Constraintes prescriptives compatibles avec OWL2-ER pour évaluer la complétude d'ontologies

Philippe Martin, Jun Jo

Contraintes prescriptives compatibles avec OWL2-ER pour évaluer la complétude d'ontologies

Philippe Martin*,** Jun Jo***

*EA2525 LIM, University of La Réunion, F-97490 Sainte Clotilde, France
Philippe.Martin@univ-reunion.fr,
http://www.phmartin.info
**Adjunct researcher of the School of I.C.T. at Griffith University, Australia
***School of I.C.T., Griffith University, SOUTHPORT QLD 4222, Australia
j.jo@griffith.edu.au

Résumé. L'article définit les contraintes prescriptives comme des règles permettant aux moteurs d'inférence de vérifier que certains objets formels sont réellement utilisés – pas seulement inférés – ou non, dans certaines conditions. Il montre que ces contraintes nécessitent de ne pas exploiter de mécanisme d'héritage (ou autres mécanismes ajoutant des relations à des objets) durant les tests des conclusions des règles. Il donne une méthode générale pour effectuer cela et des commandes SPARQL pour implémenter cette méthode lorsque les règles sont représentées via des relations sous-classe-de entre conditions et conclusions. L'article illustre ces commandes avec la vérification de patrons de conception d'ontologies. Plus généralement, l'approche peut être utilisée pour vérifier la complétude d'une ontologie, ou représenter dans une ontologie (plutôt que par des requêtes ou des procédures *ad hoc*) des contraintes permettant de calculer un degré de complétude d'ontologie. L'approche peut ainsi aider l'élicitation, la modélisation ou la validation de connaissances.

1 Introduction

Les représentations de connaissances (RCs) sont des descriptions formelles permettant des inférences logiques et ainsi des comparaisons automatiques, recherches, fusions, etc. Les RCs sont des formules logiques, e.g. les *prédicats binaires* de la logique du 1er ordre aussi appelés *triplets* ou *instances de propriété* dans RDF (RDFS 1.1, 2014) et *relations binaires* dans les Graphes Conceptuels (GCs) (Sowa, 1992). Dans cet article, par souci de clarté, nous utilisons la terminologie intuitive des GCs : les *objets* d'information sont des *types* ou bien des *individus*, et les types sont des *types de relations* ou bien des *types de concepts* (*classes* et *types de données* dans RDF). Une base de connaissances formelle (BC) est une collection de tels objets écrits via un langage de RCs (LRC). Une ontologie est une BC portant essentiellement sur des types.

Créer ou évaluer une BC étant difficile, une de ses sous-tâches est souvent *l'évaluation d'un degré de complétude de la BC selon certains critères* ou, par abréviation, "sa complétude". Une telle évaluation est effectuée dans diverses tâches, mais de manière différente suivant les

outils et parfois de manière implicite ou *ad hoc*. Des exemples de telles tâches ou domaines sont : i) l'extraction automatique ou manuelle de connaissances ou la création d'une BC, ii) l'exploitation de modèles de conception d'ontologies, et iii) l'évaluation d'ontologies ou, plus généralement, de données. Dans ce dernier domaine, comme noté par Zaveri et al. (2016), "complétude" réfère communément au *degré de présence* des informations *requises* pour satisfaire certains critères ou une certaine requête. Soit les informations manquantes sont trouvées dans une *BC de référence* complète, soit ce degré est estimé via des *oracles de complétude* (Galárraga et al., 2017), i.e. des règles ou des requêtes permettant d'estimer ce qui manque à la BC pour répondre à une reqête donnée aussi bien qu'avec une hypothétique *BC de référence* complète.

Dans cet article, nous adoptons cette définition générale mais avec une légère extension ou précision : ce qui est *requis* doit être représenté explicitement, i.e. via des contraintes dans la BC, e.g. des contraintes d'intégrité. En effet, s'il existe une *BC de référence* complète, elle peut souvent être utilisée directement, et si elle n'existe pas, les *oracles de complétude* peuvent être au moins partiellement implémentés en exploitant des contraintes. Dans tous les cas, il est avantageux de représenter *dans la BC* (donc via des contraintes plutôt que par exemple via des requêtes) ce qui est requis, pour faciliter l'exploitation ou la réutilisation de ces informations. L'usage de contraintes permet de *vérifier* la complétude d'une BC avec n'importe quel moteur d'inférence pouvant exploiter ces contraintes : une implémentation *ad hoc* n'est pas requise. Si une simple *vérification* n'est pas suffisante, i.e., si un *degré de complétude* est recherché, un moyen simple de le définir précisément et de le calculer est de diviser "le nombre de faits (dans la BC) satisfaisant les contraintes" par "le nombre total de faits dans la BC". Cette méthode est basique par rapport à celles qui agrègent les résultats d'*oracles de complétude* mais elle n'est pas le sujet de cet article. Celui-ci est de fournir un moyen simple de représenter des contraintes puis de les vérifier (avec SPARQL) pour pouvoir évaluer ou calculer une complétude de BC.

La section 2 distingue les contraintes descriptives des contraintes prescriptives positives et montre que i) ces dernières ne peuvent être représentées dans les logiques classiques, même avec l'hypothèse du monde fermé, et ii) sont nécessaires pour vérifier la complétude d'une BC. Cette section introduit ensuite i) des types spéciaux indiquant que certaines expressions sont des contraintes, ii) une méthode pour vérifier des contraintes prescriptives, iii) OWL2-ER, un sous-ensemble de OWL (Profils OWL2, 2014), et iv) les *contraintes compatibles avec OWL2-ER*, comme un moyen simple de représenter des contraintes dans tout LRC dont l'expressivité est au moins égale à RDFS (RDFS 1.1, 2014).

