ScaleSem : model checking et web sémantique

Mahdi Gueffaz

To cite this version:

Mahdi Gueffaz. ScaleSem : model checking et web sémantique. Autre [cs.OH]. Université de Bourgogne, 2012. Français. NNT : 2012DIJOS080 . tel-00801730v2

HAL Id: tel-00801730
https://theses.hal.science/tel-00801730v2

Submitted on 3 Oct 2013

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.

Mémoire de Thèse

Présenté pour obtenir

Le grade de Docteur ès Sciences

Mention Informatique

Par

Mahdi Gueffaz

ScaleSem : Model Checking et Web Sémantique

Le 11 Décembre 2012

Composition du jury :

M. Patrice Moreaux, Professeur, Université de Savoie, **Rapporteur**

M. Hacène Fouchal, Professeur, Université de Reims, **Rapporteur**

M. Juan Pablo Gruer, Professeur, Université de Technologie de Belfort-Montbéliard, **Président**

M. Guillaume Doyen, Maître de conférences, Université de Technologie de Troyes, **Examinateur**

M. Christophe Nicolle, Professeur, Université de Bourgogne, **Directeur de thèse**

M. Sylvain Rampacek, Maître de conférences, Université de Bourgogne, **Co-Encadrant de thèse**

Doctorat préparé au sein de l'équipe Systèmes d'Information et Systèmes d'Images, dans l'équipe projet CHECKSEM du laboratoire LE2I de l'Université de Bourgogne.

À mes parents, ma sœur et mes frères

Remerciements

Je tiens d'abord à remercier mon directeur de thèse, M. Christophe Nicolle, Professeur des Universités de l'Université de Bourgogne et directeur adjoint du laboratoire Électronique Informatique et Image qui m'a fait l'honneur de m'accueillir au sein de son équipe. Il m'a initié à la recherche et a dirigé cette thèse en me faisant profiter de sa grande expérience et de son talent dans la recherche scientifique. Je lui adresse l'expression de ma profonde gratitude et de ma reconnaissance.

Je remercie très chaleureusement mon co-encadrant de thèse, M. Sylvain Rampacek, Maître de conférences au laboratoire Électronique Informatique et Image de l'Université de Bourgogne, pour tous ses conseils, ses recommandations, son soutien, ses encouragements et de m'avoir supporté tout au long de cette thèse. J'ai beaucoup apprécié et pris un grand plaisir à travailler avec lui.

Je remercie M. Patrice Moreaux, Professeur des Universités de l'Université de Savoie, et M. Hacène Fouchal, Professeur des Universités de l'Université de Reims, pour l'intérêt qu'ils ont pris et d'avoir accepté de rapporter mon travail. Je les remercie pour les différentes remarques et corrections qu'ils ont suggérées et qui ont permis d'augmenter la qualité de ce mémoire de thèse.

Je remercie également M. Juan Pablo Gruer, Professeur des Universités de l'Université de Technologie de Belfort-Montbéliard, et M. Guillaume Doyen, Maître de conférences de l'Université de Technologie de Troyes, pour avoir accepté d'examiner mon travail et de faire partie de mon jury.

Je tiens à remercier tous les membres de l'équipe CHECKSEM, pour leur aide et leur bonne humeur. J'ai trouvé une ambiance conviviale qui me fut précieuse.

Je tiens à remercier tous ceux qui m'ont aidé de près ou de loin afin que je puisse achever ce travail.

Je souhaite remercier spécialement Diana pour son soutien et son aide.

Merci à tous mes proches, pour leur soutien, leurs encouragements et bien plus…

Résumé

Le développement croissant des réseaux et en particulier l'Internet a considérablement développé l'écart entre les systèmes d'information hétérogènes. En faisant une analyse sur les études de l'interopérabilité des systèmes d'information hétérogènes, nous découvrons que tous les travaux dans ce domaine tendent à la résolution des problèmes de l'hétérogénéité sémantique. Le W3C (World Wide Web Consortium) propose des normes pour représenter la sémantique par l'ontologie. L'ontologie est en train de devenir un support incontournable pour l'interopérabilité des systèmes d'information et en particulier dans la sémantique. La structure de l'ontologie est une combinaison de concepts, propriétés et relations. Cette combinaison est aussi appelée un graphe sémantique. Plusieurs langages ont été développés dans le cadre du Web sémantique et la plupart de ces langages utilisent la syntaxe XML (eXtensible Meta Language). Les langages OWL (Ontology Web Language) et RDF (Resource Description Framework) sont les langages les plus importants du web sémantique, ils sont basés sur XML.

Le RDF est la première norme du W3C pour l'enrichissement des ressources sur le Web avec des descriptions détaillées et il augmente la facilité de traitement automatique des ressources Web. Les descriptions peuvent être des caractéristiques des ressources, telles que l'auteur ou le contenu d'un site web. Ces descriptions sont des métadonnées. Enrichir le Web avec des métadonnées permet le développement de ce qu'on appelle le Web Sémantique. Le RDF est aussi utilisé pour représenter les graphes sémantiques correspondant à une modélisation des connaissances spécifiques. Les fichiers RDF sont généralement stockés dans une base de données relationnelle et manipulés en utilisant le langage SQL ou les langages dérivés comme SPARQL. Malheureusement, cette solution, bien adaptée pour les petits graphes RDF n'est pas bien adaptée pour les grands graphes RDF. Ces graphes évoluent rapidement et leur adaptation au changement peut faire apparaître des incohérences. Conduire l'application des changements tout en maintenant la cohérence des graphes sémantiques est une tâche cruciale et coûteuse en termes de temps et de complexité. Un processus automatisé est donc essentiel. Pour ces graphes RDF de grande taille, nous suggérons une nouvelle façon en utilisant la vérification formelle « Le Model checking ».

Le Model checking est une technique de vérification qui explore tous les états possibles du système. De cette manière, on peut montrer qu'un modèle d'un système donné satisfait une propriété donnée. Cette thèse apporte une nouvelle méthode de vérification et d'interrogation de graphes sémantiques. Nous proposons une approche nommé ScaleSem qui consiste à transformer les graphes sémantiques en graphes compréhensibles par le model checker (l'outil de vérification de la méthode Model checking). Il est nécessaire d'avoir des outils logiciels permettant de réaliser la traduction d'un graphe décrit dans un formalisme vers le même graphe (ou une adaptation) décrit dans un autre formalisme.

L'idée générale de l'approche ScaleSem consiste à vérifier des propriétés de graphes sémantiques et leurs interrogations. Pour cela, nous proposons dans le cadre de cette thèse, des outils de transformation et d'interrogation de graphes sémantiques basés sur le Model checking.

Abstract

The increasing development of networks and especially the Internet has greatly expanded the gap between heterogeneous information systems. In a review of studies of interoperability of heterogeneous information systems, we find that all the work in this area tends to be in solving the problems of semantic heterogeneity. The W3C (World Wide Web Consortium) standards proposed to represent the semantic ontology. Ontology is becoming an indispensable support for interoperability of information systems, and in particular the semantics. The structure of the ontology is a combination of concepts, properties and relations. This combination is also called a semantic graph. Several languages have been developed in the context of the Semantic Web. Most of these languages use syntax XML (eXtensible Meta Language). The OWL (Ontology Web Language) and RDF (Resource Description Framework) are the most important languages of the Semantic Web, and are based on XML.

RDF is the first W3C standard for enriching resources on the Web with detailed descriptions, and increases the facility of automatic processing of Web resources. Descriptions may be characteristics of resources, such as the author or the content of a website. These descriptions are metadata. Enriching the Web with metadata allows the development of the so-called Semantic Web. RDF is used to represent semantic graphs corresponding to a specific knowledge modeling. RDF files are typically stored in a relational database and manipulated using SQL, or derived languages such as SPARQL. This solution is well suited for small RDF graphs, but is unfortunately not well suited for large RDF graphs. These graphs are rapidly evolving, and adapting them to change may reveal inconsistencies. Driving the implementation of changes while maintaining the consistency of a semantic graph is a crucial task, and costly in terms of time and complexity. An automated process is essential. For these large RDF graphs, we propose a new way using formal verification entitled "Model checking."

Model checking is a verification technique that explores all possible states of the system. In this way, we can show that a model of a given system satisfies a given property. This thesis provides a new method for checking and querying semantic graphs. We propose an approach called ScaleSem which transforms semantic graphs into graphs understood by the model checker (The verification Tool of the model checking method). It is necessary to have software tools to perform the translation of a graph described in a certain formalism into the same graph (or adaptation) described in another formalism.

The general idea of the ScaleSem approach is to check properties and to query the semantic graphs. To this end, we propose tools for processing and querying semantic graph-based Model checking in this thesis.

Table des matières

Remerciements ... 5
Résumé ... 7
Abstract ... 9
Liste des figures ... 15
Liste des scripts ... 17
Liste des Tableaux ... 19

Chapitre 1. Introduction ... 21

Chapitre 2. Le Web sémantique ... 29
1 Architecture du Web sémantique ... 33
2 Les langages du web sémantique ... 35
2.1 Le format de données RDF ... 35
2.1.1 La syntaxe RDF ... 37
2.1.2 Le vocabulaire RDF ... 42
2.2 Les ontologies OWL ... 44
2.2.1 Structure d'une ontologie ... 47
2.2.2 Exemple ... 48
2.3 Le langage de requête SPARQL ... 52
3 Évolution d'Ontologies ... 53
4 Conclusion ... 56

Chapitre 3. Les Méthodes Formelles et le Model checking ... 57
1 Les techniques de vérification formelles ... 61
1.1 Le Model checking ... 62
1.2 Techniques basées sur la simulation ... 63
1.3 Techniques basées sur le test ... 63
1.4 Techniques basées sur la preuve de théorème ... 64
2 Procédure du Model checking ... 65
2.1 Avantages et inconvénients du Model checking ... 67
2.2 Les model checkers qualitatifs ... 69
2.2.1 SPIN ... 69
2.2.2 NuSMV ... 70
2.2.3 ASTRAL ... 70
2.3 Les model checkers quantitatifs ... 70
2.3.1 DCVALID ... 71
2.3.2 KRONOS ... 71
2.3.3 UPPAAL ... 71
2.3.4 HyTech ... 72
2.3.5 CADP ... 72
2.4 Les algorithmes de résolution ... 73

2.4.1 Model checking LTL 74
2.4.2 Model checking CTL 77
3 L'explosion combinatoire 79
3.1 L'abstraction 80
3.2 L'interprétation abstraite 80
3.3 Le model checking symbolique 80
3.4 Les BDD (Binary Diagram Decision) 81
3.5 Les méthodes utilisant les contraintes 81
3.6 La compression mémoire 81
3.7 Les méthodes à la volée 82
3.8 L'ordre partiel 83
4 Travaux Connexes 83
5 Conclusion 84

Chapitre 4. La logique temporelle 85
1 La logique temporelle 87
1.1 La logique temporelle linéaire 88
1.2 La logique temporelle arborescente 89
2 Classification des propriétés temporelle 91
3 Choix d'une logique temporelle 93
4 Extension de la logique temporelle. 94
5 Logique temporelle et automates 95
5.1 Vérification de formule CTL 95
5.2 Vérification de formule LTL 96
6 Les requêtes en logique temporelle 97
6.1 Types de requête 98
6.2 Résolution des requêtes 99
6.2.1 Approche de Chan 101
6.2.2 Approche de Bruns & Godefroid 104
6.2.3 Approche de Schnoebelen 105
6.2.4 Approche de Chechik 106
7 Conclusion 108

Chapitre 5. Qualification de graphes sémantiques 111
1 La qualification 113
1.1 Les étapes de transformation 115
1.1.1 Exploration du graphe 115
1.1.2 Détermination d'une racine 116
1.1.3 Génération du modèle 117
1.1.4 RDF2NuSMV 118
1.1.5 RDF2SPIN 120
1.2 La vérification 124
1.3 Comparaison entre RDF2SPIN et RDF2NuSMV 124

2 Cas d'utilisation ... 126
3 Travaux Connexes ... 130
4 Conclusion ... 132

Chapitre 6. Interrogation de graphes sémantiques ... 133
1 Requête en logique temporelle ... 136
1.1 Syntaxe ... 137
2 Complexité de SPARQL ... 137
3 SPARQL et Logique Temporelle ... 144
3.1 Expressivité ... 144
3.2 La Simplicité ... 147
3.3 Langage de vérification ... 148
3.4 Extension des requêtes ... 150
4 Le STL-Resolver ... 151
5 SPARQL vers RLT ... 152
5.1 Selection ... 153
5.2 Projection ... 153
5.3 Jointure ... 153
5.4 Union ... 154
6 Travaux Connexes ... 160
7 Conclusion ... 162

Chapitre 7. La méthodologie CLOCk ... 165
1 L'évolution des ontologies ... 167
1.1 La méthodologie AIFB ... 168
1.2 La méthodologie IMSE ... 168
1.3 La méthodologie de Rogozan ... 169
1.4 La méthodologie de Djedidi ... 171
1.5 Les autres méthodologies ... 173
2 La détection des incohérences ... 175
3 La méthodologie CLOCk ... 178
3.1 OntoVersionGraph ... 179
3.2 Notre approche ... 180
4 Application de la méthode ... 184
5 Conclusion ... 187

Chapitre 8. Conclusions et Perspectives ... 189
1 Rappels ... 192
2 Perspectives ... 193

ANNEXE 1 ... 195
Bibliographie ... 201
Publications ... 220

Dépôt APP 222
Rapports techniques 223

Liste des figures

Figure 1. Architecture ScaleSem 25
Figure 2. Évolution du Web. 31
Figure 3. Couches du Web sémantique. 33
Figure 4. Exemple de graphe RDF. 36
Figure 5. Graphe RDF avec utilisation de préfixe. 37
Figure 6. Identification des types de ressources. 39
Figure 7. Description de conteneur. 40
Figure 8. Exemple de réification. 42
Figure 9. Hiérarchie de classes. 43
Figure 10. Hiérarchie de propriétés. 44
Figure 11. Le coût d'introduction, de détection et de réparation dans le cycle de vie logiciel. 60
Figure 12. L'approche du model checking. 66
Figure 13. Les différents algorithmes de transformation. 75
Figure 14. Les logiques temporelles LTL, CTL et CTL*. 88
Figure 15. L'opérateur suivant 88
Figure 16. L'opérateur éventuellement. 88
Figure 17. L'opérateur toujours. 89
Figure 18. L'opérateur jusqu'à. 89
Figure 19. L'opérateur EF. 89
Figure 20. L'opérateur AF. 89
Figure 21. L'opérateur EG. 90
Figure 22. L'opérateur AG. 90
Figure 23. L'opérateur AX. 90
Figure 24. L'opérateur EX. 90
Figure 25. L'opérateur A avec U. 90
Figure 26. L'opérateur E avec U. 90
Figure 27. Vérification de formule LTL. 96
Figure 28. La résolution des requêtes CTL^{V}. 103
Figure 29. La décomposition conjonctive approximative d'une proposition. 104
Figure 30. Architecture de l'outil TLQSolver 108
Figure 31. L'approche de la méthode ScaleSem. 114
Figure 32. Calcul des racines partielles d'un graphe RDF. 117
Figure 33. Ajout d'une Racine. 117
Figure 34. Temps de conversion de graphes sémantiques avec l'outil RDF2NuSMV. 120
Figure 35. L'architecture de vérification avec RDF2SPIN. 122
Figure 36. Temps de conversion de graphes sémantiques avec l'outil RDF2SPIN. 123
Figure 37. Temps de transformation avec les outils RDF2SPIN et RDF2NuSMV. 124
Figure 38. Taille des modèles de graphe sémantique. 125
Figure 39. Un exemple de graphe sémantique. 127
Figure 40. Exemple d'une structure de Kripke: le feu tricolore. 136
Figure 41. Le sous-graphe RDF représentant un musée. 145

