4. Méthode approchée basée sur la satisfaction de contraintes (CSP)

Selon l'analyse bibliographique de la section 2.3 du chapitre II (cf. le Tableau 4), pour beaucoup de problèmes de taille réaliste (supérieure à 80 interventions par jour, par exemple), une méthode exacte peut nécessiter un temps de calcul très long pour trouver une solution optimale, pouvant atteindre des durées incompatibles avec le besoin (plusieurs jours ou mois). Il est de grande complexité au sens de l'analyse combinatoire. Pour tenter de réduire ce temps de calcul, nous proposons une résolution de problèmes HHCRSP formulés sur la base d'un CSP dont la particularité réside dans l'utilisation active des contraintes du problème pour réduire la taille de l'espace des solutions à parcourir.

4.1. PIM : modèle en satisfaction de contrainte (CSP)

Un CSP (*Constraint Satisfaction Problem*) (Floudas and Pardalos, 1990) est une méthode de résolution très utilisée pour des problèmes réels avec des contraintes. La programmation par contraintes est un paradigme puissant pour résoudre des problèmes d'optimisation combinatoire en faisant appel à un large éventail de techniques, relevant de l'intelligence artificielle (Rossi et al., 2006). L'idée de la programmation par contraintes est de résoudre des problèmes en énonçant des contraintes sur le domaine du problème et, par conséquent, de trouver une solution les satisfaisant toutes (Barták, 1999). Selon (Russell and Norvig, 2002), un CSP est défini par un triplet (X, D, C) tel que :

- $X = \{X_1, X_2, \ldots, X_N\}$ est l'ensemble des variables du problème.
- D est une fonction qui associe à chaque variable X_i son domaine $D(X_i)$ (valeurs possibles de X_i).
- $C = \{C_1, C_2, \ldots, C_M\}$ est l'ensemble des contraintes du problème. Chaque contrainte est définie par un couple (t, R) tel que : t est l'ensemble des variables liées à la contrainte, $t \in X$, R définit une relation entre ces variables.

Obéissant à cette définition, nous construisons dans la Figure 44 un méta-modèle représentant un CSP.

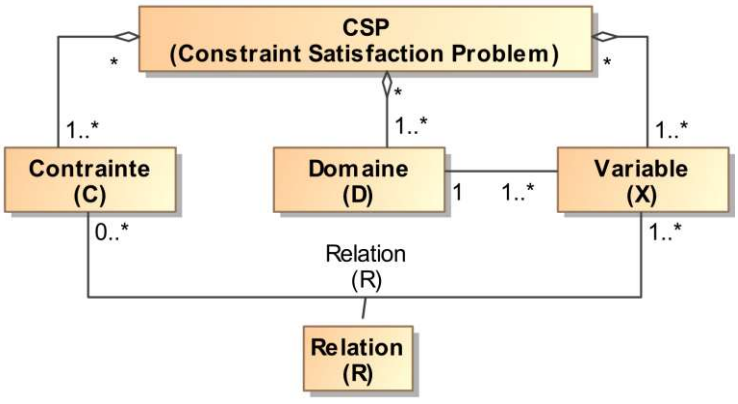

Figure 44 : Méta-modèle du CSP

Les CSP sont multiples, nous pouvons citer à titre d'exemples, le problème de la coloration de graphe (Jensen and Toft, 2011) et le problème des N-Reines (Gutiérrez Naranjo et al., 2009). La formulation en CSP est très peu utilisée pour la résolution du problème HHCRSP (7% des publications selon notre analyse bibliographique, cf. la section 2.3.4 du chapitre II et le Tableau

4). La complexité d'un CSP est NP-difficile, en général (Rossi et al., 2006). En bref, cela signifie que la solution optimale ne peut être déterminée dans un délai raisonnable, tout comme la formulation basée sur la PLNM. Avec des cas du monde réel, les objectifs de Berger-Levrault sont de résoudre un HHCRSP ayant une taille conséquente (dépassant la centaine d'interventions par jour), avec les contraintes que nous avons définies, en utilisant le moins de temps de calcul possible, et en investissant la juste proportion de ressources financières (l'achat de licence de solveur commercial, CPLEX, par exemple) et humaines dans le processus de développement de cet outil numérique de pilotage de la PAD. Afin de rentrer dans le cadre de ces exigences industrielles, nous avons opté pour une alternative, le solveur OptaPlanner (OptaPlanner, 2021).

OptaPlanner[1] est un solveur à base de satisfaction de contraintes. Nous avons trouvé des références dans la littérature où des travaux où OptaPlanner sont cités. Dans (Kosecka-Żurek, 2019), une application préexistante d'OptaPlanner a été utilisée pour résoudre le problème du VRP. OptaPlanner a également été utilisé pour résoudre une partie de VRP impliquant le mécanisme dit de collecte des déchets (Lozano Murciego et al., 2015). Dans un autre travail, il est fait référence à un problème d'attribution des tâches (Macik, 2016). Dans (Rios de Souza and Martins, 2020), le travail vise à explorer différentes solutions en utilisant OptaPlanner pour traiter un problème de conditionnement des poubelles, où il est question de comparer les résultats obtenus par des métaheuristiques différentes, embarquées dans OptaPlanner.

4.2. PSM : modèle en satisfaction de contrainte orienté OptaPlanner

4.2.1. Description d'OptaPlanner

OptaPlanner, publié sous licence Apache Software est un moteur de planification léger et ouvert. Il permet aux programmeurs Java de traiter des modèles spécifiques à un domaine en réutilisant un modèle existant. Les modèles mis à disposition dans OptaPlanner sont nombreux. Nous pouvons citer en particulier le VRP et le TSP (cf. la section 2.3.1.1 du chapitre II) qui sont voisins du sujet que nous traitons. Une des forces de cet outil numérique est de pouvoir combiner (1) des heuristiques visant à la génération d'une solution initiale du problème à résoudre, et (2) des métaheuristiques d'optimisation (la recherche tabu, par exemple) pour parcourir l'espace de recherche, avec un système de calcul de score très performant. Comme expliqué dans la section 2.3 du chapitre II, les métaheuristiques sont applicables à un très large éventail de HHCRSP. En fait, le principal avantage d'une métaheuristique est qu'elle ne nécessite pas beaucoup de réglages pour chaque cas d'utilisation. Il faut uniquement choisir un petit nombre de paramètres intrinsèques à l'algorithme (paramètres de température pour le recuit simulé, par exemple). Beaucoup de ces algorithmes démontrent qu'en leur donnant suffisamment de temps, ils convergent plus ou moins rapidement vers une solution acceptable (Blum and Roli, 2003). Pour des cas d'utilisation réalistes, une solution quasi-optimale peut souvent être considérée comme acceptable. Il appartient à l'utilisateur de trouver le bon compromis entre le temps de calcul et l'atteinte de l'optimalité du résultat (Macik, 2016).

