Automatic Construction of Drama School Timetables Based on a Generic Evolutionary Framework for Allocation and Scheduling Problems

EVOLUTIONARY COMPUTATION AND OPTIMIZATION TRACK
at the 19th ACM Symposium on Applied Computing - SAC 2004
March 14–17, 2004, Nicosia, Cyprus

Oskar Preinfalk

Department of Computer Science
University of Salzburg, Austria
ossi@cosy.sbg.ac.at

Helmut A. Mayer

Department of Computer Science
University of Salzburg, Austria
helmut@cosy.sbg.ac.at

Correspondence to:

Helmut Mayer
Universität Salzburg
Institut für Computerwissenschaften
Jakob–Haringer–Straße 2
A–5020 Salzburg
AUSTRIA

Telephone: +43-662-8044-6315
FAX: +43-662-8044-611

Automatic Construction of Drama School Timetables Based on a Generic Evolutionary Framework for Allocation and Scheduling Problems

Abstract

We present the application of the generic framework evAlloc for the solution of allocation and scheduling problems (ASPs) to a real–world problem. The solution engine integrated in the framework is based on an evolutionary algorithm (EA). The general design of the Java framework allows for application to all ASPs, whose problem data description can be fit into the generic data representation of evAlloc. The framework can be transformed into different applications by loading single XML (extended markup language) problem definition files. Experimental results for the real–world application, timetabling of the complete teaching activities at the Institute for Drama at the Mozarteum University of Music and Dramatic Arts in Salzburg, Austria, are presented.

1 Introduction

For the majority of specific instances of ASPs it is still very difficult to find "good" solutions generated by computational intelligence systems. The main reason for this situation is that above problems are NP–complete (or NP–hard) as documented in [1] giving an extensive survey of NP–hard problems in the allocation and scheduling domain [1]. Mostly, problem solvers adapted to a specific ASP are employed to tackle these problems. E.g., Schaerf and Meisels [2] use local search

techniques for timetabling problems; Drexl [3] proposes a simulated annealing approach for a knapsack problem; Hart and Ross [4] solve job shop problems with a hybrid EA (heuristic combination method).

Realizing that the essence of any ASP is the distribution of resources to consumers, e.g., containers on a ship, jobs to machines, or hours in a timetable, we designed a generic framework, which can be used to solve any ASP, whose description can be modelled according to the very general problem data structure defined in the framework. The solution engine currently implemented in the framework is an EA [1], however, due to the object–oriented design, any other optimization method could be easily added to the system. Evidently, such a generic system might not produce solutions comparable to specific problem solvers, but in many real–world applications the “only” requirement is to improve existing solutions. It may be also sufficient to generate solutions similar to those provided by human experts spending hours, days, or even weeks to construct these solutions.

A similar framework called *Vishnu* has been presented in [5, 6]. It uses a fixed–size set of formulas for the problem representation and the scheduling semantics. A web–based architecture enables multiple users to graphically create and edit problem definitions, initiate and cancel scheduler runs and view the solutions. Vishnu's problem solver is grounded on an order based genetic algorithm combined with a greedy schedule builder. A main difference between Vishnu and evAlloc is the description of the specific problem data. While Vishnu defines a specific syntax for the problem formulation, evAlloc essentially employs the Java language to capture the problem details, as all problem–specific objects are defined (and loaded at runtime) in an XML problem file. Although, this introduces the necessity to program a few additional classes for an application (by

[1] This explains the *ev* in *evAlloc*.

an application engineer), the user then, can manipulate the data in an intuitive way without any knowledge of a formal language. Moreover, with an increasing number of applications a growing number of classes might be reusable for a new application.

Section 2 presents main concepts of the evAlloc framework. A real world application of a timetable problem is presented in section 3. Results can be found in section 4.

