On-line timetabling software

Florent Devin · Yannick Le Nir

Abstract Timetable design is a really important and difficult task. Timetable hand-building consumes a lot of time. In this paper we address two main difficulties of automatic timetabling, that is data acquisition and timetable computation. The former task is made using new advanced technologies in the area of Rich Internet Application. This offers very powerful, and easy to use, interfaces to acquire data. The latter task is the computation of the timetable itself. We use the constraint programming and one implementation in swi-prolog to compute the timetable. Finally we show some results of our application on a real case study.

Keywords Timetabling · CSP · Prolog · Java framework · ZK · Google API

1 Introduction

In this paper we present a timetabling software. Timetabling application can be split into two different parts, the design of a valid solution and the data acquisition. Many operational software need to hand-build a timetable. Computing helps are only given to verify constraints and to acquire input data.

To acquire data, we create an RIA[1]. This choice allows us to provide an original solution for timetabling. First we decide to use web services to allow us using a particular algorithm to solve the problem. Moreover by using web services we are able to interact with Google Calendar. At last, this use allows the software to be built in our IT system. The figure 1 shows the general architecture of our software. This distributed approach, and using an RIA allows us to delegate data acquisition as we will see.

Florent Devin · Yannick Le Nir
EISTI, 26 avenue des Lilas, 64062 Pau Cedex 9
Tel.: +33 5 59 14 85 34
Fax: +33 5 59 14 85 31
E-mail: florent.devin@eisti.fr

E-mail: yannick.lenir@eisti.fr

[1] Rich Internet Application

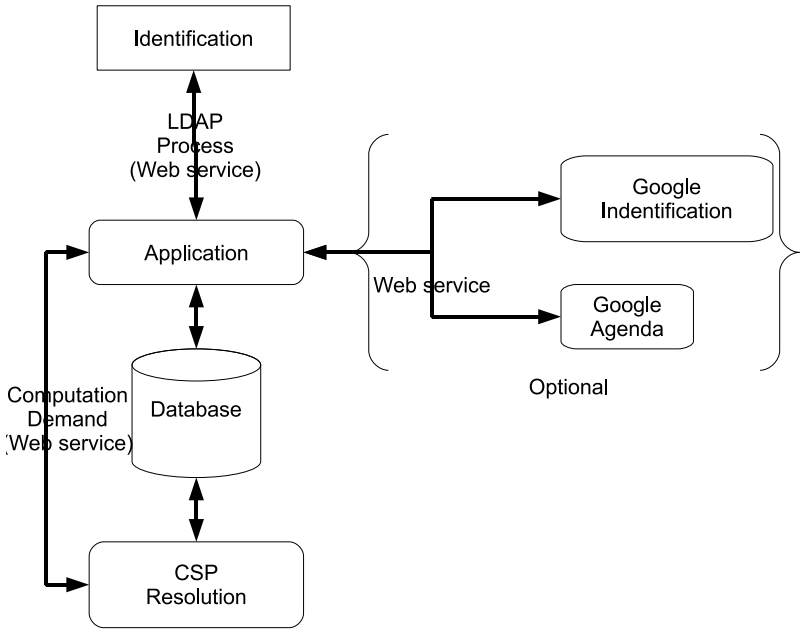


Fig. 1 General architecture system

The design part, in our solution, is made with an automated task as the solution of a CSP[2]. Such approach based on CSP has already been studied for examination and lectures timetabling in school and university (Abdennadher et al 2007)(Abbas and Tsang 2001). We present in the last sections of this paper an instantiation of our model in a similar way (university timetabling) with some computational results under global parameters variation. Other computational models can be used, as mentioned in (Qu et al 2009). We decide to choose CSP, that is easy to use and very powerful for timetabling applications (Wallace 1996), especially in our general architecture where compatibility and performance issues are not problematic.

2 Definitions

In this section, we introduce some definitions that we will use to present our solution for an on-line timetabling software in next sections.

The first notions we have to define are *timetable* and *time slot*. The timetable is defined as a consecutive list of time intervals. A time slot is the minimal time interval we can find on the timetable. Its duration is fixed and then it is only specified with its starting time.

Then, we have to define the *resources* we need to access, that is in our case the resources that describe the context of the timetable. It is specific to every timetable but belongs to one of the following categories:

[2] Constraint Satisfaction Problem

– main resources: elements to be planed in the timetable (lectures, talks, meetings, ...),
– static resources: elements that are already linked to main resources before the computation (peoples, holidays, ...),
– dynamic resources: elements that will be linked to main resources by the computation (rooms, materials, ...).

All these resources are the *descriptive resources* of the timetable. In our application, they are collected in a relational *database*.

Now, we can define the *availability* of the different descriptive resources of the database. The timetable is made of time slots, and then an availability can be associated to each resource as a starting time slot and an ending time slot. We can also define *unavailability* as the complementary of an availability. The (un)availabilities will be considered as *constraints* in the following, mainly in the section on CSP.

The last notion we have to define is the *timetabling*. It consists to an assignment of all the main resources of the database on the timetable for a specific period, with respect of all constraints on descriptive resources.

