A Standard Framework for Timetabling Problems

Matthias Gröbner[1], Peter Wilke[2], and Stefan Büttcher[1]

[1] Lehrstuhl für Informatik II,
Universität Erlangen-Nürnberg,
Martensstrasse 3, 91058 Erlangen, Germany
Groebner@informatik.uni-erlangen.de
[2] Centre for Intelligent Information Processing Systems (CIIPS),
Department of Electrical & Electronic Engineering,
The University of Western Australia,
35 Stirling Highway, Crawley WA 6009, Australia
wilke@ee.uwa.edu.au

Abstract. When timetabling experts are faced with a new timetabling problem, they usually develop a very specialised and optimised solution for this new underlying problem.
One disadvantage of this strategy is that even slight changes of the problem description often cause a complete redesign of data structures and algorithms. Furthermore, other timetabling problems cannot be fit to the data structures provided.
To avoid this, we have developed a standardised framework which can describe arbitrary timetabling problems such as university timetabling, examination timetabling, school timetabling, sports timetabling or employee timetabling. Thus, a general timetabling language has been developed which enables the definition of resources, events and constraints.
Furthermore, we provide a way to apply standard problem solving methods such as branch-and-bound or genetic algorithms to timetabling problems defined by means of the general timetabling language. These algorithms can be improved by problem-specific user-defined hybrid operators.
In this paper we present a generalised view on timetabling problems from which we derive our timetabling framework. The framework implementation and its application possibilities are shown with some concrete examples. The paper concludes with some preliminary results and an outlook.

1 Introduction

There exist many different timetabling problems such as university or examination timetabling, school timetabling, sports timetabling or employee timetabling. Furthermore, there exist many problem solving methods, which usually use the concepts of standard optimisation algorithms such as Backtracking [14] Evolutionary Algorithms [1,4,6,8] or Constraint Logic Programming [10,13].

E. Burke and P. De Causmaecker (Eds.): PATAT 2002, LNCS 2740, pp. 24–38, 2003.

Unfortunately, these standard algorithms often do not yield acceptable timetables or cannot compute solutions within a reasonable amount of time. So the standard algorithms have to be adapted to be able to handle the special concrete timetabling problem. Therefore, the search strategy is changed and special operators and problem-specific data structures are developed.

The disadvantage of these optimised implementations is that slight changes of the problem description often cause radical changes of the data structures and algorithms have to be redesigned to get acceptable solutions again. In addition, these data structures cannot be used to describe other new timetabling problems.

A further problem of missing standard timetabling descriptions is that new proposed optimisation algorithms cannot be reliably compared to existing ones with respect to performance and solution quality.

Moreover, from a theoretical point of view it would be interesting to compare the structure of different timetabling problems to each other or different problem solving strategies. Open questions such as the phase transition niche [16] could be analysed. This could be a step towards a better understanding of the timetabling research field.

In recent years this problem has been tackled, and first attempts have been made to standardise the description of timetabling problems [3,5,11,12]. In this paper we present a new way to generalise the timetabling problem that adopts some ideas from these known approaches and introduces new points of view. From this general view we derive an object-oriented standard timetabling framework which is able to describe arbitrary timetabling problems and can apply standard optimisation algorithms to the problem.

2 The General View

2.1 A Generic Timetable Scheme

As mentioned before, many types of timetabling problems exist. But all these problems have several properties in common.

One of these similarities is that certain entities have to be scheduled. For example, the German high school timetabling problem [1,15] has several entities such as classes or single students, teachers, subjects, lessons and rooms. All these entities have properties. For example classes are linked to the subject the students of this class are taught.

Usually, these entities are differentiated into *resources* and *events* (or sometimes called *meetings*). In addition, *constraints* have to be considered.

In the employee timetabling case, for instance, we find those entities, too. There are employees with different qualifications and monthly target hours or there are shifts the employees have to be assigned to.