Figure 42. Triplet RDF. 145
Figure 43. Requête SPARQL et sa représentation graphique. 147
Figure 44. Mémoire utilisé par l'outil STL Resolver. 150
Figure 45. Schéma général du fonctionnement de l'outil STL-Resolver. 151
Figure 46. Architecture de l'outil de conversion SPARQL2RLT. 157
Figure 47. Automate d'états finis reconnaissant les opérateurs booléens et les mots. 158
Figure 48. Automate à états finis de l'analyseur lexical. 158
Figure 49. Les étapes de la méthodologie AIFB. 168
Figure 50. Les étapes de la méthodologie Rogozan. 171
Figure 51. Méthodologie de gestion de changements Onto-Evo[a]l. 171
Figure 52. Le fonctionnement de OntoVersionGraph. 179
Figure 53.Le processus d'identification d'incohérences. 181
Figure 54. L'architecture de la méthodologie CLOCk. 183
Figure 55. Résultats des différents raisonneurs. 185
Figure 56. Exemple d'ontologie. 195

Liste des scripts

Script 1. Représentation RDF/XML d'un graphe RDF 38
Script 2. Représentation RDF/XML de la figure 6........ 40
Script 3. Représentation RDF/XML de la figure 7........ 41
Script 4. Représentation RDF/XML de la figure 8........ 42
Script 5. Exemple de création de classes OWL........ 49
Script 6. Exemple de création de propriétés d'objets OWL 49
Script 7. Exemple de création de propriétés de type de donnée OWL........ 50
Script 8. Exemple de peuplement d'ontologie........ 51
Script 9. Algorithme du Model checking CTL par marquage........ 79
Script 10. Algorithme générique. 105
Script 11. Algorithme générique partiel........ 106
Script 12. Algorithme de recherche en profondeur........ 115
Script 13. Algorithme de détection de racine. 116
Script 14. Modèle représentant un graphe sémantique........ 119
Script 15. Un processus PROMELA représentant un état sémantique (1ère version)........ 122
Script 16. Un processus PROMELA représentant un état sémantique (2e version) 123
Script 17. Un processus PROMELA utilisant les variables pour représenter les états sémantiques. . 123
Script 18. Le script RDF/XML associé à la Figure 39........ 127
Script 19. Un script PROMELA représentant un graphe sémantique. 129
Script 20. Un script NuSMV représentant un graphe sémantique. 130
Script 21. Algorithme de résolution de requêtes en logique temporelle. 152
Script 22. Transformation SPARQL vers requêtes avec opérateurs de la logique temporelle. 155
Script 23. Algorithme pour l'extraction et la transformation des triplets RDF. 156
Script 24. Analyse Lexicale des requêtes 159
Script 25. Grammaire des requêtes. 160
Script 26. Formules de logiques temporelles générées par l'algorithme. 186
Script 27. Résultat de NuSMV sur les formules du script 26........ 187
Script 28. Le graphe NuSMV du log d'évolution. 196
Script 29. Les patrons d'incohérences en logique temporelle........ 196
Script 30. La détection des incohérences par le model checker........ 198

Liste des Tableaux

Tableau 1. Instanciation de classes. 50
Tableau 2. Instanciation des propriétés. 51
Tableau 3. Pouvoir d'expression des logique LTL et CTL. 93
Tableau 4. Complexité des logiques temporelles. 93
Tableau 5. Les différents types de requêtés. (Gurfinkel et al., 2003) 99
Tableau 6. Tableau des triplets RDF du graphe sémantique. 127
Tableau 7. Table des ressources et des valeurs. 128
Tableau 8. Comparaison des langages de requêtes 143
Tableau 9. Comparaison des langages d'interrogation. 161
Tableau 10. Comparaison de la boite à outils CADP et l'approche ScaleSem. 162
Tableau 11. Synthèse des approches pour la maintenance de la cohérence 178
Tableau 12. Résultats retournés par CLOCk montrant la source de l'incohérence. 187

Chapitre 1

Introduction

Depuis une trentaine d'années, le domaine de l'interopérabilité a connu plusieurs évolutions successives. L'interopérabilité consiste principalement à permettre à des systèmes d'information hétérogènes de coopérer pour réaliser un but commun. Initialement, l'interopérabilité consistait à construire des architectures distribuées ou fédérées où le principal enjeu consistait à traduire des schémas de données d'un modèle source dans un modèle cible. Les verrous de l'interopérabilité étaient nombreux, on peut citer les principaux tels l'hétérogénéité syntaxique, l'hétérogénéité schématique ou le plus récent, l'hétérogénéité sémantique. Chacun de ces verrous a fait l'objet de nombreuses études proposant des architectures et des outils qui vont des traducteurs spécifiques entre couples de modèles à la conception de métamodèles, modèles canoniques ou supermodèles pour fédérer l'ensemble des ressources dans un modèle global. Chaque cycle de recherche sur ces verrous s'est clos sur l'adoption par la communauté d'un standard ou d'une norme de représentation des connaissances. On peut citer la définition de TCP/IP pour la résolution des problèmes d'hétérogénéité des protocoles d'échanges sur les réseaux, la définition de la norme XSD pour répondre aux problèmes d'hétérogénéité schématique ou la norme XML pour répondre aux problèmes d'hétérogénéité syntaxique. La seule ombre à ce cycle de solutions par adoption de standard est l'hétérogénéité sémantique. La sémantique peut être définie dans ce domaine comme la signification de la connaissance. L'hétérogénéité sémantique concerne donc une divergence de compréhension d'un élément qui peut être représenté de manière unique et commune par des systèmes d'information qui coopèrent. Cette divergence de compréhension entraîne des divergences de traitement. La coopération s'en trouve rapidement limitée.

Pour répondre au problème d'hétérogénéité sémantique, la communauté a défini de nouvelles normes. Des langages sémantiques composés d'opérateurs modélisant et manipulant des graphes sémantiques. Ces langages, tels que RDF, forment la base de ce que l'on appelle actuellement le web sémantique. Le web sémantique vise à développer un web intelligent où les informations seraient comprises par les ordinateurs. Ainsi, la notion de métamodèle des années 90 a laissé place à la notion d'ontologie. L'ontologie, initialement définie dans le domaine de la philosophie comme « l'étude de l'être en tant qu'être » (Aristote, 2012), représente, dans notre domaine de l'interopérabilité, un modèle de représentation de la signification de la connaissance d'un domaine. Pour représenter les ontologies, le W3C propose un standard, connu sous le nom OWL (Ontology Web Language). OWL est actuellement construit sur le modèle de données RDF, et il ajoute la possibilité de définir des classes dans des connecteurs plus complexes correspondant à la logique de description (intersection, union, classes disjointes, inverses ou propriétés transitives ou même les restrictions de cardinalité sur les propriétés...).

Bien que de nombreux projets de recherche proposent des outils de découverte et de conception dynamique d'ontologie, la modélisation des connaissances reste, pour les puristes, une affaire d'experts. La conception d'une ontologie comprend à la fois la construction d'un modèle de représentation des connaissances (au sens schéma) et le peuplement (l'attachement aux éléments composants le schéma d'instances issues des ressources des systèmes coopérants). Le processus de conception d'une ontologie nécessite donc l'adhésion de tous les acteurs concernés par les systèmes coopérants. Chacun présentant sa propre interprétation de la signification de la connaissance, l'expert intégrant le tout de manière homogène dans une ontologie de domaine.

Les retours d'expériences à partir de cet instant sont plus ou moins satisfaisants. Tout d'abord, ce processus n'est pas épargné par les erreurs de conceptions. Ensuite, la phase de peuplement peut générer des ontologies d'une taille très importante qu'aucun système ne peut exploiter de manière simple. Contrairement aux systèmes de gestion de base de données conçus pour maximiser l'organisation des données pour réduire les temps de réponse des requêtes, l'ontologie cherche à modéliser la connaissance exacte du domaine, sans considération pour son exploitation future. Enfin, le domaine que modélise l'ontologie est soumis comme tout système ou environnement à un cycle de vie qui peut très vite rendre obsoleète la connaissance modélisée. La conception d'outils et de méthodes pertinents pour qualifier une ontologie, pour la manipuler ou pour détecter les incohérences ou les inconsistances lors du cycle d'évolution d'une ontologie est un enjeu important dans le processus d'évolution du web sémantique.

Plusieurs travaux ont été menés pour tenter de répondre à ces verrous. Par exemple, une étude sur les langages de requêtes des graphes sémantiques a été réalisée par le W3C. Ce dernier a identifié plus de 20 langages pour manipuler les ontologies. Néanmoins, comme pour les propositions de résolution des problèmes d'hétérogénéité, le W3C a défini un standard : SPARQL. Ce langage de requête est dédié à l'interrogation des graphes sémantiques, donc des ontologies. Néanmoins, ce langage est toujours en cours d'amélioration et la communauté a souligné ses nombreuses limites. Un autre exemple concerne les propositions sur la gestion du changement dans une ontologie. Ces approches sont basées sur des méthodes composées de plusieurs étapes pour détecter des incohérences à l'aide de raisonneurs. Ces travaux prometteurs ne permettent pas toujours d'identifier précisément l'origine de l'incohérence lors des phases d'évolution de l'ontologie.

Pour répondre à notre tour à ces verrous, nous avons souhaité traiter les problèmes sous un nouvel angle encore jusqu'à présent inexploité dans le domaine du web sémantique. Nous avons souhaité dériver l'usage des algorithmes du model checking pour les appliquer à la gestion et la qualification de graphes sémantiques. Cette idée a été mise en œuvre par Christophe Nicolle et Sylvain Rampacek lors d'une collaboration avec Radu Mateescu, membre de l'équipe VASY de l'INRIA Rhônes-Alpes et coauteur du model checker CADP.

Le model checking est une technique dérivée des méthodes formelles. Les méthodes formelles offrent un grand potentiel pour une inclusion rapide de la vérification dans le processus de conception. Elles permettent de réaliser des audits techniques de manière plus efficace et de réduire le temps de vérification. Les méthodes formelles sont des techniques fortement recommandées pour le développement de logiciels. Elles ont conduit au développement de certaines techniques de vérification très prometteuses qui facilitent la détection précoce des défauts. Deux types de méthodes de vérification formelles peuvent être distinguées : les méthodes basées sur la preuve de théorème et les méthodes basées sur des modèles.

Les méthodes basées sur la preuve de théorème vérifient l'exactitude du système par des propriétés dans une théorie mathématique. Ces propriétés ont fait leur preuve, avec une grande précision, en utilisant des outils tels que les preuves de théorème (*theorem provers*) et les correcteurs de preuves (*proof checkers*). Les preuves de théorème sont aussi appelées les assistants de preuve (*proof assistant*).

Les méthodes basées sur les modèles décrivent le comportement du système d'une manière mathématique précise et sans ambiguïté. Les modèles du système sont accompagnés par des algorithmes qui explorent systématiquement tous les états du modèle représentant le système à vérifier. Ceci fournit la base pour toute une gamme de techniques de vérification allant de l'exploration exhaustive "*model-checking*" à des expériences avec un ensemble restrictif de scénarios dans le modèle "*Simulation*". La simulation permet à l'utilisateur d'étudier le comportement du système. Elle est moins adaptée pour détecter les erreurs, car il est difficile de générer tous les scénarios possibles du système et de les simuler tous. Le model checker est une technique de vérification qui explore tous les états possibles du système. Il examine tous les états du système pertinents afin de vérifier s'ils satisfont la propriété désirée. Le model checker donne un contre-exemple qui indique comment le modèle peut violer la propriété. Avec l'aide d'un simulateur, l'utilisateur peut trouver l'erreur et adapter le modèle ou la propriété afin d'empêcher la violation de cette dernière. Le dernier attrait, et non des moindres, de cette approche est que le model checker fonctionne de manière automatique.

À la fin de la collaboration entre Radu Matescu et l'équipe Checksem et au regard des résultats encourageants, l'équipe a souhaité poursuivre ces travaux sous la forme d'une thèse initiée en septembre 2009. Le projet ScaleSem a ainsi vu le jour. L'idée générale du projet ScaleSem consiste à vérifier, à l'aide de la logique temporelle, des propriétés de graphes sémantiques (Ontologie OWL ou graphe RDF). Pour utiliser le model checker, il est nécessaire de disposer de logiciels permettant de réaliser la traduction d'un graphe décrit dans un formalisme vers le même graphe (ou une adaptation) décrit dans un autre formalisme. Notre objectif est de choisir un model checker puissant et de l'adapter à une logique assez expressive pour spécifier et vérifier des propriétés sur des graphes sémantiques. Cet objectif nécessite le développement d'outils et de méthodes spécifiques. En conséquence, ce projet est articulé en sous-projets en fonction du domaine d'application ou en fonction du type d'approche utilisée (population d'ontologies, indexation sémantique, méta modélisation). La Figure 1 schématise l'architecture de notre projet.

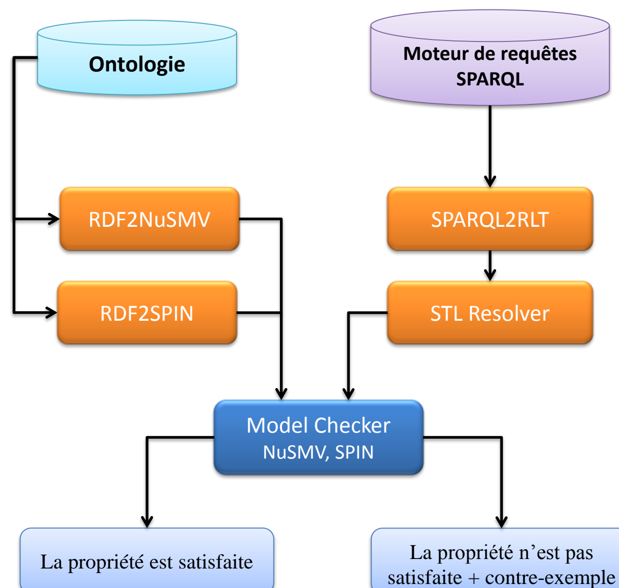


Figure 1. Architecture ScaleSem

Pour utiliser les outils du model checking, nous devons traduire les ontologies dans un formalisme spécifique aux model checkers. Suite aux travaux réalisés sur CADP, nous avons choisi de transformer les graphes sémantiques en graphes compréhensibles par deux model checkers : SPIN et NuSMV.

