An Object Oriented Timetabling Framework

Swee-Chuan Tan
Singapore Technologies Electronics Limited
24 Ang Mo Kio Street 65
Singapore 569061
tansc@stee.com.sg

Abstract: Real world timetabling applications are usually diverse in their problem structures, constraints and algorithms used. We have developed an object oriented timetabling framework that adapts to varied problem structures, and allows for easy and flexible maintenance of timetabling algorithms and constraints. We use the Unified Modelling Language for problem structure representations and Object Constraint Language for constraint expressions. The model is easier to understand as compared to mathematical formulations and results in coherent development from problem specification to software constructions. We also describe an application that we have instantiated from this framework.

1 INTRODUCTION

Timetabling problems can be found in many areas, for example, in sports league, and in educational, transport and employee timetabling. In his survey [23] on educational timetabing, Schaerf presented many problem variants and numerous solution techniques. The main reasons for the large variants are due to differences in problem structures, constraints and objectives. In addition, the type of solution technique often imposes a specific problem structure and most of the solution techniques are only good for certain cases. When there is a change in user requirements, the existing solution technique or algorithm may become unusable. Most timetabling problems, even in some simple cases, are found to be NP-Complete (e.g. see [9], [10]), and the running time of complete search algorithms grow exponentially with respect to problem size. Over the past forty years, researchers have been looking for ways to construct timetables automatically. Many models and solution techniques have been proposed, the PATAT series (e.g. see [5], [6]) are especially devoted to many aspects of timetabling.

As the object oriented (OO) paradigm emerges as a popular software construction approach in the software community, the idea of an object oriented (OO) framework [11] approach to software construction appears to be promising in providing software reusability and maintainability. An OO framework consists of a generic structure of an identified problem domain and provides a set of commonly used abstract and concrete functions. While the generic structure can be adapted into one or more specific application structures, the functions can be extended to solve specific application problems. In this way, a high degree of software reusability can be achieved. Until the late nineties, there have been increasing reports on the use of object-oriented frameworks to solve constraint-based problems. For example, Ferland [12] used the Object Pascal for an assignment type model, and implemented several local search algorithms to show that only minimal code changes are required. Andreatta [3] created a set of object-oriented local search classes in a framework to construct and compare local search heuristics under a common platform. They describe the proposed framework using design patterns and encapsulate different aspects of local search heuristics into abstract classes. They used the framework to build and compare heuristics for the one of the problems in comparative biology. Another example of local search library is LOCAL++, a C++ framework developed by Schaerf [25]. With the so-called *inverse control* mechanism, users of LOCAL++ only need to focus on the problem description and do not have to worry about the implementation details. Later, the authors abandoned the LOCAL++, in favor to a completely different and more powerful C++ framework, the EASYLOCAL++ [16]. They used this framework to solve a number of classical combinatorial problems and applied the tools in practical timetabling (e.g. see [17]). Michel [22] contributed a constraint-based architecture to supply the shortage of local search libraries in the community. Similarly, Fink [14] used an OO framework as a 'stock room' for local search heuristics. So far, most reports in the literature are restricted to frameworks for heuristic constructions and comparisons. In particular, most of the frameworks focus on local search heuristics rather than any specific timetabling problem domain. On the timetabling problems, Gröbner et al. [19] take a more general view. They propose a general timetabling langauge to describe the common, underlying structure of timetabling problems and show how the problem descriptions can be translated into Java programming langauge.

Our perspective of modelling timetabling problems is close to that of Gröbner. Although we feel that it is not possible to build a universal framework for all timetabling problems, we feel that OO frameworks should not be limited to local search heuristics, and can have wider applicability in timetabling. Our proposed timetabling framework generalizes a set of timetabling applications in a common domain rather than any particular class of heuristics. We use design patterns [15] to separate the timetabling heuristics from the timetabling domain objects. In this way, the framework allows a flexible control of algorithms and constraints.

In this paper, we present a conceptual timetabling framework, and show how it can be extended for educational timetabling domain. The framework allows easy switching between several local search heuristics during program execution, and we exploit this capability by implementing a simple hyperheuristic (e.g. see [7], [8]) to select lower heuristics that could improve the timetable solution quality. Finally, we show how an educational timetabling application can be instantiated from the framework. The educational timetabling problem involves assigning groups of students to attend lessons at particular time and venue, and may be taught using certain equipment, and by a particular instructor. In the process of developing the software framework, we modeled the software with the Unified Modelling Language (UML) [4, 13]. The UML is set of OO notations for representing objects in OO systems. We also use the Object Constraint Language (OCL) [27], a precise textual language to express the constraints in the UML diagram. One research question today is how to incorporate constraints as seamlessly as possible into constraint optimization applications such as timetabling. Traditionally, timetable problems are expressed using mathematical models and occasionally using natural language. There are also specialized languages such as the

STTL [20] for specifying timetable problems, describing instances and solutions. Mathematical formalism is rigorous but abstract, and difficult for software developers to understand. Natural language is easy to understand but ambiguous. In software development, these languages cannot reduce the distance from the problem definition stage to the software specification and construction stages. The UML and OCL can shorten this gap, and blend well with our OO framework. We believe that UML/OCL will be a suitable modelling tool for OO timetabling framework.

