Approche par contraintes pour une classe d'emplois du temps universitaires

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

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

{prenom.nom}@univ-angers.fr

Résumé

Le calcul d'emplois du temps universitaire est un problème d'optimisation combinatoire complexe tant dans sa modélisation que sa résolution. Nous proposons une approche par contraintes qui englobe la constitution de groupes, la distribution des salles et enseignants, leur allocation et l'ordonnancement des séances. Cette approche se fonde sur un langage dédié à base de règles (`UTP`) permettant de modéliser les différentes entités et contraintes d'une instance. Les instances `UTP` sont encodées en `XML` et un générateur convertit les règles en contraintes dans un format compatible avec les solveurs `MiniZinc` et `CHR++`. Nous présentons dans cet article le langage `UTP` et ces modèles de programmation par contraintes ainsi que des expérimentations préliminaires réalisées sur un cas d'études concret.

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 expriments carried out on a real case study.

1 Introduction

L'organisation d'emplois du temps universitaires met en jeu des décisions d'ordre stratégique, tactique et opérationnel qui portent sur le maquettage des formations, la constitution des classes et groupes d'étudiants, l'affectation des services d'enseignement, le provisionnement de salles et d'équipements, et, in fine, la programmation des séances et des ressources [6]. Le périmètre de ces problèmes et le processus coordonnant leur résolution varie selon les pays et les institutions ainsi que le niveau d'automatisation et les sytèmes d'aide à la décision mis en oeuvre. Au sein des universités françaises, par exemple, les maquettes de formation sont par convention revues tous les 5 ans et les étudiants s'inscrivent aux formations et personnalisent leurs parcours avant chaque période d'enseignement. Classes et groupes sont alors constitués selon les profils d'étudiants et les seuils d'effectifs par classe. Enseignants et salles sont ensuite positionnés sur les cours à dispenser avant que les séances de classe ne soient programmées et leurs ressources allouées (voir Figure 1). Ce processus reste flexible (changements de personnels, etc.) et doit s'adapter aux impondérables qui rythment l'année universitaire (inscriptions tardives, absences, etc).

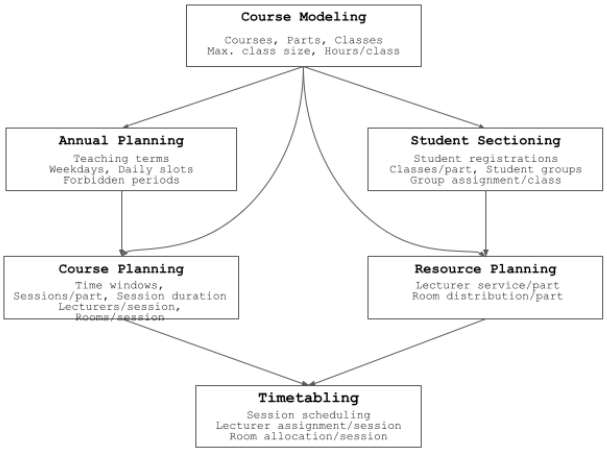


FIGURE 1 – Processus d'organisation d'emplois du temps

Nous proposons dans cet article un langage de modélisation pour une classe étendue de problèmes d'emplois du temps universitaires (`UTP`) se réduisant au problème de satisfaction de contraintes (`CSP`). Ce langage dédié permet de représenter différents aspects du problème relatifs au sectionnement de cours, la programmation de séances, la distribution de ressources et leur allocation afin d'adapter chaque instance aux exigences de son environnement. Le langage intègre un modèle formel et un langage de règles pour représenter entités et contraintes. Le modèle d'entités s'appuie sur un horizon de temps multi-échelles (i.e. semaines, jours et créneaux quotidiens), un ensemble de ressources (i.e. étudiants, salles et enseignants), et une structure hiérarchique de cours (i.e. cours, parties de cours, classes et séances). Chaque séance est à programmer individuellement sur l'horizon de temps et les ressources nécessaires

doivent lui être allouées.
Le modèle d'ordonnancement permet de représenter à la fois des séances mono-ressource et multi-ressources ainsi que des ressources disjonctives, cumulatives et hybrides. D'une part, les séances sont étiquetées à ressource unique (p. ex. cours magistral) ou à ressources multiples (p. ex. cours hybride en distanciel et présentiel) en quantifiant le nombre de salles et d'enseignants requis. Les étudiants se distribuent sur les cours selon leurs inscriptions alors que salles et enseignants se distribuent sur les parties de cours (p. ex. salles de travaux pratiques) ce qui détermine le domaine des ressources allouables à chaque séance. La charge de service est configurable pour les enseignants (i.e. nombre de séances à dispenser par partie de cours) et le volume de séances est figé pour les étudiants selon leur profil (i.e. toute partie de cours est obligatoire) tandis que les salles peuvent être utilisées à volonté.
L'utilisation simultanée d'une ressource n'est contrainte que pour les salles qui ont une capacité d'accueil indépassable, sauf dans le cas particulier des séances multi-salles qui s'appuient sur la capacité cumulée des salles allouées (p. ex. examen sur plusieurs salles). Pour ce qui concerne la programmation des séances, chaque partie de cours possède sa propre grille horaire et chaque classe impose de séquencer ses séances dans un ordre prédéfini. Chaque ressource peut donc être allouée à des séances se chevauchant (p. ex. classe obligatoire et tutorat) à l'exception des salles utilisées conjointement par une séance. Des règles peuvent être surimposées pour rendre des ressources, ou des classes de ressources, disjonctives. Enfin, le modèle impose de partitionner les étudiants en groupes et de ventiler les groupes sur les différentes classes en respectant des seuils d'effectifs et toute contrainte de sous-groupes posée entre classes.
Le langage de règles s'appuie sur un catalogue de prédicats qui permet d'exprimer des contraintes supplémentaires associant séances et entités. Chaque contrainte porte sur une ou plusieurs paires, appelées e-maps, et éventuellement des paramètres selon le prédicat utilisé. Une e-map associe une entité à un sous-ensemble de séances compatibles et s'interprète comme une affectation conditionnelle. Autrement dit, une contrainte ne s'évalue que sur les séances pour lesquelles e-map(s) et solution considérée concordent, c'est-à-dire proposent la même entité. Chaque prédicat peut s'appliquer indistinctement à des ressources ou des éléments de cours. Les e-maps peuvent donc être façonnées pour contraindre les séances allouables à une ressource (p. ex. indisponibilité d'un enseignant), les séances constitutives d'un élément de cours (p. ex. périodicité d'une classe), ou des séances individuelles (p. ex. parallélisation). À noter que les contraintes portant sur des éléments de cours sont de facto inconditionnelles. La Figure 4 présente trois contraintes : C1 et C2 portant chacune sur 2 classes, et C3 portant sur un enseignant. La contrainte C3 porte sur 4 séances mais ne s'appliquera pas qu'à celles qui seront finalement affectées à l'enseignant.
Plutôt que de poser des contraintes individuelles, les règles sont utilisées pour formuler des conjonctions de contraintes ciblant des classes d'entités et de séances (p. ex. règles disjonctives sur enseignants, restrictions temporelles sur un cursus). Chaque règle est liée à un prédicat et se définit par une contrainte quantifiée dont les quantificateurs restreignent le domaine de chaque variable e-map du prédicat. Un langage de sélecteurs est fourni pour construire et filtrer les domaines d'e-maps par rang de séances, identifiant et type d'entités, ou toute classe d'éléments étiquettés par l'utilisateur (p. ex., équipe enseignante, bloc de salles). Une règle dénote donc la conjonction de contraintes résultant de l'instanciation du prédicat sur le produit cartésien des domaines de variables. La Figure 4 décrit les contraintes générées à partir de 2 règles : les contraintes C1 et C2 résultent de la règle R1 et la contrainte C3 de la règle R2.