As already mentioned, some of these entities are linked with others. There exist links from the shifts to the employees assigned to these shifts or from the students to their teachers. Some of these links are fixed, such as the links from the shifts to the employees with the qualifications required to work on these

Table 1. Different categories of resources in timetabling (TT) problems

High school TT	Employee TT	Arbitrary course TT
Classes/Students	Employees	Participants
Teachers	Shifts	Courses
Rooms	Qualifications	Organiser
Subjects	Employer(s)	(Rooms)
Lessons		

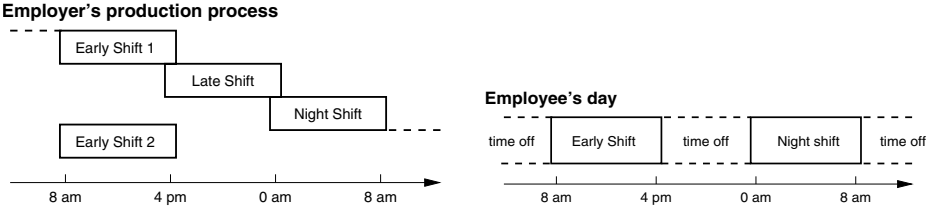

Fig. 1. The same time stream from the employer's point of view and from an employee's point of view, respectively

shifts, and cannot be changed. Others have to be assigned during a planning process, e.g. linking a lesson to a suitable room.

A planning algorithm has to construct a timetable, so we have to define what a timetable consists of. A timetable can be interpreted as an arbitrary sequence of *events*. To every event a certain number of time intervals is assigned, each having a starting and an ending point.

Each timetable can be seen from different points of view: for example, an employer has a different view compared to the view of his employees, as shown in Figure 1.

2.2 The Object-Oriented View

The considerations of Section 2.1 can be used to describe timetabling problems in an object-oriented manner:

There are different *resources* whose instances have references to each other, e.g. an instance of the *subject* class refers to instances of the *teacher* class who are able to teach that subject.

Moreover, there are entities with a certain property, called *events*. This property is a certain time interval (or several time intervals) that is assigned to these events, as shown in Figure 2.

2.3 Planning

An algorithm for constructing a timetable has to assign instances of the different resource classes to the event class instances. Some of these assignments are

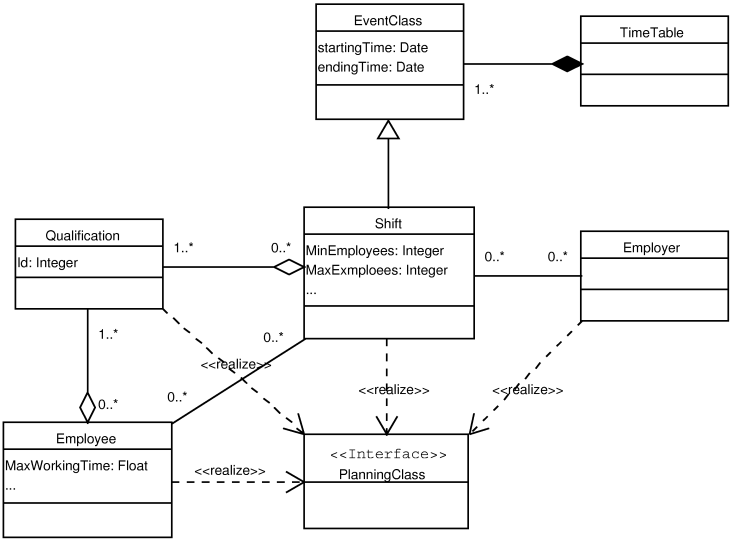

Fig. 2. Object-oriented view of the employee timetabling problem in Unified Modelling Language (UML)

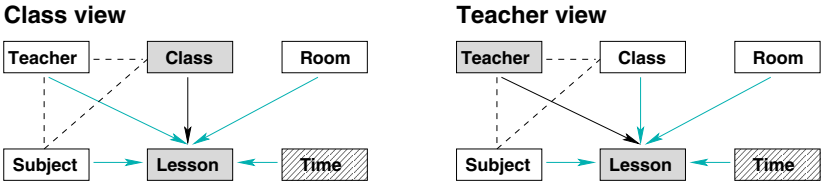

Fig. 3. Class view and teacher view. The viewing instance `class` or `teacher` is fixed (black arrows), whereas instances of the other classes to be planned have to be assigned (light arrows) to the event class `lesson`. The assignment of a time interval to each event is mandatory for all timetabling problems

predetermined and cannot be changed, and some have to be done during the planning phase.

To construct a timetable, one of the *views* mentioned in Section 2.1 is used. In the school timetabling case our algorithm might use the `class` view to assign a subject, teacher and room to a `lesson`. In this case the class is fixed and the other instances have to be assigned to (see Figure 3). Additionally, a time interval has to be assigned to each event class instance.

For each viewing perspective there are as many timetables as instances of this class to be planned exist. If we have t teachers at a high school, for example, t different teacher timetables belong to them. That is, if there exist l lessons in a high school timetabling problem, furthermore t teachers, r rooms and c classes, the number of instances of event classes including all views will be $(t+r+c) \times l$.

