Towards Constraint-Informed Information Systems

Irene Rodrigues Nuno Matos Salvador Abreu
Universidade de Évora and CENTRIA,
{ipr,nmatos,spa}@di.uevora.pt

Rebecca Deneckere Daniel Diaz
Université Paris 1 and CRI,
{Rebecca.Deneckere,Daniel.Diaz}@univ-paris1.fr

***Abstract*—Declarative techniques such as Constraint Programming are very useful in modeling complex requirements. They have the added benefit of being executable specifications and, when properly tuned, high-performance ones.**

In this paper we argue that Information Systems ought to include constraint-based techniques in their design and implementation. We support this claim by introducing tools based on constraint programming, which we apply to an actual use-case: the academic timetable construction and maintenance problem, as developed at the University of Évora. The system we built was implemented using the GNU Prolog language.

Moreover, Constraints have the potential to describe global properties that a model must observe, which makes them a semantically very interesting extension to the capabilities of present model-driven techniques and tools.

Keywords—Information System, Constraint Programming, Logic Programming, Timetable.

I. Introduction

UML [5] has brought forward a collection of methodologies for designing incremental and scalable information system applications. Its different facets have struck an effective balance between expressiveness, incrementality and levels of abstraction, to the point where it sets the grounds for data modeling and executable code generation.

Large scale information systems may be specified, modeled and designed using UML-based instruments such as Visual Paradigm or the Eclipse UML Tools. These techniques and tools greatly simplify and improve the reliability of IS design and maintenance. However, one aspect remains daunting from the UML standpoint: how to establish relations between seemingly independent components of a system, in such a way as to ensure the correctness of the global state? It is possible to specify *integrity constraints,* rules which control and aim to guarantee data validity, but these appear as checks or validations: the state generation and mutation process cannot directly benefit from this sort of annotation as it is not generative.

There have been several attempts to specify and implement higher-level frameworks, such as the Object Constraint Language (OCL), with the intention of embedding these into the UML modeling formalism, most notably to be able to make assertions about the model being described. OCL, however, is most of the time dead code: there are hardly any implementations thereof and it is not commonly used in modeling tasks.

Generically, *constraints* have long been recognized as a useful concept in modeling languages (eg. for requirements engineering) but they are also used within regular applications, in the role of combinatorial problem solver or optimizer. As a consequence of its expressiveness and flexibility, *Constraint Programming* (CP) has been successfully used for modeling in several domains: [22], [13] air traffic flow management [6], planning [11] or product-line models [18], [9], [19], to name but a few. More than a decade ago, CP was even identified by the ACM as "one of the strategic directions in computer research" [3].

Constraint Logic Programming is a declarative technique related to Constraint Programming, which adds the possibility of constructing and driving the constraint solving process by means of logic goal satisfaction.

Our claim is that constraints are useful in designing and implementing information systems, and that the latter will benefit significantly from being *constraint-informed*. To make and illustrate the point, we shall be describing an actual running example – the development of a timetabling application.

The declarativeness of Constraint Programming makes it suitable for even the most complex modeling: constraints may be thought of as a set of relations (e.g. equations), while retaining an efficient runtime execution.

The rest of the paper describes the running example – a University-wide timetabling information system – both its structure and some implementation aspects, which we proceed to critically analyze. We then describe the Constraint Programming and the GNU Prolog system and continue with how we took to modeling parts of the application with constraints. Finally we put this work in context and make considerations regarding further evolution.

II. A Case Study: the University Timetable Information System

A. The Information System

Timetabling is a difficult problem which must be dealt with year after year, by every school or University. We now present an extension of the Information System of the University of Evora, related to the specification and deployment of timetables. As this functionality was not initially planned for in the Information System, we set out to specify and implement it as an independent subsystem.

