Constraint-based approach for a class of university timetables

Vincent Barichard, Corentin Behuet, David Genest, Marc Legeay, David Lesaint

Constraint approach for a class of timetables academics

Vincent Barichard, Corentin Behuet, David Genest, Marc Legeay, David Lesaint

1 Univ Angers, LERIA, F-49000 Angers, France

{firstname.lastname}@univ-angers.fr

Summary

Calculating university timetables is a problem of complex combinatorial optimization both in its modeling and its resolution. We propose an approach by constraints that include the formation of groups, the distribution of rooms and teachers, their allocation and the scheduling of sessions. This approach is based on a dedicated rule-based language (UTP) allowing the modeling of the different entities and constraints of an instance. UTP instances are encoded in XML and a generator converts rules into constraints in a format compatible with the MiniZinc and CHR++ solvers. In this paper, we present the UTP language and these constraint programming models as well as experiments preliminary studies carried out on a specific case study.

Abstract

University course timetabling are complex combinatorial optimization problems to model and solve. We propose a constraint-based approach which encompasses student sectioning, room and teacher distribution planning, session scheduling and resource allocation. Our approach is based on a domain-specific rule-based language UTP to model instance entities and constraints. UTP instances are encoded in XML and a flattener converts rules into constraints using formats supported by MiniZinc and CHR. This article presents the UTP language and the two constraint programming models as well as early expressions carried out we are a real case study.

1 Introduction

The organization of university timetables involves strategic, tactical and operational decisions which relate to the modeling of training courses, the constitution classes and groups of students, allocation of services teaching, provision of rooms and equipment, and, ultimately, the scheduling of sessions and resources [6]. The scope of these problems and the process coordinating their resolution varies across countries and institutions as well as the level of automation and decision support systems implemented. Within French universities, for example, training models are conventionally reviewed every 5 years and students register for training and personalize their

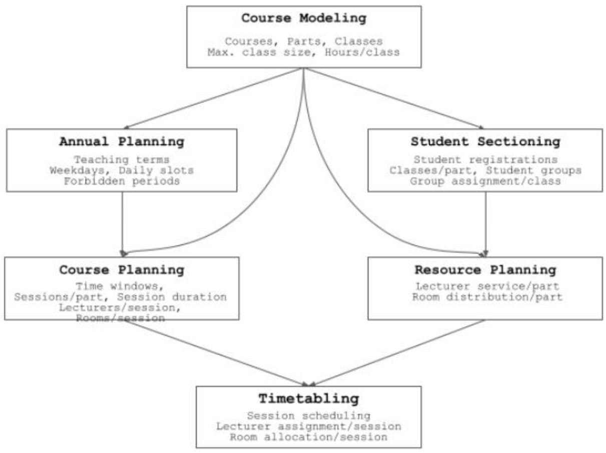


FIGURE 1 – Timetable organization process

course before each teaching period. Classes and groups are then formed according to student profiles and class size thresholds. Teachers and rooms are then positioned on the courses to be given before the class sessions are scheduled and their resources allocated (see Figure 1). This process remains flexible (changes in staff, etc.) and must adapt to the imponderables that punctuate the academic year (late registrations, absences, etc.).

In this article we propose a modeling language for a wide class of timetable problems University Problems (UTP) reducing to the Constraint Satisfaction Problem (CSP). This dedicated language allows to re-present different aspects of the problem relating to the sectioning of courses, the programming of sessions, the distribution of resources and their allocation in order to adapt each instance to the demands of its environment. Language integrates a formal model and a rule language to re-present entities and constraints. The entity model is based on a multi-scale time horizon (i.e. weeks, days and daily slots), a set of resources (ie students, rooms and teachers), and a hierarchical structure of courses (ie courses, parts of courses, classes and Each session must be scheduled individually based on the time frame and resources required.

must be allocated to it.

The scheduling model allows to represent the both single-resource and multi-resource sessions as well than disjunctive, cumulative and hybrid resources. On the one hand, the sessions are labeled as single resource (e.g. lecture) or multi-resource (e.g. hybrid course in distance and face-to-face) by quantifying the number of rooms and teachers required. Students are distributed on the courses according to their registrations then that rooms and teachers are distributed across the parts of courses (e.g. practical work rooms) which determines the area of resources allocable to each session. The service charge is configurable for teachers (ie number of sessions to be provided per part of the course) and the volume of sessions is fixed for students according to their profile (i.e. any part of the course is compulsory) while the rooms can be used at will. Simultaneous use of a resource is not constrained only for rooms which have an inexhaustible capacity, except in the specific case of multi-room screenings which are based on the cumulative capacity of the allocated rooms (e.g. multi-room review). As for the programming of the sessions, each part of the course has its own timetable and each class requires its sessions to be sequenced in a predefined order. Each resource can therefore be allocated to overlapping sessions (e.g. compulsory class and tutoring) with the exception of rooms used jointly by a session. Rules may be overlaid to render resources, or classes of resources, disjunctive. Finally, the model requires partitioning students into groups and breaking down the groups on the different classes while respecting workforce thresholds and any subgroup constraints imposed between classes.

The rule language is based on a catalog of predicates which allows additional constraints to be expressed combining sessions and entities. Each constraint relates to one or more pairs, called e-maps, and optionally parameters depending on the predicate used. An e-map associates an entity with a subset of compatible sessions and is interpreted as a conditional assignment. In other words, a constraint is only evaluated on the sessions for which e-map(s) and solution considered agree, that is, propose the same entity. Each predicate can apply indiscriminately to resources or course elements. E-maps can therefore be shaped to constrain the sessions allocatable to a resource (p. e.g. unavailability of a teacher), the sessions constituting a course element (e.g. frequency of a class), or individual sessions (e.g. parallelization). Note that constraints on course elements are de facto unconditional. Figure 4 shows three constraints: C1 and C2 each covering 2 classes, and C3 relating to a teacher. Constraint C3 relates to 4 sessions but will not apply only to those which will be finally assigned to the teacher.