La construction d'un modèle d'optimisation avec OptaPlanner se réalise en 3 étapes.

[1] https://www.optaplanner.org/

4.2.1.1. Principes de modélisation d'un CSP avec OptaPlanner

OptaPlanner est conçu selon une approche orientée objet. Cela impacte la manière de modéliser le problème spécifié. Il s'agit de définir des formules mathématiques par sélection et assemblage de classes existantes au moyen d'une interface. La configuration emploie un mécanisme d'annotations.

Nous distinguons trois catégories de classes définies ci-après.

(1) Classe d'information du problème (*Problem Fact*)

Ce sont les classes ne pouvant être changées par le solveur lors de la recherche de solution (les données restant identiques lors de la recherche de solutions), par exemple : « **Bénéficiaire** » et « **Structure de PAD** ». Toutes les propriétés d'une classe d'information d'un problème sont des propriétés du problème. Ces classes sont sans annotation. Ainsi, nous pouvons affirmer que parmi toutes les classes définies dans le modèle, si la classe n'est pas annotée, cela signifie qu'il s'agit d'une classe « *Problem Fact* ».

(2) Classe d'entité de planification (*Planning Entity*).

Ces classes peuvent subir des changements pendant le processus de recherche de la solution. Les propriétés qui changent au cours de la planification sont des variables de décision dites principales (*Planning Variable*). Ces classes sont annotées par le mot clé @PlanningEntity, leurs propriétés sont annotées par @PlanningVariable, et les autres sont des propriétés sans annotation.

Dans certains cas d'utilisation que nous avons testés, comme le TSP, le VRP ou le HHCRSP, il est nécessaire d'enchaîner les instances du *Planning Entity* dans le modèle. Ceci signifie que les *Planning Entity* forment une chaine en pointant les uns vers les autres qui conduit, *in fine*, à prendre en charge une structure d'ordre entre les entités à planifier. Chaque *Planning Entity* fait donc partie d'une chaîne ouverte (*open-ended chain*) qui part d'une ancre et finit à un autre (cf. la Figure 45). On peut noter une série d'exigences pour la construction d'une chaîne dans un tel modèle (OptaPlanner, 2021) :

- Une chaîne n'est jamais une boucle. Le maillon final est toujours ouvert.
- Chaque chaîne possède toujours exactement une « ancre » (*anchor*). Cette « ancre » n'est jamais une instance de la classe *Planning Entity*.
- Une chaîne n'est jamais un arbre, elle est toujours linéaire. Chaque « ancre » ou *Planning Entity* a donc un et un seul *Planning Entity* en bout de chaine.
- Chaque *Planning Entity* fait partie d'une chaîne.
- Une ancre sans *Planning Entity* pointant vers elle-même est également considérée comme une chaîne.

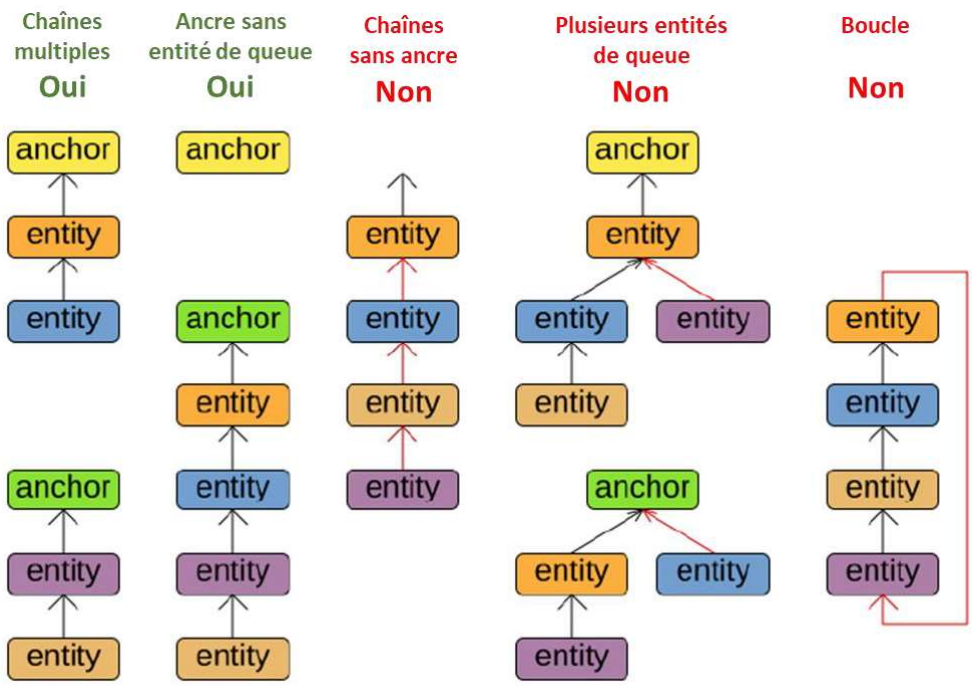

Figure 45 : Exemples de chaînes valides et invalides (OptaPlanner, 2021)

Lorsque les *Planning Entity* sont chaînés, les variables de décision (@PlanningVariable) définis dans ces entités sont de fait également chaînées :

- La variable de décision (annoté par @PlanningVariable) définie dans chaque objet du Planning Entity est considérée comme un pointeur vers un élément précédent. Elle est appelée variable principale.