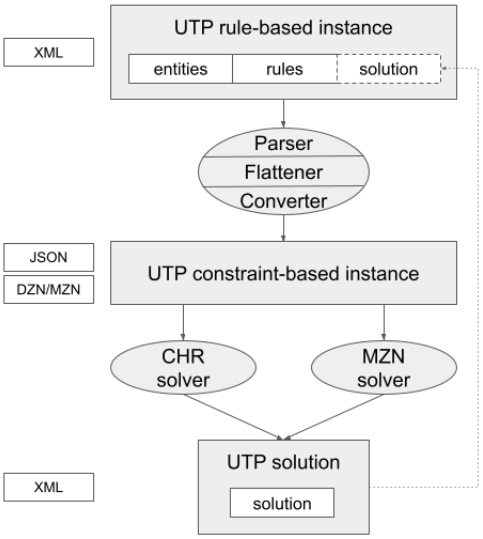


FIGURE 2 – Chaîne de traitements UTP

Le langage UTP est implémenté sous la forme d'un langage XML. L'implémentation comprend un schéma XML validant l'encodage d'instances à base de règles et une chaîne de traitements comportant un parseur XML, un générateur transformant les règles en contraintes, et un encodeur convertissant les instances résultantes dans un format approprié pour les solveurs (voir Figure 2). L'intégration d'un solveur suppose d'implanter le modèle et les prédicats du langage UTP et nous fournissons à cet effet deux implémentations alternatives en MiniZinc [18] et CHR [11].
La suite de l'article s'organise comme suit. Nous présentons brièvement le langage UTP et dressons une comparaison à l'état de l'art en section 2. La section 3 détaille les modèles de programmation par contraintes implémentés en MiniZinc et CHR. La section 4 présente les premiers résultats expérimentaux. La section 5 conclut et présente les perspectives envisagées pour la suite de ce travail.

2 Le langage UTP

Le langage UTP décompose la représentation d'une instance en 3 composantes : le modèle d'entités, l'ensemble de règles et la solution. Nous en donnons ici une description informelle. Une spécification formelle est présentée dans

[7], et [1] détaille la syntaxe XML et le format JSON des instances UTP et donne aussi accès aux codes sources des modèles MiniZinc et CHR++, aux outils, et à un benchmark d'instances.

2.1 Modèle d'entités

Le modèle d'entités d'une instance UTP est schématisé en Figure 3. Il définit l'horizon de temps, la structure des cours, l'ensemble des ressources, ainsi que des propriétés d'entités et des relations associatives. L'horizon de temps se décompose en un nombre de semaines, de journées hebdomadaires et de créneaux quotidiens qui sont propres à chaque instance. La décomposition des semaines en journées et celle des journées en créneaux quotidiens sont uniformes. Les semaines et jours se succédant sur l'horizon de temps ne sont pas supposés consécutifs alors que les créneaux quotidiens le sont. Ces derniers sont de durée égale et divisent la journée de 24h, p. ex., si elle se divise en 1440 créneaux, ils seront d'1 minute chacun. Les créneaux servent d'unité de temps pour dater démarrage et fin de séances et pour mesurer durées de séance, temps de déplacement entre salles et délais entre séances.

Les cours ont une structure arborescente, chaque cours (p. ex. Algo) se décomposant en parties de cours (p. ex. TP d'Algo), chaque partie de cours en classes (p. ex. Classe 2 de TP d'Algo) et chaque classe en séances pré-ordonnées (p. ex. 3ème séance de la Classe 2 de TP d'Algo). Les séances sont les tâches élémentaires à ordonnancer quand on résoud une instance UTP, leur nombre, durée et séquencement intra-classe étant fixés. Précisément, les classes d'une partie de cours comportent un nombre identique de séances de même durée, ces deux constantes étant propres à chaque partie de cours. D'autre part, le langage impose que les séances d'une classe soient séquencées dans toute solution selon le rang qui leur est associé dans la classe. Enfin, les séances sont non-interruptibles et en particulier, ne peuvent pas être à cheval sur deux journées.

Trois types de ressources sont modélisés : les salles, les enseignants et les étudiants (constitués en groupes). Toutes les ressources d'une instance sont déclarées et typées dans le modèle d'entités. Dans la pratique, différentes contraintes émises en amont s'appliquent aux ressources et aux créneaux de cours (p. ex., facultés imposant une grille horaire par type de cursus, départements mettant en œuvre des politiques de mise en commun de salles, étudiants s'inscrivant aux cours). Les contraintes les plus élémentaires sont des contraintes de compatibilité énumèrant les salles appropriées, les enseignants éligibles, les étudiants candidats et les horaires autorisés pour les différents cours. Précisément, à chaque partie de cours est assigné l'ensemble des créneaux de départ, de salles et d'enseignants qui sont autorisés pour toutes les séances de la partie (cf. Figure 3). Concernant les étudiants, l'inscription se fait au niveau des cours, un étudiant devant participer à toutes les parties d'un cours. La constitution des groupes d'étudiants s'effectue à la résolution du problème ou peut être fournie dans la composante solution.