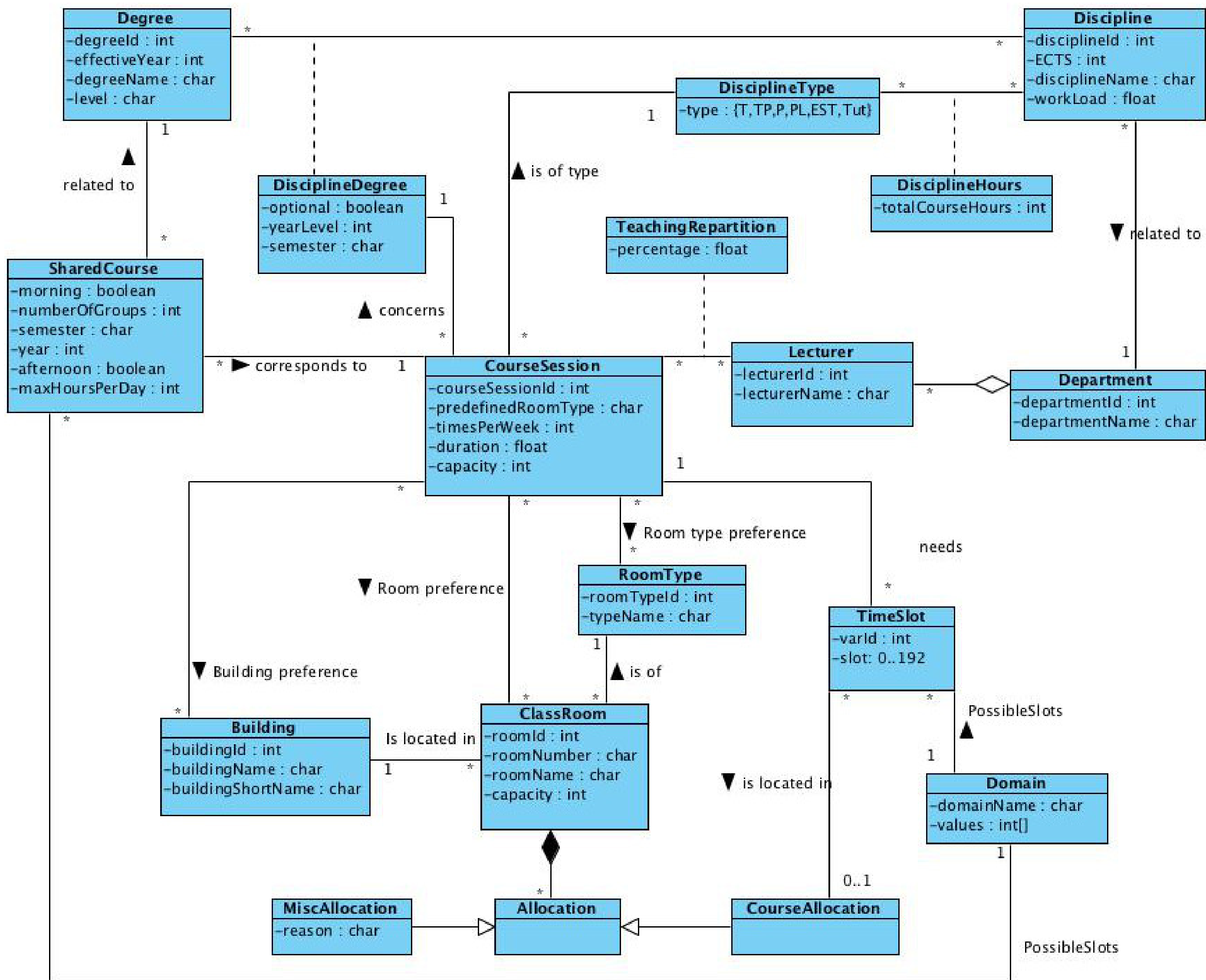

Fig. 1. Timetable information system model

The basic statement of the problem is rather simple: allocate classrooms to course sessions satisfying obvious constraints (only one course session per room at a given time, a lecturer cannot be in two places at once). But in practice several other requirements have to be taken into account. For instance, some sessions need specific classroom specificities, features or furniture. Because the University is physically dispersed, both in and out of town, with buildings being up to 20 Km apart, the transit time may become significant and must be taken into consideration. It is also important to try and satisfy lecturers' preferences (which can be diverse: a lecturer may prefer to teach on a given weekday, or at a given location or a given discipline, chalkboard vs whiteboard, etc.) Usually, these kinds of constraints are not integrated into the timetable IS. Most of the time, it is the person responsible for the timetabling who, by knowing those preferences, tries to satisfy part of them. In order to highlight this "knowledge" we decided to integrate most of the timetable related information into the IS. Figure 1 presents the IS model using UML formalism. It is composed of 3 main parts.

1. Educational information. The University offers several degree programs (e.g., Bachelor of Computer Science). Each degree includes a set of disciplines (e.g., UML modeling) that a student enrolled in this degree program must attend over the year. Each discipline is related to a given department and consists in lectures, tutorials, etc. Each discipline can be taught to several degrees: e.g., UML modeling is both optional on the 3rd year of Bachelor of Computer Science (at semester 1) and mandatory on 1st year of Master Mathematics (at semester 2). Each of these degree-specific disciplines gives rise

to several course sessions. These course sessions can occur several times per week. The total hours of a course session can be divided into different lecturers. Moreover, a degree may share courses with other schools and the classrooms are considered a common pooled resource, shared by all courses.

2. Resources. The University is physically composed of a set of buildings offering classrooms. A classroom has a type depending on its specificities (e.g., amphitheater, for chemistry work practice, etc.).

3. Timetabling Information. This last part includes needed information for timetabling. A timetable *unit of time* is an interval of 30 minutes. Hence, our weekly timeline goes from 1 to 192, corresponding to the 192 half-hours from Monday 8:00 AM to Saturday 11:30 PM. For instance a two hours class will have a timetable with 4 consecutive unit of time. In order to restrict allocation of rooms to some given set of unit times, we introduced the notion of *domain* which is simply a set of integers (in 1..192) describing all allowed half-hours. A domain is identified by a name which can be reused to reference the same set of values. For instance the domain "Morning", which refers to the interval of time 8:30..12:30, from Monday to Saturday consists in the following set of unit of time: $\{1..10, 33..42, 65..74, 97..106, 129..138, 161..170\}$. The Information System needs to know the preferences to take into account when computing the timetable. There are three kinds of preferences currently integrated to the IS for a given course sessions: building and/or classroom and/or classroom type. It is also important to store the occupancy of all classrooms. A classroom can be used, for a given time slot, either for a course session or for other purpose (e.g., meeting). The most important attribute is `Slot` in `TimeSlot` concept. `Slot` $= 0$ if not yet allocated else it contains the assigned unit of time.

Usually timetabling is a task which is handled outside the Information System model, as it depends on several informal aspects such as the lecturers' personal preferences about a specific classroom, building or the required day period (morning, afternoon, evening, weekday, etc.).

It is not possible to use UML to express global constraints. For instance one cannot express that the sum of all course session hours of a course equals the total number of hours of the course, over the semester.

B. Implementation

The need to express complex program logic with access to information stored in an information system led to the development of the ISCO framework [2], a seamless integration of object-relational databases into a constraint logic language, which has been used to implement parts of the University of Evora's general-purpose Information System.

It is not the purpose of this article to introduce the ISCO language, but a short review of its main features is useful for the discussion ahead. An ISCO program may transparently access data from several distinct sources in a uniform way: all will behave as regular Prolog database predicates, even though they may reside in separate relational databases or SPARQL agents. Some relevant advantages ISCO holds over competing approaches are its ability to concurrently interface to several legacy systems, its simplicity and high performance.