2 The evAlloc framework

The main components of the framework are the problem representation, the solution engine (an EA), and the graphical user interface (GUI) for manipulation of problem–specific data and visualization of solutions. In order to employ evAlloc for a specific application, an application engineer has to model the problem according to evAlloc's problem representation, add (or reuse) a few software components, and save the full application to the XML problem definition file. The user of the system does not have to bother with any technical details concerning the framework, but simply loads the XML problem file and manipulates problem–specific data via the GUI.

2.1 The generic problem representation

The problem (the data describing it) is hierarchically organized in a tree data structure. Each node of the tree can have an arbitrary number of children. The root of the tree is the *AllocationProblem*, which is decomposed into *AllocationGroups*. The allocation groups resemble the sets of resources and consumers of the ASP. Each group contains a number of *AllocationItems* representing the set

of items to be allocated/scheduled. In the next tree level the *States* are accomodated describing the state of each allocation item. In software terms the state is a *Dynamic Variable*, which may be a simple parameter changing its value, but also a much more complex structure such as a timetable.

Each State may have associated *Constraints* (children in the tree structure) controlling the States' values. If a Constraint defined to be *hard* is violated during the construction of a solution, it immediately stops the construction so as to save computation time (2.2). States and Constraints resemble the problem–specific code an application engineer has to provide in order to transform the evAlloc framework into a full application.

Figure 1 shows screen shots of a job shop and a knapsack problem modelled according to evAlloc's problem representation.

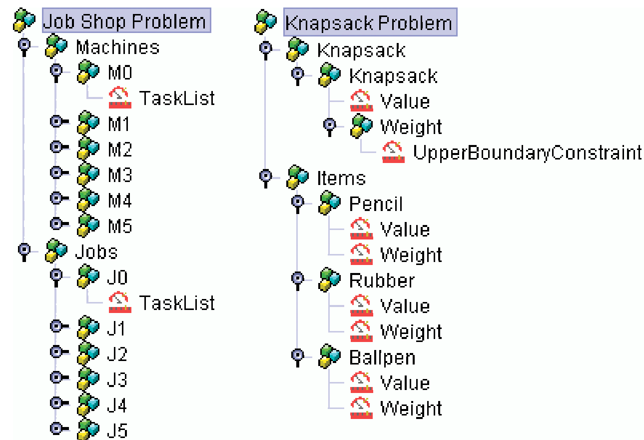

Figure 1: Problem representation of a Job Shop (left) and a Knapsack problem (right).

E.g., in the job shop problem representation *Machines* and *Jobs* are allocation groups. *Value* and *Weight* are States of a knapsack, whose weight is constrained by an upper boundary (*UpperBoundaryConstraint*). All the parameters of the problem representation can be graphically edited by the user (including the addi-

tion/removal of allocation items, e.g., add machine *M6* to the job shop problem). The complete problem can be saved in an XML problem definition file, whose structure reflects the problem tree.

2.2 The construction of solutions

The basic software technique to generate a solution is based on cloning the problem tree. This copy becomes the solution tree, where the problem solver manipulates the data to construct a solution. The solution engine (here an EA) provides a sequence of resource items, which are allocated to consumers by a *Negotiator*. The negotiator is another key feature of evAlloc's design, as it allows any degree of incorporation of problem–specific heuristics in a single well–defined class. E.g., the negotiator can operate without any specific problem knowledge by mechanically allocating the resource items to the consumers, or it can implement any number of problem–specific heuristics before passing an item to a consumer.

The fully constructed solution can then be evaluated by objective (fitness) functions, which can be defined without writing a line of code. Any node in the problem tree may be used as a term in the fitness function, which can be graphically constructed in a special editor. As often the user cannot know all the details necessary to construct a proper fitness function, pre–defined functions can be provided in the XML problem file (the user can select among them by name).

It should be emphasized that the construction of a solution is an iterative process closely modelling the real–world allocation process. E.g., in case of the knapsack problem (Figure 1) the resource items suggested by the solution engine are put into the knapsack. If the *Pencil* is moved to the knapsack, the corresponding States change its values, hence, the knapsack's State always reflects the current *Value* and *Weight*.