Rather than imposing individual constraints, the rules are used to formulate conjunctions of constraints targeting entity and session classes (e.g., distributed rules)

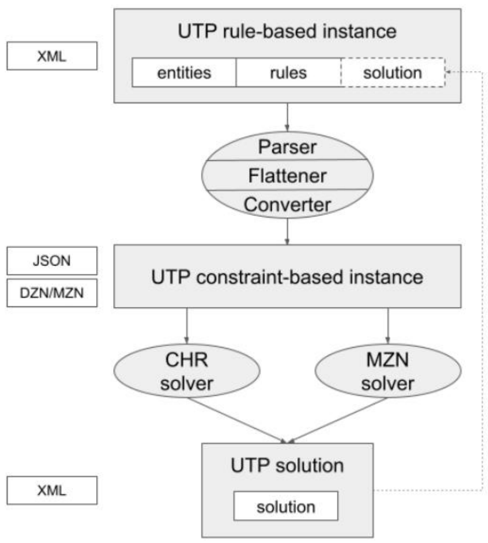


FIGURE 2 – UTP processing chain

junctives on teachers, time restrictions on a Each rule is linked to a predicate and is defined by a quantified constraint whose quantifiers restrict the domain of each e-map variable of the predicate. A selector language is provided to construct and filter e-map domains by session rank, entity ID and type, or any class of tagged elements by the user (e.g., teaching team, room block). A rule therefore denotes the conjunction of constraints resulting from the instantiation of the predicate on the Cartesian product variable domains. Figure 4 describes the constraints generated from 2 rules: constraints C1 and C2 result from rule R1 and constraint C3 from rule R2.

The UTP language is implemented as an XML language. The implementation includes an XML schema validating rule-based instance encoding and a processing chain comprising an XML parser, a generator transforming the rules into constraints, and an encoder converting the resulting instances into a format suitable for solvers (see Figure 2). The integration of a solver involves implementing the model and the predicates of the UTP language and we provide for this purpose two alternative implementations in MiniZinc [18] and CHR [11].

The remainder of the article is organized as follows. We briefly present the UTP language and draw a comparison with the state of the art in section 2. Section 3 details the constraint programming models implemented in MiniZinc and CHR. Section 4 presents the first experimental results. Section 5 concludes and presents the perspectives envisaged for the continuation of this work.

2 The UTP language

The UTP language breaks down the representation of an instance into 3 components: the entity model, the set of rules and the solution. We give a description here informal. A formal specification is presented in

[7], and [1] details the XML syntax and JSON format of UTP instances and also provides access to the source codes of the MiniZinc and CHR++ models, tools, and an instance benchmark.

2.1 Entity Model

The entity model of a UTP instance is schematized in Figure 3. It defines the time horizon, the structure of the course, all resources, as well as properties of entities and associative relationships. The time horizon is broken down into a number of weeks, weekly days and daily slots which are specific to each instance. The decomposition of weeks into days and that of days into daily slots are uniform. The weeks and days succeeding each other on the horizon of times are not assumed to be consecutive while daily slots are. The latter are of equal duration and divide the 24-hour day, e.g., if it is divided into 1440 slots, they will be 1 minute each. The slots serve as a unit of time to date the start and end of sessions and to measure session durations, travel time between rooms and time between sessions.

The courses have a tree structure, each course (p. ex. Algo) broken down into course parts (e.g. TP d'Algo), each part of the course in classes (e.g. Class 2 of Algo TP) and each class in pre-ordered sessions (e.g. 3rd session of Class 2 of Algo TP). The sessions are the elementary tasks to be scheduled when resolving a UTP instance, their number, duration and intra-class sequencing being fixed. Precisely, the classes of a part of the course includes an identical number of sessions of the same duration, these two constants being specific to each part of the course. On the other hand, the language requires that the sessions of a class are sequenced in any solution according to the rank associated with them in the class. Finally, the sessions are non-interruptible and in particular, cannot not to be on two days.

Three types of resources are modeled: rooms, teachers and students (formed into groups). All resources of an instance are declared and typed in the entity model. In practice, different constraints issued in advance apply to resources and course slots (e.g., faculties imposing a timetable by type of course, departments implementing room sharing policies, students registering for courses). The most basic constraints are

compatibility constraints listing suitable rooms, eligible teachers, candidate students and the authorized times for the different courses. Specifically, each part of the course is assigned all the starting slots, rooms and teachers that are authorized for all sessions of the game (see Figure 3). For students, registration is done at the course, a student having to participate in all parts of a courses. The constitution of student groups is carried out at the resolution of the problem or can be provided in the solution component.

The use of resources is also subject to demand and capacity constraints. Since modalities differ from one environment to another, the language supports disjunctive and cumulative resources as well as single- and multi-resource sessions. Students, teachers and rooms are considered cumulative resources if they can attend, teach or host sessions in parallel. Cumulative resources are essential for modeling non-mandatory courses (e.g., optional tutorial sessions that can

overlapping required courses) and to manage multi-class events (e.g. rooms hosting shared exams). The language does not impose any limits on the number of simultaneous sessions attended by teachers and students. Conversely, rooms can only host sessions whose cumulative number of participants is less than their capacity. The capacity of the rooms and the staff thresholds classes are encoded in the entity model which also allows for rooms with unlimited capacity (e.g. rooms Note that any resource is assumed to be cumulative by default but disjunctive rules can be imposed by resource or by resource class.