La section 3 montre comment cette méthode de vérification de contraintes prescriptives peut être implémentée via un ensemble réduit de commandes SPARQL. La section 4 illustre l'exploitation de ces commandes et ainsi aussi les limites de ce que la *compatibilité avec OWL2-ER* induit. La section 5 compare notre approche avec d'autres, l'évalue et conclut.

2 Contraintes prescriptives compatibles avec OWL2-ER

2.1 Contraintes : positives vs. négatives, descriptives vs. prescriptives

Dans cet article, comme dans (Chein et Mugnier, 2008), les contraintes peuvent être *positives* ou *négatives*, pour respectivement exprimer des faits de la forme "*si A est vrai, B doit aussi l'être*" et "*si A est vrai, B ne doit pas l'être*".

Comme noté par Assmann et Wagner (2006), les modèles d'ingénierie peuvent être i) *descriptifs* d'une réalité, tels la plupart des ontologies, ou ii) *prescriptifs* de ce que les informations représentées doivent être, tels les spécifications de systèmes, les méta-modèles, les schémas XML ou de bases de données, les spécifications avec SHACL (Shapes Constraint Language) (SHACL, 2017), etc. Similairement, nous distinguons deux sortes de contraintes positives. Les *contraintes descriptives* sont comme des définitions ou des axiomes : ils permettent aux moteurs d'inférence de vérifier *l'utilisation* de certains termes formels (s'ils sont utilisés). Les *contraintes prescriptives* permettent aux moteurs d'inférence de vérifier que certains termes formels sont réellement utilisés (pas seulement inférés) ou bien non utilisés, dans certaines conditions. E.g., les contraintes prescriptives peuvent être utilisées pour vérifier que si un type est défini comme ayant (nécessairement) certaines relations, ces relations sont *explicitement* données par les utilisateurs lorsqu'ils créent une instance d'un tel type. Le mot "explicitement" signifie ici que ces relations ne doivent pas simplement exister suite à une déduction automatique, e.g. par héritage, mais parce qu'elles ont été créées par un utilisateur.

Peu de LRCs permettent d'utiliser des contraintes et ils ne permettent généralement pas de spécifier des *contraintes prescriptives positives*. À titre d'exemple, supposons qu'une BC inclut la règle "Si X est une personne, X a un parent" ou la définition "toute personne a (nécessairement) un parent", et qu'un utilisateur ajoute le fait "Jean est une personne". Même si dans cette BC, la règle ci-dessus est aussi une contrainte *descriptive*, aucun message d'erreur n'a à être généré par un mécanisme de vérification puisque cette contrainte est satisfaite (par inférence) sans que l'utilisateur n'ait à représenter un parent pour Jean. Par contre, si cette même règle est aussi marquée comme *prescriptive* (au lieu de descriptive), l'ajout de Jean en tant que personne mais sans relation à un parent doit alors être refusé. En d'autres termes, si un mécanisme associe automatiquement des relations à des objets – e.g., par héritage dynamique ou, statiquement, via une saturation par chaînage avant – ce mécanisme doit être adéquatement désactivé ou contourné pour vérifier des contraintes prescriptives positives. Les contraintes négatives sont à la fois descriptives et prescriptives puisqu'elles permettent la détection de RCs incorrects qu'ils aient été ajoutés automatiquement ou non.

Les contraintes prescriptives permettent ainsi des vérifications autres que via les contraintes descriptives, et elles ne sont *pas équivalentes à l'utilisation de l'hypothèse de monde fermé*. Les expressions logiques classiques sont seulement descriptives. E.g., comme vu ci-dessus, indiquer que "toute personne a (nécessairement) un parent" est seulement descriptif. Des règles (marquées comme) représentant des contraintes prescriptives positives nécessitent donc une interprétation spéciale, e.g. via une commande ou procédure spéciale.

2.2 Méthode générale et types pour contraintes prescriptives

Tao et al. (2010) montrent que représenter et vérifier des contraintes d'intégrité exploitant certaines formes de *l'hypothèse du nom unique* ou de *l'hypothèse de monde fermé* peut être effectué via des *requêtes* SPARQL, une requête différente pour chaque contrainte différente. Notre but est plutôt de permettre la représentation de contraintes dans les BCs, et ce quel que soit le LRC utilisé, afin d'augmenter les possibilités d'exploitation et de réutilisation de ces contraintes, e.g. via *quelques* requêtes SPARQL prédéfinies. De plus, nous souhaitons aussi et surtout prendre en compte les contraintes prescriptives.