Pour chacun, nous avons développé une méthode spécifique de conversion et un algorithme. Les outils de transformations obtenus ont fait l'objet de dépôt de protection logicielle (4 dépôts APP) à l'Université de Bourgogne. Les résultats de cette première étape de transformation ont été publiés sous la forme de deux articles de recherche dans des journaux internationaux[1] et de 4 articles dans des conférences internationales[2]. L'objectif de la transformation est d'obtenir une représentation de l'ontologie qui puisse être manipulée en utilisant des propriétés en logique temporelle. Pour faciliter la manipulation des ontologies dans notre système, nous avons développé un outil de transformation de requêtes en SPARQL en requêtes en logique temporelle (SPARQL2LTL). Ce travail nous permettra, dans la suite de ce document, de démontrer que la logique temporelle est plus simple à manipuler que le SPARQL et la logique temporelle permet l'écriture de requêtes complexes que le SPARQL ne peut représenter. Cette partie des travaux de recherche a été publiée dans un journal international et une conférence internationale[3].

Ensuite, nous avons combiné notre approche avec un travail de recherche sur la gestion du changement au sein des ontologies. Ce travail, mené par Perrine Pittet au sein de l'équipe Checksem a permis de développer le projet CLOCk afin de prédire et détecter les incohérences dans les évolutions des ontologies. Les résultats de ce travail ont été publiés dans une conférence internationale sous la forme d'un poster[4].

Pour exposer notre travail et présenter nos propositions de solutions aux verrous énoncés dans cette partie, nous avons structuré ce document en 5 chapitres.

Le premier chapitre (chapitre 2) a pour objectif de construire une vision claire des éléments composants le web sémantique. Ce terme désigne un ensemble de technologies visant à rendre le contenu des ressources du WWW (World Wide Web) accessible et utilisable par les programmes et agents logiciels, grâce à un système de métadonnées formelles, utilisant notamment la famille de langages développés par le W3C. Le but du web sémantique est donc d'ajouter une couche d'information dont le sens devient exploitable et compréhensible par les machines. "L'enjeu étant que les logiciels acquièrent la capacité d'établir entre eux un véritable dialogue, c'est-à-dire un échange pertinent d'informations immédiatement exploitables sans intervention humaine." (Tim Berners-Lee, directeur du Web Consortium). Le W3C propose des normes pour représenter la sémantique (la signification de la connaissance d'un domaine) à l'aide de modèles communément appelés ontologie. À l'instar des bases de données, la structure et le contenu des ontologies doivent évoluer dans le temps afin de rester en adéquation avec les concepts et les connaissances du domaine modélisés. Ce chapitre définit le web sémantique et ses différents langages. Les langages

---

[1] Journal of Software (2012), International Journal of Digital Information and Wireless Communications (2011) & (2012)

[2] Webist (2012), AFIN (2011), ICDIPC (2011), DICTAP (2011), ACM SIGMIS (2011), Innovations'11 (2011)

[3] Journal of Software (JSW), Selected Best papers on IIT, 2012, International Journal of Digital Information and Wireless Communications (IJDIWC) 1, 2 366-380, 2012, International Conference on Digital Information Processing and Communications (ICDIPC 2011), Ostrava, République Tchèque, Springer-Verlag, Juillet 2011

[4] 8th International Conference on Web Information Systems and Technologies , Porto, Portugal, INSTICC, ACM SIGMIS, April 2012

RDF et OWL sont les plus importants. Ils servent à représenter des graphes qu'on nomme les « graphes sémantiques ». RDF est représenté par un ensemble de triplets, chaque triplet est constitué de sujet, prédicat, objet. Le langage RDF définit le premier niveau sémantique au-dessus du web. Le langage OWL est utilisé pour représenter des ontologies. Il est basé sur le langage RDF et il ajoute la possibilité de définir des classes avec des connecteurs plus complexes. Comme nous l'avons déjà souligné, ces ontologies évoluent très rapidement et leur évolution peut provoquer des incohérences et des inconsistances. Dans sa dernière partie, ce chapitre présentera les différents travaux liés au versioning (évolution) d'ontologie.

Le second chapitre (chapitre 3) a pour objectif de présenter le domaine du model checking. Comme nous l'avons dit, le développement de systèmes est une activité complexe, et les défauts résiduels sont fréquents dans les versions déployées. Une des approches proposées pour résoudre ce problème consiste à utiliser des outils mathématiques pour spécifier, valider et vérifier les systèmes informatiques. On désigne communément ces approches basées sur les mathématiques comme des méthodes formelles. Les méthodes formelles sont des techniques permettant de raisonner rigoureusement, à l'aide de logique mathématique, sur des programmes informatiques ou des matériels électroniques, afin de démontrer leur validité par rapport à une certaine spécification. Par « formelles », on désigne des méthodes dont le but principal est de rendre précises des idées de développement auparavant vagues ou simplement intuitives. Ce chapitre donne un bref aperçu des différentes techniques existantes des méthodes formelles et présentera en détail une de ces techniques qui est la technique du model checking. Le model checking est une méthode formelle très puissante et très utilisée ces dernières années dans la vérification des systèmes concurrents. Le model checking est un ensemble de techniques de vérification automatique de propriétés temporelles sur des systèmes réactifs. Schématiquement, un algorithme de model checking prend en entrée une abstraction du comportement du système réactif (un système de transitions) et une formule d'une certaine logique temporelle, et répond si l'abstraction satisfait ou non la formule. On dit alors que le système de transitions est un modèle de la formule, d'où le terme anglais de model checking. Le milieu industriel a largement adopté le model checking, car c'est un processus automatique qui retourne un contre-exemple quand la propriété n'est pas vérifiée. La dernière partie de ce chapitre présentera les différents model checker existant sur le marché et plus spécifiquement les model checkers NuSMV et SPIN que nous utiliserons dans la suite.

Le troisième chapitre (chapitre 4) présente la logique temporelle et les requêtes en logique temporelle. Les formules en logique temporelle sont utilisées pour représenter un ensemble de propriétés à vérifier sur un système donné. Une hypothèse sur le comportement du système est exprimée comme une formule en logique temporelle, et le model checker est utilisé pour valider l'hypothèse. Le processus est réitéré pendant que l'utilisateur prend connaissance du système. Il existe deux grandes catégories de logique temporelle, la logique temporelle linéaire et la logique temporelle arborescente. Il existe aussi plusieurs extensions de ces deux types de logique temporelle. Pour aider l'utilisateur à comprendre les comportements du système, nous utilisons les requêtes en logique temporelle combinées au model checking pour déterminer les propriétés temporelles plutôt que de simplement les contrôler. Ce chapitre introduit les différentes logiques temporelles et leur extension. La différence entre les deux principales logiques arborescentes et linéaires ainsi que les requêtes en logique temporelle sont présentées.

Le quatrième chapitre (chapitre 5) traite de la qualification de graphes sémantiques. L'utilisation croissante des graphes sémantiques et les coûts liés à leurs modifications requièrent des outils fiables pour prendre en charge leur évolution. Une erreur recherchée couramment dans les graphes sémantiques est la résolution de contradictions logiques appelées incohérences et inconsistances. Dans ce chapitre, nous proposons une nouvelle approche pour la vérification de ces différentes contradictions dans les graphes sémantiques. Nous verrons que cette approche de qualification ne se résume pas seulement à la vérification des incohérences dans les ontologies. Elle peut également servir à l'interrogation de ces modèles afin d'étendre l'expressivité des requêtes SPARQL tout en simplifiant l'écriture des requêtes. Ce chapitre définira notre stratégie de vérification des graphes sémantiques. Cette stratégie utilise des formules en logique temporelle pour décrire les propriétés à vérifier sur le graphe et un modèle du graphe sémantique. Ce modèle représente le graphe sémantique sans perte d'informations. Ce chapitre définit les algorithmes et les étapes de transformation d'un graphe sémantique en un graphe compréhensible par le model checker.

Le cinquième chapitre (chapitre 6) complète la proposition initiée dans le chapitre précédent. Il décrit les processus d'interrogation des graphes sémantiques à l'aide des requêtes en logique temporelle. Les requêtes SPARQL sont définies à partir des patrons de graphes qui sont fondamentalement des graphes RDF avec des variables. Les requêtes SPARQL restent limitées, car elles ne permettent pas d'exprimer des requêtes avec une séquence non bornée de relations (par exemple, « Existe-t-il un itinéraire d'une ville A à une ville B qui n'utilise que les trains ou les bus ? »). Notre approche se base sur la proposition de (Chan, 2000). Pour aider l'utilisateur à comprendre les différents comportements du système, Chan introduit les requêtes en logique temporelle (Le Query Checking) et utilise une technique similaire au model checking symbolique pour déterminer les propriétés temporelles plutôt que de simplement les contrôler. Dans ce chapitre, nous proposons une nouvelle approche utilisant les requêtes en logique temporelle pour l'interrogation des graphes sémantiques.

Le sixième chapitre (chapitre 7) présente l'adaptation de notre approche à une méthode de gestion du changement dans le domaine de l'ontologie (Pittet et al., 2011). Tout changement peut mener à des contradictions au sein de l'ontologie. Ces contradictions apparaissent pour plusieurs raisons telles que des erreurs de modélisation lors de la construction d'une ontologie et de la migration ou fusion d'ontologies. Il est indispensable de vérifier la consistance de l'ontologie. Dans ce chapitre, nous proposons une nouvelle méthodologie basée sur l'approche ScaleSem pour prévoir et identifier les contradictions dans l'évolution des ontologies. Cette approche est basée sur la logique temporelle et les patrons de conception d'ontologies. Nous mettons en œuvre l'approche proposée en utilisant le model checker NuSMV. Sur la base de ces modèles, nous proposons un processus automatisé pour guider et surveiller la mise en œuvre du changement tout en veillant à la cohérence de l'ontologie à évoluer.

Chapitre 2

Le Web sémantique

Résumé

L'expression Web sémantique, définie par Tim Berners-Lee (Berners-Lee et al., 2001) au sein du W3C (*World Wide Web Consortium*), fait référence à la vision du Web de demain comme un vaste espace d'échange de ressources entre les êtres humains et les machines. Le web sémantique (Laublet et al., 2002) (Laublet et al., 2004) désigne un ensemble de technologies visant à rendre le contenu des ressources du Web accessible et utilisable par les programmes et agents logiciels, grâce à un système de métadonnées formelles, utilisant notamment la famille de langages développés par le W3C : le langage RDF (Becket et McBride, 2004), les ontologies OWL, et leur véhicule XML (Bray et al., 2006).

Ce premier chapitre présente le Web sémantique. La première partie présente l'architecture du web sémantique. Cette partie met en évidence une architecture nécessaire et suffisante pour la réalisation du web sémantique avec une description des couches principales. Dans la seconde partie, je présente les différents langages du web sémantique et dans la dernière partie, je présente les différents outils existants pour la création et l'interrogation de graphes sémantiques. Ce chapitre permet, en outre, d'identifier les limites des solutions existantes par rapport à nos objectifs.

Plan

1 Architecture du Web sémantique 33
2 Les langages du web sémantique 35
2.1 Le format de données RDF 35
2.1.1 La syntaxe RDF 37
2.1.2 Le vocabulaire RDF 42
2.2 les ontologies OWL 44
2.2.1 Structure d'une ontologie 47
2.2.2 Exemple 48
2.3 le langage de requête SPARQL 52
3 Évolution d'Ontologies 53
4 Conclusion 56

Le « Web » est en perpétuelle évolution depuis sa conception à la fin des années 60 avec le réseau ArpaNet (acronyme anglais de « Advanced Research Projects Agency Network »). ArpaNet est le premier réseau à transfert de paquets développé aux États-Unis par la DARPA. L'objectif du Web est de construire un réseau de ressources matérielles et logicielles interopérables. Le principal frein de ce projet reste à ce jour l'hétérogénéité. Cette hétérogénéité entre ressources peut être décomposée en niveaux : hétérogénéité syntaxique, hétérogénéité schématique, hétérogénéité dans les protocoles de communications, dans le type de langage de développement… Au cours du temps, toutes ces formes d'hétérogénéité ont été résolues par l'adoption de normes et standards (TCP/IP, HTML, XML, SOA…). La dernière forme d'hétérogénéité identifiée à ce jour est l'hétérogénéité sémantique. Malgré l'adoption de normes et de langages communs tels que RDF ou OWL pour représenter la sémantique des ressources et des données et processus échangés sur le web, la modélisation de la signification des éléments du web reste un problème actuel. Cette signification évolue constamment, peut changer de sens selon les contextes et la combinaison de plusieurs éléments peut rendre inconsistant ou incohérent l'ensemble des connaissances associées. Pour arriver au web sémantique d'aujourd'hui, dénommé Web 3.0, de nombreuses évolutions liées davantage à l'usage plus qu'aux technologies ont été proposées (Figure 2).

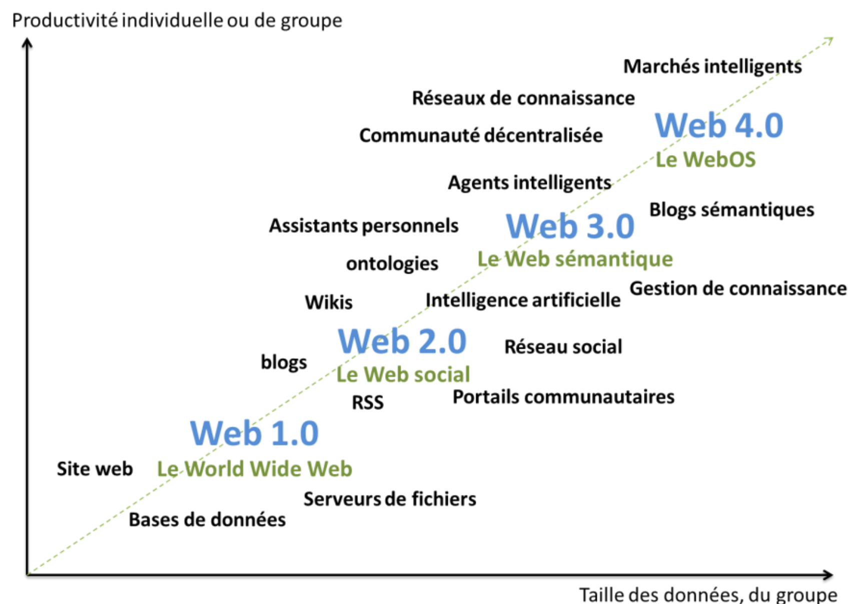


Figure 2. Évolution du Web.

