Hybridation opérationnelle des logiques OWL2 et ASP pour améliorer l'expressivité déclarative

X Goblet, C Rey, A Collange

Hybridation opérationnelle des logiques OWL2 et ASP pour améliorer l'expressivité déclarative

X. Goblet[1], C. Rey[2], A. Collange[1,2]

[1] Jeolis Solutions, Clermont-Ferrand, France
[2] LIMOS, Université Clermont Auvergne, Aubière, France

xavier.goblet@lojelis.com, christophe.rey@uca.fr, adrien.collange@uca.fr

Résumé

Dans le cadre d'un outil de recommandation d'activités dédié à l'éducation thérapeutique du patient, nous avons proposé la solution ORALOoS combinant OWL2 + SWRL, augmentée du module Python Owlready. L'usage de code impératif permet de pallier certaines limitations d'expressivité, comme l'absence de négation dans les règles SWRL. Cependant, il contrebalance fortement le caractère déclaratif de l'approche permis par OWL2. Tout en préservant cette caractéristique, nous présentons une solution pour remplacer règles SWRL et code impératif par un programme en Answer Set Programming. Cette idée n'est pas nouvelle mais peu de travaux ont abouti à des solutions réellement opérationnelles dans un contexte industriel. Notre proposition est abordée en deux étapes : 1) utilisation du solveur Hexlite qui donne de bons résultats en termes d'expressivité déclarative mais reste difficile à mettre en œuvre, et 2) développement d'une nouvelle librairie opérationnelle, hybridant OWL2 et ASP avec des fonctionnalités similaires à Hexlite mais d'un usage facilité.

Mots-clés

Hybridation de langages logiques, OWL2, ASP, Module Owlready, Système Clingo.

Abstract

As part of an activity recommendation tool dedicated to therapeutic patient education, we proposed the ORALOoS solution combining OWL2 + SWRL, augmented with the Python Owlready module. The use of the imperative code makes it possible to overcome certain limitations of expressiveness, such as the absence of negation in the SWRL rules. However, it strongly counterbalances the declarative character of the approach allowed by OWL2. While preserving this characteristic, we present a solution to replace the SWRL rules and the imperative code by a program in Answer Set Programming. This idea is not new, but few studies have led to truly operational solutions in an industrial context. Our proposal is suggested in two steps : 1) use of the Hexlite solver which gives good results in terms of declarative expressiveness but remains difficult to operate with, and 2) development of a new operational library, hybridizing OWL2 and ASP with features similar to Hexlite but easy to use.

Keywords

Hybridization of logical languages, OWL2, ASP, Owlready Framework, Clingo System.

1 Introduction

Dans le cadre d'un outil de recommandation d'activités dédié à l'éducation thérapeutique du patient, nous avons proposé la solution ORALOoS combinant une ontologie OWL2 et des règles SWRL, augmentée du module Python Owlready [13] pour pallier les limitations comme l'absence de négation dans SWRL et de traitements dynamiques en OWL2. Dans l'optique d'apporter une solution complètement déclarative, nous proposons de combiner (hybrider) un langage logique avancé (ASP) avec les logiques de description (OWL2). Les premiers travaux dans ce domaine remontent à une trentaine d'années. De nombreux résultats théoriques ont été obtenus et quelques solutions opérationnelles ont vu le jour. En particulier, Hexlite a été proposé récemment et étendu pour offrir une interaction bidirectionnelle entre un programme ASP étendu et une ontologie OWL2 via l'API Java OWLAPI [23]. Nous avons donc, dans un premier temps, testé Hexlite pour notre contexte applicatif. Si les résultats sont bons (remplacement de SWRL + code impératif), des difficultés de mise en œuvre d'Hexlite apparaissent. Ainsi, dans un second temps, nous proposons une nouvelle librairie hybridant ASP et OWL2, similaire à OWLAPI Hexlite mais exclusivement à partir des fonctionnalités natives du système ASP Clingo et du module Python Owlready pour l'interface OWL2.

Le papier est organisé comme suit : la section 2 présente notre contexte applicatif ORALOoS, ainsi qu'un état de l'art des travaux combinant la programmation logique avec les logiques de description. La section 3 présente notre utilisation de la proposition récente HexLite pour ORALOoS et les limites de cette approche. La section 4 présente notre librairie alternative : Exialis, son application spécifique pour ORALOoS ainsi que sa généralisation. La section 5 conclut et évoque les perspectives.

2 Travaux antérieurs

2.1 Le contexte applicatif ORALOoS

Dans le cadre d'applications de suivi de patients à distance, nous posons la question "Comment motiver et engager des patients envers une application numérique de suivi médical, qui plus est sur un temps long ?". C'est particulièrement vrai dans les protocoles d'Education Thérapeuthique du Patient (ETP), mais aussi en suivi de maladies chroniques, en pré ou post chirurgie ambulatoire où la motivation du patient à utiliser très fréquemment l'application reste primordiale. Encore faut-il que les interactions proposées soient engageantes ; un simple échange de SMS, une lecture de pages web, etc. ne sont pas suffisantes. Nous avons donc proposé un outil de recommandation d'activités (ORALOoS) [13] permettant de :

- Gamifier l'ETP en proposant un contenu ludique comme des mini jeux (quizz, défis personnels, etc.).
- Personnaliser le parcours de chaque patient en se basant sur ses succès ou échecs rencontrés lors des activités et lui proposer une progression adaptée.

Les sections suivantes 2.1.1 et 2.1.2 reprennent les éléments de [13] pour en expliciter les limites.

2.1.1 Éléments fonctionnels

Pour maintenir la motivation d'un utilisateur, le premier principe fondateur d'ORALOoS, issu des sciences psychologiques (motivation intrinsèque, théorie du flow, zone proximale de développement...) est de proposer un large espace d'activités ludiques (appelées défis dans la suite). Chaque défi concerne un domaine de compétences à déployer par l'utilisateur lorsqu'il l'exécute. Les experts du domaine définissent aussi pour chaque défi un niveau de difficulté [2]. Enfin, l'activation d'un défi peut être conditionnée par un contexte spécifique à l'utilisateur (par exemple, c'est un enfant) ou à son parcours (il a lu une fiche conseil), etc.

Il semble aussi évident que, pour chaque utilisateur, l'exploration de cette espace d'activités ne peut se faire de façon aléatoire et il doit être guidé selon les principes :

- d'une recommandation d'un ou plusieurs prochains défis en tenant compte de son retour immédiat (feedback),
- d'une pédagogie béhavioriste (conditionnement par renforcement) : en cas de succès pour l'activité courante, lui proposer une activité de difficulté un peu supérieure et inversement en cas d'échec [12].