Pour cela, notre approche est d'introduire des types de contraintes. En reliant des RCs à ces types via des relations instance-de ou sous-type-de, les créateurs de BCs

peuvent signifier que ces RCs sont des contraintes. Ainsi, elles peuvent être retrouvées ou interprétées d'une manière spéciale par des moteurs d'inférence. L'ontologie OWL2 est similairement utilisée pour augmenter l'expressivité de RDF. Le nom de notre ontologie de types de contraintes est CSTR. Dans cette ontologie, `cstr:Constraint` est le supertype de tous les types de contraintes. Similairement, le type `cstr:Prescriptive_constraint`, un sous-type de `cstr:Constraint`, permet de retrouver ou spécifier (seulement) des contraintes prescriptives. Le type `cstr:Constraint_via_types` est le supertype des types de contraintes qui se représentent via des relations entre types. Ce dernier est instance du type de 2e ordre `cstr:Type_of_constraint_via_types`. Indiquer qu'un type est sous-type de `cstr:Constraint_via_types` est suffisant pour spécifier que toutes les définitions de ce type sont aussi des contraintes. Pour éviter un tel héritage, il faut plutôt indiquer que le type est instance de `cstr:Type_of_constraint_via_types`. La partie "`cstr:`" de ces identifiants est une abréviation de l'espace de noms `http://www.webkb.org/kb/it/CSTR`.

Pour une vérification adéquate des contraintes prescriptives positives, la sous-section 2.1 a introduit la nécessité de temporairement désactiver ou contourner les mécanismes d'inférence associant automatiquement des relations à des objets. Toutefois, ces mécanismes sont utiles pour vérifier si un objet vérifie ou non la condition d'une contrainte prescriptive positive. Ainsi, ils ne doivent être désactivés ou contournés que pour le principal (alias, 1er) objet de la conclusion de cette contrainte, i.e. l'objet dont les relations sont obligatoires pour tous les objets satisfaisant la condition de la contrainte. Nous proposons la méthode de contournement suivante : statiquement via un pré-traitement de la BC ou dynamiquement lors de la vérification de telles contraintes, créer un "clone sans type" de chaque objet vérifiant la condition d'une telle contrainte puis, lors de la vérification de sa conclusion, l'effectuer sur ce clone. Un tel clone a les mêmes relations que l'original sauf pour les relations *instance-de* (il n'en a pas ; de plus, s'il s'agit d'un individu non anonyme, il doit avoir un identifiant différent de l'objet original). Avec un tel clone, les "inférences exploitant les types pour associer les relations à un objet" sont évitées. Pour abrévier, nous écrirons désormais que cette méthode permet d'éviter *l'héritage*. Cette méthode ne fonctionne pas pour les "inférences n'exploitant pas les types" (e.g., celles basées sur le "typage canard" plutôt que l'héritage), ni si une *saturation par chaînage avant* est automatiquement effectuée avant le pré-traitement cité ci-dessus, mais ces deux cas sont rares.

Cette méthode repose sur une modification temporaire des RCs avant leur vérification par un moteur d'inférence. Ainsi, cette méthode ne repose *pas* sur – i.e., est *indépendante* de – un LRC ou un moteur d'inférence particulier. Ainsi, pour différent domaines ou applications, des moteurs d'inférence différents peuvent être utilisés pour vérifier ou évaluer la complétude d'ontologies. Avec certains langages de requête tels que les versions actuelles de SPARQL, la modification temporaire ne peut être effectuée dynamiquement, un pré-traitement de la BC est nécessaire. Ceci est une limitation car peu de serveurs de BC autorisent (la plupart de) leurs utilisateurs à modifier la BC pour la vérifier.

2.3 Contraintes compatibles avec OWL2-ER

Comme les contraintes sont des règles particulières et comme nous souhaitons représenter les contraintes de manière simple et indépendante de LRCs ou notations particulières, nous avons tout d'abord considéré OWL2-RL (Profils OWL2, 2014). Celui-ci peut être entièrement défini via des règles de Horn définies et l'égalité, donc avec Datalog, un sous-ensemble

purement déclaratif de Prolog. Cependant, dans de telles règles, la conclusion ne peut inclure des objets anonymes existentiellement quantifiés. Ceci est possible avec les "règles existentielles" ou Datalog+. Baget et al. (2015) montrent qu'un sous-langage de OWL2 qu'ils nomment OWL2-ER peut représenter de nombreuses sortes de règles existentielles (d'où le suffixe "-ER") simplement en utilisant une relation `rdfs:subClassOf` (RDFS 1.1, 2014) entre deux expressions de classe de OWL2, l'expression de la superclasse représentant la conclusion de la règle. En d'autres termes, la relation `rdfs:subClassOf(C1,C2)`, ici exprimée dans une notation fonctionnelle, peut être traduite de la manière suivante dans une notation Peano-Russel pour la logique du 1er ordre : $\forall$`x(C1(x)` $\Rightarrow$ `C2(x))`. OWL2-ER correspond donc à peu près à la partie de Datalog+ pouvant être exprimée en utilisant seulement un sous-ensemble de OWL2, donc seulement avec des relations binaires et sans variables partagées à la fois par la condition et la conclusion d'une règle. Dans OWL2-ER, une contrainte négative peut être représentée de deux manières : i) via une expression de classe équivalente au type `owl:Nothing` dans la conclusion d'une règle, i.e. via une règle de la forme $\forall$`x (ClassExpression(x)` $\Rightarrow$ $\bot$`)`, et ii) via le type `owl:NegativeObjectPropertyAssertion` pour exprimer une négation de la forme $\neg$ $\exists$`x ClassExpression(x)`. Ni OWL2-ER ni Datalog+ ne peuvent *directement* représenter une contrainte positive mais, comme expliqué dans la sous-section 2.2, ceci peut être exprimé en spécifiant qu'une règle est instance de `cstr:Constraint`.

