the problem are evaluated. Each evaluation produces a list of violated constraints. For each constraint, a list of repair steps is defined that can be used to increase the score of this constraint in further iterations of the optimization. A domain can be fine-tuned through modifications of constraints, through editing their repair lists, and through change in the optimizing parameters. A well-tuned domain can be successfully applied for optimization. Object-oriented design and implementation makes this module easy to modify and to reuse. Definition of new domains, system extensions with new optimizing algorithms, and definition of specific domain-dependent repair steps can be done efficiently. DomFLIP++ is tested on real-world example, namely scheduling the steel plant LD3 in Linz, Austria.

1 Introduction

DomFLIP++ is part of the *FLIP++ library[1,2] for real-world decision making. The *FLIP++ library allows optimizing under vague constraints of different importance using uncertain data, where compromises between antagonistic criteria can be modeled. Typical application areas include scheduling, design, configuration, planning, and classification.

1.1 FLIP++ as part of a scheduling project

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

- FLIP++: the basic fuzzy logic inference processor library.
- ConFLIP++: the static fuzzy constraint library.
- 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 to fine-tune 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.
- ReaFLIP++: the reactive optimizer as an extension for DomFLIP++.
- NeuroFLIP++: the learning library for FLIP++ using neural nets.
- TestFLIP++: the version control and test environment for the complete library set.
- SimFLIP++: the simulation toolkit library.

1.2 Organization of this paper

This paper will give a short overview of how to build an application with DomFLIP++. We go through the specific steps that are necessary to define a domain for an application. With the notion *domain* we refer to the definition of a specific scheduling problem. In a three-step description we show definitions of scheduling objects, aggregates, and domains including the evaluation/optimization process. Furthermore, we shortly touch upon the concepts of modeling constraints both statically and dynamically. A final conclusion provides an outlook on future work.

2 A DomFLIP++ application

To demonstrate the features of DomFLIP++ we give an explanation of a simple example which is focused on planning school timetables. Yoshikawa et al. [3] have recently shown a similar approach for solving such problems. Differently from our current system they put strong emphasis on finding sound initial assignments, arguing that these lead to better schedules. Furthermore, in their paper the authors use a minimum conflict hill climbing algorithm for the fine-tuning optimization of initial assignments.

The optimizations methods in our approach are also based on hill climbing algorithms, but we do no specifically focus on finding an appropriate initial assignment for the schedule since practical results as shown in Dorn et al. [4] have proven that even bad initial assignments lead to very good solutions.

2.1 Defining scheduling objects

With a scheduling problem at hand, there always exists a sequence of atomar objects that can be evaluated according to certain criteria. By means of scheduling we try to find an optimal sequence of these objects. Furthermore, it must be possible to change evaluation results by moving scheduling objects.

For our example we choose a *lesson* of the school timetable as the basic scheduling object.

2.2 Defining attributes of scheduling objects

Every scheduling object is required to have at least a unique key attribute. Furthermore, it can have other attributes with a static meaning, i.e., they will not be changed during scheduling.

The *lesson* object consists of the following attributes: *group*, which identifies the pupils in a class; *teacher*, who is the person holding a *lesson*; *subject*, which defines the topic of a *lesson*. The assignment of these attributes is not subject of the optimization. As we have no single key at hand, we define a key attribute *lesson_id* that uniquely identifies a triple of the above attributes. E.g., group *1A* is required to have two hours of *biology* a week, therefore, we choose *1A_biol_1* and *1A_biol_2* to identify the two *lesson* objects.

2.3 Defining constraints for scheduling objects

For all attributes needed for the evaluation, the definition of constraints is necessary. The ConFLIP++ library provides mechanisms for modeling constraints.

In ConFLIP++, the first step is creating linguistic variables with associated terms. For instance, we define a variable *number_of_lessons_per_day* with terms like *low*, *normal*, *high*, and *very high* and membership functions for each of the terms as shown in Figure 1. For instance, we can say that 0-3 lessons correspond to the term *low*, 4-5 lessons to the term *normal*, 6-7 lessons to the term *high*, and 8 lessons to the term *very high*.

Variables like the one above are then used to define simple constraints like

$$\mathit{number\_of\_lessons\_per\_day} \leq 7$$

and named, e.g., 'max_lessons_per_day-constraint', using the objects and methods defined in ConFLIP++. Certainly it is necessary to define constraints in the described way for every day of the week, whereby the optimal range varies from day to day, e.g. on Saturday afternoon lessons may not be preferable.