Some more specific definitions will follow in the next sections. However, these specifics terms are not a prerequisite for understanding.

3 Rich Internet Application for Timetabling

3.1 Motivation for rich interface

Whilst there is no need to argue for a tool for timetabling, we have to discuss why we have to create a rich interface. In an ideal world, all constraints are known long before they occur. In this same world, the constraints do not often change. But this never happens in reality, where very often constraints are modified. There is a real need to change the constraints, whatever the real motivation; for example, we need to provide a tool for specifying unavailability[3].

In this case, we have two choices:

– A centric application: This means that there is only one person who creates or edits the timetable. This amounts to saying also, that only one person takes responsibility for validating or invalidating the constraint. But this step is a little tricky, because you must contact the person who has created the constraint.
– A distributed application: That means you delegate checking constraints to the user.

The former approach has several disadvantages. Such as the problem of knowing the validity of the constraints. The latter approach allows all users to input or delete their constraints. This is a real advantage for the distributed approach.

Once we have chosen a distributed approach, again there are two choices:

– a classical third party application;
– a Rich Internet Application.

Rich Internet Application (RIA) does not require any installation on the client side. Also, neither a particular operating system nor a particular software. RIA allows users

[3] A constraint for a contributor can be an unavailability

to use the application without any requirement. Nevertheless using an RIA implies that you have to look after the security of your software(Lehtinen 2009), mainly if this one is accessible from the Internet.

3.2 Implementation

ZK is an open-source Ajax Web application framework written in Java(Seiler 2009). It uses the server centric approach(Yeh 2007) so that the communication between components is done by the engine. Security process is involve, synchronisation with a database is easier. In the other hand as ZK can use Java, we are able to use Hibernate framework. Also ZK provide a framework for mobiles(Yeh 2007). Even if we do not, for instance, provide a mobile interface this can be very useful for several contributor.

Timetabling involve a lot of contributors. Some of them will have some constraints to input, modify or delete, others will have to create the planning, etc. All of them need to use the RIA. In addition, we want to keep the job process in a single location. This can be done if you use a framework that keeps it on the server. For all these reasons, and (Yeh 2006), we choose to use the ZK framework[4].

3.3 Presentation

The entry point is the RIA. Currently in our application, users[5] have different roles:

- User: a contributor who is allowed to submit some constraints on his own timetable and view all timetables (rooms, class, users, etc).
- Admin : someone entitled to accept or refuse the constraints created by a user. The admin can also generate a timetable and export it into all contributors' Google Calendar[6], if they want to.

3.3.1 User

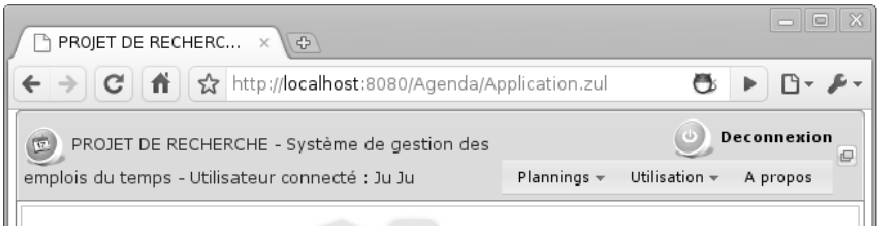


Fig. 2 User welcome screen

At present users have two options to enter their constraints: either through our application or through Google Calendar application. If they choose the second option,

[4] ZK framework : `http://zkoss.org`
[5] users : someone who have to use our application. We use the term of contributor as well.
[6] We will see the usage of Google Calendar, later on.

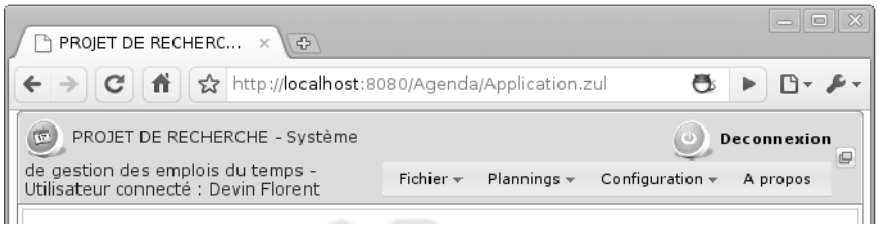


Fig. 3 Admin welcome screen

they have either to provide to the administrator a link to their Google Calendar, or to log in into our application once to save this URL. Once this is done, the users may put their constraints in their own calendar. Later on, the *admin* will validate or invalidate users' constraints. Validation by the *admin* is required by the policy of our institution and to avoid any abuse. To allow the *admin* to make better decisions, in case he has to invalidate a constraint, we plan to use a scale for fixing the degree of unavailability. As soon as the *admin* creates the timetable, the corresponding time slot will appear in all users' calendar. The same operation can be done in our application. There is also an interface to create, modify or delete any users' constraints as is shown in figure 6.

3.3.2 Admin

The *admin* work load is significantly reduce thanks to our application.