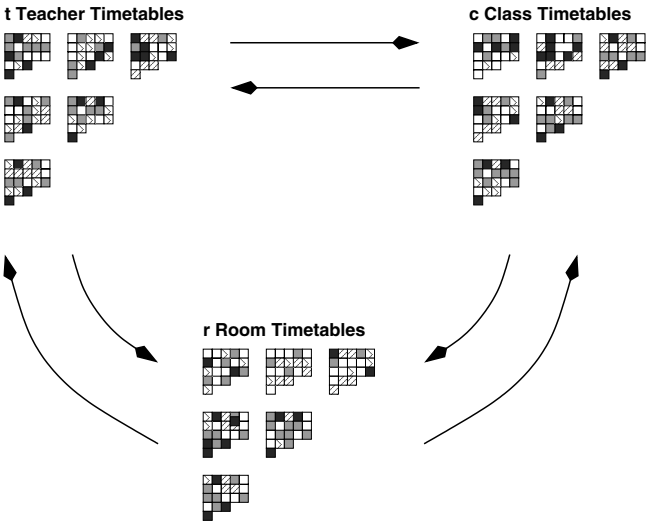

Fig. 4. The different timetables can be mapped to each other

The timetables of the instances of one planning class contain all information necessary to construct the timetables for the instances of the other planning classes: i.e. the timetables of the different views can be mapped to timetables of other views. From the employees' timetables the employer's timetable can be constructed or in the school timetabling case the t teachers' timetables can be mapped to the r rooms' timetables (see Figure 4).

That is why it is usually sufficient for a timetabling program to save the timetables of one resource type only. This avoids data redundancy caused by storing the same event information in different places, i.e. from different views (Figure 5). Nevertheless, to be able to check constraint violations (see the next section), translations to other views have to be done, for example to compute the number of assigned lessons of a teacher when working with the class *view*. Otherwise expensive computing time has to be accepted in order to compute the necessary information.

2.4 Constraints

Assignments usually cannot be done arbitrarily, but many constraints have to be considered. We distinguish two different types, namely hard and soft constraints. A solution is feasible if no hard constraints are violated. A feasible solution is better than another if fewer soft constraints are violated.

A timetabling algorithm can use different strategies to get a solution without violations of hard constraints. Violations can either be avoided from the outset [13,14] or penalised to lead the algorithm towards better solutions and introduce repair mechanisms [7].

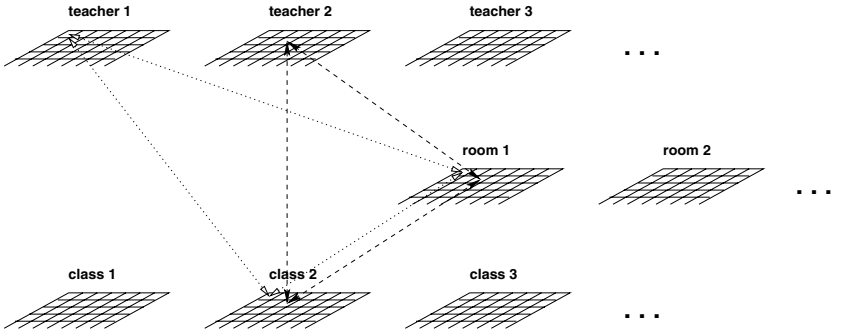

Fig. 5. Information redundancy is necessary to compute constraint violations of all resources

3 A General Timetabling Language

To describe the concepts introduced in Section 2, we implemented a general timetabling language (GTL). A GTL file starts with the declaration of the name of the timetabling problem, e.g. **TimetablingProblem** `School { ... }`.

3.1 Resources and Events

The language describes resources as instances of the `Resource` class and events as instances of the `Event` class. The super-class of both `Resource` and `Event` is the so-called `PlanningClass` class.

To declare members of these derived classes, the standard data types `boolean`, `float`, `double`, `int`, `short`, `long` and `string` are provided. The `reference` keyword declares a reference to objects of a certain type, where multi-dimensional arrays are indicated by the dimension count included in brackets.

To give an impression, the declaration of an *employee* resource in an employee timetabling example is shown:

```
class Staff extends Resource {
    string firstName;
    double workLoad;  // target working hours

    // list of shifts the employee cannot be assigned to
    reference(Shift)[1] absent;
} // end of class Staff
```

Additional standard member variables, such as an array containing references to the assigned events, are automatically provided by the timetabling framework as described in Section 4.