Dans cet article, les contraintes sont toujours représentées par des règles (nous n'exploitons pas la forme négative citée ci-dessus) et en utilisant des relations `rdfs:subClassOf` car c'est une façon simple (quoique restrictive) de représenter des règles. Baget et al. (2015) montrent que OWL2-ER peut être traduit en Datalog+ (même si tout OWL2 ne peut être représenté en Datalog+), puis dans RuleML (RuleML 1.01 deliberation, 2015). Comme elles n'exploitent que des relations `rdfs:subClassOf`, nos techniques peuvent travailler avec n'importe quel LRC ayant au moins l'expressivité de RDFS (RDFS 1.1, 2014) ou, tel RDF, permettant de réutiliser RDFS. Ce dernier point est le sens de l'expression "contraintes compatibles avec OWL2-ER". En d'autres termes, ces techniques n'exigent *pas qu'au moins ou au plus* OWL2-ER soit utilisé dans les RCs exploitées. Il ne serait donc pas pertinent pour cet article de donner plus de détails sur OWL2-ER ou une formalisation de ce que représenter une règle avec une relation `rdfs:subClassOf` implique. Baget et al. (2015) et Swan (2016) offrent une exploration formelle de ce que ceci implique. Pour cet article, le code SPARQL de la section suivante fournit les détails formels nécessaires. De même, pour cet article, une formalisation des contraintes prescriptives autres que celle donnée via le code SPARQL ne serait pas utile.

Les *contraintes compatibles avec OWL2-ER* pourraient être généralisées en utilisant une relation sous-type-de au lieu d'une relation `rdfs:subClassOf` entre la condition et la conclusion d'une contrainte. Conformément à la terminologie donnée dans l'introduction, ceci permettrait l'utilisation de types de relations ou de données (donc pas seulement des classes) dans la condition et la conclusion d'une contrainte. Cependant, la vérification d'objets tels les relations ou les instances de types de données peut le plus souvent s'effectuer via des contraintes sur des instances de classe liées à ces objets. E.g., les relations peuvent généralement être vérifiées via leurs sources ou destinations, ou via chaque *fait portant sur des relations* tel chaque instanciation de `owl:NegativeObjectPropertyAssertion`. Dans cet article, nous n'utilisons donc que des relations `rdfs:subClassOf` entre les conditions et conclusions de contraintes.

Pour signifier qu'une (expression de) classe est la condition d'une contrainte compatible avec OWL2-ER, le type de 1er ordre `cstr:OWL2-ER-like_constraint_condition` et le type

de 2e ordre `cstr:Type_of_OWL2-ER-like_constraint_condition` sont proposés par notre ontologie CSTR. Ils s'utilisent respectivement comme indiqué plus haut pour les types `cstr:Constraint_via_types` et `cstr:Type_of_constraint_via_types`. Les requêtes SPARQL de cet article n'utilisent que `cstr:OWL2-ER-like_prescriptive_constraint_condition` et `cstr:OWL2-ER-like_constraint_condition` car ce sont des types du 1er ordre. En effet, la plupart des *moteurs d'inférence pour logiques de description* ne peuvent gérer une BC exploitant des types de 2e ordre non prédéfinis dans ces logiques. Pour cette même raison, si des contraintes portant sur des définitions de type doivent être vérifiées avec *ces* moteurs d'inférence, un pré-traitement de la BC afin d'en ôter les individus est requis. Dans un tel cas, les classes de 1er ordre deviennent des individus et leurs relations sous-type doivent également être retirées. Inversement, avant de vérifier les individus *avec de tels outils*, les types de 2e ordre non pré-définis doivent être supprimés.

Plus d'exemples sont donnés dans (Martin, 2017), le document Web associé à cet article.

3 Commandes SPARQL pour l'exploitation de contraintes prescriptives compatibles avec OWL2-ER

Dans des extensions de SPARQL telle LDScript (Corby et al., 2017), les commandes suivantes peuvent être séquencées dans des scripts ou des fonctions. Dans SPARQL, les noms de variables commencent par " ?". Dans cet article, pour des raisons de clarté, les noms des types de relations débutent par une lettre minuscule tandis que les autres noms débutent par une majuscule. Comme SPARQL réutilise la notation Turtle (Turtle, 2014), nous l'utilisons aussi dans la section 4. Un fait de la forme " Source rel1 Destination1_1 , Destination1_2 ; rel2 Destination2_1 , Destination2_2 " peut être lu " Source a pour rel1 Destination1_1 et Destination1_2, et a pour rel2 Destination2_1, et a pour rel2 Destination2_2 ".

Commande 1 : pré-traitement de la BC retirant temporairement des individus pour vérifier des contraintes sur des types via des moteurs d'inférences de logiques de description classiques. La clause `WHERE` de la commande ci-dessous sélectionne chaque objet `?o` qui n'a *pas* de type `rdfs:Class`, donc qui est un individu. La clause `DELETE` supprime les relations `rdf:type` depuis `?o` et, depuis leurs destinations, supprime les relations `rdfs:subClassOf` lorsqu'elles existent. Pour remplacer ces relations `rdfs:subClassOf`, la clause `INSERT` ajoute les relations `cstr:type` et `cstr:subClassOf`. En effet, elles n'ont de signification particulière pour un moteur d'inférence et donc ne gênent pas l'appariement d'objets avec des classes, i.e. l'inférence de relations `rdf:type`. Enfin, elles permettent plus tard de ré-affirmer les relations `rdf:type` et `rdfs:subClassOf` initiales via une commande similaire.

```
DELETE { ?o rdf:type  ?t .  ?t rdfs:subClassOf ?superClass . }
 INSERT{ ?o cstr:type ?t .  ?t cstr:subClassOf ?superClass . }
 WHERE { ?o rdf:type  ?t .  ?t rdfs:subClassOf ?superClass .
         FILTER NOT EXISTS { ?o rdf:type rdfs:Class }        }
```