Le **Web 1.0** a d'abord été un Web publicitaire, avec des *webmestres* qui rédigent et envoient des informations vers les internautes sans que ces derniers puissent réellement participer, hormis dans les forums ou encore par mails. La production et l'hébergement de contenus se font essentiellement par les entreprises (Champeau, 2007) (Gai, 2007). Les pages sont statiques, quelquefois sans réactualisation des informations communiquées (Liotard, 2008). Dans le **Web 2.0** (Guillaud, 2008), les usagers entrent dans un mode actif : au fur et à mesure de leur navigation, les utilisateurs ajoutent du contenu au travers de liens hypertextes et autres tags, annotations ou commentaires. Le contenu est généré par les utilisateurs grâce à l'émergence des blogs, des wikis, des journaux citoyens. Wikipédia (le plus grand wiki du Web), Dailymotion, Youtube, Flickr, Delicious, etc., et tous les nouveaux médias sont de véritables espaces de discussion, d'expression, d'échange

et de débat. L'utilisateur devient alors source d'information et de création : le concept d'intelligence collective émerge. La production de contenu se fait par les internautes et l'hébergement par les entreprises (Champeau, 2007) (Gai, 2007). Les données sont partagées, l'internaute est acteur et fournit ses propres contenus au travers des réseaux sociaux (comme Facebook, MySpace) et des blogs. Grâce au phénomène de syndication (flux RSS), l'internaute peut se tenir au courant en temps réel des dernières parutions sur les thèmes qui l'intéressent. Le terme **Web 3.0** est apparu pour la 1ère fois en 2006 dans un article du blog de Jeffrey Zeldman (Zeldman, 2006). Ce Web a pour ambition de regrouper tous les savoir-faire en matière de représentation de la connaissance. Dans ce Web 3.0, on retrouve l'Internet classique (surfer sur son ordinateur via une connexion filaire ou Wifi), l'Internet mobile (sur son téléphone portable, son PDA, son smartphone) et l'Internet des objets (Roynette, 2009). L'internet des objets représente une extension de l'Internet dans le monde réel grâce à un système d'étiquettes (Bastide, 2008) associant des URLs aux lieux ou aux objets lisibles par des dispositifs mobiles sans fil ou des puces RFID (Manach, 2009). Dans ce cadre, le logiciel se libère des contraintes physiques des ordinateurs personnels. De nombreuses applications sont désormais directement accessibles "en ligne". Au-delà de cet aspect, Internet se transforme peu à peu en un véritable écosystème informationnel dans lequel l'utilisateur sera immergé (Liotard, 2008).

Dans ce nouvel environnement, tous les sites sont liés d'une façon ou d'une autre. Ainsi, l'internaute est "fiché", notamment au travers de sa navigation et de ses différents profils sur les réseaux sociaux. De plus, les sites sont envahis de publicités contextuelles en rapport avec le document consulté. Avec le Web sémantique, qui se met en place grâce à l'impulsion du W3C (*World Wide Web Consortium*) (Gandon, 2008), se dessine un Web intuitif, une "constellation" d'informations, compatible avec tous les systèmes d'exploitation et tous les objets reliés. Les sites tiennent compte de nos visites précédentes, de notre navigation, de notre profil ; ils proposent des recherches associées à celles que nous faisons, des retours plus adaptés et plus intelligents aux requêtes… Avec le web sémantique, la demande appelle un résultat cohérent, méticuleusement assemblé. Le système travaille pour nous : il classe tous les commentaires et trouve, par déduction, une correspondance adéquate. Des technologies sont mises en place pour permettre de comprendre l'information et adapter les réponses fournies dans un contexte plus riche et plus finement proposé. Une meilleure connaissance de l'utilisateur permettra d'obtenir des réponses plus ajustées à son profil, voire proposera d'autres résultats potentiellement acceptables par l'usager.

Certains conceptualisent aujourd'hui le **Web 4.0** comme une dimension supplémentaire de la Toile. Le Web 4.0 prolonge naturellement le concept de fédération des sources de données du Web 3.0 en étendant le type de ressource à des objets ambiants qui seront connectés (par exemple des éléments d'un bâtiment ou d'une voiture…). Pour le Web 4.0, les futurologues prédisent l'omniprésence d'Internet (par exemple, la notion de Cloud Computing préfigure pour (De Rosnay, 2009) le Web 4.0). Les grandes entreprises productrices de logiciels et de matériels ont bien compris que les services allaient prendre une part de plus en plus importante dans le monde de l'Internet. Elles commencent donc à proposer des solutions disponibles sur le «nuage» (CBS, 2009). Le marketing dans le Web 3.0 puis dans le 4.0 pourrait tenir une place essentielle. Globalement, au regard des gains financiers très importants générés par les deux premières "dynasties" du Web, il semble évident que la manne pécuniaire affluera pour qui sera le premier à se placer sur le Web 3.0.

La combinaison des connaissances devient dans ce cas un enjeu crucial. Il est important de trouver des méthodes de combinaison, mais aussi des méthodes de qualification des connaissances modélisées et combinées. Ce travail de recherche s'inscrit dans cet objectif et propose d'adapter des techniques du model checking à la qualification de connaissance du web sémantique. Néanmoins, avant d'exposer notre approche, il est nécessaire de présenter en détail les principales notions du web sémantique comme son architecture (section 1), les langages utilisés (section 2) et surtout les méthodes proposées pour gérer le cycle de vie des connaissances (section 3).

1 ARCHITECTURE DU WEB SEMANTIQUE

La vision courante du Web sémantique proposée par (Berners-Lee et al., 2001) peut être représentée dans une architecture en plusieurs couches (Figure 3). Les couches les plus basses assurent l'interopérabilité syntaxique : la notion d'URI[5] fournit un adressage standard universel permettant d'identifier les ressources tandis qu'Unicode, qui est au même niveau que l'URI, est un encodage textuel universel pour échanger des symboles : il permet à tous les langages humains d'être utilisés sur le Web. Rappelons que l'URL (Uniform Resource Locator), comme l'URI, est une chaîne courte de caractères qui est aussi utilisée pour identifier des ressources (physiques) par leur localisation.

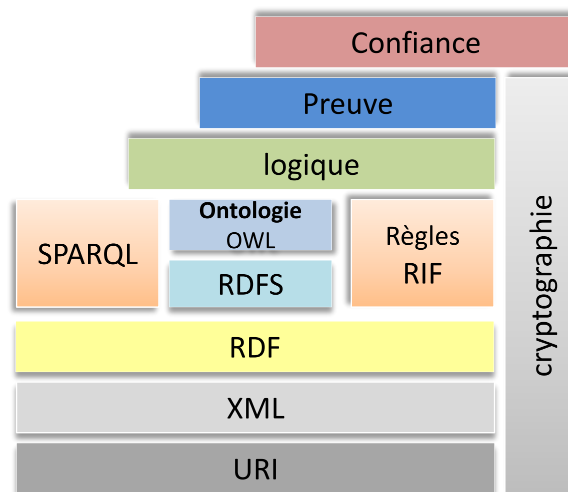


Figure 3. Couches du Web sémantique.

XML[6] fournit une syntaxe pour décrire la structure du document, créer et manipuler des instances des documents. Il utilise l'espace de nommage (*namespace*) afin d'identifier les noms des balises (*Tags*) utilisés dans le document XML. Le schéma XML permet de définir les vocabulaires pour des documents XML valides. Cependant, XML n'impose aucune contrainte sémantique à la signification de ces documents. L'interopérabilité syntaxique n'est pas suffisante pour qu'un logiciel puisse comprendre le contenu des données et les manipuler d'une manière significative.

Les couches ***RDF***[7] **et** ***RDF-Schema*** (Brickley et al., 2004) sont considérées comme les premières fondations de l'interopérabilité sémantique. Elles permettent de décrire les taxonomies des concepts et des propriétés (avec leur signature). RDF fournit un moyen d'insérer la sémantique dans un

[5] Uniform Resource Identifier
[6] eXtensible meta Language
[7] Resource Description Framework

document. L'information est conservée principalement sous forme de déclarations RDF. Le schéma RDF (RDFS) décrit les hiérarchies des concepts et leurs relations, les propriétés et les restrictions domaine/co-domaine pour les propriétés. RDF, acronyme de *Resource Description Framework*, est un modèle de données. Par abus de langage, il est commun de dire que c'est un « langage » d'assertion et d'annotation. RDF est un outil fondamental du web sémantique : il permet de définir des métadonnées afin de préciser les caractéristiques d'une information. Il établit des relations entre ressources. Il est donc particulièrement adapté aux annotations associées aux ressources du Web. RDFS (RDF Schéma) définit un vocabulaire utilisé dans les modèles de données RDF. Un document RDFS précise également les propriétés des différents objets modélisés, les domaines de valeurs possibles et décrit les relations entre ces différents objets. Les mécanismes de raisonnement mis en œuvre dans RDFS trouvent leurs fondements dans les premiers langages à base de logique pour données orientées objet, tels que F-logique (Kifer et al., 1995) et la logique de description (Baader et al., 2003). Une formalisation du modèle théorique de RDFS sémantique a été fournie dans (Marin, 2004). Les fondations et les aspects avancés de bases de données RDFS ont été étudiés dans (Gutierrez et al., 2004). Ce dernier travail couvre des problèmes tels que le raisonnement sur RDFS et présente un ensemble de règles d'inférence qui mettent en œuvre le cœur de la sémantique du modèle théorique RDF. En outre, il étudie les problèmes tels que la complexité de l'implication (si un graphe RDFS implique logiquement un autre graphique), les formes normales de RDFS, et les requêtes réalisées sur les bases de données RDFS. Similaire dans le style, un langage de requête d'enregistrement de données de style RDF, appelée RDFLog, a été proposé dans (Bry et al., 2008). Dans (Munoz et al., 2007) un système minimal déductif pour RDFS est présenté. Ce travail donne des indications précieuses sur le processus de raisonnement sur RDFS.

La couche suivante est **l'*ontologie OWL***. Elle décrit des sources d'information hétérogènes, distribuées et semi-structurées en définissant les consensus du domaine commun et partagées par plusieurs personnes et communautés (la signification de la connaissance). OWL, acronyme de *Web Ontology Language*, permet d'étendre le vocabulaire et les propriétés de RDF. Il offre une grande souplesse dans la définition des relations. Par exemple, OWL permet de préciser qu'une propriété est l'inverse de l'autre, ce qui permet d'inférer des relations non explicites. « À est le père de B» nous dit également que B est fils de A. Ceci paraît trivial, mais en informatique, il faut établir ce genre de raisonnement de base afin d'avoir des informations traitées «intelligemment». OWL permet de construire des relations de disjonctions ou de faire des unions. Les ontologies aident la machine et l'humain à communiquer avec concision en utilisant l'échange sémantique plutôt que syntaxique. Au même niveau, on trouve la couche du langage SPARQL. Ce langage est désigné par le W3C comme le standard pour l'interrogation des graphes RDF et OWL. SPARQL est l'équivalent de SQL[8] (Chebotko et al., 2006) pour le Web des données. La syntaxe SPARQL varie quelque peu de celle du SQL et il nécessaire de déclarer les espaces de nom utilisé lors de la requête. SPARQL se base directement sur les métadonnées RDF. Cela permet aux machines ou aux humains d'interroger des bases de données sur le Web, sans forcément en connaitre le schéma au préalable, ce qui permettrait un accès aux données sans intermédiaire. Le langage SPARQL a été standardisé par le DAWG (*RDF Data Access Working Group*) le 15 janvier 2008 (Herman, 2007). Ce langage a été inventé pour interroger une base de données RDF, et signifie littéralement « protocole d'interrogation en langage RDF ».

[8] Structured Query Language

La couche de règles a pour objectif de normaliser la représentation des règles RDF. Elle comporte deux langages de règles : SWRL (*Semantic Web Rule Langage*) et RIF (*Rule Interchange Format*). SWRL est une extension d'OWL. RIF ne repose pas directement sur RDF, mais sur XML et il permet de faciliter l'utilisation et l'échange de règles entre les formats déjà existants.

La couche *logique* se trouve au-dessus de la couche ontologie. Certains considèrent ces deux couches comme étant au même niveau : c'est-à-dire comme des ontologies basées sur la logique et permettant des axiomes logiques. En appliquant la logique déductive[9], on peut inférer de nouvelles connaissances à partir d'une information explicitement représentée.

Les couches *preuve* et *confiance* fournissent des éléments pour réaliser la vérification des déclarations effectuées dans le web sémantique. On s'oriente vers un environnement du web sémantique fiable et sécurisé dans lequel nous pouvons effectuer des tâches complexes en sûreté. D'autre part, la provenance des connaissances, des données, des ontologies ou des déductions est authentifiée et assurée par des signatures numériques. Dans le cas où la sécurité est importante ou le secret est nécessaire, le chiffrement est utilisé.

La couche *Cryptographie* a pour but de s'assurer et de vérifier que les déclarations issues du web sémantique proviennent d'une source sure, ce qui peut être réalisé par la signature numérique des déclarations RDF.

2 LES LANGAGES DU WEB SEMANTIQUE

Dans le contexte du Web sémantique, plusieurs langages ont été développés (Baget et al., 2005). La plupart de ces langages reposent sur XML ou utilisent XML comme syntaxe. XML est le langage de base. Il est donc naturellement utilisé pour encoder les langages du Web sémantique. Mais il a surtout la propriété d'être un métalangage. XML est limité, car il ne dispose pas d'une sémantique, ce qui nécessite le développement d'autres langages pour le web sémantique. Dans la suite de cette section, nous étudierons plus particulièrement les formats RDF, RDF Schéma, OWL et SPARQL.

2.1 LE FORMAT DE DONNEES RDF

RDF (*Resource Description Framework*) n'est pas à proprement parler un langage (Gagnon, 2007). Il s'agit plutôt d'un modèle de données pour décrire des ressources sur le web. On entend par ressource toute entité que l'on veut décrire sur le web, mais qui n'est pas nécessairement accessible sur le web. Par exemple, on pourrait fournir des informations sur l'auteur de ce document, même si la personne décrite n'est pas accessible sur le web. On trouve plutôt des ressources, comme une page personnelle, ou une photo, qui peuvent être obtenues à partir de leur URL (*Universal Resource Locator*). Ces ressources sont reliées à cette personne, mais ne sont pas cette personne. Pour désigner cette personne, on utilisera une URI (*Universal Resource Identifier*), un nom unique qui ressemble syntaxiquement à une URL, sans qu'il soit nécessaire que celle-ci soit accessible sur le web. D'une certaine manière l'ensemble des URL est inclus dans l'ensemble des URI. La raison d'être de RDF est de permettre que les informations sur les ressources soient manipulées par des applications,

[9] Une logique déductive est une méthode scientifique qui considère que la conclusion est implicite dans les prémisses.

plutôt que d'être simplement affichées aux utilisateurs du web. Pour cette raison une syntaxe XML a été proposée pour véhiculer des informations modélisées en RDF. Parmi les caractéristiques importantes de RDF, on retrouve la flexibilité et l'extensibilité.

Par exemple, supposons que nous voulions décrire une personne qui s'appelle Mahdi Gueffaz, qui est doctorant au département d'informatique de l'Université de Bourgogne et dont la page personnelle se trouve à l'URL suivante : http://checksem.u-bourgogne.fr/mgueffaz

Par la description ci-dessus, il faut reconnaître les quatre entités référencées (la personne en question, son nom, l'endroit où elle travaille et sa page personnelle). Pour chacune, sauf le nom, il faut donc une URI pour la représenter. Le nom est un cas spécial, puisqu'il s'agit en fait d'une chaîne de caractères. Nous pourrons dans ce cas utiliser la chaîne directement, sans passer par l'intermédiaire d'une URI pour la désigner. Les quatre entités de notre description sont désignées de la façon suivante :

La personne décrite	https://checksem.u-bourgogne.fr/doctorant#MahdiGueffaz
Le nom de la personne	"Mahdi Gueffaz"
Le lieu de travail	https://checksem.u-bourgogne.fr/lieu#Dijon
La page personnelle	http://checksem.u-bourgogne.fr/mgueffaz

La personne et son lieu de travail sont désignés par une URI. Cela ne signifie pas nécessairement qu'il s'agisse d'une URL correspondant à un document auquel on peut accéder sur le web. Il s'agit tout simplement d'une convention utilisée pour standardiser les URI. L'intérêt de cette convention est qu'elle laisse la possibilité de traiter cette URI comme une URL. Elle permet d'associer sur le web un document correspondant à cette URL. Il n'est pas nécessaire qu'il y ait un document sur le web pour chaque URI.