L'utilisation des ressources est également soumise à des contraintes de demande et de capacité. Comme les modalités diffèrent d'un environnement à un autre, le langage prend en charge les ressources disjonctives et cumulatives ainsi que les séances à ressources uniques ou multiples. Étudiants, enseignants et salles sont qualifiés de ressource cumulative s'ils peuvent suivre, enseigner ou héberger des séances en parallèle. Les ressources cumulatives sont indispensables à la modélisation de cours non-obligatoires (p. ex. séances de tutorat facultatives pouvant chevaucher des cours obligatoires) et pour gérer des événements multi-classes (p. ex. salles hébergeant des examens mutualisés). Le langage n'impose aucune limite sur le nombre de séances simultanées auxquelles participent enseignants et étudiants. À l'inverse, les salles ne peuvent héberger que des séances dont l'effectif cumulé est en deça de leur capacité. La capacité des salles et les seuils d'effectif des classes sont encodés dans le modèle d'entités qui autorise par ailleurs des salles à capacité illimitée (p. ex. salles virtuelles). Notons que toute ressource est supposée cumulative par défaut mais des règles disjonctives peuvent être imposées par ressource ou par classe de ressources.

Les séances sont dites multi-ressources si on peut leur allouer plusieurs ressources du même type. Ce type de séances présente un intérêt pratique (p. ex. séances multi-salles pour enseignement hybride, séances de travaux pratiques supervisées par plusieurs enseignants, examens nécessitant plusieurs surveillants) et des contraintes s'appliquent alors aux volumes de ressources requis par séance. Ces dernières s'expriment dans le modèle par des contraintes de cardinalité déclarées sur les parties de cours, chaque partie indiquant le nombre d'enseignants requis par séance (potentiellement aucun) et s'il s'agit de séances à salle unique ou non (nrRoomsPerSession et nrTeachersPerSession en Figure 3). À noter qu'une instance peut mixer des séances mono-ressource et multi-ressources et des ressources disjonctives et cumulatives.

Le modèle d'entités incorpore également des contraintes de flot qui régissent la distribution des étudiants et des enseignants sur les cours. Ces contraintes sont habituellement émises en amont de la génération d'emplois du temps durant les phases d'inscription et de planification de capacité (p. ex. distribution des volumes horaires entre enseignants d'un département). Comme mentionné précédemment, les étudiants s'inscrivent uniquement aux cours. Résoudre une instance UTP implique donc de placer les étudiants dans les classes conformément à la structure des cours et aux inscriptions demandées. La règle adoptée est qu'un étudiant soit assigné à toutes les séances d'une seule classe dans chaque partie de cours. Des contraintes d'imbrication de groupes peuvent être posées entre classes (p. ex. aggréger des groupes de travaux pratiques pour constituer un groupe de cours magistral, préserver les mêmes groupes entre différents cours d'un cursus). Pour l'emploi du temps du personnel, chaque enseignant a un volume fixe de séances dans les parties de cours où il intervient, l'affectation des séances restant à déterminer par le solveur. En complément des types d'entités prédéfinis, le langage offre la possibilité d'étiqueter librement les entités du modèle. Les entités qui

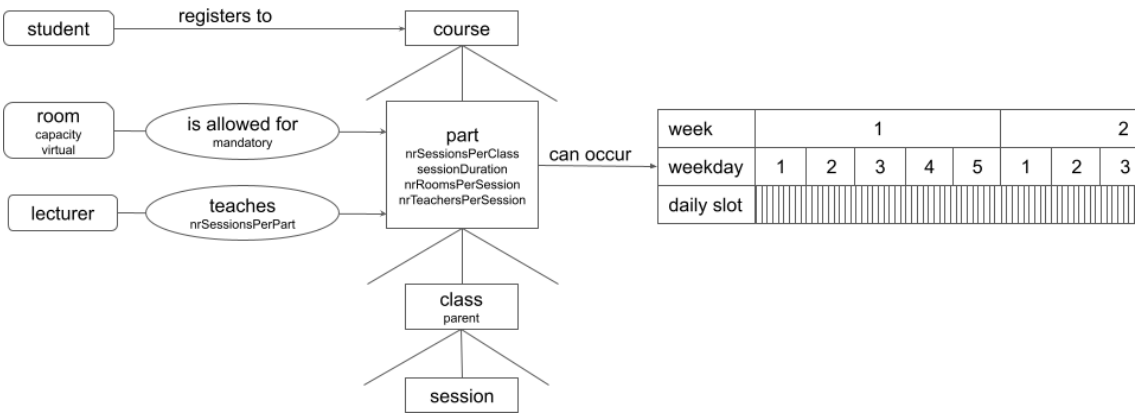


FIGURE 3 – Modèle d'entités

partagent la même étiquette forment un type à part entière. Ces étiquettes peuvent être utilisées à l'instar des types prédéfinis pour sélectionner les entités dans les règles.

2.2 Ensemble de règles

Les règles sont utilisées pour formuler des conjonctions de contraintes. Il s'agit de pouvoir exprimer, de manière concise, une ou plusieurs contraintes liées à un même prédicat. L'expression d'une règle implique d'identifier l'ensemble des séances que l'on souhaite contraindre et de choisir le prédicat à appliquer. La table 1 liste les prédicats `UTP` actuellement implémentés et utilisés dans nos instances.

Une règle porte, selon l'arité de son prédicat, sur un ou plusieurs domaines d'e-maps qui associent chacune une entité à un ensemble de séances compatibles. Les domaines d'e-maps ne sont pas représentés en extension mais à l'aide de sélecteurs. Un sélecteur permet de cibler des entités selon leur type, leur étiquette ou leur identifiant et de filtrer les ensembles de séances associées selon leurs rangs et leur compatibilité avec d'autres entités. Une règle se traduit ensuite par la conjonction de contraintes obtenue en instanciant le prédicat sur le produit cartésien des domaines d'e-maps sélectionnées.

La Figure 4 illustre la sélection de séances sur un exemple jouet et la génération automatique de contraintes à partir de deux règles. Le cours `algorithms` est divisé en une partie de cours magistraux `algoLec` et une partie de travaux pratiques `algoLab`. Le cours magistral est dispensé par `lecturer1` et ne contient qu'une classe de 4 séances. Les travaux pratiques sont encadrés par `lecturer1` et `lecturer2`, et sont constitués de 2 classes de 2 séances.

La première règle (R1) stipule que les travaux pratiques de chaque classe ne peuvent commencer qu'après la troisième séance de cours magistral (entités et séances annotées avec une étoile). Elle est associée au prédicat `sequenced` et utilise deux sélecteurs : le premier sélectionne la troisième séance de la partie `algoLec`, le second sélectionne, pour chaque classe, les premières séances de la partie `algoLab`. La partie `algoLab` ayant deux classes, la règle produit deux contraintes liées à `sequenced` : la première (C1) avec les séances `algoLec1:3` et `algoLab1:1`, la seconde (C2) avec les séances `algoLec1:3` et `algoLab2:1`.

