



HAL
open science

Import, export et traduction sémantiques génériques basés sur une ontologie de langages de représentation de connaissances

Jérémy Bénard

► To cite this version:

Jérémy Bénard. Import, export et traduction sémantiques génériques basés sur une ontologie de langages de représentation de connaissances. Théorie et langage formel [cs.FL]. Université de la Réunion, 2017. Français. NNT : 2017LARE0021 . tel-01761397

HAL Id: tel-01761397

<https://theses.hal.science/tel-01761397>

Submitted on 9 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE LA RÉUNION
ÉCOLE DOCTORALE SCIENCES TECHNOLOGIES ET SANTÉ

THÈSE

pour l'obtention du grade de

Docteur en Sciences

de l'Université de La Réunion

Mention : Informatique

Présentée et soutenue par

Jérémy Bénard

**Import, export et traduction sémantiques génériques
basés sur une ontologie de
langages de représentation de connaissances**

Thèse dirigée par : **Philippe MARTIN**

soutenue le 12 juin, 2017

Jury :

Rapporteurs :	Nada MATTA	Professeur Université de Technologies de Troyes
	Nhan LE THANH	Professeur Institut Universitaire de Technologies de Nice
Directeur :	Philippe MARTIN	Maître de Conférences, H.D.R. Université de La Réunion
Président :	Nhan LE THANH	
Examineurs :	Juliette DIBIE BARTHELEMY	Professeur AgroParisTech
	Rallou THOMOPOULOS	Chargé de Recherche, H.D.R. INRA Montpellier
	Anil CASSAM-CHENAÏ	Directeur de Global Technologies Holding

Résumé

Les langages de représentation de connaissances (LRCs) sont des langages qui permettent de représenter et partager des informations sous une forme logique. Il y a de nombreux LRCs. Chaque LRC a un modèle structurel abstrait et peut avoir plusieurs notations. Ces modèles et notations ont été conçus pour répondre à des besoins de modélisation ou de calculabilité différents, ainsi qu'à des préférences différentes. Les outils actuels gérant ou traduisant des RCs ne travaillent qu'avec quelques LRCs et ne permettent pas – ou très peu – à leurs utilisateurs finaux d'adapter les modèles et notations de ces LRCs. Cette thèse contribue à résoudre ces problèmes pratiques et le problème de recherche original suivant : “une fonction d'import et une fonction d'export de RCs peuvent-elle être spécifiées de façon générique et, si oui, comment leurs ressources peuvent-elles être spécifiées?”. Cette thèse s'inscrit dans un projet plus vaste dont l'objectif général est de faciliter le partage et la réutilisation des connaissances liées aux composants logiciels et à leurs présentations. L'approche suivie dans cette thèse est basée sur une ontologie de LRCs nommée KRLO, et donc sur une représentation formelle de ces LRCs.

KRLO a trois caractéristiques importantes et originales auxquelles cette thèse a contribué : i) elle représente des modèles de LRCs de différentes familles de façon uniforme, ii) elle inclut une ontologie de notations de LRCs, et iii) elle spécifie des fonctions génériques pour l'import et l'export de RCs dans divers LRCs. Cette thèse a contribué à améliorer la première version de KRLO (KRLO_2014) et à donner naissance à sa seconde version. KRLO_2014 contenait des imprécisions de modélisation qui rendaient son exploitation difficile ou peu pratique. Cette thèse a aussi contribué à la spécification et l'opérationnalisation de “Structure_map”, une fonction permettant d'écrire de façon modulaire et paramétrable toute autre fonction utilisant une boucle. Son utilisation permet de créer et d'organiser les fonctions en une ontologie de composants logiciels. Pour implémenter une fonction générique d'export basée sur KRLO, j'ai développé SRS (Structure_map based Request Solver), un résolveur d'expressions de chemins sur des RCs. SRS interprète toutes les fonctions. SRS apporte ainsi une validation expérimentale à la fois à l'utilisation de cette primitive (Structure_map) et à l'utilisation de KRLO.

Directement ou indirectement, SRS et KRLO pourront être utilisés par GTH (Global Technologies Holding), l'entreprise partenaire de cette thèse.

Mots clés : langage de représentation, connaissances formelles, ontologie, interopérabilité, traduction, export, sémantique, syntaxe.

Abstract

Knowledge Representation Languages (KRLs) are languages enabling to represent and share information in a logical form. There are many KRLs. Each KRL has one abstract structural model and can have multiple notations. These models and notations were designed to meet different modeling or computational needs, as well as different preferences. Current tools managing or translating knowledge representations (KRs) allow the use of only one or few KRLs and do not enable – or hardly enable – their end-users to adapt the models and notations of these KRLs. This thesis helps to solve these practical problems and this original research problem: “Can a KR import function and a KR export function be specified in a generic way and, if so, how can their resources be Specified?”. This thesis is part of a larger project the overall objective of which is to facilitate i) the sharing and reuse of knowledge related to software components, and ii) knowledge presentations. The approach followed in this thesis is based on an ontology of KRLs named KRLO, and therefore on a formal representation of these KRLs.

KRLO has three important and original features to which this thesis contributed: i) it represents KRL models of different families in a uniform way, ii) it includes an ontology of KRLs notations, and iii) it specifies generic functions for KR import and export in various KRLs. This thesis has contributed to the improvement of the first version of KRLO (KRLO_2014) and to the creation of its second version. KRLO_2014 contained modeling inaccuracies that made it difficult or inconvenient to use. This thesis has also contributed to the specification and the operationalization of “Structure_map”, a function enabling to write any other function that uses a loop, in a modular and configurable way. Its use makes it possible to create and organize these functions into an ontology of software components. To implement a generic export function based on KRLO, I developed SRS (Structure_map based Request Solver), a KR retrieval tool enabling the use of KR path expressions. SRS interprets all functions. SRS thus provides an experimental validation for both the use of this primitive (Structure_map) and the use of KRLO.

Directly or indirectly, SRS and KRLO may be used by GTH (Global Technologies Holding), the partner company of this thesis.

Keywords: representation language, formal knowledge, ontology, interoperability, translation, export, semantic, syntax.

Remerciements

En premier lieu, je tiens à remercier Anil Cassam-Chenai (le directeur du groupe GTH qui inclut les entreprises GTH, logiCells et Logicells Business Solutions), l'entreprise GTH, Philippe Martin, l'Université de La Réunion et l'ANRT pour m'avoir donné l'occasion de réaliser cette thèse en CIFRE.

Je remercie tout particulièrement Anil Cassam-Chenai pour le soutien technique qu'il m'a apporté pendant cette thèse et pour le soutien financier qu'il a apporté à la clôture de la CIFRE (qui a duré de 2013 à 2016). Je remercie aussi tous les autres employés du groupe GTH qui m'ont aidé directement ou indirectement dans le développement de SRS : Imran Sidat, Ulrich Francomme, Julien Boyer, Sébastien Fontaine, Mickaël Francomme.

Pour finir, je remercie tout particulièrement Philippe Martin pour sa patience et ses conseils.

1. <u>Introduction</u>	7
1.1. <u>Préambule</u>	7
1.2. <u>Projet global dans lequel s'inscrit cette thèse</u>	7
1.2.1. <u>Objectifs</u>	7
1.2.2. <u>Types d'approches générales actuelles pour les deux premiers objectifs</u>	7
1.2.3. <u>Type d'approche proposée pour le projet global dans lequel s'inscrit cette thèse</u>	8
1.3. <u>Une nouvelle approche pour l'import, l'export et la traduction de LRCs</u>	8
1.3.1. <u>Contexte et intérêt de cette thèse</u>	8
1.3.2. <u>Types d'approches actuelles pour la traduction de LRCs</u>	9
1.3.3. <u>Objectif</u>	10
1.3.4. <u>Problèmes de recherche</u>	11
1.3.5. <u>Approche proposée dans cette thèse</u>	11
1.4. <u>Publications</u>	12
1.5. <u>Notes de présentation</u>	13
Partie 1 : État de l'art	14
2. <u>Notions importantes utilisées dans cette thèse</u>	15
2.1. <u>Définitions</u>	15
2.2. <u>Difficulté des tâches de modélisation et/ou de programmation pour le partage de connaissances</u>	20
2.2.1. <u>Création d'un modèle conceptuel via un LRC</u>	20
2.2.2. <u>Création directe d'un modèle exécutable via un LRC exécutable</u>	21
2.2.3. <u>Création directe d'un modèle exécutable via un langage qui n'est pas un LRC</u>	21
2.2.3.1. <u>Le langage utilisé pour créer le modèle exécutable est un langage de description dont la structure peut être manipulée via des outils ou un langage</u>	22
2.2.3.2. <u>Le langage utilisé pour créer le modèle exécutable est un langage de programmation</u>	25
2.3. <u>Précisions et exemples pour l'homo-iconicité</u>	26
2.4. <u>Conventions et exemples de traductions</u>	27
3. <u>Approches classiques pour l'import, l'export et la traduction de LRCs</u>	31
3.1. <u>Import de connaissances</u>	31
3.1.1. <u>Import via un générateur d'analyseurs lexico-syntaxiques</u>	32
3.1.2. <u>Import via un générateur d'environnement de programmation interactif</u>	34
3.1.3. <u>Import via l'ajout d'extensions à un langage</u>	35
3.1.4. <u>Import exploitant des métalangages de description de structures de données</u>	36
3.1.5. <u>Import exploitant une ontologie de modèles abstraits de LRCs</u>	36
3.2. <u>Export de connaissances</u>	39
3.3. <u>Traductions</u>	41
3.3.1. <u>Spécifications de traductions directes</u>	43
3.3.2. <u>Traduction exploitant un langage pivot</u>	46
3.3.3. <u>Traduction via une "famille de langages"</u>	47
3.3.4. <u>Spécifications de traductions exploitant une ontologie de langage existante</u>	50
3.3.4.1. <u>Règles de traductions proposées par le W3C</u>	50
3.3.4.2. <u>Ontologie de langages de l'OMG</u>	50
3.3.4.3. <u>HETS avec LATIN et DOL</u>	51
3.4. <u>Bilan</u>	55

Partie 2 : Travail effectué	57
4. <u>KRLO, une ontologie de LRCs</u>	58
4.1. <u>Contexte</u>	58
4.1.1. <u>Idées clefs sous-jacentes</u>	58
4.1.1.1. <u>Une ontologie d'instruments de description</u>	58
4.1.1.2. <u>Opérateur, arguments et résultat</u>	59
4.1.1.3. <u>Présentation ordonnée des parties</u>	61
4.1.2. <u>Modèles abstraits de LRCs</u>	61
4.1.2.1. <u>Ontologie de haut niveau</u>	61
4.1.2.2. <u>Ontologies de modèles particuliers</u>	63
4.1.3. <u>Notations (modèles concrets) de LRCs</u>	65
4.1.3.1. <u>Relations entre notations et types d'éléments de notation de haut niveau</u>	65
4.1.3.2. <u>Spécifier des notations dans KRLO 2014</u>	67
4.1.3.3. <u>Spécifier des notations dans KRLO 2015+</u>	68
4.1.4. <u>Spécifications de notations d'éléments de modèles abstraits dans KRLO 2015+</u>	69
4.2. <u>Contributions à KRLO</u>	70
4.2.1. <u>Identification de problèmes de modélisation dans KRLO 2014</u>	70
4.2.2. <u>Spécifications de notations des éléments de modèle abstrait dans KRLO 2014</u>	73
4.2.3. <u>Spécifications de notations des éléments de modèle abstrait dans KRLO 2015+</u>	74
4.3. <u>Expressivité</u>	75
4.4. <u>Complétude de KRLO</u>	77
5. <u>Résolveur de requêtes développé pendant cette thèse</u>	78
5.1. <u>Raisons pour la création de ce résolveur et comparaisons avec d'autres résolveurs</u>	78
5.1.1. <u>But et comparaisons</u>	78
5.1.2. <u>Fonctions de parcours et de manipulation de structures de données</u>	85
5.2. <u>Mon implémentation de Structure_map</u>	91
5.3. <u>État des lieux et perspectives</u>	98
5.3.1. <u>État de l'implémentation</u>	98
5.3.2. <u>Perspectives pour SRS</u>	99
6. <u>Import, export et traduction de LRCs exploitant KRLO</u>	101
6.1. <u>Export depuis un modèle abstrait vers un modèle concret</u>	101
6.1.1. <u>Spécifications d'exports</u>	101
6.1.1.1. <u>Fonctions primitives utilisées par les spécifications d'export</u>	101
6.1.1.2. <u>Fonction d'export par défaut spécifiée dans KRLO</u>	102
6.1.1.3. <u>Fonction d'export spécifiée via Structure_map</u>	103
6.1.1.4. <u>Comparaison des spécifications</u>	104
6.1.2. <u>Validation, mise en œuvre et exemples</u>	106
6.1.3. <u>Complétude</u>	
6.2. <u>Traduction entre éléments abstraits</u>	113
6.2.1. <u>Nécessité d'une traduction entre EAs</u>	113
6.2.2. <u>Traduction entre Frame_as_NR-phrase et Conjunction_of_links_from_a_same_source</u>	114
6.2.3. <u>Traduction entre une relation binaire et une relation n-aire</u>	116
6.3. <u>Import depuis un modèle concret vers un modèle abstrait</u>	117
6.3.1. <u>Principe et spécifications</u>	117
6.3.2. <u>Complétude</u>	120
7. <u>Conclusion</u>	121
7.1. <u>Contributions scientifiques apportées par cette thèse</u>	121
7.2. <u>Solutions apportées aux problèmes de recherche</u>	122
7.3. <u>Perspectives</u>	123
7.3.1. <u>Perspectives scientifiques</u>	123
7.3.2. <u>Perspectives pour GTH</u>	123

<u>Références</u>	124
<u>Lexique</u>	128
<u>Annexes</u>	132
<u>Annexe 1 : quelques fonctions de parcours et de manipulation de structures de données</u>	132
<u>Annexe 2 : quelques fonctions du framework LAS représentées via Structure_map</u>	136
<u>Annexe 3 : documentation de BW Notation</u>	142
<u>Annexe 4 : documentation de la version 2 de RichGraph</u>	145

1. Introduction

1.1. Préambule

Le rapport fournit deux versions d'un lexique des principales notions utiles dans cette thèse. Le contenu de ces deux versions est identique mais ordonné différemment. La section [2.1](#) présente la version ordonnée par thématique, la section "[Lexique](#)" présente la version ordonnée par ordre alphabétique.

De décembre 2012 à fin novembre 2015, cette thèse a été réalisée dans le cadre d'un dispositif CIFRE (Convention Industrielle de Formation par la REcherche) avec un partenaire industriel, le groupe GTH (Global Technologies Holding). Ce groupe inclut les trois entreprises suivantes.

- GTH (Global Technologies Holding) est une *société faitière [Holding] mixte*, i.e., une société qui i) détient les actions d'autres sociétés afin de former un groupe, et ii) conserve une activité qui lui est propre. En l'occurrence, GTH a conservé une activité de R&D (Recherche et Développement) et son produit principal est RichGraph, une structure de données pour représenter des graphes qui est présentée dans la section [5.1.1](#).
- logiCells est une entreprise d'édition de logiciels qui développe un progiciel de gestion intégré [ERP : Enterprise Resource Planing] *sémantique* nommé logiCells Application Server (LAS). Le site web de logiCells est accessible à l'adresse suivante : <http://www.logicells.com/>.
- LBS (logiCells Business Solutions) est une entreprise d'édition de logiciels qui développe des modules métier pour LAS.

De décembre 2015 à juin 2017, cette thèse a été financée par le groupe GTH avec l'aide du CIR (Crédit Impôt/Recherche).

Le sujet initial "Construction collaborative d'ontologies pour évaluer ou indexer des services ou créer des programmes complètement paramétrables par leurs utilisateurs" s'est avéré significativement plus complexe que Philippe Martin et moi même ne l'avions prévu. Le CST a accepté que le sujet de thèse initial soit restreint. Le sujet de thèse retenu est "Import, export et traduction sémantiques génériques basés sur une ontologie de langages de représentation de connaissances".

Le projet circonscrit par le sujet initial de cette thèse n'est cependant pas abandonné. D'une part, ce projet sera poursuivi après la fin de la thèse, il est brièvement décrit dans la section [1.2](#), ci-dessous. D'autre part, j'ai pu réutiliser mon travail sur Structure_map – une fonction qui permet de représenter et organiser toute autre fonction qui utilise une boucle – pour concevoir et implémenter un résolveur de requête hautement paramétrable. La section "[5](#). Résolveur de requêtes développé pendant cette thèse" décrit ce résolveur et ses paramètres.

Le sujet actuel de cette thèse est présenté dans la section [1.3](#), ci-après.

1.2. Projet global dans lequel s'inscrit cette thèse

1.2.1. Objectifs

1. Faciliter la recherche et l'exploitation de connaissances liées à la structure des programmes et aux langages pour les écrire – c'est à dire, les langages de représentation de connaissances (LRCs) ou les langages de programmation – sans limiter l'expressivité de ces connaissances. C'est ici que se situe cette thèse dans le cadre du projet global.
2. Faciliter la recherche de composants logiciels indexés sémantiquement selon des critères plus complexes que la structure de ces composants. Ces critères peuvent être des critères de qualité logicielle, par exemple, l'extensibilité ou la facilité d'utilisation.
3. Faciliter la construction collaborative d'ontologies i) par des agents sans perte d'informations et sans consensus sur la terminologie à adopter, et ii) via des échanges de connaissances entre des serveurs pour créer une "base de connaissances globale virtuelle" sans recourir à un système de centralisation.

1.2.2. Types d'approches générales actuelles pour les deux premiers objectifs

Pour faciliter la recherche des connaissances relatives aux programmes, les principales techniques actuellement utilisées sont les suivantes.

1. Comparaison lexicale entre des mots, comme le font la plupart des moteurs de recherche de documents ou de parties de documents. Certains moteurs, comme Google, utilisent quelques représentations de connaissances (RCs) d'expressivité très faible, ce qui assure de bonnes performances au détriment de la précision (et de la complétude) des résultats.

- Utilisation d'annuaires stockant les buts, signatures et manières d'appeler les procédures de services informatiques (comme dans CORBA ou UDDI). Ces annuaires sont généralement décrits dans des langages qui ne sont pas des LRCs, ce qui ne facilite pas la comparaison (et donc la recherche) des services ainsi indexés.

1.2.3. Type d'approche proposée pour le projet global dans lequel s'inscrit cette thèse

L'approche utilisée dans ce projet (et dans cette thèse) est d'utiliser de façon systématique des ontologies. Les ontologies suivantes sont nécessaires.

- **Une ontologie – FLO (Formal Language Ontology) – qui représente et organise des langages formels (LRCs et langages de programmation) et leurs composants, ce qui inclut les composants logiques de base (par exemple, les prédicats et les quantificateurs).** La section “4. KRLO, une ontologie de LRCs” présente KRLO l'ontologie pour les LRCs à laquelle j'ai contribué. Des agents peuvent exploiter FLO pour comparer des éléments de langages, par exemple, afin de les traduire. FLO a deux sous-parties : une ontologie des modèles abstraits et une ontologie des modèles concrets.
- **Une ontologie qui organise des composants logiciels de base.** Des agents peuvent exploiter cette ontologie pour rechercher, assembler, paramétrer ces composants logiciels. De plus, via FLO, ces composants peuvent être convertis dans divers langages, y compris des LRCs. Lorsqu'ils sont convertis dans des LRCs, le résultat de la conversion est une généralisation du composant en entrée, définie par des spécifications formelles déclaratives. Les travaux liés à cette ontologie pourraient intéresser à la fois la communauté de l'ingénierie dirigée par les modèles et la communauté de la représentation des connaissances. Ils pourraient donc être publiés – entre autres – dans FOIS (Formal Ontology in Information Systems) ou ICSE (International Conference on Software Engineering).
- **Une ontologie de critères pour l'évaluation de composants logiciels ou de services.** Des agents peuvent exploiter cette ontologie pour comparer des composants logiciels suivant des critères complexes. Par exemple, cette ontologie peut être exploitée pour comparer certains attributs de qualité logicielle de composants. Ainsi, il est possible d'évaluer la facilité de ré-utilisation des composants ou certains éléments liés à la sécurité comme le niveau de confidentialité d'un service. Les travaux liés à cette ontologie pourraient également être publiés FOIS ou ICSE.

Afin de faciliter la collaboration pour la construction de ces ontologies, [Martin, 2009c] et [Martin, 2010d] décrivent les protocoles suivants.

- Un protocole d'édition qui permet à plusieurs utilisateurs de créer de façon collaborative une seule ontologie. Ce protocole conserve la base cohérente et bien organisée sans restreindre ce que chaque utilisateur souhaite entrer.
- Un protocole qui permet de créer une seule “base de connaissances virtuelle” à partir de plusieurs serveurs de bases de connaissances, sans recourir à un système de centralisation, via la répllication d'informations et de requêtes entre les serveurs de BCs.

1.3. Une nouvelle approche pour l'import, l'export et la traduction de LRCs

1.3.1. Contexte et intérêt de cette thèse

Les LRCs permettent de représenter des informations sous une forme logique – les RCs – dans des bases de connaissances (BCs). Les RCs peuvent être exploitées par des moteurs d'inférences ou par des systèmes de gestion de BCs. Ainsi, les RCs facilitent la recherche de connaissances et la résolution de problèmes. Le W3C a popularisé l'intérêt d'utiliser et d'interconnecter des “RCs sur le web”. Le web sémantique est l'ensemble des RCs qui sont écrites via les LRCs promus par le W3C.

Un LRC a un *modèle abstrait*. *Modèle abstrait* désigne ici un type abstrait de données, pas un modèle de la théorie des modèles. Par exemple, RDF est un modèle conçu par le W3C tandis que Common Logics (CL) [CL, 2007] est un modèle conçu par l'ANSI. Un modèle abstrait est un modèle abstrait structurel comme les modèles ou méta-modèles utilisés en ingénierie dirigée par les modèles (IDM) [MDE]. Un modèle abstrait suit une logique, par exemple, la logique du premier ordre ou la logique de description SHOIN(D). Un modèle abstrait peut être présenté avec différentes notations, par exemple, Terse RDF Triple Language (Turtle) ou une notation basée sur XML. Les notations sont aussi appelées modèles concrets, syntaxes concrètes ou présentations formelles. Dans cette thèse, “modèle” réfère à un modèle abstrait et “notation” ou “modèle concret” réfèrent à une notation.

Beaucoup de LRCs existent. Un LRC unique ne serait approprié ni pour tout type de modélisation ou d'exploitation de connaissances, ni pour toute personne ou outil. Un LRC expressif avec une notation riche et concise est utile pour la modélisation et le partage d'informations complexes telle que i) le contenu de certaines phrases en langage naturel, ou ii) une ontologie qui définit des types de concepts ou de relations. Dans la phase de modélisation, pour des raisons d'interopérabilité et de lisibilité, il peut être nécessaire d'utiliser des LRCs avec une notation de second ordre et du sucre syntaxique pour des meta-phrases et des quantificateurs numériques. D'un autre côté, pour la phase d'opérationnalisation, des LRCs peu expressifs peuvent être plus faciles à apprendre et assurent de bonnes performances pour la résolution de problèmes.

Le W3C propose plusieurs LRCs. L'expressivité de ces LRCs est réduite de sorte que leur calculabilité soit inférieure à celle de la logique d'ordre 1. Pour la phase de modélisation, des restrictions d'expressivité sont généralement peu profitables. En effet, à cause de ces restrictions, certaines connaissances ne peuvent pas être représentées de façon précise ou correcte. Ainsi, pour ces connaissances, ces restrictions conduisent à créer des représentations biaisées, incorrectes ou incomplètes. Les RCs biaisées, incorrectes ou incomplètes pourront alors difficilement être réutilisées. De plus, certains outils de recherche qui utilisent uniquement des techniques simples de reconnaissance de spécialisation de graphe [pattern matching] “plutôt que des techniques de déduction complètes” peuvent être utilisés pour rechercher des RCs complexes même s'ils ne peuvent faire que peu d'inférences sur ces RCs.

La traduction de RCs décrites dans des LRCs différents permet de faciliter la réutilisation de ces RCs et d'améliorer l'interopérabilité des systèmes qui les exploitent.

Cette thèse propose une approche originale pour traduire des *phrases décrites dans un LRC* vers des *phrases décrites dans un autre LRC*. Une telle traduction ne concerne que les LRCs (leur syntaxe, leur sémantique) et donc pas le contenu des phrases – c'est à dire, les objets et les relations qui sont décrits par cette phrase. Une telle traduction ne peut pas être utilisée seule pour des traductions de contenus, par exemple, pour l'alignement d'ontologie. En revanche, lorsque les ontologies à aligner sont décrites dans des LRCs différents, une traduction de LRCs peut être utilisée pour faciliter l'alignement. Peu de travaux proposent des approches pour la traduction de LRCs. Certains de ces travaux – par exemple, [Chalupsky, 2000] ou [Corcho & Gomez-Perez, 2007] – proposent des langages pour spécifier des traductions entre ces LRCs. D'autres travaux – par exemple, [Gruber, 1993] ou [Euzénat & Stuckenschmidt, 2003] – proposent des stratégies pour réduire le nombre de traductions entre LRCs à spécifier. Aucun travail antérieur à celui qui est présenté dans cette thèse ne propose une approche générique pour la traduction de LRCs. La conception de cette approche générique a nécessité beaucoup de travail, ce qui explique le nombre relativement peu important de publications. La section 1.3.4 donne plus de détails sur ce qui fait l'originalité de l'approche proposée dans cette thèse et sur les avantages de cette approche.

1.3.2. Types d'approches actuelles pour la traduction de LRCs

La traduction entre LRCs peut être décomposée en trois processus : i) import, ii) traduction entre éléments de modèle abstraits, et iii) export.

Les approches pour l'import et l'export sont basées sur des langages de programmation et exploitent des fonctions conçues pour importer vers un modèle abstrait particulier ou exporter vers une notation particulières. Par exemple, EasyRDF est un outil accessible à <http://www.easyrdf.org/converter> qui exploite plusieurs fonctions d'export et d'import écrites en PHP. Pour chaque nouvelle notation, une nouvelle fonction d'import et une nouvelle fonction d'export doit être écrite. De plus, ces fonctions sont très peu paramétrables par les utilisateurs du service et ne peuvent être utilisées que pour le modèle RDF.

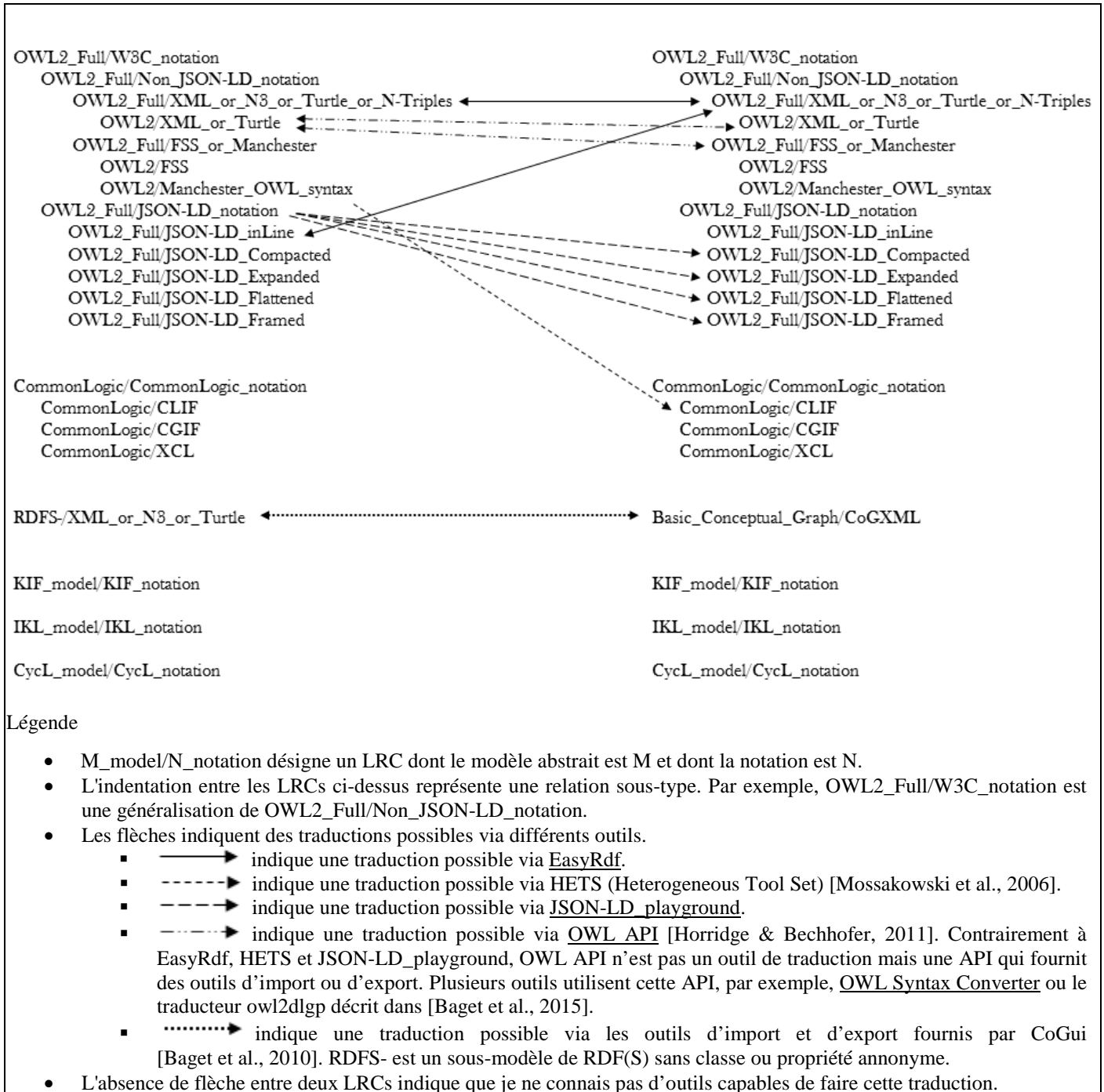
Les approches pour la traduction entre modèles abstraits i) ne proposent pas de solution pour l'import et l'export, et ii) exploitent des fonctions de traduction qui sont conçues pour traduire entre des modèles particuliers.

Un outil de traduction générique sans perte d'information et non lié à une implémentation particulière permettrait d'éviter l'usage de standards ou le développement fonctions de traductions pour des couples de langages. La liste suivante détaille les problèmes que posent i) le développement de fonctions de traductions pour des couples de langages, et ii) l'usage de standards.

- i. Problèmes posés par le développement de fonctions de traductions pour des couples de langages
 - Certains agents utilisent en interne des langages adaptés à leurs besoins. Ces langages peuvent être des langages de description de structure, des LRCs non standards ou des versions spéciales de LRCs standards. Par exemple, Wikibase utilise en interne un langage non standard (cf. <https://www.mediawiki.org/wiki/Wikibase/DataModel>) et Yago utilise en interne des triplets RDF réifiés et uniquement des relations temporelles et spatiales sur ces réifications.
 - Le développement d'outils de traductions entre des langages est coûteux. En effet, pour n langages, il y a n^2 fonctions de traduction à développer et donc n^2 fonctions d'import et n^2 fonctions d'export (cf. “3.3.1. Spécifications de traductions directes”). C'est pourquoi les agents se limitent généralement au développement de traductions vers quelques standards. Actuellement, ces standards servent de langages pivot. Ainsi, le nombre de traductions à spécifier reste élevé : $2*n$, n étant le nombre de langages, cf. “3.3.2. Traduction exploitant un langage pivot”.
- ii. Problèmes posé par l'usage de standards
 - Aucun standard de LRC actuel n'a toutes les qualités pour faire un bon langage pivot complet, c'est à dire, représenter et normaliser toutes sortes de connaissances. La sous-liste suivante illustre deux problèmes majeurs de ces standards.
 - **Faible expressivité**
Les LRCs standards actuels – en particulier les LRCs du W3C autres que RIF-FLD – sont trop peu expressifs pour la représentation de connaissances dans le cas le plus général. Par exemple, ils sont trop peu expressifs pour être utilisés afin de représenter le contenu de certaines phrases en langage naturel.
 - **Pas d'ontologie de modélisation**
Les standards de LRC ne fournissent pas d'ontologie de modélisation pour aider les concepteurs à choisir une représentation plutôt qu'une autre. Ainsi, ces concepteurs peuvent être amenés à faire des choix de représentation arbitraires qui sont sources d'ambiguïtés et de problèmes d'interopérabilité [Guizzardi et al., 2010].

- Les organismes de normalisation ou les chercheurs proposent régulièrement de nouveaux standards ou extensions de standards. Par exemple, OWL2 est une extension de OWL ; CL a été conçu avec l'intention de remplacer KIF ; IKL est une extension de CL/CLIF. Actuellement, peu d'outils peuvent faire des traductions entre les modèles des LRCs standards. Par ailleurs, aucun outil ne peut utiliser toutes les notations communes. Par exemple, même EasyRdf – qui est le plus complet des traducteurs pour les LRCs de W3C – i) ne peut être utilisé que pour des LRCs dont le modèle est OWL+RDF, et ii) ne peut traiter que quelques unes des notations proposées par le W3C. La figure suivante montre que plusieurs LRCs ne peuvent pas être traduits entre eux.

Figure 1.1. Quelques traductions existantes entre des LRCs



1.3.3. Objectif

L'objectif général de cette thèse est de faciliter le partage et l'exploitation de connaissances impliquant plusieurs LRCs, par exemple, lorsque ces langages ont des expressivités et des notations différentes. Faciliter ce partage et cette exploitation permet de faciliter plusieurs types de cas d'utilisations. Les trois cas suivant ont retenu notre attention.

1. **Implémentation d'outils exploitant plusieurs LRCs.**
2. **Adaptation des LRCs et des fonctions ou paramètres de présentation de RCs par les utilisateurs finaux.**
3. **Comparaison et évaluation des LRCs vis à vis de critères ou de bonnes pratiques.**

Le partage et l'exploitation de connaissances impliquant plusieurs LRCs nécessite des traductions entre LRCs et donc également la spécification de fonctions d'import, d'export ou de transformations entre modèles abstraits. Aucun outil de traduction actuel et aucune combinaison de tels outils ne permet de traduire depuis et vers tout LRC. En effet, via les approches actuelles (cf. chapitre 3, “Approches classiques pour l'import, l'export et la traduction de LRCs”), i) les importeurs et exporteurs sont spécifiques à un LRC particulier, et ii) les transformateurs nécessitent une spécification de transformation pour chaque paire de modèle, de plus, pour chacune de ces spécifications, le programmeur ou l'utilisateur doit spécifier beaucoup de règles.

Pour faciliter les traductions entre LRCs, la solution proposée dans cette thèse est de spécifier – via des fonctions génériques – l'import, l'export et les transformations entre quelques types d'éléments de modèles abstraits (EAs) généraux. Pour spécifier de telles fonctions ainsi que leurs entrées et sorties, cette thèse préconise l'utilisation d'une ontologie représentant des LRCs de manière homogène. La section 1.3.5 donne plus de détails sur l'approche proposée dans cette thèse.

Le point 3 (“Comparaison et évaluation des LRCs vis à vis de critères ou de bonnes pratiques”) nécessite une ontologie homogène de LRCs. L'évaluation de la structure des RCs nécessite aussi un outil qui peut les parser et i) effectuer des commandes sur les AEs pour ces RCs, ou ii) exporter ces AEs, i.e., leurs structures, dans un LRC cible donné. Ce dernier cas, qui est possible aussi avec nos outils, permet l'utilisation d'un autre moteur d'inférences pour vérifier la structure des RCs liées aux critères ou bonnes pratiques. Ceci peut par exemple être utile pour sélectionner des ressources RCs ou pour vérifier les compétences d'étudiants en gestion des connaissances. Philippe Martin et moi avons effectué une synthèse et une extension de certaines bonnes pratiques liée à la structure des RCs [Martin & Bénard, 2017a].

1.3.4. Problèmes de recherche

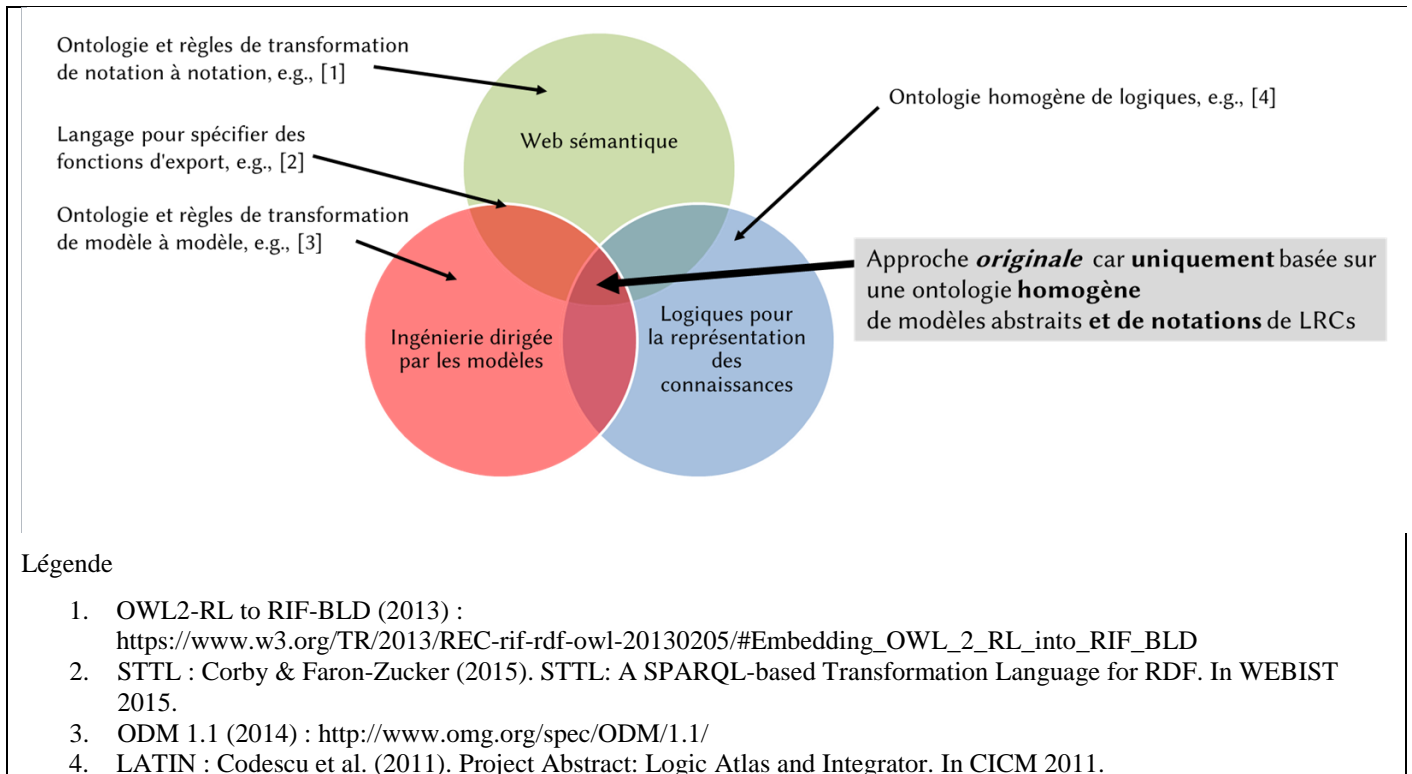
1. Comment spécifier une fonction *d'import* et ses entrées pour que cette fonction soit générique (c'est à dire, utilisable pour n LRCs définis de manière adéquate) ?
2. Comment spécifier une fonction *d'export* et ses entrées (définitions de types ou, alternativement à ces définitions, des règles ou des fonctions) pour que cette fonction soit générique ?
3. Comment spécifier ces fonctions et/ou leurs entrées de sorte qu'elles soient suffisantes pour des traductions entre LRCs ? Ceci implique aussi de déterminer si une sous-fonction (générique) de traduction entre des éléments de modèles abstraits de LRCs est utile ou si une définition précise des entrées suffit.

La résolution de ces questions permet d'utiliser n'importe quel LRC avec les mêmes avantages que s'il était un standard ayant un traducteur depuis/vers les autres LRCs qui sont définis par rapport à ce standard. Comme expliqué dans la prochaine section, dans notre approche, les LRCs sont définis dans une ontologie d'une manière particulière qui permet la traduction entre tous ces LRCs (quelle que soit leur expressivité grâce à, si besoin, des extensions de leur modèle). Ainsi, les problèmes des standards ne sont plus des obstacles au partage aisé (passant à l'échelle) de connaissances.

1.3.5. Approche proposée dans cette thèse

L'approche proposée dans cette thèse est entièrement basée sur une ontologie de LRCs : KRLO. Comme le montre la figure ci-après, cette approche se positionne à l'intersection de trois grandes communautés. Ce rapport de thèse présente KRLO et son exploitation pour la traduction de LRCs. En effet, outre des LRCs et des éléments de LRCs, KRLO a la particularité de permettre la spécification de fonctions génériques d'import et d'export. Actuellement, KRLO spécifie déclarativement i) deux fonctions pour exporter des éléments de modèle abstrait vers des notations textuelles, et ii) deux fonctions d'imports. L'une de ces fonctions d'import est déclarative, l'autre est implémentée en Javascript. Ces fonctions d'import seront testées pendant l'année 2017.

Figure 1.2. Positionnement de mes travaux par rapport à trois grandes communautés



Puisque l'approche présentée dans cette thèse est entièrement basée sur une ontologie, elle permet de gagner en généricité et en flexibilité par rapport toute autre approche qui n'est pas basée sur une ontologie. Une telle autre approche pourrait cependant garantir de meilleures performances, par exemple en vitesse d'exécution. En effet, dans un programme, *l'exploitation d'une ontologie plutôt que de structures de données* est analogue à *l'utilisation de variables plutôt que de constantes*. Par exemple, via l'assignation de valeurs à des variables, un développeur peut paramétrer des fonctions et ainsi réutiliser du code, ce qui est impossible dans un programme n'utilisant que des constantes.

L'approche proposée dans cette thèse s'appuie sur l'hypothèse suivante : “La traduction d'une phrase depuis un LRC vers un autre ne nécessite aucun paramètre lié au contenu de cette phrase”. L'approche proposée dans cette thèse n'est donc pas conçue pour la traduction ou l'alignement d'ontologies.

L'approche proposée dans cette thèse a été évaluée expérimentalement pour l'export :

- de façon procédurale via un outil de traduction – accessible à l'adresse suivante <http://kr-translation.logiccells.net/> – qui prend en paramètre une ancienne version de KRLO, cet outil est décrit dans la section 6.1.2 ;
- de façon déclarative via le résolveur de requêtes présenté dans la section 5.2 pour résoudre les fonctions d'export présentées dans la section 6.1.1.3.

Le principal avantage de l'approche proposée dans cette thèse est de permettre la spécification d'extensions à des LRCs existants ou de nouveaux LRCs, par des utilisateurs et sans tâche de programmation. Lorsque les outils listés ci-dessus seront plus rapides, ils pourront être utilisés pour traduire KRLO ou d'autres ontologies (aussi grandes ou plus grandes) vers d'autres LRCs. Avec de telles traductions de KRLO, les éléments qui suffisent pour exploiter toute base de connaissances sont un moteur d'inférences, un interpréteur d'un LRC et un résolveur de requêtes suffiront.

Les points durs que j'ai rencontré au cours de cette thèse étaient i) trouver les problèmes de KRLO_2014, ii) implémenter SRS, et iii) trouver le principe de la fonction d'import.

1.4. Publications

Les travaux présentés dans ce rapport de thèse ont été en partie publiés dans les articles suivants.

1. Martin Ph. & Bénéard J. (2017b). *Creating and Using various Knowledge Representation Model and Notation*. Proceedings of ECKM 2017, 18th European Conference on Knowledge Management, 7-8 September 2017, Barcelona, Spain.
2. Martin Ph. & Bénéard J. (2016b). *Top-level Ideas about Importing, Translating and Exporting Knowledge via an Ontology of Representation Languages*. ACM proceedings of Semantics 2016 (doi: [10.1145/2993318.2993324](https://doi.org/10.1145/2993318.2993324); pp. 89-92), Leipzig, Germany, 12th to 17th September, 2016.

3. Bénard J. & Martin Ph. (2015). *Improving General Knowledge Sharing via an Ontology of Knowledge Representation Language Ontologies*. Chapter 23 (pp. 364-387: 22 pages) of CCIS 553: book from the Springer-Verlag Lectures Notes series "Communications in Computer and Information Science" (CCIS). Book title: "Knowledge Discovery, Knowledge Engineering and Knowledge Management".
This book chapter is an extension of our "KEOD+KDIR 2014 best paper award" article listed below (selection rate: 12,9% -- 37 papers out of 287 submissions).
4. Martin Ph. & Bénard J. (2014). *An Ontology for Specifying and Parsing Knowledge Representation Structures and Notations*. Proceedings of KEOD 2014 (6th International Conference on Knowledge Engineering and Ontology Development, Rome, Italy, 21-24/10/2014, ISBN: 978-989-758-049-9), pp. 96-107 (-> "Full paper"). Selected for the "KDIR 2014 best paper award". KEOD and KDIR (International Conference on Knowledge Discovery and Information Retrieval) are joint conferences. Selection rate of "Full papers" at KEOD 2014: 18% (78 submissions).

Par ailleurs, dans le cadre de cette thèse, j'ai aussi été le co-auteur de l'article suivant. Les fonctions ou règles décrites dans cet article seront intégrées dans KRLO.

1. Martin Ph. & Bénard J. (2017a). *Categorizing or Generating Relation Types and Organizing Ontology Design Patterns*. Proceedings of KAM 2017, 23rd IEEE Conference on Knowledge Acquisition and Management, 3-6 September 2017, Prague, Czech Republic.
2. Martin Ph. & Bénard J. (2016a). *Deriving Binary Relation Types From Concept Types*. Supplementary proceedings of ICCS 2016, pp. 9-12, 22nd International Conference on Conceptual Structures, Annecy, France, 5th to 7th July, 2016.

1.5. Notes de présentation

Dans ce mémoire, les types de relation et les points importants seront mis en italique. Les relations structurelles seront mises entre guillemets. À la première occurrence, les références vers le lexique auront une police différente telle que pour le mot suivant : référence. Afin de clarifier la lecture, d'autres occurrences de ces références, donc pas uniquement la première, seront souvent aussi écrites dans cette police. Les hyperliens sont soulignés. Pour des raisons de clarté, certains titres de sections ne figurent pas dans la table des matières, ces sections ont une parenthèse plutôt qu'un point après leur numéro.

Partie 1 : État de l'art

2. Notions importantes utilisées dans cette thèse

Ce chapitre présente les différentes sortes de représentations de connaissances (RCs) à prendre en compte dans l'import, l'export et la traduction de RCs, c'est à dire, les différentes sortes de RCs utilisées dans la conception d'un modèle conceptuel ou exécutable d'une base de connaissances (BC) pour le partage de connaissances ou pour créer un logiciel utilisant une BC.

La section “2.1. Définitions” présente un lexique des termes importants utilisés dans le rapport ; une autre version de ce lexique – ordonnée alphabétiquement – est présentée en fin de rapport, après les références bibliographiques.

La section “2.2. Difficulté des tâches de modélisation et/ou de programmation pour le partage de connaissances” présente les différentes approches utilisées pour créer un modèle conceptuel ou exécutable de BC. Des RCs précises sont plus faciles à exploiter et donc à traduire. Pour les approches dans lesquelles la précision des RCs est limitée, l'utilisation d'un langage homo-iconique (cf. définition de l'homo-iconicité dans le lexique ci-dessous) peut permettre de faciliter quelque peu l'exploitation de ces RCs.

La section “2.3. Précisions et exemples pour l'homo-iconicité” présente une définition plus complète (que celle présentée dans le lexique) de la notion d'homo-iconicité. Via un exemple, elle illustre également les différences entre un langage homo-iconique et un langage non homo-iconique.

La section “2.4. Conventions et exemples de traductions” présente les deux principales notations utilisées dans ce rapport et illustre la difficulté des traductions entre LRCs via un exemple.

2.1. Définitions

La terminologie utilisée dans cette thèse est dérivée des terminologies utilisées par les communautés de recherche dans la représentation des connaissances, les approches basées sur les modèles et l'analyse de langages formels. Les définitions présentées proviennent de [Barwise & Perry, 1983], [Sowa, 1984], de CommonKADs [Breuker & van de Velde, 1994] et des documents du W3C relatifs à RDF, OWL et RIF. Ces définitions sont suffisantes dans le cadre de cette thèse. La plupart servent à éviter des répétitions dans l'état de l'art.

Ce lexique permet d'alléger la présentation de ce mémoire et de la rendre plus claire.

Chose [thing] : tout ce à quoi quelqu'un peut penser est une chose. Une chose peut avoir une description (cf. contenu de description, instrument de description, conteneur de description).

Situation : toute chose qui “arrive” dans une région réelle ou imaginaire de l'espace et du temps, par exemple, un processus ou un état.

Entité : chose qui n'est pas une situation qui peut participer à une situation.

Référence : identifiant, variable ou fonction.

Terme : valeur (par exemple, une valeur atomique comme un booléen ou un caractère, ou une valeur structurée comme une collection) ou une référence.

Phrase [(declarative or not) sentence] : une combinaison de termes et de sous-phrases décrivant une chose.

Description [(declarative) sentence, e.g., (logic) statement] : une phrase déclarative qui décrit le monde, par opposition aux phrases qui prescrivent ce qui peut être entré, par exemple dans les bases de données.

Définition d'un terme T : phrase quantifiant universellement T mais, contrairement à une observation, une définition n'est ni vraie ni fausse, elle ne fait que définir un terme par conditions nécessaires et/ou suffisantes.

Primitive : terme n'ayant pas de définition.

Relation *specialisation* : relation *subtype* ou *instance*.

Méta-phrase : phrase dont au moins un objet est une phrase. Par exemple, “John pense que 'il fait beau'”, “il existe une pensée qui a pour agent john et pour objet 'il fait beau'”, “Tom#birds_fly = John#"birds fly"” (ici la méta-phrase a deux objets qui sont des phrases reliées par la relation “=”) et “Cette phrase est fausse”.

Méta-phrase contextualisante : méta-phrase spécifiant des conditions pour que sa ou ses phrases objets sont vraies.

Contexte d'une phrase : toutes les méta-phrases qui directement ou non, la contextualisent.

Une sémantique : un sens, c'est à dire, une signification.

Relation structurelle : désigne la relation “parent/child” (généralement implicite) ou “attribut” dans i) les objets/éléments des langages orientés objets (e.g., Java), de description de documents (e.g., XML) ou de bases de données, ou ii) les structures de données comme les tableaux associatifs ou les “enregistrements” [records]. Contrairement aux relations sémantiques des LRCs, les relations structurelles ne sont pas des entités de 1er ordre : elles ne peuvent être référées, elles n'ont pas de types (donc pas de sémantiques explicites exploitables automatiquement) et ne peuvent donc pas être organisées par des relations *subtype*.

L'utilisation de relations sémantiques (voir définition ci-après) plutôt que structurelles a des avantages qui sont décrits entre autres sur cette page : <https://www.w3.org/DesignIssues/RDF-XML.html>. Ainsi, le W3C recommande l'utilisation de RDF plutôt que de XML pour le Web Sémantique.

Langage de description de structures de données : langage permettant d'écrire des termes et des relations structurelles entre ces termes. XML est un langage de description de structures de données.

Représentation de connaissances (RC) : phrase ayant une interprétation dans une logique et donc interprétable par un moteur d'inférences logique.

Langage de représentation de connaissances (LRC) : langage permettant d'écrire des **représentations de connaissances (RCs)**.

Langage de description : LRC ou langage de description de structures de données.

LRC exécutable : par définition, tous les LRCs ont une sémantique connue. Des moteurs d'inférence peuvent donc être construits pour interpréter ces LRCs. Dans cette thèse, nous appelons "LRC exécutable" un LRC dans lequel les problèmes de décision ont une complexité inférieure ou égale aux problèmes qui peuvent être exprimés en OWL2-DL (logique SHOIN(D)). Ceci inclut OWL-DL et les langages de programmation logique (voir la définition ci-après), e.g., Prolog, Datalog, RIF-BLD.

Langage déclaratif : langage via lequel tout processus ou programme peut être décrit sans description de son flux de contrôle.

Structure de données : ensemble de relations structurelles entre des données. Par exemple, un tableau est une structure de données.

"Type" de structure de données : spécification de relations structurelles entre des données depuis une même source. Par exemple, en XML, un "type" de structure de données peut être décrit dans une DTD ; dans un langage orienté objet, une classe est un "type" de structure de données.

"Instance" de structure de données : structure de données conforme à un "type" de structure de données, i.e., les relations structurelles de "l'instance" ont le même nom que les relations structurelles du "type". Par exemple, en Java, une "instance" de la classe *Vector* a la relation structurelle "elementCount" ainsi que toutes les autres relations structurelles spécifiées dans cette classe.

Description procédurale ou fonctionnelle : ensemble de descriptions de structures de données et de traitements (fonctions ou procédures) sur ces structures de données. Si un langage impératif – comme Java – ou fonctionnel – comme Haskell – est utilisé, une description procédurale peut inclure une description du flux de contrôle des traitements.

Programme : une dafp d'un processus – par exemple, via le langage Java – par opposition à une description déclarative d'un processus, par exemple, via un réseau de Pétri ou des règles. Dans tous les cas, ces descriptions de processus sont des ensembles de phrases dont au moins un élément est un processus. Une description fonctionnelle est une dafp de processus écrite dans un paradigme fonctionnel, typiquement via un langage de programmation fonctionnel, comme Haskell, par exemple.

Langage de programmation : langage permettant d'écrire des programmes.

Langage de programmation logique : langage permettant d'écrire un programme avec des expressions logiques, ce programme peut alors être exécuté via un algorithme de recherche de preuves. Prolog est un exemple de langage de programmation logique.

Objet d'information [informational object, resource] : phrase ou référence à une chose. Cette chose peut être une "ressource Web" si elle est référençable par un URI.

Objet lexicalement formel : objet d'information lexicalement unique dans les fichiers où il est déclaré ou utilisé. Un URI est un objet lexicalement formel car c'est un terme unique à l'échelle du Web.

Sémantique opérationnelle : une sémantique pour des descriptions écrites dans un langage de programmation.

Objet sémantiquement formel (objet sémantique) : objet dont la source (par exemple, son auteur) a déclaré qu'il avait un sens unique. Pour cela, il peut utiliser un LRC ou lui donner une définition non ambiguë. Pour avoir un sens unique, cet objet doit aussi être lexicalement formel. Dans la suite de cette thèse, nous abrégions "objet sémantiquement formel" par "objet sémantique". Une représentation de connaissance est un objet sémantique.

Comparaison sémantique : deux choses ou descriptions de choses sont *sémantiquement comparables* si l'une généralise l'autre, c'est à dire, si l'une peut être déduite logiquement de l'autre. Plusieurs outils, par exemple, les moteurs de résolution de requête, sont basés sur la comparaison de deux choses. Un outil tel qu'un démonstrateur de théorèmes peut être utilisé pour effectuer cette comparaison de façon automatique. Il y a deux cas.

1. La relation de déduction logique, par exemple, une relation *subtype* est déjà représentée de façon explicite entre les deux choses. Par exemple, deux primitives différentes peuvent être sémantiquement comparables si l'une est un sous-type de l'autre.
2. Les deux choses ont des définitions totales, c'est à dire, elles sont définies par des conditions nécessaires et suffisantes. Si ces descriptions sont sémantiquement comparables, les deux choses le sont aussi.

Précision sémantique : plus les définitions de deux choses sont précises (donc, entre autres, ni sur-spécialisées, ni sur-généralisées), plus elles peuvent être comparées. Ainsi, plus des descriptions sont précises, plus les requêtes pour les retrouver pourront être précises. Par exemple, la définition “toute voiture a exactement 4 roues” est plus précise que “toute voiture a exactement 4 parties”.

Réutilisabilité sémantique : plus des choses sont sémantiquement comparables, plus elles sont sémantiquement réutilisables. C'est à dire, plus elles peuvent être sémantiquement comparées, plus elles peuvent être retrouvées ou agrégées. Ainsi, plus des choses peuvent être retrouvées ou agrégées, plus elles peuvent être sémantiquement réutilisées. Par exemple, la définition “toute voiture a exactement 4 roues” peut être automatiquement retrouvée par les utilisateurs qui recherchent des “choses avec exactement 4 parties ayant chacune une forme arrondie”.

Donnée [data] : ensemble de termes, qui n'est pas un objet sémantique.

Information : donnée ou représentation de connaissance.

Conteneur de description : conteneur non physique d'une description mais qui a un support physique comme, par exemple, du papier ou une clef USB. Un fichier est un conteneur de description.

Instrument de description : objet d'information permettant de décrire une information. Un instrument de description est contenu dans un conteneur de description.

Contenu de description : contenu de la description d'une chose. Un contenu de description est décrit via un instrument de description.

Module : ensemble d'informations isolées d'autres informations, par exemple, en utilisant fichier distinct pour chaque module ou via un AE prévu à cet effet pour chaque module.

Web sémantique : ensemble des RCs qui utilisent les LRCs promus par le W3C.

Information implicitement représentée : information décrite par un objet d'information qui n'est pas un objet sémantique. La description peut alors être ambiguë. Par exemple, une relation structurelle a une sémantique implicitement représentée.

Information explicitement représentée : information décrite par un objet sémantique.

Base de connaissances (BC) [knowledge base (KB)] : collection de représentations de connaissances.

Commande : soit “phrase affirmée” dans une BC, soit requête sur une BC.

Modèle conceptuel : BC non conçue pour une ou plusieurs applications particulières et donc dont l'expressivité n'est pas restreinte pour faciliter ces applications.

Modèle de conception : BC dans laquelle des choix techniques ont été faits pour une ou plusieurs applications particulières, par exemple, un moteur d'inférence particulier.

Modèle exécutable : programme ou BC exécutable pour une application particulière avec un interpréteur particulier. Cet interpréteur est typiquement un moteur d'inférences tel que celui de Prolog qui permet la prise en compte de l'ordre des règles et utilise la négation par l'échec [negation as failure].

Élément de LRC : terme ou phrase faisant partie de ce LRC.

Élément abstrait (EA) [Abstract element (AE)] : élément de LRC qui peut décrire, par exemple :

- une formule, c'est à dire un phrase qui dénote un fait ;
- un terme abstrait, comme par exemple, un appel de fonction ou un type de relation binaire.

Élément concret (EC) [Concrete element (CE)] : élément de LRC qui décrit la représentation concrète d'un élément abstrait. Par exemple, “ $3 = 2 + 1$ ” et “ $(= (3 +(2 1)))$ ” sont deux représentations concrètes d'un même élément abstrait qui est une formule d'égalité entre 3 et l'addition de 2 et 1.

Modèle abstrait : ensemble d'AEs. Par exemple, une grammaire abstraite est un modèle abstrait.

Modèle concret (ou notation) : ensemble de CEs. Par exemple, une grammaire concrète est un modèle concret.

CST [Concrete Syntax Tree] : arbre syntaxique concret et donc un ensemble de CEs.

AST [Abstract Syntax Tree] : arbre syntaxique abstrait et donc un ensemble d'AEs.

ASG [Abstract Semantic Graph] : graphe sémantique abstrait et donc un ensemble d'AEs.

Analyse lexicale [Tokenization] : processus qui prend en entrée une représentation textuelle ou graphique et a en sortie une liste d'ECs atomiques.

Analyse syntaxique [Parsing] : processus qui prend en entrée une liste de CEs et a en sortie un CST ou un AST ou un ASG incluant quelques CEs.

homo-icongité : propriété de certains langages dans lesquels la structure des CEs reflète la structure des AEs.

Type : référence à un ensemble de choses (les instances du type) et représentant certaines caractéristiques communes à ses instances. Un type peut être un type de concept [concept type] ou un type de relation [relation type].

Lambda-abstraction : définition d'un type de fonction non nommé via des conditions nécessaires et suffisantes.

Kappa-expression : définition d'un type de relation non nommé via des conditions nécessaires et suffisantes.

Concept-type-expression : définition d'un type de concept non nommé via des conditions nécessaires et suffisantes.

Individu [individual] : instance qui ne peut avoir d'instance.

Quantificateur de la logique du premier ordre : quantificateur existentiel (i.e., "il existe", c'est à dire "au moins un") et un quantificateur universel ("quel que soit" ["for all"], "chaque" ["every", "each"]).

Quantificateur numérique : quantificateur individuel (par exemple, "2", "entre 2 et 3", "au moins 4") ou statistique (par exemple, "35%").

Noeud (ou expression) : un terme et éventuellement un quantificateur sur ce terme ; de plus, un noeud est soit

- un **noeud concept** si le terme est un type de concept ou bien un individu ;
- un **noeud relation** si le noeud permet de créer une relation entre plusieurs noeuds concept. Le terme doit alors être un type de relation.

Relation sémantique : objet d'information composé d'un noeud relation et des objets d'information qui sont directement reliés par ce noeud relation. Dans cette thèse, les types de relation sémantique seront en italique et le mot "sémantique" après "relation" sera généralement omis. Voici quelques exemples de types de relations sémantiques :

- généralisation (des sous-types sont, par exemple, *logical_deduction*, *supertype*),
- sous-partie (dont les sous-types sont, par exemple, *subprocess*, *physical_part*) ;
- relations depuis un processus (par exemple, *input*, *output*, *destinataire*, *acteur*).

Une relation est orientée et peut être unaire, binaire, ternaire, etc. Une relation binaire a un seul noeud origine et un seul noeud destination. Le W3C, par exemple, utilise un vocabulaire différent. Ainsi, dans le modèle RDF :

- une relation est appelée un triplet ;
- un type de relation est appelé une propriété ou un prédicat ;
- un noeud origine est appelé un sujet ou une source ;
- un noeud destination est appelé un objet ou une destination.

Relation lexicale : objet formel (non sémantique) ne pouvant être utilisé qu'entre des symboles informels. Voici quelques types de relation lexicale : "homonyme", "antonyme", les relations entre chaînes de caractères.

Lien [Link] : relation binaire. Dans OWL2, un type de lien est une instance d'un des types du second ordre suivant "owl2:ObjectProperty" ou "owl2:DatatypeProperty".

Notation de 2nd-ordre : notation permettant d'utiliser des variables pour les types de relations et de les quantifier. Une notation du 2nd-ordre n'implique pas nécessairement un moteur d'inférence du 2nd-ordre pour l'utiliser, en particulier si les phrases du 2nd-ordre sont des définitions. En effet, dans une BC donnée, il n'existe souvent qu'un nombre fini de types de 1er ordre concernés par ces définitions et il est alors souvent possible d'appliquer (c'est à dire, instancier) la définition du 2nd-ordre à ces catégories. Par exemple, définir la transitivité demande une logique du 2nd-ordre, mais définir une relation transitive particulière ne demande qu'une logique du premier ordre. Le moteur d'inférences peut alors ne pas prendre en compte la définition du 2nd-ordre.

Notation de haut niveau : notation qui permet de représenter des phrases générales de manières concises et normalisées et donc dont les représentations expressives et normalisées sont faciles à comparer, par exemple, par comparaison de graphes. Ceci implique (entres autres) la possibilité d'utiliser des méta-phrases contextualisantes, des quantificateurs numériques et d'autres raccourcis.

Représentation infixée d'un lien ou d'une fonction r (dans une notation textuelle) : notation textuelle dans laquelle le type de r est positionné entre la source et la destination de r , comme dans la notation Turtle. Ainsi, en Turtle, la phrase "Tom is on a mat" peut s'écrire :

ex:Tom ex:place ex:Cat. // "ex" est l'identifiant de l'espace de nommage dans lequel Tom est déclaré

Représentation préfixée d'un lien ou d'une fonction r (dans une notation textuelle) : notation textuelle dans laquelle le type de r est positionné avant la source et la destination de r , comme dans la notation CLIF. Ainsi, en CLIF, la phrase "Tom is on a mat" peut s'écrire : (place Tom mat).

Représentation postfixée d'un lien ou d'une fonction r (dans une notation textuelle) : notation textuelle dans laquelle le type de r est positionné après la source et la destination de r comme, par exemple, dans la notation polonaise inversée [reverse polish notation]. Ainsi, dans un LRC qui utiliserait une telle notation, la phrase "Tom est un chat" pourrait s'écrire : Tom mat place.

Ontologie : ensemble de termes formels avec, associés à ceux-ci et portant sur eux, des définitions sémantiques partielles ou totales. Une ontologie est généralement logiquement définie, par exemple, via un LRC. En programmation orienté objet, il est possible de définir un type de concept via une classe, des propriétés (au sens UML) et des méthodes et donc également de décrire une ontologie qui n'est pas logiquement définie. Certaines BCs sont des ontologies, toute ontologie est une BC.

Ontologie de LRCs : ontologie dont les termes décrivent un modèle de données, ainsi que sa logique associée, utilisables pour représenter des connaissances.

Ontologie lexicale (par exemple, de langage naturel) : ontologie dont les termes formels représentent le sens de termes informels d'un lexique ou d'un langage naturel.

Ontologie de haut niveau : ontologie dont les termes formels représentent des distinctions générales (par exemple, Entité, Situation, Espace, Temps, *part*, ...) généralement utiles pour organiser ou contrôler toute ontologie qui n'est pas une ontologie de langage.

Ontologie fondamentale [foundational ontology] : ontologie formelle de haut niveau dont les termes sont très précisément définis et, le plus souvent, sont relatifs à un seul thème, par exemple, les relations spatiales, temporelles, *partie-de*.

Ontologie de domaine : ontologie dont les termes sont relatifs à un domaine, par exemple, la radiographie, l'élevage des poules, les livres, etc.

Ontologie homogène en interne : ontologie dans laquelle toutes les descriptions sont basées sur quelques primitives, par exemple, quelques types de relation sémantique.

Ontologie hétérogène en interne : ontologie qui n'est pas homogène en interne.

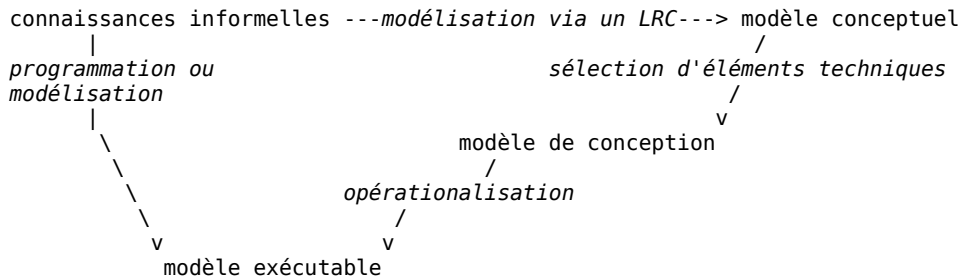
Ontologies globalement homogènes (ou homogènes entre elles) : ontologies dont l'agrégation est homogène en interne.

2.2. Difficulté des tâches de modélisation et/ou de programmation pour le partage de connaissances

Cette section décrit des difficultés à modéliser et/ou programmer des phrases de sorte qu'elles soient le plus réutilisables possible. Ces difficultés sont examinées en détail dans les sous-sections. Nous verrons qu'il est plus facile d'écrire des phrases réutilisables dans les cas suivants.

1. Dans un modèle conceptuel plutôt que dans un modèle exécutable.
2. Avec un LRC plutôt qu'avec langage qui n'est pas un LRC.
3. Avec un langage de description plutôt qu'avec un langage de programmation.

Le schéma ci-dessous présente les relations entre les différents modèles.



Un modèle exécutable peut être créé de deux façons :

1. directement via des tâches de programmation ou de modélisation ;
2. indirectement via des choix techniques depuis un modèle conceptuel et une phase d'opérationnalisation depuis un modèle de conception

Les critères pour déterminer la difficulté des tâches de modélisation et/ou de programmation (pour le partage de connaissances) sont les suivants.

1. Il est plus facile de réutiliser que de re-crée. En particulier, créer un modèle conceptuel puis générer plusieurs modèles exécutables est plus facile que créer directement plusieurs modèles exécutables. Ce critère est utilisé dans les sections [2.2.2.](#) et [2.2.3.](#)
2. Modéliser est plus facile que programmer. Ce critère est utilisé dans la section [2.2.3.](#)
3. Ne pas programmer est plus facile que programmer. Ce critère est utilisé dans la section [2.2.3.](#)

Des connaissances représentées dans un modèle conceptuel peuvent toujours être extraites pour être utilisées dans un modèle exécutable. Dans cette thèse, nous écrirons également qu'un modèle exécutable peut toujours être "extrait" d'un modèle conceptuel. Ainsi, lorsqu'un modèle exécutable est construit indirectement, la seule difficulté de modélisation vient de la construction du modèle conceptuel.