This work covers the analysis, design and development of an automated timetabling framework for educational domain using the OO paradigm. Section 2 covers the OO timetabling framework. Section 3 describes an example timetabling application instantiated from the framework. Section 4 presents test results and analysis. Section 5 concludes this project.

2 AN OBJECT ORIENTED TIMETABLING FRAMEWORK

The framework is divided into three levels, the conceptual (abstract) level that has no code written, the first level and second level that are implemented in Java.

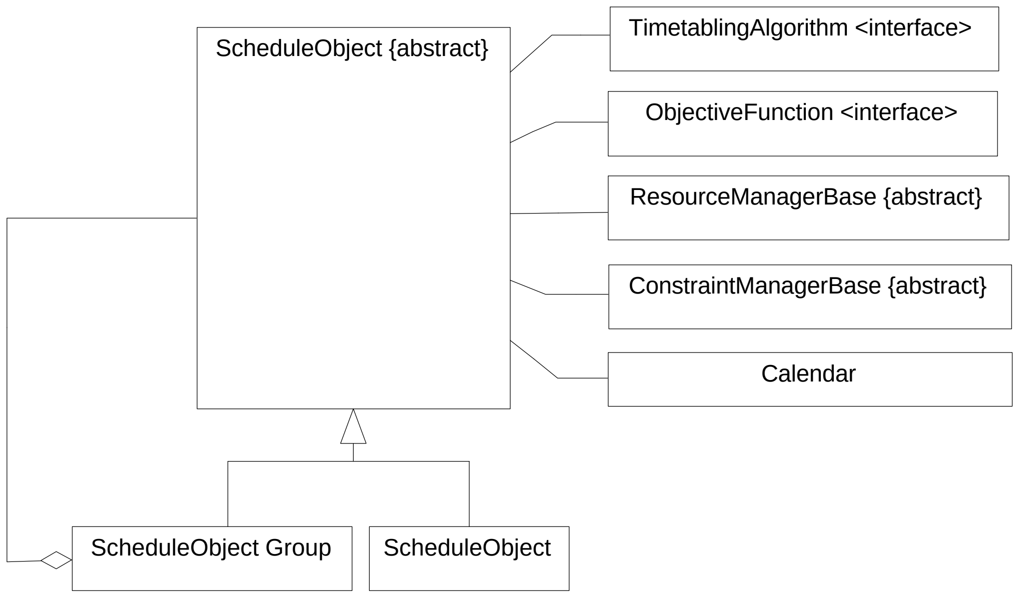
ScheduleObject {abstract}
TimetablingAlgorithm <interface>
ObjectiveFunction <interface>
ResourceManagerBase {abstract}
ConstraintManagerBase {abstract}
Calendar
ScheduleObject Group
ScheduleObject

Fig. 1: The conceptual level framework

The conceptual level framework

Figure 1 shows a composite of schedule objects associated with a set of classes. The *ScheduleObject* represents a task (or a group of tasks), and it allows different *timetabling algorithms* and *objective functions* to be registered. The *ResourceManagerBase* is responsible for managing the resources requirements and resource allocation logic. The *ConstraintManagerBase* manages all the constraints and is in charge of the constraint activation logic. The *Calendar* manages the time domain and keeps track of the availability of the *ScheduleObject*. This pattern succinctly represents a great variety of timetabling problems. However, it can be overly abstract and lack semantic clarity of the overall hierarchical structure. When using the OCL to express the constraints in the model, it is better to present the actual decomposed hierarchical class diagram rather than to use the composite pattern. The rest of this section describes the decomposed model in the first level and second level framework.