Commande 2 : pré-traitement de la BC créant des "clones sans type" d'objets pour exploiter ces objets sans mécanisme d'héritage. SPARQL ne fournit pas de moyen pour supprimer l'héritage lors de l'exécution d'une requête. E.g, il ne permet pas de sélectionner un

"régime d'inférence" (SPARQL 1.1 entailments, 2013) particulier dans une requête. Toutefois, la méthode de contournement de l'héritage donnée en section 2.2 peut être implémentée en SPARQL. La commande ci-dessous fournit un exemple où, par souci de clarté, il est supposé que la BC ne contient pas de type de 2e ordre. Pour chaque objet `?o` dans la BC, si cet objet est un individu, la commande crée `?o2`, une copie partielle de `?o` qui a les mêmes relations moins les relations `rdf:type`. Cette copie partielle a pour identifiant celui de `?o` mais avec le suffixe "_cloneWithoutType". Cette commande relie aussi `?o` à `?o2` par une relation `cstr:cloneWithoutType`. Martin (2017) montre qu'avec une extension de SPARQL telle que STTL (Corby et Faron-Zucker, 2015), il n'est pas utile d'effectuer un tel pré-traitement du BC pour la commande 3 (seule commande qui le nécessite) car une requête `CONSTRUCT` peut être inclue dans une requête `SELECT` et permettre ainsi la création des clones "à la volée ".

```
INSERT { ?o cstr:cloneWithoutType ?o2 . ?o2 ?r ?dest } WHERE
{ ?o ?r ?dest .   FILTER (?r != rdf:type)
  FILTER NOT EXISTS { ?o rdf:type rdfs:Class }
  BIND (uri(concat(str(?o), "_cloneWithoutType")) as ?o2)
}
```

Commande 3 : vérification des contraintes prescriptives positives. Cette commande est une requête. Elle liste tous les objets violant une contrainte prescriptive *positive*. Comme le montre le code, un tel objet satisfait deux conditions. Tout d'abord, il peut être apparié – et donc avoir pour type – la condition d'une contrainte `?posConstr` qui est sous-classe de `cstr:OWL2-ER-like_prescriptive_constraint_condition` et qui n'a pas `owl:Nothing` dans sa conclusion. Par ailleurs, un tel objet ne peut être apparié – et donc ne peut avoir pour type – la conclusion de la contrainte, i.e. sa superclasse. Ainsi, cette requête nécessite un moteur SPARQL qui a un régime d'inférence permettant l'appariement (alias, catégorisation) d'un individu par rapport à une expression de classe et donc la déduction d'une relation `rdf:type` entre eux. Dans le code des commandes ci-dessous, chaque ligne utilisant une telle déduction finit par un commentaire débutant par "`#appariement`". Si, par exemple, tous les individus, conditions de contraintes et conclusions de contraintes sont décrits dans OWL2-QL, un *régime d'inférence* OWL2-QL est requis et suffisant. Dans ce cas, un moteur d'inférence capable de gérer l'expressivité de OWL2-QL est requis.

```
SELECT ?objectNotMatchingPosConstr ?posConstr  WHERE
{ #corps de cette seconde commande: entre ce premier '{' et le dernier '}'
  ?posConstr rdfs:subClassOf cstr:OWL2-ER-like_prescriptive_constraint_condition ,
                             ?posConstr_conclusion .  #initialisation de ?posConstr
  FILTER NOT EXISTS { ?posConstr rdfs:subClassOf owl:Nothing } #-> contrainte non négative
  ?objectNotMatchingPosConstr rdf:type ?posConstr. #appariement de la condition
  FILTER NOT EXISTS  #-> objets satisfaisant la conclusion non listés
  { ?objectNotMatchingPosConstr rdf:type ?posConstr_conclusion } #appariement
}
```

Commande 4 : vérification des contraintes prescriptives négatives. Cette requête liste chaque objet violant une contrainte négative, i.e. chaque objet s'appariant – et donc ayant pour type – un type `?negConstr` sous-classe du type `cstr:OWL2-ER-like_constraint_condition` et qui a `owl:Nothing` dans sa conclusion. Le fait qu'il n'y ait pas de distinction entre descriptif

et prescriptif pour les contraintes négatives rend cette requête plus simple que la précédente, et même inutile si le moteur d'inférence utilisé pour l'appariement est utilisé directement pour vérifier toute la BC.

```
SELECT ?objectMatchingNegConstr ?negConstr  WHERE
{ ?negConstr rdfs:subClassOf cstr:OWL2-ER-like_constraint_condition , owl:Nothing .
  ?objectMatchingNegConstr rdf:type ?negConstr . #appariement
}
```

Commande 5 : vérifier des relations binaires au lieu d'individus. Pour lister des *relations binaires* violant des contraintes prescriptives, plutôt que de lister des individus ayant des relations violant de telles contraintes, il suffit de remplacer la relation `rdf:type` par une relation d'implication entre formules logiques dans les deux requêtes précédentes. Pour référer à une telle implication, Tim Berners-Lee utilise l'identifiant `log:implies` (Berners-Lee et al., 2008). Pour que ce remplacement fonctionne, le moteur SPARQL utilisé doit exploiter un moteur d'inférence pouvant déduire l'existence d'une telle implication quand elle existe entre les formules appariées. Comme les requêtes sur les individus, celles sur les relations peuvent utiliser des filtres supplémentaires. E.g., pour que la commande 4 fonctionne *seulement* sur les formules négatives, on peut ajouter à la fin de son corps :

```
?objectMatchingNegConstr rdf:type owl:NegativeObjectPropertyAssertion. #appariement
```