Events are declared in the same way. The concept of links from the events to the resources is covered by the timetabling framework, thus arrays containing references to the assigned resources need not be declared, as shown in the shift example:

```
class Shift extends Event {
    int shiftType;
} // end of class Shift
```

3.2 Constraints

In GTL constraints are introduced by a `Constraint` class. Instances of this class are assigned at a later stage in the timetabling framework to each planning class. A constraint can be derived either from the `HardConstraint` or from the `SoftConstraint` class. The declaration of a constraint class itself does not imply any direct consequences yet, but the different handling of the `Constraint` instances as hard or weak has to be managed by the algorithms applied. The `Constraint` class has a `compute` method to compute the constraint violations. This method must be implemented in the GTL file by the user. It returns a `ConstraintViolation` object which contains information about the penalty points and information for a `repair` method of the corresponding constraint class that can be implemented, too. The syntax of GTL is Java-like.

The `repair` method can be called by timetabling algorithms and uses problem-specific knowledge, which is usually necessary to get feasible solutions with Genetic Algorithms [2,8,15,17]. The example below shows the GTL definition of `ClashConstraint` and its `compute` method. Methods like `getAssignments` and `clashes` are provided by the standard library of the framework (Section 4):

```
class ClashConstraint extends HardConstraint {
   public ConstraintViolation compute(PlanningClass owner) {
      double penalty = 0.0;
      List assignments = ((Resource) o).getAssignments();
      List clashList = new ArrayList();
      for (int j = 0; j < assignments.size() - 1; j++) {
          Assignment ass1 = (Assignment)assignments.get(j);
             for (int k = j + 1; k < assignments.size(); k++) {
                Assignment ass2 = (Assignment)assignments.get(k);
                if (ass1.event.clashes(ass2.event)) {
                   penalty += 10.0; clashList.add(ass2.resList);
      }   }  }
      if (penalty > 0) {
          return new ConstraintViolation(this, owner, "Clash at "
                 + owner + ": " + clashList, penalty, clashList);
      } else return null;
   } // end of compute(PlanningClass)
   public boolean repair(ConstraintViolation violation) {
   ...
   } // end of repair(ConstraintViolation)
} // end of class ClashConstraint
```

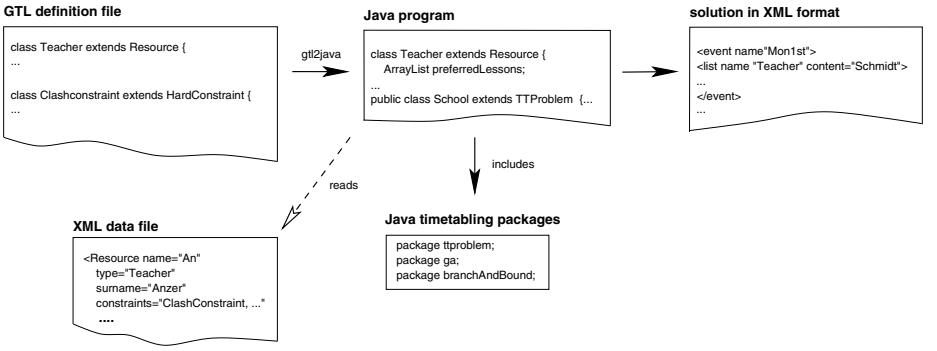

Fig. 6. The road-map to get a solution for a specific timetabling problem from a GTL description and an XML data file

4 The Timetabling Framework

All necessary classes of a timetabling problem can be declared with the timetabling language GTL. But to be able to store all necessary information about a specific instance of a timetabling problem, a framework is needed that instantiates the actual resources and events, applies algorithms to get solutions and for that purpose computes the constraint violations.

In order to ensure this, we developed a framework [9] that is able to read information about the actual instances of the corresponding timetabling problem using the GTL class declarations of the underlying GTL file. This information has to be defined in an XML data file. Thus, in contrast to STTL [12] the information about the specific instances of the timetabling problem is separated from problem definition itself.

The framework is then able to apply standardised algorithms to compute a solution of the timetabling problem, which can be exported to XML, HTML or text format. The solution contains information about the timetables of the different views and all remaining constraint violations.

Currently the framework is available in a beta version providing the complete functionality described in this section. The GTL description file (declaration of resources and the definition of the constraints) and the XML data file has to be created by the user. A friendly user interface does not exist at the current stage of development. Especially for the design of the `compute` method of the constraints some understanding of the data structures is required.