Creating a timetable by hand is an extremely time consuming tasks, which might take up to 3 days a week. This is a huge task, because of many reasons, for example: *A videoconferencing can be used for teaching.* This is a really strong constraint because it occurs in two different places. On another hand, in our institution a school term is about 16 weeks long, during which many courses last for about 20. Each course has different contributors and durations in a week, and in a period. This involves creating a different timetable for almost every week. On top of this, contributors' constraints may change according to their other activities, rendezvous, and so on.

Fig. 4 Lecture time-line

By using our application the task of the *admin* comes down to putting all lectures in a time-line, as is shown in figure 4. By doing this, the *admin* also specifies the number of courses per week, number of time slots devoted to practice, ... This simply task takes roughly one hour per term. Then he has to specify which contributor will give what lecture, and also which features are required, and for which class the lecture is addressing as is show in figure 5. This step may take up to four hours once. This is what we can call the initialisation phase.

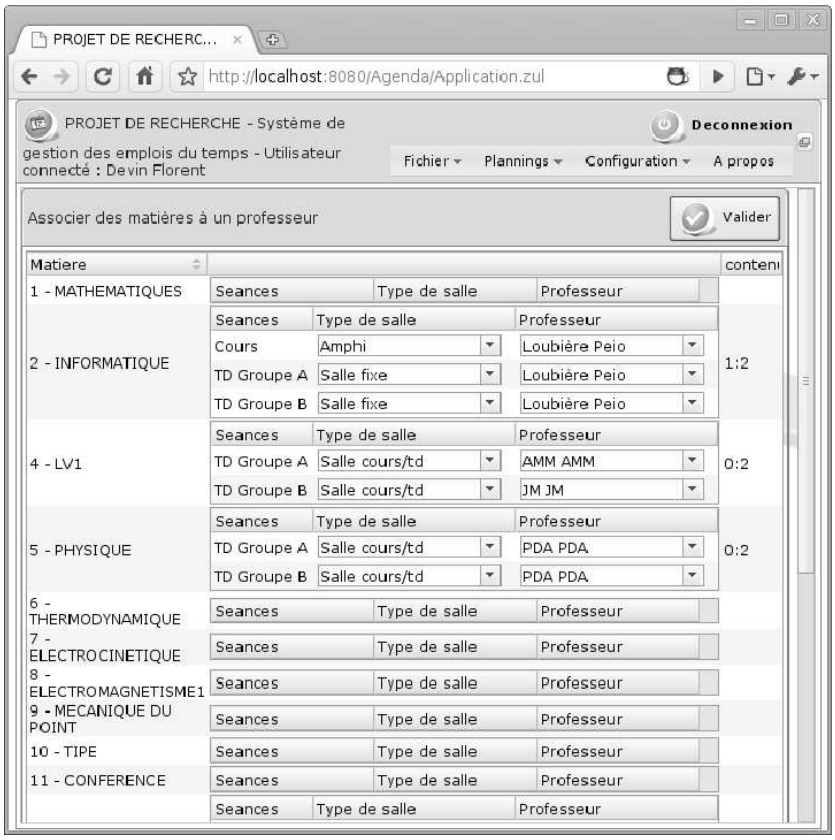
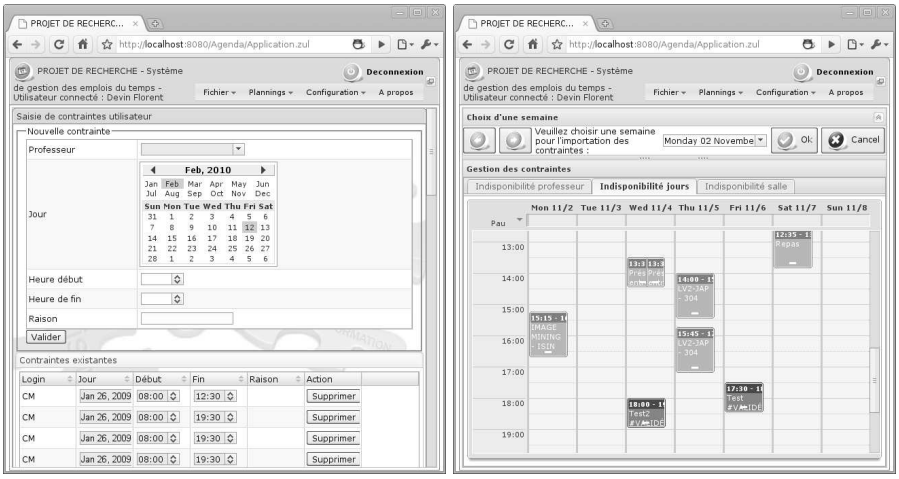


Fig. 5 Courses, features, contributor association

Now we have all the elements to compute an intermediate solution. The result is a timetable which does not take into consideration the users' constraints. If no users' constraint has been specified, this is a valid timetable. Otherwise the *admin* has to validate or invalidate users' constraints. To simplify this step, our application presents two different screens.