2.1.2 Éléments technologiques

Modélisation statique OWL2. La figure 1 donne un aperçu graphique (pseudo diagramme de classes UML) d'une ontologie applicative conçue avec ORALOoS dédiée à la prévention de l'obésité infantile. Nous retrouvons les principaux concepts définis dans la section précédente sous forme de classes, d'attributs et de relations :

- La classe Challenge conceptualise les défis avec ses attributs hasLevel, hasDomain, ... qui modélise des propriétés de donnée (DP) OWL. Les conditions d'activation d'un défi sont définies comme une propriété objet (OP) OWL par la relation hasCondition* [1].
- Un utilisateur est défini par la classe Person qui possède une propriété DP hasAge. Lorsque que l'on asserte (instancie) un individu dans l'ontologie avec son âge, un raisonnement terminologique permet de le classifier en Enfant ou Adulte. Le concept Person est aussi en relation avec les défis par les deux propriétés non fonctionnelles fails* et succeeds* qui mémorisent alors les défis déjà rencontrés par l'utilisateur et que l'on interdit de rejouer (relation forbids*).
- Le concept Condition est spécialisé en deux sous-classes : ClassCond qui exprime une caractéristique intrinsèque d'un individu de l'ontologie ; par exemple : EnfantCond ou AdulteCond. RelationCond est la définition d'une condition qui exprime une relation entre deux concepts de l'ontologie. Par exemple, un défi peut être conditionné (SuccessCond) à la réussite d'un ou plusieurs autres défis pré-requis.
- Le retour utilisateur est défini par la classe FeedBack et est spécifique à une personne par la relation fonctionnelle concernsPerson et un défi courant par concernsChallenge. Faisant suite à l'évaluation, tout feedback porte aussi les prochains défis candidats dans la relation usable*, eux-mêmes partitionnés dans des relations non fonctionnelles xxxLevel. En appliquant un raisonnement OWL, en fonction de l'attribut DP hasSuccess, le feedback est spécialisé en PositiveFB ou NegativeFB.

Modélisation dynamique SWRL. La figure 2 présente les règles logiques SWRL pour la mise en oeuvre d'une pédagogie béhavioriste définie en section 2.1.1 ; avec l'ajout d'une règle palier R3 qui propose des défis de difficulté similaire au défi évalué (limite l'effet "montagnes russes" si on applique successivement R1 ou R2).

SWRL est aussi une proposition [2] W3C pour spécifier des règles logiques dans les ontologies en introduisant la notion de variables ?x dans les prédicats. Ces prédicats viennent d'une ontologie ou bien sont prédéfinis (préfixés par swrlb) et sont enchaînés par un ET logique uniquement. Les inconvénients principaux de SWRL sont l'absence de négation explicite dans les règles, ainsi que l'impossibilité d'utiliser des prédicats plus complexes (termes fonctionnels, par exemple) ; ce qui nécessite d'implémenter le prédicat usable() avec un code impératif comme le montre la section suivante.

Owlready2. C'est un module Python permettant la manipulation (création, lecture, modification et suppression) d'on-

1. Le caractère "*" spécifie que la relation OP est non fonctionnelle ; i.e. qu'un même individu/instance Challenge peut être associé plusieurs individus Condition. Une propriété OP ou DP fonctionnelle OWL équivaut à une relation 1..1 en UML.

2. A la différence d'OWL et OWL2, SWRL n'a jamais été promu au rang de standard W3C pour le Web Sémantique. Par contre, il reste utilisable dans des raisonneurs comme HermiT ou Pellet.

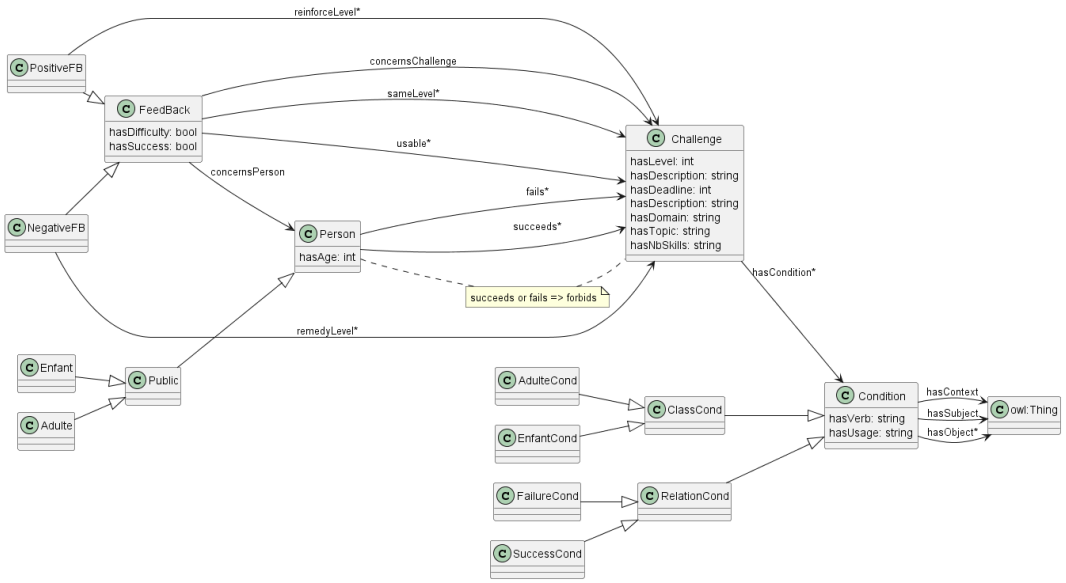

FIGURE 1 – Extrait de l'ontologie applicative ETP Proximité Obésité.

```
R1: Feedback(?fb) ^ concernsChallenge(?fb, ?ch) ^ hasLevel(?ch, ?lv) ^ usable(?fb, ?nch)
    ^ hasLevel(?nch, ?nlv) ^ swrlb:greaterThan(?nlv, ?lv) ^ hasSuccess(?fb, true) -> reinforceLevel(?fb, ?nch)

R2: Feedback(?fb) ^ concernsChallenge(?fb, ?ch) ^ hasLevel(?ch, ?lv) ^ usable(?fb, ?nch)
    ^ hasLevel(?nch, ?nlv) ^ swrlb:lessThan(?nlv, ?lv) ^ hasSuccess(?fb, false) -> remedyLevel(?fb, ?nch)

R3: Feedback(?fb) ^ concernsChallenge(?fb, ?ch) ^ hasLevel(?ch, ?lv) ^ usable(?fb, ?nch)
    ^ hasLevel(?nch, ?lv) -> sameLevel(?fb, ?nch)
```

FIGURE 2 – Règles SWRL.

tologies au format OWL2 [15]. A la différence d'API en Java comme OWLAPI, Owlready réifie les éléments d'une ontologie sous la forme d'objets, de classes et de méthodes de la programmation orientée objet, ce qui constitue une approche simple et puissante. Ce module inclut nativement une base de données graphe (appelée quadstore) stockant des triplets RDF associés à des identifiants d'ontologies, ainsi que les raisonneurs HermiT et Pellet permettant de faire des raisonnements par classification (terminologique) et aussi SWRL. Avec Owlready, les classes OWL sont des classes Python que l'on peut étendre en définissant des méthodes objet associées. C'est ce code impératif qui nous permet de pallier les limitations d'OWL2 et SWRL dans le traitement d'un feedback et des conditions

Avec Owlready, les classes OWL sont des classes objet Python commme les autres ; cela permet tout simplement d'étendre les concepts ontologiques en y incluant des méthodes. Pour ORALOoS, nous l'utilisons pour compléter les classes OWL2 Feedback, ClassCond et RelationCond avec des méthodes preFilter() et execute() respectivement.