The sessions are said to be multi-resource if we can allocate multiple resources of the same type. This type of sessions are of practical interest (e.g. multi-room sessions for hybrid teaching, practical work sessions supervised by several teachers, exams requiring several invigilators) and constraints then apply to the volumes of resources required by

session. These are expressed in the model by cardinality constraints declared on parts of course, each part indicating the number of teachers required per session (potentially none) and whether it is single room sessions or not (nrRoomsPerSession and nrTeachersPerSession in Figure 3). Note that a instance can mix single-resource and multi-resource sessions and disjunctive and cumulative resources.

The entity model also incorporates constraints of flow that govern the distribution of students and teachers on courses. These constraints are usually issued upstream of the generation of timetables during the registration and capacity planning phases (e.g. distribution of time volumes between teachers) of a department). As mentioned earlier, the Students only register for courses. Solve a UTP instance therefore involves placing students in the classes in accordance with the course structure and the requested registrations. The rule adopted is that a student be assigned to all sessions of a single class in each part of the course. Nesting constraints of groups can be asked between classes (e.g. aggregate practical work groups to form a group of lectures, preserve the same groups between different courses of a curriculum). For the staff timetable, each teacher has a fixed volume of sessions in the parts of the course where he intervenes, the allocation of sessions remaining to be determined by the solver. In addition predefined entity types, the language offers the possibility to freely label the entities of the model. The entities that

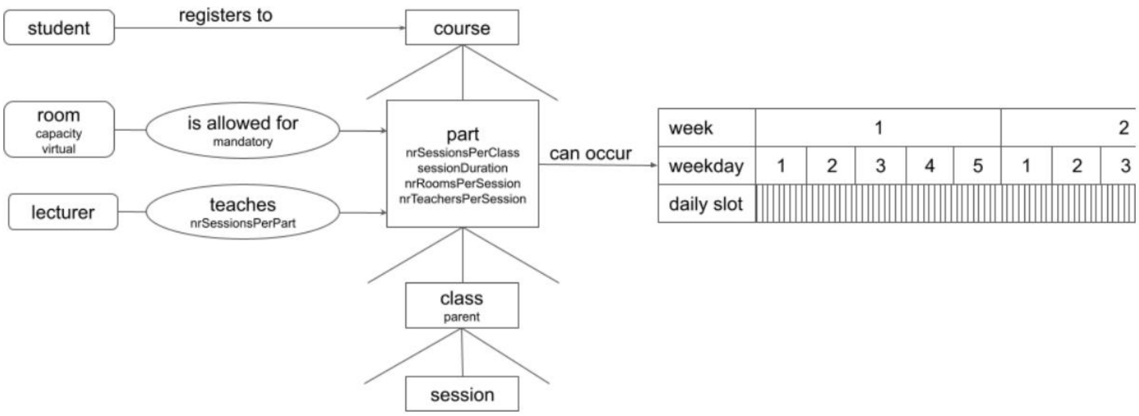


FIGURE 3 – Entity Model

share the same label form a type in their own right. These labels can be used like the predefined types to select entities in rules.

2.2 Ruleset

Rules are used to formulate conjunctions of constraints. It is about being able to express, in a way concise, one or more constraints related to the same predicate. Expressing a rule involves identifying the set of sessions that one wishes to constrain and choosing the predicate to apply. Table 1 lists the UTP predicates currently implemented and used in our instances. A rule applies, depending on the arity of its predicate, to one or more domains of e-maps which each associate an entity to a set of compatible sessions. The e-map domains are not represented in extension but using selectors. A selector allows you to target entities according to their type, label or identifier and to filter the sets of associated sessions according to their ranks and their compatibility with other entities. A rule is then translated by the conjunction of constraints obtained by instantiating the predicate on the Cartesian product of the domains of selected e-maps.

Figure 4 illustrates the session selection on an example toy and automatic generation of constraints from of two rules. The algorithms course is divided into a part of algoLec lectures and part of algoLab practical work. The lecture is given by lecturer1 and contains only one class of 4 sessions. The practical work is supervised by lecturer1 and lecturer2, and are made up of 2 classes of 2 sessions. The first rule (R1) states that practical work of each class can only begin after the third lecture session (entities and sessions marked with a star). It is associated with the predicate sequenced and uses two selectors: the first selects the third session of the algoLec part, the second selects, for each class, the first sessions of the algoLab part. The algoLab part having two classes, the rule produces two constraints related to sequenced: the first (C1) with the sessions algoLec1:3 and algoLab1:1, the second (C2) with the algoLec1:3 and algoLab2:1 sessions.

The second rule (R2) states that lecturer2 is unavailable over a given period (diamonds). It is associated with the forbidden_period predicate and uses a selector that targets the teacher's sessions lecturer2. The rule produces a single constraint (C3) linked to forbidden_period (with the parameters specifying the period of absence of the teacher, here the period between slots 9120 and 9240) relating to all the sessions of the algoLab part where lecturer2 can intervene. The constraint will only be effective on two of these sessions given that lecturer2 supervises two practical work sessions; these sessions will be identified during the resolution.

2.3 Solution

The solution element includes choices of slots and resources for sessions, groups for students, and classes for groups. The solution thus represented can be partial, or even empty, and is not necessarily consistent with the constraints of the instance. The support of partial solutions makes it possible to target and resolve subproblems. For example, an instance is reduced to a scheduling problem if it is based on a solution complete for group formation and assignment resources. Similarly, support for inconsistent solutions is a prerequisite for repairing solutions that would have become inconsistent following changes unanticipated (e.g. absence of a teacher, unavailability of a room due to work).