La seconde règle (R2) stipule que `lecturer2` est indisponible sur une période donnée (losanges). Elle est associée au prédicat `forbidden_period` et utilise un sélecteur qui cible les séances de l'enseignant `lecturer2`. La règle produit une seule contrainte (C3) liée à `forbidden_period` (avec les paramètres spécifiant la période d'absence de l'enseignant, ici la période entre les créneaux 9120 et 9240) portant sur l'ensemble des séances de la partie `algoLab` où peut intervenir `lecturer2`. La contrainte ne sera effective que sur deux de ces séances étant donné que `lecturer2` encadre deux séances de travaux pratiques ; ces séances seront identifiées pendant la résolution.

2.3 Solution

L'élément solution comporte des choix de créneaux et de ressources pour les séances, de groupes pour les étudiants, et de classes pour les groupes. La solution ainsi représentée peut être partielle, voire vide, et n'est pas nécessairement consistante avec les contraintes de l'instance. Le support des solutions partielles permet de cibler et résoudre des sous-problèmes. Par exemple, une instance se réduit à un problème d'ordonnancement si elle se base sur une solution complète pour la constitution des groupes et l'affectation des ressources. De même, le support des solutions inconsistantes est un pré-requis pour la réparation de solutions qui seraient devenues inconsistantes suite à des changements non-anticipés (p. ex. absence d'un enseignant, indisponibilité d'une salle suite à des travaux).

Les groupes d'étudiants sont considérés comme le résultat du problème de sectionnement. Pour cette raison, les groupes font partie de l'élément solution, et définissent à la fois l'ensemble des étudiants qui les composent et les classes auxquelles ils appartiennent. Ce processus de sectionnement est soumis à différentes contraintes. D'une part, les groupes ne peuvent être constitués que d'étudiants qui sont inscrits aux mêmes cours. Ensuite, chaque groupe est insécable sauf dans le cas de séances multi-salles. Enfin, l'affectation des groupes aux classes doit satisfaire aux contraintes d'inclusion entre classes définies dans le modèle d'entités.

Nom	Arité	Paramétrique	Sémantique
same_daily_slot	1	non	Les séances démarrent le même créneau quotidien
same_weekday	1	non	Les séances démarrent la même jour de la semaine
same_weekly_slot	1	non	Les séances démarrent les mêmes créneau et journée
same_week	1	non	Les séances démarrent la même semaine
same_day	1	non	Les séances démarrent le même jour
same_slot	1	non	Les séances démarrent en même temps
forbidden_period	1	oui	Les séances ne peuvent pas débuter dans la période donnée
at_most_daily	1	oui	Le nombre de séances dans la période journalière définie est limité
at_most_weekly	1	oui	Le nombre de séances dans la période hebdomadaire définie est limité
sequenced	≥ 2	non	Les séances sont séquencées
weekly	1	non	Les séances démarrent les mêmes créneaux et jours de semaines successives
no_overlap	1	non	Les séances ne peuvent être en parallèle
travel	1	oui	Définition du temps de trajet entre salles
same_rooms	1	non	Les séances ont lieu dans les mêmes salles
same_students	1	non	Les mêmes étudiants suivent les séances
same_teachers	1	non	Les séances sont encadrées par les mêmes enseignants
adjacent_rooms	1	oui	Les séances doivent être dans des salles adjacentes
teacher_distribution	≥ 2	oui	Distribue la charge des enseignants dans les classes

TABLE 1 – Catalogue des prédicats UTP

2.4 Etat de l'art

Nous dressons ici une comparaison du langage UTP et du cadre de représentation ITC implémenté en XML [17, 15].

Les deux approches se distinguent d'abord par la modélisation des programmes (ordonnancements) possibles par classe. Le langage UTP définit chaque classe par une simple séquence de séances de durée égale et le problème consiste à programmer chaque séance. Le schéma ITC procède en extension et associe différents programmes à chaque classe (élément times du schéma). Le problème se ramène alors au choix d'un programme par classe où chaque programme est figé et se définit par la répétition sur plusieurs semaines d'un planning hebdomadaire comportant une ou plusieurs séances de durée égale, placées sur différentes journées et partageant le même créneau quotidien. Les deux représentations ne se réduisent pas l'une à l'autre. Par exemple, UTP ne peut modéliser une classe dont les séances sont de durées variables. A l'inverse, ITC ne peut modéliser une classe programmée sur différents créneaux quotidiens. Toutefois, certains programmes d'intérêt pratique se représentent dans l'une ou l'autre approche en contraignant classes et séances de manière appropriée. Par exemple, une classe hebdomadaire devant se rencontrer sur le même créneau se modélise en combinant des contraintes same_daily_slot, weekly et forbidden_period. L'implémentation d'une méthode de réduction plus complète est en cours d'étude.

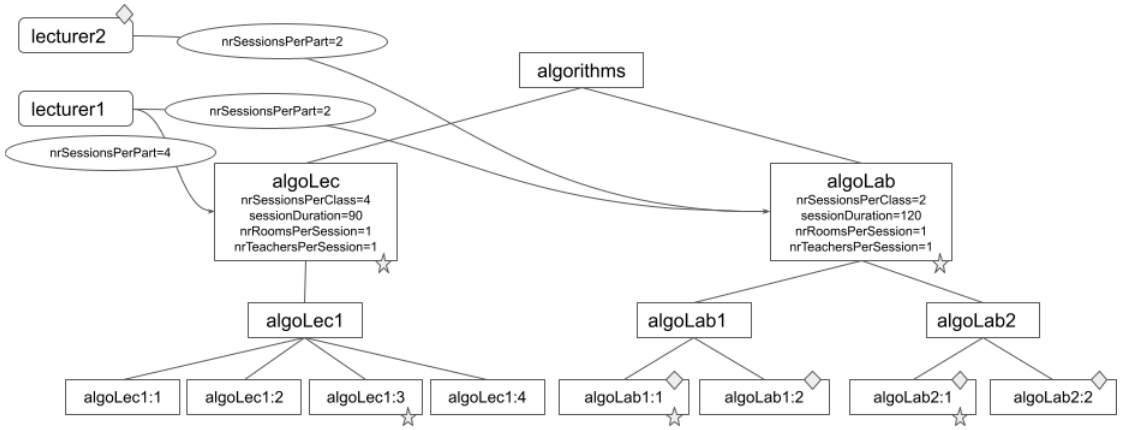
Pour ce qui concerne l'organisation hiérarchique des cours, ITC introduit un niveau intermédiaire modélisant un choix de configuration par cours (élément configuration). Chaque cours possède une ou plusieurs configurations qui sont indépendantes quant à leur décomposition en parties, classes et séances. Le schéma ITC impose simplement qu'un étudiant inscrit à un cours assiste à toutes les parties d'une seule configuration, deux étudiants pouvant être associés à des configurations différentes. Ce concept n'est pas intégré dans la version actuelle du langage UTP. Pour ce qui concerne les ressources, le langage UTP représente explicitement les enseignants à l'instar des salles alors qu'ITC ne modélise que les salles. Il permet aussi d'allouer différentes ressources aux séances d'une même classe alors que le schéma ITC impose qu'une même salle leurs soit allouée. En outre, UTP autorise des séances multi-ressources alors qu'ITC est restreint aux séances mono-salles.