Si le modèle exécutable est créé directement, les descriptions ne sont pas extraites d'un modèle conceptuel : elles doivent être "créées", c'est à dire, extraites directement, par un modélisateur ou un programmeur à partir de connaissances informelles. Selon le langage utilisé, le type des descriptions à écrire est différent. En effet, lorsqu'un LRC est utilisé, un modélisateur écrit des représentations de connaissances (RCs). Lorsqu'un langage de description de structures est utilisé, un modélisateur ou un programmeur n'écrit que des descriptions structurelles (cf. la définition de Langage de description de structures de données) et éventuellement des descriptions procédurales ou fonctionnelles. Lorsqu'un langage de programmation est utilisé, un programmeur écrit des descriptions procédurales ou fonctionnelles. L'écriture de descriptions procédurales ou fonctionnelles est une tâche de programmation, l'écriture de RCs ou de descriptions structurelles est une tâche de modélisation.

Note : chaque sous-section de la section courante (2.2.) est résumée dans un paragraphe ayant la mise en forme de cette phrase.

2.2.1. Création d'un modèle conceptuel via un LRC

Dans un modèle conceptuel, avec un LRC, les connaissances peuvent être représentées de façon précise et un moteur d'inférences peut être utilisé.

Dans un modèle conceptuel, les descriptions des processus sont supposées être très précises, réutilisables et comparables. Ces descriptions sont donc supposées être écrites dans un LRC de haut niveau avec une ontologie pour la modélisation [Guizzardi, 2005]. Lorsqu'un LRC est utilisé, les descriptions peuvent être plus précises qu'avec un autre langage déclaratif, elles ont par exemple une interprétation dans une logique. Ainsi, des inférences logiques peuvent être réalisées via un outil tel qu'un démonstrateur de théorèmes.

Lorsque la modélisation concerne les processus d'import, d'export et de traduction *de LRCs*, un modèle conceptuel *de LRCs* que nous appelons *complet* doit contenir une ontologie de modèles abstraits de LRCs et une ontologie de modèles concrets de LRCs. Nous verrons dans la section “4. KRLO, une ontologie de LRCs” la façon dont nous avons créé de telles ontologies et permis leur exploitation.

2.2.2. Création directe d'un modèle exécutable via un LRC exécutable

Dans un modèle exécutable, des contraintes liées aux applications doivent être prises en compte. Ainsi, l'expressivité des RCs peut être limitée et des représentations peuvent être biaisées, incorrectes ou incomplètes. Ceci ne facilite pas la représentation précise des connaissances et donc leur réutilisation (cf. critère 1 dans la section 2.2. ci-avant).

Beaucoup de modèles exécutables, pour des raisons de performances, limitent l'expressivité des RCs, par exemple, en imposant l'utilisation de langages comme RDF+OWL-DL pour les descriptions. Comme un modèle conceptuel n'a pas été créé, des RCs optimisées pour l'exécution ne peuvent pas être extraites d'un tel modèle. Ces RCs optimisées doivent donc être décrites directement. Si des connaissances complexes doivent être représentées, le modélisateur peut être amené à créer une représentation incorrecte ou incomplète. Par exemple, la phrase (inventée) : “à La Réunion, en 2016, selon une étude de Carole Payet, tous les oiseaux volent” peut difficilement être représentée en RDF+OWL-DL. En effet, contrairement à KIF [Genesereth & Fikes, 1992], par exemple, RDF+OWL-DL ne fournit aucun moyen de i) représenter des méta-phrases contextualisantes telle que “A La Réunion” ou “en 2016”, ni de ii) différencier une définition d'une observation quantifiée universellement.

Des représentations incorrectes ou incomplètes peuvent mener à des résultats d'inférence incohérents. De telles représentations sont plus difficiles à réutiliser pour le partage de connaissances ; plus de détails sont donnés dans la définition de la réutilisation sémantique.

Dans un modèle exécutable, les représentations sont généralement biaisées pour convenir à un type particulier d'applications ou de moteur d'inférences. La dépendance des RCs vis-à-vis d'une application particulière ou d'une technique de résolution particulière ne favorise pas la réutilisation des RCs. Par exemple, la technique de résolution de l'interpréteur Prolog utilise une recherche en profondeur d'abord. Ainsi, l'ordre des descriptions a une importance. Par exemple, *les descriptions ci-dessous* sont logiquement équivalentes, mais la seconde fait entrer le moteur Prolog dans une boucle infinie. Un programmeur doit tenir compte de la stratégie de recherche en profondeur d'abord effectuée par le moteur lorsqu'il écrit des descriptions. D'autres interpréteurs peuvent être insensibles à l'ordre des descriptions. Les RCs d'autres modèles exécutables peuvent donc être difficiles à réutiliser pour le partage de connaissances.

1) Cas où la requête "ancestor(X,Y)." renvoie true.

```
parent(Zo,Meu).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

2) Cas où la requête "ancestor(X,Y)." conduit le moteur dans une boucle infinie.

```
parent(Zo,Meu).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
ancestor(X,Y) :- parent(X,Y).
```

2.2.3. Création directe d'un modèle exécutable via un langage qui n'est pas un LRC

Lorsqu'un LRC n'est pas utilisé, les descriptions sont essentiellement structurelles, leur sémantique est donc implicitement représentée. Elles sont difficilement comparables et donc peu réutilisables (cf. critère 1 dans la section 2.2. ci-avant). Plus de détails sur la comparabilité sémantique et sur la réutilisabilité sémantique sont donnés dans les définitions. Des techniques d'inférence – comme la transitivité – et donc une partie d'un moteur d'inférences doivent être programmées ou spécifiées (cf. critère 3 dans la section 2.2. ci-avant).

Puisqu'un LRC n'est pas utilisé, les relations sont purement structurelles. Leur sémantique est connue du modélisateur ou de ses collaborateurs et éventuellement des personnes qui ont lu et compris la documentation. Elles ne peuvent donc pas être interprétées par un moteur d'inférences logiques. Pour traiter ces relations comme si elles avaient une sémantique, des règles ou des techniques d'inférences, comme l'héritage de propriétés, doivent être programmées via un langage de programmation ou

spécifiées via un langage déclaratif. Ainsi, une comparaison de la sémantique de deux descriptions nécessite une comparaison manuelle ou semi-automatique de deux programmes.

2.2.3.1. Le langage utilisé pour créer le modèle exécutable est un langage de description dont la structure peut être manipulée via des outils ou un langage

2.2.3.1.1) Le modèle abstrait du langage a une structure de graphe

L'utilisation d'un langage qui n'est pas un LRC et dont le modèle abstrait a une structure de graphe incite les concepteurs de modèles exécutables à créer des interpréteurs ad hoc pour ce langage. Ceci implique des tâches de programmation (cf. critère 2 dans la section [2.2](#), ci-avant) et complique le partage et la réutilisation des connaissances (cf. critère 1 dans la section [2.2](#), ci-avant).

Pour traiter la sémantique de descriptions écrites dans un langage de description qui n'est pas un LRC, un interpréteur de ce langage doit être adapté ou complété. Ainsi, chaque concepteur de modèle exécutable peut créer un interpréteur spécifique à ce modèle. Ceci complique le partage de connaissances car une même description peut être interprétée différemment d'un modèle exécutable à l'autre. Trois exemples de descriptions et de spécifications d'inférences sont présentées ci-dessous dans le langage JSON.

1. Dans l'exemple suivant, le langage JSON est utilisé pour représenter le contenu de la phrase "Socrate est un homme de plus d'un mètre cinquante". Comme JSON n'est pas un LRC, la règle "tout homme de plus d'un mètre cinquante est heureux" est spécifiée ci-dessous en Javascript pour compenser l'absence de moteur d'inférences.

```
{ "id": "Socrate",
  "instanceOf": "homme",
  "height": { "id": "taille_de_Socrate",
              "value": 150,
              "unit": "cm"
            }
}

function /*Boolean*/ isHappyMan (aMan)
{ return ((aMan.height.value > 150) && (aMan.height.unit == "cm"));
}
```

2. Dans l'exemple suivant, le langage JSON est utilisé pour représenter : "Socrate est un homme, tout homme est mortel". Comme JSON n'est pas un LRC, la transitivité de la relation nommée : "instanceOf" est spécifiée ci-dessous en Javascript pour compenser l'absence de moteur d'inférences.

```
{ "id": "Socrate",
  "instanceOf": { "id": "homme",
                  "subtypeOf": "mortel"
                }
}

function /*Boolean*/ hasInstance (conceptType, anObject)
{ if (equals(anObject.instanceOf, conceptType)) return true;
  else
    return (hasSubtype(conceptType, anObject.instanceOf));
}

function /*Boolean*/ hasSubtype (superType, type)
{ iterator = getIteratorFromList(superType.subtypes);
  while (iterator.hasNext())
  { if (equals(type, iterator.current()) return true;
    else if (hasSubtype(iterator.current(), type)) return true;
    iterator.next();
  }
  return false;
}
```

3. L'exemple suivant présente une description qui peut être interprétée de plusieurs façons. Sans connaître le code de l'interpréteur à utiliser, il est impossible de déterminer la sémantique que son auteur entendait donner à cette description. Cette dernière peut représenter, entre autres, les phrases suivantes.
- “Socrate est un homme particulier qui est mortel et qui a exactement une tête” ;
 - “Socrate est un homme, tout homme est mortel et a exactement une tête” ;
 - “Socrate est un homme, tout homme est mortel et a au moins une tête, et toute tête est une partie d'un homme”.

```
{ "id": "Socrate" ;
  "instanceOf": { "id": "homme" ;
                  "subTypeOf": "mortel" ;
                  "part": "tete"
                }
}
```

L'utilisation de certains langages de programmation supportant le code-as-data, comme Lisp, peut limiter les difficultés de programmation à celles qui sont présentées ci-dessus, dans cette section. Par exemple, le contenu de la phrase “Socrate est un homme, tout homme est mortel” pourrait être décrit par une série de listes en Lisp :

```
'(concept mortel)
'(concept homme)
'(individual Socrate)
'(relation subtype mortel homme)
'(relation instance homme Socrate)
```

Ces listes peuvent ensuite être analysées par des fonctions reproduisant des techniques d'inférences. Par exemple, cette fonction Lisp pourrait reproduire la transitivité de la relation instance :

```
(defun /*boolean*/ hasInstance (aConceptList anIndividual KB)
  (if (null aConceptList) false
      (if (find `(relation instance ,(car aConceptList) ,anIndividual) KB :test 'equal) true
          (if (hasInstance (cdr aConceptList) anIndividual KB) true
              (hasInstance (superTypes (car aConceptList)) anIndividual KB) ) ) ) )
```

Le paradigme code-as-data est important. Il permet au programmeur d'écrire des descriptions interprétables par les interpréteurs du langage de programmation sans avoir à écrire un analyseur lexico-syntaxique spécifique pour ces descriptions.

2.2.3.1.1) Le modèle abstrait du langage a une structure d'arbre

Lorsque le langage a une structure d'arbre, les “circuits” ne peuvent pas être représentés. De plus, des relations structurelles peuvent être nommées uniquement de façon ad hoc, un interpréteur du langage doit donc être adapté via des tâches de programmation (cf. critère 2 dans la section [2.2.](#) ci-avant). Pour ces deux raisons, les descriptions tendent à être imprécises, incorrectes ou incomplètes, ce qui complique leur réutilisation (cf. critère 1 dans la section [2.2.](#) ci-avant).

Un arbre est un graphe acyclique orienté possédant une racine unique. Dans un arbre, les arcs représentent des relations “parent-child” et chaque noeud a au plus un parent. Les relations “parent-child” sont des relations structurelles, pas des relations sémantiques. XML est un exemple de langage dont le modèle abstrait a une structure d'arbre.

Un langage dont le modèle abstrait a une structure d'arbre est différent d'un langage dont le modèle abstrait a une structure de graphe. Certains langages qui ont un modèle concret avec une structure d'arbre autorisent les modélisateurs à utiliser certains noeuds comme des références vers d'autres noeuds. Le modèle abstrait de ces langages *n'a pas* une structure d'arbre. Les difficultés de modélisation pour les langages qui ne sont pas des LRCs et dont le modèle abstrait a une structure de graphe sont détaillées dans la section 2.2.3.1.1.

RDF/XML est un exemple de langage :

- dont le modèle concret a une structure d'arbre,
- dont le modèle abstrait a une structure de graphe, et
- dont les relations ont une sémantique comprise par ses interpréteurs.

RDF/XML est un LRC. Les difficultés de modélisation pour les LRCs sont décrites dans les sections [2.2.1.](#) et [2.2.2.](#)

Dans un langage dont le modèle abstrait a une structure d'arbre, des structures particulières doivent être utilisées pour représenter des relations structurelles nommées. Par exemple, elles peuvent être représentées par des noeuds. Dans certains langages, de telles structures peuvent être spécifiées, comme, par exemple, en XML via une DTD. Une fois spécifiée, la syntaxe de ces structures peut être vérifiée. Ces structures ne sont cependant pas interprétables par les interpréteurs du langage. Ainsi, un

interpréteur doit être adapté pour chaque type de structure représentant une relation structurelle nommée. Puisque des structures particulières interprétées de façon ad hoc sont utilisées pour représenter des relations :

- un grand nombre de structures différentes peuvent représenter une même relation – voir les exemples ci-après ;
- une même structure arborescente peut représenter plusieurs choses différentes – voir les exemples ci-après.

Quelques types de structures standard ont été proposés. Certains d'entre eux, pour XML, sont présentés sur cette page : <http://www.ltg.ed.ac.uk/~ht/normalForms.html>. Par exemple, l'un d'eux impose une alternance entre les noeuds représentant des relations et les autres. Via ces standards, la représentation des relations est moins ad hoc et il y a moins d'ambiguïté sur les types de relation. Dans tous ces exemples, les noms des relations commencent par une minuscule, tous les autres noms commencent par une majuscule.

Quelques représentations XML possibles pour la phrase : “John, a person, has for employer the IBM company”. “John” et “IBM” sont des instances de “person” et “company”, respectivement ; “employer” est un type de relation.

Représentations XML avec des relations ad hoc

1. Dans l'exemple ci-dessous, une notation XML alternant les types de relations et les types de concepts est utilisée (cf. *Alternating Normal Form* sur la page <http://www.ltg.ed.ac.uk/~ht/normalForms.html>).

```
<Person>
  <name>John</name>
  <employer><Company>IBM</Company></employer>
</Person>
```

2. Dans l'exemple ci-dessous, les relations *instance* sont représentées via des attributs. La source de la relation *employer* est spécifiée via l'attribut "instance" du noeud parent. Ainsi, la source de cette relation *employer* est le “John”. La destination de cette relation est spécifiée via l'attribut "instance" du noeud enfant “Company”. Ainsi, la destination de cette relation *employer* est “IBM”.

```
<Person instance="John">
  <employer><Company instance="IBM" /></employer>
</Person>
```

3. Les exemples ci-dessous montrent que les structures XML n'ont pas toujours de sémantique (même implicite).
 - Dans l'exemple ci-dessous, le noeud “WorkingDetails” n'a pas de sémantique.

```
<Person instance="John">
  <WorkingDetails> <employer><Company instance="IBM" /></employer> </WorkingDetails>
</Person>
```

- L'exemple ci-dessous montre une description sans relation.

```
<Person instance="John">
  <Employer><Company instance="IBM" /></Employer>
</Person>
```

4. L'exemple ci-dessous montre deux relations "parent-child" utilisées pour représenter deux choses différentes. Ainsi, cette relation est utilisée pour i) spécifier la source de la relation *employer*, c'est à dire “John”, ii) pour représenter une relation *instance* depuis “Person” vers “John”.

```
<employer>
  <Person>John</Person>
  <Company>IBM</Company>
</employer>
```

5. Dans l'exemple ci-dessous, les relations sémantiques *employer* et *instance* sont représentées via des structures différentes. La source de la relation *instance* est le noeud “Person”, sa destination est le noeud “John”. Cette représentation est très différente de celle de la relation *employer*. Pour cette dernière, la source est “John” et la destination est “IBM”.

```
<employer>
  <Person><instance>John</instance></Person>
  <Company><instance>IBM</instance></Company>
</employer>
```

Dans un langage ayant une structure d'arbre, les "circuits" ne peuvent pas être représentés. Un exemple de "circuit" est donné dans la phrase suivante : "John a deux enfants, l'un d'eux est employé dans l'entreprise de John.". L'un des enfants de John devrait être la destination de deux relations structurelles nommées : "employeur" et "enfant". Or, dans un arbre, un noeud ne peut être la destination que d'une unique relation. Ainsi, un modélisateur est amené à créer des représentations peu précises, incorrectes ou incomplètes.

2.2.3.2. Le langage utilisé pour créer le modèle exécutable est un langage de programmation

Beaucoup de structures de données sont créées et manipulées uniquement de façon indirecte via des traitements prédéfinis par un programmeur. Si un langage impératif est utilisé, les flux de contrôles doivent également être décrits (cf. critère 3 dans la section [2.2.](#) ci-avant). La spécification des traitements – et du flux de contrôle – sont des tâches de programmation. Ces tâches sont plus difficiles que des tâches de modélisation (cf. critère 2 dans la section [2.2.](#) ci-avant).

Comme un langage de programmation est utilisé, les connaissances sont représentées via des descriptions procédurales. Dans certains langages, quelques relations sémantiques entre des "types" de structures de données peuvent être interprétées. Typiquement, dans les langages orienté objet, des relations *subtype* entre les classes sont interprétées.

Les langages de programmation – en particulier les langages orientés objet et fonctionnels – incluent généralement un langage de description de structure pour spécifier quelques structures de données. Par exemple, dans les langages orienté objet, ce langage de description intégré permet de décrire les classes.

L'utilisation de "types" et "d'instances" de structures de données pour représenter des connaissances pose des problèmes similaires à ceux présentés dans la section "[2.2.3.1. Le langage utilisé pour créer le modèle exécutable est un langage de description dont la structure peut être manipulée via des outils ou un langage](#)". En effet, les données ont alors structure de graphe alors que les relations entre elles sont uniquement structurelles. Comme en XML, des "types" de structure de données peuvent être décrits. Par exemple, en Orienté Objet, des classes et des attributs peuvent être spécifiés. Ainsi, un ensemble de relations structurelles accessibles depuis une "instance" de l'un de ces types peut être prédéfini. Comme en XML, la sémantique des structures est implicite, pour plus de détails voir la section [2.2.3.1.1.1](#) ci-avant.

Lorsque des descriptions procédurales sont utilisées pour décrire les structures de données, celles-ci sont difficiles à réutiliser et à partager. En effet, comme un langage de description n'est pas utilisé, un modélisateur ne peut décrire directement ni les structures "instance", ni les relations structurelles entre ces structures. Elles doivent être décrites indirectement : un programmeur doit décrire des traitements pour créer et manipuler en mémoire de telles structures et de telles relations. En mémoire, elles peuvent difficilement être partagées. Pour les diffuser, un programmeur est généralement obligé d'écrire un processus d'export vers un langage de description, typiquement, XML.

Retrouver et donc réutiliser des structures types, des "instances" ou des relations structurelles décrites via des descriptions procédurales peut être difficile. En effet, ces structures et relations peuvent être recherchées uniquement via quelques requêtes prédéfinies par un programmeur. Lorsque le langage utilisé est un langage réflexif, ces requêtes peuvent prendre en paramètre et exécuter du code écrit dans ce langage. Un outil de manipulation de programme – comme Coccinelle pour le C ou DMS qui est plus générique – peut être utilisé à cette fin pour des langages non réflexifs. Ainsi, les structures et les relations peuvent être exploitées de façon non prédéfinie. Cette exploitation nécessite cependant des tâches de programmation qui peuvent être difficiles, en particulier pour les utilisateurs qui ne sont pas des programmeurs ou même pour des programmeurs ayant peu de connaissances sur le programme à manipuler.

2.3. Précisions et exemples pour l'homo-iconicité

Par définition, dans un langage homo-iconique, le CST – privé d'éléments purement syntaxique comme les délimiteurs, par exemple, les parenthèses en Lisp – et l'AST ou l'ASG ont des structures isomorphes. Lorsque la notation utilisée a les deux caractéristiques suivantes, un langage est homo-iconique.

1. La notation force l'utilisateur à mentionner le nom ou le type de tous les éléments de l'AST. C'est par exemple le cas avec XML et les notations régulières et restreintes comme celle de Lisp (où toute structure de l'AST est une fonction) ou celle de N-triples (où toute structure est une relation binaire).
2. La notation permet d'utiliser du sucre syntaxique pour éviter de nommer un élément ou type d'élément sans pour autant modifier la structure du CST.

Le langage Prolog est un exemple de langage homo-iconique. En effet, sa notation force l'utilisateur à mentionner le nom de tous les éléments de l'AST et n'offre du sucre syntaxique que pour abrévier certaines relations. Par exemple, “:-” est une abréviation pour la relation d'implication. Ainsi, tout langage dont le modèle abstrait n'est basé que sur des relations n-aire et dont la notation est celle de Prolog est homo-iconique.

RDF avec la notation N-triples est aussi un langage homo-iconique. RDF avec la notation JSON-LD n'est pas homo-iconique. Ci-dessous, un exemple extrait de la documentation de JSON-LD (<https://www.w3.org/TR/json-ld/>, exemple 73) est présenté puis traduit dans le modèle RDF avec la notation N-Triples. Pour une meilleure lisibilité, les IRI de la notation N-Triples sont abrégées via l'utilisation d'espaces de noms. Par exemple, <http://example.org/people#joebob> est abrégé `ex:joebob`.

<pre>RDF/JSON-LD : { "@context": { "foaf": "http://xmlns.com/foaf/0.1/", "ex": "http://example.org/people/" }, "@id": "ex:joebob", "@type": "foaf:Person", "foaf:name": "Joe Bob", "foaf:nick": { "@list": ["joe", "bob", "jaybee"] } }</pre>	<pre>RDF/N-Triples : <ex:joebob> <rdf:type> <foaf:Person> . <ex:joebob> <foaf:name> "Joe Bob" . <ex:joebob> <foaf:nick> _:L1 . _:L1 <rdf:first> "joe" . _:L1 <rdf:rest> _:L2 . _:L2 <rdf:first> "bob" . _:L2 <rdf:rest> _:L3 . _:L3 <rdf:first> "jaybee" . _:L3 <rdf:rest> <rdf:nil> .</pre>
---	--

2.4. Conventions et exemples de traductions

2.4.1) Conventions de nommage dans KRLO

- Un nom de type de concept ou d'individu est une expression nominale qui commence par une majuscule, par exemple : “Model” et “KRL_Model”.
- Dans les expressions nominales, “_” et “-” sont utilisés pour séparer les mots. Quand ils sont utilisés tous les deux, “-” connecte des mots qui sont plus étroitement liés.
- Les types de relations qui sont *sous-types* de Type_as_description_instrument sont préfixés par “r_” ou “rc_” si la destination de cette relation est un EC ; toutes les autres sont préfixés par “has_”.
- Les noms de types de concepts ou de relations sont préfixés par un espace de nom sauf lorsque celui-ci est laissé implicite. C’est le cas lorsque l’espace de nom est krlo ou bien pm. Par défaut, l’espace de nom implicite est krlo. Lorsque l’espace de nom implicite est pm, cela sera signalé.
- Ainsi, dans ce mémoire, lorsque l’espace de nom n’est pas laissé implicite, les noms qui ne suivent pas ces conventions et ne sont pas préfixés par un espace de nom sont des mots clefs de LRCs.

2.4.2) Convention de lecture

- Les liens de la forme “X R: Y” peuvent être lus “X a pour R Y”, par exemple, “1..* Man r_name: 'Joe'” peut se lire : “un homme a pour nom 'Joe'”.
- Les liens de la forme “X R of: Y” peuvent être lus “X est le R de Y”, par exemple, “'Joe' r_name of: 1..* Man” peut se lire : “'Joe' est le nom d'un homme”.

2.4.3) Langage utilisé


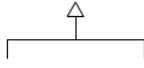


KRLO a été décrite dans le langage FL [Martin, 2009c]. FL a l'avantage d'être régulier, concis, expressif et normalisant.

Dans la notation FL :

- les relations peuvent être écrites avec une notation préfixée, infixée ou postfixée.
- le symbole “:” est utilisé pour séparer la relation et ses destinations dans la notation infixée.
- deux relations sont séparées par une virgule dans la notation infixée, par exemple :
1..* Man name: "Joe", //un homme a pour nom “Joe” et
part: 1..* Leg. //a pour partie au moins une jambe.
- par défaut, le symbole “#” est utilisé pour séparer l'espace de nom et le nom d'un concept ou d'une relation.
- si un lien n'est ni un lien de *sous-type*, ni un autre “lien depuis un type”, sa source est quantifiée et le quantificateur par défaut est un quantificateur universel de définition.
- le quantificateur universel de définition est représenté par “any”, le quantificateur universel factuel est représenté par “every”, le quantificateur existentiel est représenté par “1..*” ou “a” ou “some”.
- la destination d'une relation peut être utilisée comme une source pour une (ou plusieurs) relation(s), par exemple :
Human
> (Man //“>” est une abréviation pour r_subtype
r_part: a Leg) //Tout homme est un humain et a au moins une jambe.
- un concept-type-expression est délimitée par les symboles “^”(et “)”, par exemple :
Man //Tout homme a pour partie
r_part: a ^(Leg r_part: 1 Foot) //au moins une jambe particulière qui a pour partie un unique pied
- les arguments d'une fonction sont délimités par les symboles “_”(et “)”.

2.4.4) Notation pm#UML

La notation pm#UML est une notation graphique adaptée de UML par Philippe Martin pour des raisons de concision et donc de lisibilité. Dans cette thèse, j'utiliserai cette notation dans la plupart des figures. Les éléments graphiques spécifiques de pm#UML sont décrits ci-dessous. Ces éléments graphiques seront à nouveau (partiellement) présentés dans la légende figure 3.1.

1. Les flèches  représentent des relations *subClassOf*.
2.  représente un ensemble de sous-types. Cet ensemble est disjoint et complet par défaut. Dans le cas contraire, une annotation précisant les propriétés de cet ensemble est ajoutée, par exemple, “{incomplete, disjoint}”.
3. Les flèches  représentent les autres relations binaires ; le type de ces relations est précisé au dessus de la flèche en italique. Les cardinalités sont 0..* (alias 0-N) par défaut dans les deux sens pour toutes ces relations.
4. Les flèches  représentent des relations *instance*.
5. Les boites ne sont pas dessinées.
6. “i”, “o”, “part” et “param” sont respectivement les abréviations pour les noms des types de relations *has_input*, *has_output*, *has_part* et *has_parameter*.
7. Les parenthèses délimitent des commentaires.

2.4.5) Exemples de traductions

Des traductions depuis une phrase en anglais vers différents LRCs sont données ci-dessous. Les familles de notations les plus courantes – à l'exception de celles basées sur XML – sont illustrées. Ces exemples montrent la diversité des modèles et notations de LRC, et donnent donc un aperçu de la difficulté de traduire entre des LRCs. La phrase est une définition et utilise des restrictions de cardinalités. Pour les notations ne possédant pas de sucre syntaxique pour les restrictions de cardinalité, le modèle OWL2 est utilisé, sauf dans KIF puisque ce LRC permet de définir le quantificateur numérique “exactlyN”. OWL-Lite aurait été suffisant mais les restrictions de cardinalité qualifiées de OWL2 sont plus lisibles. La phrase en anglais porte sur les oiseaux : nous avons réutilisé et adapté un exemple classique en Intelligence Artificielle. Elle est représentée dans différents LRCs, y compris OWL. Ces représentations sont ordonnées en fonction de leur proximité avec l'anglais. Les noms de certains de ces LRCs sont composés de celui du modèle puis de celui de la notation de de LRC. Par exemple, “RIF+OWL/RIF-PS” suit le modèle RIF-FLD avec l'ontologie OWL (afin de représenter les restrictions de cardinalité) et utilise la notation RIF-PS (RIF Presentation Style).

En dehors de KIF, de OWL Functional Style (FSS) [FSS, 2012] et de RIF-PS, toutes les notations ci-dessous sont basées sur des graphes : elles montrent directement les noeuds concepts – et les noeuds relations les reliant – d'un modèle à base de graphe. Outre UML, ces notations à base de graphes sont quelques fois basées sur des frames, c'est à dire que l'ordre de leurs noeuds concepts peut être important pour les comprendre. Une notation qui n'est pas basée sur des graphes est positionnelle ou basée sur des noms : les noeuds concept apparaissent comme des *arguments positionnels* ou nommés d'un noeud de relation qui ressemble à un appel de fonction dans les langages de programmation traditionnels. Aucun exemple d'argument nommé n'est donné ci-dessous. Dans cette table, l'espace de nom implicite n'est pas krlo mais pm (Philippe Martin).

Représentation du contenu d'une phrase en anglais dans différents LRCs.

<p>Anglais :</p> <p>By definition, a “flying_bird_with_2_wings” is a bird that flies and has two wings.</p>
<p>FE, un langage conçu par le directeur de cette thèse :</p> <pre>any Flying_bird_with_2_wings has for type Bird, is agent of a Flight, has for part 2 Wing.</pre>
<p>FL, un langage conçu par mon directeur de thèse :</p> <pre>Flying_bird_with_2_wings = ^(Bird // "^(...)" : concept-type-expression agent of: a Flight, // "of" reverses the direction of a relation part: 2 Wing); //as the symbol "^" in SPARQL</pre>
<p>CGLF (Conceptual Graph Linear Form) :</p> <pre>type Flying_bird_with_2_wings (*b) [[Bird: *b]-{ ->(has_agent)->[Flight]; ->(has_part)->[Wing:{*}@2]; }]</pre>
<p>KIF (Knowledge Interchange Format) :</p> <pre>(defrelation Flying_bird_with_2_wings (?x) := (exists ((?f Flight)) (and (Bird ?x) (has_agent ?f ?x) (exactlyN 2 '?w Wing ^ (has_part ,?x ?w)))))</pre>
<p>CL+OWL2/CGIF+XMLnamespace, un LRC qui utilise i) pour modèle abstrait Common Logics et OWL2, ii) pour notation CGIF avec des espaces de noms XML :</p> <pre>[If: [Flying_bird_with_2_wings: *b] [Then: [Flight: *f] [Bird: *b] (has_agent ?f ?b) ("rdf:type" *b ["owl:Restriction": *r]) ("owl:onProperty" *r has_part) ("owl:onClass" *r Wing) ("owl:qualifiedCardinality" *r 2)]] //et inversement avec les parties droites du "If:" et du "Then:" interverties</pre>
<p>RIF+OWL2/RIF-PS, un LRC qui utilise i) pour modèle abstrait RIF et OWL2, ii) pour notation RIF-PS. RIF-PS est une notation préfixée sauf pour i) certains éléments, comme l'implication, qui sont infixés et ii) les relations des éléments abstraits de type frame qui s'écrivent également de façon infixée :</p>

```

Forall ?b ?r ( ?b[rdf:type Flying_bird_with_2_wings] :-
    And( ?b[rdf:type->Bird rdf:type->?r]//"?b" est la source de la frame,
        //"?rdf:type->Bird" est une relation vers "Bird"
        Exists ?f ( ?f[rdf:type->Flight has_agent->?b] )
        ?r[owl:onProperty->has_part owl:onClass->Wing owl:qualifiedCardinality->2] ) )
//et inversement avec les parties droites et gauches du ":-" interverties

```

RDF+OWL2/N3, un LRC entièrement basé sur des frames. Dans celles délimités par “[” et “]”, la source des propriétés est implicite :

```

:Flying_bird_with_2_wings owl:intersectionOf
  (:Bird [rdf:type owl:Restriction; owl:onProperty :has_agent; owl:someValuesFrom :Flight]
   [rdf:type owl:Restriction; owl:onProperty has_part;
    owl:qualifiedCardinality 2;
    owl:onClass :Wing] ) .

```

OWL Manchester :

```

Class: Flying_bird_with_2_wings
EquivalentTo:
  Bird
  and has_agent some Flight
  and has_part exactly 2 Wing

```

OWL Functional-style :

```

EquivalentClasses( Flying_bird_with_2_wings
  ObjectIntersectionOf( Bird
                        ObjectSomeValuesFrom( :has_agent :Flight )
                        ObjectExactCardinality( 2 :has_part :Wing ) ) )

```

RDF+OWL/XML :

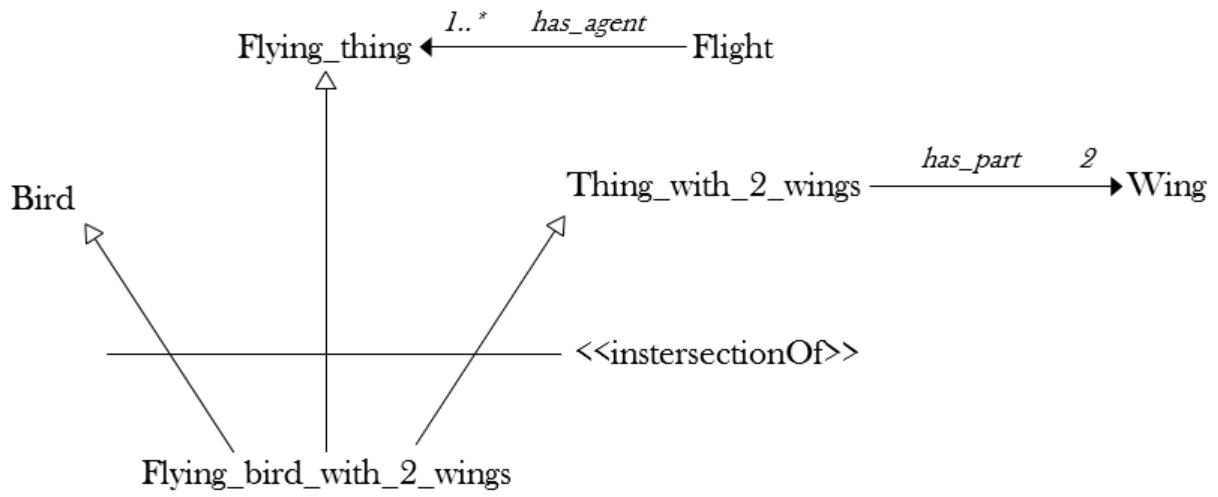
```

<owl:Class rdf:about="Flying_bird_with_2_wings">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Bird"/>
        <owl:Restriction> <owl:onProperty rdf:resource="#has_agent"/>
          <owl:someValuesFrom rdf:resource="#Flight"/> </owl:Restriction>
        <owl:Restriction> <owl:onProperty rdf:resource="#has_part"/> <owl:onClass rdf:resource="Wing"/>
          <owl:qualifiedCardinality rdf:datatype="xsd:nonNegativeInteger"> 2
          </owl:qualifiedCardinality>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

```

pm#UML :

Figure 2.1. "flying_bird_with_2_wings" en pm#UML.



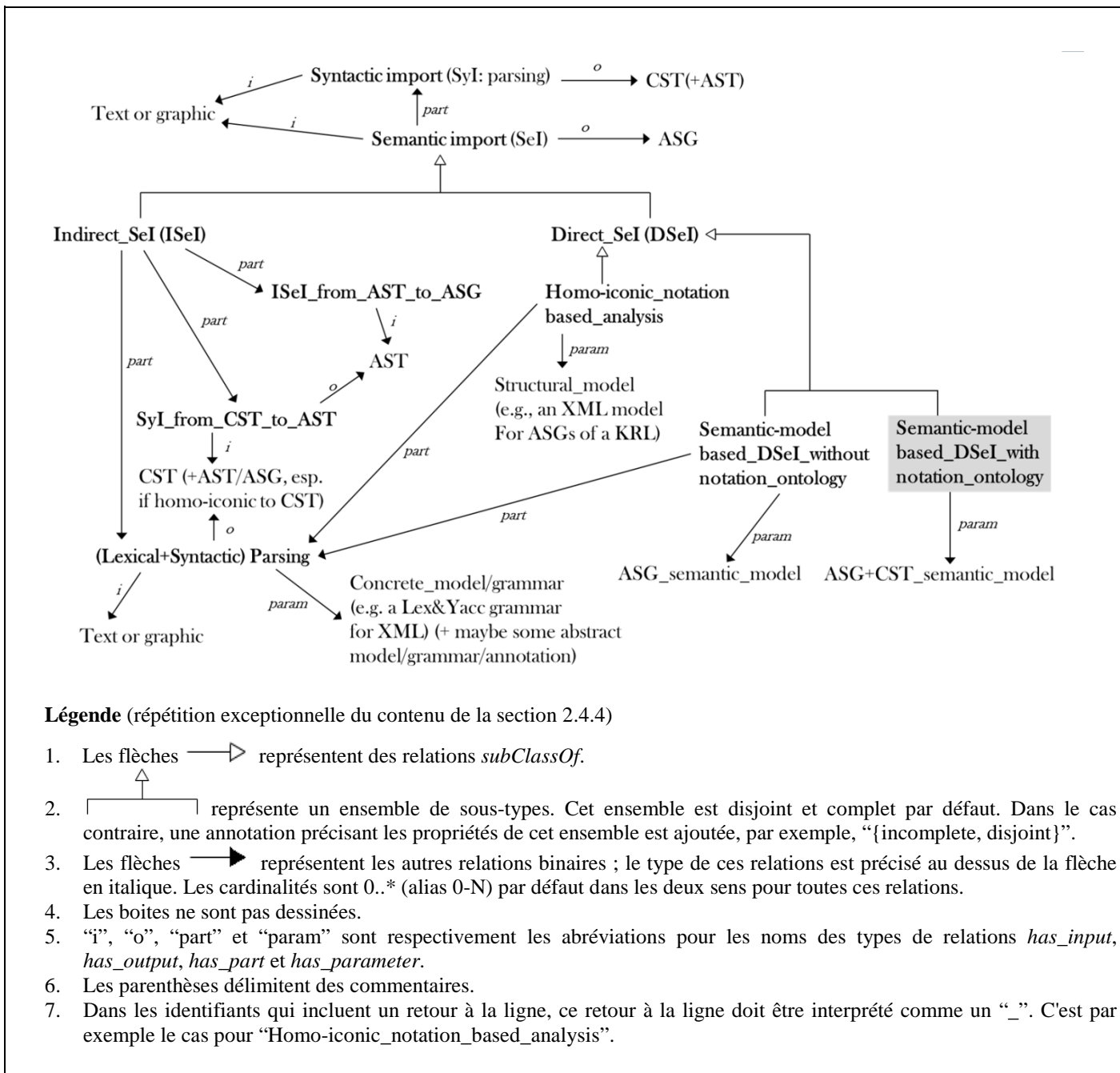
3. Approches classiques pour l'import, l'export et la traduction de LRCs

3.1. Import de connaissances

Cette section décrit les approches existantes pour les imports syntaxiques ou sémantiques.

La figure ci-dessous présente quelques types de processus d'imports avec quelques relations telles que les relations *input* ou *output*. Quelques explications sont données ensuite avec des références vers les sous-sections.

Figure 3.1. Représentation d'une tâche d'import en pm#UML.



Dans cette thèse, nous appellerons *import de LRCs* ou *import* un processus avec en entrée des représentations textuelles ou graphiques et en sortie un CST ou un AST ou un ASG. Un import de LRCs ayant en sortie un ASG peut être réalisé via un analyseur lexico-syntaxique ou un analyseur sémantique. L'utilisation de générateurs d'analyseurs lexico-syntaxiques pour l'import est examinée dans les sections “3.1.1. Import via un générateur d'analyseurs lexico-syntaxiques” et “3.1.3. Import via l'ajout d'extensions à un langage”. Des types d'analyseurs sémantiques sont décrits ci-après. Leur utilisation pour l'import sera examinée plus en détail dans les sections “3.1.2. Import via un générateur d'environnement de programmation interactif” et “3.1.4. Import exploitant des métalangages de description de structures de données”.

Un analyseur sémantique a pour partie un analyseur lexical [tokenizer/lexer] et a en sortie des AEs organisés dans un ASG. Il a en entrée, entre autres, des ECs, généralement organisés dans un CST. Un analyseur sémantique peut être direct ou indirect. Ces deux types d'analyseurs sémantiques sont décrits ci-dessous.

1. Un analyseur sémantique direct a en paramètre un modèle sémantique ou structurel. Pour tout langage représenté dans ce modèle, un tel analyseur peut extraire directement un ASG à partir d'un CST. Deux types d'analyseurs sémantiques directs sont décrits ci-dessous.
 - Un analyseur sémantique direct basé sur un modèle sémantique a en paramètre une ontologie de LRCs. Pour tout LRC représenté dans cette ontologie, un tel analyseur peut extraire un ASG à partir d'un CST en exploitant uniquement i) des représentations d'EAs et d'ECs, et ii) un moteur d'inférences. L'analyseur sémantique direct proposé dans cette thèse a en paramètre KRLO. D'autres ontologies de LRCs existent, leur utilisation pour la traduction est examinée dans la section "3.3.4. Spécifications de traductions exploitant une ontologie de langage existante". Leur utilisation pour l'import est examinée dans la section "3.1.5. Import exploitant une ontologie de langages". Une "famille de langage" peut également être vue comme une ontologie de LRCs, l'utilisation d'une telle famille pour la traduction est examinée dans la section "3.3.3. Traduction via une "famille de langages"". Comme KRLO est à ce jour la seule ontologie incluant des représentations de modèles concrets, d'autres ontologies ne peuvent pas être exploitées par un analyseur sémantique direct.
 - Un analyseur sémantique direct basé sur un langage homo-iconique a en paramètre un modèle structurel. Un modèle XML dans lequel des EAs sont déclarés est un exemple de modèle structurel. Un tel analyseur peut être utilisé uniquement pour les langages homo-iconiques ou pour les langages créés via un langage homo-iconique. L'utilisation d'un analyseur sémantique exploitant un langage modèle structurel pour importer des RCs est examinée dans la section "3.1.4. Import exploitant des métalangages de description de structures de données".
2. Un analyseur sémantique indirect a pour partie i) un générateur d'analyseurs lexico-syntaxiques, ii) un analyseur qui prend en entrée un CST et a en sortie un AST, et iii) un analyseur qui prend en entrée un AST et a en sortie un ASG. L'utilisation de spécifications de modèle concret et de modèle abstrait de langage pour paramétrer des analyseurs ayant en sortie des AST ou des ASG est examinée dans la section 3.1.2.

3.1.1. Import via un générateur d'analyseurs lexico-syntaxiques

(cf. figure 3.1, "Lexical+Syntactic Parsing")

3.1.1.1) Description

Un générateur d'analyseurs lexico-syntaxique [parser generator] prend en paramètre une grammaire abstraite et/ou une grammaire concrète et a en sortie un analyseur lexical et un analyseur syntaxique. Si un tel générateur est utilisé pour l'import, des tâches de programmation sont nécessaires pour importer des connaissances depuis cette grammaire abstraite et/ou cette grammaire concrète. Ceci limite la réutilisabilité des importeurs ainsi spécifiés (cf. section 2.2.3.1). Ceci n'est pas le cas avec l'approche proposée dans cette thèse (utilisation d'une ontologie de LRCs pour importer des connaissances depuis ces LRCs).

3.1.1.2) Exemples

Les exemples ci-dessous présentent des règles pour des grammaires concrètes ou abstraites dans différents langages. Dans ces exemples, un arbre est représenté avec la convention suivante : un noeud est entouré par des parenthèses, un arc représenté par le symbole "/" ou par le symbole "\".

1. La règle ci-dessous est issue d'une grammaire concrète dans le langage utilisé par le générateur d'analyseurs lexico-syntaxique *GoldParser*. Cette règle spécifie un type de CE nommé *term* qui a pour "parties" des ECs de "type" Numeric. "Type" et "partie" sont des relations structurelles.

```
term = {Numeric}+
```

2. La règle ci-dessous est elle aussi issue d'une grammaire concrète dans le langage utilisé par le générateur d'analyseurs lexico-syntaxique *GoldParser*. Cette règle spécifie un type de CE nommé *<expr>* qui a pour "parties" quatre ECs de type *term*, '+' ou ';'.

```
<expr> ::= term '+' term ';' ;
```

Pour une chaîne de caractère telle que "1+11;", un analyseur lexico-syntaxique avec en entrée la règle ci-dessus a en sortie le CST suivant.

```
(expr)
 /  /  \  \
(1) (+) (11) (;)
```

3. Certains générateurs peuvent prendre en paramètre une grammaire abstraite. Par exemple, dans ANTLR, des annotations sur des règles de grammaire concrètes spécifient la structure d'un AST. Ainsi, via grammaire pour ANTLR, il est possible de spécifier – pour cette chaîne de caractères : "1+1;" – l'AST ci-après.

```

      (PLUSNode = +)
      /      \
(CommonAST = 1) (CommonAST = 1)

```

Voici un exemple de règles pour spécifier un tel AST :

```

// spécification de l'analyseur lexical
// Les trois règles suivantes spécifient des types de CEs
PLUS : '+' ;
DIGIT: '0'..'9' ;
INT  : (DIGIT)+ ;

// spécification de l'analyseur syntaxique
tokens {
    PLUS<AST=PLUSNode>; // spécifie un type d'AE nommé PLUSNode
                        // qui a pour notation le type de CE
                        // nommé PLUS
    // Le type d'AE CommonAST est une représentation abstraite
    // par défaut.
}
// Un AE qui a pour notation un CE annoté par ^ a pour "parties"
// les représentations abstraites des CEs qui ne sont pas annotés par !.
expr:  INT PLUS^ INT ';' !
      ;

```

- Des traitements procéduraux peuvent être appelés lorsque des règles sont reconnues par l'analyseur syntaxique. De tels traitements sont parfois appelés des *actions*. Ces actions peuvent être utilisées pour extraire une structure de données, un AST ou un ASG. L'exemple ci-dessous présente une règle de grammaire concrète pour l'analyseur syntaxique *Yacc* ainsi que des actions.

```

expr: term '+' term ';' { rootNode= createNode($2);
                        createEdge(rootNode, createNode($1));
                        createEdge(rootNode, createNode($3)); }

```

3.1.1.3) Avantages par rapport à un processus d'import exploitant KRLO

Actuellement, un processus d'import exploitant KRLO est limité aux langages de représentation de connaissances. En effet, nous n'avons pas encore spécifié de primitives pour les éléments abstraits ou concrets des langages de programmation. Une grammaire abstraite ou concrète n'a pas ce genre de limite.

3.1.1.4) Désavantages par rapport à un processus d'import exploitant KRLO

3.1.1.4.1) Les éléments des "actions" ou d'une grammaire abstraite ou concrète peuvent difficilement être retrouvés, par exemple, via une requête

Les éléments des actions ou des grammaires abstraites ou concrètes sont décrits dans un langage qui n'est pas un LRC. Ainsi, ces éléments – par exemple, les règles de la grammaire ou les procédures des actions – ne sont pas des objets sémantiquement formels. En effet, ils sont reliés entre eux par relations structurelles uniquement. Par exemple, dans GoldParser, les éléments des règles sont reliés par des relations structurelles "::<=" ou "=" et les éléments des actions sont des DAFP. Ces éléments sont donc difficilement comparables de façon automatique (cf. Comparaison sémantique). Ils sont donc également difficilement réutilisables via des requêtes (cf. Réutilisabilité sémantique). Un exemple illustrant la difficulté de comparer ces éléments est donné ci-dessous. Dans cet exemple, une relation structurelle "::<=" est spécifiée depuis `Lisp_like_function` vers i) `Standard_lisp_like_function` et, ii) chacun des éléments suivant : `(' term Param_list')`. Dans le cas i) la relation structurelle "::<=" devrait être interprétée comme une relation *subtype*. Dans le cas ii) la relation structurelle "::<=" devrait être interprétée comme une relation *part*.

```

Lisp_like_function
 ::= Standard_lisp_like_function // subsumed by Standard_lisp_like_function
 | '(' term Param_list ')' // parts of Lisp_like_function

```

3.1.1.4.2) Des règles et des "actions" doivent être ré-adaptées pour chaque nouveau langage

Pour générer un analyseur pour un nouveau langage, un modélisateur doit créer de nouvelles règles et de nouvelles actions. Il peut adapter les règles d'une grammaire existante créée pour une notation proche de celle de ce nouveau langage. Avec de l'expérience, cette tâche est rapide, elle peut durer une journée, par exemple. C'est une tâche de modélisation (cf. 2.2.3.1.1.1). Lorsque les règles sont sensibles aux différences de modèle abstrait, comme via ANTLR, l'adaptation est plus difficile. Les "actions" sont sensibles aux différences de notation et aussi aux différences de modèle abstrait des langages. Elles peuvent

difficilement être réutilisées et leur conception est une tâche de programmation (cf. “[2.2.3.2](#) Le langage utilisé pour créer le modèle exécutable est un langage de programmation”).

3.1.1.4.3) Comparaison avec KRLO

Avec notre approche, spécifier la structure abstraite ou concrète d'un nouveau langage peut être fait *uniquement* en copiant et en adaptant une représentation déjà existante d'un langage similaire. Aucune tâche similaire à une tâche de programmation n'est nécessaire.

L'exemple en FL ci-dessous présente deux sous-type du type d'EA Typing ; l'un de ces sous-types est dans le modèle RIF-FLD [RIF-FLD, 2013], l'autre est dans le modèle RIF-BLD [RIF-BLD, 2013]. Dans cet exemple, le sous-type de Typing dans le modèle RIF-BLD est spécifié sans tâche de programmation. Seule une relation *spécialisation* (>) et une relation *r_argument* sont spécifiées. Ainsi, cet exemple montre comment un type d'EA peut être spécifié via l'héritage et l'adaptation de la définition d'un super-type.

```
f_in_(Typing, RIF-FLD) //fonction qui renvoie le sous-type de Typing dans le modèle RIF-FLD
// En RIF-FLD, l'opérateur de Typing s'écrit "#", et
// a pour arguments des individus de type rif-fld#Termula.
= f_link_type _("#", rif-fld#Termula, rif-fld#Termula);

f_in_(Typing, RIF-BLD)
< f_in_(Typing, RIF-FLD) //“Typing dans RIF-BLD” hérite de “Typing dans RIF-FLD”
r_argument =: 1..* rif-bld#Term; //En RIF-BLD, Typing a pour arguments des individus
//de type rif-bld#Term.

rif-fld#Termula > rif-bld#Term;
```

Des changements simples, comme des changements de notation, peuvent être faits en quelques minutes. Par exemple, donner une forme préfixée ou infixée ou postfixée à la structure des AEs comme avec : "3 = 2 + 1" (notation infixée), "= (3 +(2 1))" (notation préfixée) ou "(3 (2 1)+)=" (notation postfixée). Les utilisateurs finaux peuvent donc adapter les notations suivant leurs préférences ou les outils qu'ils utilisent. Par exemple, il est possible de travailler sur un langage avec un modèle RDF et une syntaxe proche de celle de lisp mais dans laquelle les parenthèses sont remplacées par des accolades.

3.1.2. Import via un générateur d'environnement de programmation interactif

(cf. figure 3.1, “Indirect_Sel”)

3.1.2.1) Description

Une approche un peu différente de celle présentée dans la section précédente ([3.1.1](#)) consiste à utiliser un outil tel qu'un générateur d'environnements interactifs.

Un environnement de programmation interactif a pour partie un processus d'import – par exemple, un analyseur lexico-syntaxique – et une interface utilisateur. Un tel environnement peut analyser le code écrit par un programmeur et interagir avec lui. Par exemple, des erreurs peuvent être détectées avant la compilation d'un programme et être présentées au programmeur via une interface graphique. Cet environnement peut ainsi faciliter les tâches de programmation.

Un générateur d'environnement interactif prend en paramètre une spécification formelle d'un langage. Il a en sortie un environnement de programmation interactif. Ainsi, après avoir spécifié un LRC, un programmeur peut exploiter un tel générateur pour importer des connaissances. Cette spécification se fait soit via une grammaire et des actions, soit via un LRC exécutable (par exemple, Prolog). Contrairement aux spécifications de LRCs présentées dans le chapitre 4 de cette thèse, de telles spécifications décrivent un modèle exécutable et sont donc moins génériques (cf. section [2.2.2.](#)).

3.1.2.2) Exemple

Centaur [Borras et al., 1988] est un exemple de générateur d'environnement interactif. Il prend en paramètre une grammaire concrète, une grammaire abstraite, des règles de conversions entre ces deux grammaires et une spécification de la sémantique opérationnelle du langage. Les grammaires concrète et abstraites, ainsi que les règles de conversion sont écrites dans le langage Metal [Kahn, Lang, Mèlèse, 1983]. Centaur a en sortie un éditeur de texte, un vérificateur de types et un interpréteur et donc également un analyseur lexico-syntaxique. Des actions sont associées à cet analyseur afin qu'il ait en sortie un AST. Dans un environnement généré via Centaur, tous les éléments abstraits sont organisés dans un AST.

La sémantique opérationnelle est écrite dans le langage Typol [Despeyroux, 1988]. Sa compilation produit des règles dans le langage Prolog. Un système de communication vers un serveur Prolog a été mis en place afin que le moteur Prolog puisse servir

de moteur d'inférences. Ainsi, il est possible d'envoyer au serveur Prolog un but dont un paramètre est un AST de l'environnement généré.

Centaur a été utilisé principalement pour des langages de programmation mais également pour un LRC [Corby & Dieng, 1996].

3.1.2.3) Avantages par rapport à un processus d'import exploitant KRLO

KRLO est encore loin d'avoir des spécifications déclaratives pour toutes les fonctionnalités que ce type d'outil peut fournir via son code procédural ou logique (comme avec Prolog). Par exemple, il n'y a pas encore de spécifications pour la sémantique formelle des langages de programmation dans KRLO, ni pour les représentations graphiques des éléments de langage.

3.1.2.4) Désavantages par rapport à un processus d'import exploitant KRLO

Spécifier de nouveaux langages via une grammaire et des actions ou via un LRC exécutable – comme Prolog – est plus long et difficile que spécifier de nouveaux langages dans une ontologie. Ces difficultés sont explicitées dans les sections [2.2.2](#) et [2.2.3](#), ci-avant. Or, dans les générateurs d'environnement de programmation interactifs, les langages utilisés pour les spécifications sont des langages d'opérationnalisation. Ce ne sont pas des langages de modélisation. Ainsi, les spécifications sont difficiles à organiser et à réutiliser (cf. section [2.2](#), ci-avant). De petits changements dans les grammaires concrètes et abstraites mènent souvent à des changements importants dans les spécifications.

3.1.3. Import via l'ajout d'extensions à un langage

(cf. figure 3.1, “Indirect_SeI”)

3.1.3.1) Description

Pour donner un peu de flexibilité aux utilisateurs finaux, certains langages leur autorisent à étendre la syntaxe concrète de base. Dans la suite de cette section, nous appellerons *extension-utilisateur* les extensions qui sont créées par des utilisateurs finaux. Contrairement aux extensions qui peuvent être créées par des concepteurs, les extensions de ce type ne nécessitent pas la recompilation d'un interpréteur du langage.

Pour interpréter des phrases écrites via une syntaxe étendue par des utilisateurs finaux, des macros, des parseurs intégrés, des fonctions d'ordre supérieur et des fonctions interprétant des phrases générés peuvent être utilisées. Ce dernier type de fonctions est également appelé évaluateur méta-circulaire. Les fonctions *eval* et *apply* des langages de programmation sont des exemples de tels évaluateurs. Ainsi, un interpréteur du langage peut interpréter des phrases dont la notation ne correspond pas à celle du langage avant l'extension-utilisateur. Dans la plupart de ces langages, les extensions-utilisateur sont très limitées. Par exemple, certains symboles sont réservés et ne peuvent donc pas être utilisés pour un usage différent dans ces extensions.

3.1.3.2) Exemple

XBNF [Botting, 2012] est un cas extrême dans cette approche. C'est un LRC (cf. justification 1. ci-dessous) qui permet la spécification d'extensions-utilisateur (cf. justification 2. ci-dessous). Un interpréteur du LRC XBNF est aussi un générateur d'analyseurs lexico-syntaxiques prenant en paramètre des *spécifications écrites dans le langage de spécification de syntaxe concrète XBNF*. Ainsi, des RCs écrites dans toute extension-utilisateur du LRC XBNF peuvent être interprétées.

1. En XBNF, une règle syntaxique est aussi une définition de type. Pour ces définitions, un utilisateur peut exploiter i) des relations logiques usuelles telles que "ET" ou "OU", ii) des ensembles, et iii) des fonctions.
2. Comme EBNF, XBNF permet à un utilisateur de spécifier des syntaxes concrètes de langages (cf. modèle concret).

3.1.3.3) Avantages par rapport à un processus d'import exploitant KRLO

Exploiter des extensions-utilisateur d'un langage pour réaliser un processus d'import a des avantages d'ordre sociaux. Par exemple, les utilisateurs déjà familiers d'un langage peuvent avoir des facilités pour travailler avec le langage étendu.

3.1.3.4) Désavantages par rapport à un processus d'import exploitant KRLO

Quel que soit le langage, des extensions-utilisateur peuvent être utilisées pour changer la structure des EAs mais ne peuvent pas être utilisées pour changer leur sémantique. L'approche présentée dans cette thèse est une solution à ce problème. En effet, tout EA peut être spécifié dans KRLO via quelques primitives et toutes ces spécifications peuvent être exploitées par un processus d'import. Pour l'instant, KRLO inclut uniquement des éléments de LRCs. À court terme, une extension pour quelques langages de description de structures de données comme JSON sera conçue après l'achèvement de cette thèse. À long terme, une extension pour quelques langages de programmation comme Prolog ou Lisp sera conçue après l'achèvement de cette thèse. Cette extension inclura i) des primitives pour représenter des EAs ou des Ecs de langages de programmation, et ii) des spécifications pour les modèles abstraits ou concrets de quelques langages de programmation.

3.1.4. Import exploitant des métalangages de description de structures de données

(cf. figure 3.1, “Homo-iconic_notation_based_analysis”)

3.1.4.1) Description

Certains métalangages – par exemple, XML – forcent les utilisateurs à écrire le nom ou le type de tous les EAs. Avec ces métalangages, les notations et les modèles sont homo-iconiques (cf. section “[2.3](#). Précisions et exemples pour l'homo-iconicité”). Par exemple, la notation RDF/XML est homo-iconique au modèle RDF. Le métalangage MOF (Meta-Object Facility) [MOF, 2016] proposé par l'OMG et ses notations – par exemple, MOF-HUTN [HUTN, 2004] – est un autre métalangage de ce type. D'autres métalangages du même genre sont utilisés en ingénierie dirigée par les modèles (IDM) [MDE].

Des RCs écrites avec des notations créées via un métalangage peuvent facilement être importées ou exportées via des outils assez génériques. Par exemple, lorsque RDF/XML est utilisé comme notation de LRC i) un analyseur XML tel que SAX, peut être utilisé pour analyser des RCs, ii) des fonctions d'export peuvent être créées via XSLT, et iii) certains éléments de présentation peuvent être spécifiés via CSS. Cependant, de telles notations n'offrent que des relations structurelles entre leurs éléments. Ainsi, pour chacune de ces notations, un processus d'import doit être spécifié et cette spécification nécessite des tâches de programmation (cf. section [2.2.3.1](#)).

3.1.4.2) Avantages par rapport à un processus d'import exploitant KRLO

L'adoption de solutions qui améliorent des outils déjà existants est plus simple que l'adoption de solutions entièrement nouvelles. Certaines notations créées via un métalangage sont i) standardisées par des organismes comme le W3C ou l'OMG, et ii) utilisées par un grand nombre de personnes. Les outils d'import ou d'export exploitant ces notations sont eux aussi assez répandus. KRLO et les outils qui l'exploitent n'ont pas actuellement ces avantages mais en ont d'autres (cf. section ci-dessous).

3.1.4.3) Désavantages par rapport à un processus d'import exploitant KRLO

Les notations de description de structure – comme celles créées via XML – n'offrent que des relations structurelles entre leurs éléments. Ainsi, les outils pour importer des RCs depuis ces langages peuvent difficilement extraire des relations sémantiques à partir des représentations textuelles ou graphiques. Par exemple, un analyseur lexico-syntaxique standard pour XML tel que SAX n'est pas capable d'extraire un ASG à partir d'une description écrite en RDF/XML, ni même à partir d'une description écrite en XML et suivant la norme *Alternating Normal Form* (cf. section [2.2.3.1.1.1](#)). Pour extraire de telles relations, ces outils doivent être adaptés via des tâches de programmation. Pour plus de détails concernant la difficulté de ces tâches, le lecteur peut consulter ces parties : “[2.2](#). Difficulté des tâches de modélisation et/ou de programmation pour le partage de connaissances” et “[2.2.3.2](#). Le langage utilisé pour créer le modèle exécutable est un langage de programmation”). Les outils disponibles pour importer depuis ou exporter vers ces langages ne réalisent pas d'inférence logique. Les conséquences de ces inférences doivent donc être programmées.

Le point précédent est aggravé par le fait que les descriptions sont souvent i) trop peu concises pour être facilement compréhensible par un humain, ou ii) de trop bas niveau pour être utilisées directement. Par exemple, nous ne connaissons aucun SGBC (Système de gestion de BC) utilisant des objets XML en interne.

L'approche décrite dans cette section n'élimine pas la nécessité de créer des analyseurs et des fonctions ou règles d'exports pour plusieurs notations. Certains outils pour le MDE sont décrits comme ayant des notations en entrée extensibles, par exemple, BAM [Feja *et al.*, 2011] dans le domaine de la modélisation de processus (process modelling). En réalité, ils gèrent un modèle expressif qui inclut les primitives pour plusieurs langages de modélisation de processus déjà existants. Ainsi, ils peuvent gérer chacun d'eux. Des traductions depuis ou vers d'autres modèles ou notations restent nécessaires.

3.1.5. Import exploitant une ontologie de modèles abstraits de LRCs

(cf. figure 3.1, “Semantic-model_based_DSeI_without_notation_ontology”)

3.1.5.1) Description

Via une ontologie de modèles abstraits, une fonction d'import peut être paramétrée de sorte que toute RC puisse être importée vers tout modèle abstrait représenté dans cette ontologie. Cependant, sans une ontologie de notations, un langage particulier est nécessaire pour spécifier des notations particulières pour des EAs. A notre connaissance, aucune ontologie – autre que KRLO – définissant des notations n'existe. Des ontologies de modèles abstraits autres que KRLO existent ; par exemple, l'OMG (Object Management Group) a publié un modèle UML et un modèle XML pour ODM (Ontology Definition Meta-model) [ODM, 2014]. L'utilisation de telles ontologies pour la traduction est présentée dans la section “[3.3.4](#). Spécifications de traductions exploitant une ontologie de langage existante”.

Un outil d'import exploitant l'ontologie de modèles abstraits LATIN Atlas of logics [Codescu et al., 2011] – qui organise des EAs de plusieurs logiques et LRCs – est présenté dans la section ci-dessous. La section “3.3.4.3. HETS avec LATIN et DOL” donne plus de détails sur la conception de l'ontologie LATIN et sur l'organisation des EAs dans cette ontologie.

3.1.5.2) Exemple

MMT (Meta-Meta-Theory) [Rabe & Kohlhase, 2013] est un langage qui permet de spécifier des modèles de LRCs et des notations pour des EAs de ces modèles via une méta-théorie. Pour éviter de gérer des spécifications de notation concurrentes, MMT permet également à l'utilisateur d'utiliser des relations “précédence” depuis une spécification de notation vers un entier. L'utilisateur peut ainsi ordonner les spécifications de notation de façon absolue. Une spécification d'un LRC nommé Logic et une spécification d'un LRC nommé PLSyntax (Propositional Logic Syntax) sont présentées ci-dessous. Ces spécifications sont extraites de la documentation en ligne de MMT (cf. <http://uniformal.github.io/doc/tutorials/jedit/2theories.html>). Pour clarifier ces spécifications, j'ai ajouté des commentaires délimités par les symboles “/*” et “*/”.