The dynamics of the construction of multiple solutions driven by an evolutionary solution engine (Section 2.2.1) are visualized in Figure 2.

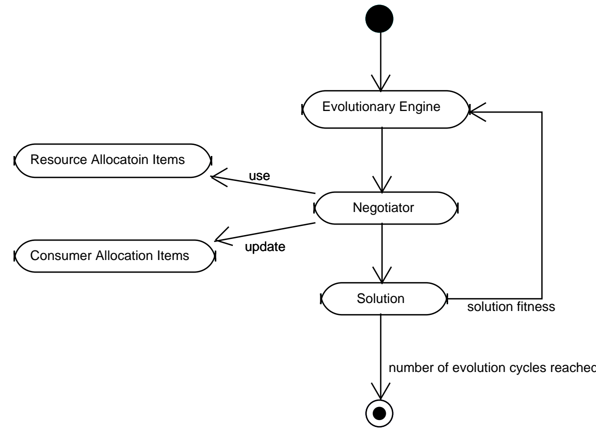

Figure 2: Dynamics of the construction of solutions.

2.2.1 Evolutionary Engine

The solution engine in evAlloc is intended to generate the processing order of allocation items. This "raw solution" can either be used directly for allocation of items in the given order (direct representation), or may be additionally processed by heuristics in the negotiator (implicit representation). E.g., if we want to schedule a job (an item in the sequence) on a machine (a consumer), the negotiator could simply try all machines (consumer items) until it finds one accepting the job (negotiator needs no problem knowledge), or it could try to find the machine, where the job can be executed optimally (needs problem knowledge).

The specific solution engine implemented with evAlloc is based on an EA. For most problems a single chromosome with a simple permutation encoding indexing the resource items would be sufficient. As genotype encoding is known to greatly influence the quality of EA solutions, we also suppport a variety of encoding options facilitated by the Java package *JEvolution* written by one of

the authors. Currently, JEvolution provides four different chromosome types: A permutation chromosome, a bitstring chromosome, an integer, and a real number chromosome.

A very interesting feature (which we are currently also investigating with evolution of fuzzy controllers and artificial neural networks) is *Multi–Chromosomal* encoding, i.e., the genotype consists of a pre–defined number of chromosomes. With this technique one chromosome could encode the order of resource items and another chromosome the associated consumer items. Also, different chromosomes could have different encodings.

According to evAlloc's strict object–oriented design the negotiator only may receive item sequences (which makes solution engines exchangeable), thus, chromosomes must be transformed to item sequences. In the case of a permutation chromosome this is a simple exercise, but for multi–chromosomal encoding we have recently started to add *Mergers* performing the transformation, e.g., a simple merging operation would be a concatenation of items the single chromosomes are encoding.

Another option to improve (or speed up) the evolutionary process is repair (reordering) of the genotype [7], which is essentially a form of *Lamarckian* evolution. Hence, we have also introduced a repair method in the negotiator. Obviously, the repair operation has to have exact problem knowledge to perform a meaningful reordering of the genotype. E.g., we could order the encoded allocation items according to the number of constraints so as to allocate the most constrained items before others.

When a solution has been constructed its fitness is evaluated and passed to the evolution engine, where the genetic operators are applied at the transition from one generation to the next. The solution(s) with the best fitness are stored

and can be viewed even while evAlloc is still searching for better solutions. All the specific evolution and encoding parameters can be set by the application engineer (in a specific GUI mode supporting these actions), and are also stored in the XML problem file. Hence, the user does not have to bother with settings of specific parameters like mutation rate or population size. He or she does not even have to know anything about the specifics of the solution engine.

The borders between direct and implicit representation are certainly not strict, but the specific choice also influences the design of the fitness function. With a direct representation (put resource 5 into consumer 3) chances are high to create illegal solutions. In this case the fitness function could include penalty terms to account for the degree of violation. Besides the problem to properly define the penalty terms, evolution may take a long time until it discovers the first feasible solution. As constraint violations could be detected during construction of the solution, an implicit representation (allocate resource 5) may improve speed. Although, the construction time of a solution will be increased by checking all constraints and applying heuristics to locally optimize allocation, each solution is guaranteed to be legal, and can be evaluated using a straight–forward fitness function.