One is textual screen, figure 6(a), and another one is a dashboard, figure 6(b), which partitions all constraints into three categories:

– rooms (un)availability constraints;



(a) Constraints textual input screen shot (b) Constraints dashboard screen shot

Fig. 6 Constraints input

- contributors' constraints, as we have seen before;
- free session, holidays, which are used to set some empty time slots for student activities, or holidays.

For each category, the dash board uses a color code to show the state of the constraint. There are three possibilities:

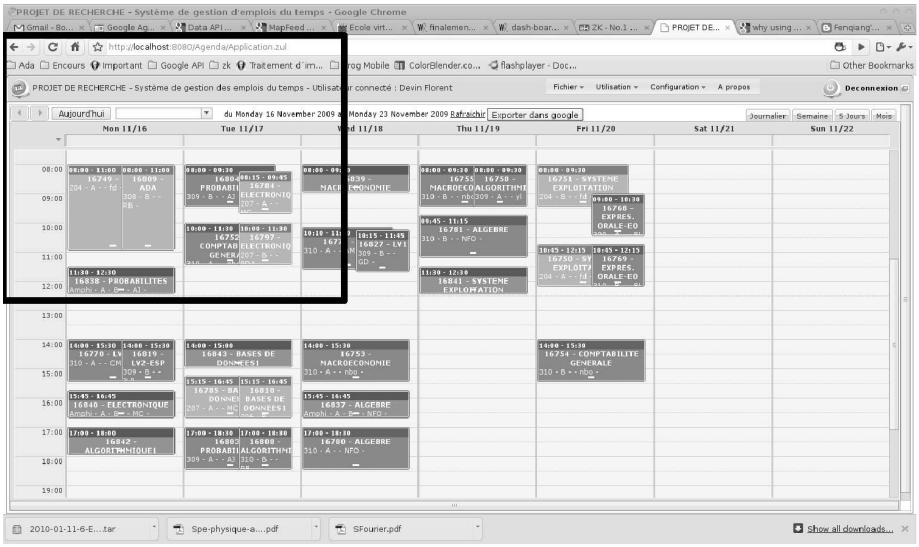
- Green : the constraint has already been validated. This usually means a recurrent constraint. The *admin* can still invalidate it.
- Orange : the constraint has never been validated. The *admin* must validate it if he wants to take it into account.
- Red : the constraint has been validated, but the contributor has changed it afterwards.

Another possibility is to fix a time slot. This will later be taken by the computation as a hard constraint. Eventually, the admin can create a real timetable, by invoking the computational service as mentioned before. Figure 7 shows a computed timetable involving all the required constraints.

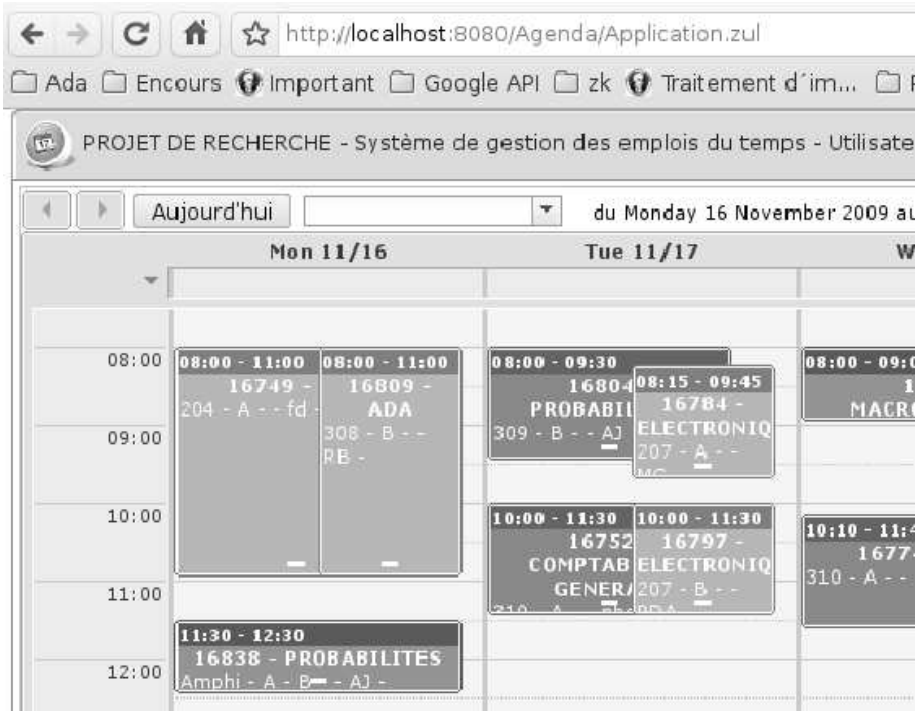
4 Distributed approach

4.1 Motivation for using web services

Creating a new application entails deploying it in an existing IT system. In this system, there is always a user identification process. This process can be used as a service. That was the first reason why we decided to use web services; on the other hand, we want to create an RIA, but we have to compute a timetable. To solve the timetabling problem, as mentioned above, a backtrack search algorithm is used. This algorithm was implemented in Prolog. This implies that the RIA, and the Prolog program have to interact. We choose to use the web services technology for doing this.