```
/*Extrait 1 : définition d'un nouveau LRC nommé "Logic"
   avec la méta-théorie LF (Logical Framework) [Harper et al., 1987]
*/
theory Logic : ur:?LF = /*ur est l'espace de noms http://cds.omdoc.org/urtheories*/
/*Les spécifications ci-dessous sont structurées de la façon suivante :
   spécification d'un EA pour le LRC Logic # spécification de la notation de cet EA
*/
prop : type # 0 /*l'EA prop (proposition) est un type (défini dans LF)
                 et est noté "o"*/
proof : o → type # 1 prec -100 /*l'EA proof a pour :
                                 partie une proposition,
                                 résultat un type,
                                 notation :
                                 le symbole → suivi par
                                 la notation de sa première partie*/
```

```
/*Extrait 2 : définition d'un nouveau LRC nommé "PLSyntax"
   avec la méta théorie LF
*/
theory PLSyntax : ur:?LF =
  include ?Logic
  True : o # 1 /*l'EA True est une proposition (le type proposition est
                défini dans le LRC Logic) et a pour notation "1"*/
  False : o # 0
  not : o → o # 1 prec 50
  and : o → o → o # 1 ∧ 2 prec 45 /*l'EA and a pour :
                                     partie deux propositions,
                                     résultat une proposition,
                                     notation :
                                     la notation de sa première partie suivie par
                                     le symbole "∧" suivi par
                                     la notation de sa seconde partie*/
  or : o → o → o # 1 ∨ 2 prec 40
  imp : o → o → o # 1 ⇒ 2 prec 35
  equiv : o → o → o # 1 ⇒ 2 prec 30
```

Comme le montre l'extrait ci-dessus, la notation MMT est très concise mais a les défauts suivants.

- Entre un EA et ses parties, la notation MMT ne permet de spécifier que peu de relations sémantiques. En effet, seules des relations part et result peuvent être spécifiées. À titre de comparaison, trois principaux types de relations et plusieurs sous-types sont spécifiés dans KRLO (cf. “4.1.1.2. Opérateur, arguments et résultat”).
- Entre un EA et un EC, la notation MMT permet de spécifier qu'une relation structurelle, elle ne permet pas de spécifier une relation sémantique.
- La notation MMT ne permet pas de spécifier des notations de façon modulaire car les parties relatives à la notation ne peuvent être isolées des parties relatives au modèle. Ainsi, pour chaque notation, un utilisateur doit spécifier un nouveau LRC.

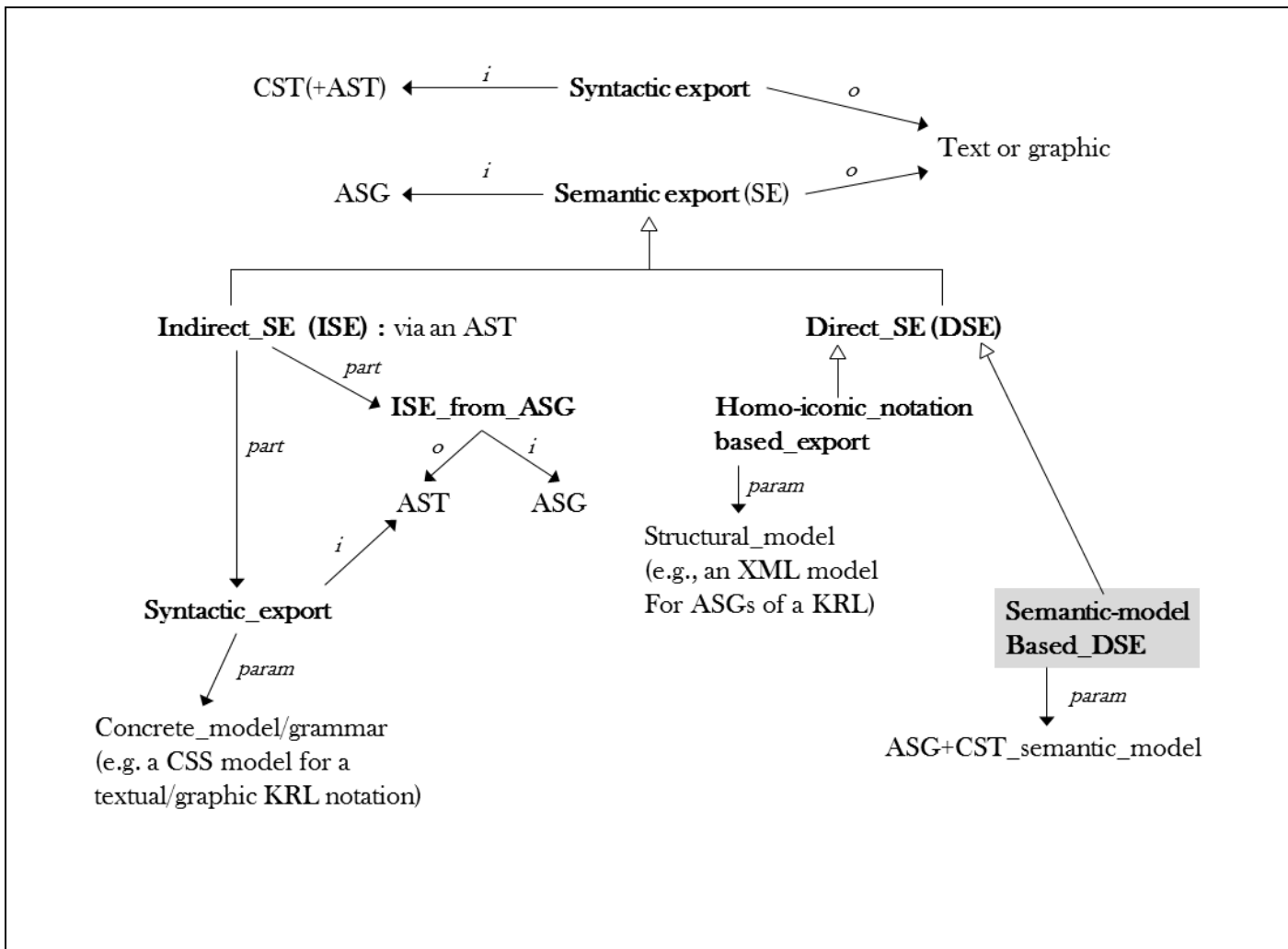
3.1.5.3) Désavantages par rapport à un processus d'import exploitant une ontologie telle que KRLO (i.e. une ontologie de modèles abstraits et de notations)

Puisqu'une ontologie de définitions de notations n'est pas utilisée, la spécification de notations ne peut être faite que via un langage particulier (cf. exemple ci-dessus). Les spécifications ne sont donc interprétables que par quelques outils qui exploitent un interpréteur pour ce langage. Ceci est une limite à la réutilisabilité de ces spécifications. De plus, comme les spécifications de notations ne sont pas organisées dans une ontologie, elles peuvent difficilement être comparées (cf. comparaison sémantique) et donc retrouvées (cf. réutilisabilité sémantique). Ainsi, sans ontologie de notations, un utilisateur ne peut pas facilement rechercher des spécifications de notations particulières et donc les réutiliser, par exemple pour spécifier de nouvelles notations via l'héritage et/ou le paramétrage d'une spécification existante.

3.2. Export de connaissances

Dans cette thèse, un processus d'export est un processus qui prend en entrée des EAs et a en sortie un fichier contenant des éléments textuels ou graphiques. La figure ci-dessous présente quelques types de processus d'exports avec quelques relations telles que les relations *has_input* ou *has_output*.

Figure 3.2. Représentation d'une tâche d'export en pm#UML.



Comme indiqué dans la figure ci-dessus, un processus d'export sémantique direct basé sur un modèle sémantique prend en paramètre une ontologie de modèles abstraits et de notation de LRCs. Ainsi, un tel processus peut être utilisé pour exporter des RCs depuis tout modèle représenté dans cette ontologie vers toute notation également représentée dans cette ontologie. En dehors de KRLO, nous n'avons trouvé aucune ontologie de notation même pour un LRC standard tel que RDF. Par conséquent, il apparaît qu'il n'y a également aucun autre exporteur sémantique basé sur un modèle sémantique.

Dans quelques outils – tel Centaur – des fonctions d'export prennent en paramètre des modèles concrets et abstraits qui sont décrits via un langage de description de structure. Par exemple, dans Centaur, des fonctions d'export prennent en paramètre des grammaires abstraites ou concrètes. Comme expliqué dans la section “2.2.3. Création directe d'un modèle exécutable via un langage qui n'est pas un LRC”, ces modèles abstraits ou concrets sont difficilement réutilisables pour le partage de connaissances et nécessitent des tâches de programmation qui peuvent être longues et difficiles.

Certains modèles de LRCs ont une notation basée sur un méta-langage de description de structure comme MOF ou XML. Dans ce cas, des langages de manipulation de structure spécifique comme Mof2Text [Mof2Text, 2008], XSLT et CSS peuvent être utilisés pour spécifier des fonctions d'export. Ainsi, plusieurs auteurs ont proposé des langages pour spécifier la façon dont des EAs de RDF peuvent être présentés : dans une certaine notation, dans un certain ordre, en gras, dans une fenêtre, etc. Xenon [Quan, 2005], Fresnel [Pietriga et al., 2006], OWL-PL [Brophy & Helfin, 2009] et SPARQL Template [Corby & Faron-Zucker, 2015] [Corby et al., 2015] sont des exemples de tels langages. Comme expliqué dans la section “2.2.3.1. Le langage utilisé pour créer le modèle exécutable est un langage de description dont la structure peut être manipulée via des outils ou un langage”, les fonctions d'export ainsi spécifiées sont difficilement réutilisables et peuvent nécessiter des tâches de programmation.

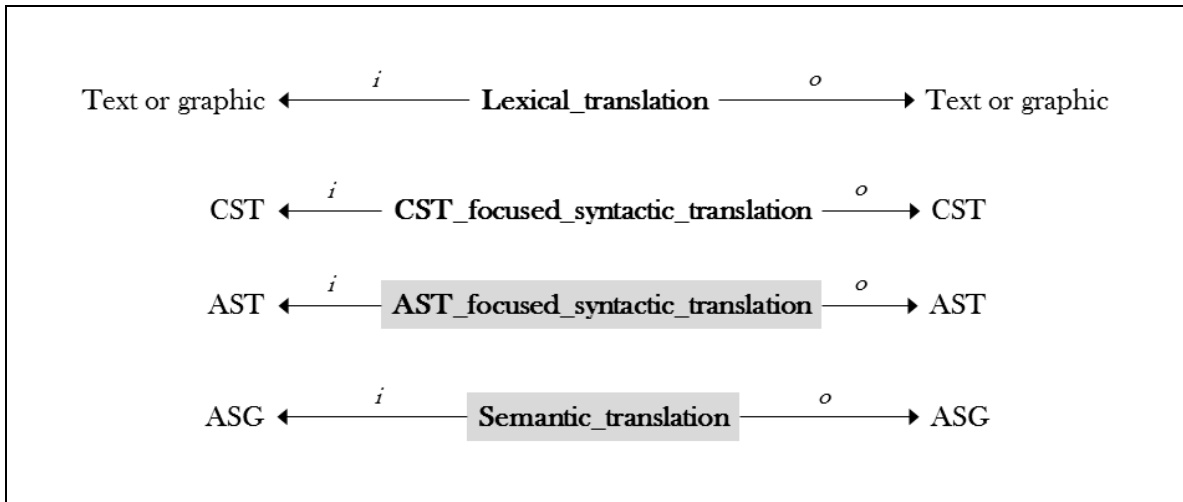
KRLO ne dépend pas d'un langage ou LRC particulier. Certains des langages listés dans le paragraphe précédent pourraient donc être utilisés conjointement à KRLO. Par exemple, des EAs de KRLO peuvent être utilisés dans des requêtes SPARQL Template. Pour cela, KRLO doit être représenté en RDF+OWL2-RL [OWL2, 2012] ou SWRL [Horrocks et al., 2004] qui est un LRC basé sur RDF pour représenter des règles de Horn. Les règles de traduction représentées dans KRLO ne peuvent cependant pas être représentées en RDF+OWL2-RL car elles nécessitent l'emploi d'un quantificateur existentiel dans la partie conclusion de la règle.

Si aucune description du modèle abstrait et du modèle concret n'est fournie, un programmeur doit spécifier une fonction d'export via un langage de programmation. Les difficultés liées à l'écriture de ces spécifications et à leur réutilisation sont décrites dans la section "2.2.3.2. Le langage utilisé pour créer le modèle exécutable est un langage de programmation."

3.3. Traductions

Dans cette thèse, un processus de *traduction sémantique* (entre EAs) prend en entrée un modèle source, a en sortie un modèle cible. D'autres types de processus de traductions existent. Par exemple, [Euzénat, 2001b] distingue les types de processus de traduction suivants : “traduction lexicale”, “traduction syntaxique”, “traduction sémantique” et “traduction sémiotique” (traduction pragmatique). La traduction sémiotique/pragmatique réfère aux bonnes pratiques d'écriture de RCs et dépasse donc le cadre de cette thèse. La figure ci-dessous présente les différents types de processus de traduction qui seront comparés dans cette section (3.3).

Figure 3.3. Représentation de tâches de traduction en pm#UML.



Comme le montre la figure ci-dessus, des fonctions ou règles de traduction lexicale ont en entrée et en sortie des éléments graphiques ou des éléments textuels. Des fonctions ou règles de traduction syntaxique ont en entrée et en sortie des ECs ou des *EAs liés à la syntaxe* et généralement organisés dans un AST. Des fonctions ou règles de traduction sémantique ont en entrée et en sortie des EAs typiquement organisés dans un ASG. Cette section (3.3.) décrit les approches existantes pour les traductions syntaxiques ou sémantiques d'EAs.

3.3.1) Les entrées d'un processus de traduction sont fournies par d'autres processus

Un processus de traduction ne gère pas d'import de RCs. Les EAs en entrée d'un processus de traduction peuvent être importés via l'un des processus examinés dans la section “3.1. Approches classiques pour l'import”.

3.3.2) Description de cette section (3.3.) avec des références vers ses sous-sections et quelques éléments de comparaison

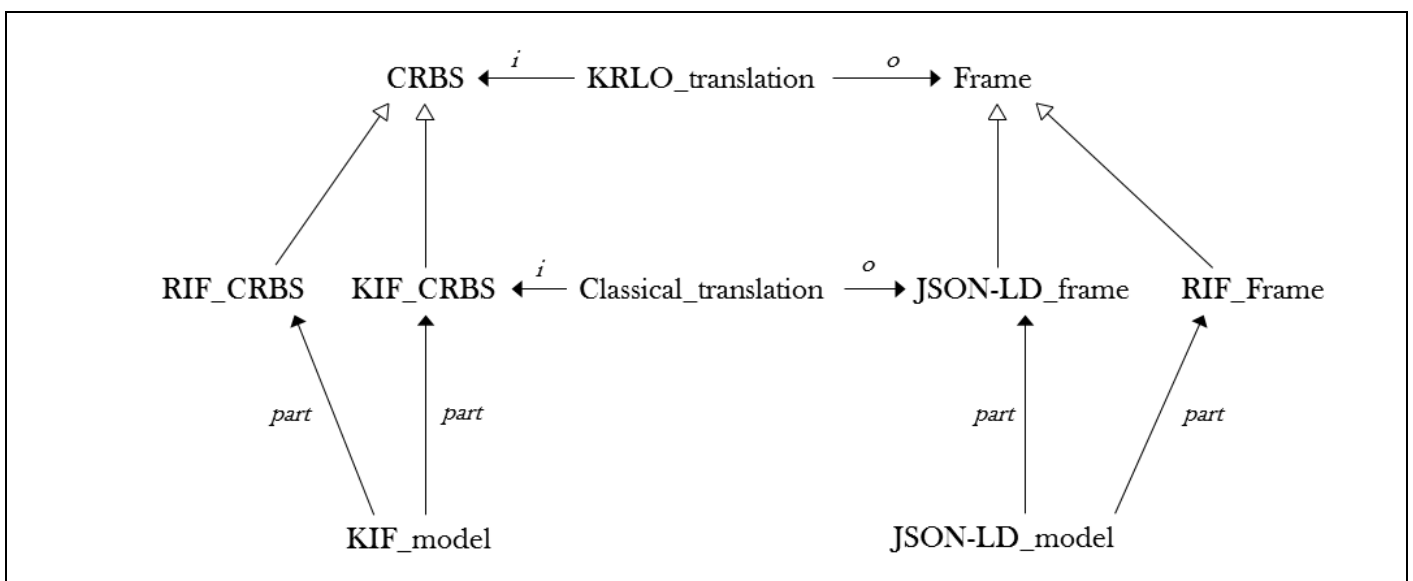
Les approches existantes pour les traductions syntaxiques ou sémantique d'EAs se distinguent par i) les types de règles ou fonctions utilisées, et ii) par le type des relations exploitées. Dans cette thèse, je distingue les quatre approches suivantes pour la spécification de processus de traduction de LRCs et donc également de fonctions ou règles de traduction d'EAs.

- Une première approche consiste à spécifier pour chaque couple de LRC une fonction ou règle de traduction depuis chaque EA du modèle source vers un EA du modèle cible. Ainsi, dans le pire des cas, pour N modèles de langage, il faut spécifier $N*(N-1)$ fonctions ou règles de traduction. Cette approche est présentée dans la section “3.3.1. Spécifications de traductions directes”.
- Une seconde approche consiste à spécifier des fonctions ou règles de traduction depuis ou vers un unique modèle de langage. Nous appellerons alors ce modèle *modèle pivot*. Dans la littérature, une *notation pivot* est souvent également utilisée pour faciliter l'import et l'export, c'est pourquoi la dénomination *langage pivot* est davantage utilisée. Cette approche est présentée dans la section “3.3.2. Spécifications de traductions vers ou depuis un langage pivot”.
- Une troisième approche consiste à exploiter une structure particulière de modèles de LRCs appelée “famille de langages” dans [Euzénat & Stuckenschmidt, 2003]. Dans cette structure, des EAs de ces modèles – et donc également ces modèles eux-mêmes – sont reliés par des “relations de traduction”. Chacune de ces relations est associée à une fonction ou règle de traduction. Cette approche est présentée dans la section “3.3.3. Traduction via une “famille de langages””. Les structures en “couches” pour les modèles de LRCs sont généralisées par les structures de type “famille de langages”. En effet, dans une structure en “couches”, les langages sont organisés par ordre d'expressivité croissante et des traductions sont spécifiées entre chaque niveau d'expressivité. C'est, par exemple, le type de structure adopté par le W3C.
- La quatrième approche consiste à exploiter une ontologie de LRCs. Dans une telle ontologie, des EAs sont représentés formellement. Une originalité de KRLO – l'ontologie présentée dans cette thèse – est de proposer également des règles de traduction entre des *EAs structurels*. Par exemple, KRLO propose une règle pour traduire un EA représentant une “structure de relations binaires” en un EA représentant une “structure de relations n-aires”. La section “6.2. Traduction

entre éléments abstraits” donne davantage de détails sur ces traductions. La section “4.3. Expressivité” fournit des détails sur l’expressivité qu’un moteur d’inférence doit pouvoir gérer pour réaliser ces traductions. D’autres ontologies de LRCs sont présentées dans la section “3.3.4. Spécifications de traductions exploitant une ontologie de langage existante”. La section “3.3.4.2. Spécifications de traductions via Hets avec Latin et DOL” présente une approche similaire à la notre mais centrée sur la traduction entre des *EAs logiques* dont la sémantique est issue de la théorie des modèles ou de la théorie de la preuve.

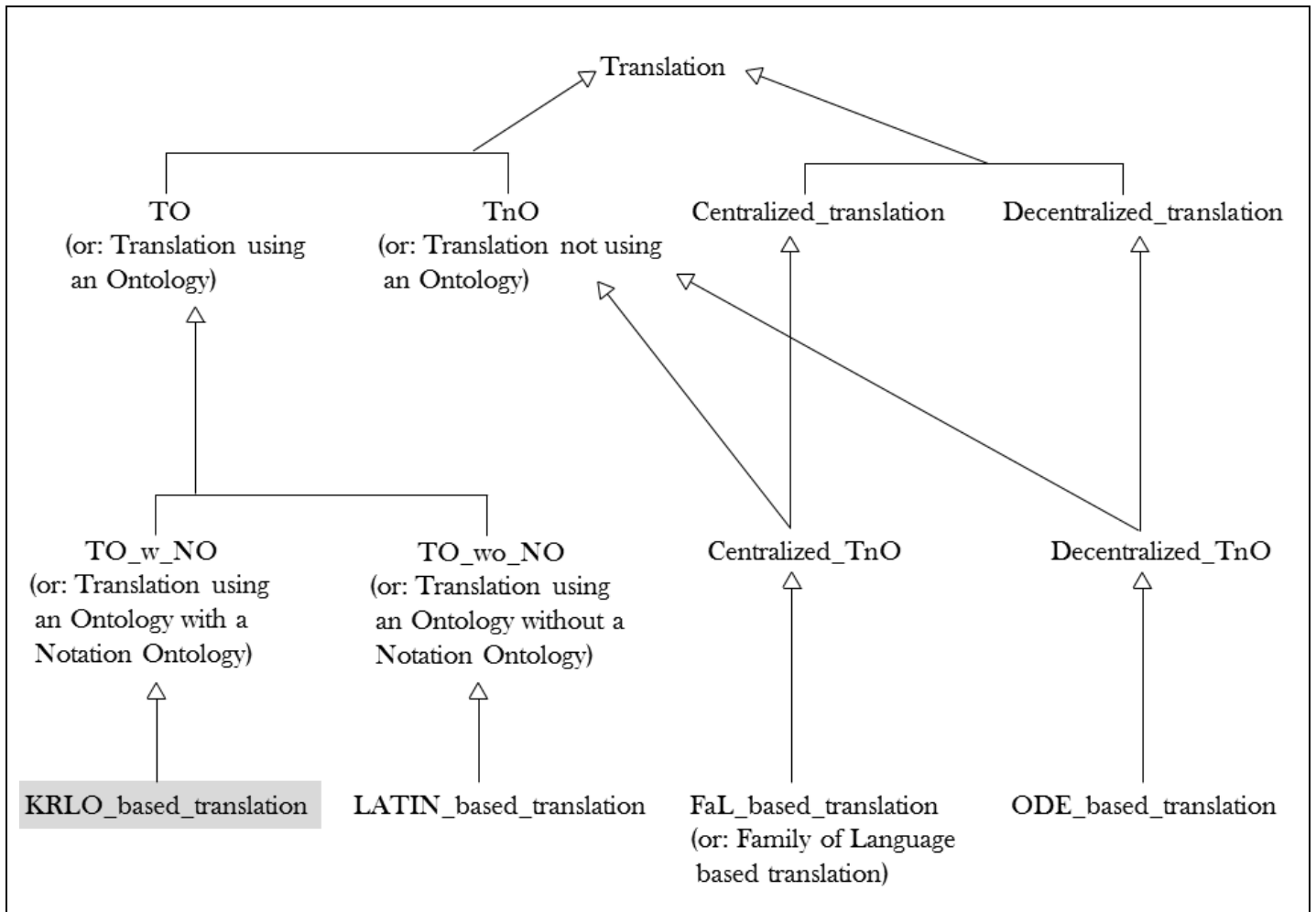
Les trois premières approches décrites ci-dessus exploitent des fonctions ou règles de traduction depuis un type d’AE d’un modèle vers un type d’AE d’un autre modèle. Par exemple, pour spécifier un processus de traduction depuis le modèle de KIF vers le modèle de JSON-LD, un utilisateur doit spécifier une fonction ou règle de traduction depuis une “conjonction de relations binaires depuis une même source, exprimée en KIF” vers un “frame, exprimé en JSON-LD”. Par contre, dans la 4e approche et plus précisément dans celle qui est préconisée dans cette thèse, des règles ou fonctions de traduction peuvent être spécifiées entre des types d’AEs, comme illustré dans la figure ci-dessous. Par exemple, un utilisateur peut utiliser une fonction ou règle de traduction depuis une “conjonction de relations binaires depuis une même source” vers un “frame”. Ainsi, ces règles ou fonctions peuvent être ré-utilisées pour tout AE représenté dans KRLO.

Figure 3.4. Comparaison entre tâche de traduction dans KRLO et tout autre type de tâche de traduction (représentée) en pm#UML.



Les approches pour la traduction peuvent être représentées et organisées dans une ontologie. La figure ci-après présente une telle organisation. Les sous-sections suivantes (3.3.1, 3.3.2, 3.3.3 et 3.3.4) montrent que les approches qui n’utilisent pas une ontologie de modèle et une ontologie de notation ne peuvent pas être génériques. Ici, “être générique” signifie permettre l’utilisation d’une unique fonction de traduction pour tout LRC en paramètre sans pour autant que la description de ces LRCs en paramètre fournisse une traduction explicite vers tout autre LRC.

Figure 3.5. Catégorisation des différents types d'approches pour la traduction (représentée) en pm#UML.



3.3.1. Spécifications de traductions directes

(cf. figure 3.5, “Decentralized_TnO”)

3.3.1.1) Description

Dans cette approche (i.e., spécifications de traductions directes), un processus de traduction est spécifié pour chaque paire de modèle de LRC. Pour chaque couple “(AE source, AE destination)” une nouvelle fonction ou règle de traduction doit être spécifiée. Le langage utilisé pour ces spécifications est un langage de *programmation* (fonctionnel, impératif ou logique). Ainsi, si N est le nombre de modèles de langage, le concepteur ou l'utilisateur d'un outil de traduction utilisant cette approche doit spécifier $N*(N-1)$ processus de traduction. Ceci rend difficile le passage à l'échelle de solutions mettant cette approche en oeuvre.

3.3.1.2) Exemples

3.3.1.2.1) ODEDialect

ODEDialect [Corcho & Gomez-Perez, 2007] est un langage spécialisé pour la description de “traductions entre des EAs ou des ECs”. Les prérequis nécessaires pour l'exploitation des fonctions de traduction ainsi spécifiées sont décrits ci-dessous.

- Un transcompilateur (ou compilateur source à source) depuis ODEDialect vers Java est nécessaire. Ce transcompilateur peut être vu comme un interpréteur pour ODEDialect.
- Pour chaque langage à traduire, un outil d'édition d'ontologies tel que Protégé ou PowerLoom est nécessaire. Cet outil doit pouvoir être utilisé pour importer et exporter des RCs décrites dans ce langage et disposer d'une API Java. En effet, les descriptions ODEDialect dépendent directement de ces APIs. PowerLoom ou Protégé sont des exemples de tels outils.

Bien que ODEDialect soit déclaratif, des appels de fonctions Java sont autorisés pour réaliser des traductions qui ne peuvent pas être spécifiées via ODEDialect. Par exemple, dans l'extrait de code ci-après, une fonction Java nommée convertToURI est appelée dans la zone d'initialisation qui – en ODEDialect – est appelée "INIT".

ODEDialect a trois sous-langages : ODELex, ODESyntax et ODESem. Chacun d'eux, respectivement, est spécialisé pour la spécification de traductions sur trois niveaux : lexical, syntaxique et sémantique. Ci-dessous, un extrait de spécification d'une fonction de traduction en ODELex est présenté. Cette fonction non nommée prend en entrée un AE de type *AdHocRelation* dans *WebODE* et a en sortie un EA de type *ObjectProperty* dans *OWL*. Cet extrait ci-dessous illustre une différence entre le niveau lexical tel qu'il est décrit dans [Corcho & Gomez-Perez, 2007] et le niveau lexical tel que cette thèse le décrit. En effet, dans cette thèse, une traduction lexicale a en entrée et en sortie des éléments textuels ou graphiques. Or, dans KRLO, un identifiant est un EA. La spécification ci-dessous ne décrit donc pas une transformation lexicale au sens employé dans cette thèse.

```
%WebODE.AdHocRelation IDENTIFIER /* %1 : cette relation de type
                                AdHocRelation dans WebODE */
    WebODE.Concept IDENTIFIER // %2 : cette source de la relation
    WebODE.Concept IDENTIFIER // %3 : cette destination de la relation
INIT: {$1=convertToURI(%1)} // $1 : est l'identifiant de la relation dans OWL
/* La traduction lexicale d'une relation de type WebODE.AdHocRelation
   en une relation de type OWL.ObjectProperty se réduit une conversion
   d'identifiants. */

// Informations à transmettre aux fonctions de traduction
// des niveaux syntaxiques et sémantiques :
TABLE: {[WebODE.AdHocRelation,%1,%2,%3], [OWL.ObjectProperty,$1]}
```

ODEDialect et ses sous-langages sont orientés vers l'opérationnalisation et sont proches de langages des programmation (cf. section “2.2. Difficulté des tâches de modélisation et/ou de programmation pour le partage de connaissances”). Dans des phrases écrites avec des langages de programmation, seules des relations structurelles existent entre les éléments. C'est aussi le cas avec ODEDialect. Par exemple, dans l'extrait ci-après, la règle %AddInstanceAttributes est utilisée par la règle %AddClasses mais aucune relation n'est définie entre ces deux règles. Seul un programmeur ayant une certaine connaissance du langage est capable de comprendre que %AddInstanceAttributes est un composant – ou une partie – de %AddClasses. Le langage utilisé dans cet extrait est ODESem.

```
%AddClasses
WebODE.Concept concept --> // --> signale une règle de production
{ C = CREATE(OWL.Class, 1, concept.name);
  if (concept.description != null && !concept.isImported)
    ADD(C,description, concept.description);
  ADD(C,subClassOf, concept.parentConcepts);
  forEach ia IN concept.instanceAttributes
    // force l'exécution de la règle %AddInstanceAttributes
    EXEC(%AddInstanceAttributes, WebODE.InstanceAttribute, ia);
}
```

3.3.1.2.2) OntoMorph

Comme ODEDialect, OntoMorph [Chalupsky, 2000] propose un langage pour spécifier des traductions directes entre LRCs. Contrairement à ODEDialect, le langage proposé par OntoMorph permet de spécifier des traductions directement entre des CEs.

3.3.1.2.3) MOF QVT

Le standard QVT (Query View Transformation) [QVT, 2016] préconisé par l'OMG est un ensemble de langages. Ces langages peuvent être utilisés pour spécifier des traductions de LRCs. Dans ODM 1.1 (Ontology Definition Meta-model) [ODM, 2014], le langage QVT-operational est utilisé pour décrire quelques traductions d'EAs. Un exemple d'une telle traduction est donné ci-dessous. Cet exemple est une partie de la spécification d'une traduction depuis RDFS vers CL.

```
// traduction depuis une "relation" en RDF (RDFStatement) vers une "relation" en CL (AtomicSentence)
mapping RDFStatement::convertTripleDirect() : AtomicSentence
{ //self.RDFpredicate réfère à l'opérateur de RDFStatement
  //predicate réfère à l'opérateur de AtomicSentence
  predicate := self.RDFpredicate.map convertRessource(); //convertRessource() est la fonction de traduction
                                                    //sur RDFpredicate

  arguments := { //réfère aux arguments de AtomicSentence
    self.RDFsubject.map convertRessource(); // traduction du "sujet" de la propriété
    self.RDFobject.map convertRessource(); // traduction de "l'objet" de la propriété
  };
}
```

Tel ODEDialect, QVT est orienté opérationnalisation et est proche des langages de programmation.

3.3.1.2.4) SPARQL-Generate

Il existe des outils permettant d'extraire des connaissances implicitement représentées dans des langages de description de structure de données, par exemple, le langage R2RML [R2RML, 2012] ou le langage RML [Dimou et al., 2014] qui est plus générique. Tous ces outils ont été initialement conçu pour l'extraction de contenus (cf. contenu de description). SPARQL-Generate fait partie de ces outils. Cependant, SPARQL-Generate pourrait aussi être utilisé pour spécifier des traductions entre EAs depuis des LRCs vers RDF.

SPARQL-Generate étend SPARQL via les clauses présentées dans la liste ci-dessous. SPARQL-Generate permet de générer des graphes RDF depuis quelques structures de données. Pour ce graphe, des noeuds concepts peuvent être extraits depuis une structure de données particulière pour laquelle des fonctions de parcours sont prédéfinies ou ont été programmées par l'utilisateur. Les noeuds relations sont spécifiés via la clause GENERATE (voir ci-dessous) et ne peuvent pas être extraits depuis une structure de données.

- SOURCE : permet de référer à un document via une variable. Ci-dessous, la variable ?source réfère au document <http://country.io/capital.json>.
- ITERATOR : permet l'utilisation d'une fonction d'importer une liste de noeuds RDF depuis un document puis de parcourir cette liste. Ci-dessous, siter:JSONListKeys est la fonction d'import utilisée ; cette fonction prend en argument un document JSON et retourne un noeud RDF pour chaque clef de ce document.
- GENERATE : étend la clause "CONSTRUCT" pour permettre l'utilisation de sous-requêtes.

Un exemple de spécification via SPARQL-Generate depuis un document écrit en JSON est donné ci-dessous. La requête présentée dans cet exemple doit être écrite par un utilisateur. Elle permet de générer un graphe RDF dont les noeuds concepts peuvent être extraits depuis des phrases en JSON telles que : {"CN": "Beijing", "RU": "Moscow"; "FR": "Paris"}. Un résultat renvoyé par cette requête peut être "[] a Country ; capital "Paris" ; code "BV" ." Pour des phrases JSON structurées différemment, les clauses WHERE et ITERATOR doivent être modifiées.

```
BASE <http://example.com/>
PREFIX iter: <http://w3id.org/sparql-generate/iter/>
PREFIX fn: <http://w3id.org/sparql-generate/fn/>

GENERATE
{ [ ] a <Country> ;
  <code> ?key ;
  <capital> ?capital.
}
SOURCE <http://country.io/capital.json> AS ?source
ITERATOR iter:JSONListKeys(?source) AS ?key
WHERE
{ BIND(CONCAT('$.', ?key) AS ?query)
  BIND(fn:JSONPath(?source, ?query ) AS ?capital)
}
```

3.3.1.3) Comparaison avec l'approche préconisée dans cette thèse

Pour chaque couple de modèle de langage, cette approche nécessite une règle ou fonction de traduction spécifique. Ceci rend difficile le passage à l'échelle de solutions mettant cette approche en oeuvre. En effet, pour N le nombre de modèles de langage, il faut N*(N-1) processus spécifiques de traduction. La difficulté ici provient à la fois du nombre de langages et de modèles de langages, et également à la faible réutilisabilité des processus de traduction.

Il existe des centaines de LRCs différents. Par exemple :

- tous les sous-modèles d'OWL : OWL-Lite, OWL-DL, OWL-Full, OWL2-RL... et toutes leurs notations : XML, turtle, N3, FSS, etc. ;
- tous les sous-modèles de graphes conceptuels : "simple conceptual graph", "nested graphs" [Chein et al., 1998], "colored simple graphs" [Baget & Mugnier, 2002], "conceptual graph assemblies" [Croitoru & Compatangelo, 2006], ... et toutes leurs notations : CGLF, CGDF, CGIF, FCG, CoGXML, etc. ;
- tous les modèles et notations inspirés de KIF : CycL, CLIF, etc. ;
- etc.

Spécifier des traductions entre chaque paire de modèle ou de notation de LRC n'est pas réalisable en pratique compte tenu du grand nombre de traductions à spécifier.

3.3.2. Traduction exploitant un langage pivot

(cf. figure 3.5, “Centralized_TnO”)

3.3.2.1) Description

La spécification de traductions directes entre les modèles de LRCs est une approche qui ne passe pas l'échelle (cf. section “3.3.1.3 Comparaison avec l'approche préconisée dans cette thèse”). L'utilisation d'un langage pivot (ou d'échange) permet de réduire à $O(2n)$ le nombre de traductions à spécifier. Le langage pivot doit être suffisamment expressif pour assurer des traductions sans perte d'informations. L'approche “utilisation d'un LRC pivot pour des traductions” est généralisée par l'approche préconisée dans cette thèse (utilisation d'une ontologie de LRCs pour des traductions) où l'ontologie fait office de pivot.

Plusieurs langages ont été ou sont utilisés comme langage pivot. Par exemple, dans les années 1990, KIF (Knowledge Interchange Format) était un langage d'échange de-facto. KIF est un LRC basé sur la logique du premier ordre et dont la notation est de second ordre. KIF est un précurseur de CL et de IKL [Hayes, 2006], beaucoup de ceux qui ont travaillé sur KIF ont également travaillé sur CL et sur IKL [Hayes, 2006].

CL est un modèle pour des LRCs basés sur la logique du premier ordre. CL est né d'efforts de rationalisation et de mise à jour de la conception de KIF. CL et KIF ont plusieurs différences. Entre autres, KIF est un langage avec un modèle abstrait et une notation basés sur Lisp alors que CL est uniquement un modèle abstrait et n'est pas basé sur Lisp. Plusieurs notations avec le modèle CL ont été standardisées. Elles sont décrites ci-dessous.

- CLIF [Common Logic Interchange Format] est une notation ressemblant à celle de KIF ; CLIF peut être considéré comme une forme simplifiée de KIF. Par exemple, KIF a un type de CE pour définir des fonctions, CLIF n'en a pas.
- CGIF [Conceptual Graph Interchange Format] est une notation inspirée de celles utilisées pour les graphes conceptuels.
- XCL [eXtended Common Logic markup language] est une notation créée via XML.

Dans le standard ISO 24707, qui définit CL, ces notations sont appelées dialectes. Parmi ces dialectes, seul XCL est recommandé comme langage d'échange. En effet, des LRCs basés sur des métalangages de description de structure sont souvent utilisés comme langages d'échange. Ainsi, l'import de connaissances est facilité pour certains SGBCs. Cette approche pour l'import est décrite dans la section “3.1.4. Import exploitant des métalangages de description de structures de données”.

Les notations des dialectes de CL ne sont pas spécifiées dans une ontologie.

IKL est un LRC dont le modèle est une extension de CL. Comme KIF, IKL permet la représentation de notions importantes pour le partage de connaissances. Certaines de ces notions sont généralement représentables uniquement dans les LRCs basés sur des logiques d'ordre supérieur. C'est le cas, par exemple, pour des quantificateurs numériques ou pour certains types de méta-phrases comme des méta-phrases contextualisantes. Un exemple de méta-phrase contextualisante dans IKL est donné ci-dessous. Une traduction de cet exemple en français pourrait être : “la proposition : ‘un humain démocrate est président des états unis’, est vraie en 1995.” L'opérateur “that” permet de réifier la phrase en argument.

```
IKL :  
(ist TemporalContextYear1995  
  (that (exists ((x isHuman)) (and (Democrat x)(PresidentOfUSA x))))  
)
```

Une autre approche pivot semblable celle qui est présentée dans cette section consiste à utiliser d'un protocole commun ou une API commune entre plusieurs outils permettant de manipuler des RCs. OKBC [OKBC, 1998] est un exemple d'une telle approche.

3.3.2.2) Exemple

Ontolingua [Gruber, 1993][Farquhar et al., 1997] est un serveur d'ontologies et un “système de traduction” exploitant un langage pivot. Ce langage est KIF.

Une traduction depuis un langage source vers un langage cible se fait en trois temps :

1. export depuis dans le langage source vers KIF ;
2. import depuis KIF vers un langage interne d'Ontolingua ;
3. export depuis ce langage interne vers le langage cible.

Le langage interne est un sous langage de KIF. Son modèle abstrait est défini dans une ontologie : la *Frame Ontologie*. Cette ontologie définit quelques types d'AEs : 18 classes, 30 relations, 12 fonctions. Par exemple, elle *définit* des types pour :

- les relations transitives (Transitive-Relation et Weak-Transitive-Relation) ;
- les restrictions de cardinalités (Maximum-Slot-Cardinality, Maximum-Value-Cardinality, Minimum-Slot-Cardinality et Minimum-Value-Cardinality).

Le choix des types d'AEs définis dans la *Frame Ontologie* a été fait en fonction des types fréquemment utilisés dans les langages basés sur les Frames. Ces types d'AEs sont similaires à ceux qui sont *déclarés* dans les schémas XML pour les divers modèles OWL (OWL Lite, OWL Full, OWL 2 EL, etc.).

Une traduction depuis KIF vers un langage cible est réalisée uniquement pour ces quelques types d'AE. Les autres AEs ne sont pas traduits. Ontolingua les renvoie en KIF et en informe l'utilisateur avec un message.

3.3.2.2.1) import vers le langage interne d'Ontolingua

L'import depuis KIF vers le langage d'ontolingua est une *analyse sémantique directe* basée sur la *Frame Ontology*.

3.3.2.2.2) traduction vers le langage cible

Contrairement à KRLO, la *Frame Ontology* n'inclut pas de spécifications pour des modèles abstraits autres que celui du langage d'ontolingua. Elle n'inclut pas non plus de modèles concrets particuliers ni pour le langage d'Ontolingua, ni pour d'autres LRCs. Par exemple, elle n'inclut de spécifications ni pour le modèle abstrait, ni pour le modèle concret du langage PowerLoom. La *Frame Ontology* peut difficilement être exploitée par des processus d'import, d'export ou de traduction. Une traduction depuis le langage d'ontolingua doit être spécifiée pour chaque langage cible.

3.3.2.3) Comparaison avec KRLO

Un processus de traduction exploitant une ontologie de langages généralise un processus de traduction exploitant un langage pivot. Cette ontologie peut être appelée une "ontologie pivot" par analogie. Cependant, dans une approche exploitant une ontologie de langage telle que KRLO, l'utilisation de relations de généralisation entre les AEs, les modèles ou les notations permet de maximiser la réutilisation de ces AEs, modèles ou notations.

3.3.3. Traduction via une approche "famille de langages"

(cf. figure 3.5, "Centralized_TnO")

3.3.3.1) Description

La notion de famille modulaire de langages a été introduite par [Euzénat, 2001f]. C'est un ensemble $S1$ de langages dont les modèles abstraits partagent certains éléments. Il existe au moins une relation d'ordre partiel entre les éléments de $S1$: pour tout modèle abstrait $M1$ d'un langage $L1$ et $M2$ d'un langage $L2$, il en existe toujours au moins un, $M3=M1 \cup M2$ d'un langage $L3$, tel que tout EA de $M1$ et $M2$ soit également un EA de $M3$. Dans des familles particulières (i.e. des instances de famille modulaire de langage), d'autres relations d'ordre partiel entre les langages peuvent être représentées. Les types de relation utilisables sont prédéterminés pendant la conception de chaque famille.

L'utilisation d'une famille de langages pour des traductions de LRCs est proposée dans [Euzénat, 2001b]. Un système exploitant cette approche est composé d'une famille de langage et de spécifications de traductions entre certains des langages de la famille. En enchainant ces traductions, il est possible de traduire entre tous les langages de la famille. Cependant, lorsque N est le nombre de langages dans la famille, le nombre de processus de traduction à spécifier par le concepteur de la famille est proche de N^2 (voir section 3.3.3.3 ci-après pour les détails du calcul). Ceci rend difficile le passage à l'échelle de solutions mettant cette approche en oeuvre.

Chaque relation dans la famille est associée à une unique fonction ou règle de traduction. Cette relation associée décrit une propriété de la traduction, comme par exemple, la conservation de l'interprétation entre la source et la destination.

Pour traduire depuis un langage vers un autre dans une famille, les deux étapes sont nécessaires.

1. Recherche d'un chemin depuis le langage source vers le langage cible.
2. Exécution des processus de traductions associées aux relations composant ce chemin.

Ainsi, un processus de traduction peut être paramétré (voir le point 1 ci-dessous) ou analysé (voir le point 2 ci-dessous) via quelques propriétés de traduction.

1. Un filtrage sur les types de relations peut être utilisé pendant la recherche de chemin. Ainsi, un utilisateur peut sélectionner des propriétés de traduction pour un processus. Par exemple, il est possible de paramétrer un processus de sorte que la phrase traduite conserve la consistance des RCs, i.e., si les connaissances représentées par les RCs dans la phrase en entrée sont logiquement cohérentes, alors les RCs dans la phrase en sortie seront également logiquement cohérentes.
2. Il est possible d'analyser un chemin donné pour déterminer les propriétés d'une traduction.

[Euzénat & Stuckenschmidt, 2003] propose six types de relation (relation-critère) représentant chacune une propriété de traduction d'un langage à un autre. Ces relations sont les suivantes.

- Une relation représentant une propriété de *conservation des éléments du modèle* : les éléments du modèle du langage source sont également dans le modèle du langage destination.
- Une relation représentant une propriété de *conservation de l'interprétation* : toute interprétation des RCs source est également une interprétation des RCs en résultat.
- Une relation représentant une propriété de *conservation de l'expressivité* : toute RC source peut être exprimée dans le modèle du langage cible.
- Une relation représentant une propriété d'*épimorphisme des modèles* : la transformation se fait sans perte d'information ;
- Une relation représentant une propriété de *conservation des déductions* : les déductions possibles dans les RCs source sont possibles dans les RCs en résultat.
- Une relation représentant une propriétés de *conservation de la cohérence* : si les RCs source sont cohérentes, alors les RCs en résultat sont cohérentes.

Une relation représentant une propriété de conservation de l'interprétation depuis un modèle Ls vers un modèle Ld est représentée si et seulement si une traduction préservant l'interprétation est possible depuis Ls vers Ld.

3.3.3.2) Exemple

Une implémentation de l'approche décrite dans cette section est également proposée dans [Euzénat & Stuckenschmidt, 2003]. Cette implémentation exploite DLML [Euzénat, 2001e], une ontologie d'EAs pour des LRCs basés sur des logiques de descriptions. DLML est représentée directement en XML. Quelques fonctions de traduction sont spécifiées via des scripts XSLT. Quelques exemples de descriptions de types d'EAs dans DLML sont présentés ci-dessous. Ces EAs sont des EAs structurels.

```
<!-- CDESC : Concept node Description ; réfère aux 4 éléments suivants -->
<!ELEMENT dl:AND ((%dl:CDESC;)+)>
<!ELEMENT dl:CPRIM (dl:CATOM,%dl:CDESC;)> <!-- CPRIM : Concept Primitif -->
<!ELEMENT dl:CATOM (#PCDATA)> <!-- CATOM : Concept Atomique, représenté par un string -->
<!ELEMENT dl:RATOM (#PCDATA)> <!-- RATOM : Relation Atomique -->
<!ELEMENT dl:ALL (%dl:RDESC;,%dl:CDESC;)>
```

Dans la première ligne de cet exemple des relations *argument* sont implicitement spécifiées entre l'EA dl:AND et les EAs de type dl:CDESC. Les structures extraites par l'analyseur XML sont exploitées par des fonctions XSLT, donc de manière structurelle, mais implicitement elles représentent un ASG avec un seul type de relation *argument*.

Dans DLML, il n'y a pas de modèle concret ; un langage n'est donc qu'un modèle abstrait et a pour partie des types d'EAs. Par exemple, le modèle d'un langage basé sur la logique de description ALN est décrit de la façon suivante :

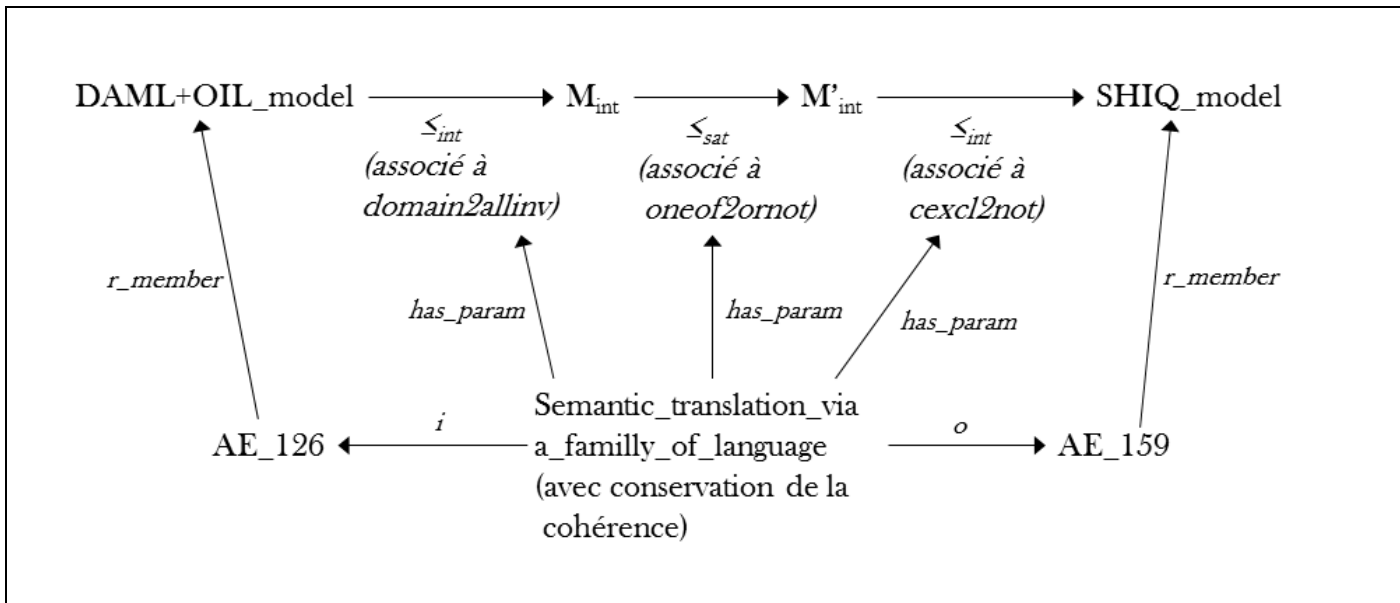
```
<dlml:logic name="aln">
  <dlml:atoms/>
  <dlml:cop name="anything"/>
  <dlml:cop name="nothing"/>
  <dlml:cop name="and"/>
  <dlml:cop name="anot"/> <!-- Négation restreinte aux concepts atomiques -->
  <dlml:cop name="all"/>
  <dlml:cop name="atleast"/>
  <dlml:cop name="atmost"/>
  <dlml:cop name="some"/>
  <dlml:cint name="cdef"/>
  <dlml:cint name="cprim"/>
</dlml:logic>
```

Ci-dessous, la phrase “All CSmaster students (implicitly universally quantified) are bachelor students whose advisor is computer scientist (implicitly existentially quantified)” est représentée via DLML.

```
<CPRIM>
  <CATOM>CSMasterStudent</CATOM> <!-- implicitly universally quantified -->
  <AND>
    <CATOM>Bachelor</CATOM>
    <CATOM>Student</CATOM>
  <ALL>
    <RATOM>advisor</RATOM>
    <CATOM>ComputerScientist</CATOM> <!-- implicitly existentially quantified -->
  </ALL>
</AND>
</CPRIM>
```

La figure ci-dessous illustre les différentes étapes dans la traduction d'un EA de DAML+OIL_model vers un EA de SHIQ_model. Dans cet exemple, domain2allinv, oneof2ornot, cexcl2not sont des scripts XSLT. \leq_{synt} (préservation syntaxique), \leq_{int} (préservation d'interprétation), \leq_{sat} (préservation de conséquence) sont des "relations-critères". M_{int} et M'_{int} sont des langages qui servent d'intermédiaires.

Figure 3.6. Étapes d'une tâche de traduction via une famille de langages avec les relations proposées dans [Euzénat & Stuckenschmidt, 2003].



3.3.3.3) Comparaison avec KRLO

L'approche proposée dans cette thèse peut être vue comme une extension de l'approche décrite dans cette section. En effet, avec un peu de travail, des propriétés de traduction pourraient être représentées dans KRLO et la plupart des traductions exploitées dans [Euzénat & Stuckenschmidt, 2003] pourraient être spécifiées. De plus, contrairement à KRLO, une famille de langages ne permet pas de représenter les modèles concrets des LRCs. Une extension de KRLO incluant des critères de sélection est prévue à long terme.

L'approche décrite dans cette section nécessite soit l'utilisation de stratégies générales complexes de ré-écriture de graphes [graph/term rewriting], soit une énumération exhaustive de tous les modèles de LRCs "intermédiaires" (cf. M_{int} et M'_{int} dans le schéma ci-dessus). En effet, si les relations entre les langages ne sont pas prédéfinies alors ces relations doivent être générées. Ainsi, une recherche de chemin est également une génération de graphe. Par exemple, pour chercher un chemin depuis DAML+OIL_model vers SHIQ_model, chaque fonction de traduction depuis DAML+OIL_model doit être évaluée, puis pour chaque modèle intermédiaire ainsi généré chaque fonction de traduction depuis ce modèle doit de nouveau être évaluée et ainsi de suite jusqu'à ce qu'un modèle SHIQ soit généré. D'un autre côté, si les relations entre les langages sont prédéfinies, l'identification de tous les modèles intermédiaires utilisables et la représentation de toutes les relations-critères entre chaque modèle nécessite beaucoup de travail. Plus précisément, *pour chaque langage intermédiaire* il faut représenter ces relations vers chaque langage intermédiaire de la couche supérieure. Si i est le nombre de langages intermédiaires, c le nombre de "couches" dans la famille, i/c le nombre moyen de langages intermédiaires par "couche" et r le nombre de types de relation-critère alors il y a $O((r \times i)/c)$ relations à représenter *pour chaque langage intermédiaire* et donc $O((r \times i^2)/c)$ relations à représenter dans la famille. C'est comparable au nombre de traductions à spécifier via l'approche décrite dans la section "3.3.1. Spécifications de traductions directes", c'est-à-dire $O(N^2)$ où N est le nombre de langages. L'approche décrite dans cette section "3.3.3. ne pass' donc pas l'échelle".

3.3.4. Spécifications de traductions exploitant une ontologie de langage existante

(cf. figure 3.5, “TO_wo_NO”)

Trois organismes de standardisation ont proposé des ontologies de langage pour les modèles des LRCs qu'ils préconisent.

- Le W3C a publié des modèles XML pour, entre autres, RDF, OWL et RIF. Il a également publié des règles de traductions entre certains d'entre eux.
- L'ANSI offre un modèle UML et un modèle XML pour CL.
- L'OMG a également publié un modèle UML et un modèle XML pour ODM.

Toutes ces ontologies *déclarent* des EAs, elles ne définissent pas leurs sémantiques. Elles ne spécifient pas de ECs.

L'ontologie proposée par l'OMG, inclus d'autres ontologies de langages. Les autres ontologies proposées n'en incluent pas. Le modèle proposé par l'ANSI est inclus dans celui proposé par l'OMG.

3.3.4.1. Règles de traductions proposées par le W3C

[[OWL2-RL to RIF-BLD, 2013](#)] et l'annexe 8 de [[OWL2-RL in RIF-BLD, 2013](#)] donnent des règles pour traduire des ECs de la syntaxe fonctionnelle de OWL2-RL (FSS) vers des ECs de la notation RIF-PS pour le modèle RIF-BLD. Dans [[OWL2 FSS from/to RDF triples, 2012](#)] le W3C propose des règles pour traduire des ECs de FSS vers des ECs des notations triplets pour le modèle RDF.

Le W3C ne propose pas de traductions entre des EAs puisque :

- RIF-BLD n'est pas représenté dans une ontologie, et
- les ECs de FSS représentent directement les EAs de la *spécification structurelle* de OWL2 ; il y a ici une forme d'homocité.

3.3.4.2. Ontologie de langages de l'OMG

ODM 1.1 déclare et donne des définitions structurelles pour les EAs de quatre modèles : RDF, OWL, CL et Topic Maps [Topic Maps, 2003]. Chaque EA est spécifique à un modèle. Par exemple, ODM ne définit pas et ne déclare pas d'EA représentant une relation binaire dans tous les modèles. ODM 1.1 a quelques *relations sémantiques* – telles que des relations de généralisation ou d'équivalence – entre des EAs de modèles différents. Beaucoup de ces EAs ne sont pas reliés par de telles relations. Ces EAs de différents modèles et non reliés ne sont pas automatiquement sémantiquement comparables : les ontologies de modèles sont globalement hétérogènes.

Un modèle dans ODM est implicitement homogène. En effet, les relations entre les EAs sont des associations UML. Les associations UML sont composées d'un nom d'association et de deux terminaisons (source et destination de la relation) qui peuvent être nommées. Comme les associations UML sont construites sur un même plan, les relations entre les EAs dans ODM sont également construites sur un même plan. Le modèle est donc homogène. Ce plan est cependant implicite :

- le nom d'association désigne le type d'une relation entre des EAs,
- ses terminaisons désignent les parties de cette relation,
- un nom de terminaison désigne un opérateur de relation, la notion "d'opérateur d'EA" est décrite dans la section “4. KRLO, une ontologie de LRCs”.

Les EAs dans ODM sont exploitables de façon uniforme bien que cette exploitation soit difficile car seulement implicitement homogène. Un autre élément renforce cette difficulté : les types de relation entre les EAs sont spécifiques. Par exemple, le type de relation `unionClassForUnion` est spécifique à l'EA `UnionClass` et le type de relation `intersectionClassForIntersection` est spécifique à l'EA `IntersectionClass`. Des généralisations de ces types de relation ne sont ni définis, ni déclarés. Par exemple, `unionClassForUnion` et `intersectionClassForIntersection` sont des spécialisations d'un type de relation opérateur+destination ; ce type n'est ni déclaré, ni défini. Ainsi, les relations entre les EAs doivent être traitées de façon distincte. Par exemple, pour exporter `UnionClass` et `IntersectionClass`, il faut écrire une règle (ou une partie de règle) prenant en compte la relation `unionClassForUnion` et une autre prenant en compte la relation `intersectionClassForIntersection`. Il est possible d'utiliser des astuces pour faciliter l'exploitation uniforme de ces EAs. Par exemple, si `UnionClass` et `IntersectionClass` ont une et une seule propriété UML, alors il est possible de les exploiter de façon uniforme : il suffit de prendre en compte cette unique propriété.

Les ontologies de modèles dans ODM sont hétérogènes globalement et implicitement homogènes en interne. Elles sont difficilement exploitables de façon générique même pour concevoir des règles d'export.

KRLO :

- représente les relations de généralisation, d'implication ou d'équivalence entre les EAs des différents modèles de LRCs ;
- représente uniformément leurs structures via quelques types de relations – ceci défini partiellement la sémantique des EAs ;
- représente uniformément les ECs via les mêmes types de relations ;
- relie les EAs à des ECs, c'est à dire, à leurs présentations dans certaines notations. Ces liens entre les EAs et des ECs impliquent que les modèles de LRCs sont reliés aux notations de LRCs.

Ainsi, pour chaque modèle, les différents types d'éléments de langage sont gérés de la même façon. Seules les quelques primitives utilisées pour décrire leurs structures doivent être prises en compte. Ces primitives peuvent être combinées pour obtenir les mêmes fonctionnalités que celles qui seraient fournies par un grand nombre de types de relations.

3.3.4.3. HETS avec LATIN et DOL

HETS est un outil qui permet l'utilisation de plusieurs LRCs dans un même fichier afin de décrire une ontologie. HETS fonctionne de la façon suivante.

1. Chaque LRC est décrit en Haskell. Haskell est un langage de programmation fonctionnel fondé sur le lambda-calcul et la logique combinatoire. Dans HETS, la description d'un LRC spécifie un certain nombre de méthodes avec des noms prédéfinis (interface en orienté objet).
2. Pour chaque LRC ainsi décrit, un analyseur syntaxique est implémenté. Chacun d'eux a en sortie un AST avec à l'intérieur des objets Haskell prédéfinis.
3. De ces ASTs, un analyseur sémantique extrait un ASG appelé *graphe de développement*. Ce graphe est un graphe de relations entre des logiques, typiquement, des relations *part* ou représentant des propriétés de traduction. Ces propriétés sont typiquement des propriétés de conservation des informations. Certaines de ces propriétés sont présentées dans la section “3.3.3. Traduction via une "famille de langages"”. Certaines relations – appelées *theorem links* – permettent de décrire des objectifs de preuves. Les démonstrations pour ces objectifs peuvent être trouvées par des moteurs d'inférences fournis par HETS.
4. Des spécifications de traductions entre des LRCs peuvent être implémentées en Haskell. Ceci rejoint l'approche décrite dans la section “3.3.1. Spécifications de traductions directes”.

Le code de HETS peut être trouvé à <https://github.com/spechub/Hets>.

HETS prend en entrée des phrases écrites dans un métalangage, c'est à dire un langage dont certains éléments sont des langages. HetCASL est l'un des métalangages acceptés par HETS. L'exemple ci-après est adapté du guide d'utilisation de HETS pour CL. Il illustre des spécifications pour importer et traduire des RCs dans différents LRCs. Les trois prochains paragraphes fournissent des explications sur le code donné dans l'exemple qui suit.

Le LRC utilisé pour décrire une ontologie doit être spécifié par le mot clef “logic”, le parser approprié peut ainsi être sélectionné.

Un processus de traduction “Trans” peut être appliquée à une ontologie “Ont” via la syntaxe : { Ont with logic Trans }. Ainsi, dans l'exemple ci-après, le processus de traduction OWL22CommonLogic est utilisé pour traduire l'ontologie TimeOWL. Une traduction ne semble pas pouvoir être utilisée en dehors d'une “vue”.

En HetCASL, une “vue” permet de spécifier une relation représentant une propriété de conservation d'informations pour une traduction entre deux ontologies. Une telle relation (*TimeOWLtoCL*) est spécifiée dans l'exemple ci-dessous entre l'ontologie “TimeCL” et le résultat d'une traduction “OWL22CommonLogic” appliquée sur l'ontologie “TimeOWL”. La traduction “OWL22CommonLogic” est implémentée en Haskell.

```
view TimeOWLtoCL : { TimeOWL with logic OWL22CommonLogic } to TimeCL

logic OWL
spec TimeOWL =
  ObjectProperty: before    %% pour ces 4 lignes, la syntaxe Manchester est utilisée
  Domain: TemporalEntity    %% source
  Range: TemporalEntity     %% destination
  Characteristics: Transitive %% la relation before est transitive
end

logic CommonLogic
spec TimeCL =
  . (forall (t1 t2)          %% définition de la source et de la destination
    (if (before t1 t2)      %% de la relation before
      (and (TemporalEntity t1)
            (TemporalEntity t2))))
  . (forall (t1 t2 t3)      %% transitivité
    (if (and (before t1 t2)
              (before t2 t3))
        (before t1 t3)))
end
```

Différents types de traductions et certaines de leurs propriétés ont été étudiés dans LATIN (Logic ATlas and INtegrator) [Codescu et al., 2011] et dans [Kutz & Mossakowski, 2011]. L'ontologie LoLa (Logic and Language) [Lange, Mossakowski, Kutz, 2012] exprime une petite partie de ces résultats en OWL. Ainsi, dans ces travaux, certains LRCs et certaines logiques et traductions sont déclarés et hiérarchisés. Les signatures de ces traductions sont également représentées. Dans LATIN et KRLO, les traductions ne sont pas seulement déclarées avec leurs entrées et sorties mais sont également définies.

Des fonctions d'exports vers des notations sont également déclarées, hiérarchisées d'une façon similaire aux traductions. Un exemple est donné ci-dessous. Ces spécifications sont extrêmement partielles comparées à celles qui sont définies dans KRLO. De plus, contrairement à KRLO, LoLa ne spécifie pas d'ECs.

Ci-dessous, deux exemples de spécification dans LoLa pour une traduction et une notation sont présentés. Nous avons réécrit ces exemples avec la notation Manchester pour plus de lisibilité. Le premier exemple est une traduction de type "mapping" entre deux logiques. Une telle traduction est simplement définie comme un mapping depuis une logique vers une autre. Le second exemple est une notation (appelée "serialization"). Une notation est définie comme une "entité linguistique" (LinguisticEntity) et comme étant la sortie d'un export depuis un LRC.

```

Class: LogicMapping
  Annotations:
    rdfs:label "logic mapping",
    rdfs:comment "a mapping (translation or projection) between two logics",
    propagatesToAdjoint "true"^^xsd:boolean,
    skos:definition "mapping between logics"
  EquivalentTo:
    mapsFrom exactly 1 Logic and
    mapsTo exactly 1 Logic
  SubClassOf:
    Mapping
  SubClassOf:
    mapsFrom exactly 1 Logic and
    mapsTo exactly 1 Logic

Class: Serialization
  Annotations:
    rdfs:label "OMS serialization",
    skos:example "Common Logic uses the term ``dialect``;
      the following are standard Common Logic dialects:
      Common Logic Interchange Format (CLIF), Conceptual Graph Interchange Format (GCIF),
      eXtended Common Logic Markup Language (XCL).",
    skos:example "OWL uses the term 'serialization' the following are standard OWL serializations:
      OWL functional-style syntax, OWL/XML, OWL Manchester syntax,
      plus any standard serialization of RDF (e.g., RDF/XML, Turtle, ...).
      However, RDF/XML is the only one tools are required to implement.",
    skos:note "Serializations serve as standard formats for exchanging OMS between tools.",
    rdfs:comment "a serialization of a language",
    skos:definition "specific syntactic encoding of a given OMS language",
    rdfs:comment "synonymous to 'concrete syntax'"
  SubClassOf:
    LinguisticEntity,
    serializes some OMSLanguage

```

LATIN est un projet qui vise à développer des méthodes, des techniques et des outils afin d'améliorer l'interopérabilité des démonstrateurs de théorèmes, solveurs de contraintes et autres outils similaires. [Codescu et al., 2010] précise que ceci passe par *l'intégration complète des frameworks basés sur la théorie des modèles et sur la théorie de la preuve i) en préservant leurs avantages respectifs, et ii) en créant des formalisations modulaires pour les logiques souvent utilisées avec les traductions logiques qui les relient*. Ce projet a débouché sur la création du "LATIN Atlas of logics", que nous appellerons simplement LATIN dans la suite de cette thèse. Plusieurs logiques, théories de types et LRCs y sont décrits, par exemple : Pure logique (une logique utilisée par le démonstrateur de théorèmes Isabelle), λ -cube (une théorie de types), CASL (un LRC qui suit une logique du premier ordre). Le langage utilisé pour ces descriptions est Twelf [Frank Pfenning & Carsten Schürmann, 1999]. Twelf est basé sur LF [Harper et al., 1987], un framework logique dont le méta-langage est le $\lambda\Pi$ -calcul. Un framework logique fournit un moyen de définir une logique comme une signature dans une théorie de type d'ordre supérieur de sorte que la prouvabilité d'une formule dans cette logique se réduise à un problème d'instanciation de type [type inhabitation problem] dans la théorie de type du framework. Un système d'import dans Twelf permet de modulariser les descriptions Twelf. Ainsi, il est possible, par exemple, de combiner plusieurs théories de types avec différentes logiques, ou d'assembler une nouvelle logique à partir d'EAs. Dans LATIN,

une logique ou un LRC a pour partie un modèle théorique, un modèle de preuve et un modèle (de données) abstrait, c'est à dire un ensemble d'EAs. Seules les descriptions des modèles (de données) abstrait entrent dans le cadre de cette thèse. L'exemple ci-dessous présente une description pour le modèle abstrait de la logique du premier ordre classique sans égalité dans LATIN. Ces descriptions peuvent être trouvées en ligne à l'adresse : <https://svn.kwarc.info/repos/twelf/logics/first-order/syntax/>, accessible depuis le site de LATIN : <https://trac.omdoc.org/LATIN/>. Ce modèle est un assemblage des 4 *modules* décrits ci-après.

- BaseFOL, qui contient les composants logiques primitifs de la logique du premier ordre :

```
%sig BaseFOL = {
  %include Base %open o ded.    %% o représente les propositions, ded les preuves
  i : type.    %% i représente les individus
}.

```

- PL, les descriptions des composants de la logique des propositions ;

```
Forall :
%sig Forall = { %% définition de la signature de forall
  %include BaseFOL %open.    %% sans %open, il faudrait écrire BaseFOL.i et BaseFOL.o
  forall : (i -> o) -> o.    %% fonction qui associe un individu à une proposition
}.

```

- Exists :

```
%sig Exists = {
  %include BaseFOL %open.
  exists : (i -> o) -> o.
}.

```

Ces descriptions sont ensuite réutilisées pour d'autres logiques, ou théories de types, ou LRCs. L'exemple ci-dessous illustre une réutilisation de la description précédente (Forall) dans une description de quantificateurs proches (iforall et sforall) pour CL. Cet exemple est incomplet, la version complète peut être trouvée à : https://svn.kwarc.info/repos/twelf/logics/common_logic/syntax.elf. Les individus dans la logique du premier ordre sont traduits dans CL soit comme des individus (notés *i* dans l'extrait ci-dessous), soit comme des marqueurs de séquence (notés *sq* dans l'extrait ci-dessous) ; iforall est une spécialisation de Forall avec des individus de CL ; sforall est une spécialisation de Forall avec des marqueurs de séquence.

```
%sig BaseCL = {
  %include Base.    %% import des composants de base de la logique des propositions
  i : type.
  sq : type.
}
%% %%view permet de spécifier une traduction
%view UnivToInd : BaseFOL -> BaseCL = {i := i. }.    %% BaseFOL.i devient BaseCL.i
%view UnivToSeq : BaseFOL -> BaseCL = {i := sq.}.    %% BaseFol.i devient BaseCL.sq

%sig CL = {
  %include BaseCL.
  %include PL.

  %% %struct permet de spécifier une relation de spécialisation
  %struct iforall : Forall = {%include UnivToInd.}    %open forall.
  %% %as désigne un alias
  %struct sforall : Forall = {%include UnivToSeq.}    %open forall %as foralls.
}

```

Toutes ces descriptions sont orientées opérationnalisation et forment une ontologie de modèles abstraits. Les modèles concrets (= syntaxe concrète) ne sont pas représentés. Contrairement à KRLO, la majeure partie des travaux sur LATIN se concentre sur les modèles théoriques et sur les modèles de preuve des langages et des logiques.

LATIN a été spécifié dans HETS, en HetCASL, après que ce langage ait été étendu afin que de nouvelles logiques puissent y être spécifiées. Ainsi, de nouveaux LRCs et de nouvelles logiques peuvent être maintenant ajoutés à HETS de façon déclarative [Codescu et al., 2010]. Après recompilation (de HETS), ces logiques sont également ajoutées dans le graphe de développement. Par exemple, la logique du premier ordre classique avec égalité peut être ajoutée à HETS via le code HetCASL ci-après.