— Feedback.prefilter() : les paramètres d'entrée de cette méthode sont l'utilisateur courant et le défi courant portés par les relations "concerns" de l'entité FeedBack. La 1ère étape est de chercher dans l'ontologie tous les défis de même domaine que le défi courant. Ensuite, on élimine de cet ensemble le défi courant lui-même et les défis déjà joués par l'utilisateur en utilisant sa relation forbids. Enfin, parmi tous ces défis potentiellement candidats, on garde uniquement les défis dont les conditions sont vérifiées pour ajouter les relations usable au feedback. Après l'exécution de cette méthode, il suffit de lancer un raisonnement Owlready pour déclencher les règles SWRL qui calculent les prochains défis recommandables définis ontologiquement par les OP reinforceLevel, remedyLevel et sameLevel et classer le feedback en FB+ ou FB-.
— ClassCond.execute() : Cette méthode utilise les définitions ontologiques hasSubject, hasVerb et hasObject de la classe Condition (cf. 1. Avec comme spécificité qu'une ClassCond a toujours comme verbe : "is_a", on vérifie que le sujet de la condition est bien du type/classe défini dans l'objet.
— RelationCond.execute() : vérifie dans l'ontologie qu'il existe bien la relation (propriété objet) spécifiée par hasVerb entre les deux individus hasSubject et hasObject de la condition.

2.2 Combiner les ontologies avec la programmation logique et réciproquement

Au sein du domaine de la représentation des connaissances et des raisonnements (KRR), la question de l'intégration de

la programmation logique (PL) avec les logiques de descriptions (LD) est étudiée depuis plus de 30 ans. L'objectif est de profiter des atouts des deux approches, et de pallier leurs inconvénients respectifs, tout en conservant la décidabilité et l'efficacité des raisonnements associés. Le tableau 1 présente un résumé des avantages et inconvénients de la LP et des LD.

Dans cette problématique, on peut classer les approches existantes en deux catégories : celles proposant une intégration de type fusion, qui consistent à n'exécuter qu'un seul raisonnement sur les connaissances modélisées sous forme de règles et d'axiomes ontologiques, et celles proposant une intégration de type API, qui consistent à faire dialoguer deux raisonneurs (un pour les règles et l'autre pour les axiomes ontologiques) via des interfaces fonctionnant comme des API.

Hybridation par fusion. Ces approches "sur étagère" permettent à l'utilisateur (ingénieur des connaissances) l'usage d'un seul raisonnement ; ce qui a comme avantage de lui simplifier le travail en se concentrant sur la modélisation des connaissances et n'a très peu, voire aucun code annexe supplémentaire à créer pour exécuter le raisonnement. La contrepartie est que les systèmes associés sont moins flexibles et qu'il est souvent difficile voire impossible de les étendre afin de construire un raisonneur personnalisé. Par ailleurs, comme l'explique bien [19], l'intégration de type fusion est un vrai défi dès lors que l'on veut assurer la décidabilité du raisonnement et son efficacité, et que l'on veut définir une sémantique rigoureuse qui ne soit pas limitante par rapport aux sémantiques respectives des règles et des axiomes ontologiques, c'est-à-dire qui permette un enrichissement réciproque des connaissances et qui gère de manière claire l'interaction entre les hypothèses du monde ouvert et du monde fermé (qui sont respectivement les hypothèses des LD et de la PL).

Un des premiers travaux d'intégration de type fusion est décrit dans [16], où les auteurs proposent de combiner des clauses de Horn sans fonction avec des ontologies exprimées avec la LD $\mathcal{ALCNR}$. Le formalisme obtenu, appelé CARIN, est utilisé pour permettre la médiation entre des requêtes d'utilisateurs et des sources de données hétérogènes. Le raisonnement associé, appelé "expansion de requêtes", utilise conjointement les règles et axiomes ontologiques pour réécrire la requête donnée en entrée en requêtes exécutables sur les sources. Cependant, les auteurs montrent que même avec deux formalismes séparément décidables, la fusion peut être indécidable. Plus récemment, [19] propose le formalisme MKNF$^+$, basé sur la logique de la connaissance minimale et la négation par échec fini (logic of minimal knowledge and negation as failure). Les auteurs proposent des contraintes nécessaires à la décidabilité, des algorithmes de raisonnement pour vérifier si un atome clos est vrai dans tous les modèles de la base de connaissance, et des bornes pour la complexité de ce raisonnement. Ils montrent comment MKNF$^+$ permet de raisonner avec des axiomes ontologiques et des programmes ASP, en permettant de choisir de raisonner en monde ouvert ou fermé pour chaque prédicat utilisé. Au niveau théorique, ce travail apparaît comme un cadre commun possible pour d'autres approches comme $\mathcal{DL}$+***log*** [20], disjunctive dl-programs [18], dl-programs [7], ou encore pour des approches basées sur différentes logiques du premier ordre, comme la logique auto-épistémique [5], un langage de requête du premier ordre orienté logiques de description [3], et la logique des défauts [1]. Tous les détails sont donnés dans [19]. On peut enfin citer le système DLog [17] qui est un raisonneur pour les LD basé sur l'inférence prolog. Cela permet un traitement bien plus efficace des raisonnements assertionnels impliquant de très grands ensembles de données.

Hybridation par API. Les approches d'intégration de type API sont des approches plus personnalisables. Le principe est que l'utilisateur modélise son problème avec un langage de PL et que sa résolution utilise en plus une ontologie du domaine en LD pour enrichir l'inférence. La personnalisation tient en ce que l'utilisateur a une certaine latitude pour choisir le raisonneur LD (et donc le raisonnement et la LD sous-jacents), alors que le raisonneur PL est le programme principal. Les deux raisonneurs communiquent en général dans les deux sens via une interface qui s'utilise comme une API. Les raisonnements sur les règles de PL et sur les connaissances ontologiques, dites "externes", peuvent être entrelacés, mais les raisonnements restent séparés. L'utilisateur peut avoir à programmer lui-même quelques scripts permettant la communication entre les raisonneurs, mais cela ne requiert pas d'être spécialiste des langages de KRR sous-jacents.