1) Accessing Databases: Predicates that are to be associated with an external representation must be declared. This is necessary because DBMSs need to have table fields *named* and *typed* and none of this information is derivable from a regular Prolog predicate.

```
class lecturer.
    name:       text. key.
    department: department.id.
    qual:       text.
```

Fig. 2. ISCO class`lecturer`

A class `lecturer` can be declared in ISCO as in Figure 2. This defines predicate `lecturer/3`, which behaves as a database predicate but relies on an external system (e.g. an RDBMS) to provide the actual facts. Class declarations may reflect inheritance, although that isn't shown in the previous example. Several features of SQL have been mapped into ISCO: keys and indexes, foreign keys and sequences to name a few. The purpose has always been to ensure efficient execution while retaining the simplicity of Prolog.

Class predicates may be used similarly to Prolog database predicates, i.e. they may have full CRUD [17] functionality:

- insertion of new facts (Create),
- non-deterministic sequential access to all clauses (Read),
- updating of an existing fact (Update),
- removal of specific facts (Delete)

These operations may include constraints over their arguments to limit the tuples they apply to. These may be actual FD constraints or more specific syntactic constructs, designed to provide a useful set of features to tap into the potential efficiency provided by the RDBMS: for example, there are notations to specify solution ordering or substring matching.

2) Contextual Logic Programming: is a clean and practical object-oriented extension to Prolog [20], [1]. A *named* set of clauses is called a *unit*, and is similar to a class in OO languages. We emphasize the OOP aspects by means of a stateful model, by means of unit arguments, which are like instance variables. Consider a unit called `lecturer` to represent some basic facts about the teaching at a University:

```
:-unit(lecturer(N, D, Q)).

name(N).   % access predicate
dep(D).    % access predicate
qual(Q).   % access predicate

lecturer :- lecturer@(name=N, dep=D, qual=Q).
```

Fig. 3. CxLP unit lecturer

The difference between the code of figure 3 and a regular logic program is the first line that declares the unit name and introduces *unit-global variables* `N, D` and `Q`. Consider another unit to represent information about courses, in figure 4: A collection of units is designated as a *contextual logic program*. An ordered, bound, collection of units is called a *context* and is

```
:-unit(course(N, C)).

lecturer(N).
course(C).

course :- course@(lecturer=N, course=C).
```

Fig. 4. CxLP unit course

akin to an object, in OO speak. Each computation has a notion of its *current context*. Goals may be executed in contexts, which is like sending a message to an object.

To build contexts, we have the *context extension* operation given by the operator `:>`. The goal `U :> G` extends the *current context* with unit `U` and calls goal `G` in the new context. For instance, to gather the academic qualification of the person who taught the Aspect-Oriented Programming course (AOP), we could launch the goal:

```
lecturer(N,_,Q) :>
 course(N,aop) :> (
  course,
  lecturer)
```

Here, we start by extending the initially empty (`[]`) context with unit `lecturer`, obtaining context `[lecturer]`. This context is again extended with unit `course`, yielding the context `[course, lecturer]`, and it is in the latter that goal `course, lecturer` is evaluated.

III. Constraint Programming

"Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it." (E. Freuder [10])

Constraint Programming (CP) is a recent software technology for declarative description and efficient solving of (very) difficult combinatorial problems. CP has proven to be successful in many relevant application areas such as scheduling, planning, vehicle routing, resource allocation, configuration, networks or bio-informatics.

The key idea of Constraint Programming is to allow the user to reason about partial information in terms of constraints without worrying about how they will be satisfied. This declarative aspect of CP is very important: the user specifies what relationship must hold without any requirement of providing a procedure to ensure the satisfaction of the relations. This is possible because constraints are handled by a dedicated constraint solver that is responsible for ensuring the consistency of the set of constraints. The solver is then seen as a "black-box" by the user.

Constraints have emerged in the context of the Logic Programming (LP) in late 1980's. LP is the ideal framework for constraints since 1) its variables are really mathematical unknowns and 2) it is able to deal with non-determinism (important to find for instance all solutions satisfying the constraints). In fact, LP is nothing else but a particular case of Constraint Programming. The basic idea of Constraint Logic Programming (CLP) [15] is to replace unification by constraint solving over a particular domain of interest (Finite Domains, real numbers, trees, lists, finite sets, etc.) [16]. CLP combines the declarativity of logic programming and the efficiency of constraint solvers. CLP opened Logic Programming up to a wide range of real-life applications [13]. However, Constraint Programming is not restricted to CLP and several solvers are available (as libraries) for imperative languages like Java or C++.

Constraint Programming (CP) has proved to be very successful for Problem Solving and Combinatorial Optimization applications by combining the declarativity of a high-level language with the efficiency of specialized algorithms for constraint solving, borrowing sometimes techniques from Operational Research and Numerical Analysis, see [22], [14] for general surveys, [10] for a quick introduction and [16]for a complete description of the paradigm of Constraint Logic Programming on which this work has been based.

A constraint is a logical relation between several variables restricting the values these variables can simultaneously take.

Constraints naturally enjoy several interesting properties :

- constraints may specify partial information, i.e. constraint need not to uniquely specify the values of its variables,
- constraints are non-directional, typically a constraint on two variables X, Y can be used to infer a constraint on X given a constraint on Y and vice versa,
- constraints are declarative, i.e. they specify what relationship must hold without specifying a computational procedure to enforce that relationship,
- constraints are additive, and the solving process is order-independent (constraints can be added incrementally).
- constraints are rarely independent, typically they share variables.