Il existe dans OptaPlanner un concept de **variable secondaire** intitulée « variable d'ombre » qui est dépendante de la **variable principale**. Cette dépendance est de nature mathématique, c'est une relation de causalité pour le calcul du secondaire à partir du principal. Une « variable d'ombre » est une variable de planification dont la valeur correcte peut être déduite de l'état des variables de décision principales (@PlanningVariable) (OptaPlanner, 2021). Par exemple dans le TWVRP (*Time Windowed Vehicle Routing Problem*), le temps d'arrivée d'un véhicule (variable secondaire) chez un client peut être calculé en fonction des clients précédemment visités par ce véhicule (variable principale). Lorsque les clients pour un véhicule changent, l'heure d'arrivée prévue pour chaque client est automatiquement ajustée. Du point de vue de l'optimisation, OptaPlanner n'optimise effectivement que les variables de décision principales (@PlanningVariable) : il s'assure simplement que si une variable principale est modifiée, toutes les variables d'ombre dépendantes sont adaptées en conséquence. Ainsi, chaque variable d'ombre doit indiquer sa source par une annotation complémentaire comme par exemple : @CustomShadowVariable `(sourceVariableName="nom de variable principale")`.

Trois catégories de variables secondaires sont disponibles dans OptaPlanner comme suit :

- Une variable d'ombre customisée (annotée par @CustomShadowVariable) dépend de la variable de décision principale. Un *écouteur* doit être défini pour recalculer la valeur de la variable d'ombre customisée.

- Une variable d'ombre inversée (annotée par @InverseRelationShadowVariable) définie dans chaque objet du *Planning Entity* est considérée comme un pointeur vers l'élément suivant. Ce dernier agit sur le prochain objet du *Planning Entity* ou est nul si cet élément est le dernier. Cette variable coexiste en dualité avec la variable d'ombre customisée (@CustomShadowVariable)
- Un *Planning Entity* a une variable d'ancre (annotée par @AnchorShadowVariable), de sorte qu'il n'est pas nécessaire de traverser toute la chaîne pour obtenir son ancre.

La Figure 46 illustre par un exemple le codage en OptaPlanner des relations entre 4 types de variables dans le cas d'un modèle HHCRSP. Notons que l'« ***arrêt précédent*** » (@PlanningVariable) dans l'« **Intervention** » et la « ***prochaine intervention*** » (@InverseRelationShadowVariable) dans l'« **Arrêt** » visent à construire une chaîne entre les objets de l' « intervention ». Puis, l'« ***intervenant affecté*** » (@AnchorShadowVariable) à l' « **Intervention** » signifie que les objets d'« **Intervention** » chaînés doivent avoir un intervenant (l'objet de l'« **Intervenant »**) affecté. Enfin, le « temps de début planifié » (@CustomShadowVariable) dépend de l'« ***arrêt précédent*** » (@PlanningVariable), sa valeur doit être recalculée lorsque le chainage des interventions est modifié.

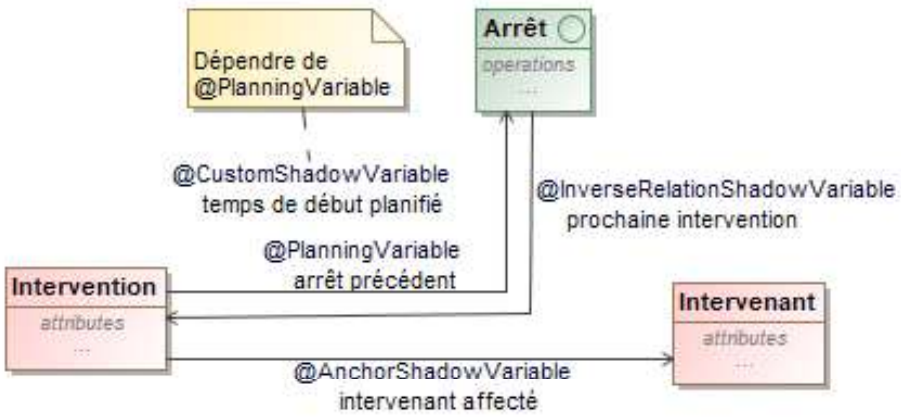

Figure 46 : Illustration de la relation entre quatre types de variable dans OptaPlanner

(3) Classe de solution de planification (*Planning Solution*).

Cette classe représente la solution du problème (annotée par @PlanningSolution), elle contient l'ensemble des objets (annotés par @ProblemFactCollectionProperty) instanciés de classes « *Problem Fact » et « Planning Entity ».*

OptaPlanner possède un système de score sur chaque contrainte pour juger la qualité d'une solution. Par défaut, ce système de score est composé de 3 niveaux (dur, moyen et souple) ou de 2 niveaux (dur et souple). Un score dur est significatif du respect de « contrainte dure ». A contrario, si le score dur d'une solution est inférieur à 0 (la valeur de pénalité quant à la violation de contrainte est souvent une valeur négative), cette solution est donc considérée comme une solution « infaisable ». Un score moyen ou souple est lié au respect de « contrainte souple ». Le score moyen et souple le plus élevé conduit à une meilleure solution.

Ce système de score est utilisé pour hiérarchiser un groupe de contraintes selon leur types. Par exemple, dans notre HHCRSP avec un score à 2 niveaux, un intervenant ne peut pas exécuter deux interventions en même temps et cela constitue une contrainte dure. Le respect de cette

contrainte est associé à un score dur. Le respect de la préférence personnelle d'un intervenant va relever d'une contrainte souple. Il est associé à un score souple. Les niveaux de deux scores différents sont comparés lexicographiquement, c'est-à-dire que la solution avec le score (0 dur / -200 souple) est meilleure que la solution avec le score (-1 dur / 0 souple). Car le score dur prime sur le score souple.

Le Tableau 12 est une synthèse du mécanisme d'annotations dans OptaPlanner.