Les deux languages de contraintes se démarquent aussi l'un de l'autre. D'une part, les prédicats ITC s'appliquent aux classes alors que les prédicats UTP s'appliquent à des ensembles de séances quelconques - et en particulier à des séances individuelles - qui peuvent être conditionnés au choix des ressources allouées. D'autre part, le langage de règles et de sélecteurs UTP permet de contraindre n'importe quelle classe de ressources ou d'éléments de cours de manière concise et plus adaptée pour l'expression des besoins.

Enfin, le schéma ITC prescrit une résolution du problème par optimisation combinatoire en intégrant une fonction coût pondérant 4 critères qui pénalisent respectivement les choix de créneaux et de salles pour les classes, les violations de contraintes et le chevauchement de séances par étudiant. Dans sa version actuelle, le langage UTP traite le problème comme un problème de satisfaction de contraintes dures. L'intégration de contraintes souples et la possibilité d'aggréger pénalités ou préférences, que ce soit dans des contextes de construction ou de réparation de solution, est à l'étude.

3 Modèles MiniZinc et CHR

Nous présentons dans cette section deux modèles d'instances UTP développés en MiniZinc et CHR. Ces modèles mettent en jeu des contraintes relatives au partitionnement des étudiants en groupes et à l'attribution des groupes aux classes, à la distribution des ressources sur les séances, à l'ordonnancement des séances, et à l'allocation de leurs




```
sequenced(<(K,_,{3}),(P,algoLec,_)>, <(K,_,{1}),(P,algoLab,_)>)                    (R1)
forbidden_period((<(T,lecturer2,_)>,9120,9240)                                     (R2)
sequenced((algoLec1,{algoLec1:3}), (algoLab1,{algoLab1:1}))                        (C1)
sequenced((algoLec1,{algoLec1:3}), (algoLab2,{algoLab2:1})                         (C2)
forbidden_period((lecturer2,{algoLab1:1,algoLab1:2,algoLab2:1,algoLab2:2}),9120,9240)  (C3)
```

FIGURE 4 – Sélection de séances par des règles

ressources. Nous présentons tout d'abord les données d'instance ainsi que les variables de décision qui sont communes aux deux modèles. La table 2 liste les plages d'entiers identifiant les différents ensembles d'objets manipulés et définit les structures utilisées pour représenter les données d'instance.

3.1 Modèle `MiniZinc`

`MiniZinc` est un langage de modélisation haut-niveau de problèmes d'optimisation sous contraintes [18, 3]. Les modèles `MiniZinc` sont traduits dans le langage cible `Flatzinc` [4] qui permet d'interfacer différents types de solveurs dont les solveurs de programmation par contraintes sur domaines finis tels `Gecode` [2]. `MiniZinc` intègre de nombreuses contraintes globales et le modèle `UTP` présenté en Table 4 et utilisant les variables de décision présentée en Table 3 s'appuie sur quelques contraintes dédiées aux problèmes d'ordonnancement.

Les contraintes de sectionnement répartissent les étudiants dans les groupes et assigne chacun de ces groupes à différentes classes conformément aux règles de sectionnement et aux seuils d'effectifs. La contrainte (1) n'autorise le regroupement d'étudiants que s'ils sont inscrits aux mêmes cours. (2) impose que tout étudiant, assimilé à son groupe, assiste à toute partie de cours dans lequel il est inscrit. (3) assure que les classes d'une partie de cours n'ont aucun groupe en commun. (4) implémente la relation de parenté entre classes. Enfin, (5) vérifie que l'effectif cumulé des groupes attribués à une classe ne dépasse pas le seuil autorisé.

La distribution des ressources s'appuie sur des contraintes de domaine, de cardinalité et de sommes. Les contraintes (6) et (7) définissent les salles et enseignants allouables à chaque séance. (8) contraint le nombre de salles allouées à une séance selon que sa partie de cours est sans salles, mono-salle, ou multi-salles. (9) attribue le nombre attendu d'enseignants à chaque séance et (10) vérifie que chaque enseignant dispense le volume de séances requis par partie de cours où il est pré-positionné.

La programmation des séances et l'allocation des ressources met en jeu des contraintes de positionnement, de séquencement, de non-chevauchement et de capacité. La contrainte (11) définit les créneaux autorisés pour chaque séance. (12) interdit qu'une séance soit à cheval sur 2 journées. (13) séquence les séances d'une classe selon leur rangs. Les contraintes (14) et (15) modélisent les séances multi-salles et l'accès exclusif à leurs ressources. (14) impose qu'une ressource allouée à une séance multi-salles soit disjonctive le temps de son utilisation. (15) assure que le nombre d'étudiants attendus n'excède pas la capacité cumulée des salles allouées. À noter que cette contrainte est purement quantitative et autorise toute répartition d'étudiants dans les salles indépendamment de la structure de groupes. (16) modélise les salles à allouer obligatoirement à toute séance d'une partie de cours. (17) modélise la contrainte de capacité cumulative qui s'applique par défaut à toute salle allouée hors séances multi-salles.

La table 4 présente les variantes de quelques prédicats `UTP` dans le cas où les entités ciblées sont des salles. (18) implémente le prédicat `forbidden_period` qui prend en paramètres les 2 créneaux modélisant la période interdite. (19), (20) et (21) modélisent de manière directe les prédicats `same_weekday`, `same_rooms` et `sequenced`. (22) implémente le prédicat `no_overlap` en s'appuyant sur la contrainte globale `disjunctive`.