4.1 XML Data File

The syntax of the XML input data file is defined as follows. The file starts with the definition of the timetabling problem, where `layout` describes the structure of the timetable. That information is only used for graphical output, e.g. inside an HTML file. The meaning of the three parameters is *Days per Period, Time Intervals per Day* and *Periods*.

```
 <TTProblem layout="5,11,1" description="Name of the
school">
```

Definition of the resources can be done by a `Resource` tag which contains arbitrary elements and their values according to the member variables defined in the GTL file. `type` describes the special type a resource is of and `name` is a kind of ID. The `constraints` list contains the constraints that have to be considered for this resource instance. The example shows the definition of a specific employee of a rostering problem in a hospital:

```
 <Resource type="Staff" name="1126"
description="Schlueter, Anna"
    workLoad="15.0" features="Nurse,Chief"
    absent="" constraints="ClashConstraint,DiffHourMonthConstraint,
              OnlyOneDayFreeConstraint,OnlyOneWorkingDayConstraint,..."/>
```

There exists a pre-defined resource type `TimeSlot` with the variables *name, from, until* and *layout* which are defined in the same way. *layout* defines a position in the timetable grid as explained above.

Events also have a name, type (i.e. the event class) and constraints. In addition, events have a number of resource lists which contain all possible assignable resources, indicated by the `takefrom` keyword. In the following example the assigned time slot is fixed and cannot be changed, so the variable `fixed` is set to true. During the planning process, to all resource lists which are not fixed arbitrary resources are assigned taking members from the `takefrom` list. The minimum and maximum number of resources to be selected for assignment from the resource list by the timetabling algorithm are described by the `min` and `max` keywords:

```
 <Event type="Shift" name="February 26th (early
shift)"
       constraints="OutOfMinMaxConstraint" shiftType="0">
  <ResourceList name="TimeSlot" min="1" max="1"
                takeFrom="February 26th 06:30" fixed="true"/>
  <ResourceList name="Nurse" min="2" max="4" target="3"
                takeFrom="2176,2277,2282,2570,2770,2793,2807,2829,..."/>
  <ResourceList name="Student" min="0" max="2" target="1"
                takeFrom="3065,3069,3173,3178,3295,3296,3297"/>
  <ResourceList name="Chief" min="0" max="1" target="1" takeFrom="2277"/>
</Event>
```

4.2 Timetabling Packages

In order to generate an executable program from the GTL description as indicated in Figure 6 we need some standard timetabling packages which provide the basic timetabling class structures. These packages are included into the main program.

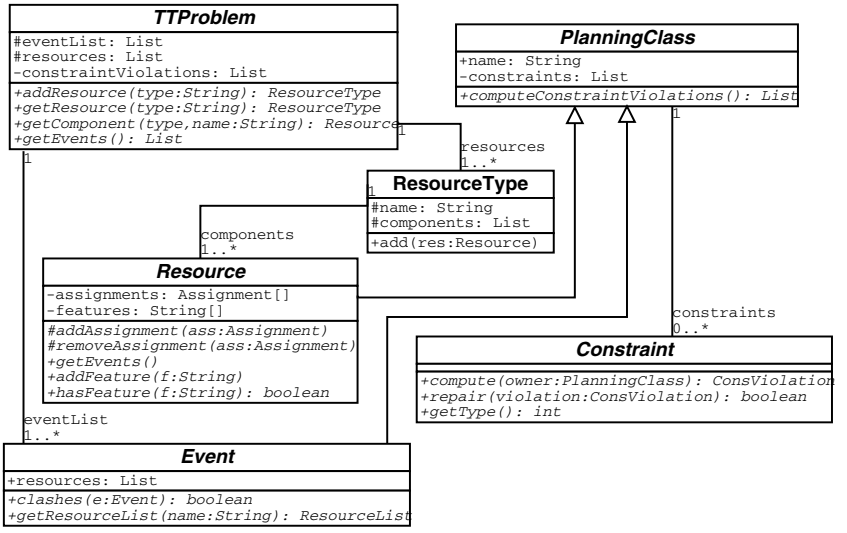

Fig. 7. Data structure of timetabling problem as provided by the `ttclasses` package

The structure of the `ttclasses` package is shown in Figure 7. A timetabling problem `TTProblem` holds an event list with all events and lists with resources of different types. Both `Resource` and `Event` are derived from the basic class `PlanningClass`. Each instance of `PlanningClass` refers to a number of constraints which should not be violated for this instance. The actual resource and event instances are read from the XML data file, are instantiated and then added to the corresponding `eventList` or `resourceList`, respectively.