Tableau 12 : Mécanisme d'annotations dans OptaPlanner

Nom	Annotation	Description
Classe d'information du problème	-	Ce sont les classes ne pouvant être changées par le solveur lors de la recherche de solution
Classe d'entité de planification	@PlanningEntity	Ces classes peuvent subir des changements pendant le processus de recherche de la solution
Classe de solution de planification	@PlanningSolution	Cette classe représente la solution du problème
	@ProblemFactCollectionProperty	L'ensemble des objets instanciés de classes « *Problem Fact* » et « *Planning Entity* »
Variable de décision principale	@PlanningVariable	Les propriétés qui changent au cours de la planification, appartenant à la classe d'entité de planification
Variable d'ombre customisée (Variable de décision secondaire)	@CustomShadowVariable	La valeur de cette variable est déduite de l'état des variables de décision principales
Variable d'ombre inversée (Variable de décision secondaire)	@InverseRelationShadowVariable	Cette variable coexiste en paire avec la variable d'ombre customisée
Variable d'ancre (Variable de décision secondaire)	@AnchorShadowVariable	Les entités de planification chaînées possèdent une « ancre », qui indique le début d'une chaîne. Par exemple un intervenant est une « ancre » quant à sa tournée.

4.2.1.2. Spécification des contraintes du problème

Cette phase consiste à écrire les règles représentant les contraintes du problème à respecter. OptaPlanner propose 4 méthodes à cette fin :

- Codage en Java : Implémentation de toutes les contraintes dans une méthode en Java. Cette manière de déclarer n'est pas évolutive.
- Définition par des flux de contraintes (*Constraint streams*) : Les *ConstraintStreams* sont une forme de programmation fonctionnelle pour le calcul incrémental de score en Java simple, qui est facile à lire, à écrire et à déboguer. Cette façon de déclarer les contraintes est évolutive.
- Codage en Java incrémental : Implémentation de plusieurs méthodes de bas niveau en Java pour déclarer les contraintes. Cette manière de déclarer est évolutive, mais très difficile à mettre en œuvre et à maintenir.
- Usage d'un moteur de règle, en l'occurrence, il s'agit du moteur Drools[1]: Implémentation de chaque contrainte comme une règle de score distincte en utilisant ce moteur de règles. Cette manière de déclarer est évolutive.

[1] Moteur de règle Drools :https://www.drools.org/

4.2.1.3. Résolution du problème

OptaPlanner opère généralement en deux étapes (cf. la Figure 47) : (1) la résolution du problème par le solveur et (2) l'évaluation de la solution par le calculateur de score.

(1) Pour la résolution du problème, une solution initiale est d'abord générée à l'aide d'heuristiques de construction, dans un temps limité et avec la meilleure qualité possible. Il existe 5 heuristiques de construction qu'OptaPlanner prend en charge : *First fit* avec sa variante *Weakest fit* et *Strongest fit* (Bays, 1977), *Allocate entity from queue* (Semeria, 2001), *Cheapest insertion* (Hassin and Keinan, 2008) et *Regret insertion* (Diana and Dessouky, 2004), *Allocate from pool* (Ke and Fang, 2004), *Scaling construction heuristics* (Katayama et al., 2009).

La deuxième phase consiste à rechercher continuellement par itération une meilleure solution que la solution courante grâce à des algorithmes de type métaheuristiques. Dans OptaPlanner, la configuration permet de choisir ces algorithmes d'optimisation et de définir leurs paramètres intrinsèques. L'optimisation se termine lorsque des critères d'arrêt sont atteints. Un critère d'arrêt est souvent défini comme l'atteinte, soit d'une limite maximum de temps de calcul, soit d'un certain nombre d'itérations. Tout comme l'heuristique de construction, OptaPlanner propose de nombreux types de métaheuristiques pour la recherche locale :

- La recherche locale simple : *Hill climbing* (Goldfeld et al., 1966) et *Late acceptance* (Burke and Bykov, 2012).
- Les métaheuristiques : TS (*Tabu Search*) (Glover and Laguna, 1998), SA (*Simulated Annealing*) (Van Laarhoven and Aarts, 1987), *Great deluge* (Dueck, 1993), VND (*Variable Neighborhood Descent*) (Gao et al., 2008).

(2) L'évaluation d'une solution est faite par le calculateur de score (*ScoreDirector*). Le score d'une solution caractérise la qualité d'une solution, et permet donc de savoir si une solution est meilleure qu'une autre. Plus le score est élevé, meilleure est la solution.

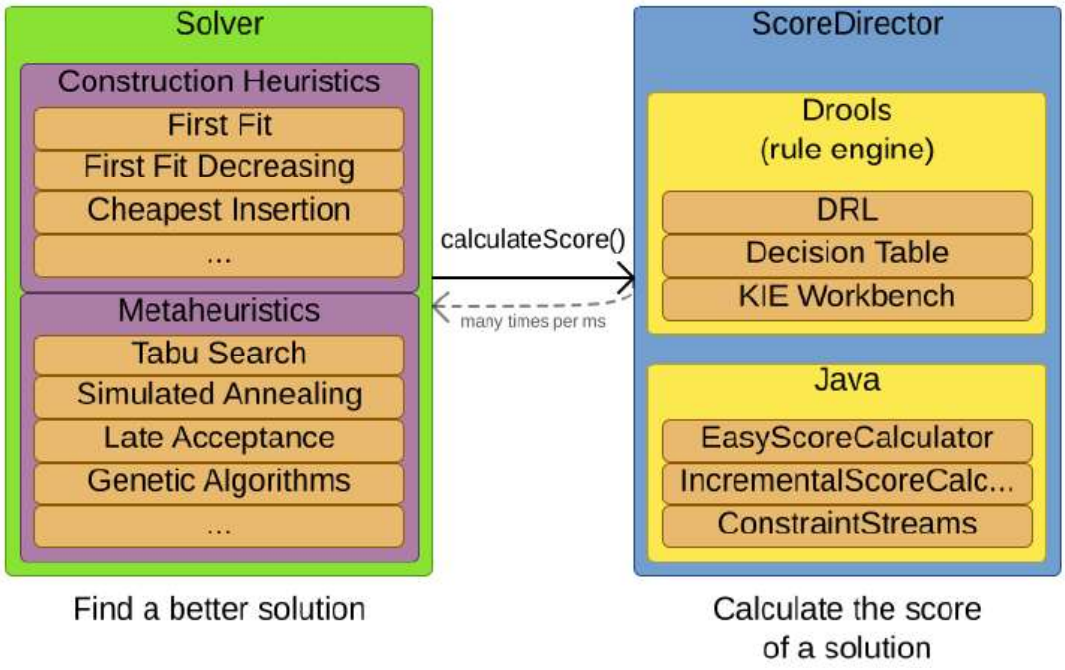

Figure 47 : Aperçu synthétique de mécanismes de résolution du problème par OptaPlanner (OptaPlanner, 2021)

4.2.2. Modélisation du problème

Les sous-sections suivantes utilisent les étapes évoquées précédemment. Nous illustrons ici notre développement de la méthode CSP avec OptaPlanner pour construire le modèle de prise de décision d'un HHCRSP.