(a) A compute timetable



(b) A partial zoom of a timetable

Fig. 7 Timetable

In addition, some contributors may wish to use their own calendar. In our institution there is no shared calendar, but we plan to use Google Calendar soon, which

many contributors are already using.. So we decided to allow the contributors to use their own calendar. This implies that the contributors' constraints are generated in two different ways. Some can simply put their constraints on Google Calendar, the others may use our application once again; we need web services[7] to import and export some slots. Some others reasons to use web services will be explained later.

4.2 Export and import from Google Calendar

4.2.1 Google API

If there are two main components that access the database, 1, there is one which does not access the database. In our application, we have a component that can export from or import to a Google Calendar. This component does not directly interact with the database, but it can modify the database through the RIA. Google releases an API to use a lot of their components. This API allows our contributor to use their own calendar.

4.2.2 Constraints acquisition

A major feature of our application is the acquisition of the users' constraints. This can be done by the RIA, or by putting the constraints in the Google Calendar. The only restriction is to have a particular calendar, which is used to show unavailability. This calendar will be used later to input the compute time slot. The contributor has to put his constraints on Google calendar. He can use all of the Google features. These constraints will be validated by the *admin*. When the *admin* checks the users' constraints, he has the choice to validate or to invalidate it. Both of the choices will inform the user by putting a message in the appropriate time slot.

To perform this kind of operation, the contributor has to allow our application to modify his own calendar. Ideally, the contributor creates a particular calendar, and uses this one to manage his professional time slot.

4.2.3 Timetable visualisation

When we have full access to a contributor's calendar, we are able to create a full timetable. For instance, we have created a specific user. This user has several calendars, one for each room, one for each class, and possibly one for each contributor. This allows when needed, to visualize all timetables with all possible views. Of course, the compute timetable can only be viewed by classes, but we provide any possible view.

As the contributor has a Google calendar, his own timetable is created on it. This allows a convenient view for all who need it.

[7] As Google calendar is accessible via web services.

5 Database modeling for Timetabling

5.1 Required resources for timetabling

The main goal of timetable generation is to compute time slots from descriptive resources. Such resources are needed to provide the context model we want to use. For example, to compute the timetabling of lectures, we need to know all information dealing with lectures such as:

- who will follow this lecture
- who will teach this lecture
- how long is this lecture
- which material is needed

Less specifically, we can split such information in two distinct categories: inner and outer properties. Inner properties deal with information directly linked to lectures (duration, starting time, fixed time). Outer properties deal with information linked to external resources (people, materials, calendar). Therefore, the global data architecture is made of resources associated to time values. We can assume that time values associated to each resource contain an interval of availabilities. Then, the database should contain tables for each type of resource and at least a field for its availability. Therefore, to compute a time slot associated to a main resource, we then first have to collect all the associated availabilities that can be found in the records of the associated resources. Without loss of generality, we can assume that in this particular case, the values that we have to compute (time slots), can be represented by integers corresponding to an interval of times between 1 and N, where N depends on the number of time slots of the timetable. This value can change with the granularity of the timetabling, that is the difference between two consecutive time slots. With this modeling, we can transform our problem in a constraint satisfaction problem on finite domains (Apt and Zoeteweij 2007), what is detailed in section 7.

6 Database implementation

6.1 Motivation to use a central database

Once we have all constraints, we need to compute a timetable which complies with all of them. We need to save these constraints somewhere. This can be done by using a database, XML or anything that stores data. As the computation is done by Prolog, we have to provide an access to this data.

A central point to notice is that our application use the web services to communicate. This means that you design several small applications and assemble them to make a bigger one. The communication between each part of the application has to send or receive some data. Possibly we could use a data feed. It is really simple to send the data to the computational part, and it is also as simple to receive the result. But for providing the users with an interface, we have to keep the generated data. Also, we have to keep the users' entries. That means that we need something to save the data. The best solution for us was to use a database. Indicating this has changed the approach of our application. We choose to put the database at the center of the application.

Also, using a central database allows the RIA to modify data, while Prolog is computing a timetable. Why can this be done ? The database itself prevents multiple modification on the same data. Therefore once Prolog has gathered the data to compute the timetable, they are not missing anymore, while modifying another data. As the computation is done by invoking a web service, we can focus the computation on a particular week, and then begin to modify this week. In fact, the computation accesses the data to get *all* the constraints in one shot, then computes the timetable and then commits the result to the database. As a web service is an asynchronous method, we can run the computation, and leave it. As the same time you may modify the data you need. But if you want this new data to be considered, you have to rerun the process.

On the other side, we have the RIA that has to use the database. Since ZK framework is a java framework, we can use Hibernate tools to do the mapping from database to objects. By using this technology, we can save time(Minter and Linwood 2006) when creating the RIA. It also provides us with a way to automatically create an RIA for a CSP.

6.2 Shared Database for asynchronous communication

Currently, our application is made up of several *black boxes*. Each of these has to access and modify the database. For now, there are two main components which use the database : the RIA, and the computational program. The benefits of doing this is that we can keep running the service at all times. There is no need to run all parts of our application for it to work. You can use only the RIA to update the data, or use only the computed part to create a timetable.