Après avoir représenté les entités, il faut représenter leurs relations. Les relations sont représentées par des entités particulières appelées *propriétés*. Leur rôle est d'établir un lien entre deux entités. Dans l'exemple, il faut utiliser trois propriétés pour relier les quatre entités (la personne avec son nom, son lieu de travail et sa page personnelle). Les propriétés sont aussi représentées par des URI (Figure 4).

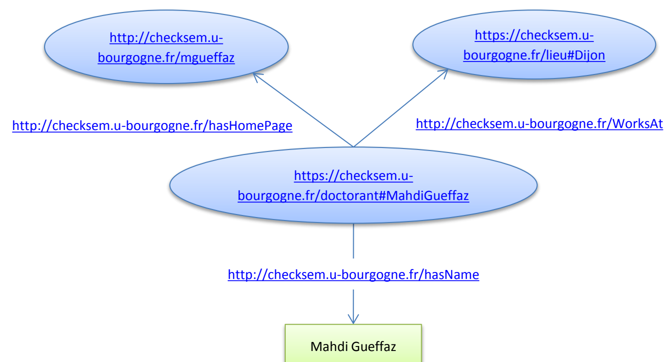


Figure 4. Exemple de graphe RDF.

Les URI de la Figure 4 ont le même préfixe. Il est possible d'alléger la représentation en utilisant un alias pour réduire la taille des URI (Figure 5). Un modèle RDF est un graphe, avec deux types de nœuds et une entité pour les relier. Certains nœuds, représentés par une ellipse, désignent une entité référée par une URI. D'autres nœuds, représentés par un rectangle, représentent un littéral, c'est-à-dire une entité exprimée directement, une chaîne de caractères par exemple. Un littéral peut aussi être d'un autre type, comme un entier, une valeur booléenne, une date, etc.

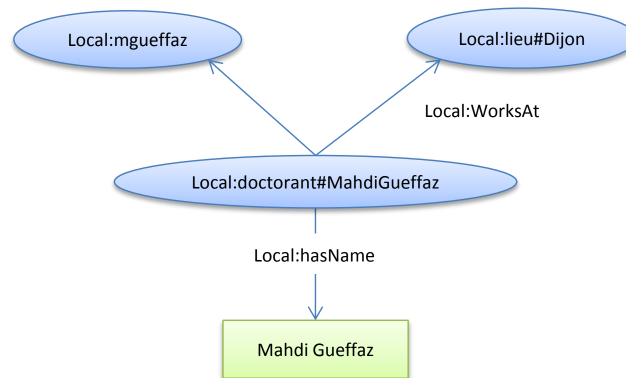


Figure 5. Graphe RDF avec utilisation de préfixe.

2.1.1 LA SYNTAXE RDF

RDF est un modèle de données sous forme de graphe, il n'a pas à proprement parler de syntaxe. Plusieurs syntaxes peuvent être utilisées pour représenter une description RDF. De manière abstraite, la structure sous-jacente à toute expression RDF est une collection de triplets, chacun constitué d'un sujet, d'un prédicat et d'un objet. Un ensemble de tels triplets forme un graphe RDF. Le sujet d'un triplet peut être une URI ou un nœud vide, c'est-à-dire un nœud qui désigne une ressource sans la nommer. Le prédicat, qui représente une propriété, est toujours une URI. Finalement, l'objet représente la valeur de la propriété et peut être une URI, un nœud vide ou un littéral. Un nœud vide est tout nœud qui n'est ni une URI, ni un littéral. Il s'agit d'un nœud unique qui peut apparaitre dans plusieurs triplets, et qui n'a pas de nom intrinsèque. Un nœud vide représente une ressource anonyme. L'utilisation des URI pour désigner les ressources décrites par un graphe RDF a plusieurs avantages. Elles permettent de ne pas mélanger les désignations utilisées et de partager le même vocabulaire tout en évitant les conflits de noms à plusieurs applications. Un graphe peut contenir deux types de littéral :

- Un **littéral simple** est constitué d'une chaîne de caractères, appelée forme lexicale, et facultativement d'un attribut indiquant la langue.
- Un **littéral typé** est formé d'une chaîne de caractères et d'une URI indiquant un type qui sera utilisé pour décoder cette chaine. Par exemple, la chaîne typée "10"^^xsd:integer indique que la chaîne 10 doit être interprétée comme un entier, selon le type xsd:integer défini dans la norme de XML Schéma.

RDF n'a pas de types prédéfinis. Il faut utiliser une ressource externe à RDF pour interpréter un littéral typé. Une application peut définir ses propres types. Plus formellement, un type de donnée en RDF est toujours identifié par une URI et suppose l'existence d'un espace lexical *L*, d'un espace de valeur *V* et d'une fonction $m : L \rightarrow V$. l'espace lexical L est l'ensemble de toutes les chaînes de

caractère possibles pour ce type, alors que *V* est l'ensemble de toutes les valeurs possibles. L'application *m* doit être définie telle que :

- Chaque membre de *L* est associé à exactement un membre de V
- Pour chaque valeur de V, il peut y avoir zéro ou plusieurs membres de L qui lui sont associés.

La syntaxe RDF/ XML est la plus utilisée pour la représentation des graphes RDF. Cette syntaxe est un format basé sur XML et recommandé par le W3C pour la représentation RDF. La représentation du graphe RDF de la Figure 4 est la suivante :

```
<?xml version="1.0"?>
<rdf:RDF
       xmlns:local="https://checksem.u-bourgogne.fr/"
       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
       xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdf:Description rdf:about="https://checksem.u-bourgogne.fr/doctorant#MahdiGueffaz">
      <local:worksAt rdf:resource="http://checksem.u-bourgogne.fr/lieu#Dijon"/>
</rdf:Description>

<rdf:Description rdf:about="https://checksem.u-bourgogne.fr/doctorant#MahdiGueffaz">
      <local:hasName>Mahdi Gueffaz</local:hasName>
</rdf:Description>

<rdf:Description rdf:about="https://checksem.u-bourgogne.fr/doctorant#MahdiGueffaz">
      <local:hasHomePage rdf:resource="https://checksem.u-bourgogne.fr/mgueffaz"/>
</rdf:Description>
</rdf:RDF>
```

Script 1. Représentation RDF/XML d'un graphe RDF

Dans le Script 1, le graphe RDF est décrit à l'intérieur des balises *rdf:RDF*. Plusieurs espaces de nommage y sont d'abord spécifiés, en particulier celui qui correspond à notre préfixe local, par exemple, on pourra utiliser d'autres espaces de nommage comme :

- FOAF (Friend Of A Friend) vocabulaire RDF permettant de décrire des personnes et les relations qu'elles entretiennent entre elles ;
- Dublin Core pour la description de documents.

Viennent ensuite les trois descriptions de la ressource MahdiGueffaz. La description d'une ressource est spécifiée par la balise *rdf:Description*, avec l'URI de la ressource indiquée par l'attribut *rdf:about*. Pour les deux descriptions dont l'objet est une URI, on remarque que cette URI est elle aussi représentée par un attribut, cette fois-ci dans une balise correspondant au prédicat. Dans le cas du littéral, la valeur est mise tout simplement entre les balises qui correspondent au prédicat. Finalement, notons que dans les attributs, les URI doivent être indiquées au format long, en accord avec la norme XML.

Les nœuds vides. Dans la syntaxe RDF/XML, un nœud vide est représenté tout simplement par une balise *rdf:Description* ne contenant aucun attribut *rdf:about*, ce qui revient à dire qu'il s'agit d'une description qui ne précise pas la ressource décrite. Tout comme dans la syntaxe N-Triples, un nœud vide peut avoir un identificateur. Il suffit pour cela d'ajouter l'attribut *rdf:nodeID* dans la balise *Description* et d'y associer un identificateur unique.Un nœud vide ne peut être que le sujet ou l'objet d'un triplet. En aucun cas on ne pourra utiliser une propriété anonyme, ce qui signifie que le prédicat sera toujours désigné par une URI.

Les littéraux. Un littéral simple est représenté en l'insérant tout simplement entre les balises qui désignent le prédicat ayant ce littéral comme valeur. Si en plus la langue utilisée est spécifiée, alors il suffit de mettre l'attribut *xml:lang* dans la balise du prédicat et de spécifier la langue. En ce qui concerne les littéraux typés, ils sont représentés par l'attribut *rdf:datatype* dont la valeur indique le type utilisé pour interpréter le littéral.

Identification de l'URL de base. Rappelons qu'une des caractéristiques importantes du web sémantique est la possibilité de décrire une même ressource dans des documents différents. Jusqu'à maintenant, nous avons toujours identifié une ressource en utilisant l'attribut *rdf:about*.

N'importe quel document du web peut utiliser cette URI pour fournir des descriptions de cette ressource. Mais il n'y a pas à proprement parler d'endroit où cette URI est définie. Au lieu de désigner une ressource en utilisant une URI complète associée à l'attribut *rdf:about*, on utilise plutôt l'attribut *rdf:ID* avec comme valeur un fragment. Ceci a pour effet de désigner implicitement la ressource par une URI qui résulte de la fusion de l'URI correspondant au document où se trouve la description avec le fragment en question. Un des intérêts d'utiliser ce mécanisme est de contraindre l'unicité de l'identificateur dans le document. On s'assure donc ainsi que l'URI ne sera définie qu'à un seul endroit. Cette solution, bien qu'intéressante, crée un problème lorsque l'on veut définir une URI en utilisant la même base, mais dans des documents différents. Pour répondre à ce problème, une autre approche consiste à d'utiliser un mécanisme qui permet de spécifier explicitement la base qui sera utilisée pour former l'URI, en utilisant l'attribut *xml:base* dans la balise RDF du document. Chaque fois que l'on aura un attribut *rdf:ID* dans ce document, l'URI sera formée en prenant non pas l'URI du document, mais plutôt celle spécifiée par cet attribut. C'est ainsi que l'on pourra avoir la même base dans des documents différents.

Identification des types de ressources. En RDF, il est important de noter que les ressources n'entrent pas toutes dans la même catégorie. Par exemple, un professeur, un département universitaire et une page personnelle sont des entités de types différents. L'ajout d'une propriété entre une ressource et son type permet de les différencier. Pour différencier ces types, la norme RDF utilise l'alias *rdf:* pour désigner le préfixe http://www.w3.org/1999/02/22-rdf-syntax-ns#. Notre exemple peut être complété en indiquant les types des ressources (Figure 6)

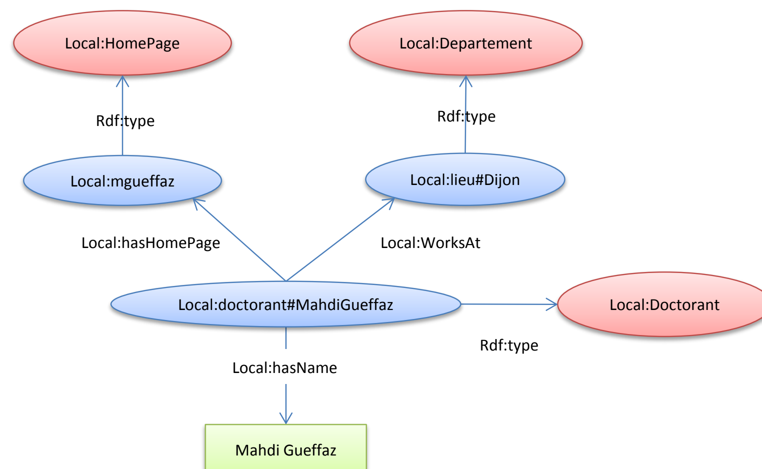


Figure 6. Identification des types de ressources.

```
<?xml version="1.0"?>
<rdf:RDF
        xmlns:local="https://checksem.u-bourgogne.fr/"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdf:Description rdf:about="https://checksem.u-bourgogne.fr/doctorant#MahdiGueffaz">
        <rdf:type rdf:resource="https://checksem.u-bourgogne.fr/Doctorant"/>
        <local:worksAt>
                <rdf:Description rdf:about="https://checksem.u-bourgogne.fr/lieu#Dijon">
                        <rdf:type rdf:resource="https://checksem.u-bourgogne.fr/Department"/>
                </rdf:Description>
        </local:worksAt>

        <local:hasName>Mahdi Gueffaz</local:hasName>

        <local:hasHomePage>
                <rdf:Description rdf:about="https://checksem.u-bourgogne.fr/mgueffaz">
                        <rdf:type rdf:resource="https://checksem.u-bourgogne.fr/HomePage"/>
                </rdf:Description>
        </local:hasHomePage>
</rdf:Description>
</rdf:RDF>
```

Script 2. Représentation RDF/XML de la figure 6

RDF n'interdit pas qu'une ressource ait plus d'un type. Mais un seul de ses types pourra faire l'objet de l'abréviation de RDF/XML qui permet de remplacer la balise *rdf:Description* par le type. Tous les autres types devront être définis explicitement.

Conteneurs. Un conteneur est une ressource qui contient d'autres ressources. RDF propose trois classes de conteneur : *rdf:Bag*, *rdf:Seq* et *rdf:Alt*. Le premier désigne un conteneur dont les membres n'ont aucun ordre entre eux, contrairement au second. Le conteneur *rdf:Alt* désigne un conteneur présentant des alternatives parmi lesquelles une seule doit être sélectionnée. Le conteneur est relié à chacun de ses membres par une relation *rdf:_n*, où *n* est un entier. Dans le cas d'un conteneur de type *rdf:Seq*, on s'attend à ce que *n* représente l'ordre du membre en question. Il n'y pas vraiment de contraintes sur la manière de décrire les conteneurs. RDF n'interdit pas d'avoir deux membres avec la même valeur *n*, même avec le conteneur *rdf:Seq*. Il n'interdit pas non plus qu'il y ait des sauts dans la numérotation.

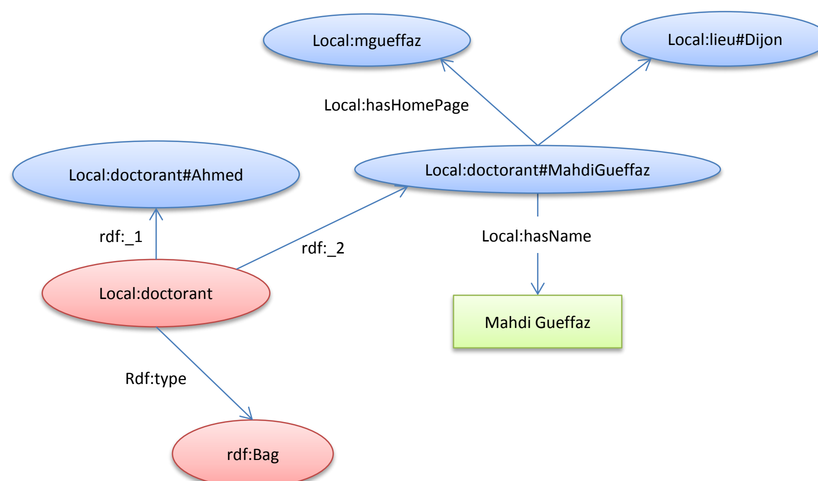


Figure 7. Description de conteneur.

```xml
<?xml version="1.0"?>
<rdf:RDF
       xmlns:local="https://checksem.u-bourgogne.fr/"
       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
       xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdf:Description rdf:about="https://checksem.u-bourgogne.fr/doctorant#MahdiGueffaz">
       <local:worksAt rdf:resource="https://checksem.u-bourgogne.fr/lieu#Dijon"/>
       <local:hasName>Mahdi Gueffaz</local:hasName>
       <local:hasHomePage rdf:resource="https://checksem.u-bourgogne.fr/mgueffaz"/>
</rdf:Description>

<rdf:Bag rdf:about="https://checksem.u-bourgogne.fr/doctorant">
       <rdf:li rdf:resource="https://checksem.u-bourgogne.fr/doctorant#Ahmed"/>
       <rdf:li rdf:resource="https://checksem.u-bourgogne.fr/doctorant#MahdiGueffaz"/>
</rdf:Bag>
</rdf:RDF>
```

Script 3. Représentation RDF/XML de la figure 7

Pour simplifier l'écriture, RDF/XML fournit une abréviation, en utilisant la relation *rdf:li* pour chaque membre, au lieu de la relation spécifique *rdf:_n*. Une telle notation suppose que les relations *rdf:_1*, *rdf:_2*, et ainsi de suite, sont générées automatiquement.