```

%% imports de modules
from logics/first-order/syntax/fol get FOL_truth
%% FOL_truth contient les descriptions des EAs de la logique du premier ordre.
from logics/first-order/model_theory/fol get FOL_mod
%% FOL_mod est une théorie de modèles pour la logique du premier ordre.
from logics/meta/sttifol get STTIFOLEQ
%% STTIFOLEQ (Simple Type Theory + Intuitionistic First Order Logic with Equality) est un fragment
%% de la théorie Zermelo-Fraenkel.
from logics/first-order/proof_theory/fol get FOL_pf
%% FOL_mod est une théorie de preuve pour la logique du premier ordre.

newlogic FOL =
  meta LF                %% Le framework logique utilisé est LF.
  syntax FOL_truth      %% spécification de la syntaxe abstraite
  models FOL_mod
  foundation STTIFOLEQ
  proofs FOL_pf
end

```

DOL (Distributed Ontology, Modeling, and Specification Language) [DOL, 2016] est un langage proposé par l'OMG dont le modèle abstrait est décrit dans l'ontologie LoLa. Il permet donc d'inclure différents langages prédéfinis dont HetCASL, CL, F-Logic et OBO. Ainsi via HETS et LATIN ce langage permet :

- d'exploiter des traductions entre des logiques ;
- d'exploiter des exports vers des notations ;
- de spécifier des traductions de contenu ;
- d'utiliser plusieurs LRCs dans un même document pour décrire ou étendre une ontologie.

3.4. Bilan

Rappel des questions de recherche

1. Comment spécifier une fonction d'import et ses entrées pour que cette fonction soit générique (c'est à dire, utilisable pour n LRCs définis de manière adéquate) ?
2. Comment spécifier une fonction d'export et ses entrées (définitions de types ou, alternativement à ces définitions, des règles ou des fonctions) pour que cette fonction soit générique ?
3. Comment spécifier ces fonctions et/ou leurs entrées de sorte qu'elles soient suffisantes pour des traductions entre LRCs ? Ceci implique aussi de déterminer si une sous-fonction (générique) de traduction entre des éléments de modèles abstraits de LRCs est utile ou si une définition précise des entrées suffit.

Désavantages des réponses apportées par les approches classiques

Aucune des approches présentées dans ce chapitre 3 ne répond aux questions de recherche de cette thèse. En effet, pour les entrées des fonctions d'import, d'export ou de traduction, aucune de ces approches ne propose l'exploitation de spécifications écrites i) via un LRC conçu pour la modélisation, et ii) pour un modèle conceptuel de modèles abstraits *et* de notations. Ainsi, les entrées des fonctions d'import, d'export ou de traduction utilisées dans ces approches ne sont pas spécifiées de façon assez précise pour que ces fonctions soient génériques. La liste ci-dessous reprend les principaux désavantages des approches classiques. La section “[2.2.2](#). Création directe d'un modèle exécutable via un LRC exécutable” fournit plus d'explications sur ces désavantages.

1. **EAs spécifiés via un langage de description de structure de données.**
Lorsque les entrées sont spécifiées via un langage de description de structure de données (comme illustré dans la section “[3.1.1](#). Import via un générateur d'analyseurs lexico-syntaxiques”), la sémantique de ces spécifications n'est pas explicite et du code ad hoc (donc non générique) est nécessaire pour les interpréter.
2. **EAs et ECs spécifiés via un LRC exécutable.**
Lorsque les entrées sont spécifiées via un LRC exécutable (comme illustré dans la section “[3.1.2](#). Import via un générateur d'environnement de programmation interactif”), elles peuvent difficilement être représentées de façon précise et donc réutilisées. Par exemple, elles peuvent difficilement être spécialisées ou paramétrées afin de spécifier de nouveaux LRCs.
3. **EAs spécifiés via une ontologie de modèles mais ECs non spécifiés via une ontologie de notations.**
Lorsque les entrées sont spécifiées uniquement via une ontologie de modèles de LRCs (comme illustré dans “[3.1.5](#). Import exploitant une ontologie de modèles abstraits”), les notations ne peuvent être spécifiées que via des relations structurelles. Ces spécifications de notations sont peu réutilisables et leurs sémantiques sont peu explicites.

Pour éviter à l'utilisateur la charge d'écrire une spécification de traduction pour chaque paire de langage, les approches présentées dans les sections [3.3.2](#) et [3.3.3](#) proposent d'organiser les modèles de LRCs via des relations d'ordre partiel (typiquement, par ordre d'expressivité croissante). Ainsi, les fonctions de traduction sont implicitement organisées via des relations part. Cependant, beaucoup de spécifications doivent tout de même être écrites.

Le tableau ci-après permet de comparer les différentes approches pour l'import selon les quatre critères présentés ci-dessous. Comme le montre ce tableau, seule l'approche proposée dans cette thèse – c'est à dire, l'exploitation d'une ontologie unique et homogène de modèles abstraits et de notations de LRCs par les fonctions d'import, d'export et de transformations – permet de satisfaire ces critères.

- Critère 1 : la description d'une fonction d'import est réutilisable pour plusieurs LRCs. La satisfaction de ce critère est nécessaire pour répondre aux questions de recherche.
- Critère 2 : les utilisateurs finaux peuvent paramétrer la fonction d'import. Ce critère est nécessaire pour la réalisation de l'objectif cette thèse (cf. section [1.3.3](#). “Objectif”).
- Critère 3 : la description d'une fonction d'import est déclarative et logique. Les avantages de telles descriptions par rapport à des descriptions procédurales ou non logiques sont décrites dans la section [2.2](#). Pour ces avantages, la satisfaction de ce critère est souhaitable pour répondre aux questions de recherche.
- Critère 4 : la description d'une fonction d'import est lisible. La satisfaction de ce critère est souhaitable.

	Description d'une fonction d'import pour une extension d'un langage existant via un évaluateur méta-circulaire	Description d'une fonction d'import conçue pour être directement interprétable par un générateur d'analyseur lexico-syntaxique (et donc pas sémantique)	Description d'une fonction d'import pour un langage décrit un métalangage de description de structures de données	Description d'une fonction d'import pour tout langage décrit via une ontologie de LRCs telle que KRLO
Description réutilisable pour importer plusieurs LRCs	non	peu	peu	oui
Description paramétrable pour importer plusieurs LRCs	non	peu	non	oui
Description déclarative et logique	non	non	non	oui
Description lisible	peu	peu	peu	oui

Des tableaux comparatifs pour les mêmes critères pourraient être construits pour comparer les approches utilisées pour l'export et la transformation de modèles abstraits. Cependant, comme pour l'import, de tels tableaux ont un intérêt limité car seule l'utilisation d'une ontologie telle que KRLO permet de satisfaire les critères présentés ci-dessus. En effet, l'utilisation d'une ontologie *complète* – c'est à dire, contenant suffisamment de connaissances précisément représentées pour être exploitable – a les mêmes avantages quelle que soit la fonction qui exploite cette ontologie. Ces avantages sont similaires à ceux qu'offre l'utilisation de variables plutôt que de constantes dans un programme.

La partie suivante "Partie 2 :Travail effectué" montre que KRLO permet de répondre aux questions de recherche de cette thèse. Elle présente également le résolveur de requêtes que j'ai développé pendant cette thèse et que j'ai utilisé pour tester l'export via KRLO.

Partie 2 : Travail effectué

4. KRLO, une ontologie de LRCs

Un des apports de cette thèse fut de contribuer à la conception de KRLO. KRLO est la première ontologie qui représente i) des modèles abstraits de LRCs de différentes familles, par exemple, le modèle de Common Logics et les modèles RDF+OWL, ii) des notations de LRCs, et iii) des fonctions ou des règles spécifiant des méthodes d'import, de traduction et d'export de RCs. Cette section présente la façon dont les modèles et notations de LRCs sont représentés dans KRLO. La section “[4.1. Contexte](#)” présente les parties de KRLO et leurs idées sous-jacentes qui ont été conçues par Philippe Martin. La section “[4.2. Travail effectué](#)” présente mes apports dans KRLO. La section “[4.3. Expressivité](#)” fournit des détails sur l'expressivité requise pour i) représenter des éléments de LRCs et donc les LRCs eux mêmes, et ii) pour importer, exporter ou traduire des RCs via les fonctions ou règles par défaut spécifiées dans KRLO. La section “[4.4. Complétude de KRLO](#)” fournit des détails sur les protocoles permettant de déterminer la complétude de KRLO. La section “[6. Import, export et traduction de LRCs exploitant KRLO](#)” présente ces fonctions ou règles.

La conception de la première version de KRLO s'est achevée en 2014, dans ce rapport, cette version est nommée KRLO_2014 [Martin & Bénéard, 2014][Bénéard & Martin, 2015]. Dans cette première version, j'ai conçu des spécifications de notations des EAs de JSON-LD, KIF ainsi que des spécifications de notations pour RIF. Une seconde version nommée KRLO_2015+ est en cours de conception depuis début 2015 [Martin & Bénéard, 2016b, 2017b]. Dans cette seconde version, les descriptions des EAs et de leurs représentations concrètes sont plus modulaires – et donc plus réutilisables – et des traductions entre EAs sont spécifiées. Une de mes contributions dans KRLO est d'avoir identifié que des descriptions dans KRLO_2014 manquaient de précision ; les parties “[6.1.2. Validation, mise en œuvre et exemples](#)” et “[6.2.1. Nécessité d'une traduction entre EAs](#)” donnent des détails sur ces manques et présentent des comparaisons entre les deux versions de KRLO.

KRLO est accessible à cette adresse http://www.webkb.org/kb/it/o_knowledge/o_KRL. KRLO a plusieurs fichiers d'entrée/sauvegarde. Dans ces fichiers les symboles “\.” et “/^” sont utilisés pour représenter les relations de spécialisation. Voici une catégorisation de ces fichiers.

- Notions sémantiques ou structurelles
 - `d_KRLmodelsTop.html` : ontologie de haut niveau des types d'EAs
 - `d_KRLmodels.html` : types d'EAs de modèles de LRCs particuliers
 - `d_KRLmodelsTranslation.html` : règles de traduction entre quelques types d'EAs de différents modèles
- Notions de présentation
 - `d_KRLnotationsTop.html` : types (de concept/relation/fonction) primitifs pour spécifier des notations de LRCs et des fonctions d'exports
 - `d_KRLnotations.html` : types d'ECs de notations particulières de LRCs
 - `d_presentation.html` : ontologie de haut niveau pour des notions de formatage de documents ou de fenêtres
- Autres langages particuliers
 - `d_grammars.html` : spécifications de langages de définition de grammaires

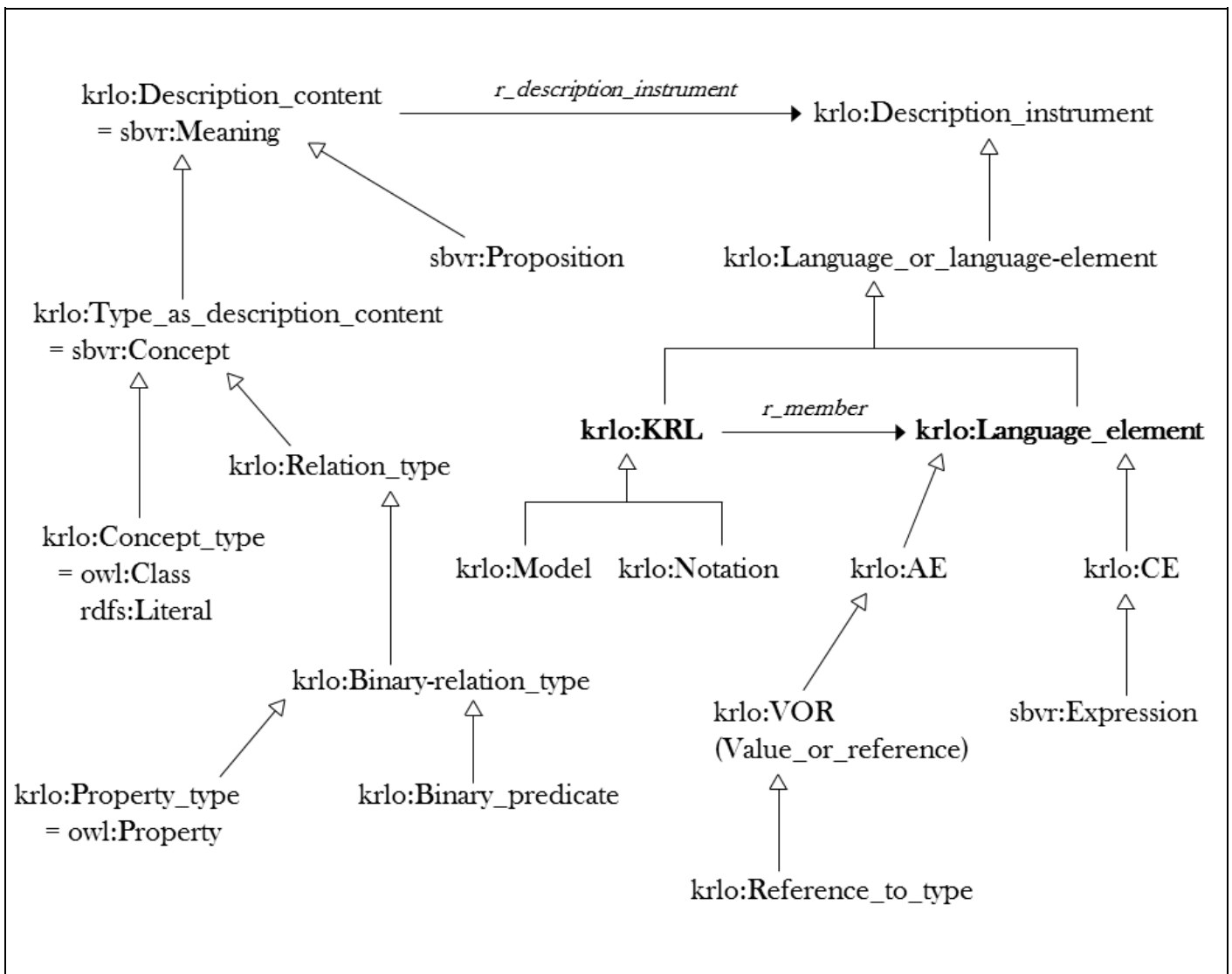
4.1. Contexte

4.1.1. Idées clefs sous-jacentes

4.1.1.1. Une ontologie d'instruments de descriptions

La figure ci-après présente quelques *sous-types* de contenus de description et d'instruments de description, c'est à dire, des instruments pour décrire ces contenus. Dans KRLO, Philippe Martin et moi avons représenté des instruments de description, quelques contenus de descriptions sont déclarés et exploités. Ainsi, KRLO est essentiellement une ontologie d'instruments de descriptions et non une ontologie de contenus de descriptions. La figure ci-après situe quelques types de SBVR (Semantic of Business Vocabulary and Business Rules ; un LRC proposé par l'OMG) [SBVR, 2008], de OWL et de RDFS dans KRLO. Pour les espaces de noms, dans cette figure, j'utilise la notation XML pour spécifier l'ontologie d'origine des types.

Figure 4.1. Contenus de descriptions et instruments de descriptions dans KRLO en pm#UML.



Les contenus de description sont par exemple des types de concepts, des types de relations ou des phrases. Les instruments pour décrire ces contenus sont typiquement des éléments de langages. Les contenus de description et les instruments de description sont respectivement semblables aux “sens” (parfois également appelé “signifiés”) et aux “symboles” (parfois également appelés “signifiants”) dans les *triangles sémiotiques*. Dans de tels triangles, des relations “symbolise” ou “signe” sont utilisées pour relier les contenus de description aux instruments de description. Dans KRLO, le type de relation sémantique *r_description_instrument* est utilisé pour relier les contenus de description aux instruments de description. Cette distinction permet des descriptions sémantiquement plus précises et donc plus réutilisables. Plus de détails sont donnés dans la définition de Précision sémantique dans la section “2.1. Définitions”.

Dans les ontologies de modèles abstraits existantes et dans les documentations formelles ou informelles de ces modèles, la distinction entre contenu et instrument de description est rarement spécifiée. Plus précisément, dans les ontologies de modèles abstraits existantes, les types de contenus et les types d'instruments sont mélangés et seule la documentation permet parfois de les différencier. C'est, par exemple, le cas dans l'ontologie de SBVR, bien que la documentation permette de distinguer quelques types de contenus et quelques types d'ECs. Dans KRLO, grâce à ses types de haut niveau, les distinctions sont formelles et spécifiées de façon systématique. Ainsi, dans KRLO, les EAs et les ECs sont des éléments de langages donc des instruments de description et non des contenus de descriptions.

4.1.1.2. Opérateur, arguments et résultat

Dans KRLO, les EAs et les ECs sont décrits via un opérateur, des arguments et un résultat.

La structure de tout élément de LRC peut être représentée de façon uniforme via un opérateur, des arguments et un résultat. Implicitement, les éléments de langage dans la programmation fonctionnelle sont aussi représentés de cette façon. Dans KRLO, les six types primitifs les plus importants de relations entre les EAs sont nommés *r_operator*, *r_argument*, *r_arguments*, *r_result*, *r_parts* et *r_part*. Les types *r_operator* et *r_argument* sont des sous-types de *r_part*. Organisée via ces relations, la structure de tout élément de LRC a une forme d'arbre. Les extraits ci-après décrivent – en FL – des relations sous-type depuis ou vers ces six

types (cf. caractères gras) ou d'autres types de relations de haut niveau. La description complète peut être trouvée dans [d_KRLmodelsTop.html](#).

```

(r_relation_to_AE ~[Thing, AE] // ~[ ] délimite une signature de relation
  > (r_relation_from_and_to_AE ~[AE, AE]
    > (r_relation_to_AE_part
      < r_part, // donc r_part > r_relation_to_AE_part
      > (r_reification_relation_to_AE
        > (r_operator ~[AE, Operator]
          > r_frame_head // similaire à rdf#predicate
        )
      (r_argument
        > (r_frame-or-link_argument
          > r_half_link
            r_link_source // similaire à rdf#subject
            r_link_destination // similaire à rdf#object
          ) ) )
      r_result ~[AE, AE]
    ) );

```

Les structures de quelques instances d'EAs – donc d'éléments de langages – sont présentées et commentées ci-dessous. Pour rappel (cf. section “[2.4. Conventions et exemples de traduction](#)”), dans toutes ces descriptions, les types de relations qui sont *sous-types* de `Type_as_description_instrument` sont préfixés par “`r_`” tous les autres sont préfixés par “`has_`”.

Voici une description en FL pour une instance du type d'EA “Link” représentant une relation de partie entre les USA et l'Iowa. Une présentation concrète de cet EA dans une notation préfixée (comme celle de KIF) peut être “`has_part USA Iowa`”.

```

AE__has-part_USA_Iowa
  r_type: Link,
  r_operator: a ^{(ae // "(" délimite un Concept-type-expression
    r_description_instrument of: (has_part
      r_type: Type_as_description_content)),
  r_argument: (AE__USA r_type: Constant)
    (AE__Iowa r_type: Constant),
  r_result: True;

```

Voici une description en FL pour une instance du type d'EA “Existentially_quantified_formula” représentant une relation de partie entre les USA et au moins un état des USA. Une présentation concrète de cet EA en KIF peut être “`(Exists ((State ?s)) (has_part USA ?s))`”.

```

AE__exist_s_a_state_such_that_has-part_USA_s
  r_type: Existentially_quantified_formula,
  r_operator: ^{(ae r_description_instrument of: (Existential_quantifier
    r_type: Type_as_description_content) ),
  r_argument: (AE__has-part_USA_s r_type: Link)
    (Guard r_type: Guard_type),
  r_result: True;

```

Voici une description en FL pour une instance du type d'EA “Function_call” représentant un appel d'une fonction `getStates`. La présentation concrète de cet EA dans une notation préfixée (comme celle de KIF) peut être : `(getStates USA)`.

```

AE__getStates_USA
  r_type: Function_call,
  r_operator: ^{(ae r_description_instrument of: (get_states r_type: Function_type)),
  r_argument: (AE__USA r_type: Constant),
  r_result: (AE__Result_of_getStates_USA r_type: List);

```

Voici une description en FL pour une instance du type d'EA “variable” qui est une référence vers l'AE “Iowa”. La présentation concrète de cet EA en KIF est : `?i`.

```

AE__variable_i
  r_type: Variable,
  r_operator: ^{(ae r_description_instrument of: ("?i" r_type: Name)),
  r_result: (AE__Iowa r_type: Constant);

```

Comme illustré dans les exemples ci-avant, un opérateur d'EA peut être un type de fonction, de relation ou même de collection, par exemple une liste. Les relations *r_operator*, *r_argument* et *r_result* sont des relations sémantiques permettant de spécifier la structure des éléments de langage. Un EA est donc une représentation réifiée, structurelle et sémantique d'un élément de langage. En particulier, un opérateur d'EA joue uniquement un rôle structurel. Ainsi, lorsque un EA représente une formule, il ne représente pas directement sa sémantique. Par exemple, une relation n'a pas réellement de résultat. Dans RDF, la réification d'un triplet est aussi une description structurelle sémantique et cette description ne représente pas directement la sémantique du triplet.

4.1.1.3. Présentation ordonnée des parties

La présentation concrète d'un EA dans une notation est représentée comme une présentation ordonnée de chaque partie de cet EA. Grâce à une telle représentation, la présentation concrète d'une instance d'EA peut être obtenue en :

1. parcourant les parties de cet EA ;
2. traduisant chacune de ces parties lorsque le modèle cible n'admet pas ce type d'EA ;
3. récupérant l'EC spécifié pour chacune de ces parties dans la notation cible ;
4. concaténant, dans l'ordre spécifié, chacun des ECs ainsi récupéré.

Dans une notation homo-iconique, si un EA est structuré – c'est à dire, a un opérateur ou des arguments – la présentation concrète de cet EA dans une notation donnée est un EC structuré. Cet EC structuré est dérivé de la présentation concrète des parties de cet EA avec éventuellement du sucre syntaxique pour délimiter ces présentations concrètes. Dans les notations textuelles, cette dérivation est un ordonnancement de la présentation concrète de l'opérateur et des arguments de l'EA. Par exemple, dans la notation de KIF, une relation de type *r* depuis le noeud *source* vers le noeud *destination* peut s'écrire “(*r* source destination)” ; comme le montre cet exemple, la représentation concrète de l'opérateur de la relation est placée avant la représentation concrète des arguments de cette relation et des parenthèses sont utilisées comme délimiteurs.

Dans une notation qui n'est pas homo-iconique, si un EA est structuré la présentation concrète de cet EA dans une notation donnée n'est pas nécessairement un EC structuré. Dans les notations textuelles, la dérivation de la présentation des parties d'un EA est :

- un ordonnancement, et/ou
- un filtrage – et, dans ce cas, certaines présentations concrètes ne figureront pas dans la présentation finale de l'EA –, et/ou
- une abbréviation – par exemple, “[]” est une abbréviation pour une relation “next” sur une liste en Prolog.

Les parties “4.1.3.2. Spécifier des notations dans KRLO_2014” et “4.1.4. Spécifications de notations d'éléments de modèles abstraits dans KRLO_2015+” montrent comment spécifier des présentations concrètes d'EAs dans KRLO_2014 et KRLO_2015+.

4.1.2. Modèles abstraits de LRCs

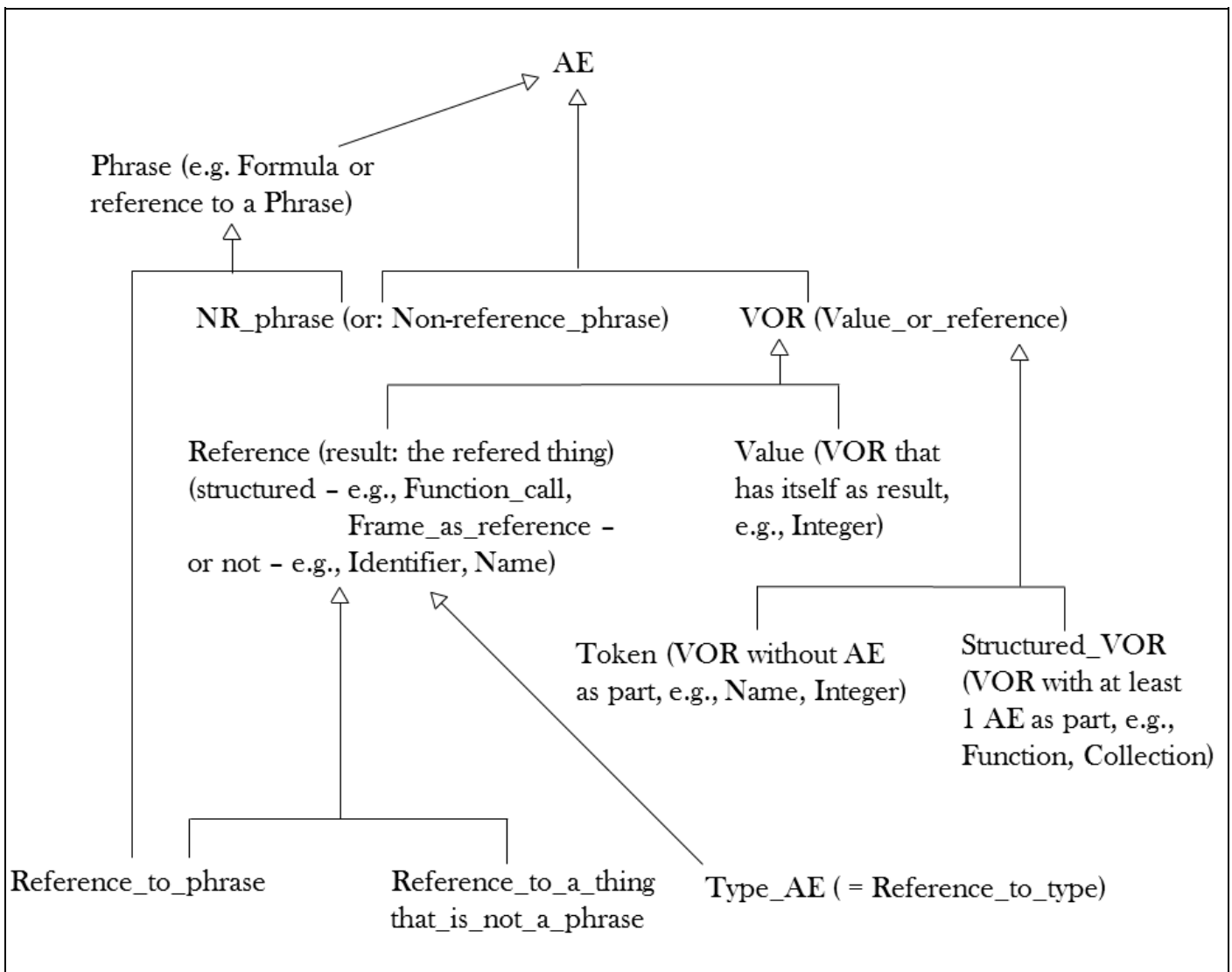
4.1.2.1. Ontologie de haut niveau

Cette section présente les principaux types d'EAs de haut niveau (cf. définition de ontologie de haut niveau) représentés dans KRLO. Le fichier qui contient les descriptions des EAs de haut niveau de KRLO est accessible dans `d_KRLmodelsTop.html`. Pour faciliter la lecture, ces types d'EAs sont organisés dans des partitions, c'est à dire, des unions disjointes. Cependant, ces partitions ne sont pas exploitées pour les fonctions d'import, de traduction et d'exports proposés dans KRLO.

Les figures ci-après présentent ces principaux types d'EAs et les partitions utilisées pour les organiser. Comme tous ces types d'EAs sont des EAs de KRLO, leur espace de nom est laissé implicite. Avant de trouver ces partitions générales, il était difficile de catégoriser les types d'EAs et de modulariser ou d'organiser les descriptions de l'ontologie des modèles de LRC. De plus, ces partitions étaient nécessaires car plusieurs généralisations de ces types se sont révélées utiles pour la représentation d'EAs de modèles particuliers. Par exemple, dans les modèles basés sur une logique d'ordre supérieur – tels RIF-FLD – une référence vers une phrase peut être utilisée chaque fois qu'une phrase peut être utilisée. Ce n'est pas le cas dans les modèles qui ne sont pas basés sur une logique d'ordre supérieur. Ainsi, il était utile de distinguer et donc de représenter les types d'EAs suivants : `Reference_to_a_phrase`, `NR_phrase` et `Phrase`.

La figure ci-après montre quelques *sous-types* importants de AE.

Figure 4.2. Quelques relations entre des sous-types importants de AE en pm#UML.



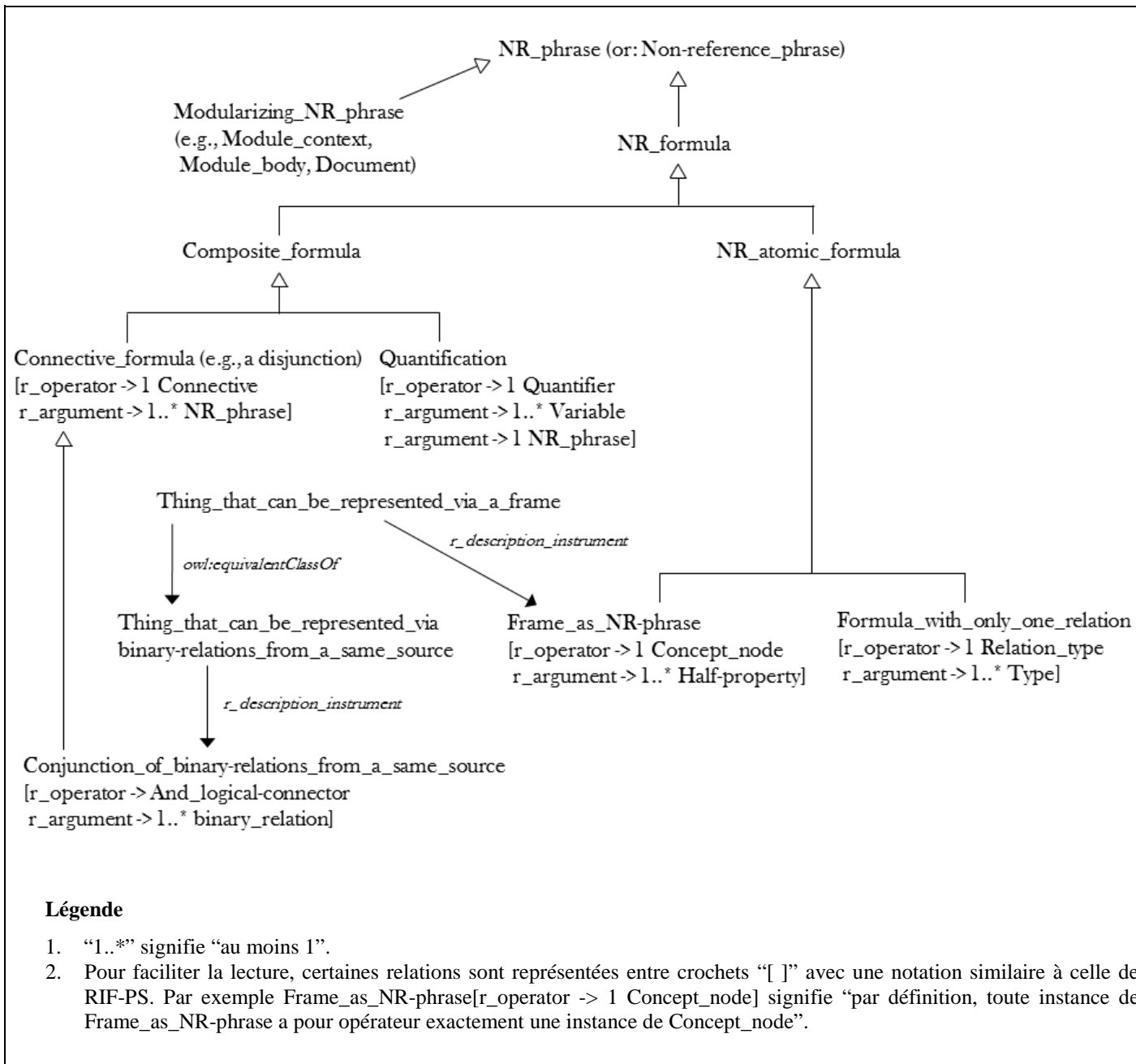
Voici une description informelle de chacun des types d'EAs représentés dans la figure ci-dessus.

- AE est une représentation d'un élément abstrait dans KRLO ; pour plus de détails, vous pouvez consulter la définition de Élément abstrait dans la section "2.1. Définitions".
- Phrase est un *sous-type* de AE qui ne représente pas une valeur. De plus, dans les logiques d'ordre 1 ou dans les logiques de description, des EAs tels que Logical_operator ne sont pas des phrases. Par exemple, And_connective un EA qui représente le connecteur logique de conjonction n'est pas une phrase.
- NR_phrase est un *sous-type* de Phrase et n'est pas un Reference_to_phrase, c'est à dire, un *sous-type* de Reference dont le résultat n'est pas *sous-type* de Phrase. Par exemple, une formule est représentée dans KRLO par le type d'EA Formula. Formula est un *sous-type* de Phrase et a pour résultat un booléen. Formula est donc un *sous-type* de NR_phrase.
- VOR (Value_or_reference) est un *sous-type* de AE qui peut représenter une valeur (le type d'EA Value) ou une référence (le type d'EA Reference). Par exemple, Integer est un *sous-type* de Value qui représente une valeur numérique entière et Function_call est *sous-type* de Reference qui représente un appel de fonction.
- Token est un *sous-type* de VOR qui n'a pas de relation *r_part*. Par exemple, Token peut représenter une variable ou une constante.
- Structured_VOR est un *sous-type* de VOR qui a au moins une relation *r_part*. Par exemple, Structured_VOR peut représenter un appel de fonction.
- Reference_to_phrase est un *sous-type* de Reference et a pour résultat un EA de type Phrase. Reference_to_phrase est donc également un *sous-type* de Phrase.
- Reference_to_a_thing_that_is_not_a_phrase est un *sous-type* de Reference et a pour résultat un EA qui n'est pas un *sous-type* de Phrase.
- Type_AE est un *sous-type* de Reference et a pour résultat un type de concept ou de relation. Par exemple, www.w3.org/2002/07/owl#Class est une référence vers un type de concept dans OWL.

La figure ci-après montre quelques *sous-types* importants de NR_phrase. Dans KRLO, les types de valeurs, de références et d'ECs sont structurés de façon similaire. Les deux sens du mot frame sont représentés via les types Frame_as_NR-phrase (*sous-type* de NR_phrase) et Frame_as_reference (*sous-type* de Reference, comme Function_call). Dans certaines notations – par

exemple, FL, la notation qui a été utilisée pour écrire les deux versions de KRLO – la syntaxe permet de faire la distinction entre ces deux types d'AEs. Dans les autres notations, la représentation d'un Frame avec ses propriétés réfère systématiquement soit à une formule (comme dans RIF-PS), soit à la source de ces propriétés (comme en JSON-LD).

Figure 4.3. Quelques relations entre des sous-types importants de Phrase en pm#UML.

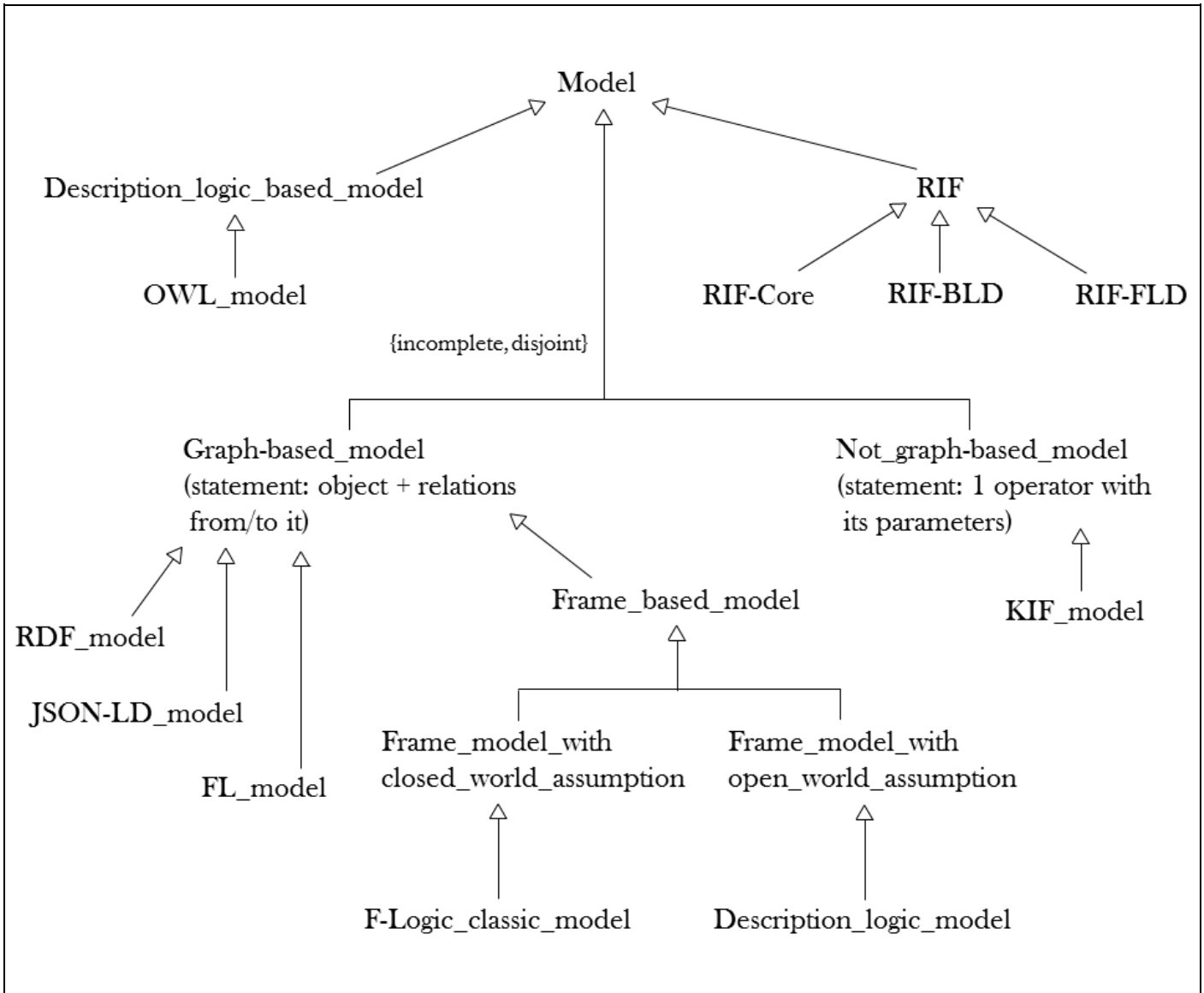


4.1.2.2. Ontologies de modèles particuliers

Cette section présente la façon dont sont organisés les modèles de LRCs particuliers et dont les EAs de ces modèles sont spécifiés. Le fichier qui contient les descriptions des EAs de modèles particuliers est accessible dans `d_KRLmodels.html`.

Dans la figure ci-après, quelques uns des modèles de LRCs représentés dans KRLO sont présentés.

Figure 4.4. Quelques relations entre des modèles de LRCs en pm#UML.



Dans KRLO, les types d'EAs de modèles particuliers sont i) reliés aux modèles abstraits par des relations *r_member*, ii) des sous-types de types d'EAs de haut niveau. Plus précisément, les types d'EAs de modèles particuliers sont spécifiés via des concept-type-expressions. Par exemple, dans la notation FL, une spécialisation du type des EAs “VOR dans un modèle X” peut être spécifiée par “^(VOR r_member of: X)”, “VOR@X” ou “f_in_(VOR, X)”.

Dans KRLO_2014, la relation *r_only_such_part_of_that_type* est utilisée plutôt que la relation *r_member* pour relier les modèles à leurs EAs. Le type *r_only_such_part_of_that_type* est une spécialisation du type *r_part*. Les relations *r_only_such_part_of_that_type* excluent de leurs destinations les généralisations de ces destinations. La définition de *r_only_such_part_of_that_type* est présentée ci-dessous en FL.

```

r_only_such_part_of_that_type ~[?g ?pt] //?g a des parties de type ?pt mais
< r_part_(?g ?pt), //aucune partie avec un type généralisant celui de ?pt.
:= [?g r_part: 1..* ?pt @^(?t != ?pt, < (?gpt r_genus_supertype of: ?pt))];
    
```

Des exemples de spécification de notation dans KRLO_2014 sont présentés dans la partie “4.2.2. Spécifications de notations des éléments de modèle abstrait dans KRLO_2014”.

L'extrait en FL ci-dessous présente quelques spécialisations de `Abstract_element` dans différents modèles dans `KRLO_2015+`. Ces descriptions sont extraites de `d_KRLmodels.html` où plus de 900 types d'EAs de modèles particuliers sont reliés.

```

Abstract_element@CL // Abstract_element in the CL model
  > partition // "partition { }" délimite une partition de sous-types
  { NR_phrase@CL, // phrase that is not a reference in CL model
    Value_or_reference@CL }
  (AE_that_can_be_annotated_without_link@CL
    > cl#Text_construction cl#Sentence cl#Term ); // cl# : namespace

Abstract_element@RIF-FLD // Abstract_element in the RIF-FLD model
  > (AE_that_can_be_annotated_without_link@RIF-FLD
    > NR_phrase@RIF-FLD
      (rif-flt#Term = Value_or_reference@RIF-FLD) ) // "=" abreviates "equivalent"
  (rif-flt#Termula
    > partition { NR_formula@RIF-FLD rif-flt#Term } );

Abstract_element@RDF // Abstract_element in the RDF model
  < rdfs#Resource
  Abstract_element@Conjunctive-existential-formula_based_model,
  > partition
  { (NR_phrase@RDF
    > partition
      { Modularizing_NR-phrase@RDF
        (NR_formula@RDF = rdf#Statement,
          > partition
            { (NR-Composite_formula@RDF
              > partition { Conjunction_of_formulas@RDF
                Existentially_quantified_formula@RDF } )
              NR_link@RDF
            } )
          } )
    Value_or_reference@RDF
  }
  (Abstract_element@OWL // Abstract_element in the OWL model
    > partition
      { (NR_phrase@OWL
        > partition
          { Modularizing_NR-phrase@OWL // e.g. : owl#Ontology or
            // owl#Ontology_Importation
            (owl#Axiom < NR_atomic_formula@OWL)
          } )
        Value_or_reference@OWL
      } );

```

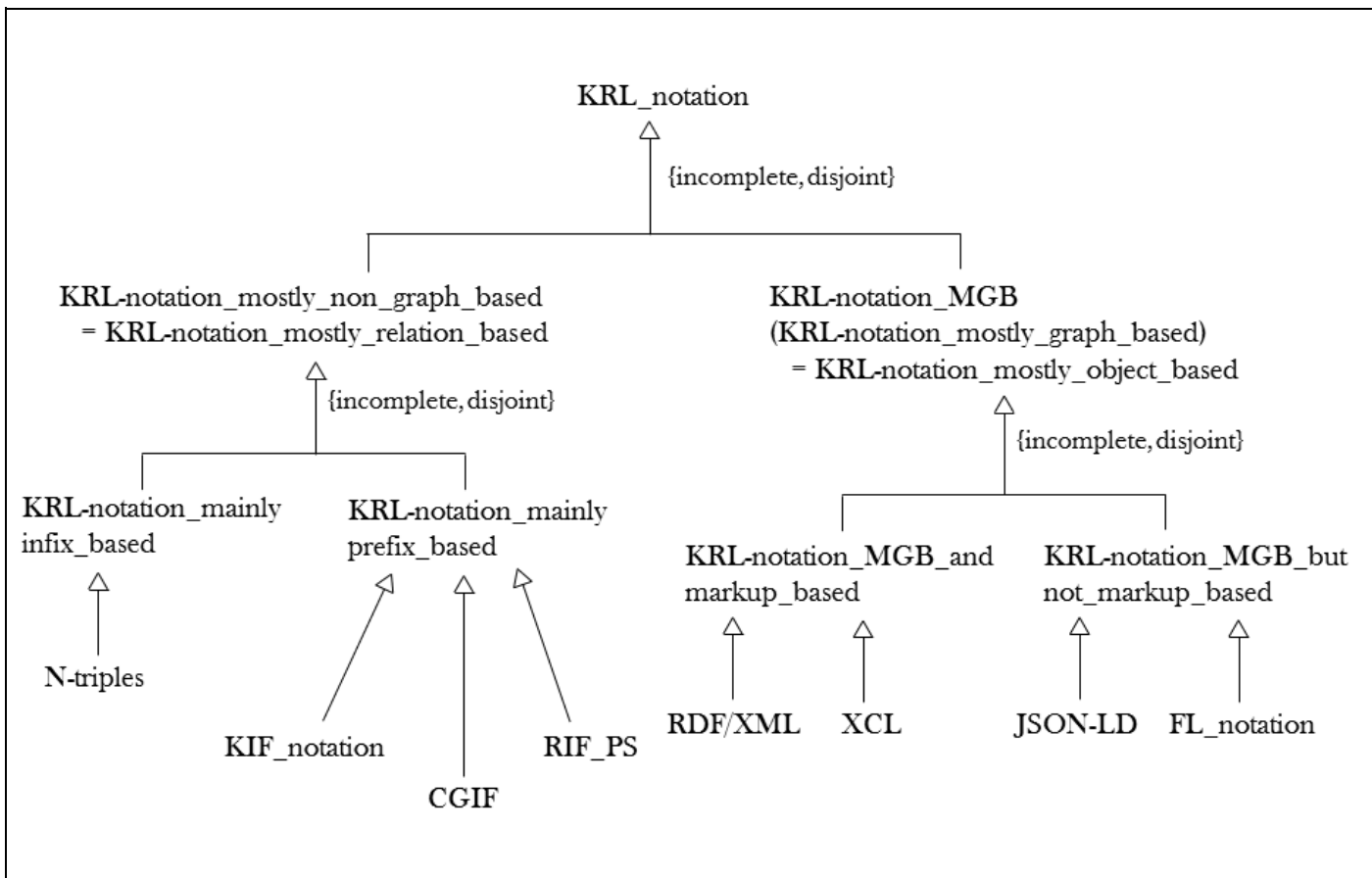
4.1.3. Notations (modèles concrets) de LRCs

4.1.3.1. Relations entre notations et types d'éléments de notation de haut niveau

Cette section présente i) la façon dont les notations de LRCs particuliers sont organisées et, ii) les principaux types d'ECs de haut niveau. Les descriptions des notations sont dans `d_KRLnotations.html`. Les descriptions des principaux types d'ECs de haut niveau sont dans `d_KRLnotationsTop.html`.

Dans la figure ci-dessous, quelques types de notations de LRCs dans KRLO sont présentés. Une version plus complète et écrite en FL est accessible dans [d_KRLnotations.html](#). Certains de ces types sont un peu arbitraires mais utiles pour la catégorisation. Par exemple, il est utile de distinguer `KRL-notation_mostly_relation_based` et `KRL-notation_mostly_object_based` puisque la plupart des EAs sont présentés différemment dans ces deux types de notations. Plus de détails sur ces différences de présentation sont donnés ci-dessous, après la figure.

Figure 4.5. Quelques relations entre des notations de LRCs en `pm#UML`.

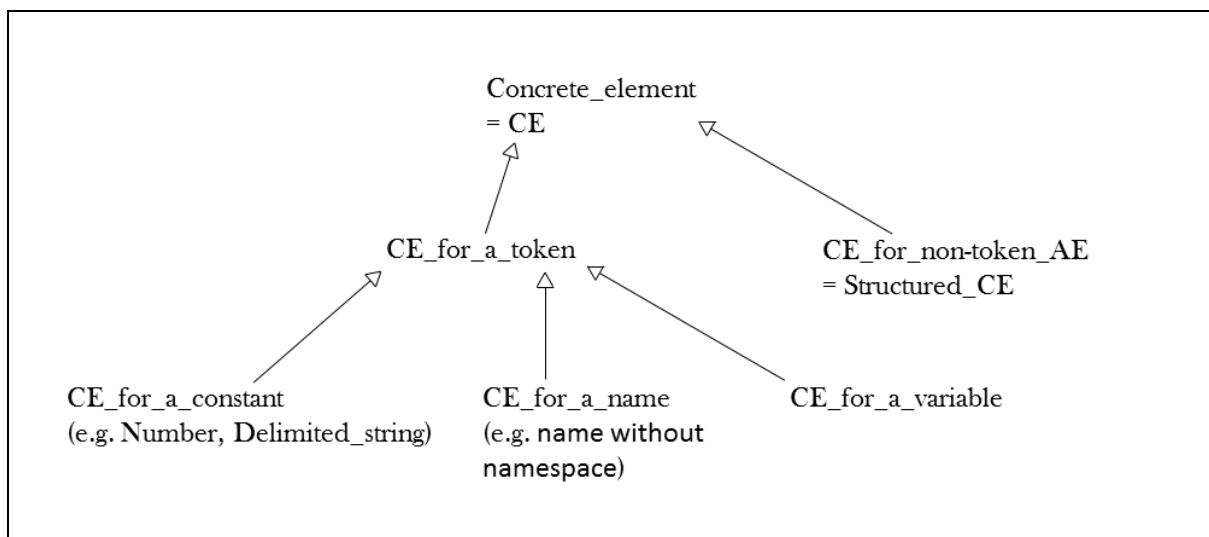


Ainsi organisées, les notations peuvent être facilement comparées et retrouvées. De plus, cela permet à leurs spécifications d’être décrites de façon modulaire et donc d’être plus réutilisables. Par exemple, une spécification de notation pour toute `KRL-notation_mainly_prefix_based` peut être ré-utilisée par défaut pour toute notation qui en hérite, comme `KIF_notation`, `CGIF` ou `RIF_PS`. Ainsi, pour chaque notation, seuls les types d’ECs spécifiques de cette notation doivent être spécifiés. Les parties “4.1.3.2. Spécifier des notations dans KRLO_2014”, “4.1.3.3. Spécifier des notations dans KRLO_2015+” et “4.1.4. Spécifications de notations d’éléments de modèles abstraits dans KRLO_2015+” donnent des détails sur les primitives utilisées pour ces spécifications. La section “4.2. Travail effectué” donne des exemples de spécifications.

Dans une “notation basée sur des relations”, les noeuds concepts sont présentés comme des arguments d’un noeud relation. Ainsi, des descriptions écrites avec une notation relationnelle ressemblent aux appels de fonctions écrits dans la plupart des langages de programmation. Dans une “notation basée sur un graphe” (frame, objet, ...), les noeuds concept ne sont pas présentés comme des arguments de noeuds relation mais reliés via du sucre syntaxique tel “:”, “;”, “,”, “[]”, “()”, etc. Certaines notations sont majoritairement basées sur des relations mais autorisent pour certains éléments de langages une “notation basée sur un graphe”. Dans RIF-PS, par exemple, un EA de type `Binary_relation` peut être présenté des deux façons suivantes : “USA[has_part->Iowa]” (notation basée sur un graphe) ou “has_part(USA Iowa)” (notation basée sur les relations).

La figure ci-dessous présente les principaux types de haut niveau d’ECs. Une version plus complète et écrite en FL est accessible dans [d_KRLnotationsTop.html](#).

Figure 4.6. Principaux sous-types de haut niveau de CE en pm#UML.



CE_for_a_token est un type d'EC qui n'a pas de sous-partie. Structured_CE est un type d'EC qui a des sous-parties.

4.1.3.2. Spécifier des notations dans KRLO_2014

Cette section présente les types majeurs utilisés pour spécifier et catégoriser des ECs dans KRLO_2014, la première version de KRLO. Ces types majeurs sont décrits dans [d_KRLnotationsTop.html#StructuredCE_initialHierarchy](#).

KRLO_2014 décrit seulement certains positionnements possibles pour les parties d'un Structured_CE. Pour cela, KRLO_2014 utilise des types dont les noms décrivent cet ordonnancement. Ces types et leur organisation sont donnés dans [d_KRLnotationsTop.html#StructuredCE_initialHierarchy](#). Par exemple, le positionnement des parties pour le type Prefix_fct-like_CE est “operator (arguments)” où les parenthèses sont du sucre syntaxique pour délimiter les arguments. Ainsi, KRLO_2014 décrit plusieurs types de Structured_CE dont les 5 sous-types majeurs de Structured_CE_with_operator. Une hiérarchie de ces sous-types de Structured_CE est présentée ci-dessous, une hiérarchie plus complète est donnée dans [d_KRLnotationsTop.html#StructuredCE_initialHierarchy](#). Dans la hiérarchie ci-dessous, les types majeurs de Structured_CE_with_operator sont en police grasse. Ces types sont qualifiés de “majeurs” car toute spécification concrète d'EA (avec un opérateur) de tout langage doit pouvoir être décrite via ces sous-types.

- Structured_CE_without_operator
 - String_no-op-CE
 - List_no-op-CE
 - Classic_list_CE
 - Set_no-op-CE
 - Set-like_frame_CE
 - Untyped_sentence-conjunction_CE
- Structured_CE_with_operator
 - Prefix_operator_CE
 - **Prefix_fct-like_CE**
 - **Prefix_list-like_CE**
 - Alternating-XML_CE
 - Postfix_operator_CE
 - **Postfix_fct-like_CE**
 - **Postfix_list-like_CE**
 - **Infix_operator_CE**
 - Un-keyed_infix_operator_CE
 - Keyed_infix_operator_CE

KRLO_2014 utilise des fonctions ou des lambda-expressions pour spécifier des spécialisations de ces types majeurs pour des notations particulières. De nombreuses fonctions portant des noms longs et – malgré cette longueur – parfois peu clairs ont ainsi été créées. Par exemple, la fonction `fc_list-like_prefix-fct_type` permet de spécifier un sous-type de `Prefix_list-like_CE`. Pour un AE “ae” ayant pour opérateur “o” et pour arguments “x” et “y”, quelques présentations concrètes exploitant certains des types majeurs de Structured_CE est illustrée ci-après. Les parenthèses sont utilisées comme délimiteurs.

- Une présentation concrète de “ae” spécifiée via Prefix_fct-like_CE est : “o (x y)”.
- Une présentation concrète de “ae” spécifiée via Prefix_list-like_CE est : “(o x y)”.
- Une présentation concrète de “ae” spécifiée via Postfix_fct-like_CE est : “(x y) o”.
- Une présentation concrète de “ae” spécifiée via Postfix_list-like_CE est : “(x y o)”.
- Une présentation concrète de “ae” spécifiée via Infix_operator_CE est : “(x o y)”.

L'exemple ci-dessous montre une spécification en FL de la présentation concrète d'un EA du modèle Not_graph-based_KRL_model dans la notation KRL-notation_lisp_based.

```

Not_graph-based_KRL_model  r_only_such_part_of_that_type:
  ^(AE rc_type: (fc_list-like_prefix-fct_type
    _ (List(KRL-notation_lisp_based), //exemple de sous-types : CLIF KIF
      " ", //présentation par défaut de l'opérateur
      "( ", //délimiteur de début des arguments
      " ", //séparateur des arguments
      ")") //délimiteur de fin des arguments
    ));

```

La section “4.2.2. Spécifications de notations des éléments de modèle abstrait dans KRLO_2014” donne des exemples de spécifications d'ECs dans KRLO_2014.

Des tests effectués sur KRLO_2014 ont montré que ces spécifications n'étaient pas assez précises et modulaires pour permettre l'export de n'importe quel EA structuré. Ainsi, les 5 types majeurs d'ECs de KRLO_2014 ne forment hélas pas une partition (complète). Par conséquent, la description des spécifications de notations de certains EAs n'est pas possible pour certaines notations, ce qui est une limite importante à l'utilisation de KRLO_2014. Cette limite peut être levée en ajoutant d'autres types à l'ensemble des types majeurs. Cependant cette solution ne passe pas l'échelle car elle nécessite la création d'un sous-type pour chaque particularité dans la notation. C'est pourquoi, dans KRLO_2015+, les sous-types de Structured_CE ne sont plus spécifiés directement dans l'ontologie ; des instances de Structured_CE sont construites à partir de quelques fonctions de spécification d'opérateurs et d'arguments d'un EA structuré et d'une seule fonction *fc_spec* qui spécifie la présentation d'un élément abstrait pour une notation. La section “4.1.3.3. Spécifier des notations dans KRLO_2015+” présente les fonctions de spécification de parties de Structured_CE. La section “4.1.4. Spécifications de notations d'éléments de modèles abstraits dans KRLO_2015+” montre comment un Structured_CE peut être spécifié pour un EA. La première section de la section “4.2.1. Problèmes identifiés” donne des détails sur le manque de précision et de modularité des spécifications de notations dans KRLO_2014.

4.1.3.3. Spécifier des notations dans KRLO_2015+

Cette section présente les primitives utilisées pour spécifier des ECs dans KRLO_2015+. Ces primitives sont décrites dans *d_KRLnotationsTop.html*.

Dans KRLO_2015+, les parties de Structured_CE peuvent être spécifiées via les quelques fonctions primitives décrites ci-dessous. Ces fonctions i) prennent en entrée une instance d'EA, ii) ont en sortie une présentation – donc un EC – pour cet EA et iii) permettent de spécifier pour cet EC en sortie un rôle dans le Structured_CE. Ainsi, chaque EC en sortie de ces fonctions est soit un opérateur, soit un argument d'un Structured_CE.

- La fonction *fc_OP* prend en entrée un EA, en paramètre une notation et a en sortie un EC qui est un opérateur d'un Structured_CE.
- La fonction *fc_ARG* prend en entrée un EA, en paramètre une notation et a en sortie un EC qui est un argument d'un Structured_CE.
- La fonction *fc_ARGS* prend en entrée un EA, en paramètre une notation et a en sortie une liste d'ECs qui sont des arguments d'un Structured_CE.
- La fonction *fc_OP_from* est similaire à la fonction *fc_OP* mais renvoie une présentation concrète de l'opérateur de l'EA en entrée. *fc_OP_from* exploite la fonction *f_OP_from* qui prend en entrée un EA et a en sortie l'opérateur de cet EA.
- La fonction *fc_ARG_from* est similaire à la fonction *fc_ARG* mais renvoie une présentation concrète de l'argument de l'EA en entrée. *fc_ARG_from* exploite la fonction *f_ARG_from* qui prend en entrée un EA et a en sortie un argument de cet EA.
- La fonction *fc_ARGS_from* est similaire à la fonction *fc_ARGS* mais renvoie une présentation concrète des arguments de l'EA en entrée. *fc_ARGS_FROM* exploite la fonction *f_ARG_from* qui prend en entrée un EA et a en sortie les arguments de cet EA.

Pour écrire les spécifications de notation, la fonction *fc_spec* permet d'utiliser directement les fonctions *f_OP_from*, *f_ARG_from* et *f_ARGS_from*. Dans cette thèse, les spécifications de notation sont écrites via la fonction *fc_spec*.

La section “4.1.4. Spécifications de notations d’éléments de modèles abstraits dans KRLO_2015+” donne des exemples de spécifications de Structured_CE. La section “6.1.1.1. Fonctions primitives utilisées par les spécifications d’export” donne les définitions en FL des fonctions listées ci-dessus.

4.1.4. Spécifications de notations d’éléments de modèles abstraits dans KRLO_2015+

Cette section présente la façon dont sont représentées les spécifications de notations d'EAs pour des notations particulières. Le fichier qui contient les descriptions de ces spécifications est d_KRLnotations.html.

Dans KRLO_2015+, la fonction `fc_spec` permet de spécifier des représentations concrètes d'EAs. Cette fonction a en sortie un EC dans une notation particulière et prend en entrée i) une liste de CE, ii) un ensemble de notations. Cette liste contient soit un `CE_for_a_token`, soit les parties d'un `Structured_CE`. Elle permet alors de spécifier un ordre pour ces parties. Ainsi, contrairement à KRLO_2014, dans KRLO_2015+, l'ordonancement des parties de la présentation d'un EA n'est pas prédéfini. L'exemple ci-dessous montre une spécification en FL de `CE_for_a_token` dans la notation CLIF et dans la notation de KIF.

```
fc_spec_(List("exists"), //"" delimits a string hence a CE for a textual notation
         List(KIF_notation CLIF)
         ); // the returned CE is an instance of CE_for_a_token
         // that belongs to KIF_notation and CLIF
```

L'exemple ci-dessous montre une spécification en FL de la présentation concrète d'un EA structuré dans la notation CLIF et dans la notation de KIF.

```
AE@Not_graph-based_KRL_model ?ae //any AE from a relation based model (e.g. CL or KIF)
rc_spec: fc_spec_( //has for concrete presentation (rc_spec) the result of
                  //the following specification (fc_spec)
                  List( "(" f_OP_from_(?ae) f_ARGS_from_(?ae) " " ),
                  List(KIF_notation CLIF) );
```

Comme les EAs sont organisés via des relations *subtype*, cette spécification est héritée par tous les sous-types de AE. Ainsi, via cet héritage, des représentations concrètes "par défaut" pour des EAs peuvent être exploitées. Pour chaque notation, une seule relation *rc_spec* est spécifiée pour un type d'EA. Cette relation peut être héritée ou redéfinie [override]. Pour importer vers ou exporter depuis un EA, seule la relation *rc_spec* la plus sémantiquement précise (cf. précision sémantique) est exploitée.

Comme les notations sont organisées via des relations *subtype*, une spécification de représentation concrète pour une notation *N* peut être exploitée pour toute notation sous-type de *N*. Par exemple, la spécification présentée ci-dessous peut être exploitée pour la notation CLIF et pour la notation de KIF.

```
AE@Not_graph-based_model ?ae
rc_spec: fc_spec_(List( "(" f_OP_from_(?ae) f_ARGS_from_(?ae) " " ),
                  List(KRL-notation_lisp_based) // KRL-notation_lisp_based > CLIF KIF
                  );
```

4.2. Contributions à KRLO

4.2.1. Identification de problèmes de modélisation dans KRLO_2014

Cette section présente les problèmes de modélisation que j'ai identifiés dans KRLO_2014.

1) Manque de précision et de modularité des spécifications

J'ai corrigé les problèmes de précision et de modularité des spécifications de manière ad hoc dans KRLO_2014. Les parties “4.1.4. Spécifications de notations d'éléments de modèles abstraits dans KRLO_2015+” et “4.2.3. Spécifications de notations des éléments de modèle abstrait dans KRLO_2015+” montrent comment KRLO_2015+ apporte des solutions plus génériques à ces problèmes.

Dans KRLO_2014, la présentation d'un EA est spécifiée via la spécialisation de l'un des cinq types majeurs de Structured_CE. Or, de telles spécifications étaient i) trop peu précises pour permettre l'export de tout EA structuré et ii) trop peu modulaires pour être combinées et ainsi permettre l'export de tout EA structuré. Ce manque de précision et de modularité cause les problèmes listés ci-après. De plus, les descriptions d'ECs dans KRLO_2014 sont rapidement devenues difficiles à lire, entre autres parce que leurs noms étaient longs et malgré cela parfois peu clairs.

- La position du sucre syntaxique est prédéfinie par un type d'EC structuré majeur. Dans certaines notations, le positionnement du sucre syntaxique ne correspond à celui d'aucun des 5 types majeurs de Structured_CE. Ce problème est illustré par l'exemple 1 ci-dessous. Dans KRLO_2015+, le sucre syntaxique peut être directement positionné dans la liste de présentations d'EAs qui est en paramètre de `fc_spec`.
- Dans KRLO_2014, pour un ensemble (contrairement à une liste) d'EAs, il n'est pas possible de spécifier un ordre sur les ECs représentant ces EAs. Or, dans certaines notations, de tels ECs doivent être ordonnés.. Ce problème est illustré par l'exemple 2 ci-dessous. Dans KRLO_2015+, une spécification de présentation inclut une liste permettant de spécifier la position de toutes les parties d'un EC structuré.
- Dans certaines notations, les parties (opérateur et/ou arguments) de certains EAs ne sont pas *présentées* (filtrage) ou bien sont présentées comme des parties d'un autre EA. Ce problème est illustré dans l'exemple 3 ci-après. Dans KRLO_2015+, il est possible de faire cela.
- Un problème auquel je n'ai pas été confronté est celui des abréviations. Par exemple, l'abréviation “|” en Prolog permet de spécifier des listes (il s'agit ici aussi d'un filtrage et d'un ordonnancement).

Exemple 1, positionnement de sucre syntaxique

Dans la notation RIF-PS, la phrase “IBM a au moins un employé” peut être présentée “Exists ?p (And(?p#Person *hasEmployer*(?p IBM)))”. Le positionnement des parenthèses écrites en caractères gras ne peut pas être spécifié via l'un des 5 types majeurs de Structured_CE de KRLO_2014. En effet, dans KRLO_2014, la présentation concrète de cette phrase doit être spécifiée via l'un des types majeurs d'ECs présentés dans la section “4.1.3.2. Spécifier des notations dans KRLO_2014”. La liste ci-dessous illustre des exports de “IBM a au moins un employé” – une instance d'EA de type Quantified_phrase – vers des notations imaginaires. Les modèles de ces LRCs imaginaires ont pour membre le type Quantified_phrase. Pour chacun de ces modèles, une notation différente est spécifiée pour Quantified_phrase. Chacune de ces notations utilise un type majeur d'EC de KRLO_2014. Pour faciliter la lecture, les notations des autres types d'EAs sont identiques à celle de la notation RIF-PS.

- Un export exploitant principalement Prefix_list-like_CE peut donner :
“(Exists ?p And(*hasEmployer*(?p IBM) ?p#Person))”.
- Un export exploitant principalement Prefix_fct-like_CE donne :
“Exists (?p And(*hasEmployer*(?p IBM) ?p#Person))”.
- Un export exploitant principalement Postfix_fct-like_CE donne :
“(?p And(*hasEmployer*(?p IBM) ?p#Person)) Exists”.
- Un export exploitant principalement Postfix_list-like_CE donne :
“(?p And(*hasEmployer*(?p IBM) ?p#Person) Exists)”.
- Un export exploitant principalement Infix_operator_CE donne :
“?p Exists And(*hasEmployer*(?p IBM) ?p#Person)”.

Exemple 2, ordonnancement de la présentation des parties d'un document

Dans KRLO_2014, un document est un ensemble de relations. Or, dans la notation RIF-PS, les relations d'un document sont présentées dans un ordre particulier qui est décrit par la règle de grammaire suivante. Avec KRLO_2014, cet ordre doit être spécifié pour le type d'EA représentant un document RIF alors que ce n'est pas utile dans KRLO_2015+.

```
//règle EBNF pour la présentation des parties d'un document dans la notation RIF-PS
Document ::= IRIMETA? 'Document' '(' Dialect? Base? Prefix* Import* Module* Group? ')'
```

Exemple 3, filtrage

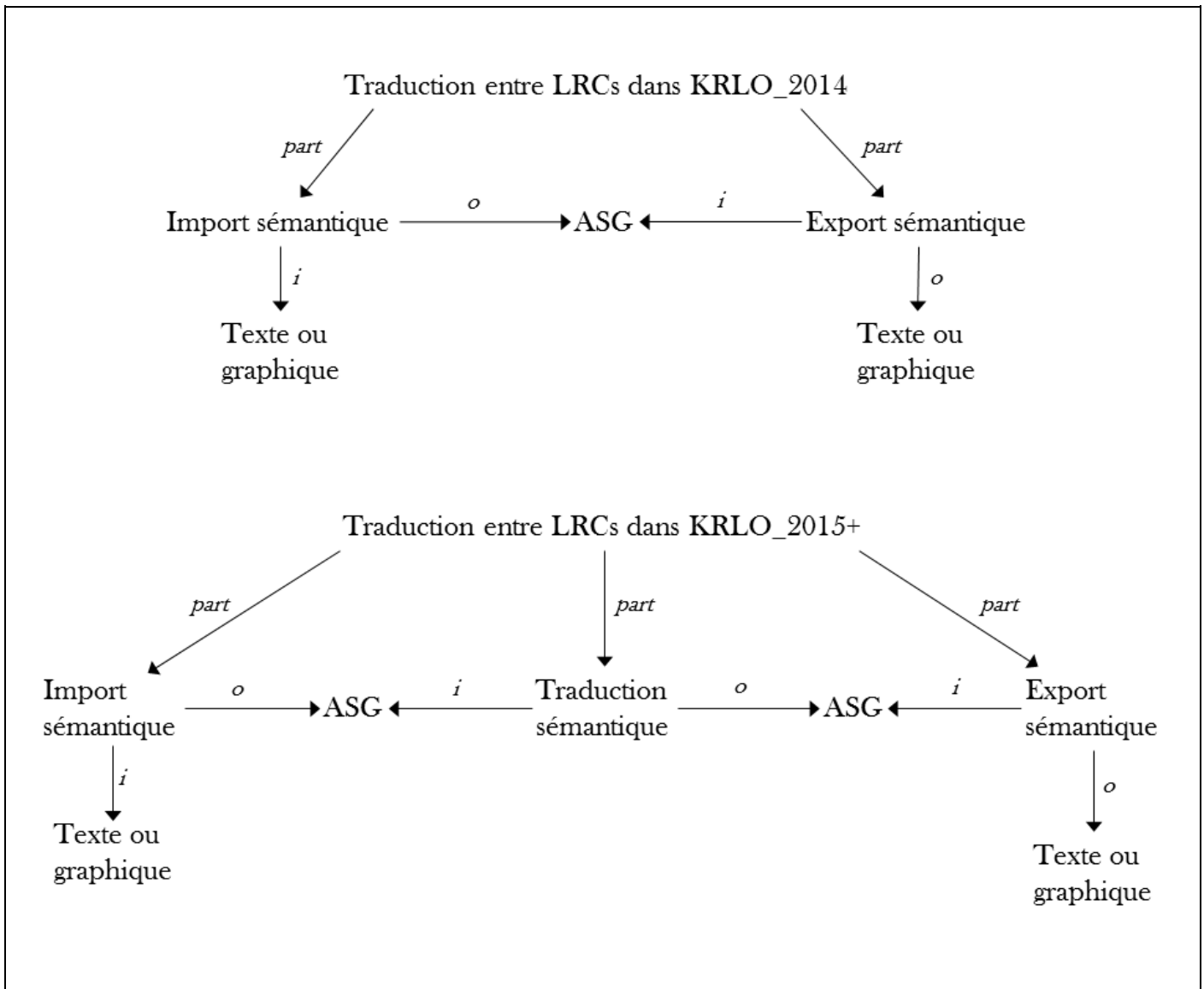
KRLO (et donc les EAs) ne dépendent pas d'une implémentation particulière de BC. Cependant, pendant cette thèse, j'ai dû implémenter une BC via `jb#RichGraph`, une structure de données représentant un graphe qui est présentée en détails dans la section 5.1.1. Pour stocker les EAs de type Quantification, j'ai fait le choix d'implémentation suivant : une instance de Quantification est stockée pour chaque variable quantifiée. Ainsi, via la BC que j'ai implémentée, la phrase "Une entreprise a au moins un employé" est présentée "**Exists ?p (Exists ?ent (And(?p#Person ?ent#Company hasEmployer(?p ?ent))))**" dans la notation RIF-PS. Comme le montre cet exemple, ce choix d'implémentation permet de d'exporter facilement des EAs vers des notations qui autorisent seulement une variable quantifiée par quantificateur. Un filtrage est nécessaire pour une présentation plus compacte telle que "**Exists ?p ?ent (And(?p#Person ?ent#Company hasEmployer(?p ?ent)))**".

2) Absence de traduction entre des EAs

J'ai corrigé le problème d'absence de traduction entre EAs en réalisant ces traductions via du code procédural. KRLO_2015+ apporte une solution plus générique à ce problème via des règles de traduction.

Dans KRLO_2014, la traduction de phrases depuis un LRCs vers un autre se fait via un import puis un export, comme le montre la première partie de la figure ci-dessous. J'ai découvert qu'une étape de traduction entre EAs était parfois nécessaire. Le paragraphe suivant énumère les cas nécessitant une telle traduction. La figure ci-dessous illustre la différence entre les traductions entre LRCs telle que nous les imaginions dans KRLO_2014 et ces traductions telles qu'elles sont spécifiées dans KRLO_2015+.

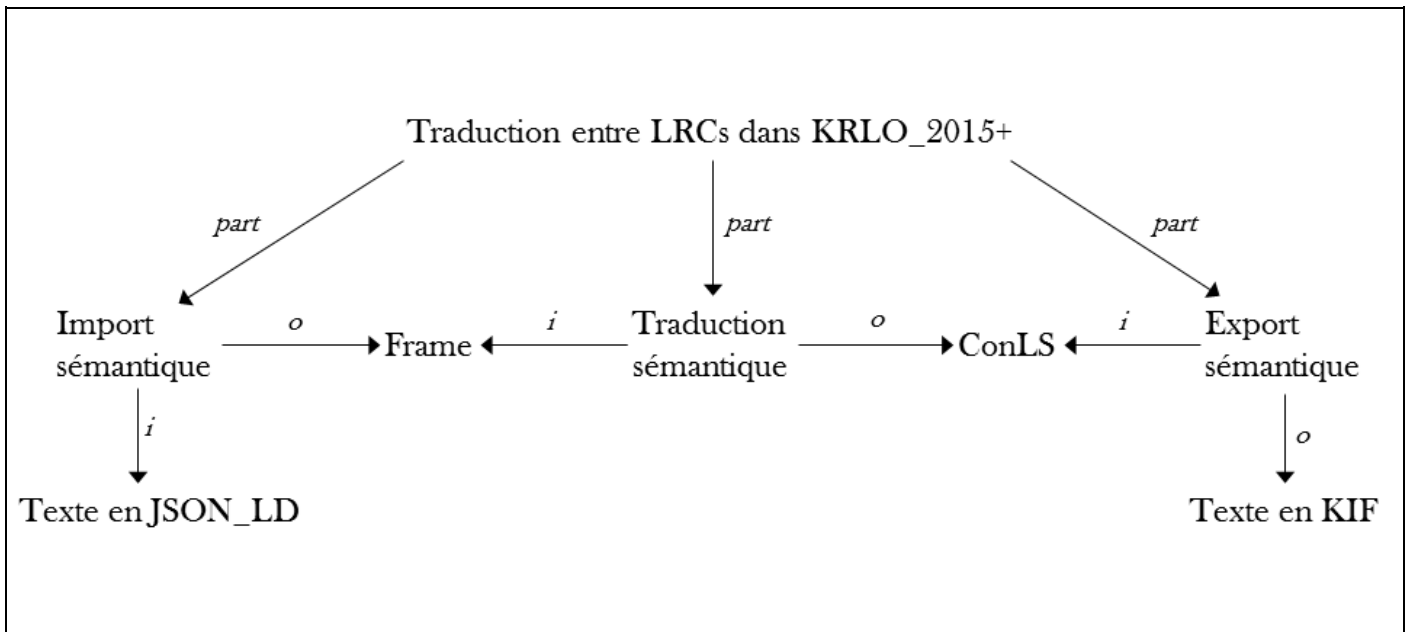
Figure 4.7. Tâches de traduction dans KRLO_2014 et KRLO_2015+ en `pm#UML`.



Après plusieurs tests de traduction entre LRCs sur des RCs types et une analyse des ASTs de ces RCs, j'ai découvert plusieurs cas où une traduction entre EAs est nécessaire. Ces cas sont présentés ci-dessous. Philippe Martin a complété cette liste de cas de façon exhaustive. Cette liste exhaustive est présentée dans la section "6.2.1. Nécessité d'une traduction entre EAs".

- Le modèle abstrait de la phrase en entrée inclut des EAs de type Frame et le modèle abstrait de la phrase en sortie inclut des EAs de type Conjunction_of_links_from_a_same_source et vice versa. La figure suivante illustre ce cas.
- Le modèle abstrait de la phrase en entrée inclut des EAs de type Half-link et le modèle abstrait de la phrase en sortie inclut des EAs de type NR_Link et vice versa. Ce cas est une partie du cas précédent puisque les EAs de type Half-link sont des parties d'EAs de type Frame.
- Le modèle abstrait de la phrase entrée inclut des EAs de type Link et le modèle abstrait de la phrase en sortie inclut des EAs de type Non-binary_relation et vice versa.
- Le modèle abstrait de la phrase entrée inclut des EAs de type Function et le modèle abstrait de la phrase en sortie inclut des EAs de type Functional_relation et vice versa.

Figure 4.8. Traduction depuis une Frame vers une conjonction de liens depuis une même source dans KRLO_2015+ en pm#UML.



La section “6.2.1. Nécessité d’une traduction entre EAs” donne plus de détails sur la nécessité de traduire des EAs d’un modèle vers un autre.

3) Language_element = sbvr:Expression

Dans KRLO_2014, sbvr:Expression et Language_element sont équivalents. Or, en vérité, ces concepts ne sont pas équivalents. En effet, Language_element réfère à des EAs et des ECs et, comme expliqué dans le paragraphe ci-dessous, sbvr:Expression ne réfère qu’à des ECs. Donc, pour une phrase donnée, un analyseur [parser] exploitant SBVR plutôt que KRLO ne peut extraire qu’une unique structure pour cette phrase et cette structure est un CST. Dans la correction que j’ai apportée, sbvr:Expression est équivalent à Concrete_element.

Dans la documentation de SBVR, sbvr:Expression a la définition suivante : “ something that expresses or communicates, but considered independently of its interpretation”. Cette documentation fournit également les exemples suivants : “the sequence of characters "car"”, “the entire text of a book”, “a diagram”. Les seules spécialisations de sbvr:Expression fournies sont sbvr:Text – toute chaîne de caractères –, sbvr:URI – un sous-type de sbvr:Text – et sbvr:Signifier – un sous-type de sbvr:Expression utilisé pour représenter un concept. Entre ces quatre types (sbvr:Expression, sbvr:Text, sbvr:URI, sbvr:Signifier), seules des relations de spécialisation sont écrites dans la documentation. Ainsi, un analyseur [parser] exploitant SBVR pour importer des phrases écrites dans un LRC ne peut extraire que des éléments de ces quatre types et des relations structurelles "r_part" entre ces éléments. De plus, ces éléments sont des éléments de notation.

4) Export incorrect vers JSON-LD+OWL/JSON-LD_notation

Une proposition de Philippe Martin pour les spécifications de présentation des EAs de JSON-LD+OWL/JSON-LD_notation était incorrecte. En exploitant ces spécifications, une fonction d’export avec en entrée la phrase “Toute entreprise a au moins un employé” a la sortie ci-après.