We use a web service to order the computational part to run. Whilst the web service allows us to mix Java and Prolog in one application, it has some disadvantages too. Among them, there is the timeout problem. This problem can occur at various times, depending on the system, the network . . . Again using a central database helps us to resolve this problem. Thus, we communicate via the database, when the RIA asks to the computational part, this simply returns an acknowledgment message. Then, later the RIA will check the database to see if the computational program has finished computing the timetable. This approach has several advantages. One of them is that we can use the RIA without waiting for the end of the computational timetable. Another one is that the computational program can put some statistics in the database . The RIA just has to query the database. Then we can create a report of the timetabling.

To simplify the work both of the component use an ODBC to interact with the database. For Prolog we use a classical ODBC, and for ZK we use hibernate. Using the ODBC allows us to change quite dynamically the database server. It also provides us with a convenient way to access to the data. We do not have to consider the multiple access. This job is done by the ODBC. We can have concurrent access, the processing is solved by the database itself.

In this section, we now explain how we can make the design part of timetabling using CSP on finite domain. We first define CSP and then formulate timetabling problem in this context. Finally we give an example of the use of our model with an instantiation in real case of university timetabling from which we explain some computational results.

7 CSP on finite domain

7.1 Definitions

A constraint satisfaction problem (CSP) is modeled with a triple $P = (X, D, C)$ where $X = \{X_1, \cdots, X_n\}$ is a finite set of variables to assign, D is the domain function of each variable, that is $(D(X_i))$ contains all the possibles values of variables. From this point forward, we will consider to finite domains. The last element of the triple P is a finite set of constraints $C = \{C_1, \cdots, C_m\}$. A constraint is a relation between sub-sets of variables $W \subseteq X$, and a sub-set of values $T \subseteq D^W$. A mapping is a set of pairs (variable-value): $A = \{(X_j \leftarrow v_j)\}$ with $X_j \in X$ and $v_j \in D(X_j)$. A mapping is total if for each variable there is a value assigned. It is valid with C if every relation of C_i is true for all variables in A. A solution to a CSP is a total and valid mapping.

7.2 Modeling of Timetabling in CSP

With this data description and computational model, we can describe the timetabling problem in CSP as follow:

- $X = \{X_1, \cdots, X_n\}$ is the finite set of time slots we have to assign to entities. X_j is the starting time of j^{th} entity. The number of entities (n) is obtained from the database.
- $D(X_i)$ is the interval of possible values for the i^{th} entity. It depends on the availability of the entity, that is also obtained from the database.
- $C = \{C_1, \cdots, C_m\}$ is the set of all constraints on variables and can be split to the following categories :
  - scattering constraint checks that among all pairs of distinct variables from a subset $S \in X$, there is a break of at least l_d:
    $scattering(S, l_d) \equiv \forall X_i \forall X_j \ ((X_i \in S) \wedge (X_j \in S) \wedge (X_i \neq X_j)) \Rightarrow ((X_j > (X_i + d_i + l_d)) \vee (X_i > (X_j + d_j + l_d)))$
    where d_i is the duration of the i^{th} entity
  - overlapping constraint checks that among all pairs of distinct variables from a subset $S \in X$, there is an overlap if we want a break of at least l_d:
    $overlapping(S, l_d) \equiv \forall X_i \forall X_j \ ((X_i \in S) \wedge (X_j \in S) \wedge (X_i \neq X_j)) \Rightarrow ((X_i < X_j < (X_i + d_i + l_d)) \vee (X_j < X_i < (X_j + d_j + l_d)))$
  - relation constraint checks that among variables of all couples from a subset $P \in X^2$, there is a relation r:
    $relation(P, r) \equiv \forall X_i \forall X_j ((X_i, X_j) \in P) \Rightarrow r(X_i, X_j)$
  - separation constraint checks that around a specific value,v_b, there is a break of at least length l_b between all pairs of variables from a subset $S \in X$:
    $separation(S, v_b, l_b) \equiv \forall X_i \forall X_j \ ((X_i \in S) \wedge (X_j \in S) \wedge (X_i < v_b) \wedge (X_j > v_b)) \Rightarrow (X_j > (X_i + d_i + l_b))$.

Therefore computing time slots consists of four consecutive steps:

1. Requests:
   Extraction from the database of all the different entities we need to apply constraints and their availabilities. We can build the X part of the CSP modeling.

2. Domains:
   Affectation of interval values to the variables from X, that is the D part of the CSP modeling.
3. Constraints:
   Application of the different constraints to each group of entities. We build the C part of the CSP modeling for timetabling.
4. Computation:
   Computing times slots for each entity, as a solution of the obtained CSP problem in the two previous step.

8 Prolog implementation

Our application is built upon a SOA architecture. The CSP part is implemented in swi-prolog and communicates with the RIA part using Web services and with the database using classic ODBC. We use the *clpfd* library of swi-prolog to implement the CSP. This library deals with finite domain, thus we can compute time slots as consecutive integers on the interval $\{i_1 \cdots, i_n\}$, where i_1 and i_n are the first and last available time slots of the timetabling period.