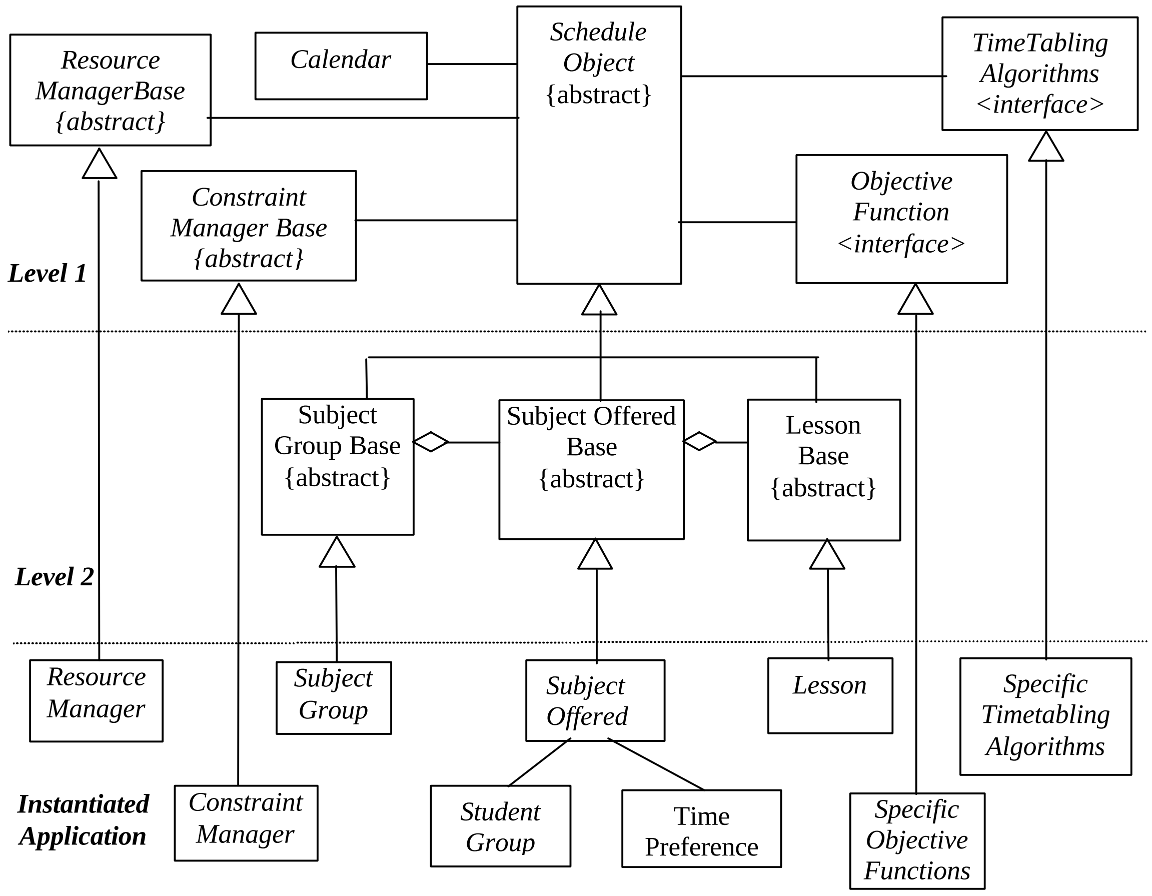
Resource ManagerBase {abstract}
Calendar
Schedule Object {abstract}
TimeTabling Algorithms <interface>
Constraint Manager Base {abstract}
Objective Function <interface>
Level 1
Subject Group Base {abstract}
Subject Offered Base {abstract}
Lesson Base {abstract}
Level 2
Resource Manager
Subject Group
Subject Offered
Lesson
Specific Timetabling Algorithms
Instantiated Application
Constraint Manager
Student Group
Time Preference
Specific Objective Functions

Fig. 2: An overview of the framework

An overview of the actual framework

Figure 2 shows an overview of the actual framework implemented. The first level shows that the *ScheduleObject* is associated with a set of abstract classes and interfaces. The first level is similar to the conceptual level except for the composite. At this level it is not clear how these classes will perform their tasks as we still do not know the exact application to be instantiated. However, we do know the basic responsibilities of these classes and we declare a set of abstract methods to ensure that these methods are implemented at the lower level framework or application. Since our interest is in building educational timetabling applications, our second level framework consists of educational timetabling classes. The *LessonBase* represents a basic teaching activity, the *SubjectOfferedBase* consists of a group of lessons, and the *SubjectGroupBase* consists of a group of subjects. These classes form a basic hierarchy of teaching activities in a school. Finally, the instantiated application consists of concrete classes inherited from the first and second level abstract classes. The exact resource management and allocation logic is implemented in the *ResourceManager*. The *ConstraintManager* defines exactly how the constraints will be activated. There are also specific *timetabling algorithms* and specific *objective functions* registered with the *ScheduleObjects*. Besides, there are some additional classes (e.g. *StudentGroup, TimePreference*) needed in the actual timetabling application model. Some key features of the framework are described as follows.

A Strategy pattern

We use the strategy design pattern to allow different algorithms to be invoked by a *ScheduleObject* during program runtime. Figure 3 shows a strategy design pattern for different algorithms to be executed by a *ScheduleObject*. The *ScheduleObject::setStrategy* method registers a particular algorithm with the *ScheduleObject*. The registered algorithm can then access the objects associated with the *ScheduleObject*. The *ScheduleObject*::*execute* method invokes a kind of *Strategy* that is registered with the *ScheduleObject.* With the Java runtime method resolution, the *setStrategy* and the *execute* method allows a *ScheduleObject* to switch between different algorithms during program execution. This is useful for implementing hyper heuristics, which are heuristics for selecting other heuristics.

We use the strategy design pattern to implement the automatic timetabling algorithms and the objective functions. As seen in Figure 2, the *Lesson*, *SubjectOffered* and *SubjectGroup* are all a kind of *ScheduleObject*, and this means that different levels of timetabling algorithms and objective functions can be implemented on each of these *ScheduleObjects*. When a timetabling algorithm is executed, automatic scheduling is performed on the *ScheduleObject*. When an objective function is executed, the objective function value of the *ScheduleObject* is evaluated.