3 The Mozarteum problem

The first real–world problem we have applied evAlloc to is the generation of weekly timetables for the complete teaching activities at the *Institute for Drama*[2] at the *Mozarteum University of Music and Dramatic Arts in Salzburg, Austria* (shortly named *Mozarteum problem*). Currently, a student working part–time is

[2] `http://www.moz.ac.at/schauspiel/`

employed to arrange the weekly schedules. A specific property of this problem is the distribution of students to single lectures. Primarily, students are member of a class (first, second, and third year). However, students of a class may be also split into groups (rehearsals of a play), or may also receive personal training (singing). These distributions create a number of implicit constraints, e.g., when a single student is in a personal lecture, the group or the class she belongs to cannot have a lecture at the same time.

The Mozarteum timetable problem is defined by:

A set of **lectures** L, a set of **rooms** R, a set of **teachers** T and a set of **students** S. The objective is to schedule the lectures L in a time interval of a week, ranging from Monday to Friday, with the following requirements:

R01 Each lecture unit has identical duration (one hour).

R02 Each lecture needs one room and one teacher to take place.

R03 Designated times may be assigned to rooms and teachers, when they are not available for lectures.

R04 Some lectures can only take place in certain rooms (e.g., dancing or acrobatics). If a lecture has no room preference, it can be assigned to an arbitrary room in R, otherwise the lecture has an ordered preference list of rooms, where it may be scheduled.

R05 There are lectures for a certain group of students or for a single student.

R06 Each teacher can only give one lecture at the same time.

R07 In each room only one lecture can be given at the same time.

R08 Any two lectures attended by an identical subset of students cannot be scheduled at the same time.

3.1 The Application

As mentioned in Section 2 an application engineer (this time the authors) has to build the full application based on evAlloc by performing the following steps.

3.1.1 Data Representation

First, the Mozarteum problem data have to be mapped to the generic data representation of evAlloc introduced in section 2.1.

Lectures This is the resource allocation group with each single lecture modelled as a resource allocation item.

Rooms This is the consumer allocation group with each single room as a consumer allocation item.

Students This allocation group of students is neither resource, nor consumer, but serves as a data base for checking and resolving implicit constraints.

Teachers Like the student group this is also a (data base) allocation group used for constraint calculations.

3.1.2 Problem–specific Code

In order to capture the specifics of the problem at hand, the States of the allocation groups have to be programmed in Java. The `MozarteumCourseState` is a single class extending the reused class *CourseState*, which has already been implemented for test purposes. The specific State contains a timetable and various

attributes describing the lecture. It also incorporates a greedy schedule builder heuristic (fit–first), which is used, when a lecture item is allocated to a room item.

Figure 3 shows the GUI for the State of `Lecture 1`. The left panel shows the problem tree, the right panel shows the selected state with the following attributes (for the sake of brevity we do not mention all of them here).

Fitness Fitness calculation is also distributed among objects. In this case the fitness "Hours" is selected meaning that the State simply returns the number of allocated lecture units, but more sophisticated measures could be implemented and selected here.

Timetable This table is the timetable of the single lecture, where times not available for scheduling can be simply marked by writing any text into the corresponding table cell. Here "Teacher 1" indicates that the teacher of the lecture is not available at these times (actually, the system has marked these times based on the teacher's timetable).

Hours per week The number of lecture units per week.

Max hours per day The maximal number of lecture units per day.

Min hours per day The minimal number of lecture units per day.

Only in these rooms A list of rooms, where the lecture has to take place (empty, if no preference).

Teacher The name of the teacher giving this lecture.

Semester Specifies the class (of students) attending this lecture.

Team Specifies the group of a class attending this lecture (empty, if class lecture).

Student Specifies the single student attending this lecture (empty, if class or team lecture).