Ces dernières années, les approches de type API ont principalement utilisé ASP [3] en tant que formalisme de PL. En particulier, les systèmes DLVHEX et HEXLite [6–8, 21, 22] implémentent le langage HEX [4], qui augmente les programmes ASP avec des atomes externes [5] afin de les enrichir avec plusieurs types d'informations externes. Ces dernières peuvent être des informations stockées (comme des données de sources externes ou des connaissances stockées dans des ontologies), mais aussi des informations calculées (comme des données issues de requêtes et des connaissances issues de raisonnements). Par ailleurs ces atomes externes permettent aussi l'envoi de données aux sources externes, leur permettant d'effectuer leurs calculs. Ainsi, le

3. Answer Set Programming est un langage logique complet permettant un raisonnement non monotone, l'usage de négations (par défaut et classique/forte), conjonctions, disjonctions entre clauses et l'ajout de contraintes. Le principe de raisonnement standard ASP se déroule toujours en deux étapes : 1) ground qui consiste à instancier les variables et 2) solve qui résout ce programme transformé en propositions par un algorithme SAT. Les solveurs ASP à l'état de l'art académique et industriel sont les systèmes Clingo (https ://potassco.org/) et DLV (https ://www.dlvsystem.it/dlvsite/).

4. Higher-order logic with EXternal atoms. L'implémentation de référence est le solveur dlvhex dont le langage principal est C++ (http ://www.kr.tuwien.ac.at/research/systems/dlvhex/). Hexlite est une implémentation légère en Python [22] d'un fragment de HEX et utilise Clingo comme backend ASP.

5. L'ajout d'atomes externes nécessitent toujours une extension syntaxique + sémantique d'ASP, ainsi qu'une couche additionnelle de préprocessing pour insérer ces atomes externes dans les étapes ground+solve.

Programmation logique (programmes faits de règles prolog, ASP, ...)	**Logiques de description** (ontologies faites d'axiomes ontologiques exprimés en OWL2)
+ Pratique et populaire pour modéliser les aspects dynamiques d'un système	+ Pratiques et populaires pour modéliser et raisonner sur la connaissance statique d'un domaine
+ Inférence non monotone	+ Langages standardisés par le W3C
+ Implémentations matures et efficaces (XSB prolog, SWI prolog, ...) qui sont des langages Türing complete avec de nombreuses librairies	+ De nombreux résultats de complexité
	+ Des outils disponibles : Protégé pour la modélisation et des raisonneurs puissants (HermiT, Pellet, ...)
+ Accès direct à la programmation par contrainte (CLP, ASP, CHR,...) pour les problèmes combinatoires	+ Modélisation et raisonnement sur deux niveaux : TBox (axiomes terminologiques, i.e. les concepts) et ABox (axiomes assertionnels, i.e. les données)
+ Raisonnement en monde fermé : liens naturels avec les bases de données	+ Intégration facile avec des bases de données graphes
- Par défaut, pas de raisonnement terminologique, focus sur le problème de répondre à des requêtes	- OWL2 nécessite une certaine expertise
	- Raisonnement en monde ouvert pas toujours adapté
- Raisonnement en monde fermé : pas de représentation de connaissances incomplètes et donc difficulté à raisonner sur des domaines infinis	- Des manques au niveau expressivité (restriction à des prédicats unaires et binaires, pas de réelle jointure)
	- Complexité des raisonnements qui augmente vite avec l'expressivité de la logique choisie
- Pas toujours 100% déclaratif	
- Attention à la décidabilité en présence de fonctions	- Problème possible de passage à l'échelle si ABox grande
	- Impossibilité de modéliser des exceptions (axiomes non toujours vrais)

TABLE 1 – Avantages (+) et inconvénients (-) de la programmation logique et des logiques de description.

moteur ASP interne résout un programme Hex en pilotant les appels au raisonneur utilisant l'ontologie, ainsi que les transferts de connaissances du programme à l'ontologie et vice versa. Il est important de remarquer que DLVHEX et HEXLite sont faits pour qu'un développeur non spécialiste d'ASP puisse facilement relier un programme Hex avec une ontologie. Le système Clingo [10] permet aussi l'importation de faits provenant de prédicats définis par des utilisateurs, comme les concepts d'une ontologie. Cela est permis par la présence de fonctions personnalisables dans le grounder Gringo de Clingo, implémentées en Lua ou Python. De plus, depuis sa version 5, Clingo permet d'intégrer des théories personnalisées dans le processus de résolution ASP [9]. Cet enrichissement de la résolution de problèmes ASP revient à une intégration à un bas niveau de capacités de raisonnement sur des théories et des applications utilisateurs. Le raisonnement ASP de base via les étapes de grounding et solving est impacté de manière importante. Pour certaines LD et sous certaines hypothèses, cette fonctionnalité peut être utilisée pour intégrer le raisonnement ontologique au sein de la résolution ASP. Avec cette fonctionnalité, Clingo se rapproche donc d'un système proposant une intégration de type fusion. Cependant, la résolution de théorie de Clingo 5 nécessite de bien connaître les différentes étapes de la résolution ASP, et est destinée principalement aux utilisateurs experts qui souhaitent développer leur propre raisonneur ASP personnalisé plutôt que simplement enrichir leurs problèmes ASP avec des sources externes.

Bien que les approches d'intégration de type fusion aient abouti à de nombreux résultats théoriques, les systèmes basés sur ces dernières sont peu nombreux et n'ont pas (encore) atteint la popularité des raisonneurs LD ou PL. Une des explications possibles tient sans doute en la difficulté à développer un tel système. Par ailleurs, les performances des raisonneurs existants LD et PL rendent très attractives les approches d'intégration de type API. Par leurs résultats théoriques importants, les approches de type fusion tendent à définir de nouveaux langages de représentation de connaissances et de raisonnement, alors que les approches de type API permettent plus naturellement l'extension de langages existants. Néanmoins, au-delà de leurs spécificités respectives, les approches de type fusion et API s'influencent mutuellement pour faire avancer le domaine vers des langages et systèmes plus expressifs et performants.

3 Utilisation de l'extension OWLAPI d'HexLite

Pour proposer une solution pleinement déclarative à ORALOoS, nous présentons ici un retour d'expérimentation avec le système récent HexLite [23]. Ce dernier propose une interface bidirectionnelle entre une ontologie OWL et un programme HEX (extension ASP). OWLAPI est une interface Java de programmation d'ontologies ; celle-ci est enveloppée dans un plugin écrit en Java pour Hexlite et qui utilise comme backend ASP le système Clingo écrit en Python.

3.1 Les principes fonctionnels

Pour des raisons de concision, nous limitons notre présentation aux fonctionnalités et renvoyons le lecteur à l'article de P. Schüller pour les définitions formelles.

Les atomes ASP de modifications d'une ontologie.