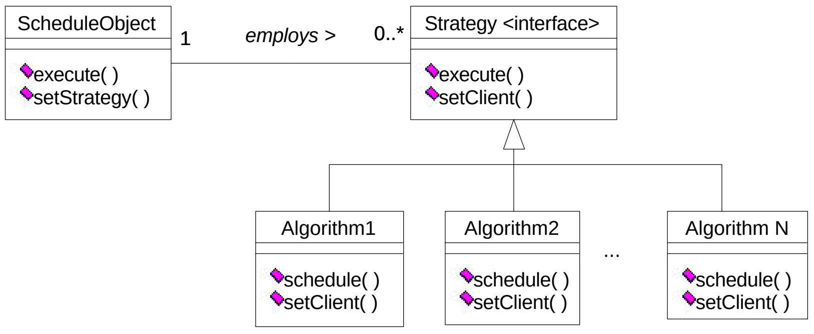

Fig. 3: A strategy pattern for flexible algorithms selection

Implementation of the timetabling algorithms

The kind of timetabling algorithm used depends on the kind of *ScheduleObject*. For the *Lesson* objects, we implemented four simple timeslot search algorithms. The first, CHRONOLOGICAL_SEARCH, searches for a suitable lesson timeslot from the earliest start time until the latest end time of a lesson. By *suitable* we mean a timeslot that satisfies all the constraints. The second, RANDOM_SEARCH, randomly searches a suitable timeslot for a lesson. The third, PREFERENCE_SEARCH, searches for a preferred and suitable timeslot. The fourth, LESSON_EXCHANGE, exchanges the timeslot positions of two lessons, if both lessons satisfy the constraints. The *Lesson* algorithms are the lowest level heuristics and are used by the *SubjectOffered* algorithms. We implemented a constructive algorithm that schedules all the *Lessons* in the *SubjectOffered*. The algorithm, CONSTRUCT, constructs a mini timetable for each *SubjectOffered* by scheduling each lesson with CHRONOLOGICAL_SEARCH. Once an initial solution is in place, it is accepted by the local search algorithms that perform local moves of lessons to timeslots with RANDOM_SEARCH, PREFERENCE_SEARCH or LESSON_EXCHANGE, and then the new neighborhood solution is computed. The decision on whether a neighborhood solution is accepted depends on the local search algorithm used. We have implemented three simple versions of local search algorithms for the *SubjectOffered*, namely the *hill climbing*, *simulated annealing* and *tabu search* (e.g. see [1]).

The first, HILL_CLIMBING, accepts a neighborhood solution (S’) only if S’ is better than the current solution (S). The search repeats until no better solution can be found or when the maximum trials have been reached.

The second, SIMULATED_ANNEALING, is a stochastic approach that simulates the slow cooling of a physical system. This approach to combinatorial problems was proposed by Kirkpatrick [21]. Since its introduction, Simulated Annealing techniques have been studied extensively and later applied in timetabling (e.g. see [2], [26]) with good results. In Simulated Annealing S’ is accepted if it is better than S, otherwise S’ is accepted only if P_{Accept} is > $exp\ (-\Delta C/T)$. P_{Accept} is the acceptance probability, ΔC is the difference between the cost of solutions S’ and solutions S, and T is the temperature that decreases slowly. Initially, T is large and $exp\ (-\Delta C/T)$ is small and most S’ are accepted. As T decreases, $exp\ (-\Delta C/T)$ gets larger and the chance of accepting a lousy S’ become smaller. At this stage of development, we experimented the algorithm at an initial temperature of 50, and a cooling rate of 0.98 and the algorithm produces reasonable outputs.

The third, TABU_SEARCH, maintains a tabu list that stores some earlier moves and forbids subsequent moves on the tabu list. Tabu search was originally proposed by Glover [18], and has been applied in classical or specific optimization problems. In particular, there are many reports on the applications of Tabu Search techniques in the timetabling literatures (e.g. see [24], [28]). TABU_SEARCH accepts a neighborhood solution only if the move is not in the tabu list and S’ is better than S, or when S’ does not deteriorate too far from S.

The three local search algorithms run at the *SubjectOffered* level. At the *SubjectGroup* level, we consider the use of hyperheuristics to schedule the *SubjectOffered*s to optimize the overall timetable solution quality. Hyperheuristics are domain independent heuristics that operates at a higher level of abstraction than the metaheuristics. We implemented the SIMPLE_HYPER, a simple hyperheuristic that selects the *SubjectOffered* heuristic to be used at any given time, depending on the contribution of each heuristic to the timetable solution quality. Firstly, it selects a construction algorithm that creates an initially feasible timetable. Secondly, it improves the timetable solution quality by using the appropriate local search algorithms. There are many ways to select which algorithms to use. The SIMPLE_HYPER chooses the algorithms that can at least contribute to the solution quality. The decision as to when to change an algorithm is dependent on the objective function improvement value F_N at iteration step N, where $F_N \leftarrow \alpha.\ F_{N-1} + \Delta F_N$. The control parameters are α and τ. α is a real value such that $0 < \alpha < 1$ and it defines the amount of importance given to the historical contributions of a heuristic. τ is a threshold value such that if F_N is less than τ, the current algorithm is replaced by another algorithm, otherwise the current algorithm is executed and ΔF_{N+1} and F_{N+1} are computed. The process is repeated until the timetable solution is of acceptable quality or when the maximum trails have been reached.