We can express all the constraints in this library using arithmetic or domain constraints and then apply the "labeling" predicate to obtain a solution to the CSP. This implementation follow the classical model of constraint programming (Jaffar and Maher 1994)(Frühwirth and Abdennadher 2003)

9 Model instantiation

9.1 A concrete example for lectures timetabling

Finally, we end this section with an application of our model to the generation of lectures in a university U during a week S. The resources are the following:

- lectures and their properties (duration, type of classroom required, week, teachers, groups of students),
- teachers and their availabilities,
- groups of students and their size,
- type of classrooms including their size and equipment.

9.2 Requests

In the first step of the computation, in this example, the resources we need to collect to apply constraints to, are the following:

- general parameters of the system:
  - list of time slots: $lts(U, S)$
  - list of students groups: $lg(U, S)$
  - list of teachers: $lt(U, S)$
  - list of type of classrooms: $lc(U, S)$
  - list of classrooms of a specific type C: $lct(C)$

- lectures lists:
  - list of lectures for a specific students group G: $llg(G)$
  - for a given group G, the list of pairs of lectures such that first one has to be given before the second one: $lpl(G)$
  - list of lectures for a specific teacher T: $llt(T)$
  - list of lectures with the specific classroom type C: $llc(C)$
- availabilities list:
  - list of availabilities for a specific teacher T: $lat(T)$
  - list of availabilities for a specific students group G: $lag(G)$

9.3 Domains

In the second step, we assigned interval values to all variables (lectures):

- lectures are assigned to time slots:
  $\forall G(G \in lg(U,S)) \Rightarrow (llg(G) \in lts(U,S))$
- teacher availabilities:
  $\forall T(T \in lt(U,S)) \Rightarrow (llt(T) \in lat(T))$
- group availabilities:
  $\forall G(G \in lg(U,S)) \Rightarrow (llg(G) \in lag(G))$

9.4 Constraints

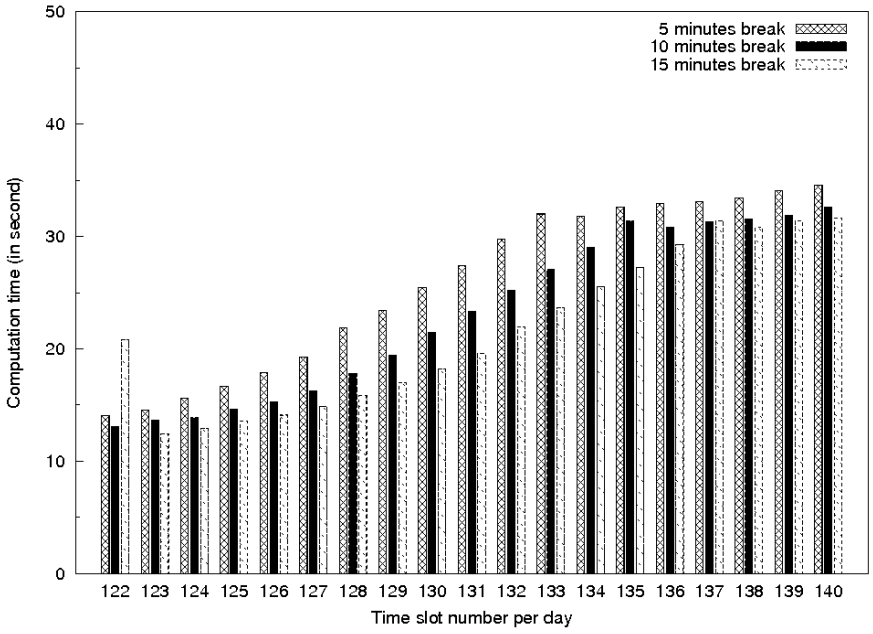
The third step of the computation is now an instantiation of constraints schemes 7.2:

- lesson lectures are always take place before practice lectures for a specific group:
  $\forall G \in lg(U,S), relation(lpl(G), <)$
- lunch break of at least duration d around time t for a specific group:
  $\forall G \in lg(U,S), separation(llg(G), d, t)$
- break of at least duration d between two lectures of a specific group:
  $\forall G \in lg(U,S), scattering(llg(G), d)$
- break of at least duration d between two lectures of a specific teacher:
  $\forall T \in lt(U,S), scattering(llt(T), d)$
- the number of overlapped lectures with classroom type C is less or equal than the number of classroom of type C:
  $\forall C, \forall T, ((C \in lc(U,S)) \wedge (T \subseteq llc(C)) \wedge overlapping(T, l_b)) \Rightarrow (|T| \leq |lct(C)|)$