Commande 6 : évaluation de la complétude d'une ontologie. Un moyen simple de définir ou calculer le degré de complétude d'une BC est de diviser "le nombre de relations (dans la BC) ne violant pas de contraintes prescriptives" par "le nombre total de relations (liées à au moins un autre objet)". Au lieu de relations, la commande ci-dessous recherche des individus mais peut être adaptée comme indiqué dans le précédent paragraphe pour implémenter la définition de complétude ci-dessus.

```
SELECT ( ((?nbObjs - ?nbAgainstPosCs - ?nbMatchingNegCs) / ?nbObjs) AS ?completeness)
{ {SELECT (COUNT(DISTINCT ?o) AS ?nbObjs)
     WHERE { ?o ?r ?o2 } } #tout objet (lié à un autre objet)
  {SELECT(COUNT(DISTINCT ?objectNotMatchingPosConstr) AS ?nbAgainstPosCs)
     WHERE { ... #le corps de la commande 3 peut être copié ici
           } }
  {SELECT (COUNT(DISTINCT ?objectMatchingNegConstr) AS ?nbMatchingNegCs)
     WHERE { ... #le corps de la commande 4 peut être copié ici
           } }
}
```

4 Exemples de contraintes prescriptives ainsi exploitables

Pour la construction de hiérarchies de sous-types, divers travaux de recherche conseillent d'utiliser des structures d'arbres, e.g. (Rector et al., 2012). Martin (2017) montre qu'utiliser des partitions de sous-types au lieu de structures d'arbre a les mêmes avantages et moins d'inconvénients, mais que cette approche est inutilement lourde pour classer des "types non naturels". Pour catégoriser des classes via un LRC réutilisant OWL2, Martin

(2017) propose donc de n'utiliser que des relations de type `cstr:equivDisjointUnion`, `cstr:subclassOfDisjointUnion`, `cstr:nonNaturalSubclass` ou des sous-types d'un de ces trois types. Il définit ces trois types en OWL2, ainsi que leur supertype commun : `cstr:nonNaturalOrPartitionSubclass`. Le patron de conception proposé peut alors être vérifié via la contrainte prescriptive positive suivante : "s'il existe une relation sous-classe-de entre deux classes, cette relation doit être de type `cstr:nonNaturalOrPartitionSubclass`". Une contrainte équivalente est : "si une classe C1 a une relation sous-classe vers une classe C2, cette relation doit être de type `cstr:nonNaturalOrPartitionSubclass`". OWL2 ne permettant pas d'utiliser de variable pour référer à ladite relation ou à C2, une version compatible avec OWL2-ER est : "si une classe C1 a une relation sous-classe, C1 doit avoir une relation de type `cstr:nonNaturalOrPartitionSubclass`". Une représentation en Turtle est ci-dessous. Pour contourner l'ambiguité inhérente à cette dernière version et effectuer la vérification voulue, il ne faut pas vérifier chaque objet de la BC (i.e., il ne faut pas utiliser la commande 3 directement) mais vérifier chaque relation, une par une (i.e., utiliser `log:implies` comme indiqué dans le texte associé à la commande 5).

```
cstr:Subclass #classe ayant une sous-classe ; condition de contrainte associée à cela
  rdfs:subClassOf cstr:OWL2-ER_prescriptive_constraint_condition ;
  owl:equivalentClass             #définition de la condition de la contrainte :
    [rdf:type owl:Restriction ;   #  "classe qui a une sous-classe"
     owl:onProperty cstr:subclass ;   owl:someValuesFrom rdfs:Class ];
  rdfs:subClassOf  #conclusion (les types de relations qui doivent être présents) :
    [rdf:type owl:Restriction ;  owl:onProperty cstr:nonNaturalOrPartitionSubclass;
     owl:someValuesFrom rdfs:Class ].
```

Martin (2017) généralise cette contrainte (et les types qu'elle exploite) à la vérification de tout type de relation transitive. Plus exactement, il propose une commande SPARQL qui génère et ajoute à la BC une contrainte pour chaque type de relation transitive instance d'un certain types du 2e ordre. Avec certains moteurs d'inférence, un pré-traitement de la BC est donc à effectuer pour temporairement supprimer de tels types avant de vérifier les contraintes. De manière similaire, Martin (2017) définit une autre contrainte (et les types qu'elle exploite) pour vérifier que les types de relations instances d'un certain type sont systématiquement utilisés lorsque leur signature le permet, sauf sous des conditions données.

5 Évaluation, comparaisons et conclusion

Notre approche permet i) la représentation de contraintes prescriptives avec tout LRC dont l'expressivité est au moins égale à RDFS, et ii) l'exploitation de n'importe quel moteur d'inférence, y compris via des requêtes SPARQL. C'est son originalité. La contribution de cet article est d'avoir montré comment, jusqu'à quel point (i.e., moyennant quelles sortes de pré-traitements de la BC pour compenser des limites de SPARQL ou de OWL2), et pourquoi : i) la possibilité de représenter des contraintes prescriptives via des RCs plutôt que via des requêtes ou des procédures *ad hoc*, et donc aussi ii) la possibilité de réutiliser ces contraintes et des moteurs d'inférences dans diverse tâches liées à la vérification ou l'évaluation de la complétude d'une BC. E.g., pour construire une BC, il est possible d'utiliser des contraintes prescriptives pour représenter des patrons de conception d'ontologies ou des modèles de tâches

génériques, puis d'utiliser un moteur d'inférences pour vérifier la complétude de la BC et, en fonction des résultats, éliciter la connaissance manquante auprès d'experts.