Nous spécifions d'abord les classes utiles pour notre de domaine d'étude. Pour chaque classe et ses attributs, nous utilisons une notation afin de faciliter la description des contraintes faite à la section 4.2.3 de ce chapitre. La Figure 48 illustre ce choix de structuration.

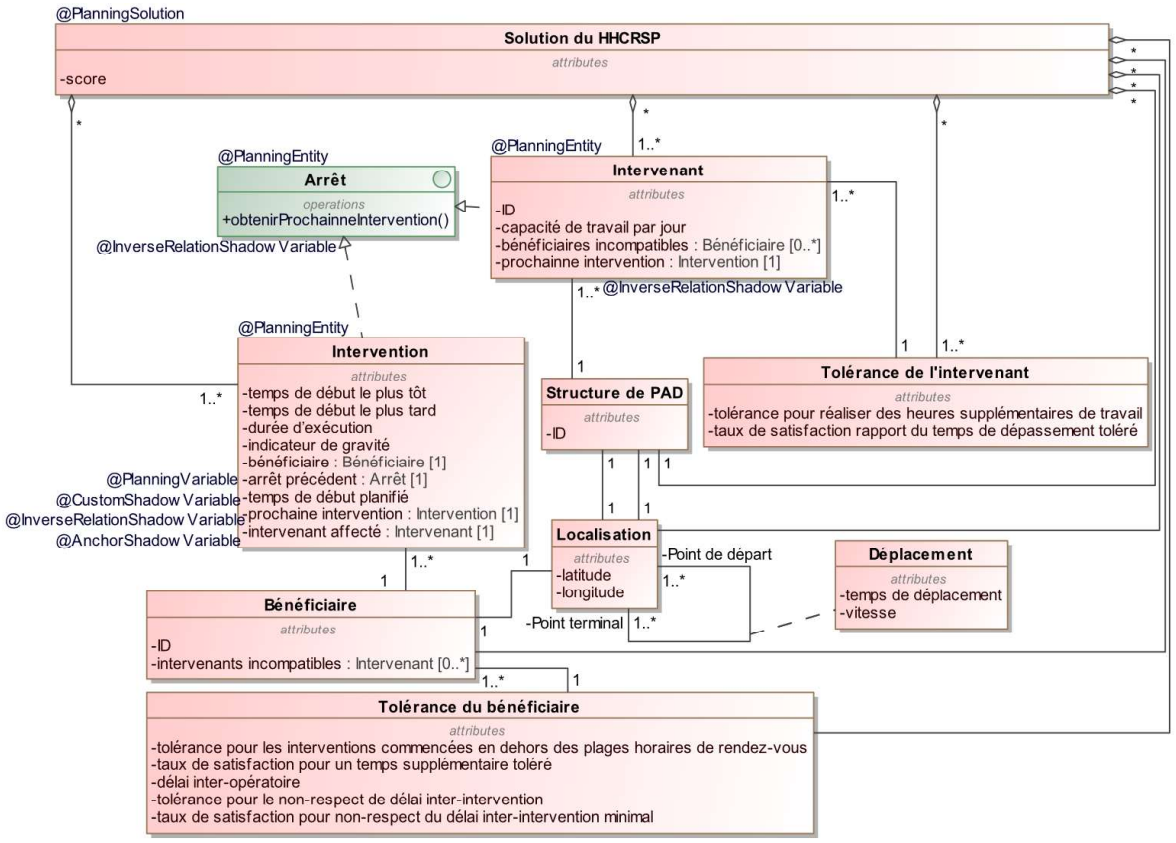

Figure 48 : Modèle du HHCRSP orienté OptaPlanner

4.2.2.1. Classe d'information du problème (*Problem Fact*)

Dans cette catégorie, les classes ne disposent pas de variables de décisions. Nous énumérons les paramètres avec leurs types en Java en les associant aux classes d'information dans le Tableau 13.

Tableau 13 : Classes d'information du modèle du HHCRSP

Nom de classe	Attribut			
	Nom	Type	Notation	Description
Structure de PAD (*PAD*)	ID	String	id_{pad}	-
Bénéficiaire (*P*)	ID	String	id_p	-
	Intervenants incompatibles	List<>	$imcomp_p$	Ces intervenants ne peuvent pas exécuter les interventions demandées par les objets de *P*.
	Délai inter-intervention	Long	δ_p	Comme δ_p dans la méthode exacte

Tolérance de bénéficiaire ($TolP$)	Tolérance pour les heures supplémentaires d'intervention	Long	α_p	Comme α_p dans la méthode exacte
	Taux de satisfaction pour le temps supplémentaire toléré	Long	α_p^*	Comme α_p^* dans la méthode exacte
	Tolérance pour non-respect des délais	Long	β_p	Comme β_p dans la méthode exacte
	Taux de satisfaction pour non-respect des délais toléré	Long	β_p^*	Comme β_p^* dans la méthode exacte
Tolérance d'intervenant ($TolK$)	Tolérance pour les heures supplémentaires de travail	Long	γ_k	Comme γ_k dans la méthode exacte
	Taux de satisfaction rapport du temps de dépassement toléré	Long	γ_k^*	Comme γ_k^* dans la méthode exacte
Localisation (LOC)	Altitude	Double	alt_{i^*},alt_{j^*}	Informations géographiques d'une localisation.
	Longitude	Double	lon_{i^*},lon_{j^*}	
Déplacement (DEP)	Temps de déplacement	Long	$temp_{i^*j^*}$	Classe relationnelle. Comme $temp_{i^*j^*}$ dans la méthode exacte, il s'agit du temps de déplacement entre deux localisations i^*, j^*.
	Vitesse	Long	vit	20 km/h comme vitesse moyenne estimée en centre-ville pour calculer $temp_{i^*j^*}$