Implementation of the Objective Functions

As mentioned, different objective functions can be registered with the *ScheduleObject*. A *Lesson* objective function evaluates the solution quality of a lesson. We use the following penalty rules to evaluate a lesson:

- If a lesson is scheduled at a preferred timeslot, the penalty value is zero.
- If a lesson is not scheduled at a preferred timeslot but is just next to the preferred slot, a penalty value of two is given.
- If a lesson is not scheduled at the preferred timeslot and is not next the preferred slots, a penalty value of five is given.
- If a lesson is not scheduled at all, the penalty value is ten.

To compute the total penalty value, the *SubjectGroup* sums the *SubjectOffered*s' objective functions, and each *SubjectOffered* in turn sums the *Lessons*' objective functions. The objective is to minimize the total penalty value.

Resource manager

As seen in Figure 4, the abstract *ResourceManagerBase* declares the basic responsibilities for managing a pool of resources. These basic responsibilities are abstract methods for resources allocation, retrieval, and availability updates. The *ResourceManager* must implement all the abstract methods as defined in *ResourceManagerBase*. For example, the *ResourceManager* implements the actual resource (venue, instructor, equipment etc.) allocation logic for each lesson. The allocation logic involves checking the type of room to be used, the room capacity and the availability of the resources.

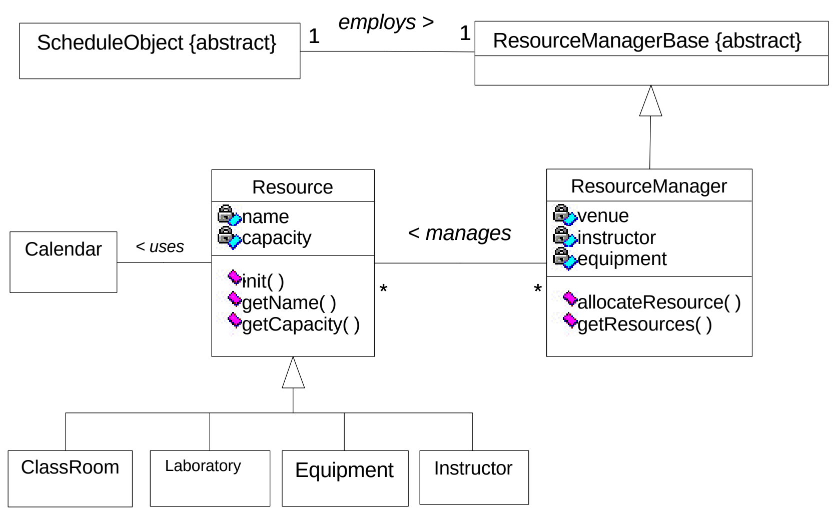

Fig. 4: A class diagram for the resource management section

Constraint manager

As seen in Figure 5, the *ConstraintManagerBase* manages a set of constraints, and decides when and what constraints to enforce during a timetabling process. The *ConstraintManager* has an integer specifying the *cutOffLimit* (which ranges from 1 to 6) and it holds a set of constraints extended from the *GeneralConstraint*. Each *GeneralConstraint* has a Boolean flag *enable* and an integer specifying the *importanceLevel* (which ranges from 1 to 6). If a constraint is enabled and its *importanceLevel* is more than the *cutOffLimit*, the constraint will be validated, otherwise it is not used in constraint validation. The constraints are validated when there is an attempt to modify the timetable schedule. Constraint violations are thrown as Java exceptions so that the violation messages can be logged or reported to the user interface. By arranging the constraints with increasing importance levels, constraints can be relaxed or enforced gradually by adjusting the *cutOffLimit*. Examples of specific constraints include "*Schedule room with correct capacity*", "*Do not schedule already scheduled lesson", "Schedule lesson within earliest start time and latest end time*", and *"Do not schedule lesson across break points"*, …etc.