```
{ "@id": "Company";
  "owl:subClassOf": { "@type": "owl:Restriction",
    "hasEmployee": { "@type": "Person" } }
}
```

Cette notation ne peut pas être interprétée par un importeur OWL. Une notation correcte est présentée ci-dessous.

```
{ "@id": "Company",
  "owl:subClassOf": { "@type": "owl:Restriction",
    "owl:onProperty": "hasEmployee",
    "owl:someValuesFrom": "Person" } }
```

5) Nommage ambiguë d'une fonction

Dans KRLO_2014, une fonction “partition_except_for_(Thing)” est utilisée pour spécifier une partition entre Concept_type et Relation_type à l'exception du type Thing. J'ai fait remarquer à Philippe Martin que l'utilisation du type Thing comme paramètre de cette fonction pouvait porter à confusion. En effet, un utilisateur pourrait croire que la sémantique de cet appel serait d'exclure toute instance du type Thing et donc toute l'ontologie.

Dans KRLO_2015+, “partition_except_for_(Thing)” n'est plus utilisé, un commentaire indique le concept Thing est une exception dans toutes les partitions.

4.2.2. Spécifications de notations des éléments de modèle abstrait dans KRLO_2014

Dans KRLO_2014, j'ai spécifié des représentations concrètes d'EAs dans les notations CLIF, RIF-PS et dans la notation de JSON-LD. Ces descriptions sont données ci-dessous.

```
JSON-LD_model r_only_such_part_of_that_type:
  ^(Phrase rc_type: fc_infix_list-like_frame_type _(List(JSON-LD_notation),"","{",",","\n","}"))
  ^(Link rc_type: fc_half-link_type _(List(JSON-LD_notation),"",": ","",","))
  ^(Fterm_or_variable > Constant_or_set_or_closed_list)
  ^(Set rc_type: fc_list_type _(List(JSON-LD_notation),"["",",","]")) //by default in JSON-LD
  //(but not in JSON)
  ^(Constant_predefined_in_a_KRL > r_header r_name r_base r_language r_type r_value
    r_container r_list r_set r_graph
    r_inverse r_index r_vocab,
    rc_type: fc_string_type _(List(JSON-LD_notation),"@",",",'"') )
  ^(Concrete-term_for_constant_or_name
    rc_type: ^(string
      rc_notation_type: .{JSON-LD_notation},
      rc_parts_begin-mark: '"',
      rc_parts_separator: ":",
      rc_parts_end-mark: '"'));
```

```

CL r_only_such_part_of_that_type:
^(Quantification
  rc_type: (fc_list-like_prefix-fct_type _(List(CLIF),"","(", " ",")"))
    rc_quantified_variable_begin_mark: "(",
    rc_quantified_variable_end_mark: ")",
    rc_separator: " ")
^(Logical_equivalence rc_type: (fc_list-like_prefix-fct_type _(List(CLIF),"iff","(", " ",")"))
  rc_separator: " ")
^(Logical_implication rc_type: (fc_list-like_prefix-fct_type _(List(CLIF),"if","(", " ",")"))
  rc_separator: " ")
^(Atomic_formula
  > (cl#Atomic_sentence < Positional-or-name-based_formula),
  rc_type: (fc_list-like_prefix-fct_type _(List(CLIF),"","(", " ",")"))
    rc_separator: " ")
^(Class-membership_formula rc_type: fc_list_type _(List(CLIF),"(", " ",")"))
^(Frame = Frame_as_conjunction_of_links_from_a_same_source)
^(Gterm_reference > Constant_gTerm)
^(Constant_concrete_term > (KIF#Name > Numeral Single-or-double_quoted_string));

RIF+OWL r_only_such_part_of_that_type:
^(RIF_annotation rc_type: fc_list_type _(List(RIF-PS),>(*,"",*))")
^(Conjunction_phrase rc_type: (fc_prefix-fct-like_type _(List(RIF-PS),"","(", " ",")"))
  rc_separator: " ")
^(Quantification > Classic_quantification,
  rc_type: (fc_prefix-fct-like_type _(List(RIF-PS),"","(", " ",")"))
    rc_separator: " ")
^(Rule rc_type: (fc_list-like_infix-fct_type _(List(RIF-PS),":-","",",",""))
  rc_separator: " ") // FORMULA ':'- FORMULA
^(Logical_equivalence rc_type: fc_prefix-fct-like_type _(List(RIF-PS),
  "owl:equivalentClassOf","(", " ",")"))
^(Equality_formula rc_type: fc_list-like_infix-fct_type _(List(RIF-PS),"=","",",",""))
^(Class-membership_formula rc_type: (fc_list-like_infix-fct_type _(List(RIF-PS),"#",",",""))
  rc_separator: " ")
^(Subclass_formula rc_type: fc_list-like_infix-fct_type _(List(RIF-PS),"##","",",",""))
^(Half_link rc_type: (fc_half-link_type _(List(RIF-PS),"","->","",",")))
^(Collection > List);

```

4.2.3. Spécifications de notations des éléments de modèle abstrait dans KRLO_2015+

Dans KRLO_2015+, j'ai décrit des spécifications de notations d'EAs du modèle JSON-LD pour la notation de JSON-LD et des spécifications de notations d'EAs du modèle KIF pour la notation de KIF. Les extraits ci-dessous présentent ces spécifications.

```

// 1. concrete specifications for KIF abstract elements in KIF notation
Abstract_element@KIF_model ?a
  rc_spec: fc_spec _( List( "(" f_OP_from_(?a) f_ARGS_from_(?a) " ) " ),
    List( KIF_notation ));

// 2. concrete specifications for JSON-LD abstract elements in JSON-LD notation
^(Conjunction_phrase@JSON-LD_model not < NR_frame) ?cp
  rc_spec: fc_spec _( List( "[" f_ARGS_from_(?cp,"") //"," is the argument separator
    "]" ),
    List( JSON-LD_notation ));

Minimal-frame_as_NR-phrase@JSON-LD_model ?mf
  rc_spec: fc_spec _( List( "{" f_ARGS_from_(?mf,"") "}" ),
    List( JSON-LD_notation ));

Minimal_half-link@JSON-LD_model ?mh
  rc_spec: fc_spec _( List( f_OP_from_(?mh) ":" f_ARG_(?mh.r_link_destination) ),
    List( JSON-LD_notation ));

```

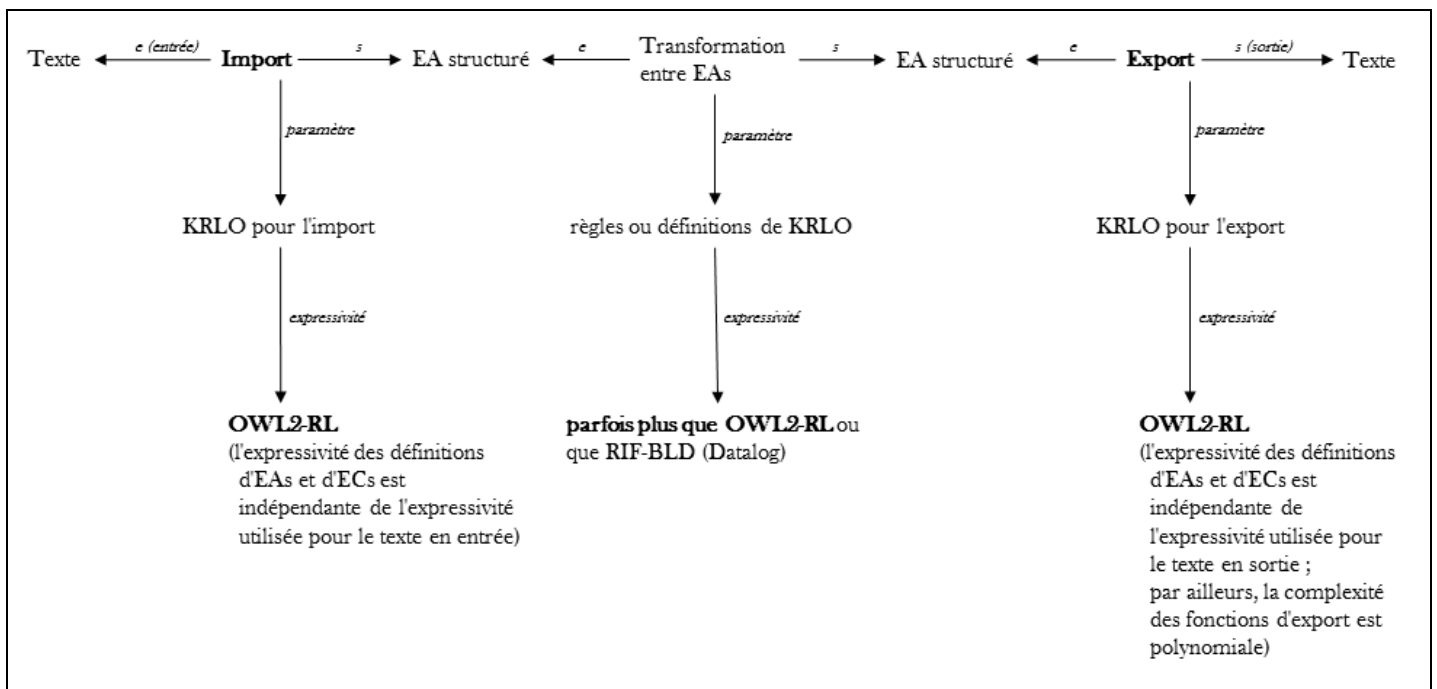
4.3. Expressivité

Cette section fournit des détails sur l'expressivité requise pour décrire KRLO. La spécification de règles pour la traduction d'EAs nécessite une expressivité supérieure à RIF-BLD. Cependant, l'expressivité de RIF-BLD est suffisante pour décrire les éléments de LRCs (c'est à dire, les ECs, les EAs, les modèles et les notations) et pour que ces descriptions puissent être exploitées par ces règles.

KRLO est un modèle conceptuel, ce qui facilite le partage et la réutilisation des descriptions. Ainsi, l'expressivité des descriptions dans KRLO n'est pas limitée à l'expressivité d'un LRC exécutable. La section "2.2.2. Création directe d'un modèle exécutable via un LRC exécutable" fournit des détails sur les difficultés résultant de l'utilisation d'un modèle exécutable pour le partage et la réutilisation des connaissances. Une description de KRLO en FL est accessible depuis cette page : http://www.webkb.org/kb/it/o_knowledge/o_KRL/o_knowledge/o_KRL/. FL est un LRC avec une notation de haut niveau et de second ordre (voir la définition de notation de 2nd-ordre). Pour décrire la version actuelle de KRLO (KRLO_2015+), une notation de premier ordre suffit.

Le schéma ci-dessous présente les différents niveaux d'expressivités requis pour l'import, l'export et les transformations entre EAs.

Figure 4.9. Expressivité nécessaire pour importer, exporter et transformer des EAs.



L'expressivité de certaines descriptions dans KRLO peut dépasser celle de OWL2-RL. Par exemple, les types d'EAs sont très souvent organisés dans des partitions, c'est à dire, des unions disjointes. Dans ce cas, pour les outils d'import ou d'export exploitant KRLO, cette précision peut être ignorée sans risque d'erreur pour l'import ou l'export de connaissances. Par exemple, les partitions peuvent être traitées comme des disjonctions. Ainsi, pour améliorer les performances lors de l'import ou de l'export de connaissances, quel que soit le LRC utilisé pour l'entrée ou la sortie, KRLO pourrait être traduite en OWL2-RL. En effet, l'expressivité de OWL2-RL est suffisante pour relier les types d'éléments de LRCs par des relations *subtype* et définir des relations structurelles pour les instances d'un type. Dans KRLO, ces relations sont *r_result*, *r_args* et *r_operator*. Pour décrire ces relations, seuls les *constructeurs de logiques de descriptions* suivants sont nécessaires :

- sous-typage et équivalence de types,
- intersection et union de types,
- cardinalité maximale 0/1.

Pour éviter d'utiliser les cardinalités 0..*, des listes de types particuliers sont nécessaires. À titre d'exemple, une liste de Link est définie ci-après en OWL2-RL/OWL_Functional-style.

```
owl2:EquivalentClasses( jb:Link_list
                        owl:ObjectIntersectionOf( jb:List
                                                    owl:ObjectAllValuesFrom( jb:r_member
                                                                                krlo:Link) )
```

OWL2-RL est un sous-langage de OWL2 qui peut être entièrement défini en RIF-BLD [OWL2 in RIF, 2013]. OWL2-RL peut être implémenté dans un raisonneur basé sur des règles capable de gérer l'expressivité de RIF-BLD, c'est à dire, l'expressivité de règles de Horn strictes [definite Horn rules] avec l'égalité et une sémantique du premier ordre standard. Plus précisément, OWL2 et RIF sont interoperables uniquement pour les sous-langages OWL2-RL et RIF-BLD. En effet, pour RIF-BLD, la "sémantique OWL2 directe" [OWL2 direct semantics, 2012] et la "sémantique basé sur RDF" [OWL2 RDF-Based semantics, 2012] peuvent être utilisées comme détaillé dans [OWL2-RL to RIF-BLD, 2013]. Lorsque KRLO est exploitée pour l'import ou l'export de connaissances, KRLO ne nécessite pas davantage d'expressivité que OWL2-RL et RIF-BLD. Ainsi, KRLO i) permet de réutiliser les graphes RDF, ii) peut être traduite dans des LRCs moins expressifs mais interprétables par beaucoup de raisonneurs sans conséquence pour l'import ou l'export, et iii) permet le passage à l'échelle des mécanismes d'inférences [OWL2, 2012] [Krötzsch et al., 2013].

La traduction entre EAs nécessite une expressivité supérieure à RIF-BLD. En effet, certaines des règles de traduction spécifiées dans KRLO nécessitent une variable existentiellement quantifiée. La section "6.2. Traduction entre éléments abstraits" donne des détails sur les règles de traduction spécifiées dans KRLO.

La traduction entre LRCs présentée dans cette thèse conserve en sortie toutes les informations en entrée. Lorsque le modèle source a une expressivité supérieure à celle du modèle cible, ce dernier est complété via des relations spéciales du modèle IKLmE – une extension du modèle d'IKL qui définit ces relations spéciales [Martin & Bénard, 2017b]. Ces relations permettent de représenter des EAs de la phrase source. Dans l'exemple ci-dessous, la relation spéciale *IKLmE:_100pc* est utilisée pour représenter un quantificateur universel. Cet exemple montre que l'usage des relations spéciales de IKLmE permet des représentations concises et lisibles. Les relations spéciales d'IKLmE sont binaires ; elles peuvent donc être exportées sans transformation vers toute notation supportant des relations binaires et donc vers toute notation.

Exemples de représentation d'une quantification universelle dans différents langages

Français : 100% des voitures de John sont blanches.

RDF+OWL/Turtle.

La représentation de la phrase en français ci-dessus est impossible car 100% est ici une observation, pas une définition (cf. Définition d'un terme T). En revanche, la phrase "Par définition du terme 'voitureDeJohn' une telle voiture est blanche" pourrait être représentée en RDF+OWL/Turtle.

RDF+Log/N3.

Log [Berners-Lee et al., 2005] est une ontologie de termes représentant des opérateurs et des quantificateurs logiques conçue par Tim Berners Lee. La notation N3 fournit une présentation concrète pour le quantificateur "100%", ainsi, en RDF+Log/N3, la phrase en français ci-dessus peut s'écrire de la façon suivante.

@forAll <#v>.

```
{v rdf:instanceOf :Voiture; :owner :John} log:implies {v :couleur [a :Blanc]}.
```

RDF+drsOnto/Turtle.

drsOnto [McDermott & Dou, 2002] est une ontologie représentant des quantificateurs et des variables. Ainsi, bien que la notation Turtle ne fournisse pas de présentation concrète pour le quantificateur "100%", la phrase en français ci-dessus peut être représentée en RDF+drsOnto/Turtle de la façon suivante.

- (1) [a drsOnto:Forall; drsOnto:quantifier_vars [a drsOnto:variables_list; rdf:_1 :var_1];
- (2) drsOnto:body [rdf:type drsOnto:Atomic_formula;
- (3) rdf:subject :var_1;
- (4) rdf:predicate :ChoseDeCouleurBlanche] .
- (5) :var_1 a drsOnto:Var; drsOnto:type :VoitureDeJohn.

drsOnto ne permet cependant pas de représentations concises car elle impose à l'utilisateur la tâche de spécifier des variables – comme à la ligne (5) ci-dessus – et des types ad hoc – comme ci-dessus :ChoseDeCouleurBlanche à la ligne (4) et :VoitureDeJohn à la ligne (5).

RDF+IKLmE/Turtle.

Le modèle IKLmE fournit le type de relation (spéciale) *_100pc* pour représenter le quantificateur "100%". Ainsi, la phrase en français ci-dessus peut être représentée en RDF+IKLmE/Turtle de la façon suivante.

```
[ IKLmE:_100pc [a :Voiture; :owner :John]; :couleur [a :Blanc] ]
```

4.4. Complétude de KRLO

Cette section (4.4) présente des protocoles qui permettent de déterminer la complétude de KRLO. Les sections [6.1.3](#) et [6.3.2](#) donnent des détails sur la complétude des fonctions d'export et d'import, respectivement.

KRLO est complète si et seulement si *tous* les EAs de tous les modèles abstraits représentés dans KRLO sont correctement, et donc *complètement*, définis. La complétude de KRLO est induite par les protocoles suivants, en cours d'utilisation dans KRLO.

1. Les types de concepts nécessaires pour représenter tous les EAs de chaque modèle sont systématiquement organisés dans KRLO via des partitions. Par exemple, la partition la plus générale pour les phrases est NR_phrase (non-reference_phrase) et Reference_to_phrase. Cette catégorisation systématique permet de détecter et de représenter tous les types de concepts généraux peu intuitifs et donc d'assurer la représentation de types d'EAs potentiellement utiles (pour d'autres LRCs) même si les LRCs représentés jusqu'alors n'en dépendent pas.
2. La traduction de RCs depuis un LRC de KRLO vers tout autre LRC de KRLO suivie de la traduction inverse des résultats doit générer le contenu textuel original des RCs, sauf pour les espaces blancs optionnels.
3. Le point 2 ci-dessus est appliqué aux RCs issues d'ontologies populaires ou faisant autorité – par exemple, DBpedia (dbpedia.org) ou les ontologies stockées sur Ontohub [Codescu et al., 2016] (ontohub.org).
4. Le point 2 ci-dessus est appliqué aux RCs issues de KRLO.

5. Résolveur de requêtes développé pendant cette thèse

5.1. Raisons pour la création de ce résolveur et comparaisons avec d'autres résolveurs

5.1.1. But et comparaisons

GTH (Global Technologies Holding) est le partenaire industriel de cette thèse (cf. section 1.1. "Préambule"). Un des objectifs de GTH est de mettre en production l'interpréteur "BaboukWeb". Cet interpréteur fait partie du LAS, le serveur d'application développé par logiCells (cf. section 1.1. "Préambule"). La version 2 de "BaboukWeb" est encore en cours de développement. Dans cette version, "BaboukWeb" pourra interpréter des descriptions fonctionnelles, des RCs et même accepter que ces descriptions soient mélangées. Plus de détails sur les langages qui seront interprétés sont donnés dans la section "But" ci-dessous. Une première version stable (1.0) de cet interpréteur est utilisée par logiCells. Une seconde version (2.0) est aujourd'hui en cours de développement. Cette section (5.1.1.) i) décrit cette version 2.0 de l'interpréteur BaboukWeb, ii) situe Structure_map Request Solver (SRS) – le résolveur de requêtes que j'ai développé pendant cette thèse – par rapport à la version 2 de l'interpréteur BaboukWeb, iii) présente les objectifs à atteindre pour SRS, et iv) offre des comparaisons entre les requêtes que SRS doit être capable de résoudre et des requêtes SPARQL. Les parties "[5.1.2. Réutilisation de travaux précédant le changement de sujet de thèse](#)" et "[5.2. Mon implémentation de Structure_map](#)" présentent les caractéristiques de ce résolveur. La section "[5.3.1. État de l'implémentation](#)" présente les objectifs qui ont été réalisés. La section "[5.3.2. Perspectives pour SRS](#)" présente une évolution possible pour ce résolveur.

But

À terme, la version 2 de l'interpréteur BaboukWeb doit pouvoir interpréter des descriptions fonctionnelles, des RCs et même accepter que ces descriptions soient mélangées. Les descriptions fonctionnelles doivent être écrites dans le langage FunctionalP – un langage de scripts développé par GTH. Les RCs peuvent être écrites dans les LRCs suivants.

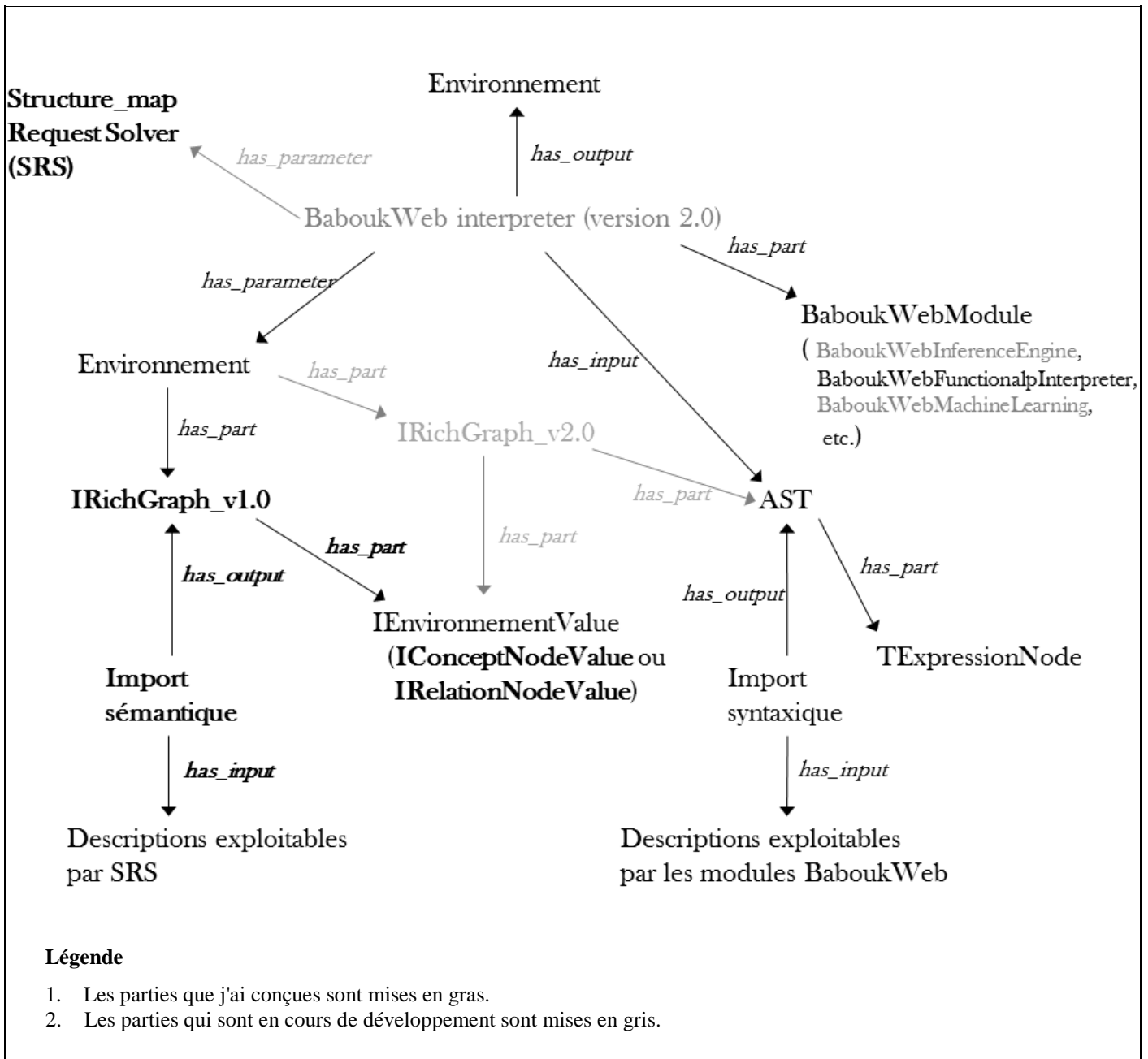
- Le modèle OWL1-Full avec la notation RDF/XML
- Un LRC développé par GTH qui a i) une notation basée sur XML, ii) un modèle basé sur DIG 1.1 [Bechhofer et al., 2003] dont l'expressivité est SHOIQD_n
- FL

Une description écrite en FunctionalP peut inclure des RCs dans la notation FL, comme illustré dans l'exemple ci-dessous.

```
//En functionalP, ?variables [...] délimite une description de requête en FL
//variables et [ ] sont optionnels,
//variables peut désigner plusieurs variables séparées par une virgule
(?p [1 Person ?p hasEmployer: IBM]).ShowObjectItemView();
```

La version 2 de l'interpréteur BaboukWeb, ses composants, ses entrées et sorties sont décrits ci-après. Tout ces éléments sont décrits après la figure suivante.

Figure 5.1. Interpréteur BaboukWeb en pm#UML.



Remarque : Delphi ne propose pas de ramasse-miettes automatique mais propose un ramasse-miettes manuel via un compteur de références sur les objets qui sont instances de classes implémentant une interface. Pour faciliter la libération de mémoire, les classes du Framework développées par le groupe GTH implémentent toutes une interface de libération de mémoire. Par convention, le nom d'une interface dans Delphi commence par un "I" et le nom des classes commence par un "T".

RichGraph est une structure de données représentant un graphe et développée sous Delphi. Anil Cassam-Chenaï a conçu la version 0.1 de gth#RichGraph. J'ai ensuite stabilisé puis étendu cette version jusqu'à la version 1.0 en collaboration avec Anil Cassam-Chenaï et des employés de logiCells. Le rôle de ces employés a essentiellement été de trouver des bugs. Anil Cassam-Chenaï et moi faisons partie de GTH. gth#RichGraph en version 1.0 a été utilisé par une équipe de logiCells chargée de développer des applications pour des clients. Alors, afin que i) mes travaux ne perturbent pas cette équipe, et ii) les travaux d'Anil Cassam-Chenaï ne perturbent pas les miens, j'ai créé une fourche [fork] : jb#RichGraph fin 2014. J'ai étendu cette fourche jusqu'à la version 1.0 pour y inclure notamment une fonctionnalité permettant de gérer des méta-phrases contextualisantes. Jusqu'à la fin de ma 3eme année de thèse, LAS pouvait encore fonctionner en utilisant jb#RichGraph, ce n'est plus le cas depuis janvier 2016. Les versions post 1.0 de gth#RichGraph ont des fonctionnalités de persistance, d'optimisation mémoire (typiquement via le motif de conception [design pattern] proxy) que jb#RichGraph n'a pas. De plus, SRS est compatible avec jb#RichGraph et gth#RichGraph en version 1.0 mais n'est pas compatible avec les versions plus récentes de gth#RichGraph. Avec un peu de travail de stabilisation, SRS pourrait être rendu compatible avec ces versions.

L'interpréteur BaboukWeb et ses modules exploitent un "environnement", c'est à dire, un ensemble de paires clef-valeurs utilisé pour associer des variables et des valeurs. Dans LAS, ces valeurs implémentent l'interface IListView_. Quelle que soit la version de RichGraph, un RichGraph et les noeuds qui le composent implémentent également cette interface. Ils peuvent donc être manipulés par l'interpréteur BaboukWeb et ses modules.

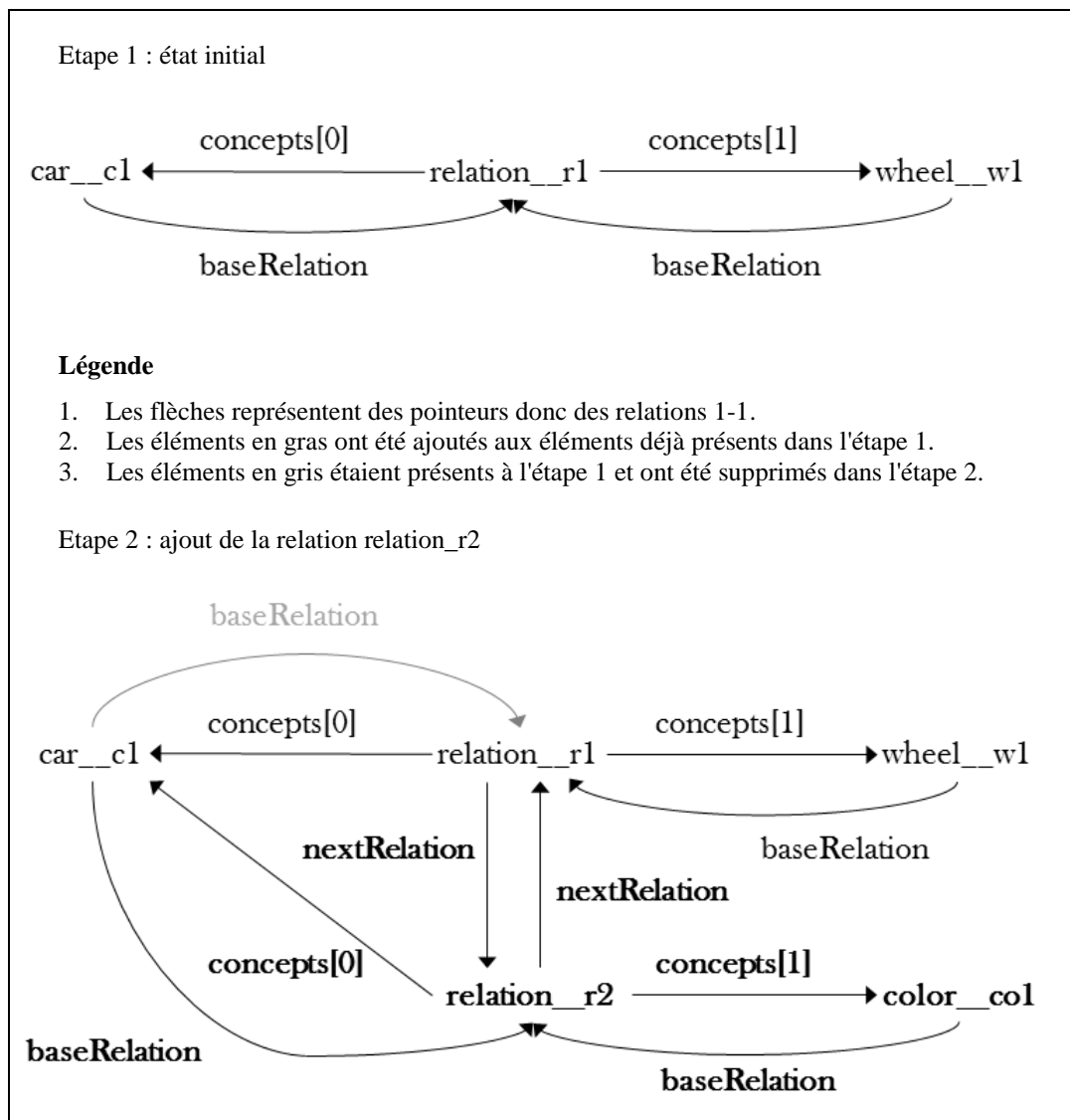
L'interface IListView_ est spécialisée entre autres par l'interface IEnvironmentValue. IEnvironmentValue est spécialisée entre autres par les interfaces IConceptNodeValue et IRelationNodeValue. IConceptNodeValue et IRelationNodeValue représentent respectivement un noeud concept et un noeud relation. L'extrait ci-dessous présente les spécifications de IEnvironmentValue, IRelationNodeValue et IConceptNodeValue.

```
IEnvironmentValue = public interface(IListView_) //IEnvironmentValue hérite de IListView_
// Les deux procédures ci-dessous sont utilisées pour la persistance
procedure OutCopyBytes(Buffer:TBytes; Position,Length:integer; aDbType:DbType; WithSizeInfo:boolean);
procedure InCopyBytes(Buffer:TBytes; Position,Length:integer; aDbType:DbType; WithSizeInfo:boolean);
function GetNodeName:String;
//Tout noeud (concept ou relation) peut stocker des quantificateurs
function GetQuantifiers: IListView_;
procedure SetQuantifiers(const Value: IListView_);
//Les relations depuis ou vers un même concept sont stockées dans une liste chaînée.
//BaseRelation est la relation qui est stockée dans le premier élément de la liste.
function GetBaseRelation:IRelationNodeValue;
procedure SetBaseRelation(const Value:IRelationNodeValue);
//GetNextOutRelation retourne la relation suivante et dont la source est le noeud courant.
function GetNextOutRelation(const RelationNode: IRelationNodeValue):IRelationNodeValue;
//GetNextInRelation retourne la relation suivante et dont la destination est le noeud courant.
function GetNextInRelation(const RelationNode: IRelationNodeValue):IRelationNodeValue;
//GetNextInOutRelation retourne la relation suivante.
function GetNextInOutRelation(const RelationNode: IRelationNodeValue):IRelationNodeValue;
end;

IRelationNodeValue = public interface(IEnvironmentValue) //relation binaire
//GetConcepts(i) renvoie le i-ème concept de la relation,
//par convention 0 est la source et 1 est la destination.
function GetConcepts(const i:integer):IEnvironmentValue;
procedure AppendConcept(const Node:IEnvironmentValue);
//Associe un quantificateur aux noeuds concepts source ou destination auquel il se rapporte
procedure AppendQuantifier(const Quantifier: IQuantifier);
end;

//Des fonctions telles que AppendQuantifier pourraient être implémentées ici.
//Ainsi, le modèle peut être relationnel ou bien basé sur des frames.
IConceptNodeValue = interface(IEnvironmentValue) end;
```

Les schémas ci-après illustrent la façon dont se construit la structure de données RichGraph à chaque ajout de relation.



BaboukWebMachineLearning, BaboukWebFunctionalInterpreter et BaboukWebInferenceEngine sont des modules de l'interpréteur BaboukWeb. Dans ces modules, des spécialisations de TExpressionNode sont déclarées et implémentées. Les classes héritant de TExpressionNode modularisent des traitements sur des informations. Par exemple, TIntersectExpression est une spécialisation de TExpressionNode pour traiter les conjonctions de concepts selon la méthode des tableaux, TIntersectExpression est déclarée et implémentée dans BaboukWebInferenceEngine ; TForEachExpression est une spécialisation de TExpressionNode pour traiter les boucles “foreach” dans le langage FunctionalP, TForEachExpression est déclarée et implémentée dans BaboukWebFunctionalInterpreter.

Chaque module de l'interpréteur BaboukWeb peut être utilisé pour importer puis interpréter des phrases d'un langage. Une catégorisation des langages pouvant être interprétés via l'un des modules de BaboukWeb est donnée ci-dessous.

- Langages de programmation
 - “FunctionalP” un langage de scripts fonctionnel basé sur le Lambda-calcul
 - une extension de Functional P basé sur les principes des algèbres de processus [process calculus]
- LRCs
 - OWL1-Full avec la notation RDF/XML
 - DIG 1.1 avec une notation XML
- Langages déclaratifs
 - une partie de Business Process Model and Notation (BPMN) [BPMN, 2011]. BPMN est une méthode de modélisation de processus métier – développée par l'OMG – pour décrire les flux d'activités et les procédures d'une organisation sous forme d'une représentation graphique standardisée.

Anil Cassam-Chenai conçoit et implémente actuellement (en 2017) la version 2 de RichGraph. Une évolution majeure dans la version 2 est qu'un IEnvironmentValue peut maintenant stocker une expression logique ou fonctionnelle (TExpressionNode) qui peut être résolue par l'interpréteur BaboukWeb 2.0. Ce dernier est capable de sélectionner le module à utiliser pour résoudre cette expression. L'annexe 4 fournit une documentation de la version 2 de RichGraph.

Une future version de LAS mettra à disposition de ses clients des services exploitant, entre autres, un résolveur de requêtes. L'entreprise GTH a décidé de développer entièrement un nouveau résolveur de requêtes plutôt que d'en utiliser un existant déjà.

Ce choix est motivé par le fait qu'un résolveur entièrement redéveloppé peut être plus facilement étendu par des employés du groupe GTH. C'est pourquoi, pendant cette thèse, j'ai développé Structure_map Request Solveur (SRS), le premier résolveur de requêtes utilisable par l'interpréteur BaboukWeb 1.0. Les contraintes que ce résolveur doit satisfaire sont listées ci-dessous.

1. SRS doit pouvoir et peut effectivement être facilement :
 - utilisé par la version 1 de l'interpréteur BaboukWeb, et
 - perfectionné ou adapté par les employés du groupe GTH.
2. SRS doit être capable de résoudre des requêtes complexes comme celles qui spécifient l'export par défaut dans KRLO. Ceci implique que SRS doit être capable i) de résoudre des fonctions récursives, ii) de gérer des méta-phrases contextualisantes ou au minimum d'utiliser une fonction spéciale telle que `if_then_else`.

Comparaisons

Quelques exemples types de requêtes que SRS est capable de résoudre sont présentés dans la table ci-après. Ces requêtes sont d'abord écrites en SPARQL puis en FC [Martin & Eklund., 1999a]. Dans sa dernière version, FC permet à l'utilisateur de combiner des commandes via des structures de contrôle semblables à celles des scripts shell. FC permet également l'utilisation d'un opérateur de requête générique, noté "?", qui généralise les opérateurs SELECT, CONSTRUCT et ASK en SPARQL. En effet, l'opérateur "?" peut prendre des paramètres. Quelques exemples de requêtes paramétrées sont présentés ci-après. Tous les opérateurs de requêtes de FC sont décrits précisément sur la page web accessible sur l'URL suivante http://www.phmartin.info/cours/ws/km_tr.html#%2841%29. De plus, en FC, différents langages peuvent être utilisés pour exprimer le graphe requête. Enfin, le graphe requête peut avoir une expressivité plus grande que dans SPARQL, par exemple, le graphe requête permet lui même de spécifier s'il s'agit d'une recherche par généralisation ou, par défaut, par spécialisation. Pour SRS, j'ai développé – et je développe encore – un analyseur lexico-syntaxique pour permettre l'interprétation d'une partie de FC. Dans cette partie de FC, le graphe requête doit être décrit en FL et les commandes ne peuvent pas être combinées. Un analyseur lexico-syntaxique pour permettre l'interprétation des requêtes écrites en SPARQL pourrait être développé par logiCells en 2017. Pour chaque requête de la table ci-après, un exemple de résultat renvoyé par SRS ou par un interpréteur de SPARQL est fourni.

Exemples types de requêtes et résultats

“dolce” réfère à l'ontologie <http://www.loa-cnr.it/ontologies/DOLCE-Lite.owl#>.

“jb” est l'auteur de l'ontologie interrogée.

“rdf” réfère à l'ontologie <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

“\$”, comme sous le shell unix, préfixe chaque ligne de requête.

L'ontologie sur laquelle les requêtes sont envoyées contient les informations décrites ci-dessous en FL. Tous les types sont distincts. Tous les individus sont distincts.

```
jb#Cat
> (jb#Cat_with_owner_Schrodinger
  rdf#type of: (jb#Felix dolce#part: jb#Felix_head) )
rdf#type of: (jb#Tom // Tom has only 3 paw
  dolce#part: jb#Tom_paw1 jb#Tom_paw2 jb#Tom_paw3 jb#Tom_head);

jb#Paw
rdf#type of: jb#Tom_paw1 jb#Tom_paw2
  (jb#Tom_paw3 dolce#part: (jb#Tom_claw rdf#type: jb#Claw));
jb#Head rdf#type of: jb#Felix_head jb#Tom_head;

//La fonction f_chain_of_part définie suivantes est utilisée dans une des requêtes ci-dessous.
//Un utilisateur peut écrire cette fonction afin de l'utiliser dans une requête.
f_chain_of_part
input: dolce#Particular ?t
  dolce#Particular ?direct_or_indirect_part_of_t,
output: pm#boolean,
:= [if_then_else_( [?t dolce#part: ?direct_or_indirect_part_of_t],
  pm#true,
  if_then_else( [?t dolce#part: ?part],
    f_chain_of_part_(?part, ?direct_or_indirect_part_of_t)
    pm#false )
  )];
```

SPARQL :

```
$ ASK { ?cat dolce:part [ a jb:Paw];
$      rdf:type jb#Cat. }
true
```

FC :

```
$ ?? [some jb#Cat dolce#part: some jb#Paw]
true
```

SPARQL :

```
$ SELECT ?p WHERE { jb:Cat dolce:part ?p . ?p rdf:type jb:Paw }
jb:Tom_paw1 jb:Tom_paw2 jb:Tom_paw3
```

FC :

```
$ ?p [some jb#Cat dolce#part: some jb#Paw ?p]
jb#Tom_paw1 jb#Tom_paw2 jb#Tom_paw3
```

SPARQL :

```
$ CONSTRUCT { ?cat dolce:part ?p } WHERE
$ { ?cat dolce:part ?p ; rdf:type jb#Cat . ?p rdf:type jb#Paw . }
jb:Tom dolce:part jb:Tom_paw1 .
jb:Tom dolce:part jb:Tom_paw2 .
jb:Tom dolce:part jb:Tom_paw3 .
```

FC :

```
$ ? [some jb#Cat dolce#part: some jb#Paw]
jb#Tom dolce#part: jb#Tom_paw1 jb#Tom_paw2 jb#Tom_paw3;
```

SPARQL avec le régime d'inférences [Entailment Regimes, 2013] de OWL-DL (pour les SPARQL endpoint, s'il en existe, qui permettent ce régime d'inférences) :

```
$ SELECT ?paw WHERE
$ { ?cat dolce:part ?paw ;
$     a jb:Cat ;
$     a [ rdf:type owl:Class ;
$         owl:equivalentClassOf [ a owl:Restriction ;
$             owl:onProperty dolce:part ;
$             owl:minCardinality "3"^^xsd:nonNegativeInteger ;
$             owl:maxCardinality "4"^^xsd:nonNegativeInteger ] ] }
jb:Tom_paw1 jb:Tom_paw2 jb:Tom_paw3
//rappel : Tom a seulement 3 pattes
```

FC :

```
$ ?p [some jb#Cat dolce#part: 3..4 jb#Paw ?p]
jb#Tom_paw1 jb#Tom_paw2 jb#Tom_paw3
//rappel : Tom a seulement 3 pattes
```

SPARQL :

```
$ SELECT ?head ?paw WHERE
$ { ?schrodinger_cat a jb:Cat_with_owner_Schrodinger ; dolce:part ?head . ?head a jb:Head .
$     jb:Cat ^rdf:type ?cat .
$     ?cat !rdf:type jb:Cat_with_owner_Schrodinger ; dolce:part ?paw . ?paw a jb:Paw . }
jb:Felix_head jb:Tom_paw1
```

FC :

```
$ ?r if_then_else _(
$     [?cat rdf#type: jb#Cat_with_owner_Schrodinger], //si ?cat est le chat de Schrödinger
$     [1 jb#Head ?r dolce#part of: ?cat], //chercher la tête de ?cat
$     [1 jb#Paw ?r dolce#part of: ?cat]) //sinon chercher une de ses pattes
jb#Felix_head jb#Tom_paw1
```

SPARQL :

```
$ SELECT ?claw WHERE
$ { ?cat a jb:Cat ; dolce:part+ ?claw . ?claw a jb:Claw . }
jb:Tom_claw
```

FC :

```
$ ?claw f_chain_of_part _(a cat, a claw ?claw)
jb#Tom_claw
```

Mon résolveur est capable de gérer des requêtes incluant des variables et des fonctions récursives. Il a plusieurs fonctionnalités communes avec des résolveurs pour SPARQL ou SPARQL Template.

Un tableau récapitulatif est fourni ci-après.

Comparaison entre mon résolveur de requête et d'autres résolveurs.

	mon résolveur	résolveur basé sur SPARQL	résolveur basé sur SPARQL Template
opérateur de requêtes	booléen, liste, graphe (le graphe renvoyé est le chemin reconnu)	booléen, liste, graphe (via le mot clef CONSTRUCT)	booléen, liste, graphe (via le mot clef CONSTRUCT)
métaphrases	oui, via la notation “__[]”	non	non
fonctions (dont des fonctions récursives)	oui	non	oui
relations inverses	oui, via le mot clef “of”	oui, via la notation “^”	oui, via la notation “^”
séquence	oui mais sans sucre syntaxique	oui, le sucre syntaxique “/” facilite l'utilisation	oui, le sucre syntaxique “/” facilite l'utilisation
alternatives	oui, via des contextes ou la fonction if_then_else	oui, via la notation “ ” ou “Union”	oui, via la notation “ ” ou “Union”
chaîne de relations	oui, mais une fonction récursive doit être écrite	oui, via les notations “*”, “+” ou “?”	oui, via les notations “*”, “+” ou “?”
expressions régulières	non	oui	oui
options de présentation des résultats	des équivalents de offset et limit	order by, distinct, reduced, offset, limit	order by, distinct, reduced, offset, limit
parallélisation des traitements	possible en version finale	possible via des outils comme Bobox [Falt et al., 2012]	possible via des outils comme Bobox

De plus, l'architecture de mon résolveur est basée sur la fonction `Structure_map` sur laquelle j'ai travaillé pendant la première partie de cette thèse. Cette fonction est détaillée dans la section “[5.1.2. Réutilisation de travaux précédant le changement de sujet de thèse](#)”. L'architecture de mon résolveur est détaillée dans la section “[5.2. Mon implémentation de Structure_map](#)”.

5.1.2. Fonctions de parcours et de manipulation de structures de données

Pendant la première année de cette thèse j'ai travaillé sur la représentation de composants logiciels afin que des programmes créés via ces composants soient facilement paramétrables et sémantiquement organisables. Cette section (5.1.2.) présente ces travaux.

1) Hypothèse fondamentale

Tout programme peut être conçu en combinant des fonctions de parcours ou de manipulation de structure de données est l'hypothèse sur laquelle s'appuient mes travaux sur la représentation de composants logiciels. Le résolveur de requêtes que j'ai implémenté – et qui est présenté dans la section “[5.2. Mon implémentation de Structure_map](#)” – est une preuve de concept partielle pour cette hypothèse. En effet, comme présenté dans la section 5.2, i) SRS est une composition de fonctions particulières de parcours ou de manipulation de structure de données, et ii) pour SRS, aucun autre type de composant logiciel n'est nécessaire.

2) Paramétrabilité et organisation des composants

Les fonctions de parcours ou de manipulation de structure de données introduites dans le paragraphe ci-dessus peuvent être représentées de façon précise dans une ontologie. Via des combinaisons de telles représentations, des composants de programmes existants peuvent être représentés et de nouveaux composants de programmes peuvent être directement créés – et donc également représentés. Les composants logiciels ainsi représentés peuvent donc être plus facilement comparés et organisés via des relations *part* ou *subtype* de façon modulaire et systématique. Par exemple, une fonction de tri sur des listes peut être représentée comme une combinaison entre une fonction de parcours de listes, une fonction de comparaison entre deux éléments et une fonction d'agrégation ayant en sortie une liste. Chaque fonction de tri peut être comparée aux autres fonctions de tri via ces trois fonctions (parcours, comparaison et agrégation). Ainsi, des fonctions de tri peuvent être organisées dans une ontologie.

3) Représentation de fonctions de parcours et de manipulation de structure de données

En début de thèse, j'ai spécifié et/ou déclaré plusieurs *fonctions de parcours et de manipulation de structure de données* utilisables par `Structure_map`. Ces spécifications et déclarations sont fournies dans l'annexe 1. Certaines des spécifications que j'ai écrites représentent des fonctions du framework du LAS.

Pour les déclarations des *fonctions de parcours et de manipulation de structure de données*, j'ai utilisé FL. Pour leurs définitions, j'ai préféré utiliser KIF car i) de par sa nature relationnelle et fonctionnelle, KIF est aussi pratique que FL pour définir des relations et des fonctions, et ii) FL inclus KIF mais KIF est plus connu que FL.

4) Combinaison de fonctions de parcours ou de manipulation de structure de données

Pour combiner les fonctions de parcours ou de manipulation de structure de données, j'utilise une fonction nommée *Structure_map* qui généralise les fonctions "map" des langages fonctionnels. Philippe Martin a conçu une première version de la fonction `Structure_map` nommée `pm#Structure_map`. J'ai créé une seconde version de cette fonction en lui ajoutant des paramètres afin de permettre le parcours de structure de données indexées telles que les tableaux associatifs. Philippe Martin a ensuite créé une troisième version prenant en compte ces changements et ayant 7 paramètres. J'ai ensuite créé une version simplifiée de cette troisième version de `pm#Structure_map` et je l'ai nommée `jb#Structure_map`. `jb#Structure_map` prend seulement 3 arguments. La contrepartie de cette simplification est une paramétrabilité moins fine. En effet, dans `jb#Structure_map`, le parcours de la structure de données en entrée est spécifié uniquement par défaut et ne peut donc pas être paramétré. Des déclarations en FL de `pm#Structure_map` et de `jb#Structure_map` sont fournies dans la sous-section "5) `Structure_map`". `pm#Structure_map` a également été implémenté et spécialisé en Javascript par Philippe Martin sous le nom de `CollectionArray_selectAndExploit`.

Une *combinaison de fonctions Structure_map représentant un composant logiciel* est une spécification déclarative à la fois fonctionnelle et logique de ce composant. En effet, `Structure_map` a des définitions à la fois en logique et dans un langage fonctionnel (par exemple, la définition en Javascript écrite par Philippe Martin). Des vérifications ou inférences logiques peuvent donc être effectuées statiquement ou bien durant l'exécution d'un programme. Cela peut permettre d'effectuer, rechercher ou vérifier certains aspects de sécurité d'un programme, son fonctionnement ou la cause des messages d'erreur qu'ils délivrent. Par exemple, la détection de boucles infinies peut être facilitée via une comparaison du parcours de structure de donnée utilisé avec la condition d'arrêt spécifiée pour ce parcours. Plus généralement, il s'agit ici d'un moyen de fusionner programmation et représentation, et non pas seulement d'indexer certaines fonctions par certains concepts dans des ontologies, comme cela se fait actuellement dans les recherches liant logiciels et représentation de connaissances – par exemple, comme dans [Parreiras & Staab, 2010]. Les travaux sur `Structure_map` présentés dans cette thèse n'ont pas encore été publiés dans des articles. Ils pourraient être publiés, par exemple, dans SIGSOFT 2018.

5) Structure_map

`Structure_map` est une fonction qui permet d'appliquer une fonction passée en paramètre sur les éléments d'une structure de données quelconque en entrée. Ainsi, `Structure_map` généralise les fonctions "map" des langages fonctionnels qui elles ne s'appliquent qu'à des collections – par exemple, des listes ou des tableaux associatifs. Quelques relations de spécialisations entre des fonctions `Structure_map` sont présentées ci-dessous.

```
pm#Structure_map /*version 3*/
  > (jb#Structure_map
    > (jb#Graph_map > jb#Simple_graph_map));
```

5.1) pm#Structure_map

La définition de `pm#Structure_map` est présentée ci-après. Les arguments en police grasse proviennent indirectement des modifications que j'ai apporté via la version 2 de `pm#Structure_map` (cf. sous-section "4) Combinaison de fonctions de parcours ou de manipulation de structure de données" ci-dessus).

```

pm#Structure_map /*version 3*/
  has_input: 1 pm#structured_object ?input_structure,
    0..1 pm#function_with_input_a_structure_and_an_index //or pm#monad
      ?fctToApply //?fctToApply includes its pre-conditions/transformation
      //Efficiency note: composing many functions into one ?fctToApply avoids to
      //later have to reprocess the result set (i.e., re-loop on it)
    0..1 pm#fct_inputSuccessor ?fct_inputSuccessor/*?s,?i*/
      //If it is '()', take the first 'successor_relation_or_function' from
      //the type of ?set (list, stack,tree,multiset,...); if none: error
      //E.g., string_left-to-right_next_uppercase_char_from_3rd_char_to_4th'A'()
      //If not functional: not deterministic
    0..1 pm#fct_outputAggregation ?fct_outputAggregation/*?r,?i*/
      //sets a successor relation between the collected results or
      // aggregates them with a function
    0..1 pm#fct_isAtStructureEnd ?fct_isAtStructureEnd/*?s,?i*/
    0..1 pm#fct_accessElement ?fct_accessElement/*?s,?i*/
    0..1 pm#fct_nextIndex ?fct_nextIndex/*?i,(?s)*/
  has_output: 0..1 pm#structured_object ?output_structure,
>:= [_ pm#language: pm#KIF] $(
  (pm%def_fct pm%fct_successor-based_map //or now: pm%fct_structure_map
    (?fctToApply/*?s,?i*/ //in pm%fct_successor-based_map, ?input_structure
    ?input_structure //has a content/element (a 'first') with a successor;
    //in pm#fct_index-based_map, it is a whole structure that has indices
    ?fct_isAtStructureEnd/*?s,?i*/ ?fct_accessElement/*?s,?i*/
    ?fct_inputSuccessor/*?s,?i*/ ?index ?fct_nextIndex/*?i,(?s)*/
    ?fct_outputAggregation/*?i,?s,?s*/
    ?outputIndex ?fct-outputAggregation_type
    ?future_output_structure /*null/toReuse/toInit/toModify
    (initialized even if empty)*/
    ) -> ?output_structure :=
  (if (value ?fct_isAtStructureEnd ?input_structure ?index) ?future_output_structure
    (if (= ?fct-outputAggregation_type
      'pm#output_aggregation_by_recursion_and_addition_at_output_beginning)
      (= ?output_structure
        (value ?fct_outputAggregation/*e.g.,cons*/ ?index
          (value ?fctToApply (value ?fct_accessElement ?input_structure ?index))
          (pm%fct_successor-based_map
            ?fctToApply (value ?fct_inputSuccessor ?input_structure ?index)
            ?fct_isAtStructureEnd ?fct_accessElement
            ?fct_inputSuccessor /*returns structure so not redundant with
            ?fct_nextIndex*/
            (value ?fct_nextIndex ?index ?s ?fct_inputSuccessor)
            ?fct_nextIndex
            ?fct_outputAggregation (+1 ?outputIndex)
            ?fct-outputAggregation_type
            ?future_output_structure/*null*/ )
            ?future_output_structure ) )
        (if (= ?fct-outputAggregation_type
          'pm#output_aggregation_by_recursion_and_addition_at_output_end)
          (= ?output_structure
            (pm%fct_successor-based_map
              ?fctToApply (value ?fct_inputSuccessor ?input_structure ?index)
              ?fct_isAtStructureEnd ?fct_accessElement
              ?fct_inputSuccessor
              (value ?fct_nextIndex ?index ?s) ?fct_nextIndex
              ?fct_outputAggregation (+1 ?outputIndex) ?fct-outputAggregation_type
              (value ?fct_outputAggregation/*e.g.,append*/
                (value ?fctToApply
                  (value ?fct_accessElement ?input_structure ?index) )
                  ?future_output_structure )
              ?future_output_structure ) )
            ) ) ) ) );

```


5.2) jb#Structure_map

Comme indiqué dans l'extrait ci-dessous, jb#Structure_map hérite de pm#Structure_map et utilise des méthodes par défaut pour la plupart des arguments spécifiés dans pm#Structure_map.

```
jb#Structure_map < pm#Structure_map, //tous les input et output sont hérités
> (jb#Structure_map_for_graph_matching_and_building = jb#Graph_map),
>:=> [_ pm#language: pm#KIF] $(
  pm%fct_successor-based_map(
    ?fctToApply
    ?input_structure
    (default_method_for_this_method_type_and_this_structure ?input_structure
      'pm%fct_isAtStructureEnd)
    (default_method_for_this_method_type_and_this_structure ?input_structure
      'pm%fct_accessElement)
    (default_method_for_this_method_type_and_this_structure ?input_structure
      'pm%fct_inputSuccessor)
    (default_index ?input_structure)
    (default_method_for_this_method_type_and_this_structure ?input_structure
      'pm%fct_nextIndex)
    ?fct_outputAggregation
    (default_outputIndex ?input_structure)
    'pm#output_aggregation_by_recursion_and_addition_at_output_end
    null) )$;
```

5.3) Versions de Structure_map sur lesquelles SRS est basé

SRS est une implémentation sous Delphi de jb#Simple_graph_map pour RichGraph. jb#Simple_graph_map est une spécialisation de jb#Graph_map. RichGraph est une implémentation de graphe. La section “[5.2. Mon implémentation de Structure_map](#)” donne plus de détails sur SRS et sur RichGraph et sur SRS. L'extrait suivant présente jb#Graph_map et jb#Simple_graph_map.