Our implementation has been done in Java. One disadvantage of Java is its low run time performance. Nevertheless, Java allows straight object-oriented design of the data structures and provides a lot of useful standard packages such as lists and containers which facilitate handling complex data as found in timetabling problems. Furthermore, Java byte code can be executed on any operating system which allows an easy application of the framework on any computer. A comparison of the run time performance of the framework and an optimised implementation for a single problem is made in the next section.

To get an executable Java class file, we implemented a GTL-to-Java compiler which translates the GTL description into Java source code and adds some additional code such as import statements, constructors of the classes and a main method for starting the program.

The framework currently provides the possibility to apply a standard hybrid Genetic Algorithm and a simple branch-and-bound algorithm to the timetabling problem. The framework utility packages are included in the main Java file. Both algorithms use problem-specific knowledge defined by means of the repair operators of the constraint violations of the corresponding GTL files. Implementation details of the algorithms are omitted here for lack of space.

5 Example

5.1 GTL

To demonstrate the ability of our framework to solve real-world problems we will have a closer look at the time scheduling for a special course week at our university. The intention of the course is to attract more students to technical-oriented studies. So a whole week packed with lab classes and social function is organised. Each participant selects four courses with first priority and another four with second priority. The participant can also name a friend he would like to join him in the lab classes. As lab space is limited most courses are run more than once to give as many participants as possible a chance to attend.

The constraints when setting up a timetable are:

- participants can be assigned only once at the same time (clash constraint),
- participants may join a course of a certain type only once,
- participants should be assigned to their preferred courses,
- participants should be assigned to exactly four courses,
- some participants have to be assigned to courses taking place at the same time due to car pools,
- the number of participants assigned to the courses must be at least *min* and at most *max*.

In 2001, 251 participants were registered and 187 courses of 37 different types were offered. Courses take place either in the morning or in the afternoon, but there exist two-part courses which comprise two sessions and thus can take place in the morning and afternoon or even on two different days.

The GTL description of the resource `Participant` and the event `Course` is given as follows:

```
class Participant extends Resource {
    string firstName;

    reference(Project)[1] firstChoice;
    reference(Project)[1] secondChoice;
    reference(Participant)[1] friends;
}

class Project extends Event {
    string signature;
}
```

`firstChoice` and `secondChoice` are containers holding the preferred and alternative projects of the participant. `friends` are the friends that should be assigned to courses taking place at the same time as the courses the participant has been assigned to. Each event has a signature which defines the course type of the project.

In addition, the constraints presented above have been defined with corresponding `compute` and `repair` methods.

5.2 XML Data File

To get an XML input database of the practical week instances, we wrote a converter program that translates the available ASCII data into the required XML input format. This XML file describes a concrete participant my means of a `Resource` tag of the type `Participant` and elements defining the values of the firstname, name, the selected projects, the friends and the constraints to be considered.

For each day timeslots are defined for morning and afternoon courses, so 10 timeslots have been defined.

Each `Project` event has the variables *signature*, which is used to identify the type a certain project is of, and *name*, which specifies the course ID. An event owns two resource lists. One resource list holds the corresponding time slot and is fixed, because each course is assigned to one of two (two-part courses) fixed time slots. The other resource list contains all participants that can be assigned to that event because the course type is on their selection list.

5.3 Results

We applied the branch-and-bound algorithm as well as the standard hybrid Genetic Algorithm to the problem. Figure 8 shows several runs of the Genetic Algorithm with different parameter values. Computing time was 75 min on a 1.2 GHz computer. As expected, the steady-state Genetic Algorithm without usage of repair operators converges much more slowly and does not reach such good results as the hybrid runs.

All results computed by the hybrid Genetic Algorithm yielded feasible timetables with less than 10 courses having few more than *max* participants. Due to high request for some courses, on average about 30 participants were assigned to only three instead of four courses. Finally, there remained about 200 participants who did not get their first-choice courses but had to join at least one of the alternative courses selected. These constraint violations explain the remaining penalty points in Figure 8.

In addition, we applied our simple branch and bound algorithm to the problem. The final results were only marginally different with respect to penalty points and number of constraint violations. But the branch and bound algorithm outperformed the Genetic Algorithm with respect to time and computed the results in only about 25 s.

To get further information about the appropriateness of the Genetic Algorithm we introduced room resources with allocation clashes to be avoided. In this case the Genetic Algorithm yielded feasible solutions where the branch and bound algorithm failed to find solutions of the same quality in reasonable time.