3.2 Modèle `CHR`

Constraint Handling Rules (`CHR`) [13, 12] est un langage à base de règles d'inférence à chaînage avant qui remplacent les contraintes du problème par d'autres plus simples jusqu'à la résolution complète. `CHR` est un langage spécialisé

H	ensemble des créneaux définissant l'horizon de temps
C	ensemble des cours
P	ensembles des parties de cours
K	ensembles des classes
S	ensembles des séances
R	ensemble des salles
T	ensemble des enseignants
G	ensemble des groupes d'étudiants
U	ensembles des étudiants

class_{sessions,parents} :
ensemble des séances (resp. classes parentes) d'une classe
part_{classes,lecturers, rooms,sessions} :
ensemble des classes (resp. enseignants, salles, séances) d'une partie
room_sessions :
ensemble des séances possibles pour une salle
session_{part,class} : la partie (resp. classe) d'une séance
student_{courses,parts} :
les cours (resp. parties) que suit un étudiant
mandatory_rooms : les salles obligatoires pour une partie
single_room_sessions : l'ensemble des séances mono-salle
capacity : capacité maximum d'une salle ou d'une classe
is_multi_rooms : indique si une séance est multi-salles
length : durée d'une séance
part_room_use :
modalité d'utilisation de salles pour une partie (none, single, multiple)
rank : rang d'une séance
service :
nombre de séances à dispenser par enseignant par partie
teams :
nombre d'enseignants requis par séance d'une partie
virtual : indique si une salle est de capacité illimitée ou non
dailyslots : créneaux quotidiens autorisés pour une partie
weekdays : journées autorisées pour une partie
weeks : semaines autorisées pour une partie
nr_daily_slots : nombre de créneaux dans une journée
nr_weekly_slots : nombre de créneaux dans une semaine

TABLE 2 – Données d'instances et fonctions utilitaires

permettant de définir des contraintes déclaratives au sens de la programmation logique par contraintes [14, 16]. CHR est une extension de langage qui permet d'introduire des contraintes définies par l'utilisateur, c'est-à-dire des prédicats du premier ordre, dans un langage hôte donné tel que Prolog, Lisp, Java, ou C/C++. Il a ensuite été étendu à $\text{CHR}^{\vee}$ [5] qui introduit le *don't know* [10]. Ce non-déterminisme est offert gratuitement lorsque le langage hôte est Prolog et il permet de spécifier facilement des problèmes de la classe de complexité NP. Pour modéliser et résoudre les instances UTP avec le langage CHR, nous utilisons le solveur CHR++ [8] (pour Constraint Handling Rules in C++), qui est une intégration efficace de CHR dans le langage de programmation C++.

Le modèle CHR est instancié à la lecture de l'instance au format JSON. Le modèle d'entité est d'abord défini, puis les contraintes issues des règles sont déclarées et enfin les domaines des variables sont mis à jour si une partie solution est fourniée, avant de lancer la résolution de l'instance. Le modèle complet pour CHR++ est trop long pour être détaillé ici [1]. Nous donnons dans le tableau 6 la liste des contraintes prises en compte par le solveur. Les variables de décisions devant être instanciées sont données au tableau 5. Elles sont en grande partie similaires à celles du modèle MiniZinc, seules les variables de fin de séances sont ajoutées.

Pour simplifier son implémentation, le modèle CHR est en partie non cumulatif et certaines ressources comme les enseignants ne peuvent se partager. Il considère également que le sectionnement et l'affectation des étudiants aux groupes est fait en amont. Ainsi, calculer une solution revient à trouver une affectation des ressources consistante tout en plaçant les horaires de toutes les séances.

Plusieurs contraintes peuvent être posées dès l'analyse de l'instance. C'est le cas des contraintes (1) à (9) du tableau 6. Les contraintes (2), (3) et (4) filtrent les domaines en retirant les salles, enseignants ou horaires impossibles par construction de l'instance. La contrainte (5) assure qu'une séance commence et se termine le même jour en retirant du domaine les valeurs qui la contredisent.

D'autres contraintes sont posées et gérées par des règles CHR surveillant les modifications des domaines des variables. C'est le cas de la contrainte (1) qui assure l'intégrité des variables de début et de fin de séance. Il en est de même pour (6) qui assure que le nombre d'enseignants demandé pour une session est valide et (7) qui vérifie que le nombre de salles d'une séance correspond bien à ce qui est demandé dans l'instance.

Nous donnons pour exemple la règle CHR++ qui vérifie l'intégrité des variables de début et de fin de séance. Celle-ci est déclenchée dès qu'un domaine d'une variable est mis à jour :

```
session_slot(_, S_Start, S_End, S_Length)
  =>> CP::Int::plus(S_Start, (*S_Length)-1, S_End);;
```

Nous utilisons CHR++ qui permet de manipuler des valeurs associées à des variables logiques et de réveiller les règles correspondantes dès qu'une modification de la valeur survient. Ce mécanisme combiné avec le chaînage avant de CHR nous permet d'implémenter un mécanisme de réveil des règles et de propagation des domaines efficace à la manière d'un solveur CSP.

Les contraintes (8) et (9) ajoutent de nouvelles contraintes CHR au modèle. En effet, les contraintes before et disjunct sont des contraintes CHR assurant la précédence et le non chevauchement de deux séances. Elles sont accompagnées de règles vérifiant la cohérence du graphe disjonctif créé implicitement par l'ajout de toutes ces contraintes. Les prédicats statiques correspondent à ceux lus depuis l'instance. Ils sont traités et des contraintes (contraintes de filtrage, contraintes CHR ou unification de variables) sont ajoutées.

Les contraintes dynamiques de (13) à (18) ne se déclenchent que dans certaines conditions. Des règles CHR

1. Le lecteur intéressé peut télécharger les sources du modèle [1]

array[U] of var G :	x_group	groupe attribué à un étudiant
array[K] of var set of G :	x_groups	ensemble de groupes alloués à une classe
array[S] of var set of R :	x_rooms	ensemble de salles allouées à une séance
array[S] of var set of T :	x_lecturers	ensemble d'enseignants alloués à une séance
array[S] of var H :	x_slot	créneau de départ attribué à une séance

TABLE 3 – Variables de décision (`MiniZinc`)