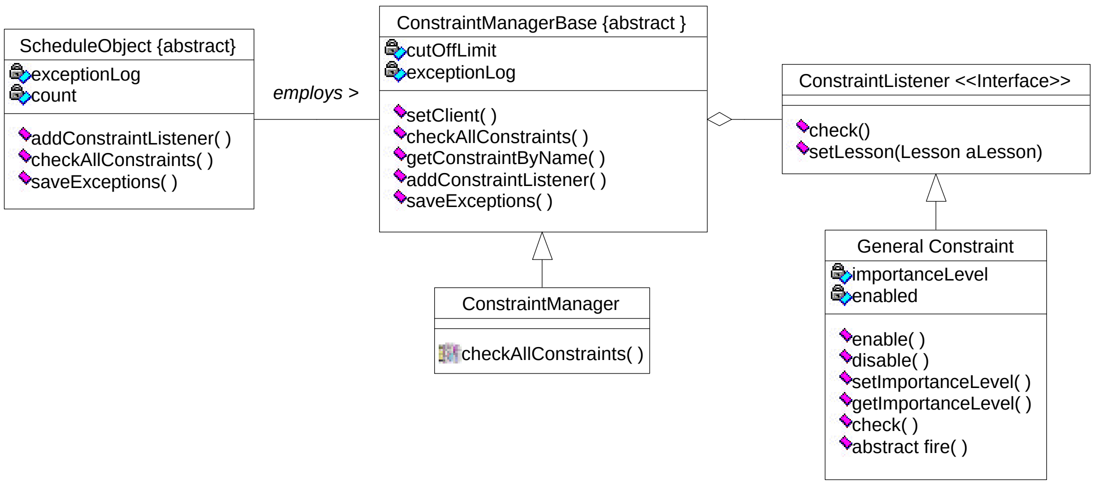

Fig. 5: A class diagram for the constraint management section

3 A PART TIME COURSE TIMETABLING APPLICATION

The problem

We built a prototype application that is instantiated from the framework. We consider a computer school offering part-time courses for working adults. The school has 7 computer labs and 7 classrooms, and they employ a number of part-time contract instructors. Students enroll at different times of the year and the school forms groups of students to take up subjects offered. Each student group has about 10 to 25 students. Each subject offered is conducted once or twice in a week and covers 10 practical and 5 theory lessons. Typically each subject offered lasts from 8 to 15 weeks, depending on how frequently the class is conducted. Currently, there is a maximum of 14 subjects offered over about 4 months, thus resulting in about 210 lessons to be scheduled over 4 months. Constructing the timetable involves entering the course, resource and student information. These include details of the subjects offered, the student groups taking up the subject, the classrooms, computer laboratories, instructors and equipment used. The part-time students attend lessons according to the following time arrangement:

Monday to Friday	1 timeslot: 7:00 PM to 10:00 PM
Saturday, Sunday	1 timeslot: 2:00 PM to 5:00 PM
Public Holiday and Holiday Eve	No Lessons

Table 1: Time arrangment for part time course

This application is to create a timetable for 120 days, that is, 120 timeslots. A timeslot translation table gives the date and time meaning of each timeslot:

Timeslot	Date/Time Details
0	01/Sep/2002 2-5 PM Sunday
1	02/Sep/2002 7-10PM Monday
…	…
119	29/Dec/2002 2-5 PM Sunday

Table 2: Timeslot translation table

Timetabling algorithms are implemented at three levels, namely at the *SubjectGroup* level, the *SubjectOffered* level and the *Lesson* level. During timetable construction, each lesson is allocated with one venue, instructor and equipment. The objective is to allocate the timeslot of each Lesson according to the predefined time preference as specified in a *TimePreference* class. Subject to:

- No *Resource* can be involved in two *Lessons* at the same time
- No *ScheduleObject* (i.e. *Lesson*, *SubjectOffered*, *SubjectGroup*) can be involved in two or more *Lessons* at the same time
- No *StudentGroup* can be involved in two or more *Lessons* at the same time
- All *Lessons* must be scheduled within the *Earliest Start time* and the *Latest End time* of the *Lesson*
- Use only venues with enough seating capacity for any *StudentGroup* attending the *Lesson*
- Do not schedule an already scheduled *Lesson*
- Do not schedule a *Lesson* on *BreakPoints* (e.g. public holidays)

The above constraints can be conceptualized using OCL and implemented using Java programming. As an example, we show how one of the constraints is defined below:

Step 1) Express the constraint in natural langauge:

- All Lessons must be scheduled within the Earliest Start time and the Latest End time of the Lesson

Step 2) Define the constraint with an OCL expression as shown in Figure 6.

OCL Expression

```
Context Lesson inv:
Lesson.allInstances->forAll(
       l: Lesson
       | l.isScheduled()
       implies (l.actualStart >= l.earliestStart
       and ((l.actualStart + l.duration) <= l.latestEnd)
)
```

Fig. 6: An OCL expression that states that a lesson must be scheduled within earliest start and latest end

Step 3) Realize the OCL constraint using Java programming as shown in Figure 7.

Java Realization

```
class ScheduleLessonWithinEarliestStart_LatestEnd extends GeneralConstraint {
  ScheduleLessonWithinEarliestStart_LatestEnd (Lesson aLesson) {
    super(aLesson);
  }

  public boolean fire()  throws Exception {
    boolean result = true;
    if (myLesson.isScheduled) {
       if ((myLesson.planStart < myLesson.earliestStart)||
          ((myLesson.planStart+myLesson.duration-1)>myLesson.latestEnd)){
               result = false;
         throw new ScheduleLessonWithinEarliestStart_LatestEnd_Violation();
       }
    }
    return result;
  }
}
```

Fig. 7: A Java program that realizes the OCL constraint stated in Figure 6

An extension of OCL to include objectives

Besides the hard constraints, soft constraints can be represented using an extended version of OCL. The explanation of the extension is as follows:

- **Constraint Type**

The keyword ***soft*** represents the type of constraint which is soft, rather than invariant.

- **Objectives**

There are two possible objectives: ***maximize*** or ***minimize***. The keyword ***maximize*** (resp. ***minimize***) represents a maximization (resp. minimization) objective. When applied to a collection, it intends to ***maximize*** (resp. ***minimize***) the size of the resultant collection. When applied to a collection operation (such as count() or sum()) or any other values returned by the OCL expresson, it intends to ***maximize*** (resp. ***minimize***) the value returned. Thus, the OCL extended to support ***soft*** constraints will have the general form:

Context class ***soft***:
    Objective (the size of the resultant collection in OCL expression)
or
Context class ***soft***:
    Objective (the value returned by the OCL expression)

As an example, we consider the objective to be the minimization of the number of lessons scheduled at undesirable timeslots, which can be declared as:

(Objective) Try to minimize the number of Lessons scheduled at undesirable timeslots.

```
Context SubjectOffered soft:
{
        minimize self.getTimePreference().computeUndesirableTimeSlots()
}
```

Fig. 9: An extended OCL constraint that expresses the optimization objective

The *computeUndesirableTimeSlots* () method assigns the penalty values to lessons based on the rules described earlier.