- Insertion d'assertions dans une ABox : addc(C,I), addop(OP, I1, I2), adddp(DP, I, D).
- Suppression d'assertions d'une ABox : delc(C,I), delop(OP, I1, I2), deldp(DP, I, D).

où C, OP et DP sont les classes OWL, les propriétés objet et les propriétés de données OWL respectivement. Les variables I représentent des individus/instances OWL et D une valeur de données (int, string, bool...).

Les modifications autorisées par cette extension HexLite ne concernent pas la création ou la suppression de nouveaux concepts (Classe ou Propriété) dans l'ontologie.

Les atomes externes. La syntaxe des atomes externes Hexlite est la suivante : &atom[in](out). Le contenu 'in' est une liste de paramètres d'entrée sous la forme de variables comme T, C, ... ou de littéraux représentant des constantes comme onto ou des prédicats comme delta. Le contenu 'out' est une liste de paramètres de sortie de l'atome externe. L'extension OWLAPI Hexlite définit plusieurs atomes externes pour interagir avec une ontologie, en écriture :

- &dlC[onto, delta ,T,C](I) : récupère les individus de la classe C.
- &dlOP[onto,delta ,T,OP](I1 ,I2) : récupère les individus I1 (Domain) et I2 (Range) de la propriété objet OP.
- &dlDP[onto,delta ,T,DP](I,D) : récupère l'individu I (Domain) et la valeur D (Range) de la propriété de donnée DP.
- &dlConsistent[onto,delta,T]() : vérifie la consistance de l'ontologie.

Les atomes qui interagissent en lecture seule sont :

- &dlCro[onto,C](I) : Requête les individus I de la classe C.
- &dlOPro[onto,OP](I1,I2) : requête les individus Domain (I1) et Range (I2) de la propriété objet DP.
- &dlDPro[onto,DP](I,D) : requête la propriété de donnée DP pour lire la valeur D pour l'individu Domain I.

Le paramètre onto est une constante référençant l'ontologie. Pour les atomes externes en écriture : delta est un prédicat contenant les atomes de modifications *add* et *del* cidessus, T représente un pas de temps ou une étape de raisonnement sous la forme d'un entier. De plus, il faut comprendre que les modifications sont effectivement propagées en amont au moment de l'interrogation de l'ontologie par un atome externe en écriture. Le type d'atome externe C, DP, OP et le temp T permettent de filtrer les bonnes informations dans le delta qui est définit en extension.
Ces atomes externes nécessitent une extension sémantique d'ASP et il faut aussi partitionner les atomes externes à interpréter au moment de l'étape ground de ceux à interpréter au moment du solve ASP. De notre compréhension des articles [23] et [22], la formalisation du partitionnement n'est pas définie de la même façon, ce qui introduit une difficulté dans l'usage d'HexLite.

3.2 Feedback ORALOoS en HexLite

Malgré des efforts importants, nous n'avons pas réussi à installer nativement l'extension OWLAPI d'Hexlite à cause d'un conflit de librairies Java utilisées par le plugin. Par ailleurs, il existe bien un packaging Docker mais ce dernier reste difficile à configurer pour notre problématique.
Nous avons donc décidé d'écrire un nouveau plugin en Python qui utilise le module Owlready2 pour s'interfacer avec une ontologie OWL en remplacement du plugin Java et de l'API OWLAPI. Par contre, ce nouveau plugin ne concerne uniquement que les atomes externes en lecture définis dans OWLAPI Hexlite, auxquels nous avons rajouté un atome externe d'accès au type/classe d'un individu de l'ontologie : &dlGetType[onto,I](C).

Encodage HexLite. La figure 3 présente le code déclaratif du feedback ORALOoS avec notre adaptation de l'extension OWLAPI HexLite. La structure du programme ASP est : (i) une phase d'initialisation d'atomes ASP à partir d'atomes externes HEX de lecture de l'ontologie (lignes 3 à 12) ; (ii) le calcul ASP des prédicats usable() en tenant compte du domaine du défi courant, l'usage de la négation par défaut (not) pour éliminer ce dernier et ne pas considérer les défis déjà joués (forbids) par l'utilisateur courant (lignes 14 à 17) ; (iii) la dernière étape du usable consiste à retenir parmi les défis candidats uniquement ceux dont les conditions d'activations sont validées (lignes 19 à 25) ; et (iv) la dernière partie du programme (lignes 27 à 35) concerne la transcription déclarative en atomes ASP des 3 règles pédagogiques, après avoir récupéré dans l'ontologie le niveau de difficulté du défi évalué ainsi que les niveaux des défis candidats usable().
Malgré une adaptation limitée [6] du plugin owlapi en Python de HexLite, notre programme logique déclaratif permet bien de remplacer le code Python + les règles SWRL présentés en section 2.1.

Intégration Hexlite. Après avoir comparé des exécutions indépendantes des programmes ORALOoS et HexLite sur les mêmes données d'entrée et obtenu les mêmes résultats, nous avons cherché à intégrer ces deux programmes. La documentation, plutôt lacunaire, du dépôt Git et les différents articles sur HexLite expliquent que les deux applications peuvent interagir uniquement à travers un échange par fichier : le passage des paramètres d'entrée (identifiant de l'individu OWL feedback) se fait par lecture d'un fichier Json par Hexlite, et les résultats du feedback HexLite (les atomes ASP reinforce_level, remedy_Level, same_Level) sont retournés à ORALOoS via un fichier Json.
Le synoptique d'intégration est le suivant. D'abord ORALOoS est responsable, via le module Owlready, de charger et de réifier en objet une ontologie applicative OWL2. A chaque demande de feedback, Oraloos crée un individu Feedback dans l'ontologie, lui associe les différentes propriétés, en particulier le défi et l'utilisateur concernés par ce retour. Oraloos doit le sauvegarder dans le fichier Owl, écrire le fichier Json de paramétrage du feedback, lancer le script HexLite de calcul. Ensuite Hexlite charge le fichier Owl de l'ontologie, via Owlready, le réifie dans un objet spécifique. Hexlite charge ensuite le programme ASP étendu et effectue son calcul en fonction du paramétrage feedback. Au final, il écrit les résultats dans un fichier Json connu d'ORALOoS. Enfin, ORALOoS doit se synchroniser sur ce fichier pour enregistrer les résultats dans l'ontologie, lancer un raisonnement OWL qui classifie le feedback en FB+ ou FB-. Ce processus nous semble difficile à synchroniser entre les différentes lectures/écritures des fichiers par les deux programmes.