forall(u, v in U where u<v) (student_courses[u] !=student_courses[v] -> x_group[u] !=x_group[v])	(1)
forall(u in U, p in student_parts[u])(exists(k in part_classes[p])(x_group[u] in x_groups[k]))	(2)
forall(p in $P, k1, k2$ in part_classes[p] where $k1$<$k2$)(x_groups[$k1$] intersect x_groups[$k2$] = {})	(3)
forall($k1$ in $K, k2$ in class_parents($k1$))(x_groups[$k1$] subset x_groups[$k2$])	(4)
forall(k in K)(maxsize[k]<=sum(g in G)(bool2int(g in x_groups[k]) $*$ sum(u in U)(bool2int(x_group[u] = g)))	(5)
forall(s in S)(x_rooms[s] subset part_rooms[session_part[s]])	(6)
forall(s in S)(x_lecturers[s] subset part_lecturers[session_part[s]])	(7)
forall(s in S, p in P where p = session_part[s])((part_room_use[p] = none -> x_rooms[s] = {}) $\wedge$ (part_room_use[p] = single -> card(x_rooms[s]) = 1) $\wedge$ (part_room_use[p] = multiple -> card(x_rooms[s])>=1))	(8)
forall(s in S)(card(x_lecturers[s]) = team[session_part[s]])	(9)
forall(p in P, l in part_lecturers[p])(sum(s in part_sessions[p]))(bool2int(l in x_lecturers[s]) = service[l, p]))	(10)
forall(p in P, s in part_sessions[p]) (week(x_slot[s]) in weeks[p] $\wedge$ weekday(x_slot[s]) in weekdays[p] $\wedge$ dailyslot(x_slot[s]) in dailyslots[p])	(11)
forall(s in S)((x_slot[s] − 1) div nr_daily_slots = (x_slot[s] + length[s] − 1) div nr_daily_slots)	(12)
forall(k in $K, s1, s2$ in class_sessions[k] where rank($s1$)<rank($s2$)) (x_slot[$s1$] + length[s]>=x_slot[$s2$])	(13)
forall(p in $P, s1$ in part_sessions[p], r in part_rooms[p], $s2$ in room_sessions[r] where is_multi_rooms[p] $\wedge$ $s1$!=$s2$) (disjunctive([x_slot[$s1$], x_slot[$s2$]], [bool2int(r in x_rooms[$s1$]) $*$ length[$s1$], bool2int(r in x_rooms[$s2$]) $*$ length[$s2$]]))	(14)
forall(p in P, s in part_sessions[p] where is_multi_rooms[p]) (sum(r in part_rooms[p])(bool2int(r in x_rooms[s]) $*$ capacity[r]) <=sum(g in G)(bool2int(g in x_groups[session_class[s]]) $*$ card(group_students[g]))))	(15)
forall(p in P, s in part_sessions[p])(mandatory_rooms[p] subset x_rooms[s])	(16)
forall(r 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]) $*$ length[s]|s in RS], [sum(g in G)(bool2int(g in x_groups[session_class[s]])) $*$ sum(u in U)(bool2int(g = x_group[u]))|s in RS], capacity[r]))	(17)
$forbidden\_period((r, S'), h1, h2)$ = forall(i in S')(r in x_rooms[i] -> (x_slot[i] + length[i]<=h_1 \/x_slot[i]>h_2))	(18)
$same\_weekday((r, S'))$ = forall(i, j in S' 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)
$same\_rooms((r, S'))$ = forall(i, j in S' 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] $\wedge$ $r2$ in x_rooms[j]) -> x_slot[i]+length[i]<=x_slot[j])	(21)
$no\_overlap((r, S'))$ = disjunctive([x_slot[i]|i in S'], [length[i] $*$ bool2int(r in x_rooms[i])|i in S'])	(22)

TABLE 4 – Contraintes et prédicats du modèle

array[S] of var set of R : x_rooms	ensemble de salles allouées à une séance
array[S] of var set of T : x_lecturers	ensemble d'enseignants alloués à une séance
array[S] of int H : x_slot_start	créneau de départ attribué à une séance
array[S] of int H : x_slot_end	créneau de fin attribué à une séance

TABLE 5 – Variables de décision (`CHR`)

avec garde sont utilisées à cet effet. (13) vérifie qu'un enseignant effectue bien la liste des enseignements auxquels il est inscrit. (14) assure que la capacité des salles est respectée et (15) vérifie que les salles marquées comme obligatoires se retrouvent bien dans la solution. Le prédicat (16) assure que des séances associées au même prédicat $same\_weekday$ sont fixées sur le même jour de la semaine.

Les contraintes (17) et (18) ajoutent des contraintes `CHR disjunct` lorsque certaines conditions sont vérifiées. Ainsi, (17) pose un `disjunct` entre deux séances lorsqu'un même enseignant y participe. (18) ajoute une contrainte `disjunct` entre deux séances si celles-ci ont lieu dans la même salle. Ces contraintes `CHR` viennent enrichir le graphe disjonctif représentant le séquencement de

Contrainte d'intégrité :

$\forall s \in S$: x_slot_end$[s]$ = x_slot_start$[s]$ + length(s) (1)

Contraintes statiques (filtrage des entrées de l'instance) :

$\forall s \in S$: x_rooms$[s] \subseteq$ part_rooms[session_part(s)] (2)

$\forall s \in S$: x_lecturers$[s] \subseteq$ part_lecturers[session_part(s)] (3)

$\forall p \in P, \forall s \in$ part_sessions(p) : $\big($week(x_slot_start$[s]) \in$ weeks$[p]\big)$
$\wedge\big($weekday(x_slot_start$[s]) \in$ days$[p]\big) \wedge \big($dailyslot(x_slot_start$[s]) \in$ dailyslots$[p]\big)$ (4)

$\forall s \in S$: x_slot_start$[s]/nr_daily_slots =$ x_slot_end$[s]/nr_daily_slots$ (5)

$\forall s \in S$: card(x_lecturers$[s]$) = team[part_sessions$[s]$] (6)

$\forall k \in K, \forall s \in$ class_sessions$[k]$:
Si $\big($part_room_use[class_part$(k)]$ = none$\big)$ alors card(x_rooms$[s]$) = 0
Si $\big($part_room_use[class_part$(k)]$ = single$\big)$ alors card(x_rooms$[s]$) = 1
Si $\big($part_room_use[class_part$(k)]$ = multiple$\big)$ alors card(x_rooms$[s]$) ≥ 1 (7)

$\forall k \in K, \forall s, s' \in$ class_sessions$[k], s.t.$ rank$(s) <$ rank(s') : before(s, s') (8)

$\forall k_1, k_2 \in K, s.t. \exists g_1 \in$ class_groups$[k_1], \exists g_2 \in$ class_groups$[k_2],$ avec $g_1 = g_2$:
$\forall s_1 \in$ class_sessions$(k_1), s_2 \in$ class_sessions(k_2) : disjunct(s_1, s_2) (9)

Prédicats statiques :

$forbidden\_period((e, S'), h, h') = \forall i \in S'$: (x_slot_start$[i] < h) \wedge$ (x_slot_start$[i] > h')$ (10)

$sequenced((e_1, S_1), (e_2, S_2)) = \forall i_1 \in S_1, \forall i_2 \in S_2$: before(i_1, i_2) (11)

$same\_rooms((e, S')) = \forall s_1, s_2 \in S', s.t.\ s_1 < s_2$: x_rooms$[s_1] \sim$ x_rooms$[s_2]$ (12)

Contraintes dynamiques :

$\forall p \in P, \forall l \in$ part_lecturers$[p]$: $\big|\big|\{x \mid x \in$ part_sessions$(p), l \in$ x_lecturers$[x]\}\big|\big| =$ service$[p, l]$ (13)