```

jb#Graph_map
  has_input: 0..1 (jb#fct_graphSpecializationFinding
    < pm#function_with_input_a_structure_and_an_index) ?fctToApply
    0..1 (jb#fct_graphSpecializationBuilding
      < pm#function_with_input_a_structure_and_an_index) ?fct_outputAggregation
    0..1 jb#fct_withOutputAGraph ?fct_returning_a_graph_to_find
    0..1 jb#fct_withOutputAGraph ?fct_returning_a_graph_to_build,
  >:=> [_ pm#language: pm#KIF] $(
(pm%def_fct jb%Graph_map
  (?fctToApply
    ?input_structure
    ?fct_isAtStructureEnd ?fct_accessElement
    ?fct_inputSuccessor ?index ?fct_nextIndex
    ?fct_outputAggregation
    ?outputIndex ?fct-outputAggregation_type
    ?future_output_structure
    ?fct_returning_a_graph_to_build
    ?fct_returning_a_graph_to_find
  ) -> ?output_structure :=
  (if (= ?fct-outputAggregation_type
    'pm#output_aggregation_by_recursion_and_addition_at_output_end)
    (= ?output_structure
      (pm%fct_successor-based_map
        ?fctToApply (value ?fct_inputSuccessor ?input_structure ?index)
        ?fct_isAtStructureEnd ?fct_accessElement
        ?fct_inputSuccessor
        (value ?fct_nextIndex ?index ?s) ?fct_nextIndex
        ?fct_outputAggregation (+1 ?outputIndex) ?fct-outputAggregation_type
        (value ?fct_outputAggregation
          (value ?fctToApply
            (value ?fct_accessElement ?input_structure ?index)
            ?fct_returning_a_graph_to_find)
          ?future_output_structure
          fct_returning_a_graph_to_build)
        ?future_output_structure)
      ) ) )$;

jb#Simple_graph_map /*Rappel, les arguments suivants sont hérités ?input_structure,
  ?fct_returning_a_graph_to_build, ?fct_returning_a_graph_to_find*/
  >:=> [_ pm#language: pm#KIF] $(
  jb%Graph_map(
    (default_method_for_this_method_type_and_this_structure
      ?input_structure
      jb%fct_graphSpecializationFinding)
    ?input_structure
    (default_method_for_this_method_type_and_this_structure ?input_structure
      pm%fct_isAtStructureEnd)
    (default_method_for_this_method_type_and_this_structure ?input_structure
      pm%fct_accessElement)
    (default_method_for_this_method_type_and_this_structure ?input_structure
      pm%fct_inputSuccessor)
    (default_index ?input_structure)
    (default_method_for_this_method_type_and_this_structure ?input_structure
      pm%fct_nextIndex)
    (default_method_for_this_parameter_and_this_structure ?output_structure
      jb%fct_graphSpecializationBuilding)
    (default_outputIndex ?input_structure)
    'pm#output_aggregation_by_recursion_and_addition_at_output_end
    null
    ?fct_returning_a_graph_to_build
    ?fct_returning_a_graph_to_find) )$;

```

6) Avantages et inconvénients pour l'utilisation systématique de fonctions basées sur Structure_map

Pour faciliter la lecture de la section courante, j'appelle *fonctions paramètres de pm#Structure_map* les fonctions suivantes ou leurs spécialisations.

- pm#function_with_input_a_structure_and_an_index
- pm#fct_inputSuccessor
- pm#fct_outputAggregation
- pm#fct_isAtStructureEnd
- pm#fct_accessElement
- pm#fct_nextIndex

L'écriture des représentations de fonctions du framework LAS m'a permis de porter un regard critique sur les apports de l'utilisation systématique de fonctions basées sur Structure_map pour programmer. Les *fonctions paramètres de pm#Structure_map* peuvent être réutilisées si (mais seulement si) elles sont organisées dans une ontologie. Cette *réutilisabilité* permet d'améliorer la *lisibilité* et la *maintenabilité* [maintainability] du code. En outre, plus ces fonctions paramètres sont représentées de façon précise, plus elles sont réutilisables. Les points suivants décrivent les cas où l'utilisation systématique de fonctions basées sur Structure_map n'est pas avantageuse.

- Si des lambda-abstractions sont utilisées comme fonctions paramètres, le code devient peu lisible, peu réutilisable et peu maintenable. En effet, une lambda-abstraction ne peut être utilisée que là où elle est définie ; elle ne peut donc pas être réutilisée via une référence quelle que soit la précision de sa définition. Pour chaque appel d'une fonction basée sur Structure_map, de nouvelles lambda-abstractions doivent être écrites ou ré-écrites. Chaque fois qu'une lambda-abstraction est ré-écrite, i) la taille du code augmente inutilement et le code devient de moins en moins lisible, et ii) une nouvelle lambda-abstraction doit être maintenue et donc la maintenabilité du code diminue.
- Si les fonctions paramètres ne sont pas organisées dans une ontologie, elles sont peu comparables (cf. Comparaison sémantique) et donc peu réutilisables (cf. Précision sémantique).

7) Comparaison de Structure_map avec les patrons de conception [design pattern] orienté objet du génie logiciel

Structure_map peut être vue comme une généralisation du patron de conception "Visiteur". En effet, une architecture qui suit ce patron permet d'appliquer quelques fonctions prédéfinies à quelques structures de données également prédéfinies. Une architecture qui suit Structure_map permet d'appliquer des traitements à toute structure de donnée. De plus, contrairement à l'utilisation des patrons de conception des langages orienté objet, l'utilisation de Structure_map n'est pas limitée à un langage particulier.

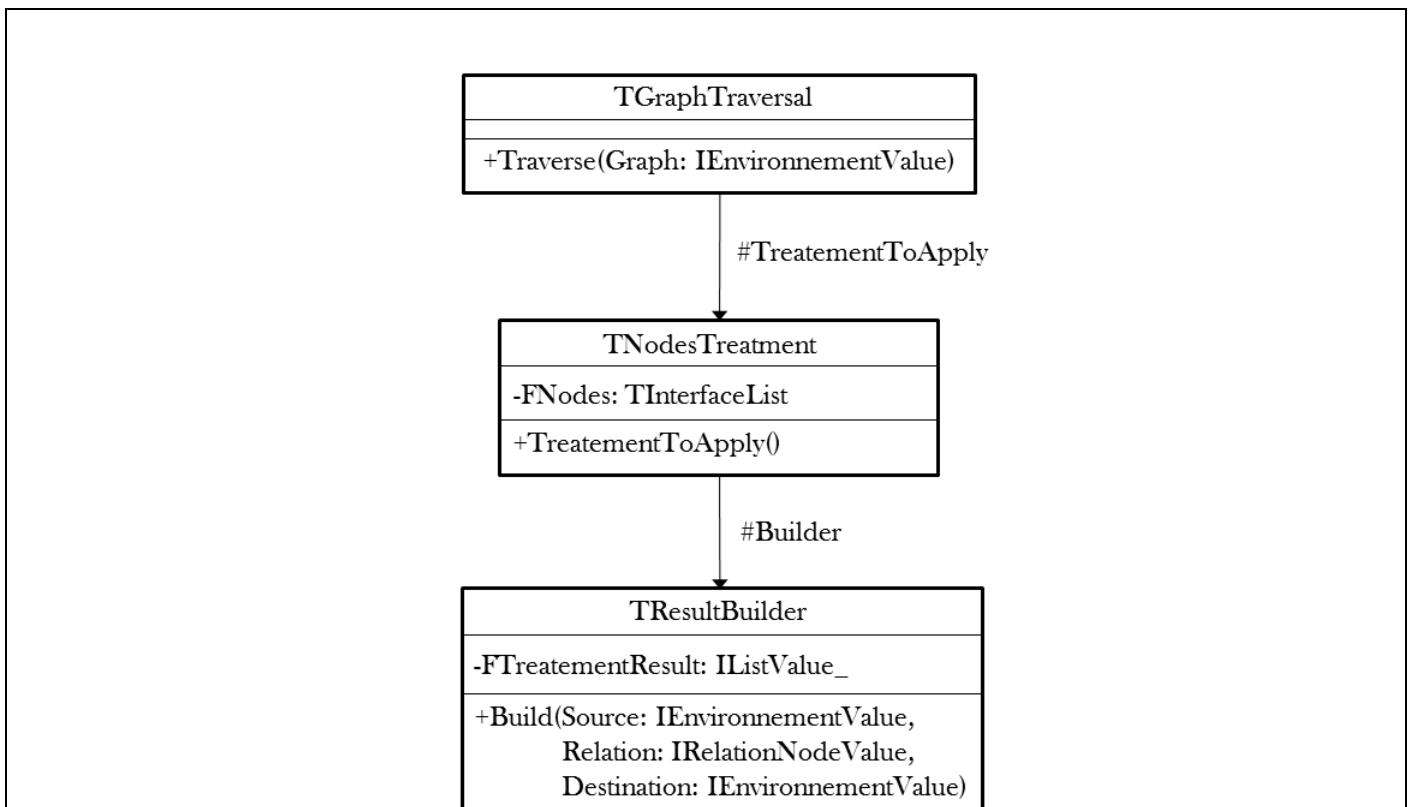
5.2. Mon implémentation de Structure_map

Pendant ma thèse, j'ai été amené à implémenter SRS, un résolveur de requêtes basé sur la fonction Structure_map. Structure_map est présentée dans la section précédente "5.1.2. Réutilisation de travaux précédant le changement de sujet de thèse". Dans la section présente 5.2, je présente les composants que j'ai développés et la façon dont ils peuvent être combinés pour former – entre autres – SRS. Ces composants sont conçus pour exploiter quelques unes des structures de données développées par logiCells et GTH pour LAS.

Composants développés

Sous Delphi, j'ai représenté par des classes les quelques paramètres (input) de pm#Structure_map suivants : pm#monad – représenté par la classe TNodesTreatment –, pm#fct_outputAggregation – représenté par la classe TResultBuilder – et une fonction pm#fct_inputSuccessor par défaut pour jb#RichGraph_v1.0 – représenté par la classe TGraphTraversal. Ces classes sont présentées dans la figure ci-après. Pour des raisons de lisibilité, certains champs et fonctions telles que les accesseurs (get et set) ont été omis.

Diagramme de classes montrant les classes principales de SRS



Légende

1. les flèches representent des pointeurs
2. IEnvironnementValue est un noeud dans un RichGraph
3. IRelationNodeValue est un noeud relation dans un RichGraph
4. TInterfaceList est une liste indexée d'interfaces
5. IListValue_ est une liste simplement chaînée d'interfaces qui hérite de IEnvironnementValue

Principe

La classe Traverse, qui implémente la fonction référée par ?fct_inputSuccessor ci-avant, permet de parcourir un jb#RichGraph_v1.0 à partir d'un noeud passé en paramètre. Pour chaque noeud parcouru, la fonction TraitementToApply de la classe TraitementToApply est appelée. Cette fonction applique un traitement sur un ou plusieurs noeuds contenus dans la liste FNodes puis appelle la fonction Build de la classe TResultBuilder. La fonction Build agrège un résultat et le stocke dans un conteneur (par exemple, une liste).

Quelques spécialisations des classes principales sont présentées dans les trois figures ci-après. Ces spécialisations permettent de paramétrer SRS.

Figure 5.2. Spécialisations de TGraphTraversal en pm#UML.

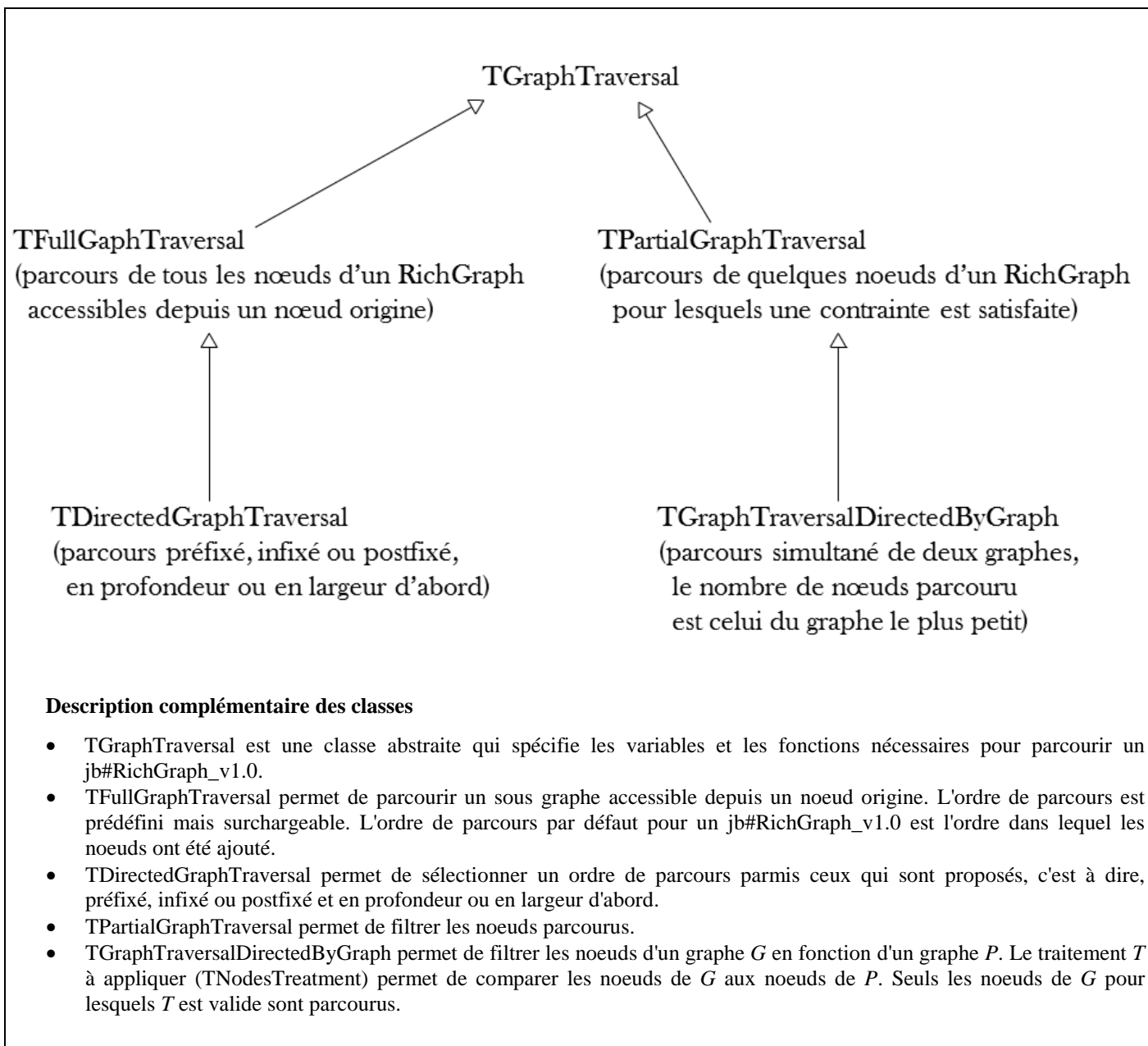


Figure 5.3. Specialisations de TNodesTreatment en pm#UML.

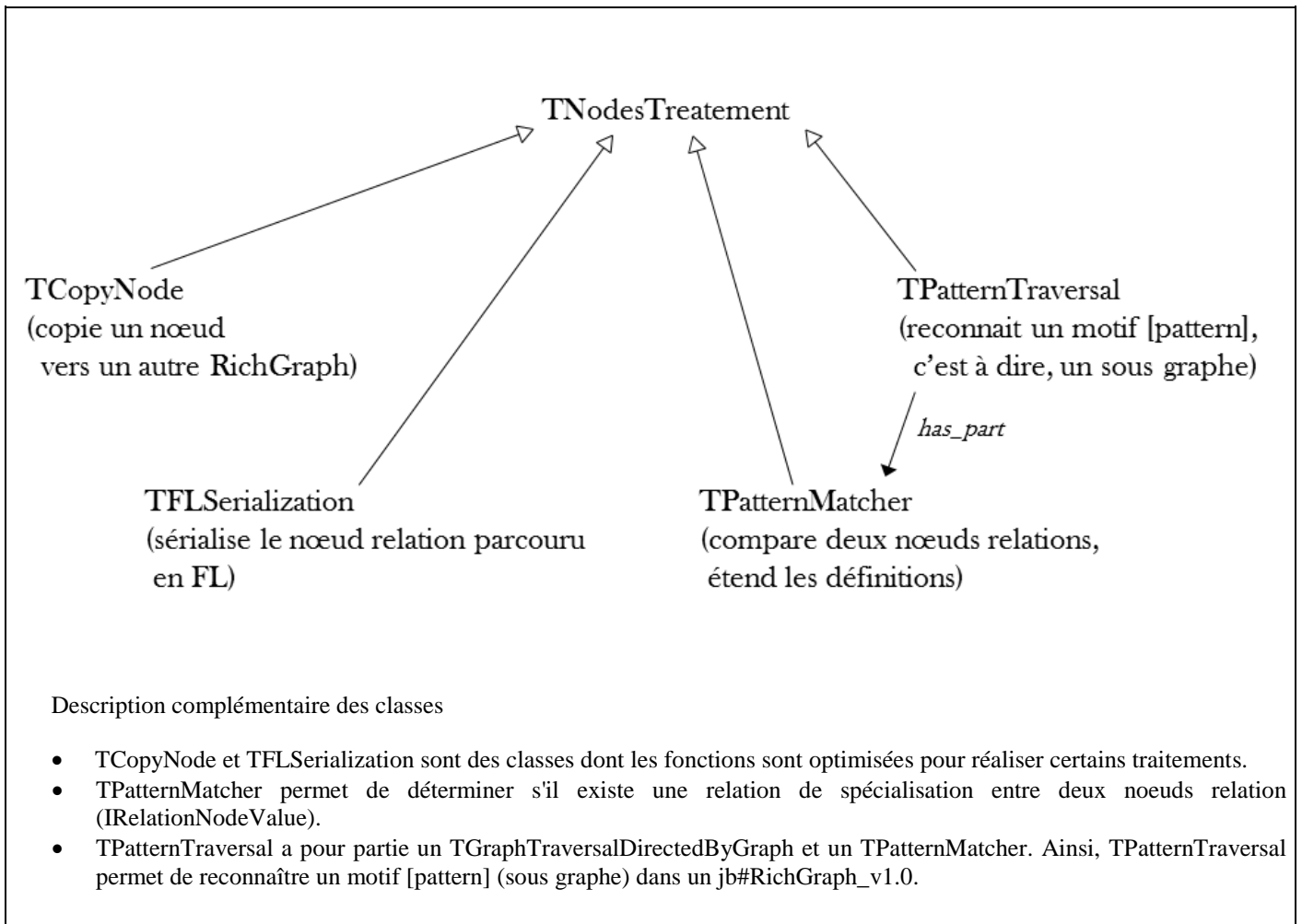
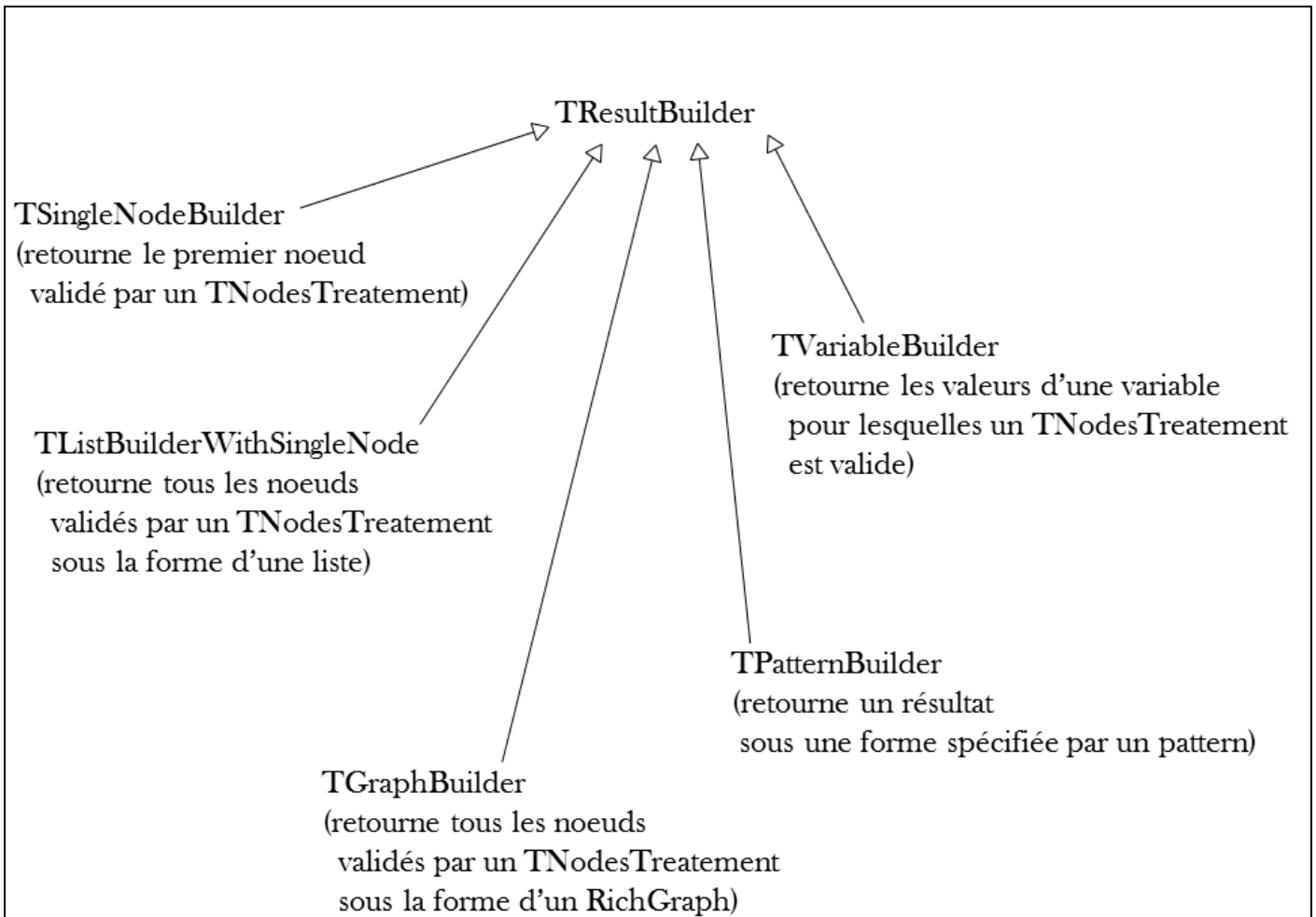


Figure 5.4. Specialisations de TResultBuilder en pm#UML.



Description complémentaire des classes

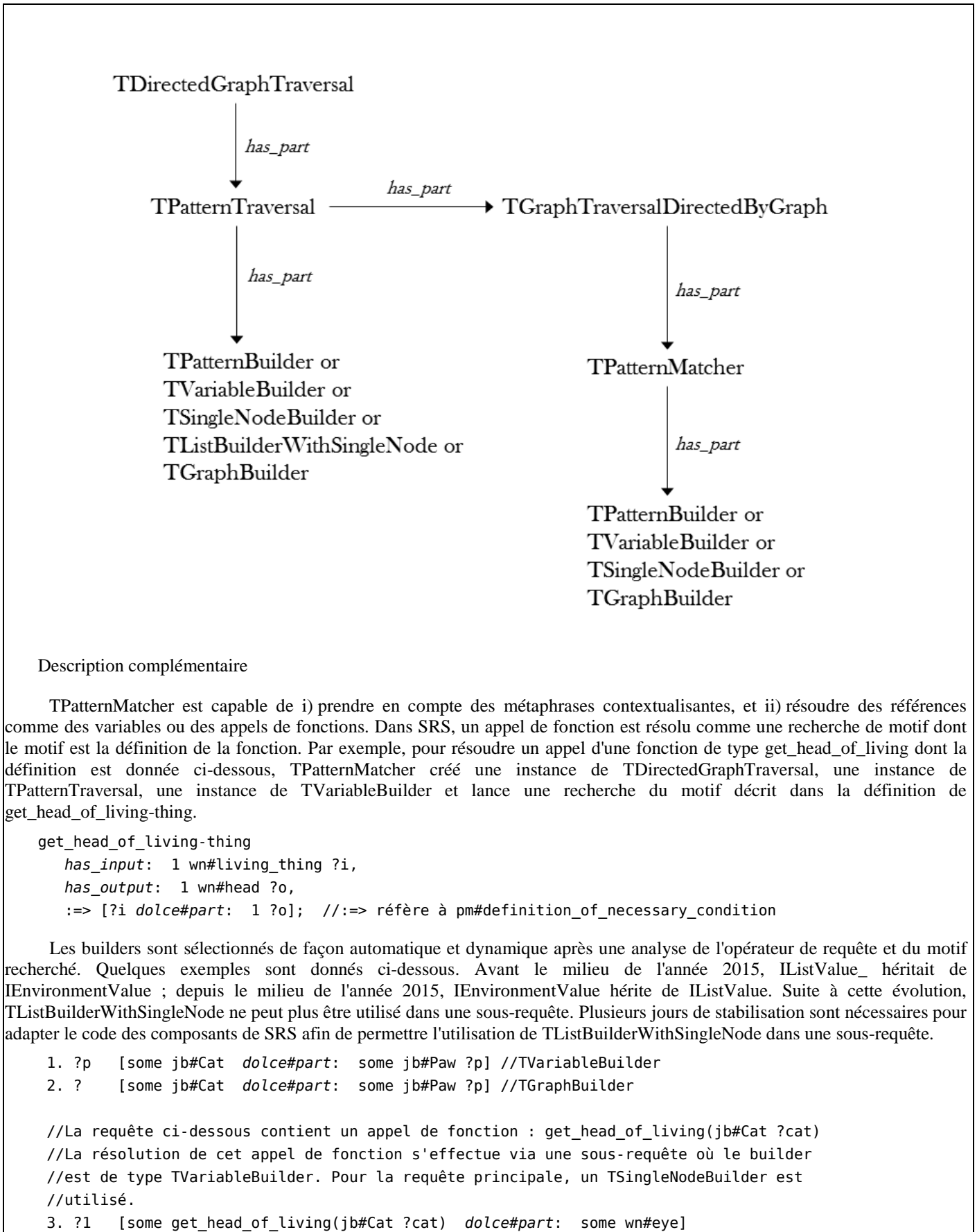
- TSingleNodeBuilder, TListBuilderWithSingleNode et TGraphBuilder sont des classes dont les fonctions sont optimisées pour agréger des éléments dans des collections prédéfinies.
 - TSingleNodeBuilder permet de construire un résultat contenant un seul noeud et d'interrompre le parcours de graphe une fois ce résultat construit. Pour résoudre la requête “?1 some jb#Cat dolce#part: some jb#Paw;”, un TSingleNodeBuilder peut être utilisé.
 - TListBuilderWithSingleNode permet de construire une liste de noeuds concept. D'autres constructeurs de listes spécialisés permettent de construire des listes contenant d'autres types de noeuds – par exemple, des noeuds relation – ou même des listes de noeuds. Ce builder peut être utilisé pour la résolution d'une requête via l'emploi du mot clef “list” en début de requête.
 - TGraphBuilder permet de construire une liste de copies des motifs retrouvés. Ce builder peut être utilisé pour la résolution d'une requête via l'emploi du mot clef “path” en début de requête.
- TVariableBuilder construit une liste des valeurs prises par une variable de la requête. La résolution de la requête “?p [some jb#Cat dolce#part: some jb#Paw ?p]” nécessite un TVariableBuilder lié à la variable “?p”.
- TPatternBuilder construit une spécialisation d'un graphe donné en paramètre de la façon suivante.
 - Pour chaque type de concept (contenu dans le graphe donné en paramètre) quantifié existentiellement, une instance de ce type est créée.
 - Les types de concept quantifiés universellement ne sont pour l'instant pas autorisés.
 - Une copie de chaque individu et de chaque noeud relation est créée.
 - Pour chaque référence (c'est à dire, une variable ou fonction), une copie du noeud référé est créée.

Par exemple, dans le graphe suivant [?l first: get_car_of(Jeremy_Benard), rest: 1 jb#List], jb#List est un type de concept, ?l et get_car_of(Jeremy_Benard) sont des références. Un TPatternBuilder qui prend ce graphe en paramètre peut créer le graphe suivant [List_123 first: JB_car_1, rest: List_894] où i) List_894 est une instance de jb#List, et ii) List_123 et JB_car_1 sont des copies des noeuds référés respectivement par ?l et get_car_of(Jeremy_Benard).

Structure_map Request Solver (SRS)

SRS est un assemblage particulier des classes décrites dans la section “Composants développés” ci-avant. Cet assemblage est décrit ci-dessous.

Figure 5.5. Classes qui composent Structure_map Request Solver (SRS) en pm#UML.



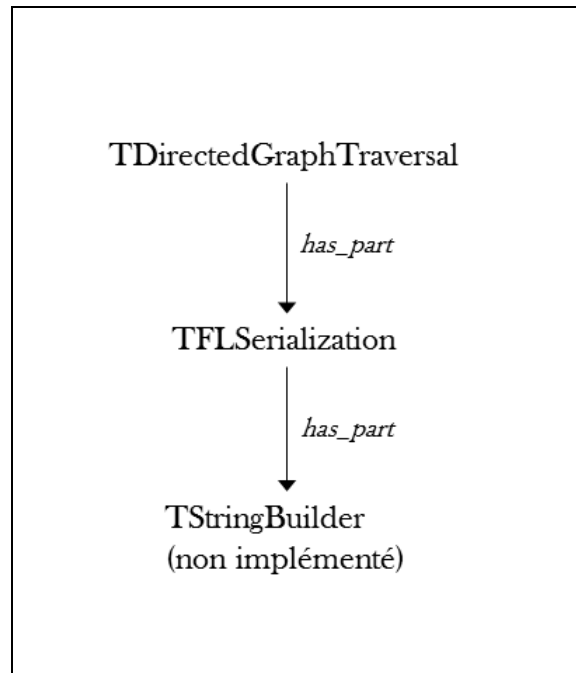
Pour résoudre des appels de fonctions basées sur Structure_map, SRS pourrait exploiter la définition en logique de Structure_map. Cependant, pour résoudre ces appels de fonction, j'ai choisi d'exploiter le fait que SRS soit une implémentation de Structure_map afin d'améliorer un peu le temps de calcul. Ainsi, ces appels sont résolus de manière un peu ad hoc car – contrairement aux autres appels de fonctions – la définition en logique de Structure_map n'est pas directement utilisée : c'est la définition de SRS – qui est une implémentation de Structure_map – qui est utilisée. La table ci-dessous permet de comparer la façon dont sont résolus les appels de fonctions basées sur Structure_map et les autres appels de fonctions.

Exemple de requête en FC	Définition en FL de la fonction appelée dans la requête	Composants utilisés pour paramétrer Structure_map
<pre>get_head_of_cat(cat_5) //cat_5 instance of: cat</pre>	<pre>get_head_of_cat has_input: 1 cat ?cat, has_output: 1 head_of_cat ?catHead, := [?catHead part of: ?cat];</pre>	<pre>//1. parcourir les relations de cat_5 TDirectedGraphTraversal(get_head_of_cat.input) //2. reconnaître [?catHead part of: ?cat] TPatternTraversal(get_head_of_cat.definition) //3. renvoyer la valeur de ?head TVariableBuilder(get_head_of_cat.output)</pre>
<pre>jb#Simple_graph_map(my_cat_list, get_head_of_cat, make_list)</pre>	<pre>/* La définition logique de jb#Simple_graph_map existe dans l'ontologie mais, pour des raisons d'efficacité, n'est pas utilisée directement pour la résolution : c'est la version procédurale (SRS) qui est utilisée. */</pre>	<pre>//1. parcourir les relations de my_cat_list TDirectedGraphTraversal(jb#Simple_graph_map.getArgument(0)) //2. appliquer get_head_of_cat TPatternTraversal(jb#Simple_graph_map.getArgument(1)) //3. appliquer make_list TPatternBuilder(jb#Simple_graph_map.getArgument(2))</pre>

Autres utilisations des composants développés

Tout comme les fonctions servant à paramétrer Structure_map, les composants que j'ai développés sont conçus pour être très modulaires et réutilisables. Toutefois, certains sont peu paramétrables car optimisés pour des tâches très spécifiques. Par exemple, TFLSerialization est optimisé pour l'export de connaissances vers la notation de FL et ne peut pas être paramétré pour exporter des connaissances vers une autre notation. La figure ci-après présente un ensemble de composants utilisant TFLSerialization.

Figure 5.6. Export ad hoc
vers FL en pm#UML.



Tout comme les TResultBuilder peuvent être sélectionnés dynamiquement, des TNodesTreatment et des TGraphTraversal peuvent également être sélectionnés dynamiquement. Par exemple, un composant TNodeTreatment de sélection peut être utilisé pour choisir un TNodeTreatment à appliquer en fonction du contexte d'un noeud (cf. Méta-phrased). Ainsi, pour résoudre une requête telle que celle qui est présentée ci-dessous, il est possible de créer et d'utiliser TGetCarCost – un TNodesTreatment spécialisé pour exécuter la fonction “get_car_cost”. La requête ci-dessous utilise la notation `_[]` pour délimiter une méta-phrased sur l'élément précédent, ici “get_car_cost(?c)”.

```
? [a car ?c
    has_attribute: a ^(cost value: "get_car_cost(?c)" _[language: Pascal],
                    unit: a euro)]
//La fonction get_car_cost ci-dessus est une fonction implémentée en Pascal.
//TGetCarCost peut être utilisé plutôt que TPatternMatcher.
```

D'après l'hypothèse fondatrice de mes travaux sur Structure_map, toute fonction – même initialement écrite dans un langage de programmation – peut être représentée comme une combinaison de fonctions Structure_map. La fonction “get_car_cost” de l'exemple ci-dessus (et donc également le TNodesTreatment qui réfère à cette fonction) peut elle aussi être représentée comme une combinaison de fonctions Structure_map. Cette représentation peut être structurellement organisée dans une ontologie de façon systématique. La fonction “get_car_cost” peut ainsi être sémantiquement indexée. La recherche et la sélection de cette fonction peut en être ainsi facilitée. Dans tous les cas, elle est exécutable de façon optimisée via son implémentation en Pascal.

En 2013, j'ai commencé à représenter des fonctions simples du framework du LAS via Structure_map. Ce sont typiquement des fonctions de traitement sur des chaînes de caractères. Quelques exemples sont donnés dans l'annexe 2. Lorsque toutes les fonctions du framework seront ainsi représentées, logiCells disposera d'une bibliothèque de composants sémantiquement organisés.

5.3. État des lieux et perspectives

5.3.1. État de l'implémentation

Pendant cette thèse, j'ai été amené à travailler sur trois éléments de programme. J'ai créé un importeur ad hoc pour un sous langage de FL en utilisant GoldParser, j'ai étendu RichGraph jusqu'à la version 1.0 et j'ai créé SRS. Les sections suivantes donnent des détails sur l'état de l'implémentation de l'importeur ad hoc, de RichGraph et de SRS.

Analyseur ad hoc pour FL

En début de thèse, FL devait servir de langage de base pour les ontologies développées pour le LAS. J'ai donc implémenté un analyseur lexico-syntaxique via GoldParser pour importer des descriptions écrites dans la notation FL vers RichGraph. Toutes les fonctionnalités de FL ne sont cependant pas gérées ou sont seulement partiellement gérées. La liste ci-dessous présente quelques unes de ces fonctionnalités qui restent à implémenter.

- FL propose des abréviations pour certaines relations. Par exemple, ">" ou "." pour pm#subtype ou "=>" pour pm#definition_of_necessary_condition. Actuellement, mon analyseur ne permet l'utilisation que de deux de ces abréviations : ">" et "=" (pour pm#equivalent).
- FL permet d'utiliser une notation préfixée, infixée ou postfixée. Seule la notation infixée est interprétée par mon analyseur.
- Dans sa notation, FL permet l'utilisation des trois éléments de sucre syntaxique suivants pour le positionnement de quantificateurs et de métaphrases. Seul le premier peut actuellement être interprété par mon analyseur.
 1. "_[]" comme dans "car part: wheel |[any->some];"
 2. "[]_" comme dans "car part |[any->some]: wheel;"
 3. "[]" comme dans "car |[any->some] part: wheel;"
- FL permet de spécifier les interprétations suivantes pour des collections. Mon analyseur ne peut interpréter que la première de ces spécifications.
 1. Une collection peut être interprétée de façon distributive. Par exemple, dans la phrase "9 juges ont chacun approuvé 50 lois différentes (donc au total 450 approbations ; et de 50 à 450 lois)", les lois ont été approuvées individuellement par chaque juge.
 2. Une collection peut être interprétée de façon collective. Par exemple, dans la phrase "un groupe de 9 juges a collectivement approuvé 50 lois", les 50 lois ont été approuvées de façon collective (par exemple via un vote).
 3. Une collection peut être interprétée de façon cumulative, les relations depuis ou vers cette collection portent alors réellement sur cette collection et pas sur ses éléments. Par exemple, dans la phrase "un groupe de juges a une taille de 9", 9 est la taille du groupe.

Depuis le milieu de l'année 2015, Anil Cassam-Chenaï a prit la décision d'utiliser une notation proche de la notation de Peano et donc de KIF comme notation pour écrire les ontologies pour le LAS. Dans cette thèse, cette notation sera nommée BW_notation (BW réfère à BaboukWeb). Les raisons de cette décision sont : i) depuis l'année 2015, Anil Cassam-Chenaï développe la version 2 de RichGraph, et ii) BW_notation est structurellement plus proche de RichGraph 2.0 que FL. Le développement de BW_notation n'est pas encore terminé, c'est pourquoi BW_notation n'est pas encore représenté dans KRLO. L'annexe 3 fournit une description plus complète de BW_notation.

RichGraph

La version 1.0 de RichGraph est utilisée dans LAS depuis fin 2015.

Anil Cassam-Chenaï travaille actuellement sur la mise en production de la version 2.0 de RichGraph. Les équipes de développement de logiCells devraient commencer à travailler avec cette version pendant l'année 2017.

SRS

Pour TDirectedGraphTraversal, je n'ai implémenté et testé que le parcours préfixé en profondeur d'abord. Avec un peu de travail, les autres types de parcours pourraient être également mis en oeuvre.

J'ai implémenté et testé TGraphBuilder fin 2014. TGraphBuilder ne fonctionne plus avec les versions actuelles de RichGraph (1.0) et de TGraphTraversalDirectedByGraph. Un peu de travail est nécessaire pour adapter les méthodes de TGraphBuilder à ces nouvelles versions.

La mise en oeuvre de différents types de parcours pour TDirectedGraphTraversal et la remise en service de TGraphBuilder permettraient de paramétrer SRS de façon plus fine. Cependant, l'implémentation actuelle est suffisante pour répondre à tous les types de requêtes décrits dans la section "Comparaisons" de la section "[5.1.1. But et comparaisons](#)". L'implémentation actuelle est également suffisante pour traiter les fonctions d'export décrites dans les sections "[6.1.1.3. Fonction d'export spécifiée via Structure_map](#)" et "[6.1.1.2 Procédure d'export par défaut spécifiée dans KRLO](#)".

Pour traiter la fonction d'export que j'ai écrite, SRS doit être capable de traiter des fonctions de génération de graphes et non uniquement des fonctions de recherche de spécialisations de graphes. Pour traiter des fonctions de génération de graphe, les trois approches décrites ci-après sont possibles.

1. **Utilisation de plusieurs spécialisations de TResultBuilder, chacune étant spécialement conçue pour traiter une fonction particulière de génération de graphe.** Une fonction particulière de génération de graphe peut être traitée par un TResultBuilder spécialement conçu à cet effet. Par exemple, un TAppendString peut être utilisé pour traiter la fonction `fc_append`. Si un TResultBuilder spécifique était implémenté pour chaque fonction de génération de graphe utilisée dans la requête d'export par défaut, SRS pourrait traiter cette requête. Ces TResultBuilder spécifiques sont simples à implémenter et peuvent être conçus pour réaliser des traitements optimisés en terme de temps de calcul et de consommation mémoire. En revanche, ils sont i) peu réutilisables car pour chaque nouvelle fonction de génération de graphe, un nouveau TResultBuilder spécifique doit être implémenté, et ii) pénibles à maintenir car chaque évolution des fonctions de génération de graphe doit être reproduite sur le TResultBuilder spécifique associé.
2. **Utilisation d'une spécialisation de TResultBuilder pour déléguer le traitement des fonctions de génération de graphe à un interpréteur d'un langage de programmation.** Si une fonction de génération de graphe est définie dans un langage de programmation, elle peut être traitée par un interpréteur de ce langage. Pour chaque interpréteur, un TResultBuilder peut être conçu pour appeler cet interpréteur via son API et réutiliser ses résultats. Cette approche est intéressante pour accélérer le traitement des fonctions de génération de graphe. En revanche, elle ne facilite pas leur réutilisation (cf. réutilisabilité sémantique).
3. **Utilisation de TPatternBuilder, une classe qui hérite de TResultBuilder, pour traiter des fonctions de génération de graphe.** Une fonction de génération de graphe peut être traitée par un TPatternBuilder (cf. Figure 5.4. Spécialisations de TResultBuilder en pm#UML). Pour les raisons données dans les deux points ci-dessus, c'est cette approche que j'ai choisie. J'ai commencé le développement de TPatternBuilder fin 2016, son développement pourrait être terminé pendant l'année 2017.

Une partie importante du travail restant sur SRS est l'optimisation du code en vue d'améliorer le temps de calcul. Ainsi, le traitement des fonctions d'import et d'export présentées dans les sections [6.1.1.2.](#), [6.1.1.3.](#) et [6.3.](#) pourrait se terminer rapidement au moins lorsque le nombre de phrases à traduire est faible. À l'heure actuelle, même lorsque la phrase à traduire est une relation binaire, une requête d'export peut prendre jusqu'à 15 secondes pour se terminer. Anil Cassam-Chenai et moi pensons pouvoir réduire le temps de calcul à quelques centièmes de secondes pour l'export d'une relation binaire en 2017. Ainsi, SRS pourrait être intégré au LAS en 2018.

Bilan

La table ci-dessous donne une idée de la charge de travail pour la réalisation d'un importeur FL, de SRS et de l'extension de RichGraph vers la version 1.0.

Bilan et répartition du travail d'implémentation

	Importeur pour FL	Extension de RichGraph 0.1 vers RichGraph 1.0	Développement de SRS
Nombre de lignes de code écrites	environ 3000	environ 2500	environ 6000
Nombre de lignes de code écrites pour les tests unitaires	environ 1000	environ 300	environ 2000
Nombre de classes créées ou modifiées	4 classes créées ou modifiées dans le framework du LAS	13 classes créées ou modifiées dans le framework du LAS	23 classes créées pour SRS

5.3.2. Perspectives pour SRS

Réutilisation d'éléments de modules BaboukWeb

SRS pourrait être adapté comme un module de l'interpréteur BaboukWeb. Ainsi, SRS pourrait exploiter des éléments d'autres modules et vice versa sans passer par les APIs de l'interpréteur BaboukWeb et de ses modules. En effet, l'interpréteur BaboukWeb est capable i) de sélectionner le parser approprié pour les langages qu'il peut interpréter, et ii) sélectionner le module approprié pour interpréter l'AST ou l'ASG renvoyé par le parser. Par exemple, SRS (via l'interpréteur BaboukWeb) pourrait exploiter les éléments de FunctionalInterpreter pour interpréter des appels de fonctions écrits en functionalP, comme dans la requête ci-dessous.

```
? [a car ?c
  has_attribute: a ^(cost value: "get_car_cost(?c)" _[language: FunctionalP],
                    unit: a euro)]
//SRS peut déléguer l'interprétation de "get_car_cost(?c)" à FunctionalInterpreter.
```

Utilisation dans LAS

Au début de l'année 2016, Anil Cassam-Chenaï a estimé que les développements nécessaires pour réduire *le temps de calcul pour la résolution des requêtes via SRS* seraient trop complexes et trop longs pour l'industrialisation à court terme i) du LAS et ii) de la gamme de produits utilisant LAS. Aussi, c'est un résolveur différent, plus simple, développé par Anil Cassam-Chenaï et quelques employés logiCells, qui sera utilisé dans LAS. SRS pourra être complété après la fin de ma thèse, entre 2017 et la fin de l'année 2018.

6. Import, export et traduction de LRCs exploitant KRLO

Outre des modèles et notations de LRCs, KRLO est la première ontologie à inclure des règles et des fonctions spécifiant des méthodes par défaut pour importer, traduire et exporter des RCs. La section “[6.1. Export depuis un modèle abstrait vers un modèle concret](#)” présente les fonctions d'export. La section “[6.2. Traduction entre éléments abstraits](#)” présente les règles de traduction. La section “[6.3. Import depuis un modèle concret vers un modèle abstrait](#)” présente les fonctions d'import.

6.1. Export depuis un modèle abstrait vers un modèle concret

La section “[6.1.1. Spécifications d'exports](#)” présente la fonction d'export par défaut de KRLO et une fonction d'export alternative que j'ai écrite et qui est basée sur `Structure_map`. Ces deux fonctions ont une complexité polynomiale car elles effectuent des comparaisons entre deux arbres sans retour sur trace [backtracking]. En effet, dans KRLO, les EAs sont structurés via des relations *opérateur* et *argument* (cf. section [4.1.1.2](#)) et ont donc une structure d'arbre. La section “[6.1.2. Validation, mise en œuvre et exemples](#)” détaille la façon dont des requêtes types utilisant des fonctions basées sur `Structure_map` sont résolues.

6.1.1. Spécifications d'exports

Cette section présente les deux spécifications d'export suivantes.

- La spécification du processus d'export par défaut dans KRLO écrite par Philippe Martin.
- Une spécification utilisant des fonctions basées sur `Structure_map` que j'ai écrites.

Rappels et références vers les sections précédentes

Dans cette thèse, un export est un processus qui prend en entrée des EAs et a en sortie des éléments textuels ou graphiques (cf. section “[3.2. Export de connaissances](#)”). Ainsi, l'export ne transforme pas les EAs mais permet d'extraire une présentation de ces EAs.

KRLO est une ontologie de LRCs qui organise des types d'EAs et des spécifications de présentation pour ces types d'EAs. Ainsi, via KRLO, la présentation d'un EA peut être dérivée des spécifications de présentation de cet EA. Les sections “[4.1.3.2. Spécifier des notations dans KRLO_2014](#)” et “[4.1.4. Spécifications de notations d'éléments de modèles abstraits dans KRLO_2015+](#)” présentent la façon dont ces spécifications sont écrites dans KRLO_2014 et KRLO_2015+.

6.1.1.1. Fonctions primitives utilisées par les spécifications d'export

Les fonctions d'export décrites dans les sections [6.1.1.2.](#) et [6.1.1.3.](#) exploitent les fonctions présentées ci-après. Dans ces fonctions, tous les types utilisés sont de Philippe Martin, l'espace de nommage `pm` est donc laissé implicite.

```

//fc_spec prend en entrée ?cePartsSpecs et ?notationSet et stocke ces entrées dans une liste
fc_spec ~[?cePartsSpecs, ?notationSet] := List(?cePartsSpecs, ?notationSet);

//f_OP prend en entrée ?ae et a en sortie une liste composée du type de relation r_operator et de ?ae
f_OP ~[?ae] := List(r_op, ?ae); //r_op réfère à r_operator
//f_ARG fonctionne comme f_OP mais avec le type de relation r_argument
f_ARG ~[?ae] := List(r_arg, ?ae); //r_arg réfère à r_argument
//f_ARGS fonctionne comme f_OP mais avec le type de relation r_arguments
f_ARGS ~[?ae] := List(r_args, ?ae); //r_args réfère à r_arguments

//f_OP_from a en sortie une liste composée du type de relation r_operator et de la destination de l'une des
//relations suivantes depuis ?ae : rc_operator_name ou r_operator (par défaut de relation rc_operator_name).
f_OP_from ~[?ae] := List(r_op, (?ae.rc_operator_name ) ? ?ae.rc_operator_name : ?ae.r_op);

//f_ARG_from a en sortie une liste composée du type de relation r_argument et de la destination d'une
relation r_argument depuis ?ae.
f_ARG_from ~[?ae] := List(r_arg, ?ae.r_arg);

//f_ARGS_from fonctionne comme f_ARG_from avec le type de relation r_arguments
f_ARGS_from ~[?ae] := List(r_args,?ae.r_args);

//?space_specs (ci-dessous) réfère aux spécifications pour les caractères blancs.
//Dans la plupart des LRCs, ces caractères sont des espaces " " ou des retour à la ligne.
f_ARGS ~[?ae, ?space_specs] := List(r_args, ?ae, ?space_specs);
f_ARGS_from ~[?ae, ?space_specs] := List(r_args, ?ae.r_args, ?space_specs);

```

6.1.1.2. Procédure d'export par défaut spécifiée dans KRLO

Philippe Martin a écrit la fonction d'export par défaut dans KRLO. Cette fonction – *fc_parts* – et ses sous-fonctions sont décrites ci-dessous en FL. KRLO ayant majoritairement été écrite par Philippe Martin, les types créés par Philippe Martin ne sont pas préfixés par *pm*, l'espace de nommage implicite reste *krlo*. Le préfixe *pm* n'a été précisé que pour éviter certaines ambiguïtés, par exemple, pour distinguer *pm#Structure_map* de *jb#Structure_map*. Tous les types que j'ai créés sont préfixés par *jb*.

```

fc_parts ~[AE ?ae, ?notationSet, ?spaceSpecs] //AE ?ae est une garde pour ?ae
//?notationSet et ?spaceSpecs n'ont pas de garde
:= fc_apply_specs _(?ae, fc_find_specs _(?ae,?notationSet),?spaceSpecs);

fc_parts ~[AE ?ae, ?notationSet]
:= fc_apply_specs _(?ae, fc_find_specs _(?ae,?notationSet),
fc_find_other_space_specs _(?ae,?notationSet) );

fc_AEs_parts ~[?List_of_AE, ?notationSet, ?spaceSpecs]
:= if_then_else _([?List_of_AE = List(%censp_1st | %censp_rest)], //si ?List_of_AE n'est pas vide
/*then*/ fc_append_with_spaces _(fc_parts _(%censp_1st,?notationSet)
fc_AEs_parts _(%censp_rest,?notationSet)
?spaceSpecs),
/*else*/ "" );

fc_AEs_parts ~[?List_of_AE, ?notationSet]
:= if_then_else _([?List_of_AE = List(%censp_1st | %censp_rest)],
/*then*/ fc_append_with_spaces _(fc_parts _(%censp_1st,?notationSet),
fc_AEs_parts _(%censp_rest,?notationSet),
fc_find_other_space_specs _(?ae,?notationSet)),
/*else*/ "" );

```

```

//Rappel (et exemple) sur la spécification de notation d'un EA :
//f_in(AE RIF) ?rifAE rc_spec: List( List(f_OP_from(?rifAE) "(" f_ARGS_from(?rifAE) ")"),
//
//                               List(RIF-PS) )
fc_find_specs ~[AE ?ae, ?notationSet] //renvoie le noeud référé par ?specs
:= (the List ?specs r_1st_arg of: (a List r_2nd_arg: ?notationSet,
//                               /*most direct*/rc_spec of: ?ae ));
fc_find_other_space_specs ~[AE ?ae, ?notationSet] //renvoie le noeud référé par ?s_specs
:= (the List ?s_specs r_2nd_arg of: (a List r_2nd_arg: ?notationSet,
//                               /*most direct*/rc_sSpec of: ?ae ));

//?ceNonSpacePartsSpec ci-dessous est une liste renvoyée par fc_find_specs.
//Par exemple, on peut avoir ?ceNonSpacePartsSpec = List(f_OP_from(?rifAE) "(" f_ARGS_from(?rifAE) ")")
fc_apply_specs ~[AE ?ae, ?ceNonSpacePartsSpecs, ?ceSpacePartsSpecs, ?notationSet]
:= if_then_else_( [?ceNonSpacePartsSpec = List(%censp_1st | %censp_rest)],
// *then*/ fc_append_with_spaces_(fc_apply_spec_(?ae,%censp_1st,?notationSet),
//                               fc_apply_specs_(?ae,%censp_rest),
//                               ?ceSpacePartsSpecs ),
// *else*/ List() );

//?ceNonSpacePart ci-dessous est un élément de ?ceNonSpacePartsSpec ci-dessus
fc_apply_spec ~[AE ?ae, ?ceNonSpacePart, ?notationSet]
:= if_then_else_( [?ceNonSpacePart /^^ String], //si ?ceNonSpacePart n'est pas de type String
// *then*/List(?ceNonSpacePart),
// *else*/if_then_else_( [r_1st_arg_(?ceNonSpacePart) = r_args],//si ?ceNonSpacePart est f_ARGS_from
// *then*/if_then_else_( [r_3rd_arg_(?ceNonSpacePart) = List()],//si ?ceNonSpacePart n'a pas
//                               //de spécifications d'espaces
// *then*/fc_AEs_parts_(r_2nd_arg_(?ceNonSpacePart),?notationSet),
// *else*/fc_AEs_parts_(r_2nd_arg_(?ceNonSpacePart),?notationSet,
//                               r_3rd_arg_(?ceNonSpacePart) ) ),
// *else*/fc_parts_(r_2nd_arg_(?ceNonSpacePart),?notationSet) );

fc_append_with_spaces ~[?str1, ?str2, ?ceSpacePartsSpecs/*e.g.,= " */]
:= if_then_else_( [?ceSpacePartsSpecs /^^ String],
// *then*/fc_append_(?str1,?ceSpacePartsSpecs,?str2),
// *else*/fc_append_(?str1," ",?str2) );

```

La requête suivante en FC permet d'exporter l'EA structuré *a_phrase* dans la notation RIF-PS.

```
? [fc_parts_(a_phrase, RIF-PS, " ") ]
```

La section “[6.1.2. Validation, mise en œuvre et exemples](#)” expose quelques exemples d'EAs structurés et montre comment les fonctions présentées ci-dessus permettent de dériver des ECs à partir de ces EAs.

6.1.1.3. Fonction d'export spécifiée via Structure_map

J'ai écrit une fonction d'export avec une fonction basée sur `jb#Simple_graph_map` (et donc `jb#Structure_map` et donc aussi `pm#Structure_map`). Cette fonction d'export est décrite ci-dessous en FL. Pour faciliter la lecture du code, j'ai supprimé les traitements des ECs d'espacement. Ces traitements sont identiques à ceux présentés dans la fonction d'export par défaut de la section précédente (voir `?ceSpacePartsSpecs` et `?spaceSpecs` ci-dessus).

```

jb#fc_parts ~[AE ?ae, ?notationSet]
:= [ [jb#fc_apply_specs_(fc_find_specs_(?ae, ?notationSet), ?notationSet) ] ];

jb#fc_apply_specs ~[?ceNonSpacePartsSpecs, ?notationSet]
:= [ [jb#Simple_graph_map_(?ceNonSpacePartsSpecs, //rappel : jb#Structure_map > jb#Simple_graph_map;
// ^ (jb#fc_apply_spec input: ?notationSet),
// jb#fc_append) ] ];

```



```

jb#fc_apply_spec ~[?ceNonSpacePart, ?notationSet]
:= [ [if_then_else _([?ceNonSpacePart /^^ String],
/*then*/ ?ceNonSpacePart,
/*else*/ if_then_else _([r_1st_arg _(?ceNonSpacePart) = r_args],
/*then*/ jb#Simple_graph_map _(r_2nd_arg _(?ceNonSpacePart),
^(jb#fc_parts input: ?notationSet),
jb#fc_append),
/*else*/ jb#fc_parts _(r_2nd_arg _(?ceNonSpacePart), ?notationSet) ) ) ] ];

jb#fc_append ~[?ce_or_list] = jb#fc_list_containing_CE_or_list,
pm#output: 1 jb#List ?result,
:= [ [?result head: ?ce_or_list,
tail: 1 jb#List] ];

```

La requête suivante en FC permet d'exporter l'EA structuré *a_phrase* dans la notation RIF-PS.

```
? [ jb#Simple_graph_map _(a_phrase, ^(jb#fc_parts input: RIF-PS) ) ]
```

La section “6.1.2. Validation, mise en œuvre et exemples” expose quelques exemples d'EAs structurés et montre comment les fonctions présentées ci-dessus permettent de dériver des ECs à partir de ces EAs.

6.1.1.4. Comparaison des spécifications

Comparaison des spécifications de *jb#fc_apply_specs* et *fc_apply_specs*

L'extrait ci-dessous permet de comparer les fonctions *jb#fc_apply_specs* et *fc_apply_specs*. Contrairement à *fc_apply_specs*, *jb#fc_apply_specs* est définie via *jb#Simple_graph_map* (et donc *Structure_map*). La section “5.1.2. Réutilisation de travaux précédant le changement de sujet de thèse” présente les avantages de l'utilisation de fonctions basées sur *Structure_map* pour programmer. Pour rappel, ces avantages – par rapport aux fonctions qui ne sont pas basées sur *Structure_map* – sont : une meilleure réutilisabilité, une meilleure lisibilité et une meilleure maintenabilité [maintainability]. L'extrait ci-dessous illustre ces avantages.

```

jb#fc_apply_specs ~[?ceNonSpacePartsSpecs, ?notationSet]
:= [ [jb#Simple_graph_map _(?ceNonSpacePartsSpecs,
^(jb#fc_apply_spec input: ?notationSet),
jb#fc_append) ] ];

fc_apply_specs ~[?ceNonSpacePartsSpecs, ?notationSet]
:= if_then_else _([?ceNonSpacePartsSpecs = List(%censp_1st | %censp_rest)],
/*then*/ fc_append_with_spaces _(fc_apply_spec _(%censp_1st,?notationSet),
fc_apply_specs _(%censp_rest,?notationSet) ),
/*else*/ List() );

```

Dans *jb#fc_apply_specs*, *jb#Simple_graph_map* permet de réutiliser une fonction de parcours de structure existante. Cette fonction de parcours est déduite du noeud référé par *?ceNonSpacePartsSpecs*. *jb#fc_apply_specs* peut donc être réutilisée pour toute structure pour laquelle une fonction de parcours par défaut a été spécifiée dans une ontologie.

fc_apply_specs, ne réutilise pas une fonction de parcours de structure existante. Un utilisateur doit re-spécifier ce parcours et donc également prédéfinir la structure parcourue. Ainsi, *fc_apply_specs* est moins lisible, réutilisable et maintenable [maintainability] que *jb#fc_apply_specs*. En effet, i) *fc_apply_specs* ne peut être ré-utilisée que pour des listes, ii) pour comprendre *fc_apply_specs*, un utilisateur doit lire et comprendre le parcours de liste spécifié pour cette fonction, et iii) comme une spécification de parcours de liste ne peut pas être ré-utilisée, chaque spécifications de parcours de liste doit être maintenue indépendamment des autres.

Comparaison des spécifications de `jb#fc_apply_spec` et `fc_apply_spec`

L'extrait ci-dessous permet de comparer les fonctions `jb#fc_apply_spec` et `fc_apply_spec`. Les différences entre ces deux fonctions sont les mêmes qu'entre `jb#fc_apply_specs` et `fc_apply_specs`.

```
jb#fc_apply_spec ~[?ceNonSpacePart, ?notationSet]
:= [ [if_then_else _([?ceNonSpacePart /^^ String],
/*then*/?ceNonSpacePart,
/*else*/if_then_else _([r_1st_arg _(?ceNonSpacePart) = r_args],
/*then*/jb#Simple_graph_map _(r_2nd_arg _(?ceNonSpacePart),
^(jb#fc_parts input: ?notationSet),
jb#fc_append),
/*else*/jb#fc_parts _(r_2nd_arg _(?ceNonSpacePart), ?notationSet) ) ) ] ];

fc_apply_spec ~[AE ?ae, ?ceNonSpacePart, ?notationSet]
:= if_then_else( [?ceNonSpacePart /^^ String],
/*then*/List(?ceNonSpacePart),
/*else*/if_then_else _([r_1st_arg _(?ceNonSpacePart) = r_args],
/*then*/fc_AEs_parts _(r_2nd_arg _(?ceNonSpacePart),?notationSet),
/*else*/fc_parts _(r_2nd_arg _(?ceNonSpacePart),?notationSet) ) );
```

Les avantages concernant l'utilisation d'une fonction `jb#Simple_graph_map` sont les mêmes que ceux donnés dans la sous-section précédente "Comparaison des spécifications de `jb#fc_apply_specs` et `fc_apply_specs`". Les inconvénients concernant la non utilisation d'une fonction `jb#Simple_graph_map` sont également expliqué dans cette même sous-section. Un inconvénient supplémentaire est que l'utilisateur doit ici (dans `fc_apply_spec`) utiliser une fonction supplémentaire (c'est à dire, `fc_AEs_parts`) pour réaliser le parcours.

Comparaison des spécifications de `jb#fc_append` et `fc_append_with_spaces`

L'extrait ci-dessous permet de comparer les fonctions `jb#fc_append` et `fc_append_with_spaces`. Contrairement aux autres fonctions qui doivent être interprétées comme des requêtes lors de l'export, `jb#fc_append` et `fc_append` sont des fonctions qui génèrent des valeurs ou des références. .

```
jb#fc_append ~[?ce_or_list] = jb#fc_list_containing_CE_or_list,
pm#output: 1 jb#List ?result,
:= [ [?result head: ?ce_or_list,
tail: 1 jb#List] ];

fc_append_with_spaces ~[?str1, ?str2, ?ceSpacePartsSpecs/*e.g.,= " */]
:= if_then_else( [?ceSpacePartsSpecs /^^ String],
/*then*/fc_append _(?str1,?ceSpacePartsSpecs,?str2),
/*else*/fc_append _(?str1," ",?str2) );
```

Comme toutes les fonctions que j'ai spécifié pour l'export, la définition de `jb#fc_append` décrit un graphe. Ce graphe peut être utilisé par i) TPatternTraversal, le composant utilisé par SRS pour reconnaître des spécialisations de graphe (cf. Figure 5.3.), ou ii) TPatternBuilder, le composant utilisé par SRS pour construire des spécialisations de graphe (cf. Figure 5.4.). Par exemple, `jb#fc_append` est utilisée pour générer une liste dans `jb#fc_apply_specs` mais est utilisée pour chercher une liste contenant un CE ou une sous-liste dans "`jb#Simple_graph_map _(?list, jb#fc_append)`".

Contrairement à `jb#fc_append`, `fc_append` doit être programmée ou spécifiée comme une fonction génératrice. Elle peut être utilisée de façon indirecte pour la recherche dans les deux cas suivants.

- Le résultat de `fc_append` est utilisé dans la partie *graphe requête* d'une requête. Par exemple, le résultat de `fc_append` est utilisée pour la recherche dans la requête suivante "`?car [?car name: fc_append _("JB", "_", "car1")]`".
- Le résultat de `fc_append` est utilisé dans une fonction de recherche. Par exemple, le résultat de `fc_append` est utilisée pour la recherche dans la fonction suivante "`fc_find_Type_with_that_name _(fc_append _("JB", "_", "car1"))`".

6.1.2. Validation, mise en œuvre et exemples

1) Validation et mise en oeuvre d'un outil utilisant KRLO pour exporter des AEs vers des notations

J'ai développé un outil de traduction et d'export que j'appelle kr-translation. kr-translation utilise KRLO_2014. Un serveur Web pour kr-translation est accessible à l'adresse suivante <http://kr-translation.logicells.net/>. Dans la version de KRLO_2014 utilisée par kr-translation, trois LRCs sont représentés ; leurs spécifications sont accessibles depuis la page du serveur et peuvent être modifiées par les utilisateurs.

kr-translation n'utilise pas Structure_map. Pour chaque fonction de KRLO_2014 spécifiant un EC, j'ai implémenté une fonction sous Delphi. Ces fonctions exploitent l'API de jb#RichGraph (version 1.0) pour i) réaliser des traductions entre des EAs des modèles prédéfinis dans l'interface du site web, ii) extraire des ECs à partir des spécifications de KRLO_2014 et les ordonner dans une chaîne de caractères. kr-translation est donc programmé pour traduire/exporter des phrases depuis un LRC vers un autre via KRLO_2014.

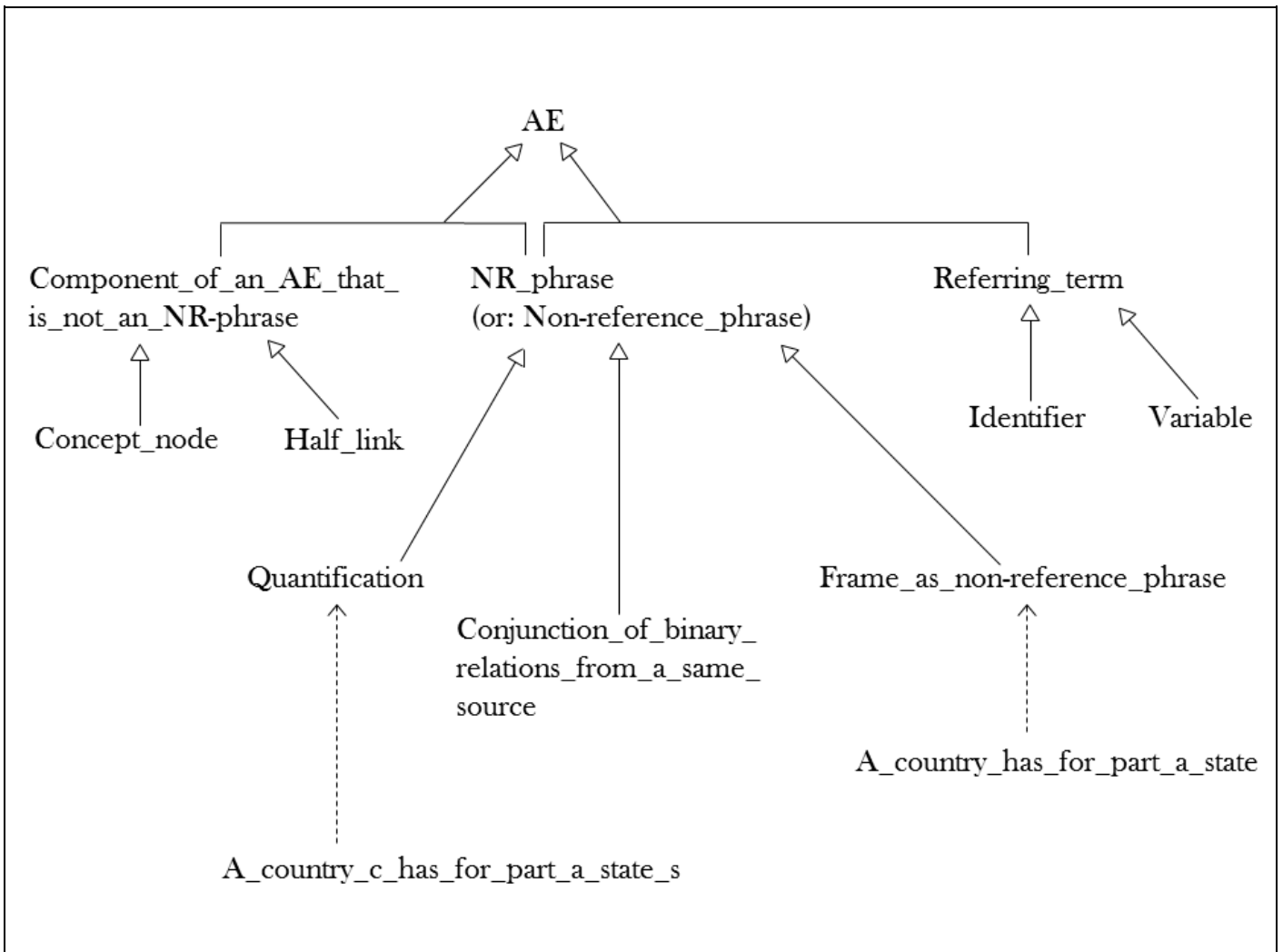
kr-translation peut traduire et exporter un noeud relation en 0.15 seconde. L'utilisation intensive de listes chaînées dans les fonctions de traduction/export implémentées sous Delphi est une des raisons pour ce temps de calcul élevé. Contrairement à SRS, kr-translation n'est pas un résolveur de requêtes générique mais un outil programmé pour des traductions/exports. Ainsi, sans modification de leur code, SRS peut utiliser KRLO_2015+ ou KRLO_2014 alors que kr-translation ne peut utiliser que KRLO_2014. Il est plus difficile (cf. [2.2. Difficulté des tâches de modélisation et/ou de programmation pour le partage de connaissances](#)) de faire évoluer le code de kr-translation – par exemple, afin que cet outil puisse exploiter KRLO_2015+ – que de modifier une requête à envoyer à SRS. De plus, l'utilisation de SRS n'est pas restreinte à l'export vers des notations. C'est pourquoi j'ai fait et je fais encore (pour l'année 2017) le choix de ne pas faire évoluer kr-translation et de ne pas l'optimiser davantage. S'il s'avère que SRS ne peut pas être suffisamment optimisé pour être utilisable à grande échelle pour la traduction de LRCs, j'envisagerai de revenir sur ce choix.

Contrairement aux fonctions implémentées pour kr-translation, les fonctions d'export présentées dans la section "[6.1.1. Spécifications d'exports](#)" ne peuvent pas être utilisées pour des traductions entre des AEs. Les règles spécifiant de telles traductions sont présentées dans les sections "[6.2.1. Traduction entre Frame et Conjunction_of_links_from_a_same_source](#)" et "[6.2.2. Traduction entre une relation binaire et une relation n-aire](#)". La sous-section "(2) Absence de traduction entre des EAs" de la section "[4.2.1. Problèmes identifiés](#)" explique la nécessité des traductions entre EAs. La sous-section "(2)" ci-dessous illustre une extraction (ou une déduction) d'ECs à partir de spécifications de notations de KRLO_2015+.

2) Exemples d'exports exploitant KRLO_2015+ avec mes spécifications de notations et les fonctions d'export présentées dans la section "[6.1.1.3. Fonction d'export spécifiée via Structure_map](#)"

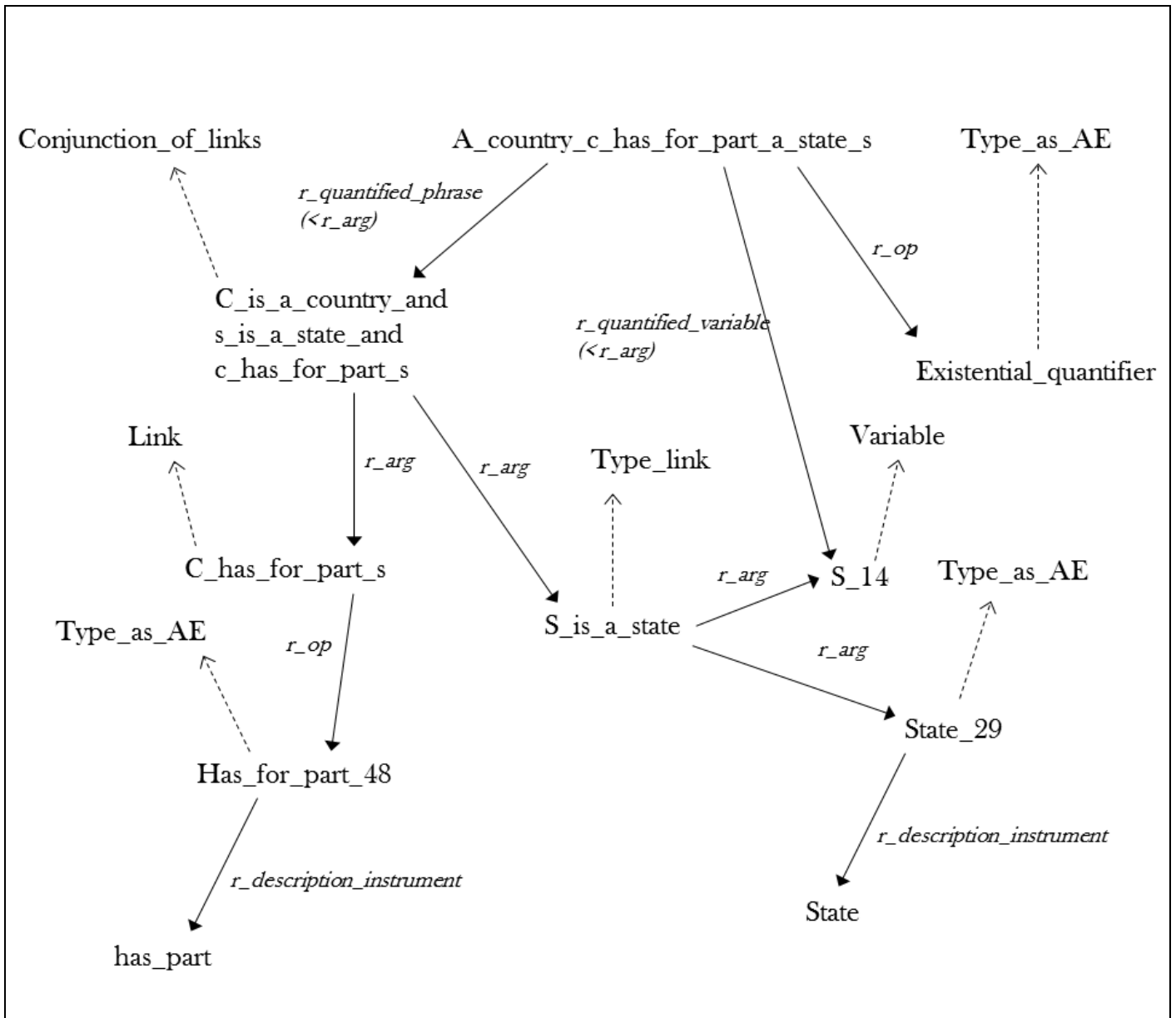
Cette section explicite les spécifications de notations qui sont utilisées pour exporter deux EAs – a_country_has_for_part_a_state et a_country_c_has_for_part_a_state_s – représentant "some country has for part a state". L'AE a_country_has_for_part_a_state suit un modèle basé sur un graphe (cf. Graph-based_model dans la section "[4.1.2.2. Ontologies de modèles particuliers](#)"). L'AE a_country_c_has_for_part_a_state_s suit un modèle qui n'est pas basé sur un graphe (cf. Not_graph-based_model dans la section [4.1.2.2.](#)). La figure suivante présente i) des *généralisations* pour a_country_has_for_part_a_state et a_country_c_has_for_part_a_state_s, et ii) quelques uns des types d'AEs dont les spécifications de notations seront utilisées pour l'export.

Figure 6.1. Types de deux instances en pm#UML.



La figure suivante présente les relations entre `a_country_c_has_for_part_a_state_s` et certaines de ses parties.

Figure 6.2. Parties de `A_country_c_has_for_part_a_state_s` en `pm#UML`.



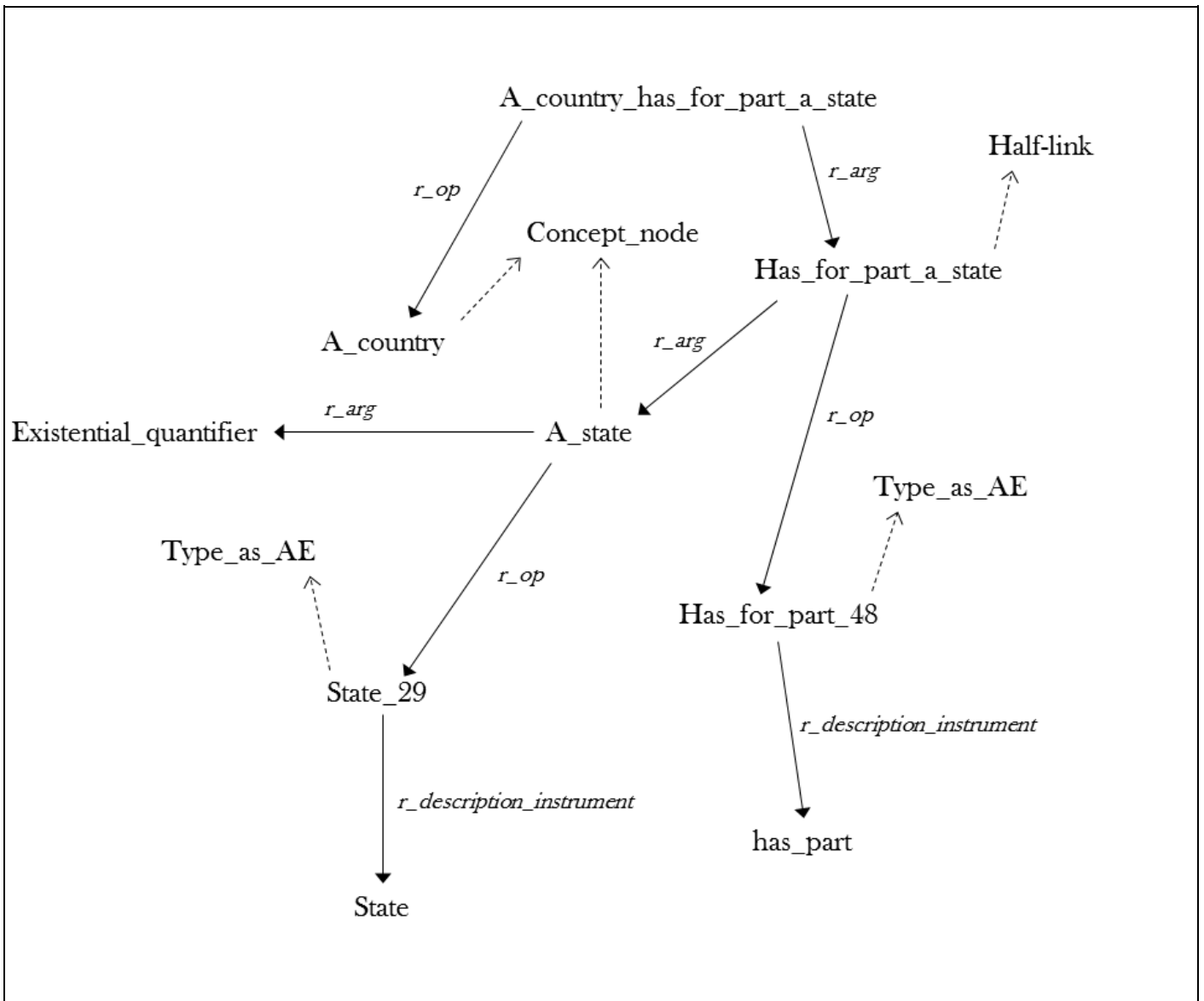
La table suivante présente i) les spécifications appliquées à `a_country_c_has_for_part_a_state_s` ou ses parties pendant l'export vers la notation RIF-PS, et ii) le résultat de l'application de cette spécification sur l'AE en début de ligne. La première ligne de la table présente `a_country_c_has_for_part_a_state_s`, les suivantes présentent ses parties.

Élément abstrait	Spécification appliquée	Résultat (les guillemets " " délimitent les ECs, les virgules séparent les éléments de la liste)
<code>a_country_c_has_for_part_a_state_s</code>	<pre>f_in_(Quantification, RIF) ?rifAE rc_spec: fc_spec_(List(f_OP_from_(?rifAE) f_VARS_from_(?rifAE) "(" f_PHRASE_from_(?rifAE) ")"), List(RIF-PS))</pre>	Existential_quantifier, c, s, "(", c_is_a_country_and_..., ")"
Existential_quantifier	<p>/*"Exists" est le CE par défaut pour Existential_quantifier. Ainsi, il n'est pas nécessaire de spécifier une notation pour Existential_quantifier dans pour la notation RIF-PS. Une spécification de notation pour Existential_quantifier dans le modèle de RIF et pour la notation RIF-PS est tout de même donnée ci-dessous comme exemple.*/</p> <pre>f_in_(Existential_quantifier, RIF) rc_spec: fc_spec_(List("Exists"), List(RIF-PS))</pre>	"Exists"
c	<p>/*Comme pour Existential_quantifier, spécifier la notation d'une variable dans la notation RIF-PS n'est pas nécessaire. La notation par défaut peut être réutilisée.*/</p>	"?c"
<code>c_is_a_country_and_...</code>	<pre>f_in_(AE, RIF) ?rifAE //rappel : //AE > Conjunction_of_links rc_spec: fc_spec_(List(f_OP_from_(?rifAE) "(" f_ARGS_from_(?rifAE) ")"), List(RIF-PS))</pre>	"And (" c_is_a_country s_is_a_state, c_has_for_part_s, ")"
<code>c_is_a_country</code>	<pre>Type_link@RIF ?l rc_spec: fc_spec_(List(?l.r_link_source "#" ?l.r_link_destination), List(RIF-PS))</pre>	"?c#Country"
<code>c_has_for_part_s</code>	<pre>f_in_(AE, RIF) ?rifAE //rappel : AE > Link rc_spec: fc_spec_(List(f_OP_from_(?rifAE) "(" f_ARGS_from_(?rifAE) ")"), List(RIF-PS))</pre>	"has_part(?c ?s)"

Le résultat de l'export de `a_country_c_has_for_part_a_state_s` dans la notation RIF-PS est "Exists ?c ?s (and (?c#Country ?s#State has_part(?c ?s)))".

La figure suivante présente les relations entre `a_country_has_for_part_a_state` et certaines de ses parties.

Figure 6.3. Parties de A_country_has_for_part_a_state en pm#UML.



La table suivante présente i) les spécifications appliquées à `a_country_has_for_part_a_state` ou ses parties pendant l'export vers la notation de FL, et ii) le résultat de l'application de cette spécification l'AE en début de ligne. La première ligne de la table présente `a_country_has_for_part_a_state`, les suivantes présentent ses parties.

Élément abstrait	Spécification appliquée	Résultat (les guillemets " " délimitent les ECs, les virgules séparent les éléments de la liste)
<code>a_country_has_for_part_a_state</code>	<code>f_in_(AE, FL) ?f_lAE rc_spec: fc_spec_(List(f_OP_from_(?f_lAE) f_ARGS_from_(?f_lAE, ",") ";"), List(FL_notation))</code>	<code>a_country, part_a_state, ";"</code>
<code>a_country</code>	<code>f_in_(Concept_node, FL) ?c rc_spec: fc_spec_(List(f_ARG_from_(?c) f_OP_from_(?c)), List(FL_notation))</code>	Existential_quantifier, "Country"
Existential_quantifier	<code>f_in_(Existential_quantifier, FL) rc_spec: fc_spec_(List("some"), List(FL_notation))</code>	"some"
<code>part_a_state</code>	<code>f_in_(Half-link, FL) ?hl rc_spec: fc_spec_(List(f_OP_from_(?hl) ";" f_ARGS_from_(?hl)), List(FL_notation))</code>	"has_part: some State"

Le résultat de l'export de `a_country_has_for_part_a_state` dans la notation de FL est "some Country has_part: some State".

Pour un AE, un utilisateur de KRLO peut spécifier plusieurs présentations, une pour chaque notation spécifiée. Par exemple, il peut spécifier une notation KIF pour un AE du modèle RIF. Cet utilisateur peut aussi inventer de toutes pièces une notation ou créer une variante pour une notation existante. Par exemple, si j'aimais la notation "{some Country has_part->some State}", je pourrais la spécifier via la table suivante. Cette table a la même organisation que les tables précédentes. Elle présente les spécifications appliquées à `a_country_has_for_part_a_state` ou ses parties pendant l'export vers `my_beloved_notation`. `my_beloved_notation` est une notation que j'ai inventée. Elle est inspirée des notations de JSON-LD, FL et de la notation RIF-PS. Comme son nom ne l'indique pas, elle n'est pas représentative de mes préférences personnelles.

Élément abstrait	Spécification appliquée	Résultat (les guillemets " " délimitent les ECs, les virgules séparent les éléments de la liste)
<code>a_country_has_for_part_a_state</code>	<code>f_in_(AE, FL) ?f_lAE rc_spec: fc_spec_(List("{" f_PARTS_from_(?f_lAE, ",") "}"), List(my_beloved_notation))</code>	<code>"{ " a_country ", " part_a_state, }"</code>
<code>part_a_state</code>	<code>f_in_(Half-link, FL) ?hl rc_spec: fc_spec_(List(f_OP_from_(?hl) "->" f_ARG_(?hl.r_link_destination)), List(my_beloved_notation))</code>	"has_part->some State"

6.1.3. Complétude

Définitions

Complétude d'une fonction d'export : tout EA en entrée est associé à un EC en sortie, en supposant que l'ontologie de LRCs en paramètre soit complète.

Complétude des fonctions d'export présentées dans la section 6.1.1

Pour les raisons présentées ci-dessous, si les descriptions du modèle en entrée et des spécifications de notation sont complètes alors les fonctions d'export présentées dans la section 6.1.1 sont complètes (cf. section "4.4, Complétude de KRLO").

1. Ces fonctions d'export prennent en entrée i) une notation N, et ii) un EA structuré et ont en sortie un EC dérivé de l'application des spécifications de présentation de l'EA en entrée et de ses sous-parties.
2. Pour l'EA en entrée et ses sous-parties, une seule spécification de présentation pour la notation N est spécifiée.
3. Via une spécification de notation chaque EA en entrée est associé à un EC en sortie.

6.2. Traduction entre éléments abstraits

Philippe Martin a écrit des règles en FL pour des traductions entre des EAs. Les traductions ainsi spécifiées sont listées ci-dessous. Les traductions dont j'ai découvert la nécessité sont en italiques. Ce sont les seules traductions qui sont présentées dans les sections suivantes. Les autres sont décrites dans [d_KRLmodelsTranslation.html](#).

- Traduction entre Property et `rdf:Predicate`
- *Traduction entre `Frame_as_NR-phrase` et `Conjunction_of_binary-relations_from_a_same_source` (et donc également entre `Half-link` et `Binary_relation`)*
- *Traduction entre `Binary_relation` et `Non-binary_relation`*
- Traduction entre Function et `Functional_relation`

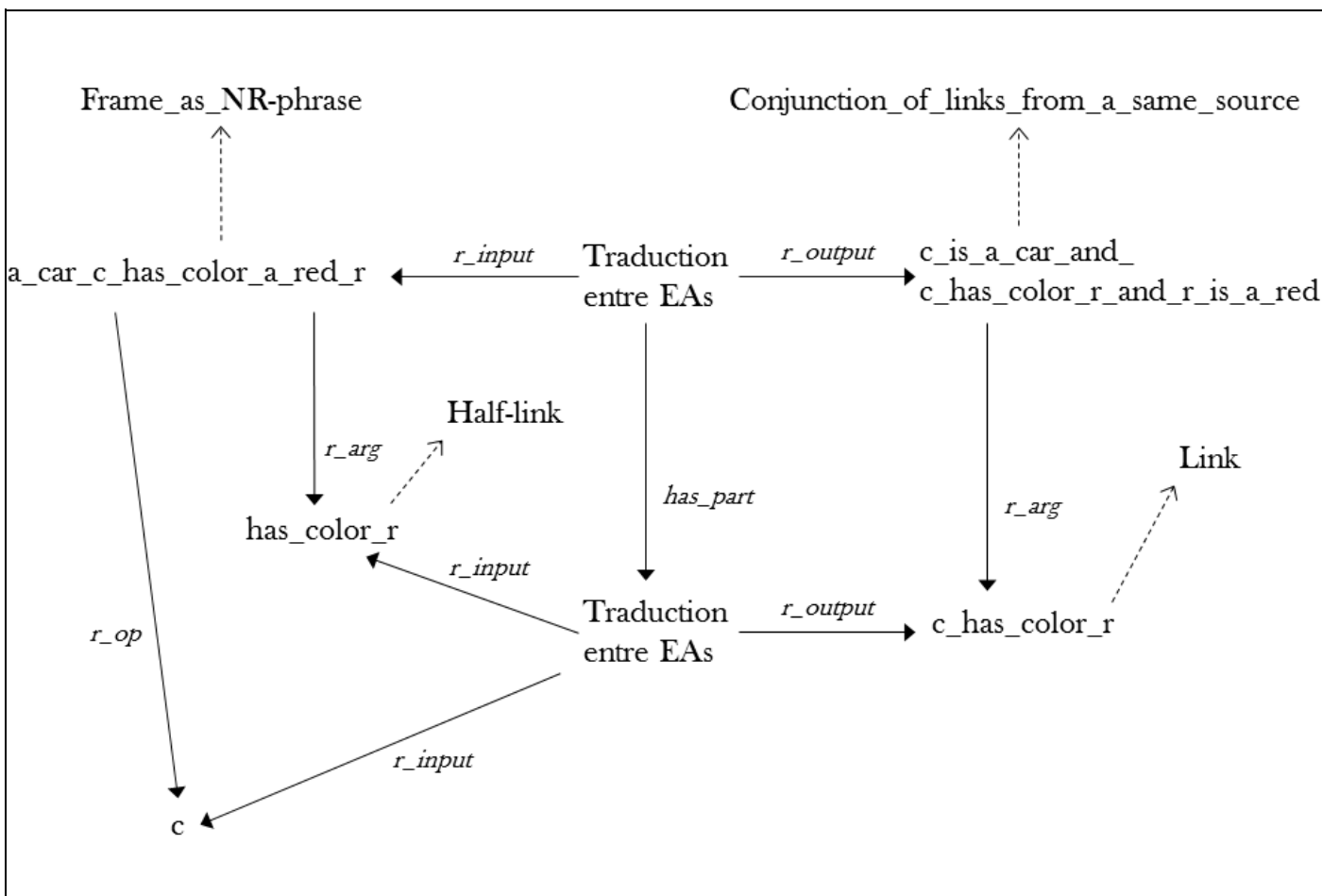
Contrairement à ce que nous pensions, la traduction entre EAs ne peut pas être spécifiée en RIF-BLD. Peu de moteurs d'inférences peuvent gérer une expressivité supérieure à celle de RIF-BLD. En effet, i) dans certaines règles, un quantificateur existentiel est nécessaire dans la partie conclusion de la règle, ce qui est interdit en RIF-BLD, et ii) certaines règles sont des méta-règles, c'est à dire, des règles sur des règles, ce qui est également interdit en RIF-BLD. Comme une expressivité supérieure à RIF-BLD est nécessaire pour la traduction entre EAs, la spécification de ces traductions via des règles n'est pas avantageuse. C'est pourquoi Philippe Martin convertira ces règles en définitions plus expressives dans un avenir proche.

6.2.1. Nécessité d'une traduction entre EAs

Pourquoi une traduction entre EAs est-elle nécessaire ?

Des traductions entre EAs sont nécessaires car deux EAs peuvent être des instruments de description d'un même contenu de description sans pour autant être comparables (cf. comparaison sémantique). En effet, ils peuvent être structurés différemment. Par exemple, dans la figure ci-dessous, les EAs `a_car_c_has_color_a_red_r` et `c_is_a_car_and_c_has_color_r_and_r_is_a_red` sont des instruments de description pour le contenu "une voiture rouge". Les types de ces EAs sont respectivement `Frame_as_NR-phrase` et `Conjunction_of_links_from_a_same_source`. `a_car_c_has_color_a_red_r` et `c_is_a_car_and_c_has_color_r_and_r_is_a_red` sont structurés différemment, `a_car_c_has_color_a_red_r` a pour argument des EAs de type `Half-link` et `c_is_a_car_and_c_has_color_r_and_r_is_a_red` a pour argument des EAs de type `Link`.

Figure 6.4. Traduction entre deux EAs incomparables en `pm#UML`.



Comme le montre l'extrait ci-dessous, `Frame_as_NR-phrase` et `Conjunction_of_links_from_a_same_source` ne sont pas comparables.