6. Interface unidirectionnelle OWL –> ASP.

```
#const onto="in/oraloos-meta.json".
% === INIT from ontology ===
challenge(Challenge) :- &dlCro[onto, "Challenge"](Challenge).
domain(Challenge, Domain) :- &dlDPro[onto, "hasDomain"](Challenge, Domain).
forbids(Person, Challenge) :- &dlOPro[onto, "forbids"](Person, Challenge).

has_condition(Challenge, Object) :- &dlOPro[onto, "hasCondition"](Challenge, Condition),
    &dlDPro[onto, "hasVerb"](Condition, Verb), &dlOPro[onto, "hasObject"](Condition, Object),
    &dlOPro[onto, "hasSubject"](Condition, Subject), Verb = is_a.
has_condition(Challenge, Verb, Object) :- &dlOPro[onto, "hasCondition"](Challenge, Condition),
    &dlDPro[onto, "hasVerb"](Condition, Verb), &dlOPro[onto, "hasObject"](Condition, Object),
    &dlOPro[onto, "hasSubject"](Condition, Subject), Verb != is_a.
% === USABLE ===
usable_before(Challenge) :- challenge(Challenge), domain(Challenge, Domain), current_domain(Domain),
    not current_challenge(Challenge), current_user(User), not forbids(User, Challenge).
% Challenges without condition
usable(Challenge) :- usable_before(Challenge), not has_condition(Challenge, _), not has_condition(Challenge, _, _).

is_checked(Challenge, ClassCond) :- usable_before(Challenge), has_condition(Challenge, ClassCond), current_user(User),
    &dlGetType[onto, User](ClassCond).
is_checked(Challenge, Verb, RelationCond) :- usable_before(Challenge), has_condition(Challenge, Verb, RelationCond).
% Challenges with condition(s)
usable(Challenge) :- usable_before(Challenge), N={has_condition(Challenge, Condition)},
    M={is_checked(Challenge, Condition)}, N=M,  I={has_condition(Challenge, Verb, Condition)},
    J={is_checked(Challenge, Verb, Condition)}, I=J.
% === LEVEL MANAGEMENT ====
current_level(CurrentLevel) :- current_challenge(Challenge), &dlDPro[onto, "hasLevel"](Challenge, CurrentLevel).
usable_level(Challenge, UsableLevel) :- usable(Challenge), &dlDPro[onto, "hasLevel"](Challenge, UsableLevel).

reinforce_level(Challenge) :- usable_level(Challenge, UsableLevel), current_level(CurrentLevel),
    UsableLevel > CurrentLevel.
remedy_level(Challenge) :- usable_level(Challenge, UsableLevel), current_level(CurrentLevel),
    UsableLevel < CurrentLevel.
same_level(Challenge) :- usable_level(Challenge, UsableLevel), current_level(CurrentLevel),
    UsableLevel = CurrentLevel.
```

FIGURE 3 – Extrait du feedback en Hexlite.

3.3 Conclusions sur HexLite

Fonctionnellement, l'interface bidirectionnelle entre une ontolologie OWL et un programme ASP augmenté par Hexlite proposé dans [23] est très prometteur. Nous avons montré que cette approche permet bien de remplacer des régles SWRL + du code impératif et augmente sensiblement l'expressivité déclarative d'une application.

Cependant, la mise en œuvre de bout en bout de l'extension OWLAPI d'Hexlite reste très difficile :

- Installation native du package Hexlite + OWLAPI impossible à mener à son terme.
- Incompréhension sur comment sont partitionnés les atomes externes entre les étapes "ground" et "solve" de l'exécution ASP.
- Documentation à minima au travers d'exemples dans le dépot Git et les différents articles HexLite ; le code github ne semble plus maintenu.
- Problème d'intégration Hexlite avec une application tierce comme ORALOoS.

De plus, notre plugin d'interface en Python ne contient pas d'atome externe permettant de modifier l'ontologie à partir d'une solution ASP. Après plusieurs mois de travaux avec Hexlite et pour pallier les limitations ci-dessus, nous avons décidé de développer une nouvelle librairie combinant OWL2 et ASP fonctionnellement similaire à Hexlite.

4 Vers une hybridation OWL et ASP opérationnelle

Notre solution alternative prend la forme d'une librairie Python, nommée Exialis, utilisant Clingo comme backend ASP et Owlready2 pour s'interfacer avec des ontologies OWL2. L'objectif d'Exialis est d'avoir une interface bidirectionnelle entre une ontologie OWL2 et un programme ASP. Pour cela nous utilisons les fonctionnalités natives de Clingo : fonctions externes, Multi-Shot Solving (MSS) et Incremental Solving (IS). Nous utilisons cette librairie pour résoudre notre problème de feedback dans le contexte applicatif ORALOoS et des problèmes de type planification où les dimensions temporelle et incrémentale du raisonnement sont prégnantes.

4.1 Fonctionnalités natives Clingo

4.1.1 Fonctions externes

Définies dans [14], elles nous servent pour définir les opérations de lecture et d'écriture dans l'ontologie. Ces fonctions externes doivent être encapsulées dans un atome ASP name(@function(X, Y)) [7]. Les fonctions externes Exialis de lecture d'une ontologie sont les suivantes :

- @dlCro(C, (X,Y), onto) : requête les individus de la classe C.
- @dlPro(P, (X,Y), onto) : requête la propriété P (objet ou donnée) dans l'ontologie onto.
- @dlTro(I, (X,Y), onto) :requête le(s) classe(s) auquel appartient l'individu I.

Le deuxième paramètre des fonctions est un tuple qui permet de filtrer la requête ; il peut être vide. Les fonctions externes Exialis de modification d'une ontologie sont :

- @addC(C, I, onto) : écrit dans l'ontologie l'individu I de type C.
- @addOP(P, I1, I2, onto) : ajoute dans l'ontologie la propriété objet P avec l'individu I1 comme Domain et l'individu I2 comme Range.
- @addDP(P, I, V, onto) : ajoute dans l'ontologie la propriété de donnée P avec l'individu I comme Domain et la valeur V comme Range.
- @dlConsistent(onto) : effectue un raisonnement sur l'ontologie et permet de vérifier la consistance de

7. Dans un premier temps, les noms sont proches des atomes HexLite, mais nous allons les renommer pour assurer une homogénéité.

celle-ci.

Ces fonctions externes sont implémentées via un module Python clingo, cf. §4.2 dans [14]. A la différence des atomes externes d'HexLite, ces fonctions ne nécessitent pas une extension de la sémantique ASP.

4.1.2 Multi-shot solving

Rappelons que la résolution standard d'un problème en ASP se fait en une étape (single-shot) : Ground(P) –> solve(P'), où P est un programme ASP contenant des variables et P' est la transformation en logique propositionnelle. La résolution 'multi-shot' Clingo [11] permet de partitionner le programme ASP en différents sous-programmes qui peuvent être "Ground" séparément et s'intercaler dans différentes étapes de "Solve".

La déclaration d'un sous-programme Clingo dans l'encodage ASP se fait via la directive : #program n(p1 ,..., pk où n est le nom du sous-programme et pi les paramètres. Il existe un sous-programme par défaut avec le nom réservé et sans paramètre : #program base. Toutes les règles ASP non précédées d'une directive #program sont allouées à base. Le contrôle des étapes "Ground" et "Solve" est géré par l'API Clingo via une instance d'un objet Control avec lequel on peut gérer le chargement des fichiers ASP (ctl.load()), le "Ground" d'un sous-programme ASP : ctl.ground() et son "Solve" par : ctl.solve().