Student groups are considered to be the result of the sectioning problem. For this reason, the groups are part of the solution element, and define both the group of students who make them up and the classes to which they belong. This sectioning process is subject to various constraints. On the one hand, Groups can only be made up of students who are enrolled in the same courses. Then, each group is unbreakable except in the case of multi-room sessions. Finally, The assignment of groups to classes must meet the inclusion constraints between classes defined in the entity model.

Name	Semantic	Parametric Arity	
same_daily_slot	1	No	Sessions start at the same daily slot
same_weekday	1	No	Sessions start on the same day of the week
same_weekly_slot	1	No	Sessions start on the same time slot and day
same_week	1	No	Sessions start the same week
same_day	1	No	Sessions start on the same day
same_slot	1	No	Sessions start at the same time
forbidden_period	1	Yes	Sessions cannot start within the given period
at_most_daily	1	Yes	The number of sessions in the defined daily period is limited
at_most_weekly	1	Yes	The number of sessions in the defined weekly period is limited
sequenced	ÿ 2 1	No	The sessions are sequenced
weekly		No	Sessions start on the same slots and days of successive weeks
no_overlap	1	No	Sessions cannot be in parallel
travel	1	Yes	Definition of travel time between rooms
same_rooms	1	No	The sessions take place in the same rooms
same_students	1	No	The same students attend the sessions
same_teachers	1	No	The sessions are supervised by the same teachers
adjacent_rooms	1	Yes	Sessions must be in adjacent rooms
teacher_distribution ÿ 2		Yes	Distributes the teaching load in the classes

TABLE 1 – Catalog of UTP predicates

2.4 State of the art

Here we draw a comparison of the UTP language and the ITC representation framework implemented in XML [17, 15].

The two approaches are distinguished first by the modeling of the programs (schedulings) possible by class. The UTP language defines each class by a simple sequence of sessions of equal duration and the problem consists to schedule each session. The ITC scheme proceeds in extension and associates different programs with each class (times element of the schema). The problem then comes down to choosing a program per class where each The program is fixed and is defined by the repetition over several weeks of a weekly schedule comprising one or more sessions of equal duration, placed on different days and sharing the same daily slot. The two performances are not reduced to one another to another. For example, UTP cannot model a class whose sessions are of variable duration. Conversely, ITC cannot model a class programmed on different daily slots. However, some programs of practical interest represent themselves in one or the other approach by constraining classes and sessions in an appropriate manner. For example, a weekly class in front of meeting in the same slot is modeled by combining same_daily_slot, weekly and constraints forbidden_period. The implementation of a method more comprehensive reduction is under study.

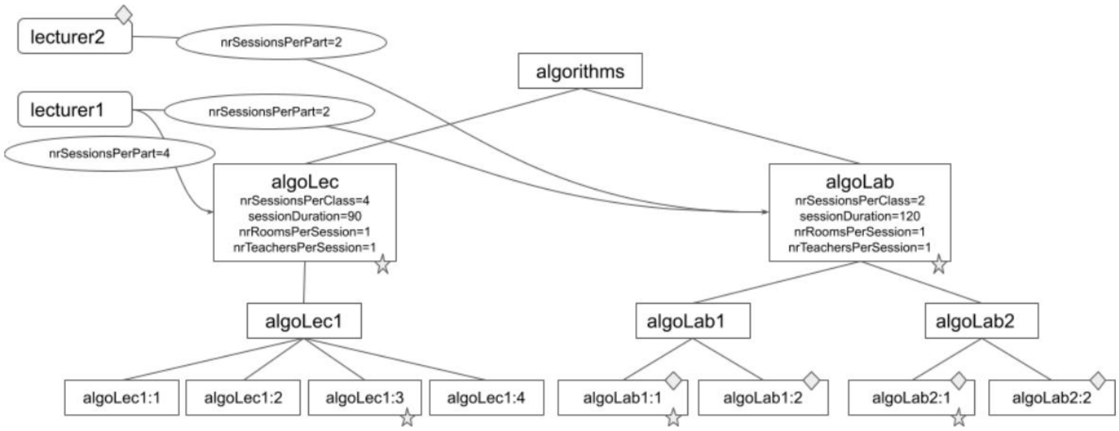
As regards the hierarchical organisation of courses, ITC introduces an intermediate level modeling a choice configuration per course (configuration element). Each course has one or more configurations that are independent as to their decomposition into parts, classes and sessions. The ITC scheme simply imposes that a student enrolled in a course attends all parts of a single configuration, two students can be associated with different configurations. This concept is not integrated into the current version of the UTP language. For what Concerning resources, the UTP language explicitly represents teachers as well as rooms while ITC only models rooms. It also allows you to allocate different resources to sessions in the same class while that the ITC scheme requires that the same room be allocated to them. In addition, UTP authorises multi-resource sessions while ITC is restricted to single-room screenings.

The two constraint languages also stand out from each other. on the other hand. On the one hand, ITC predicates apply to classes whereas UTP predicates apply to any sets of sessions - and in particular to individual sessions - which can be conditioned on choice of allocated resources. On the other hand, the language of UTP rules and selectors allow to constrain any which class of resources or course elements in a concise manner and more suited to the expression of needs.

Finally, the ITC scheme prescribes a resolution of the problem by combinatorial optimization by integrating a function cost weighting 4 criteria which respectively penalize the choice of slots and rooms for classes, violations of constraints and overlapping of sessions per student. In its current version, the UTP language treats the problem as a constraint satisfaction problem hard. The integration of soft constraints and the possibility to aggregate penalties or preferences, whether in construction or repair solution contexts, is under study.