A. Constraint Satisfaction Problems

A wide variety of all industrial constraint problems can be formulated as *Constraint Satisfaction Problem* (CSP) [24], [13]. Formally, a CSP is defined by:

- a set of n problem *variables* $\{X_1, X_2, \ldots, X_n\}$. These variables are *the unknowns* of the mathematical problem.
- a set of n *domains* $\{D_1, D_2, \ldots, D_n\}$. A domain is a finite set of values (i.e. constants). Classically a domain is a subset of the integers. The domain D_i specifies the set of possible values for the variable X_i.
- a set of m *constraints* $\{C_1, C_2, \ldots, C_m\}$. A constraint is a relationship that must hold between the variables involved in the constraint. Generally a constraint is given intentionally as a formula (e.g. $X = Y + 10$) but it can also be given extensionally as a set of (acceptable) tuples. For instance the constraint $X\ xor\ Y = Z$ can also be given by the set {<0,0,0>, <0,1,1>, <1,0,1>, <1,1,0>}.

A CSP is then a *discrete* mathematical problem. Solving a CSP consists in finding an assignment of variables (according

to their corresponding domains) satisfying all the constraints. Such a problem can also be stated as a first order logic formula:

$$\exists_{X_1,X_2,\ldots,X_n} \bigwedge_{1\leq i\leq n} X_i \in D_i \wedge \bigwedge_{1\leq j\leq m} C_j$$

A constraint program mainly states the constraints (incrementally) and asks the constraint solver to find one/the best (according to an objective function)/all solution(s). Solvers for CSP are called Finite Domain (FD) solvers (the name comes from the fact that a domain is a *finite* set of constants). Such solvers rely on efficient consistency techniques and on local propagation algorithms (e.g. arc-consistency) to reduce the domain of involved variables (i.e. removing values which will never be part of a solution). For efficiency reasons, not all reductions are performed (which would be too costly). The domain of the variables are then an approximation of the real domain of possible values (all solutions are in the domains but not all values are part of a solution). Indeed, there is a trade-off between the precision of the pruning algorithm and its cost in terms of execution time. When a solution is required, in order to eliminate impossible values, the solver iteratively "enumerates" the variables: each variable is assigned with a value in its (reduced) domain and the consequences are propagated to other involved variables (using the same consistency techniques). In case of failure another value is tried (here a chronological backtracking algorithm is generally used but other variations are possible). This *labeling* phase can be improved by using heuristics concerning the order in which variables are considered and the order in which values are considered in the variable domains.

B. The GNU Prolog constraint solver

GNU Prolog is a native Prolog compiler including a powerful constraint solver over Finite Domains [8]. Such a solver offers a wide variety of constraints, which we think have not yet been exploited to their full potential in Information Systems. For instance GNU Prolog offers:

- **arithmetic constraints** (both linear and non-linear), e.g. $X + Y < Z$ or $X * Y \neq Z$. Inside such constraints it is possible to use constraints functions like min, max, $dist$, etc.
- **symbolic constraints** e.g., *atmost(2,[X,Y,Z,T],10)* states that at most 2 variables among X, Y, Z, T can take the value 10. As another example the symbolic constraint *element(I,[*V_1*,*V_2*,...,*V_N*],X)* enforces the variable X to be equal to the Ith element of the vector of N values $V_1, \ldots, V_N$.
- **boolean constraints**: GNU Prolog offers all boolean constraints such as $\wedge, \vee, xor, \neg, \Rightarrow, \Leftrightarrow, \ldots$ Variables appearing in such constraints are implicitly constrained to the domain $0..1$.
- **reified constraints**: this important feature allows the user to reason on the issue (satisfied /unsatisfied) of a constraint. Namely, a constraint C can appear inside any (above) boolean constraint (constraints are in fact first-class objects). As an example consider the boolean constraint $X < Y \Rightarrow K = 8$. Its operational behavior is : as soon as the solver detects that $X < Y$ it enforces $K = 8$, conversely if it discovers $K \neq 8$ it enforces $X \geq Y$.

The core constructs of the constraint language are Constraints and Operators that are applied to Variables and Values. Figure 5 presents our metamodel [18] using UML notation (UML was chosen for the sake of clarity; an example of formal grammar of one popular CP notation can be found in [7]). As the metamodel shows it, a variable has a domain, and at a given moment in time, a value. The domain of variables can be boolean, integer, interval, enumeration or string. This metamodel distinguishes between constraints and operators. Thus, a constraint can be symbolic, arithmetic or boolean and contain zero or several operators. Operators are: multiplication $(*)$, addition $(+)$, subtraction $(-)$, ... These operations can take place in boolean constraints (e.g. $A * B \Leftrightarrow C$). There are three types of constraints: boolean, arithmetic and symbolic. Symbolic constraints are applied on a set of variables at a time. Constraints may be simple, but also reified. A reified constraint is a constraint whose truth value can be captured with a boolean variable, which can itself be part of another constraint. Reified constraints make it for instance possible to reason on the realization of constraints at different times.

Reified constraints can also be used to model soft-constraints (or preferences). A soft-constraint is a constraint we want to impose whenever it is possible but which can be forgotten if the problem is unsatisfiable. Based on this, GNU Prolog offers the constraint *cardinality([*C_1*,* C_2*, ...,* C_N*], Count)* which ensures *Count* is the number of constraints C_i that are true. *Count* being an FD variable, it can be subject to any constraint or to a maximization predicate, e.g. in order to maximize the number of true (soft-)constraints as we will show later on.