Notons que $temp_{i^*j^*}$ est calculé à vol d'oiseau en fonction des 2 coordonnées terrestres de la classe « **Localisation** ». Pour déduire $temp_{i^*j^*}$, nous appliquons la distance euclidienne entre deux points i^* et j^* (Danielsson, 1980) comme le montre la relation (30).

$$temp_{i^*j^*} = \frac{\sqrt{(alt_{i^*} - alt_{j^*})^2 + (lon_{i^*} - lon_{j^*})^2}}{vit} \qquad (30)$$

4.2.2.2. Classes d'entité de planification (*Planning Entity*, @PlanningEntity)

Il y a trois classes de ce type dans la Figure 48 : « **Arrêt** », « **Intervenant** » et « **Intervention** ». Nous décrivons les classes d'entité de planification dans le Tableau 14. Concernant la relation de dépendance entre les variables de décision principales et les variables de décisions secondaires, nous avons « $arret_i$ » qui est à la source de toutes les autres variables secondaires : « $debut_i$ », « $prochaC_k$ », « $prochaC_i$ », « $affec\_i$ ». Aussi, nous décrivons les domaines associés aux variables de décision dans le Tableau 15 .

Tableau 14 : Classes d'entités de planification du modèle HHCRSP

Nom de classe	**Attribut**			
	Nom	**Type**	**Notation**	**Description**
Arrêt	-	-	-	Interface avec la méthode *obtenirProchaineIntervention()*. Cette méthode est annotée par @InverseRelationShadowVariable. Les classes « Intervention » et « Intervenant » implémentent cette interface arrêt dans un but de synchronisation. La spécification de la méthode permet de faire connaître l'attribut « prochaine intervention » dans ces deux classes. Cette interface est la base de la création d'une tournée pour chaque intervenant, puisque chaque tournée est constituée d'un intervenant et des interventions qu'il réalise.
Intervenant (K)	ID	String	id_k	-
	Charge maximale de travail /jour	Long	ξ_k	Paramètre. Comme ξ_k dans la méthode exacte
	Bénéficiaires incompatibles	List<>	$imcomp_k$	Paramètre. Élimination de l'affectation des objets de K aux interventions demandées par les bénéficiaires incompatibles.

	Prochaine intervention	Objet <Intervention>	$prochaC_k$	Variable d'ombre inversée basée sur la spécification de la méthode *obtenirProchainneIntervention()*.
Intervention (C)	Temps de début au plus tôt	Long	e_i	Paramètre. Comme $[e_i, l_i]$ dans la méthode exacte
	Temps de début au plus tard	Long	l_i	Paramètre. Comme $[e_i, l_i]$ dans la méthode exacte
	Durée d'exécution	Long	$duree_{i^*}$	Paramètre. Comme $duree_{i^*}$ dans la méthode exacte
	Indicateur de gravité	Long	dif_i	Paramètre. Comme dif_i dans la méthode exacte
	Bénéficiaire	Objet <Bénéficiaire>	$benef_i$	Paramètre. Bénéficiaire d'une intervention, une intervention est demandée par un bénéficiaire.
	Arrêt précédent	Objet <Arrêt>	$arret_i$	Variable de décision principale.
	Temps de début planifié	Long	$debut_i$	Variable d'ombre représentant le temps de début de chaque intervention.
	Prochaine intervention	Objet <Intervention>	$prochaC_i$	Variable d'ombre inversée basée sur la spécification de la méthode *obtenirProchainneIntervention()*.
	Intervenant affecté	Objet <Intervenant>	$affec_i$	Variable d'ancre représentant l'intervenant affecté.

Tableau 15 : Domaines des variables de décision du modèle HHCRSP

Nom de classe	Nom de variable	Type de variable	Domaine
Intervenant (K)	Prochaine intervention	Variable d'ombre inversée	$prochaC_k = \{C_1, C_2, \dots, C_r\}\ \forall k \in K$
Intervention (C)	Arrêt précédent	Variable de décision principale	$arret_i = \{C_1, C_2, \dots, C_r\} \cup \{K_1, K_2, \dots, K_n\}\ \forall i \in C$
	Temps de début planifié	Variable d'ombre	$debut_i = [0, 1440]\ \forall i \in C$
	Prochaine intervention	Variable d'ombre inversée	$prochaC_i = \{C_1, C_2, \dots, C_r\}\ \forall i \in C$
	Intervenant affecté	Variable d'ancre	$affec_i = \{K_1, K_2, \dots, K_n\}\ \forall i \in C$

4.2.3. Spécification des contraintes

Nous avons choisi le langage DRL (*Drools Rule Language*) de Drools pour coder les contraintes. Ce choix permet de déclarer les contraintes sous forme de règles qui sont séparées du modèle, ce qui permet de facilement ajouter, modifier ou même supprimer des contraintes. Drools fournit aussi le support du calcul incrémental des scores caractérisant l'optimalité d'une solution, sans code supplémentaire. La déclaration de contrainte en Drools se compose de deux parties en logique de premier ordre : la condition « **when** » et la conséquence « **then** ».

Dans la partie « **when** », nous spécifions la condition déclenchant la contrainte et déclarons les paramètres en utilisant la syntaxe « $ + nom ».

Dans la partie « **then** », nous déclarons les éléments nécessaires en vue du calcul de score. Par exemple, pour calculer le score de la contrainte [7], nous avons besoin des paramètres concernant la tolérance de l'intervenant quant au dépassement de sa capacité maximale de travail journalier : γ_k et γ_k^* . Pour ce faire, il nous suffit d'appeler les méthodes *getTolerance()* et *getToleranceRate()* définies dans la classe « Intervenant » pour obtenir γ_k et γ_k^* de chaque intervenant.