3 MiniZinc and CHR models

In this section, we present two UTP instance models developed in MiniZinc and CHR. These models involve constraints relating to the partitioning of students into groups and the allocation of groups. to classes, to the distribution of resources during sessions, to the scheduling of sessions, and to the allocation of their



sequenced(<(K,_,{3}),(P,algoLec,_)>, <(K,_,{1}),(P,algoLab,_)>) (R1)

forbidden_period((<(T,read2,_)>,9120,9240) (R2)

sequenced((algoLec1,{algoLec1:3}), (algoLab1,{algoLab1:1})) (C1)

sequenced((algoLec1,{algoLec1:3}), (algoLab2,{algoLab2:1}) (C2)

forbidden_period((read2,{algoLab1:1,algoLab1:2,algoLab2:1,algoLab2:2}),9120,9240) (C3)

FIGURE 4 – Session selection by rules

resources. We first present the instance data as well as the decision variables that are common to both models. Table 2 lists the integer ranges identifying the different sets of objects manipulated and defines the structures used to represent instance data.

3.1 MiniZinc Model

MiniZinc is a high-level modeling language of constrained optimization problems [18, 3]. MiniZinc models are translated into the language Flatzinc target [4] which allows to interface different types of solvers including programming solvers by constraints on finite domains such as Gecode [2]. MiniZinc integrates many global constraints and the UTP model presented in Table 4 and using the variables decision presented in Table 3 is based on some constraints dedicated to scheduling problems.

Sectioning constraints distribute students in groups and assigns each of these groups to different classes in accordance with the sectioning rules and to the enrollment thresholds. Constraint (1) only authorizes the grouping of students if they are registered for the same courses. (2) requires that any student, assimilated to his group, attends to any part of the course in which he is registered. (3) ensures that the classes of a part of the course have no groups in common. (4) implements the kinship relationship between classes. Finally, (5) checks that the cumulative number of groups assigned to a class does not exceed the authorized threshold.

The distribution of resources is based on domain, cardinality and sum constraints. Constraints (6) and (7) define the rooms and teachers that can be allocated to each session. (8) constrains the number of rooms allocated to a session depending on its part of the course is without rooms, single-room, or multi-room. (9) attributes the expected number of teachers at each session and (10) checks that each teacher provides the volume of sessions required per course part where it is pre-positioned.

The scheduling of sessions and the allocation of resources involves constraints of positioning, sequencing, non-overlapping and capacity. The constraint (11) defines the slots allowed for each session. (12) prohibits a session from spanning 2 days. (13) sequences the sessions of a class according to their ranks. Constraints (14) and (15) model the sessions multi-room sessions and exclusive access to their resources. (14) requires that a resource allocated to a multi-room session be disjunctive the time of its use. (15) ensures that the number of students expected does not exceed the cumulative capacity of the allocated rooms. Note that this constraint is purely quantitative and allows any distribution of students in the rooms regardless of the structure of groups. (16) models the rooms to be allocated compulsorily at any session of a course part. (17) models the cumulative capacity constraint that applies by default to any allocated room excluding multi-room sessions.

Table 4 shows variants of some UTP predicates in case the targeted entities are rooms. (18) implements the forbidden_period predicate which takes into account parameters the 2 slots modeling the forbidden period. (19), (20) and (21) directly model the predicates same_weekday, same_rooms and sequenced. (22) implements the no_overlap predicate by relying on the disjunctive global constraint.

3.2 CHR model

Constraint Handling Rules (CHR) [13, 12] is a language with forward chaining inference rule base that replaces the constraints of the problem by simpler ones until the complete resolution. CHR is a specialized language

H set of slots defining the time horizon
C set of courses
P sets of course parts
K sets of classes
S sets of sessions
R set of rooms
T set of teachers G set of student
groups U set of students class_{sessions,parents}:
set of sessions (resp. parent
classes) of a class

part_{classes,lecturers,rooms,sessions}:
set of classes (resp. teachers, rooms, sessions) of a part room_sessions :
set of possible
sessions for a room
session_{part,class} : the part (resp. class) of a session
student_{courses,parts} : the courses (resp. parts) that a student follows
mandatory_rooms : the
mandatory rooms for a part single_room_sessions :
all single-room sessions capacity : maximum capacity of a room or a
class is_multi_rooms : indicates whether a session is multi-room length :
duration of a session part_room_use : mode of use of rooms for a part
(none, single, multiple) rank : rank of a session service :

number of sessions to be provided per teacher per part
teams :
number of teachers required per session of a game virtual:
indicates whether a room has unlimited capacity or not dailyslots: daily
slots allowed for a game weekdays: days allowed for a game weeks:
weeks allowed for a game nr_daily_slots: number of slots
in a day nr_weekly_slots: number of slots in a week

TABLE 2 – Instance data and utility functions

allowing to define declarative constraints in the sense of constraint logic programming [14, 16].
CHR is a language extension that allows to introduce user-defined constraints, i.e. first-order predicates, into a given host language such as Prolog, Lisp, Java, or C/C++. It was later extended to CHRÿ [5] which introduces don't know [10]. This nondeterminism is offered for free when the host language is Prolog and it allows to easily specify problems of the NP complexity class. To model and solve UTP instances with the CHR language, we use the CHR++ solver [8] (for Constraint Handling Rules in C++), which is an efficient integration of CHR into the C++ programming language.

The CHR model is instantiated when reading the instance in JSON format. The entity model is first defined, then the constraints from the rules are declared and finally the domains of the variables are updated if a solution part is provided, before launching the resolution of the instance. The complete model for CHR++ is too long 1 . We give in Table 6 to be detailed here the list of constraints taken into account by the solver.
The decision variables to be instantiated are given in Table 5. They are largely similar to those in the MiniZinc model, only the end-of-session variables are added.

To simplify its implementation, the CHR model is partly non-cumulative and some resources such as teachers cannot be shared. It also considers that the sectioning and allocation of students to groups is done upstream. Thus, calculating a solution comes down to finding a consistent allocation of resources while placing the schedules of all the sessions.