Collection. Si au sein d'une description, on spécifie l'existence de trois éléments dans un même conteneur, il n'y a aucune garantie que ce conteneur contienne seulement ces trois éléments. Les conteneurs ne sont pas considérés comme des ensembles fermés. Pour pallier cette lacune, RDF permet de définir des collections sous forme de liste. La liste vide est représentée par une ressource spéciale prédéfinie dont l'URI est *rdf:nil*. On construit une liste de manière récursive en utilisant le prédicat *rdf:first* pour indiquer le premier élément de la liste, et le prédicat *rdf:rest* pour indiquer le reste de la liste, qui est lui-même une liste.

Cette syntaxe est très lourde, mais il existe une abréviation, qui ne pourra être utilisée que si la liste est l'objet d'un triplet. La syntaxe RDF/XML propose l'utilisation de l'attribut *rdf:parseType="Collection"* dans la balise qui représente la propriété, et fournit la liste des éléments. Il est important de remarquer que si l'on utilise la forme non abrégée pour décrire une liste, il n'y a pas réellement de contrainte sur l'utilisation des prédicats *rdf:first* et *rdf:rest*. Rien n'empêche d'utiliser plus d'une occurrence de ces relations pour un même nœud. On pourrait les utiliser pour décrire un arbre au lieu d'une simple liste.

Réification. Pour ajouter des informations à un triplet, il faut utiliser une technique appelée réification. L'idée consiste à ajouter une ressource de type rdf:Statement. Une telle ressource représente un triplet. Pour spécifier les informations contenues dans ce triplet, les propriétés *rdf:subject*, *rdf:predicate* et *rdf:objet* sont utilisées. Elles définissent respectivement le sujet, le prédicat et l'objet du triplet concerné. La technique de réification est utilisée sur notre exemple (Figure 8, Script 4).

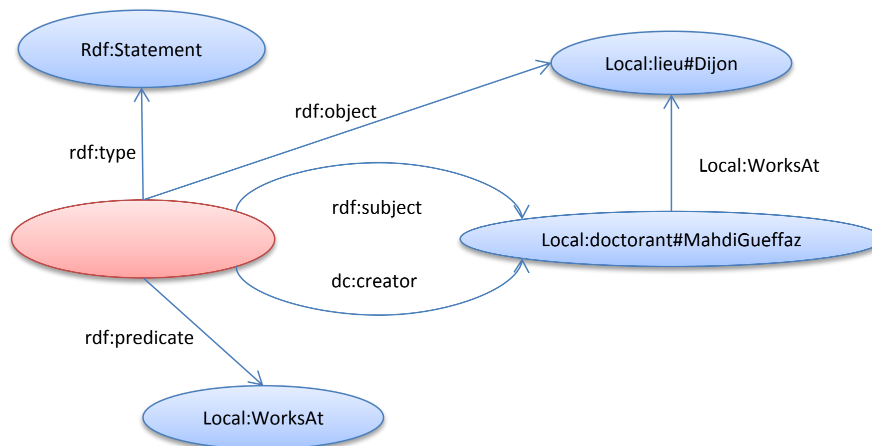


Figure 8. Exemple de réification.

```
<?xml version="1.0"?>
<rdf:RDF
        xmlns:local="https://checksem.u-bourgogne.fr/"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:dc="http://purl.org/dc/elements/1.1/">

<rdf:Description rdf:about="https://checksem.u-bourgogne.fr/doctorant#MahdiGueffaz">
        <local:worksAt rdf:resource="https://checksem.u-bourgogne.fr/lieu#Dijon"/>
</rdf:Description>

<rdf:Statement>
        <rdf:predicate rdf:resource="https://checksem.u-bourgogne.fr/worksAt"/>
        <dc:creator rdf:resource="https://checksem.u-bourgogne.fr/doctorant#MahdiGueffaz "/>
        <rdf:object rdf:resource="https://checksem.u-bourgogne.fr/lieu#Dijon"/>
        <rdf:subject rdf:resource="https://checksem.u-bourgogne.fr/doctorant#MahdiGueffaz"/>
</rdf:Statement>
</rdf:RDF>
```

Script 4. Représentation RDF/XML de la figure 8

Une ressource qui correspond à un triplet peut être identifiée. Il est possible d'indiquer quels sont le sujet, le prédicat et l'objet de ce triplet. Mais il n'est pas possible de dire que ce triplet correspond à un triplet spécifique dans un modèle RDF. Par exemple, si deux personnes décrivent une même ressource dans deux documents différents, il est possible que certains triplets identiques se retrouvent dans les deux descriptions. Rien n'empêche d'avoir deux triplets identiques dans un même modèle. Dans la syntaxe RDF/XML il existe une abréviation qui permet de créer automatiquement la réification d'un triplet. Il s'agit d'ajouter l'attribut *rdf:ID* à la propriété. Ceci aura pour effet de créer automatiquement une ressource de type *rdf:Statement* décrivant le triplet en question, et dont l'URI d'identification sera celle fournie à l'attribut *rdf:ID* de la propriété. Cette solution permet de créer un lien indirect entre le triplet et la description réifiée de ce triplet. Il est important de noter que même si l'URI de la réification est la même que celle du prédicat du triplet, cela ne représente toujours pas une manière formelle d'indiquer l'identité entre ces deux entités. Pour y arriver, il faudrait pouvoir identifier tout le triplet comme une ressource, ce qui n'est pas possible en RDF.

2.1.2 Le vocabulaire RDF

Pour des raisons historiques, le vocabulaire utilisé précédemment pour formaliser RDF a été dénommé RDF Schéma, abrégé RDFS (ou encore RDF *Vocabulary Language*). Le langage RDFS a été

développé en se basant sur RDF (Brickley and Guha, 2000). Le modèle des données RDF ne précise que le mode de description des données, mais ne fournit pas la déclaration des propriétés spécifiques au domaine ni la manière de définir ces propriétés avec d'autres ressources. RDFS a pour but d'étendre RDF en décrivant plus précisément les ressources utilisées pour étiqueter les graphes. Pour cela, il fournit un mécanisme permettant de spécifier les classes dont les instances seront des ressources, comme les propriétés. RDFS s'écrit toujours à l'aide de triplets RDF en utilisant deux propriétés fondamentales *rdfs:subClassOf* et *rdf:type* pour représenter respectivement les relations de subsomption entre classes et les relations d'instanciation entre instances et classes. Les classes spécifiques au domaine sont déclarées comme des instances de la ressource *rdfs:Class* et les propriétés spécifiques au domaine comme des instances de la ressource *rdf:Property*. Les propriétés *rdfs:subClassOf* et *rdfs:subPropertyOf* permettent de définir des hiérarchies de classes et de propriétés. D'autre part, RDFS ajoute à RDF la possibilité de définir les contraintes de domaine et co-domaine de valeurs avec l'aide des attributs *rdfs:domain* et *rdfs:range*. Les ressources instanciées sont décrites en utilisant le vocabulaire donné par les classes définies dans ce schéma. Pour résumer, XML peut être vu comme la couche de transport syntaxique, RDF comme un langage relationnel de base. RDFS offre des primitives de représentation de structures ou primitives ontologiques (Laublet et al., 2002).

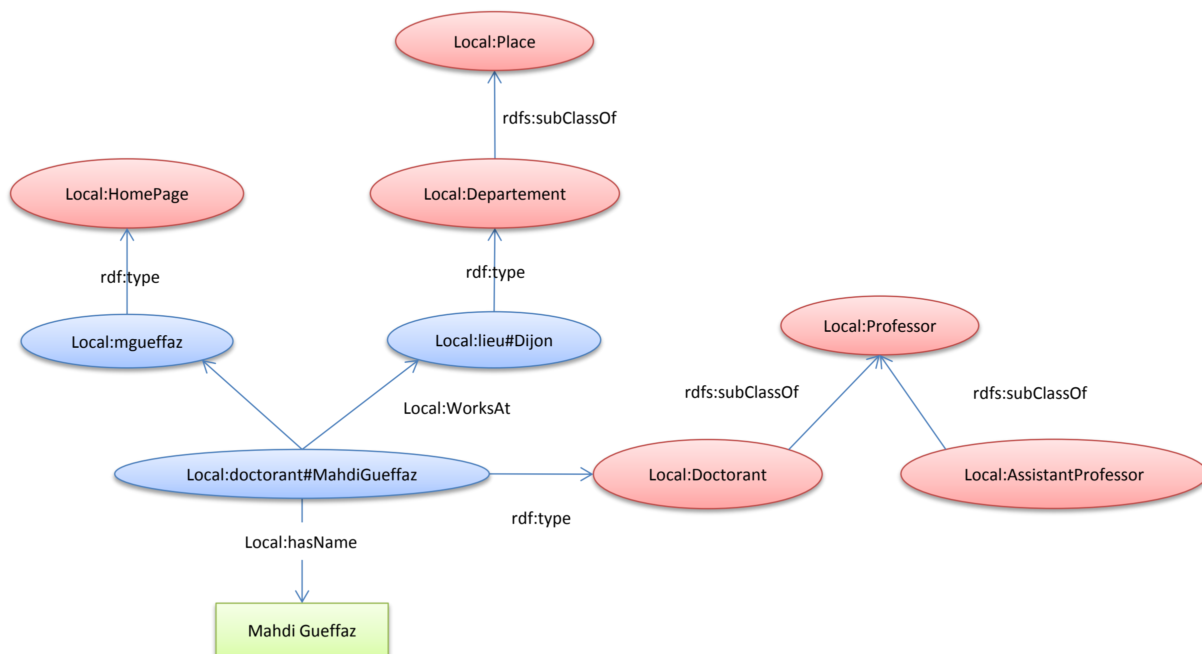


Figure 9. Hiérarchie de classes.

Le plus intéressant en RDFS est probablement la possibilité de spécifier le domaine ou l'image d'une propriété. Cette opération est réalisée par l'ajout d'un triplet qui indique un type de ressource que l'on peut retrouver comme sujet ou objet d'une propriété, en utilisant les relations *rdfs:domain* ou *rdfs:range*, respectivement. Ainsi, dans notre exemple, on peut spécifier que c'est un membre de personnel (StaffMember) qui travaille (*worksAt*) à un lieu de travail (*WorkPlace*). On peut aussi spécifier que la propriété "*travailler*" est une sous-propriété de la propriété "*avoir activité*" (hasActivity) (Figure 10).

Dans le web sémantique, il faut s'attendre à ce que les descriptions RDF se retrouvent dans des documents différents et éventuellement dispersés à travers le monde. Un agent intelligent dont les décisions dépendront de ces données pourra donc être amené à regrouper des modèles RDF provenant de sources différentes. Ce qui nous oblige à nous interroger sur la manière de combiner ces données. Nous ne pourras pas nous contenter de faire une simple union des triplets contenus dans chaque source (à cause de la présence éventuelle de nœuds vides dans un graphe RDF). Nous pouvons penser qu'il suffit de considérer que deux nœuds vides situés dans des documents différents seront toujours distincts dans le modèle résultant. Mais cette solution ne fonctionnera pas si par coïncidence les deux graphes ont chacun un nœud vide avec un identificateur identique. Il faudra donc, avant de fusionner deux graphes RDF, renommer systématiquement les nœuds vides afin d'éviter les conflits d'identificateur. La fusion de deux graphes RDF est définie formellement de la manière suivante : soit G1 et G2 deux graphes RDF que l'on veut fusionner. Si les graphes n'ont aucun nœud vide avec le même identificateur, la fusion est simplement l'union des triplets des deux graphes. Sinon, il faut créer un graphe G’2 égal à G2, à la différence près que tous les identificateurs des nœuds vides de G’2 sont renommés de telle manière qu'aucun ne soit identique à un identificateur d'un nœud vide de G1. La fusion sera alors le résultat de l'union des triplets de G1 et G’2.

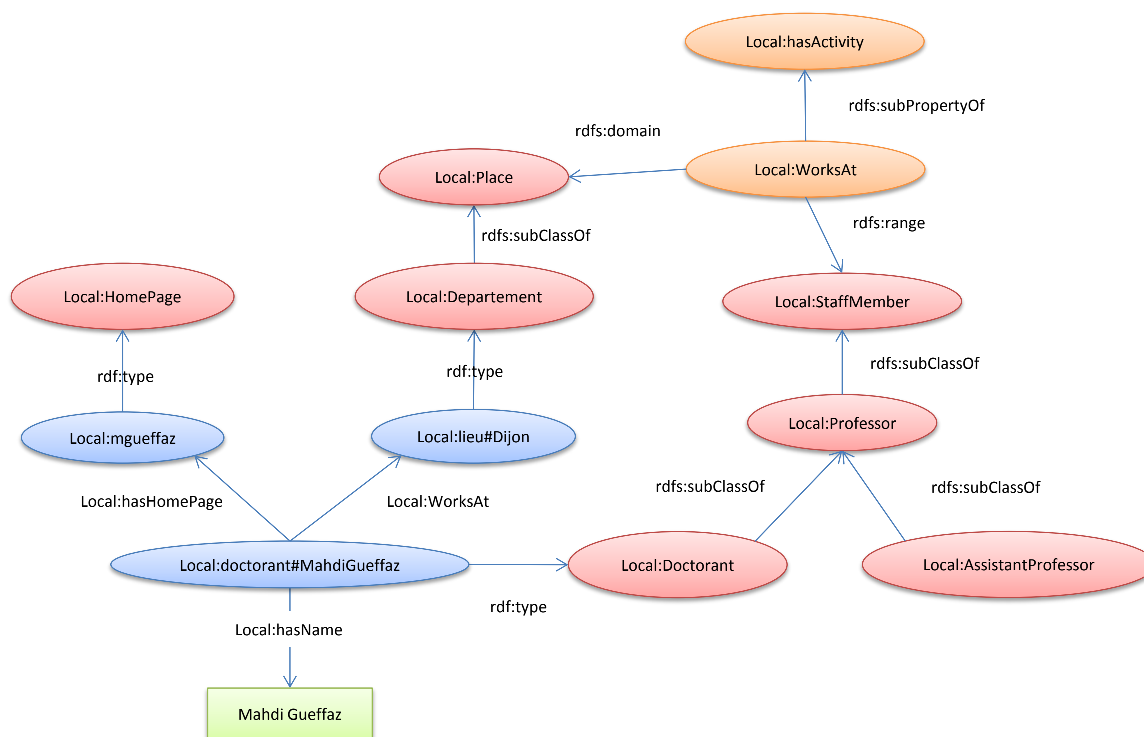


Figure 10. Hiérarchie de propriétés.