Furthermore, we compared the timetables computed by our framework to those created by a special Genetic Algorithm-based application. The algorithm has been developed for this specific course timetabling problem in C++ programming language. The special application stopped after 10 min computing time and on average created timetables with about 60 participants assigned to

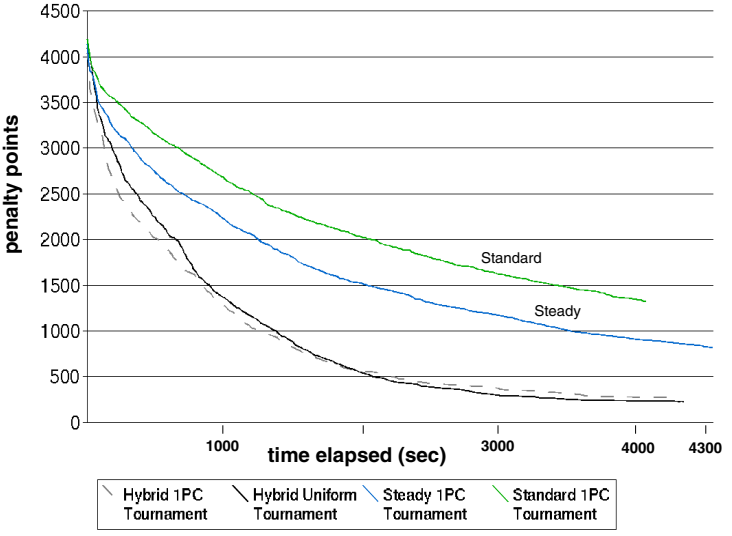

Fig. 8. Application of the Genetic Algorithm to the practical week timetabling problem. Hybrid: hybrid Genetic Algorithm with application of the repair operators; Steady: steady-state Genetic Algorithm; Standard: without any elitism or hybrid components. Crossover method is either one-point-crossover (1PC) or uniform. Selection method is always tournament selection

too few courses and more than 200 participants assigned to some alternative courses. Thus computing timetables using our framework did not result in a loss of quality, but computing time clearly increased compared to specially designed programs.

6 Conclusion and Outlook

In this paper we have presented a general object-oriented view on timetabling problems and how this description has been realized by means of a timetabling framework. The framework is able to describe arbitrary timetabling problems using the general timetabling language GTL. Actual instances are defined by an XML input file. Furthermore, standardised timetabling algorithms are provided by the timetabling framework.

One advantage of the framework is that it ensures that the application of timetabling algorithms to new timetabling problems neither enforces the redesign of the data structure nor the redesign of the algorithms. Furthermore, the widespread use of the framework could ensure a better comparability of new timetabling algorithms because it introduces standardised input and output formats.

However, one disadvantage of the presented timetabling framework is that the existing timetabling data has to be transformed into the defined XML data

format which requires some amount of preliminary work. But it is usually much less costly to convert data to a standard format than developing special data structures in a programming language.

Our next steps will be the implementation of further algorithms such as Simulated Annealing or Tabu Search so as to compare the applicability of these algorithms to different problems. To date, we have defined a school timetabling problem, a practical week problem and some employee timetabling problems using GTL and XML input format and successfully applied the Genetic Algorithm.

First results show that it is possible to compare the solubility of timetabling problems with respect to different timetabling algorithms. As a next project we want to analyse the structure of arbitrary timetabling problems with the help of the timetabling framework to get a better understanding of which algorithms should be preferably used to solve which type of problems.

Acknowledgement. This work has been partly supported by a grant from the Lehrstuhl für Programmiersprachen-und-methodik, Institut für Informatik, Friedrich-Alexander-Universitaet Erlangen-Nuernberg, Martensstrasse 3, 91058 Erlangen, Germany. We would like to thank them for their support.