A noter que la portée d'un "ground" s'applique uniquement aux sous-programmes donnés en argument, tandis qu'un "solve" déclenche un raisonnement par rapport à tout ce qui a été "ground+solve" précédemment, cf. [14].

4.2 Feedback ORALOoS en Exialis

Encodage Exialis. Nous illustrons l'usage des fonctions externes et du "multi-shot solving" de Clingo en présentant l'encodage du Feedback ORALOoS :

- La figure 4 est le sous-programme, par défaut, base de Clingo. C'est une initialisation générale par lecture dans l'ontologie des propriétés (DP et OP) des conditions via la fonction externe dlPro().
- La deuxième partie de l'encodage ASP effectue des lectures de l'ontologie contextualisées par un individu Feedback spécifique passé en paramètre du sous-programme step ; lignes 7 à 15 de la figure 5. Soulignons dans la ligne 9, l'usage dans la fonction externe dlCro d'un tuple qui filtre parmi tous les défis de l'ontologie uniquement ceux qui sont du même domaine du défi courant. Les lignes 17 à 39 sont l'encodage ASP du calcul des prédicats usable() et du partitionnement selon le niveau de difficultés des défis candidats.
- Le figure 6 est le sous-programme d'écriture dans l'ontologie pour assigner les résultats à l'individu feedback.

Intégration Exialis. A la différence d'HexLite, Exialis est une librairie Python que l'on peut importer dans une application tierce en Python, ce qui est le cas d'ORALOoS. Le synoptique d'intégration est le suivant :

- ORALOoS est responsable, via le module Owlready, de charger et de réifier en objet une ontologie applicative OWL2.
- A chaque demande de feedback, ORALOoS créé un individu Feedback dans l'ontologie, lui associe les différentes propriétés, en particulier le défi et l'utilisateur concernés par ce retour. ORALOoS délègue ensuite le calcul du feedback à une instance d'Exialis qui est contextualisé/construit avec l'objet onto et l'individu feedbackID. L'objet ctl = Control() de l'API Clingo permet le raisonnement en mode "Multi-Shot Solving" après chargement du programme ASP ctl.load("feedback-exialis.lp") :
  - ctl.ground([("base", [])]) + ctl.solve()
  - ctl.ground([("step", [String(feedbackId)])]) + ctl.solve()
  - ctl.ground([("update", [String(feedbackId)])]) + ctl.solve()
- Une fois le retour synchrone de l'instance Exialis effectué, ORALOoS reprend la main en effectuant un raisonnement OWL (classification du feedback). Si les modifications ASP laissent l'ontologie consistante, ORALOoS peut la sauvegarder.

L'intégration et l'utilisation Exialis par ORALOoS est plus facile qu'avec le programme HexLite.

4.3 Généralisation d'Exialis

Le cas d'usage d'Exialis pour ORALOoS est plutôt spécifique : il n'y a pas de dimension temporelle ni d'instanciation multiple d'un contrôleur Clingo. Cela ne permet pas de l'utiliser sur des problèmes comme la planification de robots décrit dans [23] ou des systèmes de configuration de produits industriels [4]. La fonctionnalité IS proposée nativement par le système Clingo depuis sa version 4 [11] permet d'ajouter une dimension temporelle dans les étapes de résolution ASP au moyen de sous-programmes spécifiques. IS est donc une forme spéciale du MSS de Clingo et correspond au modèle déclaratif ASP suivant :

- Un programme base qui modélise les connaissances statiques du problème.
- Un programme step(t) qui capture les connaissances évoluant au cours d'un temps discret t (une étape de résolution).
- Un programme check(t) qui permet de vérifier l'atteinte de l'objectif final du problème à chaque étape de résolution.

Un exemple de ce modèle est l'encodage incrémental des tours de Hanoï de [14]. Inclure les fonctions externes Exialis dans ce modèle permet de mettre en oeuvre une interface bidirectionnelle et incrémentale combinant ASP et OWL2.

4.4 Conclusions sur Exialis

Les principaux objectifs d'Exialis sont atteints :

- Avoir une interface bidirectionnelle entre OWL2 et ASP pour traiter le feedback ORALOoS "à la Hexlite". L'usage du "multi-shot solving" et de l'API Control de Clingo permet bien de propager les calculs et les modifications dans l'ontologie pour chaque entité feedback.

```
#const onto="in/oraloos-meta.json".
hasCondition(@dlPro("hasCondition")). hasVerb(@dlPro("hasVerb")). hasObject(@dlPro("hasObject")).
hasSubject(@dlPro("hasSubject")). hasLevel(@dlPro("hasLevel")).
```

FIGURE 4 – Base du feedback avec Exialis.

```
#program step(feedback).
current_user(@dlPro(feedback,"concernsPerson")).
current_challenge(@dlPro(feedback,"concernsChallenge")).
challenge(@dlCro("Challenge",(Challenge,"hasDomain"))) :- current_challenge(Challenge).
has_condition(Challenge, Object) :- hasCondition((Challenge, Condition)),
    hasVerb((Condition, Verb)), hasObject((Condition, Object)),
    hasSubject((Condition, Subject)), Verb = is_a.
has_condition(Challenge, Verb, Object) :- hasCondition((Challenge, Condition)),
    hasVerb((Condition, Verb)), hasObject((Condition, Object)),
    hasSubject((Condition, Subject)), Verb != is_a.
% ==== USABLE ====
usable_before(Challenge) :- challenge(Challenge),not current_challenge(Challenge),
    current_user(User), not forbids((User, Challenge)).
% Challenges without condition
usable(Challenge) :- usable_before(Challenge), not has_condition(Challenge, _),
    not has_condition(Challenge, _, _).
% Challenges with condition(s)
userIsA(@dlTro(User)):- current_user(User).
is_checked(Challenge, ClassCond) :- usable_before(Challenge), has_condition(Challenge, ClassCond),
        current_user(User), userIsA((User,ClassCond)).
is_checked(Challenge, Verb, RelationCond) :- usable_before(Challenge),
        has_condition(Challenge, Verb, RelationCond).
usable(Challenge) :- usable_before(Challenge), N={has_condition(Challenge, Condition)},
    M={is_checked(Challenge, Condition)}, N=M,  I={has_condition(Challenge, Verb, Condition)},
    J={is_checked(Challenge, Verb, Condition)}, I=J.
% === LEVEL MANAGEMENT ====
current_level(CurrentLevel) :- current_challenge(Challenge), hasLevel((Challenge, CurrentLevel)).
usable_level(Challenge, UsableLevel) :- usable(Challenge), hasLevel((Challenge, UsableLevel)).
reinforce_level(Challenge) :- usable_level(Challenge, UsableLevel), current_level(CurrentLevel),
    UsableLevel > CurrentLevel.
remedy_level(Challenge) :- usable_level(Challenge, UsableLevel), current_level(CurrentLevel),
    UsableLevel < CurrentLevel.
same_level(Challenge) :- usable_level(Challenge, UsableLevel), current_level(CurrentLevel),
    UsableLevel = CurrentLevel.
```

FIGURE 5 – Step du feedback avec Exialis.