C. Incrementality

Constraints appearing at run-time can obviously be taken into account by a naive approach: a CP program is regenerated from scratch augmented with the new constraints. However, let us recall that CP is naturally incremental and constraints are additive since the solving process is order-independent. It is thus possible to easily add a new constraint at run-time (it is also possible to retract constraints dynamically in FD as shown in [21]). Two cases have to be considered. First, the labeling phase has not yet been invoked: in that case the constraint is simply posted to the solver (resulting in further domain reductions). Second, the labeling phase has already been called (e.g. to find a possible solution): in that case the bindings done by this phase have to be undone before adding the new constraint. In this second case, it is worth noticing that this "restoring" is very easy in GNU Prolog with the underlying backtracking mechanism (a choice-point has to be created before the labeling phase). In both cases, the domain reductions performed prior to the labeling phase are kept and do not need to be recomputed (as it would be done when restarting from scratch).

This is a key feature that opens CP to applications interacting with the user (e.g., interactive configuration of a

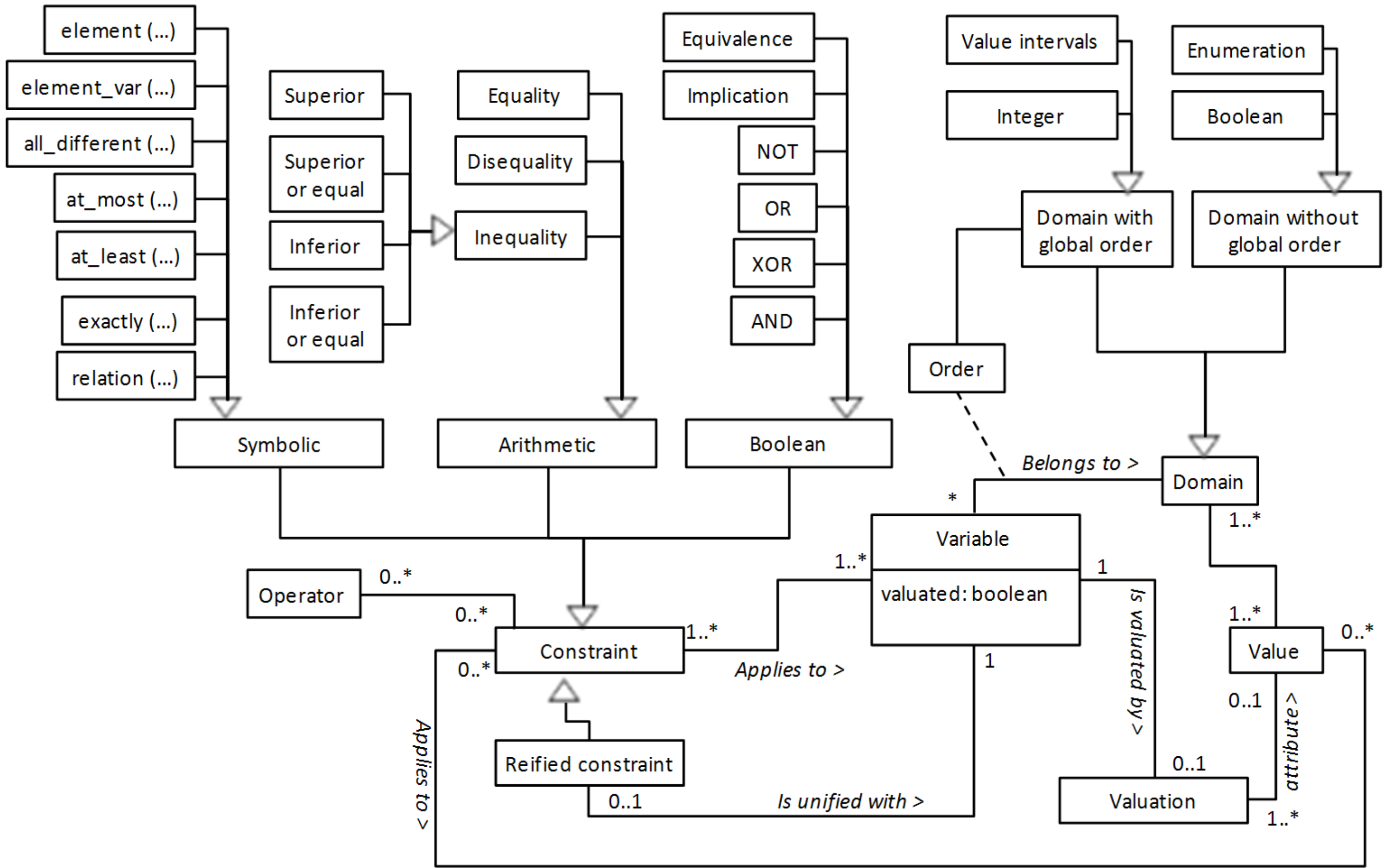

Fig. 5. Meta-model of the GNU Prolog constraint language [18]

Product Line, interactive exploration/modification in virtual environments, etc.). In our case, when deriving a timetable it is possible to interactively add new constraints, as we will discuss further in later sections.

IV. The Timetable Application

In this section we describe the application concerning timetables. It is a sub-part of the whole application of the University's IS. This web-based application allows the administrator to display, compute a timetable and to modify an existing schedule in order to tune it.

A. Computing a Timetable

As explained in Section II-A the Information System contains the strategic information to compute precise timetables. This include mandatory requirements but also preferences. The process to compute a timetable is as follows:

1) Retrieve the needed information from the database.
2) Generate the Constraint Program to solve the CSP associated to the required timetable.
3) Run the program to solve the CSP and propose one solution to the user (here it is possible to provide several solutions and let the user chose the most appropriate).
4) Record the selected timetable in the database.

It is worth noticing that all these steps are written in GNU Prolog/ISCO. Steps 1) and 4) do not need much more explanation (see Section II-B for more information on the OO and persistence aspects). Step 3) is very important since it computes the CSP associated to the required timetable and generates a GNU Prolog program whose execution (Step 4) will provide a solution (i.e. a timetable). It is worth explaining this phase (see citeediss936,rudova for more information about constraints for school timetabling).