Notre approche est difficile à évaluer théoriquement puisqu'elle repose sur d'autres méthodes ou outils. Elle hérite de leurs avancées théoriques ou pratiques. Baget et al. (2015) et Swan (2016) listent des avancées théoriques pertinentes pour cette approche. Nous n'avons validé notre approche qu'expérimentalement en l'utilisant pour vérifier des ontologies ou calculer leur degré de complétude. Ce degré est faible pour des contraintes implémentant des patrons de conception peu connus, tels ceux de la section 4. Nous poursuivrons ces expériences et ajouterons leurs analyses à (Martin, 2017). En ce qui concerne l'usage de SPARQL pour vérifier des contraintes, Tao et al. (2010) montrent que SPARQL peut être utilisé pour à la fois exprimer et valider des contraintes d'intégrité utilisant des formes partielles de "l'hypothèse du monde fermé" ou de "l'hypothèse des noms uniques", et ce de manière cohérente et complète si certaines conditions sur l'expressivité utilisée dans la BC et pour les contraintes sont respectées. Dans notre approche, les requêtes ne servent qu'à valider des contraintes, pas à les exprimer, mais ceci n'est qu'une généralisation de l'approche de Tao et al. (2010) qui ne change pas leurs résultats théoriques. Dans (Tao et al., 2010), les formes partielles des hypothèse du monde fermé ou des noms uniques sont dans SPARQL spécifiables via ses opérateurs `EXISTS` et `NOT EXISTS` de SPARQL et les relations `owl:sameAs` et `owl:differentFrom`. Ces formes peuvent similairement s'exprimer via les commandes vues en section 3 et l'usage des relations `owl:sameAs` et `owl:differentFrom` dans les contraintes.

Notre approche est basée sur une utilisation particulière de RDFS. Elle doit donc être comparée à SHACL (Shapes Constraint Language) (SHACL, 2017), une ontologie de langage (telle OWL2) proposée par le W3C pour permettre de définir des contraintes en RDF. SHACL ne réutilise pas OWL2 pour définir des contraintes : il introduit de nouveaux termes. Il ne permet donc pas de réutiliser – pour vérifier les contraintes – des moteurs d'inférence comprenant le sens spécial de termes de OWL2 : un moteur d'inférence dédié à SHACL doit être utilisé et un nouveau LRC (SHACL) doit être appris. De plus, SHACL ne fait pas de distinction entre contraintes descriptives et prescriptives, et ne traite donc que très partiellement ces dernières. En effet, le fait que SHACL permette de spécifier quel régime d'inférence doit être utilisé avec quelle contrainte, y compris l'absence d'inférence, est insuffisant : un régime expressif peut être nécessaire pour apparier des objets à la condition d'une contrainte prescriptive même si pour vérifier sa conclusion l'héritage doit en effet être supprimé. Comme une instruction SHACL peut incorporer des requêtes SPARQL SELECT, celles proposée dans cet article pourrait être réutilisées dans SHACL. Cependant, SHACL ne peut pas exploiter les requêtes SPARQL de modification Ainsi, si des pré-traitements de la BC sont nécessaires, un autre langage doit être utilisée pour les spécifier. Les exemples de contraintes de la section 4 ne semblent pas pouvoir se représenter en SHACL.

Certains langages ou systèmes de transformation exploitent des RCs. Zamazal et Svátek (2015) et Corby et Faron-Zucker (2015) présentent de tels systèmes. Quoique peu d'entre eux permettent d'utiliser directement une fonction d'appariement de motifs de RCs sans aussi transformer les RCs appariés – PatOMat (Zamazal et Svátek, 2015) est une exception – ces langages ou systèmes pourraient être adaptés pour avoir une telle fonction et ainsi être utilisés pour gérer des contraintes prescriptives. Toutefois, à notre connaissance, tous ces systèmes utilisent des langages plus expressifs que OWL2-ER. E.g., ils utilisent généralement des langages basés sur des règles permettant l'usage de variables pour relier les objets partagés

par la condition et la conclusion de la règle. Utiliser de tels langages peut simplifier l'écriture des contraintes prescriptives. Cependant, concernant *ce qui peut être exprimé* et vérifié via des contraintes prescriptives, cet article et (Martin, 2017) montrent que i) beaucoup peut être réalisé *simplement en utilisant OWL2-ER et SPARQL*, et ii) la puissance de notre approche est liée à la puissance du moteur d'inférence utilisé pour les appariements, pas seulement au langage utilisé.