2.2 Les ontologies OWL

Le W3C a mis au point OWL pour étendre le langage RDF. Ce dernier, en effet, compte un certain nombre de limites comme:

- *rdfs:range* définit le domaine de valeurs d’une propriété quelle que soit la classe concernée. Par exemple, il ne permet pas d’exprimer que les vaches ne mangent que de l’herbe alors que d’autres sortes d’animaux mangent également de la viande.

- RDFS ne permet pas d'exprimer que deux classes sont disjointes. Par exemple, les classes des hommes et des femmes sont disjointes.
- RDFS ne permet pas de créer des classes par combinaison ensembliste d'autres classes (intersection, union, complément). Par exemple, on veut construire la classe Personne comme l'union disjointe des classes des hommes et des femmes.
- RDFS ne permet pas de définir de restriction sur le nombre d'occurrences de valeurs que peut prendre une propriété. Par exemple, on ne peut pas dire qu'une personne a exactement deux parents.
- RDFS ne permet pas de définir certaines caractéristiques des propriétés : transitivité (par exemple : estPlusGrandQue), unicité (par exemple: estLePèreDe), propriété inverse (par exemple : mange est la propriété inverse de estMangéPar).

Si ces contraintes d'expressivité se montrent trop importantes, Il est nécessaire d'utiliser le langage OWL pour modéliser la sémantique des ressources. OWL est un langage utilisé pour représenter des ontologies dans le web sémantique. Il offre aux machines de plus grandes capacités d'interprétation du contenu web que celles permises par XML, RDF, RDF Schéma, grâce à un vocabulaire supplémentaire et une sémantique formelle. Inspiré des logiques de descriptions, OWL fournit un grand nombre de constructeurs permettant d'exprimer, de façon très fine, les classes de manière plus complexe correspondant :

- aux connecteurs de la logique de description équivalente (intersection, union, restrictions diverses, etc.),
- aux propriétés des classes définies (tel que la disjonction),
- à la cardinalité (par exemple «exactement un »),
- aux types des propriétés (propriétés d'objet ou d'annotation),
- aux caractéristiques des propriétés (par exemple la symétrie, la transitivité),
- et aux classes énumérées.

Le W3C a fractionné le langage OWL en trois sous langages offrant des capacités d'expression croissantes et, naturellement, destinés à des communautés différentes d'utilisateurs :

- **OWL Lite** : ne contient qu'un sous-ensemble réduit des constructeurs disponibles, mais son utilisation assure que la comparaison de types pourra être calculée (un problème de complexité NP, donc « simple » en représentation de connaissances);
- **OWL DL** : est fondé sur la logique descriptive. Il contient l'ensemble des constructeurs, mais avec des contraintes particulières sur leur utilisation qui assurent la décidabilité de la comparaison de types. Par contre, la grande complexité de ce langage semble rendre nécessaire une approche heuristique;
- **OWL Full** : est la version la plus complexe d'OWL, mais également celle qui permet le plus haut niveau d'expressivité. OWL Full est destiné aux situations où il est plus important d'avoir un haut niveau de description, quitte à ne pas pouvoir garantir la complétude et la décidabilité des calculs liés à l'ontologie. Sans aucune contrainte, dans ce cas, le problème de comparaison de types est vraisemblablement indécidable.

Toute ontologie OWL Lite valide est également une ontologie OWL DL valide, et toute ontologie OWL DL valide est également une ontologie OWL Full valide.La définition la plus utilisée d'une ontologie est celle de (Gruber, 1993) :«*une ontologie est une spécification explicite d'une conceptualisation*». Une conceptualisation est une abstraction du monde que nous souhaitons représenter dans un certain but. La conceptualisation est le résultat d'une analyse ontologique du domaine étudié. L'ontologie est une spécification parce qu'elle représente la conceptualisation dans une forme concrète. Elle est explicite parce que tous les concepts et les contraintes utilisés sont explicitement définis. Une ontologie exprime la conceptualisation explicitement dans un langage formel. Une définition explicite et formelle permet aux agents de raisonner et d'inférer de nouvelles connaissances. Les ontologies sont classées selon le sujet et la structure de la conceptualisation, on distingue :

- **Les ontologies de domaine** : Les plus connues, elles expriment des conceptualisations spécifiques à un domaine, elles sont réutilisables pour des applications sur ce domaine.
- **Les ontologies d'application** : elles contiennent des connaissances du domaine nécessaire à une application donnée ; elles sont spécifiques et non réutilisables.
- **Les ontologies génériques** : appelées aussi ontologies de haut niveau, elles expriment des conceptualisations très générales telles que le temps, l'espace, l'état, le processus, les composants, elles sont valables dans différents domaines ; les concepts figurant dans une ontologie du domaine sont subsumés par les concepts d'une ontologie génériques, la frontière entre les deux étant floue.
- **Les ontologies de représentation ou méta-ontologies** : indiquent des formalismes de représentation de la connaissance, les ontologies génériques ou du domaine peuvent être écrites en utilisant des primitives d'une telle ontologie.

Logiques descriptives. Pendant mes travaux de recherche, je n'ai utilisé que des ontologies OWL basées sur la logique descriptive (DL). Le développement des logiques descriptives fut fortement influencé par les travaux sur la logique des prédicats, les schémas (frames) (Minsky, 1981) et les réseaux sémantiques. Des correspondances existent entre les LD et ces formalismes (Sattler et al., 2003) (Baader et Nutt, 2003). La présence de catégories générales d'objets et de relations fait d'ailleurs partie de l'héritage conceptuel des schémas et des réseaux sémantiques. La modélisation des connaissances d'un domaine avec les logiques descriptives se réalise en deux niveaux. Le premier, le niveau terminologique ou TBox, décrit les connaissances générales d'un domaine alors que le second, le niveau factuel ou ABox représente une configuration précise. Une TBox comprend la définition des concepts et des rôles, alors qu'une ABox décrit les individus en les nommant et en spécifiant en termes de concepts et de rôles, des assertions qui portent sur ces individus nommés.

L'objectif principal des LD consiste à pouvoir raisonner efficacement pour minimiser les temps de réponse. Par conséquent, la communauté scientifique a publié de nombreuses recherches qui portent sur l'étude du rapport expressivité/performance des différentes LD (Nardi et Brachman, 2003). La principale qualité des LD réside dans leurs algorithmes d'inférence dont la complexité est souvent inférieure aux complexités des démonstrateurs de preuves de la logique de premier ordre (Tsarkov et Horrocks, 2003). En général, pour appliquer une procédure d'inférence, la sémantique du langage n'est pas utilisée. On utilise plutôt des règles de déduction qui respectent la sémantique du

langage, c'est-à-dire des règles qui nous permettent de déduire de nouveaux faits qui sont nécessairement des conséquences logiques des faits originaux. Toutes les règles ont la forme « Si E contient le triplet T, alors ajouter le triplet T' ». À partir d'un graphe E, on obtient alors un graphe E' = E ∪ {T'} qui est une conséquence logique de E.

Moteur d'inférence. Un logiciel qu'on appelle raisonneur ou moteur d'inférence peut faire des inférences et donc créer automatiquement de la connaissance. Par exemple, soit deux personnes et une relation « ami » dite symétrique, si la personne A est ami de la personne B alors le raisonneur peut déduire de lui-même que la personne B est ami de la personne A. Ce mécanisme peut sembler simple et logique, mais il n'est pas naturel pour une machine. Il faut donc lui apprendre. Il y a plusieurs moteurs d'inférence comme Racer, Pellet, FaCT++ (Tsarkov et Horrocks, 2006) qui sont utilisés pour vérifier la qualité des ontologies.

2.2.1 STRUCTURE D'UNE ONTOLOGIE

La conception d'une ontologie OWL prend en compte la nature distribuée du web sémantique et, de ce fait, intègre la possibilité d'étendre des ontologies existantes, ou peut employer diverses ontologies existantes pour compléter la définition d'une nouvelle ontologie.

Afin de pouvoir employer des termes dans une ontologie, il est nécessaire d'indiquer avec précision de quels vocabulaires ces termes proviennent. C'est la raison pour laquelle, comme tout autre document XML, une ontologie commence par une déclaration d'espace de nom (parfois appelée « espace de nommage ») contenue dans une balise *rdf:RDF*.

Tout comme il existe une section d'en-tête en haut de tout document bien formé (par exemple XHTML), on peut écrire, à la suite de la déclaration d'espaces de nom, un en-tête décrivant le contenu de l'ontologie courante. C'est la balise *owl:Ontology* qui permet d'indiquer ces informations. Les différents composants d'une ontologie OWL sont la classe, la propriété et l'instance.

Les classes. Une classe définit un groupe d'individus qui sont réunis par des caractéristiques similaires. Les classes peuvent être organisées hiérarchiquement selon une taxonomie. Les classes définies par l'utilisateur sont d'ailleurs toutes des enfants de la « super-classe » *OWL:Thing*. L'ensemble des individus d'une classe est désigné par le terme « extension de classe », chacun de ces individus étant alors une « instance » de la classe. Les trois versions d'OWL comportent les mêmes mécanismes de classe, sauf pour OWL FULL qui est la seule version à permettre qu'une classe soit l'instance d'une autre classe (qui est alors appelé métaclasse). Le concept d'héritage est disponible en OWL à l'aide de la propriété *subClassOf* décrite précédemment dans RDF Schéma.

Une propriété. Les propriétés OWL expriment des faits au sujet des classes et de leurs instances. Par exemple la couleur d'une voiture, son modèle, sa puissance sont des propriétés d'une classe Voiture. OWL fait la distinction entre deux types de propriétés:

- les propriétés d'objet qui permettent de relier des instances à d'autres instances
- les propriétés de type de données qui permettent de relier des individus (objet perçu dans le monde réel) à des valeurs de données.

Une propriété d'objet est une instance de la classe *owl:ObjectProperty*, une propriété de type de données est une instance de la classe *owl:DatatypeProperty*. Ces deux classes sont elles-mêmes

sous-classes de la classe RDF *rdf:Property*. Les propriétés peuvent aussi être organisées hiérarchiquement.

Une instance. La définition d'un individu consiste à énoncer un «fait», encore appelé « axiome d'individu ». On peut distinguer deux types de faits :

- **les faits concernant l'appartenance à une classe** : La plupart des faits concerne généralement la déclaration de l'appartenance à une classe d'un individu et les valeurs de propriété de cet individu.
- **les faits concernant l'identité des individus** : Une difficulté qui peut éventuellement apparaître dans le nommage des individus concerne la non-unicité éventuelle des noms attribués aux individus. C'est la raison pour laquelle OWL propose un mécanisme permettant de lever cette ambiguïté, à l'aide des propriétés *owl:sameAs*, *owl:diffrentFrom* et *owl:allDifferent*.

2.2.2 Exemple

Dans cet exemple, l'ontologie créée représente un groupe de personnes et leurs relations de parenté, ainsi que leur lieu d'habitation (Lacot, 2005). La création d'une ontologie s'effectue en deux étapes, une étape pour créer le schéma de l'ontologie et une autre étape pour peupler cette dernière. L'ontologie est composée d'humains, divisée en deux sous classes Homme et Femme, et habitant dans une certaine ville. Un humain peut avoir un lien de fraternité avec un autre humain. Un homme peut être père d'un autre humain, et un homme et une femme peuvent être mariés. Dans une première étape, nous allons créer le schéma de l'ontologie. Dans une seconde étape, nous peuplerons l'ontologie (nous associerons des instances aux éléments du modèle).

Écriture des classes. La première étape de l'écriture de l'ontologie OWL représentant cette population consiste à décrire les classes de l'ontologie. Cette ontologie comporte 5 classes : Humain, Homme, Femme, Pays et ville. On peut ajouter des contraintes, notamment sur les classes Humain et Ville. Un humain a toujours un père et une ville se trouve forcément dans un pays (Script 5).

```
<!-- Définition des classes -->
<owl:Class rdf:ID="Humain" />
<owl:Class rdf:ID="Homme">
	<rdfs:subClassOf rdf:resource="#Humain" />
</owl:Class>
<owl:Class rdf:ID="Femme">
	<rdfs:subClassOf rdf:resource="#Humain" />
</owl:Class>
<owl:Class rdf:ID="Ville" />
<owl:Class rdf:ID="Pays" />

<!--contrainte sur la classe Humain et Ville-->
<owl:Class rdf:ID="Humain">
	<rdfs:subClassOf>
		<owl:Restriction>
			<owl:onProperty rdf:resource="#aPourPere" />
			<owl:cardinality
			     rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
		</owl:Restriction>
	</rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="Ville">
	<rdfs:subClassOf>
		<owl:Restriction>
```

```
                    <owl:onProperty rdf:resource="#seTrouveEn" />
                    <owl:cardinality
                         rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
               </owl:Restriction>
          </rdfs:subClassOf>
</owl:Class>
```

Script 5. Exemple de création de classes OWL

Écriture des propriétés. L'écriture des propriétés est l'étape qui va nous permettre de détailler la population que l'on veut décrire. Il y a deux types de propriétés à définir. Les propriétés d'objet et les propriétés de type de données. Les propriétés d'objet sont :

- **HabiteA** : un humain habite une ville.
- **aPourPere** : un humain a pour père un homme.
- **aUnLienDeFraternite** : un humain a un lien de fraternité avec un autre humain.
- **estmarieA** : un humain est marié à un autre humain.
- **seTrouveEn** : une ville se trouve dans un pays.