The initial step consists in assigning lecturers to courses and classes. This is done in a first step (and does not change so often). Then it is necessary to allocate a time slot to each course session and to find a compatible classroom (i.e. one satisfying the requirements of the course).

The "link" between the data of the planning and the constraint program to generate the timetable is handled by the `TimeSlot` concept. For a given course session there are $duration \times 2$ instances of `TimeSlot`. For instance a two hours class will have a timetable that will have 4 associated slots (the constraints ensuring they must be consecutive is generated in the constraint program). The attribute `varId` is a unique integer which is used to generate a unique variable name X_{varId} in the CSP. The attribute `Slot` contains either

0 (meaning this variable has not yet been assigned) or the assigned time (a value in 1..192). Initially all `Slot` are equal to 0 (meaning nothing is allocated) but later we can assign some variables. This way we can deal with partial timetables. For unallocated slots, the possible domain is given by `Domain` as a set of possible `values` (each value is in 1..192). When a triple <`varId`, `Slot`, `values`> is retrieved if the `Slot` = 0 the the corresponding variable X_{varId} is directly set to the value `Slot` else its domain D_{varId} is defined as `values`. The constraints C_k are generated from the database. Here we benefit from the full expressive power of the GNU Prolog solver. Namely, there are different ways to encode a given constraint. We now discuss the most important constraints and, for the most relevant of them, their translations (it would be too lengthy and out of scope for this paper to exhaustively specify all generated constraints).

Hard constraints are constraints which *must* be satisfied by any solution.

- The k times $X_1, .., X_k$ in a timetable, have to be consecutive, this is done using arithmetic constraints (linear equations): $X_2 = X_1 + 1 \ldots X_k = X_{k-1} + 1$
- Degree year course part of students class. All the times that are associated with a timetable that is associated with a class that is associated with a year of a degree are collected and the *all_different* global constraint is used to ensure that no two classes of the same year of a course occurs at the same time.
- Lecturer. All the times that are associated to a class that a given lecturer teaches are collected in order to impose that they be all different (using again *all_different*).
- If a course has both lectures and lab or tutorial classes, they should not occur on the same day. This may be done by imposing that one of the times in the timetable of each class can not be on the same day as the other, e.g. $X < Y - 32 \vee X > Y + 32$.
- Two consecutive classes of the same degree (and level) or taught by the same lecturer cannot occur in different buildings. This is a hard constraint because students need to have time to go from one building to another one.

Soft constraints or *preferences* are constraints that can be violated if needed [4]. A soft constraint is a constraint we want to impose whenever it is possible, but which can be discarded if the problem becomes unsatisfiable (we say it is *over-constrained*).

- For instance, all courses occurring in the same year of a given degree should have all the classes in the morning. This is achieved by collecting all the times of the classes of the given year and degree (let us say $X_1, \ldots, X_k$, and then imposing a cardinality constraint onto the times, i.e.
  cardinality([
  $1 < V_1 \wedge V_1 \leq 10 \vee \ldots \vee 161 < V_1 \wedge V_1 \leq 170,$
  $\ldots,$
  $1 < V_k \wedge V_k \leq 10 \vee \ldots \vee 161 < V_k \wedge V_k \leq 170$*], M)*
  with M constrained as follows $0 < M$ and $M < 66$.
  Then, we can maximize M the number of time slots in the morning using the predicate *maximize*.
- A teacher does not have more than four hours a day.
  This constraint can be set up in the data base for each lecturer. The university policy, however, may impose a limit, in this case a lecturer can not have more than eight hours a day but this should be a hard constraint.
  These constraints are defined in a similar way to the previous one: using the cardinality and maximize GNU Prolog predicates.
- A lecturer will not have classes in more then three distinct days.

The assignment of a room to a class is done by using the cardinality constraint over a disjunction of conjunctions where only one must be true.

Example: a timetable H, with times associated (H_0, V_0), (H_1, V_1) that has in the room preferences $Room_1$ with slot (S_{11}, VS_{11}), (S_{12}, VS_{12}) free, and $Room_2$ with slot (S_{21}, V_{21}), (S_{22}, V_{22}) free will give rise to the following constraint:
$VS_{11} = V_0 \wedge VS_{12} = V_1 \wedge V_k = Room_1$
cardinality([
$VS_{11} = V_0 \wedge VS_{12} = V_1 \wedge V_r = Room_1, V_rS = S_{11}$
$VS_{21} = V_0 \wedge VS_{22} = V_1 \wedge V_r = Room_2, VrS = S_{21}$*], 1)*

The other constraints that we have to add are that all the slot values of a room that have a value different from 0 must be different, this is done by collecting all the room slot values from the database.

V_r and V_{rS} are variables that are used to keep track of the room and slots associated to a timetable H. This way after the variable labeling we are able to know what is the room of a timetable, and what is the room slot we have associated to each timetable slot.

We do not need to consider all the free slots in a room to place a class because we keep track of the distribution in the Prolog program, otherwise we would have to build a term for each of 192 slots, that becomes too large in the case where we have more than 100 possible rooms for a class.

It is worth noticing that even if theoretically a CSP is solved at once (assigning a value to each variable), in practice this is not possible for our University. Indeed, it is important to keep in mind the structure of the University when computing a timetable. The University is composed of several schools (physically located in different building). Each school has its own chief planner responsible for the sound schedule of the degrees/courses of the school. It is then very important to be able to compute the timetable step by step. The ability of Constraint Programming to solve constraints incrementally allows us to compute timetables at various level of granularity, e.g., degree by degree or by sets of degrees at a time.

Finally it is important to empower the user to "tune" the proposed timetable or to explore different possibilities to chose the most adequate (for instance one can prefer a timetable where tutorials are after lectures). Here again Constraint Programming offers the needed tool to reach this goal (incrementality, non-determinism and backtracking).