Several constraints can be set when analyzing the instance. This is the case for constraints (1) to (9) in table 6.
Constraints (2), (3) and (4) filter domains by removing rooms, teachers or schedules that are impossible by construction of the instance. Constraint (5) ensures that a session starts and ends on the same day by removing from the domain values that contradict it.

Other constraints are set and managed by CHR rules monitoring changes in the domains of variables. This is the case for constraint (1) which ensures the integrity of the session start and end variables. The same is true for (6) which ensures that the number of teachers requested for a session is valid and (7) which verifies that the number of rooms in a session corresponds to what is requested in the instance.

We give as an example the CHR++ rule that checks the integrity of the session start and end variables. This is triggered as soon as a domain of a variable is updated: session_slot(_, S_Start, S_End, S_Length)

=>> CP::Int::plus(S_Start, (*S_Length)-1, S_End);; We use CHR++ which allows to manipulate values associated with logical variables and to wake up the corresponding rules as soon as a change of the value occurs. This mechanism combined with the forward chaining of CHR allows us to implement an efficient rule wake-up and domain propagation mechanism in the manner of a CSP solver.

Constraints (8) and (9) add new CHR constraints to the model. Indeed, the before and disjunct constraints are CHR constraints ensuring the precedence and non-overlapping of two sessions. They are accompanied by rules verifying the consistency of the disjunctive graph created implicitly by adding all these constraints. Static predicates correspond to those read from the instance. They are processed and constraints (filtering constraints, CHR constraints or variable unification) are added.

The dynamic constraints of (13) to (18) are triggered only under certain conditions. CHR rules

1. The interested reader can download the sources of the model [1]

array[U] of var G: x_group array[K] of var set	group assigned to a student
of G: x_groups array[S] of var set of R: x_rooms	set of groups allocated to a class
array[S] of var set of T: x_lecturers set of	set of rooms allocated to a session
teachers allocated to a session	
array[S] of var H : x_slot starting slot assigned to a session	

TABLE 3 – Decision variables (MiniZinc)

```
forall(u, v in U where u<v) (student_courses[u]!=student_courses[v] -> x_group[u]!=x_group[v]) forall(u in U, p   (1)
in student_parts[u])( exists(k in part_classes[p])(x_group[u] in x_groups[k])) forall(p in P, k1, k2 in   (2)
part_classes[p] where k1<k2)(x_groups[k1] intersect x_groups[k2] = {}) forall(k1 in K, k2 in   (3)
class_parents(k1))(x_groups[k1] subset x_groups[k2]) forall(k in K)   (4)
( maxsize[k]<=sum(g in G)(bool2int(g in x_groups[k]) ÿ sum(u in U)(bool2int(x_group[u] = g))) forall(s in S)(x_rooms[s]   (5)
subset part_rooms[session_part[s]]) forall(s in S)(x_lecturers[s]   (6)
subset part_lecturers[session_part[s]]) forall(s in S, p in P where p =   (7)
session_part[s])(
    (part_room_use[p] = none -> x_rooms[s] = {}) /\ (part_room_use[p] = single -> card(x_rooms[s]) = 1)
  /\ (part_room_use[p] = multiple -> card(x_rooms[s])>=1))   (8)
forall(s in S)(card(x_lecturers[s]) = team[session_part[s]]) forall(p   (9)
in P, l in part_lecturers[p])(sum(s in part_sessions[p]))(bool2int( l in x_lecturers[s]) = service[l, p])) forall(p in P, s in   (10)
part_sessions[p])
    (week(x_slot[s]) in weeks[p] /\ weekday(x_slot[s]) in weekdays[p] /\ dailyslot(x_slot[s]) in dailyslots[p])   (11)
forall(s in S)((x_slot[s] ÿ 1) div nr_daily_slots = (x_slot[s] + length[s] ÿ 1) div nr_daily_slots) forall(k in K,   (12)
s1, s2 in class_sessions[k] where rank (s1)<rank(s2)) (x_slot[s1] + length[s]>=x_slot[s2]) forall(p in P, s1 in   (13)
part_sessions[p], r in part_rooms[p], s2 in room_sessions[r] where is_multi_rooms[p] /\ s1!=s2)
    (disjunctive([x_slot[s1], x_slot[s2]],
    [bool2int(r in x_rooms[s1]) ÿ length[s1], bool2int(r in x_rooms[s2]) ÿ length[s2]])) forall(p in   (14)
P, s in part_sessions[p] where is_multi_rooms[p] )
    (sum(r in part_rooms[p])(bool2int(r in x_rooms[s]) ÿ capacity[r])
    <=sum(g in G)(bool2int(g in x_groups[session_class[s]]) ÿ card(group_students[g])))   (15)
forall(p in P, s in part_sessions[p])(mandatory_rooms[p] subset x_rooms[s]) forall(r   (16)
in R where not(virtual[r]))(let {set of S: RS= room_sessions[r ] intersect single_room_sessions ;} in
    (cumulative([x_slot[s]|s in RS], [bool2int(r in x_rooms[s]) ÿ length[s]|s in RS],
    [sum(g in G)(bool2int(g in x_groups[session_class[s]])) ÿ sum(u in U)(
bool2int(g = x_group[u]))|s in RS], capacity[r]))   (17)
forall  bidden_period((r, Sÿ ), h1, h2) = forall(i in S ÿ)(r in x_rooms[i] -> (x_slot[i] + length[i]<=h1 \/x_slot[i]>h2))   (18)
same_weekday((r, Sÿ )) = forall(i, j in S ÿ where i<j)(
    (r in x_rooms[i] intersect x_rooms[j]) -> (x_slot[i] div nr_weekly_slots = x_slot[j] div nr_weekly_slots))   (19)
same_rooms((r, Sÿ )) = forall(i, j in S ÿ where i<j)((
    r in x_rooms[i] intersect x_rooms[j]) -> x_rooms[i] = x_rooms[j])   (20)
sequenced((r1, S1),(r2, S2)) = forall(i in S1, j in S2)(
    (r1 in x_rooms[i] /\ r2 in x_rooms[j]) -> x_slot[i]+length[i]<=x_slot[j]) ], [length[i] ÿ   (21)
no_overlap((r, Sÿ )) = disjunctive([x_slot[i]|i in S ÿ bool2int(r in x_rooms[i] )|i in S ÿ])   (22)
```