Certains systèmes de transformation, tel PatOMat (Zamazal et Svátek, 2015), génère des requêtes SPARQL (pour détecter des motifs) basées sur des spécifications de *motifs et leurs transformations* dans un autre langage. Certains autres systèmes de transformation offrent directement une extension de SPARQL telle STTL (Corby et Faron-Zucker, 2015) pour écrire des spécifications de motifs et leurs transformations. E.g., comme montré par Corby et al. (2016), STTL peut être combiné avec LDScript pour i) spécifier des requêtes STTL (compilées dans des requêtes SPARQL) détectant des modèles, puis ii) transformer les résultats. Toutefois, Corby et al. (2016) ne mentionnent pas l'exploitation de moteurs d'inférence pour apparier des objets, et ne fait pas de distinction entre contraintes descriptives et prescriptives. Nos commandes SPARQL, y compris celles générant des contraintes prescriptives, pourraient donc utilement être réutilisées dans ces systèmes de transformation, sous une forme adaptée. Nous allons explorer ceci en utilisant STTL+LDScript.

Références

Assmann, U. et G. Wagner (2006). Ontologies, Meta-models, and the Model-Driven Paradigm. In *Ontologies for Software Engineering and Software Technology*, pp. 249–273. Springer.

Baget, J., A. Gutierrez, M. Leclère, M. Mugnier, S. Rocher, et C. Sipieter (2015). Datalog+, RuleML and OWL 2 : Formats and Translations for Existential Rules. In *Challenge+DC@RuleML*. Berlin, Germany.

Berners-Lee, T., D. Connolly, L. Kagal, Y. Scharf, et J. Hendler (2008). N3Logic : A Logical Framework For the World Wide Web. *Theory and Practice of Logic Programming 8*(3), 249–269.

Chein, M. et M.-L. Mugnier (2008). *Graph-based Knowledge Representation : Computational Foundations of Conceptual Graphs*. Springer.

Corby, O. et C. Faron-Zucker (2015). STTL : A SPARQL-based Transformation Language for RDF. In *WEBIST 2015, 11th International Conference on Web Information Systems and Technologies*, pp. 466–476. Lisbon, Portugal.

Corby, O., C. Faron-Zucker, et F. Gandon (2017). LDScript : A Linked Data Script Language. In *ISWC 2017, 16th International Semantic Web Conference*, pp. 208–224. Vienna, Austria.

Corby, O., C. Faron-Zucker, et R. Gazzotti (2016). Validating Ontologies Against OWL 2 Profiles with the SPARQL Template Transformation Language. In *RR 2016, Web Reasoning and Rule Systems*, pp. 39–45. Springer.

Galárraga, L., K. Hose, et S. Razniewski (2017). Enabling completeness-aware querying in SPARQL. In *WebDB 2017*, pp. 19–22. Chicago, IL, USA.

Martin, P. (2017). OWL2-ER Compatible Prescriptive Constraints to Evaluate Ontology Completeness. Web page, http://www.webkb.org/kb/it/o_knowledge/d_constraints_owl2er.html.

Profils OWL2 (2014). OWL 2 Web Ontology Language Profiles (Second Edition), W3C Recommendation 11 December 2012. Web page, http://www.w3.org/TR/owl2-profiles/.

RDFS 1.1 (2014). RDF Schema 1.1, W3C Recommendation 25 February 2014. Web page, http://www.w3.org/TR/rdf-schema/.

Rector, A., S. Brandt, N. Drummond, M. Horridge, C. Pulestin, et R. Stevens (2012). Engineering use cases for modular development of ontologies in OWL. *Applied Ontology 7*(2), 113–132.

RuleML 1.01 deliberation (2015). Deliberation RuleML 1.01. Web page, http://wiki.ruleml.org/index.php/Specification_of_Deliberation_RuleML_1.01.

SHACL (2017). Shapes Constraint Language (SHACL), W3C Recommendation 20 July 2017. Web page, http://www.w3.org/TR/shacl/.

Sowa, J. F. (1992). Conceptual Graphs Summary. In *Conceptual Structures : Current Research and Practice*, pp. 3–51. Ellis Horwood.

SPARQL 1.1 entailments (2013). SPARQL 1.1 Entailment Regimes, W3C Recommendation 21 March 2013. Web page, http://www.w3.org/TR/sparql11-entailment/.

Swan, R. (2016). *Querying Existential Rule Knowledge Bases : Decidability and Complexity*. PhD thesis, University of Montpellier.

Tao, J., E. Sirin, J. Bao, et D. L. McGuinness (2010). Integrity constraints in owl. In *AAAI'10*, pp. 1443–1448. AAAI Press.

Turtle (2014). RDF 1.1 Turtle (Terse RDF Triple Language), W3C Recommendation 25 February 2014. Web page, http://www.w3.org/TR/turtle/.

Zamazal, O. et V. Svátek (2015). PatOMat - Versatile Framework for Pattern-Based Ontology Transformation. *Computing and Informatics 34*(2), 305–336.

Zaveri, A., A. Rula, A. Maurino, R. Pietrobon, J. Lehmann, et S. Auer (2016). Quality Assessment for Linked Data : A Survey. *Semantic Web Journal 7*(1), 63–93.

Summary

This article defines prescriptive constraints as rules enabling inference engines to check that certain formal objects are used – not just inferred – or not, in certain conditions. It shows why these constraints require not exploiting inheritance mechanisms (or other mechanisms automatically adding relations to objects) during the tests of rule conclusions. It gives a general method to do this and then SPARQL commands to implement this method when the rules are represented via a subclassOf relation from the condition to the conclusion, i.e. as in OWL2-ER. The article illustrates these commands to check some ontology design patterns. More generally, the approach can be used to check the completeness of an ontology, or to represent constraints in an ontology (rather than via requests or *ad hoc* procedures) to calculate a degree of ontology completeness. The approach can thus help elicitation, modeling or validation of knowledge.