References

1. Bufe, M., Fischer, T., Gubbels, H., Häcker, C., Haspirch, O., Scheibel, C.,Weicker, K., Wenig, M.: Automated Solution of a Highly Constrained School Timetabling Problem – Preliminary Results. In: Boers, E.J.W. et al. (Eds.): Proc. EvoWorkshops 2001. Springer-Verlag, Berlin Heidelberg New York (2001) 431–440
2. Burke E., Elliman, D., Weare, R.: Specialised Recombinative Operators for Timetabling Problems. In: Proc. AISB (AI and Simulated Behaviour) Workshop Evolut. Comput. Springer-Verlag, Berlin Heidelberg New York (1995) 75–85
3. Burke E.K., Kingston J.H.: A Standard Data Format for Timetabling instances. In: Burke, E, Carter, M. (Eds.): Practice and Theory of Automated Timetabling II (PATAT 1997, Toronto, Canada, August, selected papers). Lecture Notes in Computer Science, Vol. 1408. Springer-Verlag, Berlin Heidelberg New York (1998) 213–222
4. Caldeira, J.P., Rosa, A.C.: School Timetabling Using Genetic Search. In: Burke, E, Carter, M. (Eds.): Practice and Theory of Automated Timetabling II (PATAT 1997, Toronto, Canada, August, selected papers). Lecture Notes in Computer Science, Vol. 1408. Springer-Verlag, Berlin Heidelberg New York (1998) 115–122
5. Colorni, A., Dorigo, M., Maniezzo, V.: Genetic Algorithms and Highly Constrained Problems: The Time-Table Case. In: Proc. 1st Int. Workshop on Parallel Problem Solving from Nature. Springer-Verlag, Berlin Heidelberg New York (1990) 55–59
6. Corne, D., Ross, P., Fang, H.-L.: Evolutionary Timetabling: Practice, Prospects and Work in Progress. In: Prosser, P. (Ed.): Proc. UK Planning and Scheduling SIG Workshop. University of Strathclyde (1994)
7. Fernandes, C., Caldeira, J.P., Melicio, F., Rosa, A.: High School Weekly Timetabling by Evolutionary Algorithms. In: Proc. 14th Annual ACM Symp. on Applied Computing (San Antonio, TX, 1999)

8. Gröbner, M., Wilke, P.: Optimizing Employee Schedules by a Hybrid Genetic Algorithm. In: Boers, E.J.W. et al. (Eds.): Proc. EvoWorkshops 2001. Springer-Verlag, Berlin Heidelberg New York (2001) 463–472
9. Gröbner, M.: GTL, A General Timetabling Language, Beta Version, and a Bibliography of Timetabling Publications. Available at http://www2.cs.fau.de/Research/Activities/Soft-Computing/Timetabling (2002)
10. Gueret, C., Jussien, N., Boizumault, P., Prins, C.: Building University Timetables Using Constraint Logic Programming. In: Burke, E, Ross, P. (Eds.): Practice and Theory of Automated Timetabling I (PATAT 1995, Edinburgh, Aug/Sept, selected papers). Lecture Notes in Computer Science, Vol. 1153. Springer-Verlag, Berlin Heidelberg New York (1996) 393–408
11. Kingston, J.H.: A User's Guide to the STTL Timetabling Language, Version 1.0. Basser Department of Computer Science, The University of Sydney (1999)
12. Kingston, J.H.: Modelling Timetabling Problems with STTL. In: Burke, E, Erben W. (Eds.): Practice and Theory of Automated Timetabling III (PATAT 2000, Konstanz, Germany, August, selected papers). Lecture Notes in Computer Science, Vol. 2079. Springer-Verlag, Berlin Heidelberg New York (2001) 433–445
13. Lever, J., Wallace, M., Richards, B.: Constraint Logic Programming for Scheduling and Planning. British Telecom Technol. J. **13** (1995) No 1, January
14. Meisels, A., Lusternik, N.: Experiments on Networks of Employee Timetabling Problems. In: Burke, E, Carter, M. (Eds.): Practice and Theory of Automated Timetabling II (PATAT 1997, Toronto, Canada, August, selected papers). Lecture Notes in Computer Science, Vol. 1408. Springer-Verlag, Berlin Heidelberg New York (1998) 215–228
15. Oster, N.: Stundenplanerstellung für Schulen mit Evolutionären Verfahren. Thesis, Universität Erlangen-Nürnberg (2001)
16. Ross, P., Corne, D., Terashima, H.: The Phase Transition Niche for Evolutionary Algorithms in Timetabling. In: Burke, E, Ross, P. (Eds.): Practice and Theory of Automated Timetabling I (PATAT 1995, Edinburgh, Aug/Sept, selected papers). Lecture Notes in Computer Science, Vol. 1153. Springer-Verlag, Berlin Heidelberg New York (1996) 269–282
17. Weare, R., Burke, E., Elliman, D.: A Hybrid Genetic Algorithm for Highly Constrained Timetabling Problems. In: Eshelman, L.J. (Ed.): Proc. 6th Int. Conf. Genetic Algorithms (Pittsburg, PA). Morgan Kaufmann, San Mateo, CA (1995) 605–610