In the next step, several such constraints are logically combined, i.e., they are aggregated by one of the aggregation operators, such as AND and OR,

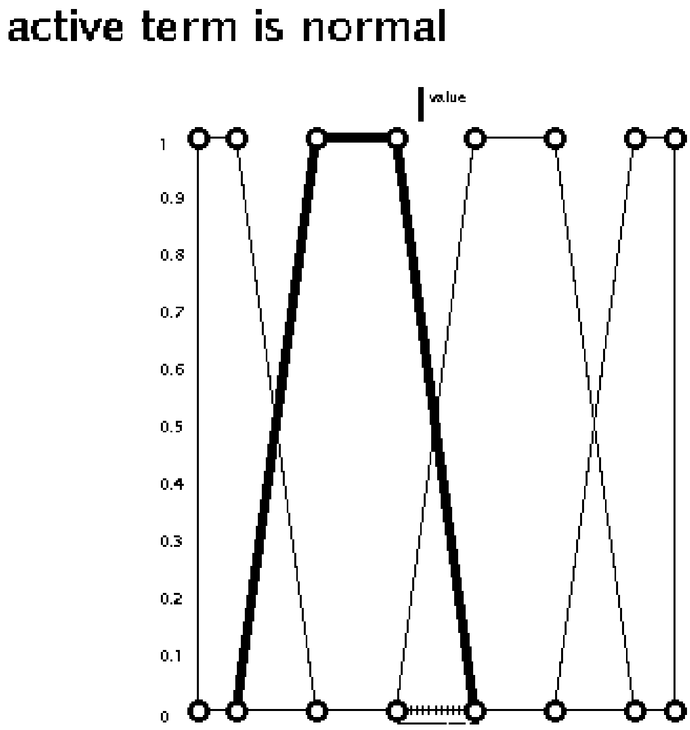

Figure 1: Membership functions for *number_of_lessons_per_day*.

which for example could be implemented as *minimum* and *maximum* operators, to build more and more complex constraints within a hierarchical structure.

In our example, this would mean that we also model a constraint 'min_lessons_per_day-constraint' for the lower boundary of *number_of_lessons_per_day*. And then we aggregate these two constraints to an overall constraint. One can think of this as building a hierarchy of constraints which eventually delivers an overall evaluation value at the top end of the hierarchy.

Furthermore, the constraints introduced are of dynamic nature, meaning that the variable *number_of_lessons_per_day* has to be initialized for each timetable schedule to be evaluated by some kind of function specified in DomFLIP++. Therefore, ConFLIP++ is enhanced with the DynaFLIP++ library which transforms dynamic constraints using such functions into static con-

Table 1: A rule specifying when and how a template constraint is used.

CONDITION:
$\text{lesson}_i.\text{day} \neq \text{lesson}_{i-1}.\text{day} \vee \text{i}=0 \wedge$
$\text{lesson}_{j+1}.\text{day} = \text{lesson}_i.\text{day}+1 \wedge$
$\text{lesson}_j.\text{day} = \text{lesson}_i.\text{day}$
CONSTRAINT:
min_lessons_per_day-constraint$\langle$count_lessons(lesson_i,lesson_{i+1})$\rangle$
max_lessons_per_day-constraint$\langle$count_lessons(lesson_i,lesson_{i+1})$\rangle$

straints that can be handled by ConFLIP++. Actually, DomFLIP++ is only interfacing with DynaFLIP++. Table 1 shows such a template constraint definition with DynaFLIP++ for our example. The conditions simply determine the boundaries for the *lessons* of a day. The static constraints for ConFLIP++ are then generated by the use of the *count_lessons* functions that computes the number of *lessons* for a day.

2.4 Defining aggregates

An aggregate comprises a list of scheduling objects. Generally, we distinguish between two different types of aggregates: the *pool* aggregate contains an unordered and not evaluated list of scheduling objects; the *plan* aggregate in turn defines an ordered list of scheduling objects on which the evaluation takes place.

In terms of our example for each *group* we define a *pool* aggregate that contains all the *lessons* that have to be taught to the *group* per week. E.g., the aggregate *pool_1A* contains at least the scheduling objects *1A_biol_1* and *1A_biol_2* as can be seen in Figure 2.