TABLE 4 – Model constraints and predicates

array[S] of var set of R : x_rooms	set of rooms allocated to a session
array[S] of var set of T : x_lecturers set of	teachers allocated to a session
array[S] of int H: x_slot_start	starting slot assigned to a session
array[S] of int H: x_slot_end	end slot assigned to a session

TABLE 5 – Decision variables (CHR)

with guard are used for this purpose. (13) checks that a teacher correctly lists the lessons to which it is registered. (14) ensures that the capacity of the rooms is respected and (15) verifies that the rooms marked as mandatory are indeed found in the solution. The predicate (16) ensures that sessions associated with the same predicate same_weekday are set to the same day of the week.

Constraints (17) and (18) add constraints CHR disjuncts when certain conditions are verified. Thus, (17) poses a disjunct between two sessions when the same teacher participates. (18) adds a constraint disjunct between two sessions if these have place in the same room. These CHR constraints enrich the disjunctive graph representing the sequencing of

Integrity constraint: ÿs ÿ
S: x_slot_end[s] = x_slot_start[s] + length(s) (1)
Static constraints (filtering instance entries): ÿs ÿ S: x_rooms[s]
ÿ part_rooms[session_part(s)] ÿs ÿ S: x_lecturers[s] ÿ (2)
part_lecturers[session_part(s)] ÿp ÿ P , ÿs ÿ part_sessions(p): (3)
week(x_slot_start[s]) ÿ weeks[p] ÿ weekday(x_slot_start[s]) ÿ days[p] ÿ
dailyslot(x_slot_start[s]) ÿ dailyslots[p] (4)
ÿs ÿ S: x_slot_start[s]/nr_daily_slots = x_slot_end[s]/nr_daily_slots ÿs ÿ S: (5)
card(x_lecturers[s]) = team[part_sessions[s]] ÿk ÿ K, ÿs ÿ (6)
class_sessions[k] :
If part_room_use[class_part(k)] = none then card(x_rooms[s]) = 0
If part_room_use[class_part(k)] = single then card(x_rooms[s]) = 1
If part_room_use[class_part(k)] = multiple then card(x_rooms[s]) ÿ 1): before(s, sÿ) (7)
ÿk ÿ K, ÿs, sÿ ÿ class_sessions[k], st rank(s) < rank(s ÿk1, k2 ÿ K, ÿ (8)
st ÿg1 ÿ class_groups[k1], ÿg2 ÿ class_groups[k2], with g1 = g2 : ÿs1 ÿ class_sessions(k1), s2 ÿ
class_sessions(k2) : disjunct(s1, s2) (9)
Static predicates:
forbidden_period((e, Sÿ), h, hÿ) = ÿi ÿ S ÿ : (x_slot_start[i] < h) ÿ (x_slot_start[i] > hÿ) (10)
sequenced((e1, S1),(e2, S2)) = ÿi1 ÿ S1, ÿi2 ÿ S2 : before(i1 , i2) same_rooms((e, (11)
Sÿ)) = ÿs1, s2 ÿ S st s1 < s2 : x_rooms[s1] ÿ ÿ x_rooms[s2] (12)
Dynamic constraints:
ÿp ÿ P, ÿl ÿ part_lecturers[p]: {x | x ÿ part_sessions(p), l ÿ x_lecturers[x]} = service[p, l] ÿs ÿ S, ÿr ÿ (13)
session_rooms(s) : {group_students[g]
| g ÿ x_groups[session_class[s]], r ÿ x_rooms[s]} ÿ room_capacity[r] (14)
ÿs ÿ S: mandatory_rooms[session_part[s]] ÿ x_rooms[s] (15)
Dynamic predicate:
same_weekday((e, Sÿ)) = ÿs1,
s2 ÿ S st s1 < ÿ s2 : x_slot_start[s1]/nr_weekly_slots = x_slot_start[s2]/nr_weekly_slots Introspective constraints: (16)

ÿk1, k2 ÿ K, ÿs1 ÿ class_sessions[k1], ÿs2 ÿ class_sessions[k2], st s1 ÿ= s2 :
x_lecturers[s1] ÿ x_lecturers[s2] ÿ= ÿ ÿ disjunct(s1, s2) ÿk1, k2 ÿ K, (17)
ÿs1 ÿ class_sessions[k1], ÿs2 ÿ class_sessions[k2] st s1 ÿ= s2 :
x_rooms[s1] ÿ x_rooms[s2] ÿ= ÿ ÿ disjunct(s1, s2) (18)

TABLE 6 – Constraints and predicates of the CHR model

all sessions.

It should be noted that the CHR model performs domain filtering but also analyzes the disjunctive graph in order to eliminate non-solutions. The edges of the disjunctive graph are oriented as the

resolution and instantiation of decision variables.

4 Experiments

We conducted experiments on a real instance modeling the second semester of the 3rd year of in-

computer science at the University of Angers. The instance is available in XML, JSON and DZN formats on the site [1].