Pour notre HHCRSP, nous développons 7 contraintes, 3 contraintes dures et 4 contraintes souples. Étant donné qu'OptaPlanner maximise le score résultant de toutes les contraintes énumérées dans le problème à résoudre, chaque contrainte non satisfaite est affectée d'une valeur négative avec la syntaxe « *scoreHolder.addHardConstraintMatch(kcontext, -1)* » pour les contraintes dures (la valeur est souvent égale à -1), « *scoreHolder.addSoftConstraintMatch (kcontext, -score value)* » pour les contraintes souples.

La contrainte [1] est une contrainte dure qui permet de garantir que si les intervenants arrivent en avance au domicile d'un bénéficiaire, ils doivent attendre l'heure de début au plus tôt de la fenêtre de temps de début d'intervention demandée par le bénéficiaire.

Contrainte 1 : le début d'une intervention avant l'heure de début au plus tôt de la fenêtre de temps bénéficiaire est interdit

```
When
        C (
        |   e_i > debut_i )
        )
Then
        scoreHolder.addHardConstraintMatch(kcontext, -1) ;
```

La contrainte [2] est une contrainte dure pour empêcher le chevauchement temporel des interventions multiples demandées par un bénéficiaire.

Contrainte 2 : le chevauchement entre les interventions multiples demandées par un bénéficiaire est interdit

```
When
        $Intervention_1 : C (
        |   debut_i ! = null,
        |   $Début_1 : debut_i ,
        |   $Durée_ 1 : duree_i*
        )
        $Intervention_2 : C (
        |   debut_i ! = null,
        |   $Début_2 : debut_i,
        |   $Durée_2 : duree_i*
        )
Then
        If ($Intervention_1 != $Intervention_2) && $Début_2  < $Début_1 +$Durée_ 1)
        |   scoreHolder.addHardConstraintMatch(kcontext, -1) ;
        End
```

La contrainte [3] est une contrainte dure pour s'assurer qu'il n'existe pas de relation d'incompatibilité entre le bénéficiaire p et l'intervenant k,.

Contrainte 3 : l'affectation d'un intervenant en cas d'incompatibilité avec un bénéficiaire est interdite

```
When
        $Intervenant : K (
        |   $bénéficiaires_incompatibles : imcomp_k
        )
        $Bénéficiaire : P (
        |   $Intervenant_ incompatibles : imcomp_p
        )
        $Intervention : C (
        |   affec_i == $Intervenant,
        |   benef_i memberOf imcomp_k,
        |   affec_i memberOf imcomp_p
        )
Then
```

```
14          scoreHolder.addHardConstraintMatch(kcontext, -1) ;
```

La contrainte [4] est une contrainte souple dont l'objectif est d'équilibrer les difficultés de tournées des intervenants. Pour ce faire, nous appliquons une stratégie de score consistant à maximiser l'écart négatif entre la difficulté totale de la tournée de chaque intervenant (*$Difficulté_tournée*) et la difficulté moyenne de chaque intervenant (*$Difficulté_totale / $Nombre_Intervenant*).

Contrainte 4 : équilibrage au mieux des difficultés de tournées des intervenants

```
1   When
2           $Intervenant :K (
3           |   $id : id_k )
4           )
5           accumulate (
6           |   $Intervention : C (
7           |   |   affec_i == $Intervenant,
8           |   |   $Difficulté : dif_i
9           |   ) ,
10          |   $Difficulté_tournée : sum($difficulté) // somme des indicateurs de difficulté par tournée
11          )
12          $Nombre_Intervenant : Number()  from accumulate (
13          |   $Intervenant : K ( ),
14          |   Count ($Intervenant) // calculer le nombre d'intervenants
15          )
16          accumulate (
17          |   $Intervention : C (
18          |   |   $Difficulté : dif_i
19          |   ),
20          |   $Difficulté_totale : sum($Difficulté) // somme des indicateurs de difficulté totaux
21          )
22  Then
23          scoreHolder.addSoftConstraintMatch (kcontext,
24          - Math.abs($Difficulté_tournée - $Difficulté_totale / Nombre_Intervenant )
```

À l'identique des contraintes (20) à (22) dans la méthode exacte, les contraintes [5] à [7] sont ici des contraintes souples représentant la satisfaction des contraintes temporelles avec les tolérances personnalisées selon les profils individuels des acteurs. La fonction de régression linéaire est encore appliquée, mais ici pour modéliser la relation entre le score et le temps de dépassement (cf. la Figure 49). Notons que si le temps de dépassement de ces 3 contraintes est inférieur à la tolérance concernée ($\alpha_p, \beta_p, \gamma_k$), la pente de la fonction linéaire étant égale à $(100 - \alpha_p^* \vee \beta_p^* \vee \gamma_k^*)$, plus l'acteur est sensible au dépassement du temps souhaité (plus la valeur de $\alpha_p^* \vee \beta_p^* \vee \gamma_k^*$ est faible), plus la pente de la fonction est élevée. Pour tenir compte de l'insatisfaction des acteurs quant aux résultats de la planification sur les 3 contraintes temporelles en cas de dépassement de leurs tolérances, la pente de la deuxième partie de cette fonction linéaire est 10 fois plus grande que la première partie. Le fait de contrôler la pente renforce également la fonction linéaire appliquée avec la méthode exacte (cf. la Figure 42), ce qui permet d'éviter d'attribuer des valeurs aux variables de décision relatives à ces 3 contraintes au-delà de la tolérance de l'utilisateur lors de la recherche de la solution optimale globale.