A *plan* aggregate represents timetables that are available for *lessons*. If we assume *lessons* to be held 6 days a week and 10 hours a day, we have a set of 60 possible slots for *lesson* objects. By default, the *plan* aggregate consists of 60 elements which are fill-ins for the scheduling objects of the *pool* aggregate. A fill-in in the *plan* aggregate means that the *group* is available for the respective time slot. If a fill-in is replaced by a scheduling object of the *pool* aggregate this means that the *group* has a *lesson* at the respective time slot. The tables shown in Figure 2 are internally defined as a linear lists of scheduling objects. Additionally, a *plan* aggregate can also be defined for a room, so that we receive an optimal timetable for the room by optimizing this aggregate. In our example we create aggregates for each group, room, and teacher. Of course optimization takes place only once, the different aggregates

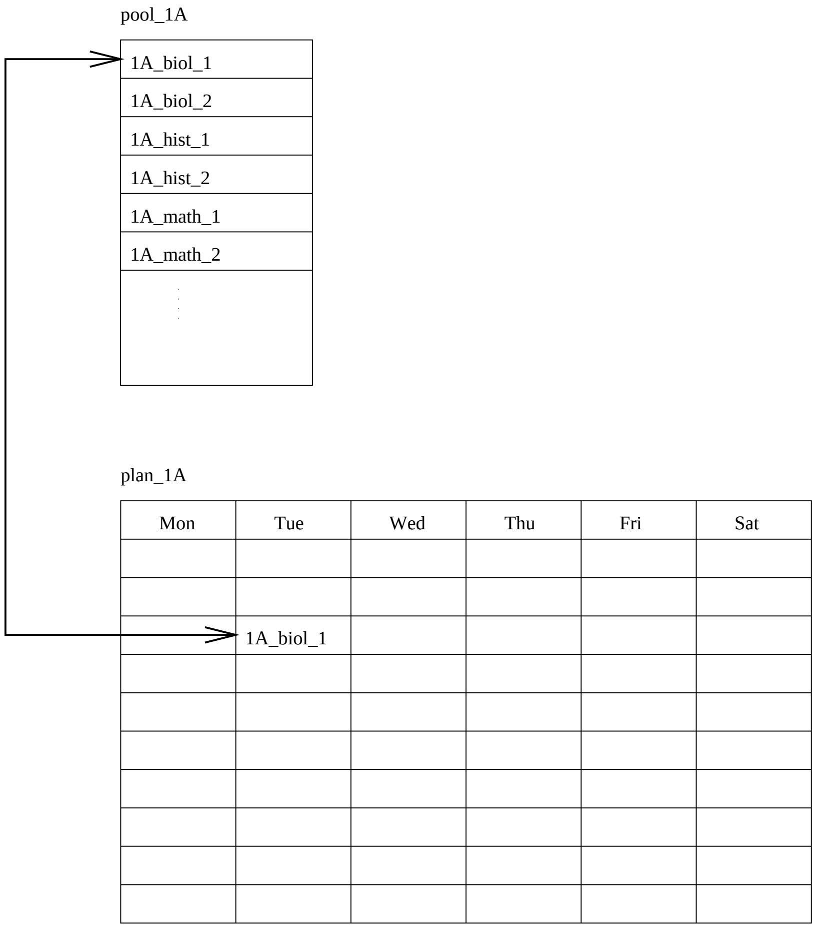

Figure 2: Aggregates

are just different views of the same scheduling problem.

2.5 The domain

A domain consists of a list of aggregates, a list of constraints for the domain, a list of repair steps, and constraint violations.

The aggregates in our example are the *plan* aggregates for rooms and groups and the *pool* aggregates for the groups.

The list of constraints for the domain defines restrictions between aggregates. For instance, we want to avoid that the same teacher holds lessons in two different groups at the same time.

After an evaluation we get a result indicating how optimal a solution is. Furthermore, we receive a list of the worst evaluated constraints. Repair steps are used by the optimizing algorithms as instructions how to make modifications to find better solutions. For each constraint a list of corresponding repair steps has to be specified, so that badly evaluated constraints can be improved.

3 Conclusion

Solving school timetable scheduling problems is only one field of application, we are currently also working on a shiftplanning application. Moreover, another important issue is the design and implementation of a graphical user interface for DomFLIP++ which also enables the non-expert to easily handle a scheduling problem interactively. Especially the optimization process still depends a lot on the fine-tuning by a human expert.

1. 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.
2. 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.
3. Masazumi Yoshikawa, Kazuya Kaneko, Toru Yamanouchi, and Masanobu Watanabe. A constraint-based high school scheduling system. *IEEE Expert*, 11(1):63–72, February 1996.
4. 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. A revised version has been accepted for publication in the European Journal of Operational Research. URL: ftp://ftp.dbai.tuwien.ac.at/pub/papers/slany/cd-tr9461.ps.gz.