4 TEST RESULTS

A prototype application is developed using Microsoft Access^{TM}. It allows users to maintain the timetabling data and generate the timetable. Timetabling data includes courses, students, resources and lessons requirements. Users can edit time preference for each subject, update the availability of lessons and resources, edit the constraints, objective functions, and the timetabling algorithms. It takes a resonable 6 to 7 minutes on a Pentium-III ® 600 MHz personal computer to construct a timetable with 210 lessons. We tested the framework application with 3 local search algorithms and observed the average running time and average solution quality over different input sizes. The test results are tabulated as follows:

	Running Time (Minutes)			**Quality (Objective Function Value)**		
Number of Lessons	***Hill Climbing***	***Simulated Annealing***	***Tabu Search***	***Hill Climbing***	***Simulated Annealing***	***Tabu Search***
56	0.3	0.3	0.4	0	0	0
140	0.9	0.6	0.9	0	0	0
224	6.0	4.9	6.7	118	157.5	121
280	9.3	7.6	10.1	704.3	694.3	701.3
420	10.4	11.2	10.2	2385	2410	2378.9

Table 3: Running time and objective function test results of different algorithms

Analysis of test results

With the algorithm parameters unchanged, the three algorithms show similar average running times and similar average solution qualities with respect to different number of lessons. The similarities occur mainly because the three algrothims use the same local search framework. However, these algorithms will behave differently if there are significant differences in their individual heuristics and parameters. Furthermore, we observed that the penalty value is zero for up to about 200 lessons. However, beyond 200 lessons the solution quality deteriorates very quickly, with 224 lessons having about 10% of the lessons (according to the penalty rules described earlier) scheduled at undesirable timeslots, and any input sizes more than 224 lessons have the penalty values grows exponentially. This phenomenon occurs because the available preferred timeslots are already very limited with 224 lessons. While the main theme of this work is not on novel algorithms, the results demonstrate that the framework is feasible with different algorithms, constraints and objective functions and can be used to build practical educational timetabling applications. To ascertain that the framework is usable at a larger scale, we expanded the problem size 5 times, that is, there are 1120 lessons to be scheduled over 600 timeslots. The average time taken is 42 minutes and the total penalty value is about 800. Although we have yet to breakdown the component values that make up the total penalty value, we know that it means a worst case of 160 (out of 1120) lessons not being scheduled to the preferred timeslots, and we are very confident to improve this situation in the near future. We also tested that the SIMPLE_HYPER is effective in choosing a proper heuristics. We intentionally assign a heuristic, TABU_SWAP, a tabu search that uses lesson exchange as its local move. With an α of 0.95 and a τ (threshold) of 0.5, the SIMPLE_HYPER is able to identify that the TABU_SWAP cannot contribute at all for 60 iterations, and another algorithm (TABU_SEARCH) is used to replace the TABU_SWAP. After the replacement of the algorithm, the total penalty value begins to drop and finally produce a reasonable timetable. Our future work is to develop more robust hyperheuristics that are domain independent, and yet able to produce reasonable result.

Evaluation of the program

This program has the following benefits and limitations:

a) Flexible Constraints Maintenance
Most commercial timetabling applications offer a set of predefined constraints and objectives. Users can enable/disable or change the weight of the constraints and objectives, but are unable to add new constraints or modify existing ones when the need arises. Adding new constraints to such programs normally involves modifying the original source code and recompiling the whole set of code and re-testing it, and this is costly! The current framework program allows the user to define new constraints and objectives using OCL, and build separately compiled Java code for the constraints, and there is no need to recompile the whole framework.

b) Flexible Timetabling Algorithms Maintenance
Commercial timetabling applications normally prescribe a fixed timetabling algorithm with the assumption that the algorithms will be always effective with a fixed set of constraints. However, real life systems are rarely static. When a timetabling system allows flexible constraints maintenance and supports a range of applications in the same domain, it is important to allow the users to change the algorithms as and when it is needed. The current timetabling framework demonstrates that it is easy to switch between different local search algorithms. In addition, programmers can add/remove/modify algorithms without changing and recompiling the invariant part of the framework.

c) Still requires programmer for maintenance
The maintenance of constraints, objective functions and algorithms would still require the expertise of programmers, and such work cannot be done by the end users alone. In particular, the programmers must understand the UML class diagram, be able to express constraints using OCL and realize the constraints using Java programming.

d) Does not suggest the best algorithm to use
Although the SIMPLE_HYPER is able to choose a suitable timetabling algorithm, it does not necessarily choose the best algorithm. Furthermore, some processing time is required for the software to 'test and see' which algorithm is suitable. We are currently considering approaches to allow the framework to learn the effectiveness of each algorithm given an input instance.