$\forall s \in S, \forall r \in$ session_rooms(s) :
$\sum\{$group_students$[g] \mid g \in$ x_groups[session_class$[s]], r \in$ x_rooms$[s]\} \leq$ room_capacity$[r]$ (14)

$\forall s \in S$: mandatory_rooms[session_part$[s]] \subseteq$ x_rooms$[s]$ (15)

Prédicat dynamique :

$same\_weekday((e, S')) =$
$\forall s_1, s_2 \in S', s.t.\ s_1 < s_2$: x_slot_start$[s_1]/nr_weekly_slots =$ x_slot_start$[s_2]/nr_weekly_slots$ (16)

Contraintes introspectives :

$\forall k_1, k_2 \in K, \forall s_1 \in$ class_sessions$[k_1], \forall s_2 \in$ class_sessions$[k_2],$ s.t. $s_1 \neq s_2$:
x_lecturers$[s_1] \cap$ x_lecturers$[s_2] \neq \emptyset \Rightarrow$ disjunct(s_1, s_2) (17)

$\forall k_1, k_2 \in K, \forall s_1 \in$ class_sessions$[k_1], \forall s_2 \in$ class_sessions$[k_2]$ s.t. $s_1 \neq s_2$:
x_rooms$[s_1] \cap$ x_rooms$[s_2] \neq \emptyset \Rightarrow$ disjunct(s_1, s_2) (18)

TABLE 6 – Contraintes et prédicats du modèle CHR

toutes les séances.

Il est à noter que le modèle CHR effectue du filtrage de domaine mais analyse également le graphe disjonctif afin d'éliminer des non solutions. Les arêtes du graphe disjonctif sont orientées au fur et à mesure de l'avancement de la résolution et de l'instanciation des variables de décision.

4 Expérimentations

Nous avons mené des expérimentations sur une instance réelle modélisant le second semestre de la Licence 3 d'informatique à l'Université d'Angers. L'instance est disponible aux formats XML, JSON et DZN sur le site [1].

L'instance comporte 5 cours communs à tous les étudiants et 2 choix d'options, chacun portant sur 2 cours, soit un total de 7 cours suivis par les étudiants sur les 9. L'instance compte 24 parties de cours et 42 classes. Les séances sont à programmer sur un horizon de 12 semaines de 5 jours chacune où chaque journée se divise en créneaux de 1 minute. Les séances doivent être placées sur une grille horaire qui commence à 08:00, se termine à 19:50 et est composée de plages d'1h20 espacées de 10 minutes. Une séance qui dure 1 plage a donc une durée de 80 créneaux et a 8 créneaux de départ possibles. Certaines séances durent 2 plages, et ont donc une durée de 170 créneaux avec 7 créneaux de départ possibles. Dans le cas où une séance dure 2h (120 créneaux), la séance doit commencer ou se terminer pour s'aligner sur la grille, soit 13 créneaux de départ possibles. L'instance est constituée de 67 étudiants prédivisés en 4 groupes, 12 enseignants et 8 salles. Elle intègre 46 règles dont une majorité de règles coordonnant les séances (parallélisation entre classes de travaux pratiques ou options, séquencement entre cours magistraux, travaux dirigés et travaux pratiques, etc.) et quelques règles restreignant salles et enseignants possibles selon les cours.

Les solveurs MiniZinc et CHR++ présentés en section 3 ont été utilisés pour résoudre cette instance avec une architecture Intel Core i7-10875H 2.30GHz et la résolvent en moins de 5s (hors flattening pour MiniZinc).

La stratégie de résolution employée dans le modèle MiniZinc consiste d'abord à allouer les salles, puis les enseignants avant de placer les séances sur l'horizon de temps. Les variables d'allocation sont ordonnées par l'heuristique first_fail et leurs domaines de valeurs explorés de manière systématique. Les variables de choix de créneaux par séance utilisent aussi l'heuristique first_fail et l'heuristique de choix de valeurs consiste à scinder chaque domaine (indomain_split). Gecode

est le solveur utilisé avec MiniZinc dans nos tests. À noter que les contraintes disjonctives y sont implémentées comme un cas particulier de la contrainte globale cumulative présentée dans [9].

La stratégie de résolution employée avec CHR++ consiste à instancier les variables de décisions en commencant par le tableau de variables x_rooms, puis x_lecturers et enfin x_slot_start (les autres variables sont déduites par propagation). Dans chaque tableau, la prochaine variable à instancier est choisie selon l'ordre de définition dans le tableau et la valeur testée est toujours la plus petite valeur possible du domaine. Entre chaque instanciation, une phase de propagation des contraintes (filtrage des domaines et analyse du graphe disjonctif) est itérée jusqu'à l'obtention d'un point fixe. En cas d'échec, la méthode revient sur son choix précédent pour essayer l'alternative suivante. Il n'y a pour l'instant aucune heuristique spécialisée, mais le choix de la plus petite valeur du domaine semble pertinent. En effet, pour construire un emploi du temps, il est naturel de commencer à fixer les séances en partant du début de l'horizon de temps.

5 Conclusion et perspectives

Dans cet article, nous avons introduit brièvement le langage UTP qui permet de modéliser la problématique de construction d'emplois du temps universitaires. Le langage est générique et permet de s'adapter à diférentes variantes du problème UTP. Par exemple, les ressources sont cumulatives par défaut mais des règles peuvent être surimposées pour rendre certaines ressources disjonctives. De plus, le langage s'appuie sur un catalogue de prédicats qui peut être enrichi afin de s'adapter aux spécificités de différents environnements, et ce sans modifier le langage lui-même.

Dans sa version actuelle, le langage UTP réduit la génération d'emploi du temps à un problème de satisfaction de contraintes dur et ne prend en compte aucun critère d'optimisation. Nous avons pour objectif de développer cet aspect, afin entre autres de pouvoir exprimer des préférences, et définir des méthodes d'évaluation d'une solution pour pouvoir choisir la plus adaptée aux souhaits des décideurs (p. ex. éviter de trop longues interruptions de cours dans une journée pour les étudiants ou regrouper les cours sur des demi-journées pour les enseignants).

Nous avons détaillé également deux modèles de programmation par contraintes implémentés en MiniZinc et CHR++. Ces deux modèles ont été développés comme preuve de concept et seront améliorés notamment en implémentant des stratégies de résolution facilitant le passage à l'échelle sur des instances plus conséquentes.

Remerciements

Ce travail est partiellement financé par le projet Thélème octroyé aux universités d'Angers et du Mans dans le cadre du PIA3.

Références

[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 Query Language. 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, 20-25/09/2019 2019.

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

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

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

[12] T.W. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[13] T.W. 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 M.J. 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] N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. Minizinc : Towards a standard cp modelling language. In *CP 2007*, pages 529–543, 2007.