3.1.3 Encoding and Negotiator

The encoding of the single chromosome is order–based using a permutation chromosome. Each index of the permutation is assigned to a resource allocation item. The sequence of items constituting an indirect representation is passed to the negotiator, which checks and resolves all constraints associated with a lecture item. The basic procedure is to mark all lecture units in all rooms (consumer items) causing a conflict as unavailable, and then passes the resource to the consumers (in a fixed order) until a consumer (room) accepts and allocates the lecture. If no consumer accepts the resource, it is not allocated. This technique guarantees that all constructed timetables are feasible.

The fitness function simply counts the number of allocated lecture units, which drives evolution to maximize this number and to schedule all lectures. More complex fitness functions could be employed evaluating subjective qualities of the timetable, e.g., the number of free hours between lectures for teachers and students. However, it should be stressed that such properties can be already enforced with the current system by restricting available times before the evolutionary search.

4 Experimental Results

Naturally, it is difficult to assess the quality of the system by objective measures, nevertheless, we try to give an impression of the system's capability based on first tests performed by the actual users at the Mozarteum drama school.

Currently, weekly timetables with 13 rooms, 97 lectures, about 40 students, and 23 teachers are constructed by the system. Without any time constraints on the teachers (each teacher available from 8 a.m. to 10 p.m.) the system does not have any problems to find a solution scheduling all lectures within the first generations of an evolutionary run.

Restricting the availability of teachers from 8 a.m. to 3 p.m a solution with maximal fitness is usually found after 15–20 generations (using basic settings of evolutionary parameters: a population size of 50, a mutation rate of $\frac{1}{l}$ with the chromosome length $l = 97$, a crossover rate of 0.6, and binary tournament selection).

Obviously, practical settings will have different restrictions for different teachers, as many of them have other obligations as well. Tests with these practical settings will be perfomed in the next weeks (and included in a possibly final paper).

A very positive result of these first tests is the time spent to find the solutions. Due to the generality of the evAlloc framework considerable software overhead is introduced, which we expected to have negative influence on the runtime. However, above solutions have been discovered within 20 seconds on a PC with a 1 GHz processor. Considering that the person constructing the schedules manually takes at least a few hours for a complete schedule the benefit is obvious. With more complex settings we can expect a runtime of a few minutes, which also

allows the user to experiment with different settings and compare the results within a short time period.

5 Acknowledgements

We want to thank Prof. Markus Trabusch, chairman of the Institute for Drama at the Mozarteum University of Music and Dramatic Arts in Salzburg, for initiating and supporting the application of evAlloc to a real–world problem. We would also like to thank our student Gregor König for recent implementations of evAlloc extensions.

References

[1] Michael R. Garey and David S. Johnson. *Computers & Intractability: A Guide to the Theory of NP-Completeness*. November 1990. (W.H. Freeman and Company, New York, 1979) ISBN – 0716710455.

[2] Andrea Schaerf and Amnon Meisels. Solving employee timetabling problems by generalized local search. In *Proceedings of the 6th Italian Conference on Artificial Intelligence (AIIA-99), Bolgna, Italy*, pages 493–502, 1999.

[3] A. Drexl. A simulated annealing approach to the multiconstraint zero-one knapsack problem. *Computing*, (40):1–8, 1988.

[4] Emma Hart and Peter Ross. A heuristic combination method for solving job-shop scheduling problems. In *5th International Conference on Parallel Problem Solving from Nature - PPSN V, Amsterdam, The Netherlands*, 1998.

[5] David J. Montana. A reconfigurable optimizing scheduler. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2001.

[6] David J. Montana. Optimized scheduling for the masses. In *GECCO-2001 Workshop: The Next Ten Years of Scheduling Research*, 2001.

[7] Tina Yu and Peter Bentley. Methods to Evolve Legal Phenotypes. In Agoston E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Fifth International Conference on Parallel Problem Solving from Nature*, pages 280–291, Berlin, Germany, September 1998. Springer.