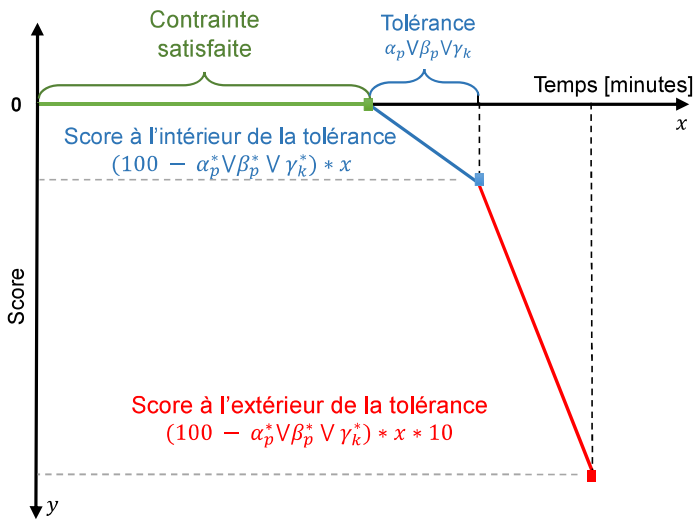

Figure 49 : Fonction de régression linéaire pour modéliser la relation entre le score et le temps de dépassement des 3 contraintes temporelles

Les contraintes [5] à [7] calculent donc respectivement le score par rapport au respect de la fenêtre de début d'une intervention demandée (contrainte 5), au respect du délai inter-intervention demandé par un bénéficiaire (contrainte 6) et au respect de la charge de travail maximale d'un intervenant (contrainte 7).

Contrainte 5 : Respect au mieux des fenêtres de temps demandées par les bénéficiaires

```
1   When
2           $Intervention : C (
3           |   l_i < debut_i,
4           |   $Fenêtre_temps_tard : l_i,
5           |   $Début : debut_i
6           )
7   Then
8           long Temps_dépassé = $Début - $Fenêtre_temps_tard
9           long Score = 100 - α*_p // α*_p est obtenu par appeler la méthode getOvertimeToleranceRate()
10          If (Temps_dépassé ≤ α_p )// α_p est obtenu par appeler la méthode getOvertimeTolerance()
11          |   scoreHolder.addSoftConstraintMatch (kcontext, - Score*Temps_dépassé)
12          Else
13          |   scoreHolder.addSoftConstraintMatch (kcontext, - Score*Temps_dépassé*10)
14          End
```

Contrainte 6 : Respect au mieux des délais inter-intervention demandés par les bénéficiaires

```
1   When
2           $Intervention_1 : C (
3           |   debut_i ! = null,
4           |   $bénéficiaire_1 : benef_i ,
5           |   $Début_1 : debut_i ,
6           |   $Durée_ 1 : duree_i*
7           )
8           $Intervention_2 : C (
9           |   debut_i ! = null,
10          |   debut_i  > $Début_1 // assurer $Intervention_2 est demandée après $Intervention_1
11          |   $bénéficiaire_2 : benef_i ,
12          |   $Début_2 : debut_i
13          )
14  Then
```

```
15          long Début_2_désiré = $Début_1 + $Durée_ 1 + δ_p ;
16          // Début de $Intervention_2 au plus tôt. δ_p est obtenu par appeler la méthode getInteroperation()
17          If ($Début_2 < Début_2_désiré) // surcharge d'un intervenant
18          |   long Non_respect_Interop = Début_2_désiré - $Début_2 ;
19          |   long Score = 100 - β*_p // β*_p est obtenu par appeler la méthode getInteropToleranceRate()
20          |   If (Non_respect_Interop ≤ β_p ) // γ_k est obtenu par appeler la méthode getInteropTolerance()
21          |   |   scoreHolder.addSoftConstraintMatch (kcontext, - Score∗ Non_respect_Interop)
22          |   Else
23          |   |   scoreHolder.addSoftConstraintMatch (kcontext, - Score∗ Non_respect_Interop ∗10)
24          |   End
25          End
```

Contrainte 7 : Respect au mieux de la charge de travail maximale d'un intervenant effectuant une tournée

```
1   When
2           $Intervenant :K (
3           |   $Charge_max : ξ_k
4           )
5           accumulate (
6           |   $Intervention_O : C (
7           |   |   affec_i == $Intervenant,
8           |   |   $Durée : duree_i*
9           |   ),
10          |   $Durée_totale : sum($Durée) // temps d'opération totaux par intervenant
11          )
12          accumulate (
13          |   $Intervention_T : C (
14          |   |   affec_i == $Intervenant,
15          |   |   arret_i != null,
16          |   |   i* ← arret_i.getBénéficiare().getLocalisation(),
17          |   |   j* ← this.getBénéficiare().getLocalisation(),
18          |   |   $Temps_trajet : temp_i*j*
19          |   )
20          |   $Temps_trajet_total : sum($temps_trajet ) // temps de trajet (sauf le retour à la structure)
21          )
22          $Intervention_dernière : C (
23          |   affec_i == $Intervenant,
24          |   attret_i != null && prochaC_i ==null,
25          |   i* ← arret_i.getIntervenant().getStructureDePAD().getLocalisation(),
26          |   j* ← this.getBénéficiare().getLocalisation(),
27          |   $Temps_retour : temp_i*j* // temps de trajet de retour à la structure
28          )
29  Then
30          long Temps_travail = $Durée_totale +$Temps_trajet_total +$Temps_retour ;
31          If (Temps_travail > $Charge_max ) // surcharge d'un intervenant
32          |   long Temps_dépassé = Temps_travail - $Charge_max ;
33          |   long Score = 100 - γ*_k // γ*_k est obtenu par appeler la méthode getOvertimeWorkloadToleranceRate()
34          |   If (Temps_dépassé ≤ γ_k ) // γ_k est obtenu par appeler la méthode getOvertimeWorkloadTolerance()
35          |   |   scoreHolder.addSoftConstraintMatch (kcontext, - Score∗Temps_dépassé)
36          |   Else
37          |   |   scoreHolder.addSoftConstraintMatch (kcontext, - Score∗Temps_dépassé∗10)
38          |   End
39          End
```

4.2.4. Résolution du problème

OptaPlanner permet de définir la condition d'arrêt des calculs, de choisir l'heuristique de construction de la solution initiale et l'algorithme d'optimisation (à base de métaheuristiques). Et cela, afin de trouver la meilleure configuration algorithmique (combinaison entre les