5 CONCLUSION

We have presented the idea of an OO framework approach, which allows us to build timetabling applications that lead to an extensible structure, and allow flexible algorithm selections and constraints maintenance. We have constructed an OO framework for the educational timetabling domain, and the framework was described using UML. We have demonstrated that the OCL is useful in expressing the constraints, and provides intuitive realization of the constraints with Java programming. We have demonstrated this by creating a Part Time Course Timetabling System. The results obtained showed that the OO framework approach is feasible for timetabling application developments and suggest wider applicability to other timetabling problem domains.

REFERENCES

[1] Aarts, E., Lestra, J.K., "Local Search in Combinatorial Optimization", John Wiley & Son, Chichester, 1997.

[2] Abramson, D., "Constructing school timetables using simulated annealing: sequential and parallel algorithms", Management Science, 37(1), 98-113, 1991.

[3] Andreatta, A.A., Carvalho, S.E.R., Ribeiro, C.C., "An object-oriented framework for local search heuristics", Technical report, Department of Computer Science, Catholic University of Rio de Janerio, 1998.

[4] Booch, G., Rumbaugh, J., Jacobson, I., "Unified Modeling Language User Guide", Addison-Wesley Object Technology Series, 1999.

[5] Burke, E., Erben, W., editors, "Proceedings of the 3rd International Conference on the Practice and Theory of Automated Timetabling", number 2079 in Springer Lecture Notes in Computer Science Series, 2001.

[6] Burke, E., De Causmaecker, P., editors, "Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling", to appear in the Springer Lecture Notes in Computer Science Series, 2003.

[7] Burke, E., Petrovic, S., Qu, R., "Case-Based Heuristic Selection for Examination Timetabling", Proceedings of the 4th Asia-Pacific Conference on Simulated Evolution and Learning, 2002.

[8] Cowling, P., Kendall, G., Soubegia, E., "A Hyperheuristic Approach to Scheduling a Sales Summit", Proceedings of the 4th Metaheuristics International Conference, 127-132, 2001.

[9] de Werra, D., "An introduction to timetabling," European Journal of Operational Research, 19, 151-162, 1985.

[10] Even, S., Itai, A., and Shamir, A., "On the complexity of timetabling and multicommodity flow problem". SIAM Journal of Computation. 5:691-703, 1976.

[11] Fayad, M.E., Schmidt, D.C., "Special issue: Object-oriented application frameworks", Communications of the ACM, 40 (10):32-87, 1997.

[12] Ferland, J.A., Hertz, A., Lavoie, A., "An Object-Oriented Methodology For Solving Assignment-Type Problems with Neighborhood Search Techniques", Operations Research Vol. 44, 347-359, 1995.

[13] Fowler, M., Scott, K., "UML Distilled", Addison-Wesley, 1997.

[14] Fink, A., Voß, S., "Reusable Metaheuristic Software Components and their Application via Software Generators", Proceedings of the 4th Metaheuristics International Conference, 637-642, 2001.

[15] Gamma, E., Helm, R., Johnson, R., Vlissides, J.,“Design Patterns: Elements of Reusable Object-Oriented Software”, Addison Wesley. October 1994.

[16] Gaspero, L.D., Schaerf, A., “EASYLOCAL++: an object-oriented framework for the flexible design of local search algorithms and metaheuristics.”, In Proceedings of the 4th Metaheuristics International Conference (MIC'2001). 2001.

[17] Gaspero, L.D., Schaerf, A., “A case-study for EasyLocal++: the Course Timetabling Problem”, Research report UDMI/13/2001/RR, University of Udine, October 2001.

[18] Glover, F., Laguna, M., "Tabu search." In: C.R. Reeves (ed.) Modern Heuristic Techniques for Combinatorial Problems. Blackwell, Oxford, 70 - 150.

[19] Gröbner, M., Wilke, P., "A General View on Timetabling Problems", Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling", 221-227, 2002.

[20] Kingston, J.H., "Modelling Timetabling Problems with STTL", Proceedings of the 3rd International Conference on the Practice and Theory of Automated Timetabling, 433-445. 2000.

[21] Kirkpatrick, S., C. D. Gelatt Jr., M. P. Vecchi, "Optimization by Simulated Annealing", Science, 220, 4598, 671-680, 1983

[22] Michel, L., Hentenryck, P.V., "A constraint-based architecture for local search", Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, 83-100, 2002.

[23] Schaerf, A., "A survey of automated timetabling", Technical Report, CWI-Amsterdam. 1995.

[24] Schaerf, A., "Tabu search techniques for large high-school timetabling problems", Technical Report, CWI-Amsterdam, 1996.

[25] Schaerf, A., Lenzerini, M., Cadoli, M., "LOCAL++: A C++ Framework for Local Search Algorithms", Software Practice & Experience, 30(3), 233-256. 2000.

[26] Thompson, J., Dowsland, K., "General Cooling Schedules for Simulated Annealing Based Timetabling Systems", Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling, 1995.

[27] Warmer, J., Kleppe, A., "The Object Constraint Language: Precise Modelling with UML", Addison-Wesley, 1999.

[28] White, G. M., Zhang, J., "Generating Complete University Timetables by Combining Tabu Search with Constraint Logic", Proceedings of 2nd International Conference on the Practice and Theory of Automated Timetabling, 268-277, 1997.