— Proposer une solution facilement intégrable et utilisable avec une application tierce.

Le dernier objectif de généraliser la librairie Exialis aux problèmes nécessitant une dimension temporelle est atteignable en utilisant la résolution incrémentale nativement présente dans Clingo.

Par rapport à la catégorisation définie dans la section 2.2, la bibliothèque Exialis se positionne comme une approche d'intégration de type API. Elle permet l'utilisation conjointe de Clingo pour la résolution de programmes logiques ASP, et un raisonneur HermiT ou Pellet pour la partie logiques de description, les deux dialoguant grâce à l'API Owlready2.

5 Conclusion et perspectives

Dans l'optique d'apporter une solution complètement déclarative, nous proposons de combiner (hybrider) un langage logique avancé (ASP) avec les logiques de description (OWL2). Nous abordons cette problématique par une démarche pragmatique et opérationnelle, à finalité industrielle. En première approche, nous utilisons la proposition récente OWLAPI HexLite. Si cette proposition donne de bons résultats pour l'aspect déclaratif, plusieurs difficultés de mise en oeuvre nous ont contraint à développer une solution alternative. La librairie Exialis est fonctionnellement similaire à HexLite, 100% Python, opérationnelle pour notre contexte applicatif et généralisable à d'autres problèmes.

Si à court terme, la généralisation d'Exialis reste à finaliser, il serait intéressant d'établir un benchmark d'Exialis par rapport à la solution actuelle ORALOoS (SWRL + Python). De plus, un travail plus théorique reste à accomplir afin d'éclairer quelques différences avec HexLite et d'évaluer l'impact sur la sémantique du raisonnement. Enfin, l'ambition finale de nos travaux est de proposer à la communauté la librairie Exialis en code ouvert, documentée et maintenue.

Remerciements

Ce travail a été réalisé dans le cadre du contrat UCA-Jeolis référence DRV_VALO_2021431 et ANR_ANR21-PRRD-0009-01_ETP, financé conjointement par la société Jeolis Solutions et l'Agence Nationale de la Recherche via le plan France Relance "Préservation des emplois R&D".

Références

[1] Franz Baader and Bernhard Hollunder. Embedding defaults into terminological knowledge representation formalisms. *Journal of Automated Reasoning*, 14 :149–180, 1995.

[2] Adrien F. Baranes, Pierre-Yves Oudeyer, and Jacqueline Gottlieb. The effects of task difficulty, novelty

```
41  #program update(feedback).
42  delta(@addOP("reinforceLevel", feedback, Challenge, onto)) :- reinforce_level(Challenge).
43  delta(@addOP("remedyLevel", feedback, Challenge, onto)) :- remedy_level(Challenge).
44  delta(@addOP("sameLevel", feedback, Challenge, onto)) :- same_level(Challenge).
```

FIGURE 6 – Update du feedback avec Exialis.

and the size of the search space on intrinsically motivated exploration. *Frontiers in Neuroscience*, 8 :317, 2014.

[3] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Eql-lite : Effective first-order query processing in description logics. In *International Joint Conference on Artificial Intelligence*, pages 274–279, 2007.

[4] Richard Comploi-Taupe, Giulia Francescutto, and Gottfried Schenner. Applying incremental answer set solving to product configuration. *26th ACM International Systems and Software Product Line Conference*, B :150–155, 2022.

[5] Jos de Bruijn, David Pearce, Axel Polleres, and Agustín Valverde. Quantified equilibrium logic and hybrid rules. In *Web Reasoning and Rule Systems*, pages 58–72, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[6] Thomas Eiter, Giovambattista Ianni, Thomas Krennwallner, and Axel Polleres. *Rules and Ontologies for the Semantic Web*, pages 1–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[7] Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12) :1495–1539, 2008.

[8] Thomas Eiter, Tobias Kaminski, Christoph Redl, Peter Schüller, and Antonius Weinzierl. Answer set programming with external source access. In *Reasoning Web*, 2017.

[9] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory Solving Made Easy with Clingo 5. In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*, volume 52 of *OpenAccess Series in Informatics (OASIcs)*, pages 2 :1–2 :15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[10] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = asp + control : Preliminary report, 2014.

[11] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, 19(1) :27–82, 2019.

[12] André Giordan. Éducation thérapeutique du patient : les grands modèles pédagogiques qui les sous-tendent. *Médecine des maladies métaboliques*, 4(3) :305–311, 2010.

[13] Xavier Goblet and Christophe Rey. Suivi thérapeutique intelligent par recommandation à base d'ontologie et de règles. *Conférence Nationale sur les Applications Pratiques de l'Intelligence Artificielle, Afia (Ed)*, pages 50–57, 2020.

[14] Roland Kaminski, Javier Romero, Torsten Schaub, and Philipp Wanko. How to build your own asp-based system ?! *Theory and Practice of Logic Programming*, page 1–63, 2021.

[15] Jean-Baptiste Lamy. Owlready : Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. *Artificial Intelligence in Medicine*, 80 :11–28, 2017.

[16] Alon Y. Levy and Marie-Christine Rousset. Combining Horn rules and description logics in CARIN. *Artificial Intelligence*, 104(1) :165–209, 1998.

[17] Gergely Lukácsy and Péter Szeredi. Efficient description logic reasoning in Prolog : The DLog system. *Theory and Practice of Logic Programming*, 9 :343–414, 05 2009.

[18] Thomas Lukasiewicz. A novel combination of answer set programming with description logics for the semantic web. In Enrico Franconi, Michael Kifer, and Wolfgang May, editors, *The Semantic Web : Research and Applications*, pages 384–398, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[19] Boris Motik and Riccardo Rosati. Reconciling description logics and rules. *J. ACM*, 57(5), jun 2010.

[20] Riccardo Rosati. DL+log : Tight integration of description logics and disjunctive datalog. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2006)*, pages 68–78, 2006.

[21] Peter Schüller. The hexlite solver - lightweight and efficient evaluation of hex programs. In *European Conference on Logics in Artificial Intelligence*, 2019.

[22] Peter Schüller. The Hexlite Solver. In Francesco Calimeri, Nicola Leone, and Marco Manna, editors, *Logics in Artificial Intelligence*, pages 593–607, Cham, 2019. Springer International Publishing.

[23] Peter Schüller. A new OWLAPI interface for HEX-programs applied to explaining contingencies in production planning. In *New Foundations for Human-Centered AI, Workshop at ECAI*, 2020.