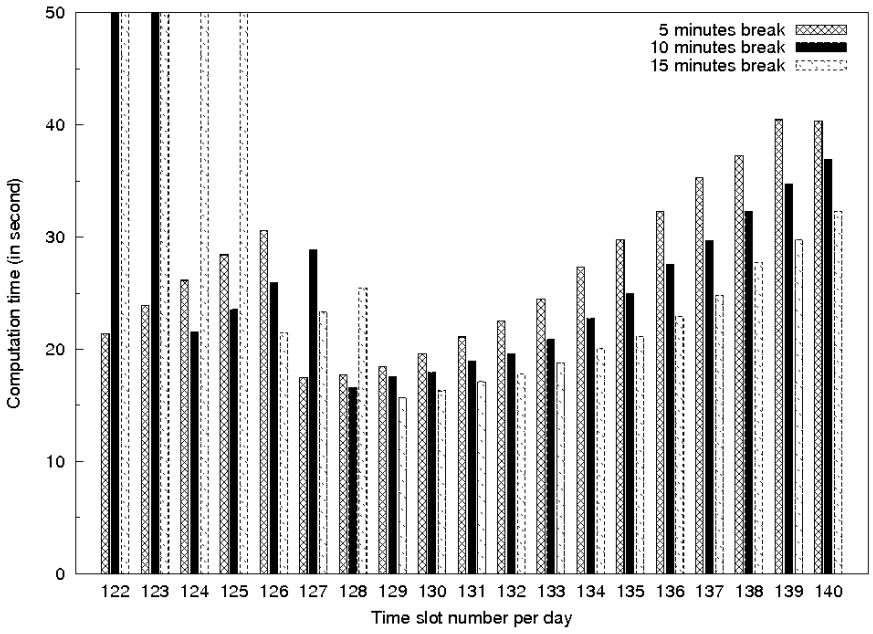
10 Results

In this part we present results on computation time of the CSP part of the application with different values for the following parameters:

- time slot number per day (from 120 to 140)
- break between two consecutive lectures (5, 10 or 15 minutes)
- lunch break (90 or 120 minutes)



(a) Lunch break: 90 minutes



(b) Lunch break: 120 minutes

Fig. 8 Computation times

The computation was done for 3 different classes ($C1$, $C2$ and $C3$). $C1$ was made of two subgroups ($C1_a$, $C1_b$), as well as C3 ($C3_a$, $C3_b$). $C1$ consumes 590 time slots. Among them 36 was shared between the two subgroups. $C1_a$ and $C1_b$ got 282 different time slots. $C2$ only consumes 372. Finally $C3$ had 84 shared time slots, and each subgroup was using 192.

There were only one global constraint : "Thursday afternoon is devoted to sport, so we do not want to have courses on this particular period".

The computation was launched once. In figure 8 the 50 seconds value, means that the algorithm was not able to find an answer in the accorded time. It does not means obligatory that there is no solution. Running more than once the computation does not change the look of the results neither the interpretation that we can make on it.

There are some interesting things to notice :

- In the general case, 5 minutes break is longer to compute than 15 minutes break. This can be explained by the fact that the definition domain is shorter for 15 minutes break than for 5 minutes break. This means that there is less case to test. You can find this case in figure 8(a) from 123 slots per day, and in figure 8(b) from 129 slots.
- In the specific case, we can find a solution for 5 minutes break, but not for 15 minutes break. This means that if you want a solution you have to less restrain the domain. You can find this case in figure 8(b) before 126.
- If we are not in the two previous case, the results may be inverted. We can explain this. There is some solution, but they are more frequent with 5 minutes break, than with 15. So the algorithm is able to find one quicker for 5 than for 15 minutes break.

11 Conclusion

In this paper we have presented a modern approach to timetabling, mixing the advantages of RIA for data acquisition and the power of constraint programming to find a solution. To validate this approach, we have created an application for university timetabling from which we have computed results with some parameters variations. The obtained application validate the choices we have made. This software is a very useful tool to design timetable, as expected. But it also provide a tool to analyse the complexity variation of timetabling under realistic data sets. In future works, we plan to design other type of timetabling with our approach and to improve the result interpretation with many more data, that is now possible thanks to our global architecture.

References

Abbas A, Tsang E (2001) Constraint-based timetabling-a case study. Computer Systems and Applications, ACS/IEEE International Conference on

Abdennadher S, Aly M, Edward M (2007) Constraint-based timetabling system for the german university in cairo. In: INAP/WLP, pp 69–81

Apt KR, Zoeteweij P (2007) An analysis of arithmetic constraints on integer intervals. Constraints 12(4):429–468

Frühwirth T, Abdennadher S (2003) Essentials of Constraint Programming. Springer Verlag

Jaffar J, Maher MJ (1994) Constraint logic programming: A survey. J Log Program 19/20:503–581

Lehtinen J (2009) Ria security. In: JAZOON09
Minter D, Linwood J (2006) Beginning Hibernate: From Novice to Professional. Apress
Qu R, Burke EK, Mccollum B, Merlot L, Lee SY (2009) A survey of search methodologies and automated system development for examination timetabling. J of Scheduling 12:55–89
Seiler D (2009) Ria with zk. In: JAZOON09
Wallace M (1996) Practical applications of constraint programming. CONSTRAINTS 1:139–168
Yeh TM (2006) Zk ajax but non javascript
Yeh TM (2007) Server-centric ajax and mobile