```
NR_formula
  > partition { (Composite_formula > Conjunction_of_links_from_a_same_source)
                (NR_atomic_formula > Frame_as_NR-phrase)
              };
```

Quelles sont les conditions qui déterminent l'application d'une traduction ?

Avant d'exporter un EA source vers un LRC cible, une traduction depuis l'EA source n'est nécessaire que si les deux conditions suivantes sont réunies :

- un type d'EA du modèle cible peut être utilisé comme instrument de description pour le même contenu de description que le type décrit par l'EA source,
- les types d'EAs source et cible ne sont pas comparables (cf. comparaison sémantique) sans la prise en compte de définitions qui lieraient ces deux types d'EAs.

Utilisation de SRS pour les traductions entre EAs

SRS ne peut pas résoudre de règles. Les règles présentées dans les sections [6.2.2](#) et [6.2.3](#) pourraient cependant être traduites en fonctions. SRS pourrait alors être utilisé pour des traductions entre EAs.

6.2.2. Traduction entre `Frame_as_NR-phrase` et `Conjunction_of_links_from_a_same_source`

L'extrait ci-dessous présente une règle de traduction en FL entre `Frame_as_NR-phrase` et `Conjunction_of_links_from_a_same_source`. Cette règle a été écrite par Philippe Martin et est spécifiée dans `d_KRLmodelsTranslation.html`.

```
And{[%relSource = %f_head]
  [%halfProperties r_half-property-item_matching_this_binary-relation: %binRel]
  [%binaryRelations r_binary-relation_item_matching_this_half-property: %halfProperty] }
<= [%t /^^ Thing_that_can_be_represented_via_binary-relations_from_a_same_source
    Thing_that_can_be_represented_via_a_frame,
    r_descr_instrument: (%c /^^ Conjunction_of_binary-relations_from_a_same_source,
                        r_args: (%binaryRelations pred#list-contains:
                                (%binRel r_1st_arg: %relSource) ) )
    r_descr_instrument: (%f /^^ Frame_as_NR-phrase, r_op: %f_head,
                        r_args: (%halfProperties pred#list-contains: %halfProperty) ) ];

//With the following definitions:
Thing_that_can_be_represented_via_a_frame
  <=> Thing_that_can_be_represented_via_binary-relations_from_a_same_source;

r_half-property-item_matching_this_binary-relation ~[?halfProperties, ?binRel]
  := [%halfProperties pred:list-contains:
      (a thing ?halfProperty r_op: (an Operator ?relType r_op of: ?binRel),
       r_1st_arg: (a Relation_node ?relDest r_2nd_arg of: ?binRel) ) ];
r_binary-relation_item_matching_this_half-property ~[?binaryRelations, ?halfProperty]
  := [%binaryRelations pred:list-contains:
      (a thing ?binRel r_op: (an Operator ?relType r_op of: ?halfProperty),
       r_2ng_arg: (a Relation_node ?relDest r_1st_arg of: ?halfProperty) ) ];
```

Ci-après, la même règle écrite en RIF-BLD avec la notation RIF-PS.

```

Forall ?t ?c ?binaryRelations ?binRel ?f ?f_head ?halfProperties ?halfProperty (
  And( ?relSource = ?f_head
    r_half-property-item_matching_this_binary-relation(?halfProperties ?binRel)
    r_binary-relation_item_matching_this_half-property(?binaryRelations ?halfProperty) )
  :- And( ?t[rdf:type->Thing_that_can_be_represented_via_binary-relations_from_a_same_source
    rdf:type->Thing_that_can_be_represented_via_a_frame
    r_descr_instrument->?c r_descr_instrument->?f ]
    ?c[rdf:type->Conjunction_of_binary-relations_from_a_same_source
    r_args->?binaryRelations ]
    ?f[rdf:type->Frame_as_non-token-phrase r_op->?f_head
    r_args->?halfProperties ]
    External(pred:list-contains(?binaryRelations ?binRel))
    ?binRel[r_1st_arg->?relSource]
    External(pred:list-contains(?halfProperties ?halfProperty)) ) )

```

//With the following definitions:

```

Forall ?t (
  ?t[rdf:type->Thing_that_can_be_represented_via_a_frame]
  :- ?t[rdf:type->Thing_that_can_be_represented_via_binary-relations_from_a_same_source] )
Forall ?t (
  ?t[rdf:type->Thing_that_can_be_represented_via_binary-relations_from_a_same_source]
  :- ?t[rdf:type->Thing_that_can_be_represented_via_a_frame] )

```

```

Forall ?halfProperties ?halfProperty ?binRel ?relType ?relDest (
  r_half-property-item_matching_this_binary-relation(?halfProperties ?binRel)
  :- And( External(pred:list-contains(?halfProperties ?halfProperty))
    ?halfProperty[r_op->?relType r_1st_arg->?relDest]
    ?binRel[r_op->?relType r_2nd_arg->?relDest] ) )

```

```

Forall ?binaryRelations ?halfProperty ?binRel ?relType ?relDest (
  r_binary-relation_item_matching_this_half-property(?binaryRelations ?halfProperty)
  :- And( External(pred:list-contains(?binaryRelations ?binRel))
    ?binRel[r_op->?relType r_2nd_arg->?relDest]
    ?halfProperty[r_op->?relType r_1st_arg->?relDest] ) )

```

```

Forall ?at ?arg1 ?l ( ?at[r_1st_arg->?arg1]
  :- ?at[r_args->List(?arg1 | ?l) ] )

```

```

Forall ?at ?arg1 ?arg2 ?l ( ?at[r_2nd_arg->?arg2]
  :- ?at[r_args->List(?arg1 ?arg2 | ?l) ] )

```

6.2.3. Traduction entre une relation binaire et une relation n-aire

L'extrait ci-dessous présente une règle de traduction en FL entre Binary_relation et Non-binary_relation. Cette règle a été écrite par Philippe Martin et est spécifiée dans d_KRLmodelsTranslation.html.

```
And{ [%binRelArg1 = %nbrArg1] [%binRelArg2 = %nbrArgsExcept1st] }
  <= [%t /^^ Thing_that_can_be_represented_via_a_non-binary_relation
      Thing_that_can_be_represented_via_a_binary_relation,
      r_descr_instrument: (%nonBinRel /^^ Non-binary_relation,
                           r_args: List(%nbrArg1 | %nbrArgsExcept1st) ),
      r_descr_instrument: (%binRel /^^ Binary_relation,
                           r_1st_arg: %binRelArg1, r_2nd_arg: %binRelArg2 ) ];

//With the following definitions:
Thing_that_can_be_represented_via_a_non-binary_relation
  has_descr_instrument<=: a Non-binary_relation, //actually "=:," not just "<=:," but "=: a ..."
<=> (Thing_that_can_be_represented_via_a_binary_relation, // cannot be translated to RIF-BLD
      has_descr_instrument<=: a Binary_relation ); // without skolem functions
```

Ci-dessous, la même règle écrite en RIF-BLD avec la notation RIF-PS.

```
Forall ?t ?binRel ?binRelArg1 ?binRelArg2 ?nonBinRel ?nbrArg1 ?nbrArgsExcept1st (
  And ( ?binRelArg1 = ?nbrArg1 ?binRelArg2 = ?nbrArgsExcept1st )
  :- And( ?t [rdf:type->Thing_that_can_be_represented_via_a_non-binary_relation
              rdf:type->Thing_that_can_be_represented_via_a_binary_relation
              r_descr_instrument->?nonBinRel r_descr_instrument->?binRel ]
        ?binRel[rdf:type->Binary_relation r_1st_arg->?binRelArg1
                 r_2nd_arg->?binRelArg2]
        ?nonBinRel[rdf:type->Non-binary_relation
                    r_args->List(?nbrArg1 | ?nbrArgsExcept1st) ] ) )

//With the following definitions:
Forall ?t ?nbr (
  ?t[rdf:type->Thing_that_can_be_represented_via_a_non-binary_relation]
  :- And ( ?t[has_descr_instrument->?nbr] ?nbr[rdf:type->Non-binary_relation] ) )

Forall ?t ?br (
  ?t[rdf:type->Thing_that_can_be_represented_via_a_binary_relation]
  :- And ( ?t[has_descr_instrument->?nbr] ?br[rdf:type->Binary_relation] ) )

Forall ?t (
  ?t[rdf:type->Thing_that_can_be_represented_via_a_binary_relation]
  :- ?t[rdf:type->Thing_that_can_be_represented_via_a_non-binary_relation] )
```

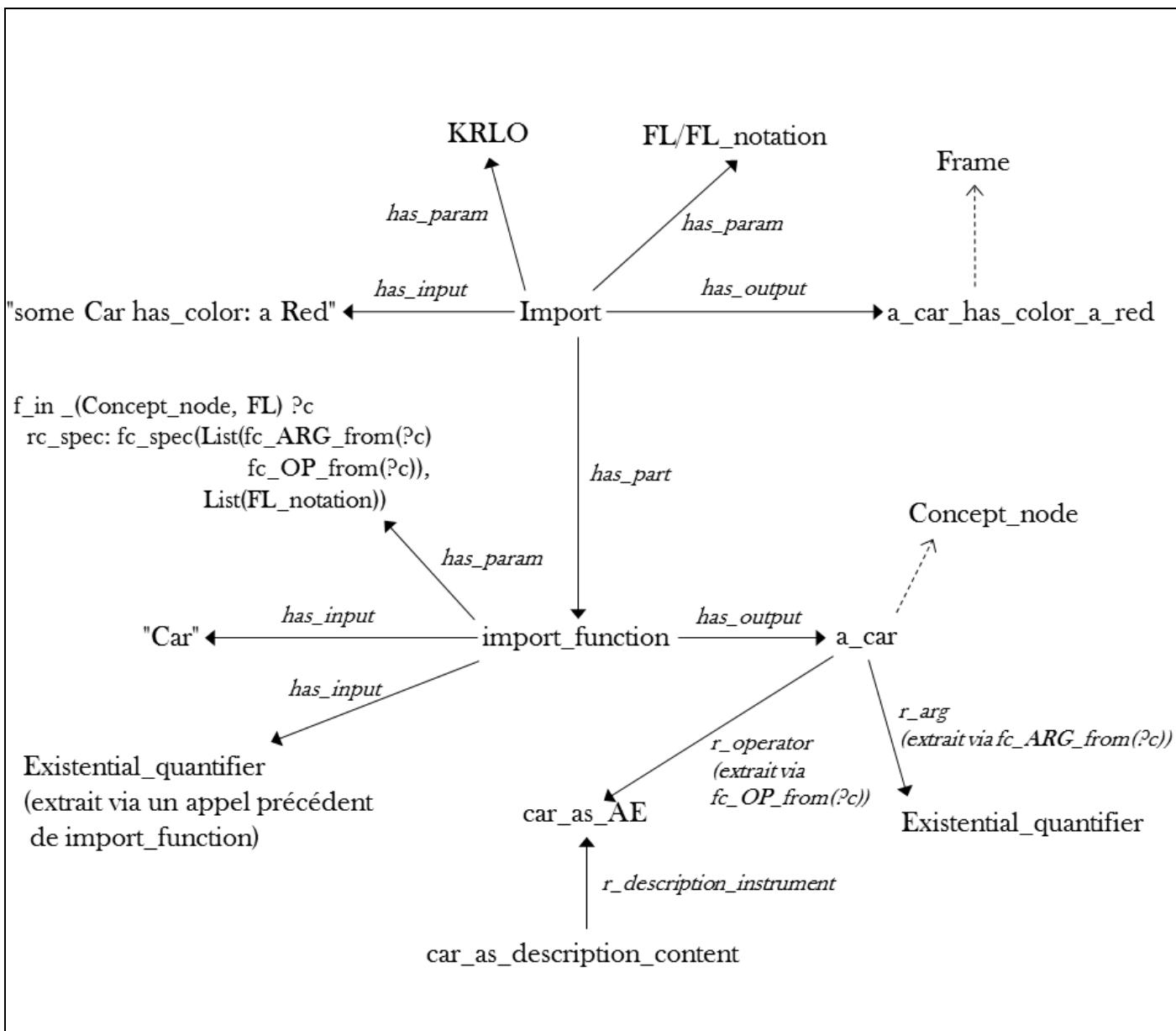
6.3. Import depuis un modèle concret vers un modèle abstrait

La section “6.3.1. Principe et spécifications” présente la fonction d'import que j'ai écrite et qui est basée sur Structure_map. Cette fonction effectue une analyse de type top-down avec retour sur trace [backtracking] et a une complexité exponentielle. La section “6.3.2. Complétude” montre que cette fonction est complète au sens où tous les EAs qui peuvent être extraits de son entrée sont effectivement extraits.

6.3.1. Principe et spécifications

Dans cette thèse, un processus d'import prend en entrée des éléments textuels ou graphiques et a en sortie des EAs (cf. section “3.1. Import de connaissances”). Ainsi, un tel processus extrait des EAs pour un modèle abstrait particulier à partir d'un texte ou d'un graphique. L'import fonctionne comme l'export mais en sens inverse. Comme le montre la figure ci-après, un processus d'import exploitant KRLO peut utiliser les spécifications de notations des types d'EAs concernés pour extraire des instances de ces types.

Figure 6.5. Fonction d'import en pm#UML.



Comme SRS ne prend en entrée que des RichGraph, SRS ne peut pas être directement utilisé directement pour l'import. SRS pourrait être utilisé dans un processus d'import où le texte (ou le graphique) en entrée serait d'abord importé dans un RichGraph. Par exemple, chaque terme pourrait être importé dans une unique `jb#List` puis cette liste pourrait être donnée en entrée à SRS. Un tel import (vers une `jb#List`) peut être réalisé via un tokenizer. Le principe de l'import depuis cette liste vers un EA structuré est décrit dans le paragraphe suivant.

Pour chaque EA du modèle (donné en paramètre de l'import), la spécification d'export pour la notation (donné en paramètre de l'import) est testée via une "fonction d'import". Cette "fonction d'import" spécifie i) le parcours de la spécification d'EC (par exemple : `List(f_OP_from_(?ae) "(" f_ARGS_from_(?ae) ")")`, cf. [4.1.3.3](#) et [4.1.4](#)), ii) développe puis compare chaque élément ainsi parcouru au token courant dans la chaîne de caractère en entrée. Cette comparaison permet soit d'invalider la règle testée, soit de poursuivre le test sur le token suivant.

J'ai écrit des fonctions en FL pour importer des EAs structurés depuis une liste d'ECs non structurés. Ces fonctions sont décrites ci-après. Dans son implémentation actuelle, SRS peut théoriquement traiter ces fonctions. En pratique, je dois d'abord i) optimiser le code afin de diminuer temps de calcul, et ii) terminer le développement de `TPatternBuilder`.

```

jb#f_import_list_of_token_to_AE
  input: 1 List ?list_of_token,  param: 1 KRL_model ?model 1 KRL_notation ?notation,  output: 1 AE,
  := jb#f_import_fct _(?list_of_token, ?model, ?notation).first;

jb#f_import_fct //retourne une liste dont le premier élément est un EA structuré
                //et donc le second élément est la liste restante des tokens à importer
  input: 1 List ?list_of_token,
  param: 1 KRL_model ?model 1 KRL_notation ?notation,  output: 1 List,
  := jb#Simple_graph_map _(jb#f_all_possible_new_AE_instances _(?model, ?notation),
                          ^( jb#f_test_spec input: ?list_of_token ?model ?notation),
                          jb#f_thing_that_does_not_have_for_r-part_a_parse_error );//si aucune règle
                                                                //valide renvoie null

//f_all_possible_new_AE_instances créée et renvoie une liste d'individus.
//Chaque individu de cette liste a pour type l'un des types d'AEs de ?model.
jb#f_all_possible_new_AE_instances
  param: 1 KRL_model ?model 1 KRL_notation ?notation,  output: 1 List,
  := jb#Simple_graph_map _( ?model.r_KRL_type_of,
                          ^(jb#fquery_has_specs input: ?notation),
                          jb#fquery_build_list_of_instances_of_AE );

jb#fquery_has_specs
  input: 1 AE ?ae 1 KRL_notation ?notation,
  := [ [?ae rc_spec: (a List r_2nd_arg: (?notationSet r_member: ?notation))] ];

jb#fquery_build_list_of_instances_of_AE
  input: 1 AE ?ae,  output: 1 List ?result,
  := [ [?result head: (Individual type: ?ae),
        tail: 1 List] ];

//f_test_spec construit les parties (opérateur et arguments) d'un AE suivant
//les spécifications de notations et la liste de tokens à importer ou renvoie une erreur
jb#f_test_spec
  input: 1 AE ?ae 1 List ?list_of_token,  param: 1 KRL_model ?model 1 KRL_notation ?notation,
  output: 1 List,
  := jb#Simple_graph_map _( jb#fc_find_specs _(?ae, ?notation)
                          ^(jb#f_compare_spec_element_with_string_token input: ?list_of_token.head,
                                                                    ?ae,
                                                                    ?list_of_token) );

```

```

//f_compare_spec_element_with_string_token i) compare chaque élément de la liste de spécifications
//avec le token courant, ii) crée les relations entre l'EA courant et ses parties,
//et iii) détecte les erreurs
jb#f_compare_spec_element_with_string_token
  input: 1 List_or_string ?spec_element //e.g. result of f_op_from(?ae) or "("
         1 String ?string_token 1 AE ?ae 1 List ?list_of_token,
  param: 1 KRL_model ?model 1 KRL_notation ?notation,
  := if_then_else _([?spec_element.first < r_part], //si ?spec_element spécifie un rôle pour l'EC
    /*then*/if_then_else _([?spec_element.second = ?string_token], //si ?string_token est une notation
      //spécifiée par défaut dans KRLO
      //le test ci-dessus ne renvoie "vrai" que si i) ?spec_element.first = r_operator et
      // ii) ?spec_element.second renvoie le résultat de rc_operator_name
      /*then*/jb#f_add_operator_name _(?ae, ?string_token),
      /*else*/if_then_else _([?spec_element.first = r_args],
        /*then*/ jb#f_create_args _(?ae, ?list_of_token, ?spec_element),
        /*else*/if_then_else _ (jb#fc_find_specs(jb#f_such_AE_part(?ae, ?spec_element.first),
          ?model,
          ?notation),
          /*then*/jb#f_test_spec _ (jb#f_build_AE _ ( ?ae, ?spec_element.first ),
            ?list_of_token),
          /*else*/ jb#f_add_relation_to_Type_as_AE _ ( ?string_token, ?spec_element ) ) ) )
    //ci-dessous, ?spec_element est un string car ne défini pas de rôle
    /*else*/if_then_else([?spec_element = ?string_token]
      /*then*/List(null, ?list_of_token.tail), //aucune erreur mais pas d'EA à importer
      /*else*/List(jb#parse_error, ?string_token) ) );//la règle courante n'est pas applicable

jb#f_add_operator_name
  < Generation_function,
  input: 1 AE ?ae 1 String ?string_token 1 List ?list_of_token,  output: 1 List,
  := List( (?ae rc_operator_name: ?string_token)
    ?list_of_token.tail );

jb#f_create_args //cherche un élément de sucre syntaxique qui sépare la séquence d'argument du prochain AE
  //et ajoute une relation r_arg depuis ?ae vers chacun des arguments importés
  < Generation_function,
  input: 1 AE ?ae 1 List ?list_of_token,
  param: 1 List_or_string ?spec_element 1 KRL_model ?model 1 KRL_notation ?notation,
  output: 1 List,
  := if_then_else _([?list_of_token.head = ?spec_element.nextElement]
    /*then*/List(?ae ?list_of_token),
    /*else*/jb#f_create_args _ ( (?ae r_arg: jb#f_import_fct _ (?list_of_token, ?model, ?notation).first),
      jb#f_import_fct _ (?list_of_token, ?model, ?notation).second,
      ?spec_element ) );

jb#f_such_AE_part
  input: 1 AE ?ae 1 r_part ?rel,  output: 1 AE ?result,
  := [ ?ae ?rel: ?result ]; //e.g. AE@FL r_op: 1 Concept_node

jb#f_build_AE
  < Generation_function,
  input: 1 AE ?ae 1 r_part ?rel,
  := (AE ?rel: ?ae);

jb#f_add_relation_to_Type_as_AE
  < Generation_function,
  input: 1 String ?string_token 1 function_or_string ?spec_element 1 AE ?ae,  output: 1 List,
  := List( (?ae ?spec_element.first: (Type_as_AE
    r_description_instrument of: find_content(?string_token)) )
    ?list_of_token.tail);

```


6.3.2. Complétude

Définition

Complétude d'une fonction d'import : tous les EA qui peuvent être extraits de l'entrée de la fonction sont en sortie, en supposant que l'ontologie de LRCs en paramètre soit complète.

Complétude de la fonction d'import présentée dans la section 6.3.1.

Pour les trois raisons présentées dans la liste suivante, la fonction d'import présentée dans la section 6.3.1. est complète si i) l'entrée de cette fonction peut être représentée par une grammaire, et ii) les descriptions du modèle en entrée et des spécifications de notation sont complètes (cf. section "4.4. Complétude de KRLO").

- La sortie de la fonction d'import – c'est à dire, un EA structuré et ses parties – est générée puis assemblée par appels récursifs de cette fonction.
- La liste d'ECs en entrée de la fonction d'import n'est parcourue entièrement que si chacun de ses éléments est associé à un élément d'une spécification de présentation.
- À chaque appel récursif, les parties de l'EA en sortie ne sont générées que si leurs spécifications de présentation correspondent aux ECs parcourus.

7. Conclusion

7.1. Contributions scientifiques apportées par cette thèse

KRLO

J'ai contribué à la création de KRLO, la première ontologie de LRCs qui représente i) des modèles abstraits de LRCs de différentes familles, par exemple, le modèle de Common Logics et les modèles RDF+OWL, ii) des notations de LRCs, et iii) des fonctions spécifiant des méthodes d'import, de traduction et d'export de RCs.

Structure_map

J'ai noté quelques paramètres manquants dans la première spécification de la définition de la fonction pm#Structure_map puis j'ai créé des spécialisations de la dernière version de cette fonction. Les fonctions de type Structure_map peuvent être utilisées pour représenter et organiser de façon systématique toute autre fonction qui utilise une boucle. J'ai utilisé des fonctions basées sur Structure_map pour représenter des fonctions d'import et d'export.

SRS

J'ai créé SRS. SRS est le premier résolveur de requêtes basé sur Structure_map. Ainsi, i) SRS est hautement paramétrable, ii) des spécialisations de SRS peuvent être systématiquement organisées dans une ontologie, et iii) SRS pourrait être facilement optimisé pour résoudre rapidement des fonctions basées sur Structure_map, de telles optimisations n'ont cependant pas encore été implémentées. De plus, SRS constitue une preuve de concept partielle pour l'hypothèse sur laquelle s'appuient mes travaux sur la représentation de composants logiciels, c'est à dire, "*Tout programme peut être conçu en combinant des fonctions de parcours ou de manipulation de structure de données*". Cette hypothèse n'a pas encore été démontrée de façon formelle.

7.2. Solutions apportées aux problèmes de recherche

Les sous-sections suivantes reprennent les questions de recherche présentées dans l'introduction.

Comment spécifier une fonction d'export et ses entrées (définitions de types ou, alternativement à ces définitions, des règles ou des fonctions) pour que cette fonction soit générique ?

Cette thèse a présenté deux fonctions d'export génériques (cf. [6.1.1.2.](#) et [6.1.1.3.](#)) de complexité polynomiale. Elles prennent en entrée KRLO, un modèle concret et un EA structuré, tous deux représentés dans KRLO. Avec de telles entrées, une fonction qui génère un EC peut être spécifiée via les sous-fonctions suivantes.

1. Une fonction qui permet de parcourir les parties *EA_part* de l'EA en entrée. Si nécessaire, l'application de cette fonction peut être suivie par une sélection et une application de transformations pertinentes sur *EA_part*.
2. Une fonction qui permet de rechercher et d'appliquer la spécification de présentation pour *EA_part* et pour la notation en entrée, dans KRLO. L'application de cette spécification retourne au moins un EC.
3. Une fonction qui permet de concaténer des ECs.

Comment spécifier une fonction d'import et ses entrées pour que cette fonction soit générique (c'est à dire, utilisable pour n LRCs définis de manière adéquate) ?

Cette thèse a présenté une fonction d'import générique (cf. [6.3.](#)). Cette fonction prend en entrée une liste d'ECs, KRLO, un modèle abstrait et un modèle concret tous deux représentés dans KRLO. Avec de telles entrées, une fonction qui génère un EA structuré peut être spécifiée via les fonctions suivantes.

1. Une fonction qui permet de parcourir *L_EC*.
2. Une fonction qui permet de parcourir la liste *L_t_EA* – vide par défaut – en entrée. Si cette liste est vide, cette fonction lui ajoute chaque type d'EA du modèle *M* qui porte une spécification de présentation *EA_spec* pour *N*. Cette spécification est une liste d'ECs ou de références vers des EAs.
3. Une fonction *f_EA_spec* qui permet de comparer les éléments de *EA_spec* avec les éléments de *L_EC*, et une fonction *f_EA_gen* qui permet de générer des éléments abstraits. Pour chaque élément *spec_element* de *EA_spec*, *f_EA_spec* applique les traitements présentés dans la sous-liste ci-dessous.
 - 3.1. Si *spec_element* est un EC (donc soit un mot clef, soit un délimiteur), cette fonction compare *spec_element* et l'EC courant dans la liste *L_EC* ; si le résultat de cette comparaison est négatif, le parcours de *EA_spec* est abandonné.
 - 3.2. Sinon (*spec_element* contient une référence vers une partie d'un EA *EA_part*), la fonction *f_EA_gen* est alors appelée avec *EA_part* en argument. Cette fonction applique les traitements présentés dans la sous-liste ci-dessous.
 - 3.2.1. Si *EA_part* est une spécialisation de *Type_as_description_content*, une instance de *EA_part* est générée.
 - 3.2.2. Sinon (*EA_part* est une spécialisation de *Language_element*), la fonction d'import est appelée avec en argument une liste *L_t_EA* ne contenant que *EA_part* ; si cet appel ne retourne pas d'EA, le parcours de *EA_spec* est abandonné.
 - 3.3. Si le parcours de *EA_spec* n'a pas été abandonné, une instance *EA_out* de l'élément courant de *L_t_EA* est générée avec ses sous-parties (générées ci-dessus par appels de *f_EA_gen*). *f_EA_spec* retourne *EA_out*.

Comment spécifier ces fonctions et/ou leurs entrées de sorte qu'elles soient suffisantes pour des traductions entre LRCs ?

Des traductions entre EAs sont nécessaires (cf. [6.2.1](#)). Cependant, ces traductions peuvent être intégrées aux processus d'import ou d'export. Dans KRLO, ces traductions sont représentées de manière déclarative via des définitions ou des règles. Ainsi, il n'est pas nécessaire d'écrire des traducteurs de manière procédurale et ad hoc pour ces EAs. Parmi les autres traducteurs (de LRCs) qui permettent des traductions (entre EAs), seuls ceux qui exploitent LATIN permettent aussi de représenter des définitions ou des règles au lieu de les représenter de manière procédurale. Cependant, LATIN ne représente que des EAs d'une logique quelconque comme les quantifications ou les conjonctions (cf. [3.3.4.3](#)). Les traducteurs (de LRCs) qui n'exploitent ni LATIN, ni KRLO proposent au mieux une traduction spécifiée de manière procédurale pour chaque couple d'EAs (voir, par exemple, [3.3.1.2.1](#)).

La partie “règles de traduction” de KRLO (fichier [d_KRLmodelsTranslation.html](#)) a montré que beaucoup de règles utilisées pour la traduction entre EAs nécessitent un quantificateur existentiel dans la partie conclusion (cf. [6.2](#)). Ces règles ne sont pas des règles de Horn strictes [definite Horn rules]. Ainsi, ces règles ne sont pas utilisables par des moteurs d'inférences conçus pour des LRCs exécutables comme Datalog, Prolog ou RIF-BLD. Par contre, de telles règles peuvent être représentées en SPARQL ou dans des LRCs suivant une logique d'ordre 1 ou supérieure. De tels LRCs sont, par exemple, exploitable dans HETS. Alternativement, les définitions ou règles de traduction entre EAs peuvent aussi être exécutées via une implémentation procédurale ou fonctionnelle des processus d'import et export génériques, comme c'est le cas dans ceux que développe Philippe Martin en Javascript.

7.3. Perspectives

7.3.1. Perspectives scientifiques

Extension de KRLO aux langages formels qui ne sont pas des LRCs

Philippe Martin et moi prévoyons d'étendre KRLO pour y représenter des langages formels qui ne sont pas des LRCs. Cette ontologie étendue sera nommée FLO. Ainsi, des phrases écrites dans tout langage formel représenté dans FLO pourront être traduites vers tout autre langage formel également représenté dans FLO. En particulier, des phrases écrites dans des langages de programmation pourront être traduites ou généralisées dans des LRCs.

Une application possible pour FLO est de permettre la création d'un éditeur générique de code sémantique. Dans un tel éditeur, les contenus de description et les instruments de description sont distincts. Ainsi, des programmeurs pourront choisir leurs instruments de description (c'est à dire, leur langage) favoris sans consensus. Pour rechercher et donc réutiliser plus facilement ces instruments ou contenus de descriptions dans un éditeur de code sémantique, une ontologie de composants logiciels est nécessaire.

Création d'une ontologie de composants logiciels de base (SCO, Software Components Ontology)

Dans cette ontologie, des composants logiciels seront représentés de façon systématique, uniforme et modulaire via la fonction `Structure_map`. Ainsi, SCO pourra être organisée de façon systématique car tout composant logiciel peut être sémantiquement comparé (cf. comparaison sémantique) à `Structure_map`. Cette organisation ne permettra à un utilisateur final de rechercher des composants logiciels autrement que via des critères structurels : entrée, sortie, fonction de parcours, etc. En effet, pour d'autres types de critères, une ontologie de critères d'évaluation est nécessaire.

Création d'une ontologie de critères d'évaluation de composants logiciels ou de services

Cette ontologie organisera des critères caractéristiques des composants logiciels ou des services. Ainsi, un utilisateur final pourra i) rechercher des critères particuliers, par exemple, en fonction d'un domaine (coopération, sécurité, qualité logicielle, etc.), ii) choisir des composants qui vérifient certains critères ou vérifier qu'un composant nouvellement créé satisfait ces critères.

Via des comparaisons avec d'autres approches selon des critères liés au partage et à la réutilisation des connaissances, cette ontologie permettra également d'évaluer de façon plus formelle l'intérêt de l'approche proposée dans cette thèse pour la traduction entre LRCs.

7.3.2. Perspectives pour GTH

J'ai développé SRS, un résolveur de requêtes paramétrable et intégrable dans le framework LAS. SCS est déjà intégré dans une ancienne version de LAS. SCS sera intégré à une prochaine version de LAS lorsque i) les capacités de LAS dans cette version le permettront, et ii) le temps nécessaire pour cette intégration me sera accordé.

Pour l'instant, les performances de SRS limitent son utilisabilité à quelques applications de type preuve de concept. Lorsque j'aurai suffisamment amélioré ces performances, la direction de GTH examinera d'autres usages. Par exemple, SRS pourrait être exploité dans les produits suivants.

- logiCells BPAO (Business Plan Assisté par Ordinateur), un assistant de création de plan d'affaire (business plan) destiné aux PME (Petites et Moyennes Entreprises).
- logiCells PIM (Personal Information Manager), une application de gestion des connaissances destinée aux PME.

Références

- Bechhofer S., Möller R. & Crowther P. (2003). The DIG description logic interface. In Proceedings of the 2003 international workshop on description logics (dl2003), rome, italy september 5-7, 2003.
- Baget J.-F., Croitoru M., Gutierrez A., Leclère M. & Mugnier M.-L. (2010). Translations between RDF(S) and conceptual graphs. In Conceptual structures: From information to intelligence, 18th international conference on conceptual structures, ICCS 2010, kuching, sarawak, malaysia, july 26-30, 2010. proceedings (pp. 28-41).
- Baget J.-F., Gutierrez A., Leclère M., Mugnier M.-L., Rocher S. & Sipieter C. (2015). Datalog+, ruleml and OWL 2: Formats and translations for existential rules. In Proceedings of the ruleml 2015 challenge, the special track on rule-based recommender systems for the web of data, the special industry track and the ruleml 2015 doctoral consortium hosted by the 9th international web rule symposium (ruleml 2015), berlin, germany, august 2-5, 2015.
- Berners-Lee T., Connolly D., Kagal L., Scharf Y., Hendler J. (2005). N3Logic: A logical framework for the World Wide Web. *Theory and Practice of Logic Programming*, 8, (3), pp. 249-269
- Bénard J. & Martin Ph. (2015). Improving General Knowledge Sharing via an Ontology of Knowledge Representation Language Ontologies. Chapter 23 (pp. 364-387: 22 pages) of CCIS 553: book from the Springer-Verlag Lectures Notes series "Communications in Computer and Information Science" (CCIS). Book title: "Knowledge Discovery, Knowledge Engineering and Knowledge Management". This book chapter is an extension of our "KEOD+KDIR 2014 best paper award" article listed below (selection rate: 12,9% -- 37 papers out of 287 submissions).
- Borras P., Clément D., Despeyrouz Th., Incerpi J., Kahn G., Lang B. & Pascual V. (1988). CENTAUR: the system. SIGSOFT'88, 3rd Annual Symposium on Software Development Environments (SDE3), Boston, USA, pp. 148-24.
- Botting R. (2012). How Far Can EBNF Stretch? <http://cse.csusb.edu/dick/papers/rjb99g.xbnf.html>
- BPMN (2011). BPMN: Business Process Model and Notation. Version 2.0. OMG document Number: formal/2011-01-03.
- Brophy M. & Heflin J. (2009). OWL-PL: A Presentation Language for Displaying Semantic Data on the Web. Technical report. Department of Computer Science and Engineering, Lehigh University.
- Chalupsky H. (2000). Ontomorph: A translation system for symbolic knowledge. In KR 2000, principles of knowledge representation and reasoning proceedings of the seventh international conference, breckenridge, colorado, usa, april 11-15, 2000. (pp. 471-482).
- Chein M., Mugnier M.-L & Simonet G. (1998). Nested graphs: A graph-based knowledge representation model with FOL semantics. In Proceedings of the sixth international conference on principles of knowledge representation and reasoning (kr'98), trento, italy, june 2-5, 1998. (pp. 524-535).
- CL (2007). Information technology – Common Logic (CL): a framework for a family of logic-based languages. ISO/IEC 24707:2007(E), JTC1/SC32.
- Codescu M., Horozal F., Kohlhase M., Mossakowski T. & Rabe F. (2011). Project Abstract: Logic Atlas and Integrator (LATIN). In *Intelligent Computer Mathematics 2011*, LNCS 6824, pp. 287-289. See also <http://trac.omdoc.org/LATIN/>
- Codescu M., Horozal F., Kohlhase M., Mossakowski T., Rabe F. & Sojakova K. (2010). Towards logical frameworks in the heterogeneous tool set hets. In *Recent trends in algebraic development techniques - 20th international workshop, WADT 2010*, etelsen, germany, july 1-4, 2010, revised selected papers (pp. 139-159).
- Codescu M., Kuksa E., Kutz O., Mossakowski T. & Neuhaus F. (2016). Ontohub: A semantic repository for heterogeneous ontologies. CoRR, abs/1612.05028.
- Corby O. & Faron-Zucker C. (2015). STTL: A SPARQL-based Transformation Language for RDF. In Proceedings of WEBIST 2015, 11th International Conference on Web Information Systems and Technologies (Lisbon, Portugal).
- Corby O., Faron-Zucker C., Gandon F. (2015). A Generic RDF Transformation Software and its Application to an Online Translation Service for Common Languages of Linked Data. The 14th International Semantic Web Conference, Oct 2015, Bethlehem, United States.
- Corcho Ó & Gómez-Pérez A. (2007). Odedialect: a set of declarative languages for implementing ontology translation systems. *J. UCS*, 13 (12), 1805-1834.
- Croitoru M., Compatangelo E. (2006). Conceptual graph assemblies. In: Hitzler, P., Scharfe, H., Ohrstrom, P. (eds.) *Contributions to ICCS 2006, 14th International Conference on Conceptual Structures*, pp. 15-28. Aalborg University Press.
- Despeyrouz T. (1988). Typol: A formalism to implement Natural Semantics. Technical Report 94, INRIA Sophia-Antipolis.
- Dimou A., Sande M. V., Colpaert P., Verborgh R., Mannens E. & de Walle R. V. (2014). RML: A generic language for integrated RDF mappings of heterogeneous data. In Proceedings of the workshop on linked data on the web colocated with the 23rd

international world wide web conference (WWW 2014), seoul, korea, april 8, 2014.

- DOL (2016). Distributed Ontology, Modeling, and Specification Language (DOL). Version 1.0.-Beta1. OMG Document Number: ptc/2016-02-37.
- Entailment Regimes (2013). SPARQL 1.1 Entailment Regimes. W3C Recommendation 21 March 2013. <https://www.w3.org/TR/sparql11-entailment/>
- Euzenat J. & Stuckenschmidt H. (2003). The 'family of languages' approach to semantic interoperability. Knowledge transformation for the semantic web (eds: Borys Omelayenko, Michel Klein), IOS press, 49-63, 2003, 1-58603-325-5.
- Euzenat J. (2001b). Towards a principled approach to semantic interoperability. In Proceedings of IJCAI 2001, Workshop on Ontology and Information Sharing (Seattle, USA), pp19-25.
- Euzenat J. (2001e). Preserving modularity in XML encoding of description logics. In Proceedings of 14th Workshop on Description Logics (Stanford, USA), pp20-29.
- Euzenat J. (2001f). An infrastructure for formally ensuring interoperability in a heterogeneous semantic web. In Proceedings of SWWS 2001, 1st Semantic Web Working Symposium (Stanford, USA), pp345-360.
- Farquhar A., Fikes R. & Rice J. (1997). The Ontolingua Server: a tool for collaborative ontology construction. International Journal of Human-Computer Studies, Vol. 46, Issue 6, Academic Press, Inc., MN, USA.
- Feja S., Witt S. & Speck A. (2011). BAM: A Requirements Validation and Verification Framework for Business Process Models. In Proceedings of QSIC 2011, 11th Quality Software International Conference (Madrid, Spain), 186-191.
- FSS (2012). OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax (Second Edition). W3C Recommendation 11 December 2012. <https://www.w3.org/TR/owl2-syntax/>
- Genesereth M. & Fikes R. (1992). Knowledge Interchange Format, Version 3.0, Reference Manual. Technical Report, Logic-92-1, Stanford University. <http://www.cs.umbc.edu/kse/>
- Gruber T.R. (1993). A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, Vol. 5, Issue 2, 199-220.
- Guizzardi G. (2005). Ontological foundations for structural concept models (PhD thesis). University of Twente.
- Guizzardi G., Lopes M., Baião F. & Falbo R. (2010). On the importance of truly ontological representation languages. International Journal of Information Systems Modeling and Design (IJISMD), Vol. 1, Issue 2, 1-22.
- Harper R., Honsell F. & Plotkin G. D. (1987). A framework for defining logics. In Proceedings of the symposium on logic in computer science (lics'87), ithaca, new york, usa, june 22-25, 1987 (pp. 194-204).
- Hayes P.J. (2006). IKL guide. Unpublished memorandum, 2006. <http://www.ihmc.us/users/phayes/IKL/GUIDE/GUIDE.html>
- Horrocks I., Patel-Schneider P.F., Boley H., Tabet S., Grosz B. & Dean M. (2004). SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Submission, <http://www.w3.org/Submission/SWRL>, 2004.
- Human-Usable Textual Notation (HUTN) Specification. Version 1.0. (April 2004). OMG Document Number: formal/04-08-01.
- J.-F. Baget & M.-L. Mugnier (2002). Extensions of Simple Conceptual Graphs: The Complexity of Rules and Constraints. Journal of Artificial Intelligence Research (JAIR) Volume 16, pages 425-465.
- Kahn G., Lang B., Melese B. & E. Morcos (1983). Metal: A formalism to specify formalisms. Science of Computer Programming, pp. 151-188.
- Krötzsch M., Rudolph S. & Hitzler P. (2013). Complexity of Horn Description Logics, ACM Transactions on Computational Logic, Vol. 14, Issue 1, 2:1–2:36.
- Kutz O. & Mossakowski T. (2011). The onto-logical translation graph. In O. Kutz & T. Schneider (Eds.), Modular ontologies proceedings of the Fifth international workshop (womo 2011) (Vol. 230, p. 94-109). IOS Press.
- Lange C., Mossakowski T. & Kutz O. (2012). Lola: A modular ontology of logics, languages, and translations. In T. Schneider & D. Walther (Eds.), Workshop on modular ontologies (Vol. 875).
- Lefrançois M., Zimmermann A. & Bakerally N. (2017). A SPARQL extension for generating RDF from heterogeneous formats. In Proceedings of the Extended Semantic Web Conference (ESWC 2017), Portoroz, Slovenia.
- McDermott D. V. & Dou D. (2002). Representing Disjunction and Quantifiers in RDF. In The semantic web - ISWC 2002, First international semantic web conference, sardinia, italy, june 9-12, 2002, proceedings (pp. 250-263).
- Martin Ph. & Bénard J. (2014). An Ontology for Specifying and Parsing Knowledge Representation Structures and Notations. Proceedings of KEOD 2014 (6th International Conference on Knowledge Engineering and Ontology Development, Rome, Italy, 21-24/10/2014, ISBN: 978-989-758-049-9), pp. 96-107 (-> "Full paper"). Selected for the "KDIR 2014 best paper award". KEOD and KDIR (International Conference on Knowledge Discovery and Information Retrieval) are joint conferences. Selection rate of "Full papers" at KEOD 2014: 18% (78 submissions).

- Martin Ph. & Bénard J. (2016b). Top-level Ideas about Importing, Translating and Exporting Knowledge via an Ontology of Representation Languages. ACM proceedings of Semantics 2016 (doi: 10.1145/2993318.2993324; pp. 89-92), Leipzig, Germany, 12th to 17th September, 2016.
- Martin Ph. & Bénard J. (2017a). *Categorizing or Generating Relation Types and Organizing Ontology Design Patterns*. Proceedings of KAM 2017, 23rd IEEE Conference on Knowledge Acquisition and Management, 3-6 September 2017, Prague, Czech Republic.
- Martin Ph. & Bénard J. (2017b). *Creating and Using various Knowledge Representation Model and Notation*. Proceedings of ECKM 2017, 18th European Conference on Knowledge Management, 7-8 September 2017, Barcelona, Spain.
- Martin Ph. & Eklund P. (1999a). WebKB and the Sisyphus-I problem. Proceedings of ICCS 1999 (Springer, LNAI 1640, pp. 315-333), Blacksburg, Virginia, USA, July 12-15, 1999.
- Martin Ph. (2010d). Collaborative Ontology Modelling. Proceedings of ICCP 2010 (pp. 59-66; ISBN: 978-1-4244-8228-3) IEEE 6th International Conference on Intelligent Computer Communication and Processing, Cluj-Napoca, Romania, August 26-28, 2010.
- Martin Ph. (2009c). Towards a collaboratively-built knowledge base of&for scalable knowledge sharing and retrieval. HDR thesis (240 pages; "Habilitation to Direct Research"), University of La Réunion, France, December 8, 2009.
- Horridge M. & Bechhofer S. (2011). The OWL API: A Java API for OWL Ontologies. Semantic Web Journal 2(1), Special Issue on Semantic Web Tools and Systems, pp. 11-21. <http://owlapi.sourceforge.net/>
- Meta Object Facility (MOF). Core Specification. Version 2.5.1. (January 2016). OMG Document Number: formal/2016-11-01.
- MOF Model to Text Transformation Language (MOFM2T). Version 1.0. (January 2008). OMG Document Number: formal/08-01-16.
- Mossakowski T., Maeder C. & Lüttich K. (2006). Hets: The heterogeneous tool set. In M. Kohlhase (Ed.), *Maya: Maintaining structured developments* (Vol. 4180, p. 286-289). Springer.
- ODM (2014). ODM: Ontology Definition Metamodel. Version 1.1. OMG document Number: formal/2014-09-02.
- OKBC (1998). Open Knowledge Base Connectivity 2.0.3. <http://www.ai.sri.com/~okbc/spec/okbc2/>. November 23, 1998. See also OKBC (1995) at <http://www.ai.sri.com/~okbc/>.
- OWL2 (2012). OWL 2 Web Ontology Language: Profiles (Second Edition). W3C Recommendation. Editors: B. Motik, B. Cuenca Grau, I. Horrocks, Z. Wu, A. Fokoue & C. Lutz. <http://www.w3.org/TR/owl2-profiles>. See also the OWL2 Primer (2012) and the RDF syntax grammar (2014).
- OWL2 direct semantics (2012). OWL 2 Web Ontology Language Direct Semantics (Second Edition). W3C Recommendation 11 December 2012. <https://www.w3.org/TR/owl2-direct-semantics>
- OWL2 FSS from/to RDF triples (2012). OWL 2 Web Ontology Language: Mapping to RDF Graphs (Second Edition). W3C Recommendation 11 December 2012. <http://www.w3.org/TR/owl2-mapping-to-rdf/>
- OWL2 RDF-Based semantics (2012). OWL 2 Web Ontology Language RDF-Based Semantics (Second Edition). W3C Recommendation 11 December 2012. <https://www.w3.org/TR/owl2-rdf-based-semantics/>
- OWL2-RL in RIF-BLD (2013). OWL 2 RL in RIF (Second Edition). W3C Working Group Note 5 February 2013, <http://www.w3.org/TR/rif-owl-rl>
- OWL2-RL to RIF-BLD (2013). RIF RDF and OWL Compatibility (Second Edition). W3C Recommendation 5 February 2013
- Parreiras F. S. & Staab S. (2010). Using ontologies with UML class-based modeling: The twouse approach. *Data Knowl. Eng.*, 69 (11), 1194-1207.
- Pfenning F. & Schürmann C. (1999). System description: Twelf - A meta-logical framework for deductive systems. In *Automated deduction - cade-16*, 16th international conference on automated deduction, trento, italy, july 7-10, 1999, proceedings (pp. 202-206).
- Pietriga E., Bizer C., Karger D. & Lee R. (2006). Fresnel: A Browser-Independent Presentation Vocabulary for RDF. In Proceedings of ISWC 2006, LNCS 4273.
- Quan D. (2005). Xenon: An RDF Stylesheet Ontology. In Proceedings of WWW 2005, 14th World Wide Web Conference (Japan).
- QVT (2016). QVT: Query/View/Transformation. Version 1.3. OMG Document Number: formal/2016-06-03.
- R2RML (2012). R2RML: RDB to RDF mapping language. W3C Recommendation. Editors: S. Das, S. Sundara, R. Cyganiak. <https://www.w3.org/TR/2012/REC-r2rml-20120927/>
- Rabe F. & Kohlhase M. (2013). A scalable module system. *Journal of Information and Computation*, vol. 230 , 1-54.
- RIF-BLD (2013). RIF Basic Logic Dialect (Second Edition). W3C Recommendation. Editors: H. Boley, M. Kifer. <http://www.w3.org/TR/2013/REC-rif-bld-20130205/>

RIF-FLD (2013). RIF Framework for Logic Dialects (2nd edition). W3C Recommendation. Editors: Boley, H., Kifer, M., <http://www.w3.org/TR/2013/REC-rif-fl-d-20130205/>

SBVR (2008). Semantics of Business Vocabulary and Business Rules (SBVR). Version 1.0. OMG document formal/08-01-02. <http://www.omg.org/spec/SBVR/1.0/>

Topic Maps (2003). Information technology – SGML applications – Topic maps. ISO/IEC 13250:2003, JTC1/SC34.

Lexique

La terminologie utilisée dans cette thèse est dérivée des terminologies utilisées par les communautés de recherche dans la représentation des connaissances, les approches basées sur les modèles et l'analyse de langages formels.

Ce lexique permet d'alléger la présentation de ce mémoire et de la rendre plus claire.

Analyse lexicale [Tokenization] : processus qui prend en entrée une représentation textuelle ou graphique et a en sortie une liste d'ECs **atomiques**.

Analyse syntaxique [Parsing] : processus qui prend en entrée une liste de CEs et a en sortie un CST ou un AST ou un ASG incluant quelques CEs.

ASG [Abstract Semantic Graph] : graphe sémantique abstrait et donc un ensemble d'AEs.

AST [Abstract Syntax Tree] : arbre syntaxique abstrait et donc un ensemble d'AEs.

Base de connaissances (BC) [knowledge base (KB)] : collection de représentations de connaissances.

Chose [thing] : tout ce à quoi quelqu'un peut penser est une chose. Une chose peut avoir une description (cf. contenu de description, instrument de description, conteneur de description).

Commande : soit "phrase affirmée" dans une BC, soit requête sur une BC.

Comparaison sémantique : deux choses ou descriptions de choses sont *sémantiquement comparables* si l'une généralise l'autre, c'est à dire, si l'une peut être déduite logiquement de l'autre. Plusieurs outils, par exemple, les moteurs de résolution de requête, sont basés sur la comparaison de deux choses. Un outil tel qu'un démonstrateur de théorèmes peut être utilisé pour effectuer cette comparaison de façon automatique. Il y a deux cas.

La relation de déduction logique, par exemple, une relation subtype est déjà représentée de façon explicite entre les deux choses. Par exemple, deux primitives différentes peuvent être sémantiquement comparables si l'une est un sous-type de l'autre.

Les deux choses ont des définitions totales, c'est à dire, elles sont définies par des conditions nécessaires et suffisantes. Si ces descriptions sont sémantiquement comparables, les deux choses le sont aussi.

Concept-type-expression : définition d'un type de concept non nommé via des conditions nécessaires et suffisantes.

Conteneur de description : conteneur non physique d'une description mais qui a un support physique comme, par exemple, du papier ou une clef USB. Un fichier est un conteneur de description.

Contenu de description : contenu de la description d'une chose. Un contenu de description est décrit via un instrument de description.

Contexte d'une phrase : toutes les méta-phrases qui directement ou non, la contextualisent.

CST [Concrete Syntax Tree] : arbre syntaxique concret et donc un ensemble de CEs.

Définition d'un terme T : phrase quantifiant universellement T mais, contrairement à une observation, une définition n'est ni vraie ni fausse, elle ne fait que définir un terme par conditions nécessaires et/ou suffisantes.

Description [(declarative) sentence, e.g., (logic) statement] : une phrase déclarative qui décrit le monde, par opposition aux phrases qui prescrivent ce qui peut être entré, par exemple dans les bases de données.

Description procédurale ou fonctionnelle : ensemble de descriptions de structures de données et de traitements (fonctions ou procédures) sur ces structures de données. Si un langage impératif – comme Java – ou fonctionnel – comme Haskell – est utilisé, une description procédurale peut inclure une description du flux de contrôle des traitements.

Donnée [data] : ensemble de termes, qui n'est pas un objet sémantique.

Élément abstrait (EA) [Abstract element (AE)] : élément de LRC qui peut décrire, par exemple :

une formule, c'est à dire un phrase qui dénote un fait ;

un terme abstrait, comme par exemple, un appel de fonction ou un type de relation binaire.

Élément concret (EC) [Concrete element (CE)] : élément de LRC qui décrit la représentation concrète d'un élément abstrait. Par exemple, " $3 = 2 + 1$ " et " $(= (3 +(2 1)))$ " sont deux représentations concrètes d'un même élément abstrait qui est une formule d'égalité entre 3 et l'addition de 2 et 1.

Élément de LRC : terme ou phrase faisant partie de ce LRC.

Entité : chose qui n'est pas une situation qui peut participer à une situation.

homo-iconicité : propriété de certains langages dans lesquels la structure des CEs reflète la structure des AEs.

Individu [individual] : instance qui ne peut avoir d'instance.

Information : donnée ou représentation de connaissance.

Information explicitement représentée : information décrite par un objet sémantique.

Information implicitement représentée : information décrite par un objet d'information qui n'est pas un objet sémantique. La description peut alors être ambiguë. Par exemple, une relation structurelle a une sémantique implicitement représentée.

“Instance” de structure de données : structure de données conforme à un “type” de structure de données, i.e., les relations structurelles de “l’instance” ont le même nom que les relations structurelles du “type”. Par exemple, en Java, une “instance” de la classe *Vector* a la relation structurelle “elementCount” ainsi que toutes les autres relations structurelles spécifiées dans cette classe.

Instrument de description : objet d'information permettant de décrire une information. Un instrument de description est contenu dans un conteneur de description.

Kappa-expression : définition d'un type de relation non nommé via des conditions nécessaires et suffisantes.

Lambda-abstraction : définition d'un type de fonction non nommé via des conditions nécessaires et suffisantes.

Langage de description : LRC ou langage de description de structures de données.

Langage de description de structures de données : langage permettant d'écrire des termes et des relations structurelles entre ces termes. XML est un langage de description de structures de données.

Langage de programmation : langage permettant d'écrire des programmes.

Langage de programmation logique : langage permettant d'écrire un programme avec des expressions logiques, ce programme peut alors être exécuté via un algorithme de recherche de preuves. Prolog est un exemple de langage de programmation logique.

Langage de représentation de connaissances (LRC) : langage permettant d'écrire des **représentations de connaissances (RCs)**.

Langage déclaratif : langage via lequel tout processus ou programme peut être décrit sans description de son flux de contrôle.

Lien [Link] : relation binaire. Dans OWL2, un type de lien est une instance d'un des types du second ordre suivant “owl2:ObjectProperty” ou “owl2:DatatypeProperty”.

LRC exécutable : par définition, tous les LRCs ont une sémantique connue. Des moteurs d'inférence peuvent donc être construits pour interpréter ces LRCs. Dans cette thèse, nous appelons “LRC exécutable” un LRC dans lequel les problèmes de décision ont une complexité inférieure ou égale aux problèmes qui peuvent être exprimés en OWL2-DL (logique SHOIN(D)). Ceci inclut :

- les langages de programmation logique (voir la définition ci-après), e.g., Prolog, Datalog, RIF-BLD ;
- OWL-DL

Méta-phrase : phrase dont au moins un objet est une phrase. Par exemple, “John pense que 'il fait beau’”, “il existe une pensée qui a pour agent john et pour objet 'il fait beau’”, “Tom#birds_fly = John#"birds fly"” (ici la méta-phrase a deux objets qui sont des phrases reliées par la relation “=”) et “Cette phrase est fausse”.

Méta-phrase contextualisante : méta-phrase spécifiant des conditions pour que sa ou ses phrases objets sont vraies.

Modèle abstrait : ensemble d'AEs. Par exemple, une grammaire abstraite est un modèle abstrait.

Modèle conceptuel : BC non conçue pour une ou plusieurs applications particulières et donc dont l'expressivité n'est pas restreinte pour faciliter ces applications.

Modèle concret (ou notation) : ensemble de CEs. Par exemple, une grammaire concrète est un modèle concret.

Modèle de conception : BC dans laquelle des choix techniques ont été faits pour une ou plusieurs applications particulières, par exemple, un moteur d'inférence particulier.

Modèle exécutable : programme ou BC exécutable pour une application particulière avec un interpréteur particulier. Cet interpréteur est typiquement un moteur d'inférences tel que celui de Prolog qui permet la prise en compte de l'ordre des règles et utilise la négation par l'échec [negation as failure].

Module : ensemble d'informations isolées d'autres informations, par exemple, en utilisant fichier distinct pour chaque module ou via un AE prévu à cet effet pour chaque module.

Noeud (ou expression) : un terme et éventuellement un quantificateur sur ce terme ; de plus, un noeud est soit

- un **noeud concept** si le terme est un type de concept ou bien un individu ;
- un **noeud relation** si le noeud permet de créer une relation entre plusieurs noeuds concept. Le terme doit alors être un type de relation.

Notation de 2nd-ordre : notation permettant d'utiliser des variables pour les types de relations et de les quantifier. Une notation du 2nd-ordre n'implique pas nécessairement un moteur d'inférence du 2nd-ordre pour l'utiliser, en particulier si les phrases du 2nd-ordre sont des définitions. En effet, dans une BC donnée, il n'existe souvent qu'un nombre fini de types de 1er ordre concernés par ces définitions et il est alors souvent possible d'appliquer (c'est à dire, instancier) la définition du 2nd-ordre à ces catégories. Par exemple, définir la transitivité demande une logique du 2nd-ordre, mais définir une relation transitive particulière ne demande qu'une logique du premier ordre. Le moteur d'inférences peut alors ne pas prendre en compte la définition du 2nd-ordre.

Notation de haut niveau : notation qui permet de représenter des phrases générales de manières concises et normalisées et donc dont les représentations expressives et normalisées sont faciles à comparer, par exemple, par comparaison de graphes. Ceci implique (entre autres) la possibilité d'utiliser des méta-phrases contextualisantes, des quantificateurs numériques et d'autres raccourcis.

Objet d'information [informational object, resource] : phrase ou référence à une chose. Cette chose peut être une "ressource Web" si elle est référençable par un URI.

Objet lexicalement formel : objet d'information lexicalement unique dans les fichiers où il est déclaré ou utilisé. Un URI est un objet lexicalement formel car c'est un terme unique à l'échelle du Web.

Objet sémantiquement formel (objet sémantique) : objet dont la source (par exemple, son auteur) a déclaré qu'il avait un sens unique. Pour cela, il peut utiliser un LRC ou lui donner une définition non ambiguë. Pour avoir un sens unique, cet objet doit aussi être lexicalement formel. Dans la suite de cette thèse, nous abrégierons "objet sémantiquement formel" par "objet sémantique". Une représentation de connaissance est un objet sémantique.

Ontologie : ensemble de termes formels avec, associés à ceux-ci et portant sur eux, des définitions sémantiques partielles ou totales. Une ontologie est généralement logiquement définie, par exemple, via un LRC. En programmation orienté objet, il est possible de définir un type de concept via une classe, des propriétés (au sens UML) et des méthodes et donc également de décrire une ontologie qui n'est pas logiquement définie. Certaines BCs sont des ontologies, toute ontologie est une BC.

Ontologie de domaine : ontologie dont les termes sont relatifs à un domaine, par exemple, la radiographie, l'élevage des poules, les livres, etc.

Ontologie de haut niveau : ontologie dont les termes formels représentent des distinctions générales (par exemple, Entité, Situation, Espace, Temps, part, ...) généralement utiles pour organiser ou contrôler toute ontologie qui n'est pas une ontologie de langage.

Ontologie de LRCs : ontologie dont les termes décrivent un modèle de données, ainsi que sa logique associée, utilisables pour représenter des connaissances.

Ontologie fondamentale [fundational ontology] : ontologie formelle de haut niveau dont les termes sont très précisément définis et, le plus souvent, sont relatifs à un seul thème, par exemple, les relations spatiales, temporelles, partie-de.

Ontologie lexicale (par exemple, de langage naturel) : ontologie dont les termes formels représentent le sens de termes informels d'un lexique ou d'un langage naturel.

Ontologie hétérogène en interne : ontologie qui n'est pas homogène en interne.

Ontologie homogène en interne : ontologie dans laquelle toutes les descriptions sont basées sur quelques primitives, par exemple, quelques types de relation sémantique.

Ontologies globalement homogènes (ou homogènes entre elles) : ontologies dont l'agrégation est homogène en interne.

Phrase [(declarative or not) sentence] : une combinaison de termes et de sous-phrases décrivant une chose.

Précision sémantique : plus les définitions de deux choses sont précises (donc, entre autres, ni sur-spécialisées, ni sur-généralisées), plus elles peuvent être comparées. Ainsi, plus des descriptions sont précises, plus les requêtes pour les retrouver pourront être précises. Par exemple, la définition "toute voiture a exactement 4 roues" est plus précise que "toute voiture a exactement 4 parties".

Primitive : terme n'ayant pas de définition.

Programme : une dafp d'un processus – par exemple, via le langage Java – par opposition à une description déclarative d'un processus, par exemple, via un réseau de Pétri ou des règles. Dans tous les cas, ces descriptions de processus sont des ensembles de phrases dont au moins un élément est un processus. Une description fonctionnelle est une dafp de processus écrite dans un paradigme fonctionnel, typiquement via un langage de programmation fonctionnel, comme Haskell, par exemple.

Quantificateur de la logique du premier ordre : quantificateur existentiel (i.e., "il existe", c'est à dire "au moins un") et un quantificateur universel ("quel que soit" ["for all"], "chaque" ["every", "each"]).

Quantificateur numérique : quantificateur individuel (par exemple, "2", "entre 2 et 3", "au moins 4") ou statistique (par exemple, "35%").

Référence : identifiant, variable ou fonction.

Relation lexicale : objet formel (non sémantique) ne pouvant être utilisé qu'entre des symboles informels. Voici quelques types de relation lexicale : "homonyme", "antonyme", les relations entre chaînes de caractères.

Relation sémantique : objet d'information composé d'un noeud relation et des objets d'information qui sont directement reliés par ce noeud relation. Dans cette thèse, les types de relation sémantique seront en italique et le mot "sémantique" après "relation" sera généralement omis. Voici quelques exemples de types de relations sémantiques :

- généralisation (des sous-types sont, par exemple, *logical_deduction*, *supertype*),
- sous-partie (dont les sous-types sont, par exemple, *subprocess*, *physical_part*) ;
- relations depuis un processus (par exemple, *input*, *output*, *destinataire*, *acteur*).

Une relation est orientée et peut être unaire, binaire, ternaire, etc. Une relation binaire a un seul noeud origine et un seul noeud destination. Le W3C, par exemple, utilise un vocabulaire différent. Ainsi, dans le modèle RDF :

- une relation est appelée un triplet ;
- un type de relation est appelé une propriété ou un prédicat ;
- un noeud origine est appelé un sujet ou une source ;
- un noeud destination est appelé un objet ou une destination.

Relation *specialisation* : relation subtype ou instance.

Relation structurelle : désigne la relation "parent/child" (généralement implicite) ou "attribut" dans i) les langages orientés objets (e.g., Java), de description de documents (e.g., XML) ou de bases de données, ou ii) les structures de données comme les tableaux associatifs ou les "enregistrements" [records]. Contrairement aux relations sémantiques des LRCs, les relations structurelles ne sont pas des entités de 1er ordre : elles ne peuvent être référées, elles n'ont pas de types (donc pas de sémantiques explicites exploitables automatiquement) et ne peuvent donc pas être organisées par des relations subtype. L'utilisation de relations sémantiques (voir définition ci-après) plutôt que structurelles a des avantages qui sont décrits entre autres sur cette page : <https://www.w3.org/DesignIssues/RDF-XML.html>. Ainsi, le W3C recommande l'utilisation de RDF plutôt que de XML pour le Web Sémantique.

Représentation de connaissances (RC) : phrase ayant une interprétation dans une logique et donc interprétable par un moteur d'inférences logique.

Représentation infixée d'un lien ou d'une fonction *r* (dans une notation textuelle) : notation textuelle dans laquelle le type de *r* est positionné entre la source et la destination de *r*, comme dans la notation Turtle. Ainsi, en Turtle, la phrase "Tom is on a mat" peut s'écrire :

`ex:Tom ex:place ex:Cat. // "ex" est l'identifiant de l'espace de nommage dans lequel Tom est déclaré`

Représentation postfixée d'un lien ou d'une fonction *r* (dans une notation textuelle) : notation textuelle dans laquelle le type de *r* est positionné après la source et la destination de *r* comme, par exemple, dans la notation polonaise inversée [reverse polish notation]. Ainsi, dans un LRC qui utiliserait une telle notation, la phrase "Tom est un chat" pourrait s'écrire : `Tom mat place`.

Représentation préfixée d'un lien ou d'une fonction *r* (dans une notation textuelle) : notation textuelle dans laquelle le type de *r* est positionné avant la source et la destination de *r*, comme dans la notation CLIF. Ainsi, en CLIF, la phrase "Tom is on a mat" peut s'écrire : `(place Tom mat)`.

Réutilisabilité sémantique : plus des choses sont sémantiquement comparables, plus elles sont sémantiquement réutilisables. C'est à dire, plus elles peuvent être sémantiquement comparées, plus elles peuvent être retrouvées ou agrégées. Ainsi, plus des choses peuvent être retrouvées ou agrégées, plus elles peuvent être sémantiquement réutilisées. Par exemple, la définition "toute voiture a exactement 4 roues" peut être automatiquement retrouvée par les utilisateurs qui recherchent des "choses avec exactement 4 parties ayant chacune une forme arrondie".

Sémantique opérationnelle : une sémantique pour des descriptions écrites dans un langage de programmation.

Situation : toute chose qui "arrive" dans une région réelle ou imaginaire de l'espace et du temps, par exemple, un processus ou un état.

Structure de données : ensemble de relations structurelles entre des données. Par exemple, un tableau est une structure de données.

Terme : valeur (par exemple, une valeur atomique comme un booléen ou un caractère, ou une valeur structurée comme une collection) ou une référence.

Type : référence à un ensemble de choses (les instances du type) et représentant certaines caractéristiques communes à ses instances. Un type peut être un type de concept [concept type] ou un type de relation [relation type].

"Type" de structure de données : spécification de relations structurelles entre des données depuis une même source. Par exemple, en XML, un "type" de structure de données peut être décrit dans une DTD ; dans un langage orienté objet, une classe est un "type" de structure de données.

Une sémantique : un sens, c'est à dire, une signification.

Web sémantique : ensemble des RCs qui utilisent les LRCs promus par le W3C.

Annexes

Annexe 1 : Quelques fonctions de parcours et de manipulation de structure de données

Cette annexe présente les fonctions de parcours et de manipulation de structures de données que j'ai spécifiées en FL et en KIF. Les types de relation utilisés ont été définis par Philippe Martin (pm). Les types de concept de Philippe Martin (pm) ne sont utilisés que pour situer mes propres types de concept (jb).