B. Application Structure

This application has the following top-level functions, which are all implemented a GNU Prolog / ISCO programs:

- Editing the Teaching Duties Assignment.
- Editing the Rooms information, this can be done by including new rooms, editing the rooms type, or other room specific information such as the number of seats. Other options include the definition of new buildings. Some courses such as sports use specialized facilities such as swimming pools that may have to be added or removed due to varying protocols with external entities, since the university does not own one.
- Editing the information on lecturers, this includes inspecting the classes they have to teach and their schedule-related constraints, which may be hard or soft.
- Check for data consistency, which entails:
  - For each valid degree/year/semester combination: check if they have all the classes they should have according to the degree curriculum.
  - Check if it is possible to build a timetable for all the classes; Sometimes, a class may have too many courses (due to some errors) and this possibility is taken into account here.
  - For all lecturers, check if the classes they have assigned are compatible with their declared hard constraints.
  - For each classroom, we collect all the related session courses to know if it possible to accommodate them. Some errors may occur when rooms are directly assigned to session courses.

  The data consistency step is required to generate a set of timetables for a triple (degree, year, semester).
- Generate Timetables, which is done by choosing a set of degrees, year and semester. This set may include all of them or just one. There is no problem in generating twice the same timetable since the persistent storage keeps the data across runs and that there is no overwriting once instantiated.
  This functionality includes an option to reset (delete) a set of timetables, and is able to locate the (degree, year, semester) triples that do not yet have a complete timetable.
- Visualize timetables: this functionality displays the timetable for a classroom, a teacher, or a degree year course.

Teaching Duties Assignment: The teaching duties assignment includes all the information on courses such as number of weekly hours, number of times by week, who teaches it, courses, some classes may be simultaneously taught to different courses, degrees, year and semester of the degree. This information mainly originates in the departments and should agree with the degree curricula. The scheduling committee may have to correct some of this information, this can be achieved through the web interface presented in figure 7.

This interface is also used to edit each class, listing them by the name of the course (*disciplina*), the degree (*curso*) or the lecturer (*docente*.)

An important feature is that the user can forcibly assign a particular time and classroom to a given scheduled class, even when the slot (time and classroom) was already taken.

Fig. 6. The web interface

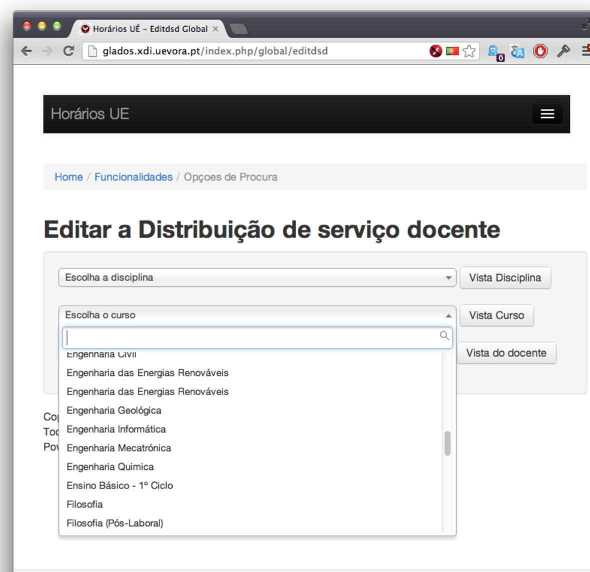

Fig. 7. Assigning teaching duties to lecturers

This way it is possible to manually fine-tune the timetable, correcting little details. The user-imposed edits only become valid when they are found to be consistent.

Per-Course Class Scheduler: Once the schedules have been generated, these may be visualized in their entirety, with an interactive weekly calendar as shown in figure 8.

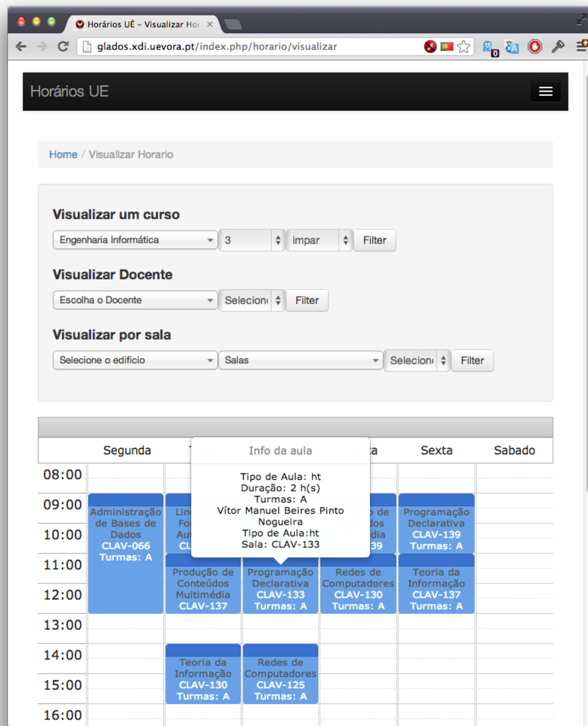

Fig. 8. Visualizing entire schedules

V. Conclusions and Future Work

In this paper we have presented a real-life case study (the University timetabling web-based application) to demonstrate that constraints are useful in designing and implementing Information Systems, and that the latter will benefit significantly from being *constraint-informed*. We have shown that Constraint Programming is almost the perfect tool for this since it combines the declarativity of high-level languages and the power and efficiency of highly optimized constraint solvers.

The timetabling application is presently managing 63 curricula, 2582 courses taught by 515 faculty in 235 classrooms distributed over 10 different buildings. The information system and constraint solver run adequately on a low-end computer (3.4GHz P4, 2GB RAM), meaning that response time is barely perceptible to solve a single timetabling problem. As the system is still under development, we don't yet have performance scaling information, but the design already allows any number of timetables to be scheduled simultaneously, with incrementality as a concern.