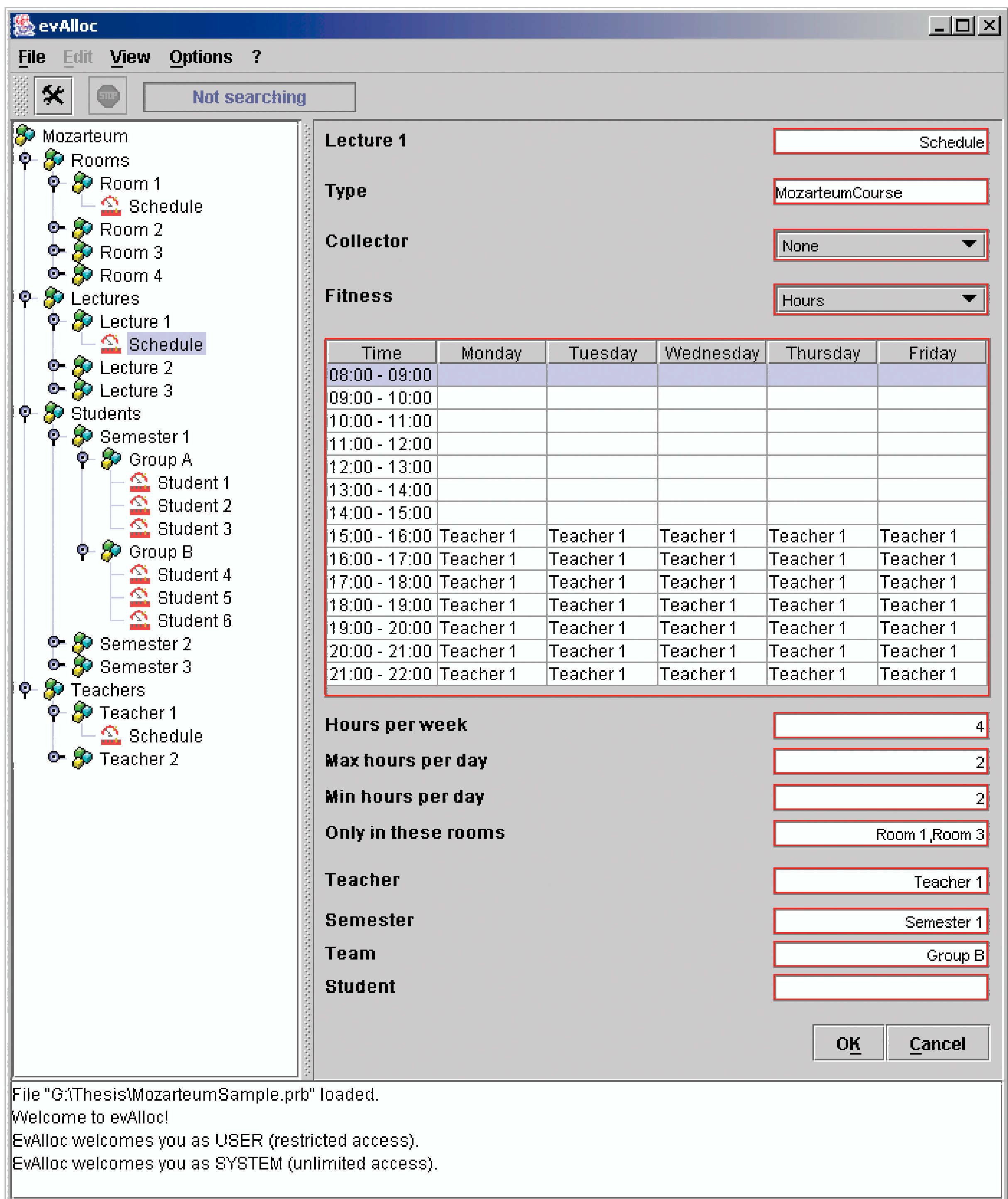

Figure 3: Mozarteum problem type template