```
jb#exploiting_and_creating-or-modifying_a_set_with_a_total_order_relation //_by_applying_a_function_to_it
< pm#exploiting_and_creating-or-modifying_a_collection, //_by_applying_a_function_to_it,
> (jb#create_subset
  pm#input: 1 jb#filter_function ?fct
            1 pm#set ?filter_parameters,
  = jb#filter_set,
  pm#part: 1 pm#set_map,
  >:= [_ pm#language: pm#KIF] $(
    (pm%def_fct pm%sub_set ?filter_parameters ?fct :=
      (pm%set_map ?filter_parameters ?fct))),
  > (jb#upper-set
    pm#input: 1 pm#set ?set,
    pm#part: 0..1 jb#element_in_subset ?filter_function
             0..1 jb#next_element
             0..1 jb#exploiting_and_creating-or-modifying_a_set ?aggregate_function,
    pm#output: (1 pm#set jb#subset: ?set) )
);

jb#function_returning_an_element_member_of_a_set
pm#annotation: "Returns an element in a set",
pm#output: (1 pm#thing pm#member_of: pm#set ?set),
< pm#function,
> (jb#function_using_a_filter_on_a_set
  > (jb#function_returning_an_element_of_an_ordered_set
    > (jb#element_from_indice
      > jb#first_element_of_a_set_ordered_with_indices
      jb#element_of_a_totally_and_directly_ordered_set
      jb#element_of_a_partially_and_directly_ordered_set
    )
    jb#next_element
  )
);

jb#next_element
pm#annotation: "Returns the element next to the element specified in input",
pm#input: 1 pm#set ?set
          (1 pm#thing ?input pm#member: ?set),
pm#output: (1 pm#thing ?output pm#member: ?set, != ?input);

jb#element_of_an_ordered_set
pm#annotation: "Returns the element at the specified indice of one
               pm#set_with_at_least_one_total_order_relation ordered by one jb#total_order_relation.",
pm#input: 1 pm#set_with_at_least_one_total_order_relation ?set
          1 pm#thing ?element_or_indice,
pm#output: 1 pm#thing;
```

```

jb#element_from_indice
  pm#annotation: "Returns the element at the specified indice of one pm#totally_ordered_set_with_indices
                 ordered by one jb#total_order_relation.",
  pm#input: 1 pm#totally_ordered_set_with_indices ?set 1 pm#thing ?indice,
  >:= [_ pm#language: pm#KIF] $(
    (pm%deffunction jb%element_from_indice (?set ?indice) :=
      (if (and (pm#totally_ordered_set_with_indices ?set)
              (holds jb%access_function ?set ?access_function))
          (map ?access_function ?set ?indice)
          nil) ) )$;

jb#first_element_of_a_set_ordered_with_indices
  pm#annotation: "Returns the first element of one pm#totally_ordered_set_with_indices ordered by one
                 jb#total_order_relation.",
  pm#input: 1 pm#totally_ordered_set_with_indices ?set 1 jb#total_order_relation ?rel,
  >:= [_ pm#language: pm#KIF] $(
    pm%deffunction jb%first_element_of_a_set_ordered_with_indices (?set ?rel) :=>
      (jb%first_element_of_a_totally_and_directly_ordered_set (jb%indices_of_set ?set) ?rel)
    )$

jb#element_of_a_totally_and_directly_ordered_set
  pm#annotation: "Returns the specified element of one pm#set_with_at_least_one_total_order_relation ordered
                 by one jb#total_order_relation.",
  >:= [_ pm#language: pm#KIF] $(
    (pm%deffunction jb%element_of_a_totally_and_directly_ordered_set (?set ?element_or_indice) :=
      (if (and (jb#totally_and_directly_ordered_set ?set)
              (pm#member ?set ?element_or_indice))
          (?element_or_indice)
          nil) ) )$;

jb#first_element_of_a_totally_and_directly_ordered_set
  pm#annotation: "Returns the element at the first indice of one
                 pm#set_with_at_least_one_total_order_relation ordered by one jb#total_order_relation.",
  pm#input: 1 pm#set_with_at_least_one_total_order_relation ?set 1 jb#total_order_relation ?rel,
  >:= [_ pm#language: pm#KIF] $(
    (pm%deffunction jb%first_element_of_a_totally_and_directly_ordered_set ?set ?rel :=
      (if (and (= ?set (setof @items ?x))
              (jb#order_relation_for_a_set_with_at_least_one_total_order_relation ?rel ?set)
              (pm#each-in (setof @items) '?i '(holds ?rel ?x ?i)))
          ?x
          nil) ) )$;

jb#element_of_a_partially_and_directly_ordered_set
  pm#annotation: "Returns the specified element of one pm#set_with_a_partial_order_relation ordered by one
                 jb#total_order_relation.",
  >:= [_ pm#language: pm#KIF] $(
    (pm%deffunction jb%element_of_a_partially_and_directly_ordered_set (?set ?element_or_indice) :=
      (if (and (jb#set_with_a_partial_order_relation ?set)
              (pm#member ?set ?element_or_indice))
          (?element_or_indice)
          nil) ) )$;

```

```

jb#first_element_of_a_partially_and_directly_ordered_set
  pm#annotation: "Returns the element at the first indice of one
    pm#set_with_at_least_one_total_order_relation ordered by one jb#total_order_relation.",
  pm#input: 1 pm#set_with_a_partial_order_relation ?set 1 jb#total_order_relation ?rel,
  >:= [_ pm#language: pm#KIF] $(
    (pm%deffunction jb%first_element_of_a_partially_and_directly_ordered_set ?set ?rel :=
      (if (and (= ?set (setof @items ?x))
        (jb%order_relation_for_a_set_with_at_least_one_partial_order_relation ?rel ?set)
        (pm%each-in (setof @items) '?i '(holds ?rel ?x ?i)))
        ?x
        nil) ) )$);

pm#relation_from_collection
  > (jb#order_relation_for_a_set_with_at_least_one_partial_order_relation
    part: 1 jb#partial_order_relation,
    >:= [_ pm#language: pm#KIF] $(
      (pm%def_rel jb%order_relation_for_a_set_with_at_least_one_partial_order_relation ?rel ?set :=>
        (and (jb%set_with_at_least_one_partial_order_relation ?set)
          (math%partial_order_relation ?rel (jb%indices_of_set ?set)))) )$
    )
  (jb#order_relation_for_a_set_with_at_least_one_total_order_relation
    part: 1 jb#total_order_relation,
    >:= [_ pm#language: pm#KIF] $(
      (pm%def_rel jb%order_relation_for_a_set_with_at_least_one_total_order_relation ?rel ?set :=>
        (and (jb%set_with_at_least_one_total_order_relation ?set)
          (math%total_order_relation ?rel (jb%indices_of_set ?set)))) )$
    );

pm#list_from_set
  > (jb#list_from_ordered_set
    pm#annotation: "Returns one pm#list from one pm#set_with_at_least_one_total_order_relation ordered by
      one jb#total_order_relation.",
    pm#input: 1 pm#set_with_at_least_one_total_order_relation ?set 1 jb#total_order_relation ?rel,
    >:= [_ pm#language: pm#KIF] $(
      (pm%def_fct pm%list_from_ordered_set (?set ?rel) :=
        (if (and (= ?set (setof @items ?first_element))
          (= ?first_element
            (jb%first_element_of_a_totally_and_directly_ordered_set ?set ?rel) ) )
          (listof ?first_element
            (jb%list_from_ordered_set (jb%rest_of_an_ordered_set (setof @items) ?rel)) )
          nil) ) )$
    );

```

```

pm#function
> (jb#sub_set
  >:= [_ pm#language: pm#KIF] $(
    (pm%def_rel jb%sub_set ?set ?subset :=
      (and (pm%set ?set) (pm%set ?subset)
        (forall (?var) (=> (pm%member ?subset ?var) (pm%member ?set ?var))))
    ) )$
)
(jb#ordered_sub_set_of_a_set
  >:= [_ pm#language: pm#KIF] $(
    (pm%def_fct jb%ordered_sub_set_of_a_set (?set ?rel) :=>
      (setofall ?var
        (and (= ?subset (setof @items ?var))
          (jb%order_relation_for_a_set_with_an_order_relation ?rel ?subset)
          (jb%sub_set ?set ?subset) ) )
    ) )$,
  > (jb#partially_ordered_sub_set_of_a_set
    >:= [_ pm#language: pm#KIF] $(
      (pm%def_fct jb%partially_ordered_sub_set_of_a_set (?set ?rel) :=>
        (jb%order_relation_for_a_set_with_a_partial_order_relation ?rel ?subset)
      ) )$,
    > (jb#totaly_ordered_sub_set_of_a_set
      >:= [_ pm#language: pm#KIF] $(
        (pm%def_fct jb%totaly_ordered_sub_set_of_a_set (?set ?rel ?first ?last) :=>
          (and (jb%order_relation_for_a_set_with_a_total_order_relation ?rel ?subset)
            (= (jb%first_element_of_a_set_ordered_with_indices ?subset ?rel) ?first)
            (= (jb%last_element_of_a_set_ordered_with_indices ?set ?rel) ?last) )
          ) )$
    )
  )
);

```


Annexe 2 : Quelques fonctions du framework LAS représentées via Structure_map

Cette annexe présente quelques fonctions du framework LAS que j'ai représenté via Structure_map en FL et en KIF. Ces fonctions sont préfixées par bw#.

```
// 1) Organisation
```

```
jb#sub_set
> (jb#sub_set_from_extreme_values
  pm#input: 1 pm#set ?set 1 jb#total_order_relation ?total_order_rel
    (1 pm#thing ?min pm#member of: ?set) (1 pm#thing ?max pm#member of: ?set),
  >:= [_ pm#language: pm#KIF] $(
    (pm%def_fct jb%sub_set_from_extreme_values ?set ?total_order_rel ?min ?max :=
      (if (and (pm%member ?set ?min) (pm%member ?set ?min))
        (jb%totaly_ordered_sub_set_of_a_set ?set ?total_order_rel ?min ?max pm%true)) ))$,
  > (jb#sub_string
    pm#input: 1 pm#string ?set 0..1 pm#function ?aggregate_function 0..* pm#function ?input_successor,
    pm#output: (1 pm#string pm#part_of: ?set),
    > (jb#right_part_of_string
      < jb#upper_set,
      > bw#TStrHelper.StripFromFront
      delphi#AnsiRightStr
      bw#TStrHelper.StripFrontChars
      bw#TStrHelper.StripFrontChar
      (bw#TStrHelper.StripFirstToken
        > bw#TStrHelper.StripFirstWord)
      delphi#ExtractFileExt
      bw#TStrHelper.GetLastToken
      delphi#ExtractFilePath
    )
    (jb#left_string
      >:= [_ pm#language: pm#KIF] $(
        (pm%def_fct jb%right_string ?set ?total_order_rel ?min ?max :=>
          (= ?min (jb%first_element_of_a_totally_and_directly_ordered_set ?set
            ?total_order_rel))
          ) )$,
      > bw#TStrHelper.StripLastToken
      bw#TStrHelper.StripEndChars
      bw#TStrHelper.StripEndChar
      bw#TStrHelper.StripBlanks
      delphi#ExtractFileDir
      delphi#ExtractFileDrive
      delphi#AnsiLeftStr
    )
    bw#TStrHelper.GetInnerString
  );
```

```
// 2) Définitions
```

```
bw#TStrHelper.StripFromFront
  pm#input: 1 pm#string ?s
            1 kif#integer ?len,
  pm#part: bw#TStrHelper.ReverseStr
            bw#TStrHelper.Shorten,
  >:= [_ pm#language: delphi] $(
    class function TStrHelper.StripFromFront (S: string; Len: Integer): string;
    begin
      S := TStrHelper.ReverseStr(S);
      S := TStrHelper.Shorten(S, Len);
      S := TStrHelper.ReverseStr(S);
      Result := S;
    end;)$
  [_ pm#language: pm#KIF] $(
    (pm%def_fct bbw%TStrHelper.StripFromFront ?s ?len :=
      (if (and (= (Length ?StripPart) ?len)
                (= ?s (append ?StripPart ?Result)))
          ?Result) ) )$
  [_ pm#language: pm#KIF] $(
    (pm%def_fct bbw%TStrHelper.StripFromFront (?s ?len) :=
      (pm#structure_map
        ?s
        (lambda doNotFilterOut-element-if-indexOfElementInInputStructure-isLessThan-len
          (?element ?indexOfElementInInputStructure)
          (if (< (pm#index-in ?s ?element) ?len) nil
              ?element)
          ) )
        jb%string-to-set
        (lambda append-result-ifNotNil-to-output-string (?result ?output)
          (if (not (null ?result)) (pm%string-append ?result ?output) nil)
          ) )
      ) ) )$;

bw#TStrHelper.StripFirstWord
  pm#input: 1 pm#string ?s,
  pm#part: delphi#pos delphi#length delphi#move bw#TStrHelper.set_length,
  >:= [_ pm#language: pm#KIF] $(
    (pm%def_fct bbw%TStrHelper.StripFirstWord ?s :=
      (pm%structure_map ?s
        nil
        (lambda append-result-after-first-word
          (?result ?s ?output)
          (if (pm%member (Nthrest ?s (code-char 32))
                        ?result)
              (pm%string-append ?result ?output)
              nil))
        (lambda next-element
          (?input-structure ?indexOfElementInInputStructure)
          (pm%get-next-element-in-string ?input-structure
            ?indexOfElementInInputStructure))
          ?s) ) )$;
```

```

bw#TStrHelper.StripFrontChars
  pm#input: 1 pm#string ?s 1 jb#character ?ch,
  pm#part: delphi#length delphi#copy bw#TStrHelper.min_index bw#TStrHelper.max_index,
  >:= [_ pm#language: pm#KIF] $(
    (pm%def_fct bbw%TStrHelper.StripFrontChars (?s ?ch) :=
      (pm%structure_map ?s
        nil
        (lambda append-result-ifNotCh-to-output-string
          (?result ?ch ?output)
          (if (or (/= ?result ?ch)
              (/= ?output nil))
              (pm%string-append ?result ?output)))
        (lambda next-element
          (?input-structure ?indexOfElementInInputStructure)
          (pm%get-next-element-in-string ?input-structure
            ?indexOfElementInInputStructure) )
        ?ch ) ) )$;

```

```

bw#TStrHelper.StripFrontChar
  pm#input: 1 pm#string ?s 1 jb#character ?ch,
  pm#part: delphi#length delphi#copy bw#TStrHelper.min_index bw#TStrHelper.max_index,
  >:= [_ pm#language: pm#KIF] $(
    (pm%def_fct StripFrontChar (?s ?ch) :=
      (pm%structure_map ?s
        nil
        (lambda append-result-if-notFirstOrNotCh-to-output-string
          (?result ?s ?ch ?output)
          (if (or (/= ?ch ?result)
              (/= (jb%first_element ?s) ?result))
              (pm%string-append ?result ?output)))
        (lambda next-element
          (?input-structure ?indexOfElementInInputStructure)
          (pm%get-next-element-in-string ?input-structure
            ?indexOfElementInInputStructure))
        ?s
        ?ch ) ) )$;

```

```

[_ pm#language: delphi#delphi] $(
  class function TStrHelper.StripFrontChar(S: string; Ch: Char): string;
  begin
    if (Length(S)<>0) and (S[TStrHelper.MinIndex] = Ch) then
      S := Copy(S,TStrHelper.MinIndex+1,TStrHelper.MaxIndex(S));
    Result := S;
  end;)$;

```

```

bw#TStrHelper.StripFirstToken
  pm#input: 1 pm#string ?s 1 jb#character ?ch,
  pm#part: delphi#pos delphi#length delphi#move bw#TStrHelper.set_length,
>:= [_ pm#language: pm#KIF] $(
  (pm%def_fct StripFirstToken ?s ?ch :=
    (pm%structure_map ?s
      nil
      (lambda append-result-after-first-token
        (?result ?s ?ch ?output)
        (if (pm%member (Nthrest ?s ?ch)
          ?result)
          (pm%string-append ?result ?output)
          nil))
      (lambda next-element
        (?input-structure ?indexOfElementInInputStructure)
        (pm%get-next-element-in-string ?input-structure
          ?indexOfElementInInputStructure)))
    ?s
    ?ch) ))$
[_ pm#language: delphi#delphi] $(
class function TStrHelper.StripFirstToken(S: string; Ch: Char): string;
var
  i, Size: Integer;
begin
  {$IFDEF MANAGED}
    i := S.IndexOf(Ch);
    if i = -1 then begin
      Result := S;
      Exit;
    end;
    Result:=S.Substring(i+1);
  {$ELSE}
    i := Pos(Ch, S);
    if i = 0 then begin
      Result := S;
      Exit;
    end;
    Size := (Length(S) - i);
    {$IFDEF DELPHI14_UP}
    Move(S[i + 1], S[1], 2*Size);
  {$ELSE}
    Move(S[i + 1], S[1], Size);
  {$ENDIF}
    TStrHelper.SetLength(S, Size);
    Result := S;
  {$ENDIF}
end;);$

```

```

bw#TStrHelper.GetLastToken
> delphi#ExtractFileName,
pm#input: 1 pm#string ?s 1 jb#character ?token,
:=> [_ pm#language: pm#KIF] $(
  (pm%def_fct bbw#TStrHelper.GetLastToken ?s ?separator :=
    (pm%structure_map ?s
      (lambda (?s ?indexOfElementInInputStructure)
        (if (< (jb%last_indice_of_element ?separator ?s)
            ?indexOfElementInInputStructure )
          ?indexOfElementInInputStructure
          nil ) )
      (lambda (?input-string ?indexOfElementInInputString)
        (pm%get-next-tokenIndex-in-string ?input-string
          ?indexOfElementInInputString
          ?separator) )
      (lambda (?result_index)
        (pm%string-append (jb%element_from_indice ?s ?result_index) ?output))
    ) ) );

```

```

delphi#AnsiRightStr
pm#input: delphi#string ?a_text delphi#integer ?a_count,
pm#part: delphi#AnsiCountChars delphi#AnsiMidStr,
>:= [_ pm#language: pm#KIF] $(
  (pm%def_fct delphi#AnsiRightStr ?a_text ?a_count :=
    (bbw#TStrHelper.StripFromFront ?a_text ?a_count)
  ));

```

```

bw#TStrHelper.StripEndChars
pm#input: 1 pm#string ?s 1 jb#char ?ch,
>:= [_ pm#language: delphi] $(
  class function TStrHelper.StripEndChars(S: string; Ch: Char): string;
  var
    i: Integer;
  begin
    i := TStrHelper.MaxIndex(s);
    while (length(S) > 0) and (S[i] = Ch) do begin
      Delete(S,i,1);
      Dec(i);
    end;
    Result := S;
  end; )$
  [_ pm#language: KIF] $(
    (pm%def_fct bbw#TStrHelper.StripEndChars ?s ?rel ?ch :=
      (pm%iterator_on_integer_range (jb#Delete ?s
        ?var ?s (bbw#TStrHelper.MaxIndex ?s) 1
        (and (= ?ch (jb%element_referenced_in_a_relation ?s ?var))) ) )
    ));

```

```

bw#TStrHelper.StripEndChar
>:= [_pm#language: delphi] $(
  class function TStrHelper.StripEndChar(S: string; Ch: Char): string;
  var
    i: Integer;
  begin
    i := TStrHelper.MaxIndex(S);
    if (length(S) > 0) and (S[i] = Ch) then begin
      Delete(S,i,1);
      Dec(i);
    end;
    Result := S;
  end; )$;

```

delphi#AnsiLeftStr

```
pm#input: delphi#string ?a_text delphi#integer ?a_count,
pm#part: delphi#AnsiCountElems delphi#lentgh delphi#copy,
>:= [_ pm#language: pm#KIF] $(
  (pm%def_fct delphi%AnsiLeftStr ?a_text ?a_count :=
    (if (and (> ?a_count 0) (> (delphi%lentgh ?a_text) 0))
      (delphi%copy ?a_text 1 (delphi%AnsiCountElems ?a_text 1 ?a_count))
      null)
  ));
```

delphi#AnsiMidStr

```
pm#input: delphi#string ?a_text delphi#integer ?a_start delphi#integer ?a_count,
pm#part: delphi#length delphi#ansi_count_elms delphi#copy,
>:= [_ pm#language: pm#KIF] $(
  (pm%def_fct delphi%ansi_mid_str ?a_text ?a_start ?a_count :=
    (if (and (> ?a_count 0)
      (> (delphi%length ?a_text) 0)
      (> (delphi%length ?a_text) ?start)
      (if (> ?a_start 0)
        (= ?start
          (+ (delphi%ansi_count_elms ?a_text 1 (- ?a_start 1))
            1))
        (= ?start 1)) )
      (delphi%copy ?a_text
        ?start
        (delphi%ansi_count_elms ?a_text ?start ?a_count))
      null) ));
```

bw#TStrHelper.ReplaceChars

```
pm#input: 1 pm#string ?s 1 jb#char ?OldCh 1 jb#char ?NewCh,
>:= [_ pm#language: delphi] $(
  Len := Length(s);
  for i := 1 to Len do
    if S[i] = OldCh then
      S[i] := NewCh;
  Result := S; )$
[_ pm#language: jb#CS] $(
  return S.Replace(OldCh, NewCh); )$;
```

Annexe 3 : Documentation de BW_Notation

Cette annexe est un extrait d'un document de GTH qui présente la version 2 de RichGraph. J'ai choisi cet extrait car il décrit la version la plus à jour de BW_Notation. Cette notation est structurellement proche de la version 2 de RichGraph et est actuellement développée par Anil Cassam-Chenaï. Je n'ai contribué ni au développement de cette notation, ni à l'écriture de sa documentation.

BW_notation

On peut définir directement des expressions sémantiques en se basant sur une syntaxe de logique d'ordre 1 :

La grammaire de cette syntaxe est la suivante :

```
"Name"      = 'BW_Notation_Parser'
"Author"    = 'GTH'
"Case Sensitive" = 'true'
"Start Symbol" = <LogicProgram>
```

! ----- Sets

```
{UpperLetter} = [ABCDEFGHJKLMNOPQRSTUVWXYZ]
{LowerLetter} = [abcdefghijklmnopqrstuvwxyz]
!{Letter} = [abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]
!{LetterOrDigit} = [abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789]
```

! ----- Terminals

```
Variable = {Letter}{Alphanumeric}*
Symbol = ( ({Letter} {Alphanumeric})*? '#' ) {Letter} {Alphanumeric}*
Integer = {Number}+
```

! ----- Rules

```
<LogicProgram> ::= <Sentence> | <Sentence> ';' <LogicProgram>
<Sentence> ::= <AtomicSentence> | <ComplexSentence>
<AtomicSentence> ::= Symbol '(' ')' ! prédicat d'arité 0 - pour différencier d'une constante
                    | Symbol '(' <Terms> ')' ! prédicat d'arité >0
                    ! | <Term> '=' <Term>
                    | <Term> '=' <Sentence>
```

```
!<Predicate> ::= Symbol
```

```
<ComplexSentence> ::= '(' <Sentence> ')'
                    | '[' <Sentence> ']'
                    | 'not' <Sentence>
                    | <Sentence> 'and' <Sentence>
                    | <Sentence> 'or' <Sentence>
                    | <Sentence> '=>' <Sentence>
                    | <Sentence> '<=>' <Sentence>
                    | <QuantifiedVariables> <Sentence>
```

```
<QuantifiedVariables> ::= <QuantifiedVariable>
| <QuantifiedVariable> ',' <QuantifiedVariables>
```

```

<QuantifiedVariable> ::= <Quantifier> Variable
| <Quantifier> Variable in Symbol

<Term> ::= <ConstantSymbol>
| Variable
| <FunctionApplication>

<FunctionApplication> ::= Symbol '(' <Terms> ')

<ConstantSymbol> ::= Symbol

<Terms> ::= <Term>
| <Term> ',' <Terms>

<Quantifier> ::= <UniversalQuantifier> | <ExistentialQuantifier>

<UniversalQuantifier> ::= 'forall' | 'any' | 'every'

<ExistentialQuantifier> ::= 'exists' <ExistentialQuantities>

<ExistentialQuantities> ::= | Integer | Integer '..' <ExtendedInteger>

<ExtendedInteger> ::= Integer | '*'

```

Des expressions types sont données dans le tableau ci-dessous :

Expression	Signification
any x in #E, any y in #F, #V(x,y) que l'on peut aussi écrire en insérant les quantificateurs à l'intérieur des relations #V(any (0) x:#E, any(1) y:#F)	définit #V comme une relation a deux variables ayant comme support #E,#F Exemple : any directeur :#Dirigeant, any entreprise :#Entreprise, #dirige(directeur,entreprise)
any #W(x) <=> any x, exists m..n y : #A #V(x,y) (ici m et n sont des entiers étendus à remplacer incluant l'infini est noté '*')	définit #W comme une relation a une variable (un ensemble) obtenue à partir de #V en prenant tous les éléments liés par #V à un élément appartenant à #W. Exemple : any x #gerant(x)<=> any directeur :#Dirigeant, exists entreprise :#PME #dirige(directeur,entreprise) ; any x #gerant_de_plusieurs_PME(x) <=> any directeur :#Dirigeant, exists 2..* entreprise :#PME #dirige(directeur,entreprise)
every entreprise : #Entreprise, exists 1 dirigeant : #Dirigeant #dirige(directeur,entreprise)	définit une contrainte sur la relation entre une entreprise et son dirigeant. Ce dernier doit exister et être unique.
any x : #E, any y : #F #A(x) and #B(x,y)	définit une relation binaire à partir d'une relation unaire #A et d'une relation binaire #B Exemple : #Jeune_dirigeant_marie = any dirigeant : #Dirigeant, exists woman :#Woman, #Jeune(dirigeant) and #marie(dirigeant,woman)
Utilisation d'un contexte : il suffit de mettre une variable de type #context dans la définition	every directeur : #dirigeant , exists entreprise, context: #context #dirige(directeur,entreprise,context)

Syntaxe de chemin de graphe (uniquement pour les relations binaires - inspirée en FL de WebKB/PM)

pour une relation binaire entre instances on peut écrire :

```
#B #s: #C #D
```

avec la capacité d'utiliser des parenthèses pour délimiter un graphe :

```
#A #r: (#B #s: #C)
```

ou encore si on a des quantificateurs

```
any pizza, exists aliments
```

```
    pizza:#Pizza #hasTopping: aliments:#Aliments
```

qui avec les quantificateurs intégrés devient :

```
any (0) pizza:#Pizza
```

```
    #hasTopping: exists(1) aliments:#Aliments
```

Annexe 4 : Documentation de la version 2 de RichGraph

Cette annexe est le début d'un document de GTH sur la version 2 RichGraph. J'ai choisi cet extrait parce qu'il est le plus simple à comprendre et que le reste de la documentation n'est pas à jour. Je n'ai écrit ni les structures de données de cette version de RichGraph, ni leur documentation.

Cette annexe permet de comparer la version 1 de RichGraph (à laquelle j'ai contribué) et la version 2 de RichGraph (à laquelle je n'ai pas contribué).

1. Introduction

Le Framework Logicells dispose d'une couche de graphe riche intégrée. Nous entendons par cela une couche de graphe qui permet à la fois :

- la manipulation des données brutes
- la mise en place d'algorithmes de type recherche opérationnelle pour, par exemple, optimiser des flux de marchandises, trouver un chemin optimal, etc
- des connaissances assertionnelles et terminologiques

le tout dans une même structure. Une description théorique de la manipulation de graphe riche peut être obtenue dans le document XXX.

On pourra donc à la fois donc construire des graphes sans sémantique et conçus pour des manipulations de type calculatoire et des graphes disposant d'une structure sémantique intégrée afin de manipuler directement des connaissances. De plus, nous avons basé toute l'architecture haute de notre serveur d'application (dont la couche de composants) sur cette couche de graphe. Cela veut donc dire que les informations et les méta-informations applicative alimentent directement le graphe et peuvent donc être manipulée (requêtes, machine learning, ...) de manière cohérente.

Pour synthétiser des idées clés des graphes riches que nous allons ensuite approfondir plus loin :

- On appelle nœud abstrait à la fois des sommets (appelés aussi nœud ou nœud simple) et des relations
- Les relations relient des nœuds abstraits et peuvent donc relier des nœuds simples entre eux mais aussi des relations entre elles et des nœuds simples à des relations.
- Les nœuds simples peuvent avoir un 'type de valeur' qui est en dehors de l'information de graphe (et donc de l'information sémantique si on utilise le graphe pour un traitement sémantique). Par exemple, le nœud simple peut correspondre à une chaîne de caractère vidéo, son type de valeur est donné par une interface IStringValue et sa valeur correspond à la récupération du fichier vidéo associé sous forme d'un flux binaire. On peut aussi faire correspondre un nœud simple sans type de valeur à un objet du monde physique ou une abstraction (par exemple, le corps des nombres réels) auquel on n'a pas directement accès.
- A ce type de valeur 'hors graphe' on peut ajouter de multiples annotations sur le nœud en lui rajoutant là aussi des 'valeurs' mise sous forme de propriétés dans le graphe (on parlera donc de propriété pour ne pas mélanger avec les types de valeur et les valeurs décrites précédemment). On a alors l'avantage de pouvoir manipuler les propriétés et en sortir des résultats (et de l'inférence si on a de l'information sémantique). Cependant, l'information présente dans le graphe est généralement beaucoup plus légère que l'information présente dans la réalité représentée par le graphe. On peut considérer les propriétés comme une projection de la réalité sur un modèle de graphe pour obtenir des spécifications corrélées à un problème réel.

2. Les API de manipulation des graphes riches

Nous allons découvrir dans les prochains paragraphes la structure des graphes riches et leur manipulation en partant de la structure des API mise à disposition pour leur manipulation.

2.1. Création d'un graphe conceptuel riche

2.1.1. En Pascal Objet

On peut facilement construire du graphe riche sous trois formes :

- Un graphe mémoire : TMemoryRichGraph
- Un graphe disposant d'une structure de persistance : TPersistentRichGraph
- Un graphe de couche/transactionnel : TLayerRichGraph

Le premier type de graphe permet d'avoir tous les sommets et relations en mémoire, le deuxième permet de décharger la mémoire en chargeant/déchargeant des segments sur le disque. Le troisième type de graphe permet de créer des surcouches 'transactionnelles' d'un des deux autres types de graphe afin de permettre des traitements/fusions atomiques pour le rajout de fragments complexes de graphes.

Pour créer un graphe riche en mémoire

```
LGraph := TMemoryRichGraph.Create;  
LGraph.Initialize;
```

Il suffit de le créer, puis de lancer son initialisation. Cette dernière va créer un certain nombre de relations par défaut nécessaires au traitement sémantique.

Dans le cas d'un graphe persistant, il faut déclarer un chemin de dossier dans lequel sera rangé les données persistantes du graphe.

```
LGraph:=TPersistentRichGraph.Create(nil, nil, GraphPath);  
Lgraph.Initialize;
```

Finalement un graphe de surcourage/transaction se crée à partir d'un graphe déjà existant via l'appel de BeginTransaction

2.1.2. En Python

On démarre une session Python et il suffit pour créer un graphe riche en mémoire de taper :

```
>>> import BaboukWebPy as bw  
>>> p=bw.MemoryRichGraph(None)  
>>> p.Initialize()
```

On peut tester la bonne initialisation en rajoutant une valeur.

```
>>> s  
<IStringValue at 20FF308>  
>>>
```

2.2. Type universel de sommets/rerelations

Les sommets et relations du graphe sont des pointeurs et peuvent tous être transformés en IEnvironnementValue par un simple cast.

En plus de cette interface universelle, les différents sommets disposent d'interfaces qui leur sont propres.

2.3. Création de sommets dans le graphe

Dans un graphe riche tout peut être un sommet de relation:

- Des sommets primitifs ayant des types de valeurs non sémantiques : node, string, integer, float, etc.
- Des relations elles-mêmes (ce qui indique qu'on peut avoir des arcs qui partent d'un arc pour rejoindre un sommet ou un autre arc)
 - Les relations unaires seront appelées des concepts

On peut facilement construire un sommet atomique en utilisant un constructeur. L'exemple suivant montre leur usage dans le cas de types primitifs entiers (IIntegerValue) et string (IStringValue):

```

try
  LGraph:=TMemoryRichGraph.Create;
  LGraph.Initialize;

  i:=TIntegerValue.Create(LGraph,'Un',1);
  j:=TIntegerValue.Create(LGraph,'Deux',2);

  s:=TStringValue.Create(LGraph,'ma première chaine','Hello World');

  Log(LGraph['Un'].AsString+'/'+LGraph['Deux'].AsString+'/'+LGraph['ma première chaine'].AsString);

  Assert(IEnvironnementValue(i)=LGraph['Un'],'Anomalie');

  // une fois un nom formel créé, il est unique,
  // donc le constructeur retourne la valeur déjà présente dans le graphe
  Assert(TIntegerValue.Create(LGraph,'Un',1)=i,'Anomalie, le nom formel doit être unique');
  // idem pour ce qui soit donc on ne peut pas changer la valeur en 2
  Assert(TIntegerValue.Create(LGraph,'Un',2)=i,'Anomalie, le nom formel doit être unique');

  // Ici on a une valeur non nommée
  k:=TIntegerValue.Create(LGraph,'',1);
  l:=TIntegerValue.Create(LGraph,'',1);
  Log(IEnvironnementValue(k).FormalName); //vide
  Assert(k<>i,'Anomalie, ici la valeur primitive entière 1 peut être dupliquée dans le graphe');
  Assert(k.AsInteger=l.AsInteger,'Anomalie, même valeur même si noeud différent');
  Assert(k<>l,'Anomalie, ici la valeur primitive entière 1 peut être dupliquée dans le graphe');
finally
  LGraph.Free;
end;

```

Attention : pour des raisons de rapidité du moteur fonctionnel et du graphe nous avons masqué des traitements directs sur des pointeurs : les constructeurs sont en fait des classes (TIntegerValue, TStringValue) qui respectent le motif de conception factory et génère ici des pointeurs de valeurs de type Record. IStringValue et IIntegerValue ne sont pas de vraies interfaces mais des méthodes de Record.

Dans l'exemple précédent, on voit que si on veut retrouver une valeur primitive simplement en lui donnant un nom formel comme deuxième paramètre du constructeur. On utilise pour cela la notation :

LGraph[NomFormel]

Les constructeurs de valeurs primitives disponibles sont :

Constructeur	Type de valeur	Usage
TIntegerValue	IIntegerValue	Manipulation de valeurs entières positives et négatives
TFloatValue	IFloatValue	Manipulation de valeurs réelles
TBooleanValue	IBooleanValue	Manipulation de valeurs booléennes
TByteValue	IByteValue	Manipulation d'un octet (entier entre 0 et 255)
TListValue	IListValue	Manipulation de listes de valeur (avec un Head et un Tail) comme en Lisp
TArrayValue	IArrayValue	Gestion d'un tableau de valeur IEnvironnementValue (tableau de pointeurs)
TIntegerVectorValue	IIntegerVectorValue	Vecteur de type entier
TFloatVectorValue	IFloatVectorValue	Vecteur de type réel (tableau de valeurs, plus efficace pour les calculs que le TArrayValue)
TComplexVectorValue	IComplexVectorValue	Vecteur de type complexe
TBooleanVectorValue	IBooleanVectorValue	Vecteur de boolean
TIntegerMatrixValue	IIntegerMatrixValue	Matrice d'entier
TFloatMatrixValue	IFloatMatrixValue	Matrice de nombres réels

On peut catégoriser les valeurs atomiques dans des types sémantiques, i.e. des nœuds concepts. Pour cela on va peut écrire :

```
TConceptNodeValue.Create(LGraph, '#nombre_pair');
TConceptNodeValue.Create(LGraph, '#nombre_impair');
TConceptNodeValue.Create(LGraph, '#nombre_premier');
TIntegerValue.Create(LGraph, '', 1, ['#nombre_impair', '#nombre_premier']);
TIntegerValue.Create(LGraph, '', 2, ['#nombre_pair', '#nombre_premier']);
TIntegerValue.Create(LGraph, '', 3, ['#nombre_impair', '#nombre_premier']);
TIntegerValue.Create(LGraph, '', 4, ['#nombre_pair']);
```

Les trois premières lignes définissent des concepts (des ensembles de valeurs). Les trois dernières sont des créations d'instances qui sont catégorisées dans ces concepts.

Les types de valeurs et les types de concepts sont indépendants : on peut très bien utiliser un type de valeur string pour représenter un nombre et le catégoriser dans un des concepts précédent.

```
TStringValue.Create(LGraph, '', '5', ['#nombre_impair', '#nombre_premier']);
```

On peut alors demander toutes les instances d'un concept donné via la propriété *instances* qui retourne ces dernières sous la forme d'une liste :

```
Lgraph['#nombre_pair'].instances
```

2.4. Manipulation des identifiants uniques

Chaque nœud d'un graphe dispose d'un identifiant unique de type TUniqueId. Cet identifiant est un couple composé d'un orderId codé sur un byte suivi d'un entier Id de 32 ou 64 bit suivant le choix que l'on fait à la compilation (note : la taille de Id n'est pas lié à la compilation en mode 32 ou 64 bit de l'application, il s'agit d'un choix de taille d'identifiant afin de limiter l'impact sur la mémoire).

La structure Pascal Objet est la suivante :

```
{$IFDEF IDNUMBER64} // structure de 128 bit. Ne pas oublie d'initialiser avec nil_
TIdNumber_=Int64;//UInt64;
{$ELSE} // structure de 64 bit. Ne pas oublie d'initialiser avec nil_
TIdNumber_=Longint;//LongWord;
{$ENDIF}
```

```
TIdNumber=packed record
  OrderId:Byte; // contient l'arité ou le data power suivant qu'on est une relation ou un noeud
  Id:TIdNumber_;
  constructor Create(const b: TArray<Byte>);
  class operator Equal(a, b: TIdNumber): Boolean;
  class operator NotEqual(a, b: TIdNumber): Boolean;
  class operator Negative(a: TIdNumber) : TIdNumber;
  class operator Subtract(a: TIdNumber; b: TIdNumber) : TIdNumber;
  class operator GreaterThan(a: TIdNumber; b: TIdNumber) : Boolean;
  class operator LessThan(a: TIdNumber; b: TIdNumber) : Boolean;
  class operator GreaterThanOrEqual(a: TIdNumber; b: TIdNumber) : Boolean;
  class operator LessThanOrEqual(a: TIdNumber; b: TIdNumber): Boolean;
  //class operator Modulus(a: TIdNumber; b: TIdNumber): TIdNumber;
  function ToString:String;
  function IntegerHash:Integer;
end;
```

Un Id unique étant juste un record peut être créé à la volée en lui passant des paramètres :

```
const
  ui:TIdNumber=(OrderId:3;Id:197) ;
```

ou en utilisant la propriété UniqueId sur un objet de type IEnvironnementValue

```
ev:=TStringValue.Creat(LGraph, '', 'Hello' ) ;
```

```
Assert(ev=LGraph.FindNode(ev.UniqueId),
```

```
'Un objet doit pouvoir être retrouvé de manière univoque par son UniqueId');
```

On peut aussi manipuler cette structure en Python, par exemple avec le code ci-après.

```

>>> import BaboukWebPy as bw
>>> p=bw.MemoryRichGraph(None)
>>> p.Initialize()
>>> s=bw.StringValue(p,"","Hello")
>>> ui=s.UniqueId
>>> s_=p.FindNode(ui)
>>> s_==s
True

```

2.5. Création de relations dans le graphe

Les graphes riches sont composés de relations N-Aire. Ces relations n-aires disposent d'informations de typage et de quantifications optionnelles permettant de manipuler des structures sémantiques.

2.5.1. Manipulation de tuples non typés

Un tuple est une liste de valeurs. On la représente généralement mathématiquement sous la forme (x_1, \dots, x_N) . N est ici l'arité du tuple.

Pour définir un tuple individuel, on utilise un constructeur de tuple auquel on va passer un tableau de sommets. Par exemple :

```

t:=TTupleNodeValue.Create(LGraph, '', [
    TStringValue.Create_(LGraph, '', 'a'),
    TFloatValue.Create_(LGraph, '', '1.2')]);

```

créé ici une paire d'un string et d'un nombre réel qui correspond à la paire (2-tuple) : ('a',1.2)

Le tuple peut aussi disposer d'un nom formel comme les sommets atomiques. On peut écrire

```

t:=TTupleNodeValue.Create(LGraph, 'MonTupel', [...

```

qui permet ensuite de retrouver le tuple comme un IEnvironnementValue dans le graphe. Pour pouvoir le manipuler comme une relation, il faut alors le projeter en IRelationNodeValue

```

IRelationNodeValue(LGraph['MonTupel'])

```

On peut obtenir l'arité du tuple via :

```

t.Arity

```

et pour accéder au ième élément du tuple on va écrire

```

ev:=t.Items[i]

```

qui est un IEnvironnementValue, donc un type de sommet. Ce type de sommet pourra ensuite être manipulé sous sa forme générique ou en le projetant sur un type de valeur IIntegerValue.

2.5.2. Manipulation de relations et de tuples typés

Une relation peut être vue comme un ensemble de tuples qui ont alors pour type la relation.

Pour créer une relation, nous pouvons appeler un constructeur de relation qui dans sa forme la plus simple est similaire au constructeur de tuple. Par exemple pour définir une relation binaire, on peut écrire :

```

TRelationNodeValue.Create(LGraph, 'MaRelation', [nil, nil]);

```

ou en indiquant directement la cardinalité comme paramètre :

```

TRelationNodeValue.Create(LGraph, 'MaRelation', 2);

```

On peut alors ajouter un tuple à cette relation en le typant. On écrit alors :

```

TTupleNodeValue.Create(LGraph, '', TStringValue.Create_(LGraph, '', 'a'),
    TFloatValue.Create_(LGraph, '', '1.2'), ['MaRelation']);

```

Les constructeurs de relations permettent cependant aussi d'ajouter des informations :

- Une information de type de relation (optionnel)
- Une information de quantificateur (optionnelle) : permet de réduire l'arité de la relation en liant certains de ses sommets (voir plus loin pour la manipulation des quantificateurs).

- Une information de mappage d'index (optionnelle) : permet des permutations ou des projections d'index au sein des tuples (voir plus loin pour la manipulation des mappages d'index).
- Une information de supertype ou de sous-type (optionnelle)
- Une information de méta-données non sémantiques : ces informations permettent en particulier de donner un nom de rôle aux différents index de relation. (voir plus loin pour la manipulation des méta-données de relations)

2.5.3. Manipulation de relations quantifiées

La notion de quantificateur est celle qu'on a en logique traditionnelle. Les relations quantifiées permettent :

- d'introduire des définitions quantifiées de relations déduites d'une relation existante, d'informations de domaines et de contraintes sur ces domaines
- d'ajouter des contraintes/propriétés sur les items reliés par la relation. On ne parlera alors pas de définition quantifiée de relation mais plutôt de contrainte de relation.

Pour construire une relation quantifiée, il faut lui passer en paramètre supplémentaire une liste de quantificateurs représentés par une valeur de type Iquantifiers et un constructeur TQuantifiers. Cette liste est composée de quantificateurs élémentaires (de type TQuantifierData) associés à chaque item de la relation, mais avec des informations supplémentaires d'ordre du quantificateur lui-même car l'ordre des quantificateurs ne respecte pas forcément celui des variables.

On a plusieurs catégories de quantificateurs élémentaires :

Nom	Symbol	Constructeur	Exemples
Quantificateur universel FOL	\forall	forall(order)	forall(0)
Quantificateur universel DL	\forall	every(order)	every(0)
Quantificateur de définition	$\dot{\forall}$	definition(order) ou any(order)	definition(0) any(0)
Quantificateur existentiel	\exists ou \exists_n $\exists_{n..m}$	exists(order) ou exists(order,quantity) exists(order,minQuantity,maxQuantity)	exists(0) equivalent à exists(0,1) exists(0,3,5) On peut utiliser Star pour représenter l'infini, ie l'absence de contrainte haute sur la définition. Star(*)
Quantificateur d'instance		instance(order)	instance(0)

On note que contrairement à d'autres types de logique, on dispose ici de trois types de quantificateurs universels. La différence entre eux est la suivante :

- Quantificateur universel FOL : $\forall x \in E, R(x)$ signifie que l'on doit prendre toutes les valeurs de E et qu'elles vérifient R. C'est le quantificateur universel de la logique du premier ordre.
- Quantificateur universel DL: $\forall x \in E, R(x)$ signifie que l'on doit prendre toutes les valeurs possibles de x vérifiant R et qu'elles sont alors dans E. C'est le quantificateur universel utilisé dans les logiques de descriptions.
- Quantificateur de définition : $\dot{\forall} x \in E, R(x)$ signifie que x est un paramètre libre qui permet de définir la relation R et que le domaine des valeurs possible pour x est l'ensemble E

On les assemble dans des expressions de quantificateurs grâce au constructeur de q-tuple (tuple de quantificateurs élémentaires) :

$\exists x, \forall y$ va se coder en terme de liste de quantificateurs sous la forme

Q:=TQuantifiers.Create(LGraph,[exists(0),every(1)]) ;

que l'on peut ensuite l'associer dans une contrainte de relation : $\exists x, \forall y R(x, y)$

TRelationNodeValue.Create(LGraph, ' ', [LGraph.Top,LGraph.Top], nil, Q, [LGrap['R']]) ;

Note : ci-dessus on a mis un domaine et un co-domaine qui correspondent à l'ensemble de toutes les valeurs possibles et notés TOP (#thing est son nom équivalent) et qui n'apparaissent pas dans l'expression mathématique

et

$$\forall y, \exists x R(x, y)$$

va se coder

```
TQuantifiers.Create(LGraph, [exists(1), every(0)]) ;
```

On peut aussi indiquer que deux variables doivent être identiques dans l'expression via une contrainte de quantificateur :

$$\exists x R(x, x)$$

va se coder en terme de liste de quantificateur sous la forme

```
q:=TQuantifiers.Create(LGraph, [exists(0), exists(0)]) ;
```

que l'on peut ensuite injecter dans la contrainte de relation :

```
TRelationNodeValue.Create(LGraph, '', [LGraph.Top, LGraph.Top], nil, q, [LGrap['R']]) ;
```

avoir le même ordre de quantificateur va générer une erreur si :

- deux quantificateurs individuels différents ont le même ordre
- les domaines associés à ces quantificateurs individuels sont différents

ainsi

```
TQuantifiers.Create(LGraph, [exists(0), forall(0)]) ;
```

génère une erreur du fait de l'existence de deux quantificateurs différents.

Sinon dans l'expression suivante :

```
q:=TQuantifiers.Create(LGraph, [exists(0), exists(0)]) ;
```

```
TRelationNodeValue.Create(LGraph, '', ['E', 'F'], nil, q, [LGrap['R']]) ;
```

c'est la relation qui va générer une erreur si les concepts E et F sont différents.

2.5.4. Manipulation des informations polyadiques

Introduction aux relations polyadiques :

Les graphes riches introduisent des informations de relations particulières, que nous désignerons sous le nom d'informations polyadiques.

En présence d'informations polyadiques on considère en fait que la relation s'applique sur des tuples via une structure interne de cette dernière. Nous clarifions cette notion par un exemple :

Par exemple soit la relation polyadique suivante :

$$R(X, Y, Z) ![[1,0],[2,4],[1]]$$

Cette notation signifie que :

1. X, Y et Z sont des ensembles de tuples
2. le premier élément de la relation R est une paire (un 2-tuple) extrait de X et qu'on l'on permute les éléments 0 et 1, que le deuxième élément est un 2-tuples extrait de Y dont on prend l'élément d'index 2 et 4, et finalement que le dernier est un singleton extrait de Z dont on prend l'élément d'index 1.
3. chacun de ces éléments doit être en correspondance avec la structure interne de la relation polyadique qui est en fait une série de tuples. Donc avec l'instanciation 'interne' de la relation sous forme de tuples (a,b,c,d,e...) on a x=(b,a), y=(c,e), z=(b) qui sont des éléments valides possibles respectivement de X, Y et Z.

Manipulation des relations polyadiques par du code :

On dispose d'un constructeur pour les informations polyadiques : TPolyadicMapValue :

```
TPolyadicMapValue.Create(LGraph, '', Map)
```

ou Perm est un tableau d'entier, par exemple [0,2,1]

ainsi que Switch par exemple [[1,0], nil, nil]

Pour déclarer une relation polyadique, il suffit de passer un PolyadicMapValue non null à une relation et de lui définir un type associé.

```
TRelationNodeValue.Create(LGraph,
    'maRelationPolyadique',
    [nil,nil,nil],
    TPolyadicMapValue.Create(LGraph,'',[1,0],nil,nil),
    nil,
    [LGraph['R']])
```

note nil correspond à un tableau [0,1,2,3,...]

Quantification des relations polyadiques

On peut ajouter aussi des quantificateurs sur les relations polyadiques. Dans ce contexte, il faut interpréter la variable quantifiée comme un tuple, i.e. comme un objet ayant une liste de propriétés.

```
TRelationNodeValue.Create(LGraph,
    'maRelationPolyadiqueQuantifiee',
    [01,02,03],
    TPolyadicMapValue.Create(LGraph,'',[0,2,1],[1,0],nil,nil),
    TQuantifiers.Create(LGraph,[forall(0),exists(1),exists(2)]),
    [LGraph['R']])
```

qui correspond dans une notation d'ordre 1 étendu par les informations de mappage à :

$$\forall O_1 \in R_1, \exists O_2 \in R_2, \exists O_3 \in R_3, R[[1,0], nil, nil](O_1, O_2, O_3)$$

Note, pour changer l'ordre des quantificateurs on change leur index sans changer leur position dans le tableau. On a alors :

```
TRelationNodeValue.Create(LGraph,
    'maRelationPolyadiqueQuantifiee',
    [01,02,03],
    TPolyadicMapValue.Create(LGraph,'',[0,2,1],[1,0],nil,nil),
    TQuantifiers.Create(LGraph,[forall(1),exists(0),exists(2)]),
    [LGraph['R']])
```

qui correspond dans une notation d'ordre 1 étendu par les informations de mappage à :

$$\exists O_2 \in R_2, \forall O_1 \in R_1, \exists O_3 \in R_3, R[[1,0], nil, nil](O_1, O_2, O_3)$$

2.6. Usages de la notion de définition

La notion des définitions est obtenue par l'usage des quantificateurs de définition dans les relations. Elle permet de définir en particulier des motifs complexes basés sur des structures déjà existantes

2.7. Manipulation de relations complexes

Dans ce paragraphe nous montrons comment définir des relations complexes

2.7.1. Conjonction de définitions n-aire

Soit la définition de la relation à trois variables

$$V(x, y) \wedge W(x, z)$$

Nous pouvons facilement la définir, en utilisant une information polyadique de mappage conjoint à une définition de sous-type.

```
TRelationNodeValue.Create(LGraph,' ',
    [nil,nil,nil],
    nil,
    nil,
    [LGraph.FirstOrderRelation], // Kind
    ssDefiningsupertype,[V,W], // Supertypes/SubTypes
    TPolyadicMapValue.Create(LGraph,'',[0,1,-1],nil),
    TPolyadicMapValue.Create(LGraph,'',[0,-1,1],nil))
```

2.8. Stratégie de parcours de graphe

Le parcours du graphe peut se faire par une stratégie de visiteur. Nous implémentons différents mode de parcours des visiteurs sur le graphe :

- parcours en largeur d'abord
- parcours en profondeur d'abord

On peut alors spécialiser les traitements faits par ces visiteurs pour disposer des algorithmes génériques de recherche opérationnelle par exemple.

2.9. Utilisation au sein d'un conteneur

2.9.1. Manipulation des graph bundle

Un conteneur dispose de plusieurs couches de graphes en plus de son graphe par défaut enregistré dans un objet appelé un GraphBundle.

On peut alors récupérer le graphe par défaut utilisé par tous les conteneurs pour archiver leurs données ou accéder à un graphe nommé partagé. Les graphes nommés permettent de faire des calculs temporaires ou de récupérer des données en provenance d'un serveur distant sans les mélanger au graphe courant.

2.9.2. Graphes persistants et graphes mémoires

Nous avons vu que nous disposons de différents types de graphes :

- un type où les données sont en mémoire : un MemoryRichGraph
- un type où les données sont persistées sur le disque : un PersistentRichGraph
- des types de graphe spécialisés par un utilisateur MonTypeGraph (pré-enregistré par un développeur dans le noyau et appelé par -g MonType au démarrage de l'application – voir ci-dessous)

On démarre automatiquement sur le mode mémoire quand on lance l'exécutable de l'application. Pour démarrer sur une base de graphe il faut :

Pour démarrer sur le graphe par défaut mis à côté de l'exécutable dans le sous dossier Graph

```
BaboukWeb.exe -g Persistent
```

sinon pour donner le chemin du graphe persistant à utiliser on fera

```
BaboukWeb.exe -g Persistent -sg 'c:\...\MonDossierDeGraph
```

Si on part d'un conteneur existant, il dispose d'un bundle de graphes sous-jacent auquel on peut accéder. On peut aussi choisir de créer une couche de graphe locale au traitement que nous voulons réaliser.

Dans le premier cas, si on veut utiliser la liasse de graphes on écrira pour obtenir le graphe par défaut (dont le type de persistance est fixé au démarrage de l'application via un paramètre de ligne de commande supplémentaire -g 'chemin de dossier de graphe') de la liasse :

```
Graph:=ObjectsContainer.GraphBundle.BaseGraphLayer;
```

et si on veut un graphe nommé dans le GraphBundle il faut d'abord le créer via

```
Graph:=ObjectsContainer.GraphBundle.CreateNamedGraphLayer[ 'MonGraph' ];
```

ensuite il est possible de retrouver le graphe via l'appel de

```
ObjectsContainer.GraphBundle.FindNamedGraphLayer[ 'MonGraph' ]
```

et finalement quand on n'a plus besoin de ce graphe nommé on appelle

```
ObjectsContainer.GraphBundle.DeleteNamedGraphLayer[ 'MonGraph' ]
```

Les graphes nommés, au contraire du graphe par défaut, sont des types mémoire.

Si on n'a pas besoin de partager le graphe, on peut directement le créer. Dans le cas d'un graphe mémoire on écrira :

```
Graph:=TMemoryRichGraph.Create; // ce type de graphe est toujours mémoire  
Graph.Initialize ;
```

La méthode d'initialisation permet de charger des relations par défaut. Elle est facultative si on n'a pas à définir des relations de typage sur le graphe (sous-type, super-type, appartient à, égal,...).

Si on veut utiliser la couche de graphe persistante on va juste ajouter comme paramètre du constructeur le chemin vers le dossier contenant la base de données.

```
Graph:=TPersistentRichGraph.Create('c:\...\maBase');  
Graph.Initialize ;
```



LETTRE D'ENGAGEMENT DE NON-PLAGIAT

Je, soussigné(e) Jérémy Bénard, en ma qualité de doctorant(e) de l'Université de La Réunion, déclare être conscient(e) que le plagiat est un acte délictueux passible de sanctions disciplinaires. Aussi, dans le respect de la propriété intellectuelle et du droit d'auteur, je m'engage à systématiquement citer mes sources, quelle qu'en soit la forme (textes, images, audiovisuel, internet), dans le cadre de la rédaction de ma thèse et de toute autre production scientifique, sachant que l'établissement est susceptible de soumettre le texte de ma thèse à un logiciel anti-plagiat.

Fait à Saint Denis, le (date) 15/05/2017

Signature :

Extrait du Règlement intérieur de l'Université de La Réunion
(validé par le Conseil d'Administration en date du 11 décembre 2014)

Article 9. Protection de la propriété intellectuelle – Faux et usage de faux, contrefaçon, plagiat

L'utilisation des ressources informatiques de l'Université implique le respect de ses droits de propriété intellectuelle ainsi que ceux de ses partenaires et plus généralement, de tous tiers titulaires de tels droits.

En conséquence, chaque utilisateur doit :

- utiliser les logiciels dans les conditions de licences souscrites ;
- ne pas reproduire, copier, diffuser, modifier ou utiliser des logiciels, bases de données, pages Web, textes, images, photographies ou autres créations protégées par le droit d'auteur ou un droit privatif, sans avoir obtenu préalablement l'autorisation des titulaires de ces droits.

La contrefaçon et le faux

Conformément aux dispositions du code de la propriété intellectuelle, toute représentation ou reproduction intégrale ou partielle d'une œuvre de l'esprit faite sans le consentement de son auteur est illicite et constitue un délit pénal.

L'article 444-1 du code pénal dispose : « Constitue un faux toute altération frauduleuse de la vérité, de nature à causer un préjudice et accomplie par quelque moyen que ce soit, dans un écrit ou tout autre support d'expression de la pensée qui a pour objet ou qui peut avoir pour effet d'établir la preuve d'un droit ou d'un fait ayant des conséquences juridiques ».

L'article L335_3 du code de la propriété intellectuelle précise que : « Est également un délit de contrefaçon toute reproduction, représentation ou diffusion, par quelque moyen que ce soit, d'une œuvre de l'esprit en violation des droits de l'auteur, tels qu'ils sont définis et réglementés par la loi. Est également un délit de contrefaçon la violation de l'un des droits de l'auteur d'un logiciel (...) ».

Le plagiat est constitué par la copie, totale ou partielle d'un travail réalisé par autrui, lorsque la source empruntée n'est pas citée, quel que soit le moyen utilisé. Le plagiat constitue une violation du droit d'auteur (au sens des articles L 335-2 et L 335-3 du code de la propriété intellectuelle). Il peut être assimilé à un délit de contrefaçon. C'est aussi une faute disciplinaire, susceptible d'entraîner une sanction.

Les sources et les références utilisées dans le cadre des travaux (préparations, devoirs, mémoires, thèses, rapports de stage...) doivent être clairement citées. Des citations intégrales peuvent figurer dans les documents rendus, si elles sont assorties de leur référence (nom d'auteur, publication, date, éditeur...) et identifiées comme telles par des guillemets ou des italiques.

Les délits de contrefaçon, de plagiat et d'usage de faux peuvent donner lieu à une sanction disciplinaire indépendante de la mise en œuvre de poursuites pénales.