Using constraint programming in management information systems has already been experimented with, see for example Gupta and Akhter [12]. When compared with systems which rely on libraries such as IBM CPlex or Gecode [23], our approach has the advantage that we can conveniently do meta-reasoning, as the CSP being solved is explicitly constructed by a Logic program. So far we have developed a flexible timetabling information system, in which the characteristics of a solution are declaratively specified by constraints. This is clearly an advantage over other approaches, not least because the schedule and the criteria themselves may be the object of introspection by the application, i.e. we have an executable specification which is both convenient and efficient.

Our plans at present include enriching the type of constraints that may be specified to indicate preferences, not only w.r.t. lecturers but also for entire weekly schedules. We may also include input regarding preferences such as the total distance traveled to attend a set of classes, or other criteria we may choose to optimize for. Having an incremental schedule builder is also a useful extension to perform, because the instant feedback may prove a time-saver when setting up timetables.

Considering the encouraging results we got from this experience, we will certainly apply a constraint-informed methodology to other Information System design and implementation situations. It is in our medium-term plans to work on the development of a formalism for general-purpose *Constraint-Informed Information Systems*. A promising approach could be to extend the OCL modeling language with generative constraints to handle complex computations (which is the essence of Constraint Programming).

References

[1] Salvador Abreu and Daniel Diaz. Objective: in Minimum Context. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, 2003. ISBN 3-540-20642-6.

[2] Salvador Abreu and Vitor Nogueira. Using a Logic Programming Language with Persistence and Contexts. In Osamu Takata, Masanobu Umeda, Isao Nagasawa, Naoyuki Tamura, Armin Wolf, and Gunnar Schrader, editors, *Declarative Programming for Knowledge Management, 16th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2005, Fukuoka, Japan, October 22-24, 2005. Revised Selected Papers.*, volume 4369 of *Lecture Notes in Computer Science*, pages 38–47. Springer, 2006.

[3] R. Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of the Week of Doctoral Students (WDS99)*, pages 555–564, 1999.

[4] Stefano Bistarelli, Ugo Montanari, Francesca Rossi, Thomas Schiex, Gérard Verfaillie, and Hélène Fargier. Semiring-based csps and valued csps: Frameworks, properties, and comparison. *Constraints*, 4(3):199–240, 1999.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language–Users Guide*. Addison-Wesley Reading, MA, 1999.

[6] D. Chemla, D. Diaz, P. Kerlirzin, and S. Manchon. Using clp(fd) to support air traffic flow management. In *Proc.of the Third International Conference on the Practical Application of Prolog*, pages 69–83, Paris, 1995.

[7] Frank S. de Boer and Catuscia Palamidessi. A fully abstract model for concurrent constraint programming. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT, Vol.1*, volume 493 of *Lecture Notes in Computer Science*, pages 296–319. Springer, 1991.

[8] Daniel Diaz, Salvador Abreu, and Philippe Codognet. On the implementation of GNU Prolog. *Theory and Practice of Logic Programming*, 12(1-2):253–282, 2012.

[9] Olfa Djebbi, Camille Salinesi, and Daniel Diaz. Deriving product line requirements: the red-pl guidance approach. In *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*, pages 494–501. IEEE, 2007.

[10] Eugene C. Freuder. In pursuit of the holy grail. *Constraints*, 2:57–61, 1997.

[11] Antonio Garrido, Marlene Arangu, and Eva Onaindia. A constraint programming formulation for planning: from plan scheduling to plan generation. *J. of Scheduling*, 12(3):227–256, June 2009.

[12] Gopal Gupta and Shameem F. Akhter. Knowledgesheet: A graphical spreadsheet interface for interactively developing a class of constraint programs. In Enrico Pontelli and Vítor Santos Costa, editors, *PADL*, volume 1753 of *Lecture Notes in Computer Science*, pages 308–323. Springer, 2000.

[13] Pascal Van Hentenryck. *Constraint satisfaction in logic programming*. Logic programming. MIT Press, 1989.

[14] Pascal Van Hentenryck and Vijay A. Saraswat. Strategic directions in constraint programming. *ACM Comput. Surv.*, 28(4):701–726, 1996.

[15] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *POPL*, pages 111–119. ACM Press, 1987.

[16] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[17] James Martin. *Managing the Data Base Environment*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1983.

[18] Raúl Mazo, Camille Salinesi, Daniel Diaz, Olfa Djebbi, and Alberto Lora-Michiels. Constraints: The heart of domain and application engineering in the product lines engineering strategy. *IJISMD*, 3(2):33–68, 2012.

[19] Raúl Mazo, Camille Salinesi, Daniel Diaz, and Alberto Lora-Michiels. Transforming attribute and clone-enabled feature models into constraint programs over finite domains. *6th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Springer Press, Beijing–China*, pages 8–11, 2011.

[20] L. Monteiro and A Porto. Contextual Logic Programming. In *Proceedings of the Sixth International Conference on Logic Prog ramming*, pages 284–299. MIT Press, June 1989.

[21] D Philippe Codognet, Daniel Diaz and Francesca Rossi. Constraint retraction in fd. *Foundations of Software Technology and Theoretical Computer Science*, pages 168–179, 1996.

[22] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[23] Guido Tack, Mikael Lagerkvist, and Christian Schulte. Gecode: An Open Constraint Solving Library. Workshop on Open-Source Software for Integer and Constraint Programming (OSSICP), May 2008. Paris, France.

[24] Edward P. K. Tsang. *Foundations of constraint satisfaction*. Computation in cognitive science. Academic Press, 1993.