The instance includes 5 courses common to all students and 2 options, each covering 2 courses, for a total of 7 courses taken by students out of 9. The instance includes 24 course parts and 42 classes. The sessions are to be scheduled over a period of 12 weeks of 5 days each where each day is divided into 1-minute slots.

Sessions must be placed on a schedule that starts at 08:00, ends at 19:50 and is made up of 1h20 slots spaced 10 minutes apart. A session that lasts 1 slot therefore has a duration of 80 slots and has 8 possible starting slots. Some sessions last 2 slots, and

therefore have a duration of 170 slots with 7 possible starting slots. In the case where a session lasts 2 hours (120 slots), the session must start or end to align with the grid, i.e. 13 possible starting slots.

The instance is made up of 67 students pre-divided into 4 groups, 12 teachers and 8 rooms. It integrates 46 rules including a majority of rules coordinating the sessions (parallelization between practical work classes or options, sequencing between lectures, tutorials and practical work, etc.) and some rules restricting possible rooms and teachers according to the courses.

The MiniZinc and CHR++ solvers presented in Section 3 were used to solve this instance with an Intel Core i7-10875H 2.30GHz architecture and solved it in less than 5s (excluding flattening for MiniZinc).

The resolution strategy used in the MiniZinc model consists first in allocating the rooms, then the teachers before placing the sessions on the time horizon. The allocation variables are ordered by the first_fail heuristic and their value domains are explored systematically. The variables for choosing slots per session also use the first_fail heuristic and the value choice heuristic consists in splitting each domain (indomain_split). Gecode is the solver used with MiniZinc in our tests. note that the disjunctive constraints are implemented there as a special case of the global constraint cumulative presented in [9].

The resolution strategy used with CHR++ consists of to instantiate the decision variables starting with the array of variables x_rooms, then x_lecturers and finally x_slot_start (other variables are derived by propagation). In each array, the next variable to be instantiated is chosen according to the order of definition in the array and the tested value is always the smallest possible value in the domain. Between each instantiation, a phase of constraint propagation (domain filtering and disjunctive graph analysis) is iterated until a fixed point. In case of failure, the method returns to its choice previous to try the next alternative. There is no at the moment no specialized heuristics, but the choice of the smallest value of the domain seems relevant. Indeed, To build a schedule, it is natural to start setting the sessions from the beginning of the horizon of time.

5 Conclusion and perspectives

In this article we briefly introduced the language UTP which allows to model the problem of construction of university timetables. The language is generic and allows to adapt to different variants of the UTP problem. For example, the resources are cumulative by default but rules can be overridden for make certain resources disjunctive. Moreover, language relies on a catalog of predicates that can be enriched in order to adapt to the specificities of different environments, without modifying the language itself.

In its current version, the UTP language reduces timetable generation to a problem of satisfying hard constraints and does not take into account any optimization criteria. We aim to develop this aspect, in order, among other things, to be able to express preferences, and define methods for evaluating a solution for to be able to choose the one most suited to the wishes of decision-makers (e.g. avoiding long interruptions of lessons in a day for students or grouping courses on half-days for teachers).

We have also detailed two constraint programming models implemented in MiniZinc and CHR++. These two models were developed as proof of concept and will be improved in particular by implementing resolution strategies facilitating the transition scaled to larger instances.

Thanks

This work is partially funded by the Thélème project awarded to the universities of Angers and Le Mans as part of of PIA3.

References

[1] University Service Planning (https://ua-usp.github.io/timetabling/).

[2] Generic Constraint Development Environment (https://www.gecode.org/), 2022.

[3] Minizinc (https://www.minizinc.org/), 2022.

[4] Specification of Flatzinc. Version 1.6 (https://www.minizinc.org/downloads/doc-1.6/flatzinc-spec.pdf), 2022.

[5] S. Abdennadher and H. Schütz. CHR: A Flexible QueryLanguage. In FAQS 1998, pages 1–14, 1998.

[6] A. Nurul Liyana Abdul and A. Nur Aidya Hanum. A brief review on the features of university course timetabling problem. AIP Conference Proceedings, 2016(1):020001, 2018.

[7] V. Barichard, C. Behuet, D. Genest, M. Legeay, and D. Lesaint. A constraint language for university timetabling problems. Submitted, 2022.

[8] V. Barichard and I. Stéphan. Quantified constraint handling rules. In ICLP 2019, volume 306, pages 210–223, Las Cruces, 09/20-25/2019 2019.

[9] N. Beldiceanu and M. Carlsson. A new multi-resource cumulative constraint with negative heights. In CP 2002, pages 63–79, 2002.

[10] H. Betz and TW Frühwirth. Linear-logic based analysis of constraint handling rules with disjunction. ACM Transactions on Computational Logic, 14(1), 2013.

[11] TW Frühwirth. Constraint Handling Rules. In Constraint Programming: Basics and Trends, pages 90–107, 1994.

[12] TW Frühwirth. Constraint Handling Rules. Cambridge University Press, 2009.

[13] TW Frühwirth. Theory and practice of constraint handling rules. Journal of Logic Programming, 37(1-3):95–138, 1998.

[14] P. Van Hentenryck. Constraint logic programming. Knowledge Engineering Review, 6(3):151–194, 1991.

[15] ITC19. International Timetabling Competition (https://www.itc2019.org/), 2019.

[16] J. Jaffar and MJ Maher. Constraint logic programming: A survey. Journal of Logic Programming, 19/20:503–581, 1994.

[17] T. Müller, H. Rudová, and Z. Müllerová. University course timetabling and International Timetabling Competition 2019. In PATAT-2018, pages 5–31, 2018.

[18] Nethercote N, Stuckey PJ, Becket R, Brand S, GJ Duck, and G.Tack. Minizinc: Towards a standard cp modeling language. In CP 2007, pages 529–543, 2007.