Les propriétés *aUnLienDeFraternite* et *estMarieA* sont définies comme des propriétés symétriques : si une telle relation lie un individu A à un individu B, alors la même relation lie également l'individu B à l'individu A (Script 6).

```
<!-- Propriétés d'objet -->
<owl:ObjectProperty rdf:ID="habiteA">
       <rdfs:domain rdf:resource="#Humain" />
       <rdfs:range rdf:resource="#Ville" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="aPourPere">
       <rdfs:domain rdf:resource="#Humain" />
       <rdfs:range rdf:resource="#Homme" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="aUnLienDeFraternite">
       <rdf:type rdf:resource="&owl;SymmetricProperty" />
       <rdfs:domain rdf:resource="#Humain" />
       <rdfs:range rdf:resource="#Humain" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="estMarieA">
       <rdf:type rdf:resource="&owl;SymmetricProperty" />
       <rdfs:domain rdf:resource="#Humain" />
       <rdfs:range rdf:resource="#Humain" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="seTrouveEn">
       <rdfs:domain rdf:resource="#Ville" />
       <rdfs:range rdf:resource="#Pays" />
</owl:ObjectProperty>
```

Script 6. Exemple de création de propriétés d'objets OWL

Les propriétés de type de données permettent d'affecter des propriétés quantifiées aux instances des classes. La classe Humain a un Nom (chaîne de caractères), un prénom (chaîne de caractères) et une date de naissance (date). La classe Ville a un nom de ville (chaîne de caractères), la classe Pays a un nom de pays (chaîne de caractères) et la classe femme a un nom de jeune fille (chaîne de caractères) (Script 7).

```
<!-- Propriétés de type de donnée -->
<owl:DatatypeProperty rdf:ID="nom">
	<rdfs:domain rdf:resource="#Humain" />
	<rdfs:range rdf:resource="&xsd;string" />
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="prenom">
	<rdfs:domain rdf:resource="#Humain" />
	<rdfs:range rdf:resource="&xsd;string" />
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="nomDeJeuneFille">
	<rdfs:domain rdf:resource="#Femme" />
	<rdfs:range rdf:resource="&xsd;string" />
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="dateDeNaissance">
	<rdfs:domain rdf:resource="#Humain" />
	<rdfs:range rdf:resource="&xsd;date" />
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="nomVille">
	<rdfs:domain rdf:resource="#Ville" />
	<rdfs:range rdf:resource="&xsd;string" />
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="nomPays">
	<rdfs:domain rdf:resource="#Pays" />
	<rdfs:range rdf:resource="&xsd;string" />
</owl:DatatypeProperty>
```

Script 7. Exemple de création de propriétés de type de donnée OWL

Après avoir défini cette ontologie OWL (classes et propriétés), il ffaut la peupler avec des individus ou avec des instances de classe. Il s'agit de l'instanciation des individus de la population, mais également de leur description par l'énonciation de leurs propriétés. Dans notre exemple, nous avons les instances de classe suivantes :

Classe	Id	Nom	Prénom	Date de naissance	Nom de jeune fille	Nom Pays	Nom ville
Homme	Pierre	Dupond	Pierre	1978-08-18	X	X	X
Homme	Paul	Dupond	Paul	1976-05-26	X	X	X
Homme	Jacques	Dupond	Jacques	1946-12-25	X	X	X
Femme	Marie	Dupond	Marie	1976-12-17	Martin	X	X
Pays	Allemagne	X	X	X	X	Allemagne	X
Pays	France	X	X	X	X	France	X
Ville	Paris	X	X	X	X	X	Paris
Ville	Ulm	X	X	X	X	X	Ulm

Tableau 1. Instanciation de classes.

Propriétés	aUnLienDeFraternite	aPourPere	HabiteA	estMarieA	SetrouveEn
Pierre	Paul	unknown	Ulm	X	X
Paul	X	Jacques	Paris	Marie	X
Jacques	X	Unknown	Paris	X	X
Marie	X	Unknown	Paris	X	X
Paris	X	X	X	X	France
Ulm	X	X	X	X	Allemagne

Tableau 2. Instanciation des propriétés.

L'individu *unknown* a été ajouté pour satisfaire la contrainte de parenté qui caractérise un humain. Sans cet individu, Jacques et Marie, dont le père n'apparait pas dans la population à décrire, poseraient un problème d'inconsistance vis-à-vis de la définition de la classe Humain (Script 8).

```
<!-- Humains -->
<Homme rdf:ID="unknown">
       <aPourPere rdf:resource="#unknown" /> </Homme>
<Homme rdf:ID="Pierre">
       <nom>Dupond</nom>
       <prenom>Pierre</prenom>
       <dateDeNaissance>1978-08-18</dateDeNaissance>
       <aUnLienDeFraternite rdf:resource="#Paul" />
       <aPourPere rdf:resource="#Jacques" />
       <habiteA rdf:resource="#Ulm" /> </Homme>
<Homme rdf:ID="Paul">
       <nom>Dupond</nom>
       <prenom>Paul</prenom>
       <dateDeNaissance>1976-05-26</dateDeNaissance>
       <estMarieA rdf:resource="#Marie" />
       <aPourPere rdf:resource="#Jacques" />
       <habiteA rdf:resource="#Paris" /> </Homme>
<Homme rdf:ID="Jacques">
       <nom>Dupond</nom>
       <prenom>Jacques</prenom>
       <dateDeNaissance>1946-12-25</dateDeNaissance>
       <aPourPere rdf:resource="#unknown" />
       <habiteA rdf:resource="#Paris" /> </Homme>
<Femme rdf:ID="Marie">
       <nom>Dupond</nom>
       <prenom>Marie</prenom>
       <dateDeNaissance>1976-12-17</dateDeNaissance>
       <nomDeJeuneFille>Martin</nomDeJeuneFille>
       <aPourPere rdf:resource="#unknown" />
       <habiteA rdf:resource="#Paris" /> </Femme>

<!-- Pays -->
<Pays rdf:ID="Allemagne">
       <nomPays>Allemagne</nomPays> </Pays>
<Pays rdf:ID="France">
       <nomPays>France</nomPays> </Pays>

<!-- Villes -->
<Ville rdf:ID="Paris">
       <nomVille>Paris</nomVille>
       <seTrouveEn rdf:resource="#France" /> </Ville>
<Ville rdf:ID="Ulm">
       <nomVille>Ulm</nomVille>
       <seTrouveEn rdf:resource="#Allemagne" /> </Ville>
```

Script 8. Exemple de peuplement d'ontologie

Une fois l'ontologie modélisée et peuplée, il est nécessaire d'utiliser des langages de requêtes spécifiques tels que le langage SPARQL.

2.3 LE LANGAGE DE REQUETE SPARQL

Le langage de requêtes SPARQL offre des services puissants pour extraire des informations sur de grands ensembles de données RDF. Pour gérer les graphes RDF, les dessins et les implémentations de plusieurs langages de requêtes RDF ont été proposés. En 2004, le Groupe de travail RDF Data Access, qui fait partie de l'activité web sémantique, a publié un premier projet de travail public d'un langage de requêtes pour RDF, appelé SPARQL (Prudhommeaux et Seaborne, 2005). Depuis lors, SPARQL a été rapidement adopté comme le standard pour l'interrogation des données du Web sémantique. En janvier 2008, SPARQL est devenu une recommandation du W3C. Les requêtes SPARQL (Chebotko et al., 2006) sont des requêtes sur les triplets qui constituent un graphe de données RDF. La requête SPARQL officielle introduit quatre formes différentes :

- Requête de la forme **SELECT**, renvoie la valeur de la variable, qui peut être lié par un modèle de requête correspondant;
- Requête de la forme **ASK**, renvoie vrai si la requête correspond aux données et faux sinon.
- Requête de la forme **CONSTRUCT**, renvoie un graphe RDF en remplaçant les valeurs dans les modèles de données;
- Requête de la forme **DESCRIBE**, renvoie un graphe RDF qui définit la ressource correspondante.

SPARQL est une technologie jeune, la spécification du W3C en cours (Schmidt, 2009) a encore quelques limitations. Ces limitations deviennent évidentes lorsqu'on compare SPARQL à des langages de requêtes établies, telles que SQL ou XQuery. La liste suivante présente les caractéristiques et les constructeurs qui sont (à ce jour) manquants dans SPARQL.

- **Agrégation** : la spécification actuelle ne prend pas en charge les fonctions d'agrégation, comme l'addition des valeurs numériques, le comptage ou le calcul de la moyenne.
- **Mises à jour** : alors que le standard SPARQL prend en charge l'extraction des données à partir de graphes RDF, des constructions pour l'insertion de nouveaux triplets dans les graphes RDF et la manipulation de graphiques existants (les clauses d'insertion et de mise à jour clauses dans le style de SQL) sont absentes.
- **Expressions de chemin** : SPARQL ne prend pas en charge la spécification des expressions de chemin, par exemple à partir d'une seule requête SPARQL il est impossible de calculer la fermeture transitive d'un graphique ou d'extraire tous les nœuds qui sont accessibles à partir d'un nœud fixe. Cette lacune a été maintes fois identifiée dans des publications antérieures et les différentes propositions pour l'intégration des expressions de chemin dans la langue ont été faites (Pérez et al., 2009) (Kochut et Janik, 2007) (Alkhateeb et al., 2009).
- **Vues** : dans les langages de requêtes traditionnels tels que SQL, des vues logiques sur les données jouent un rôle important. Ils sont indispensables à la conception de base de données et pour la gestion des accès. SPARQL ne prend actuellement pas en charge la spécification de vues logiques sur les données ; cependant les vues matérialisées au cours des données peuvent être extraites à partir du graphique d'entrée en utilisant des requêtes basées sur la forme Construct.
- **Prise en charge des contraintes** : le mécanisme d'affirmation et de vérification des contraintes d'intégrité dans la base de données RDF n'est pas géré dans la spécification

actuelle de SPARQL. En SQL, les contraintes d'intégrité sont dérivées implicitement des spécifications des clés primaires et étrangères établies dans la phase de conception du schéma. Au-delà de cela, il est possible d'appliquer des contraintes définies par l'utilisateur en utilisant la déclaration de création d'assertions.

Le travail dans (Lausen et al., 2008) montre que SPARQL - avec quelques extensions mineures - peut être utilisé pour exprimer une large classe de contrainte. SPARQL permet d'extraire des contraintes de graphes RDF lorsque celles-ci sont spécifiées en utilisant un vocabulaire prédéfini pour les contraintes d'encodage. Les fonctions d'agrégation pour SPARQL ont été proposées dans (Polleres et al., 2007). Ce travail définit une extension de SPARQL, appelé SPARQL++, qui intègre les fonctions standard d'agrégation dans les clauses *Construct* et *Filter*. L'objectif de ces travaux est de gérer les mappages de schéma à travers des requêtes SPARQL de type *Construct*. Il convient également de mentionner que certains moteurs SPARQL existants, par exemple ARQ (ARQ, 2012) et Virtuoso (Blakeley, 2007), on déjà mit en œuvre leurs propres stratégies pour l'agrégation. Les expressions de chemin par SPARQL ont été identifiées comme un élément important dans plusieurs contributions de recherche (Kochut et Janik, 2007) (Pérez et al., 2008) (Alkhateeb et al., 2009). L'idée commune à toutes ces approches est d'étendre SPARQL par des constructions qui permettent d'exprimer les relations entre les nœuds qui dépassent ce qui peut être exprimé par de simples motifs de graphes de base, par exemple les nœuds connectés transitivement. Il est naturel de supposer que l'interrogation des chemins est un élément important dans le contexte d'un modèle de données graphique structuré comme RDF. L'approche adoptée dans (Kochut et Janik, 2007) utilise des motifs de chemins ordinaires, semblables à des expressions régulières, pour exprimer les relations complexes entre les nœuds de chemin dans les graphes RDF. Ces motifs de chemin réguliers sont utilisés pour étendre SPARQL dans un dialecte appelé SPARQLeR. Dans (Pérez et al., 2008) une extension de SPARQL appelée nSPARQL est proposée. L'objectif est de naviguer dans un graphe RDF en utilisant un ensemble d'axes prédéfinis, dans le style des chemins XPath pour naviguer dans des documents XML. Une autre approche proposée est le langage de requête PSPARQL (Alkhateeb et al., 2009). Il s'appuie sur une version étendue de RDF, appelé PRDF (Alkhateeb et al., 2009), là où les arêtes du graphe (les prédicats dans les triplets RDF) peuvent transporter les motifs d'expressions régulières sous forme d'étiquettes. Le langage de requête PSPARQL est alors défini par rapport aux motifs PRDF.

Cette section sur SPARQL démontre que ce standard n'est pas encore stable et que de nombreuses propositions sont publiées ces dernières années pour compléter l'ensemble des éléments constitutifs de ce langage.

3 Évolution d'Ontologies

La dernière partie de ce chapitre est consacré à une brève description du domaine de la gestion des changements dans une ontologie. Les éléments présentés dans cette partie nous permettront d'aborder plus facilement les concepts présentés dans le chapitre 7 de ce document. La structure et le contenu des ontologies doivent évoluer dans le temps afin de rester en adéquation avec les concepts et les connaissances du domaine modélisé. Tout au long de leur cycle de vie, les ontologies évoluent pour répondre à différents besoins de changements. (Maedche et al., 2003) et (Stojanovic, 2004) pensent l'évolution comme « la modification appropriée d'une ontologie et la

propagation des changements dans les autres ontologies qui en dépendent ». En parallèle (Klein, 2004) et (Noy, 2004) ont proposé une autre définition plus précise: « l'évolution d'une ontologie est la capacité de gérer les changements apportés lors de l'évolution en créant et en maintenant différentes versions d'une ontologie. Cette capacité consiste à identifier et à différencier les versions, à modifier les versions, à spécifier des relations qui rendent explicites les changements effectués entre les versions ». La deuxième définition suppose qu'une ontologie peut avoir plusieurs états (appelés versions) qui peuvent être explicitement différenciés. Plusieurs recherches ont montré l'importance majeure de l'évolution d'ontologie ainsi que les lacunes pour gérer cette évolution. Il existe plusieurs méthodologies sur l'évolution d'ontologies.

La méthode AIFB (Institute of Applied Informatics and Formal Description Methods) proposée par l'université de Karlsruhe (Allemagne) repose sur six étapes décrites par (Stojanovic et al., 2002). L'étape de Détection des changements se base sur les besoins explicites (ajout, suppression d'une entité ontologique) exprimés par les développeurs d'ontologies et sur l'application de méthodes heuristiques sur trois niveaux différents : la structure, les instances et les usages. La représentation des changements, elle vise l'édition des changements élémentaires, composites ou complexes de façon à les adapter au langage ontologique utilisé. L'étape de sémantique des changements a pour objectif de conserver la consistance de l'ontologie après évolution en résolvant systématiquement des changements additionnels et en proposant à l'utilisateur des stratégies d'évolution. La quatrième étape est l'implémentation des changements qui consiste à notifier, à appliquer et à garder trace de tous les changements appliqués. L'avant-dernière étape est la propagation des changements, cette étape assure la consistance des objets dépendants après qu'une mise à jour de l'ontologie ait été effectuée. Ces objets peuvent inclure des ontologies et des applications fonctionnant avec l'ontologie. L'étape de validation permet aux utilisateurs d'évaluer les résultats de l'évolution. Si la qualité de l'ontologie est préservée ou améliorée, alors les changements peuvent être validés ; sinon il est possible de les annuler ou reprendre les phases d'évolution selon un processus cyclique.

La méthode IMSE (Information Management and Software Engineering) permet d'identifier et de différencier les versions, de modifier les versions, de spécifier des relations qui rendent explicites les changements effectués entre les versions et d'utiliser des mécanismes d'accès pour les artefacts dépendants (il s'agit objets qui dépendent de l'ontologie à évoluer comme les ontologies, les applications, etc.). Ce processus se base sur deux modules fondamentaux pour une méthodologie de 'versionning' des ontologies (Noy et Musen, 2004) (Klein, 2002) (Klein et Fensel, 2001). Ces deux modules sont :

- (i) le module d'analyse de relation entre deux versions d'ontologies qui permet la mise en évidence des changements effectués dans la définition des entités ontologiques ainsi que la spécification de la relation sémantique entre les entités ontologiques et la description des changements effectués par un ensemble de méta données.
- (ii) le module d'identification des versions d'ontologies qui permet de rendre opérationnelle la nouvelle version de l'ontologie après évolution.

Selon Rogozan (Rogozan, 2008), les auteurs de la méthodologie AIFB ne proposent aucune étape d'analyse des effets des changements sur la relation entre l'ontologie évoluée et les artefacts dépendants. De plus, l'étape de propagation des changements est essentiellement unidirectionnelle,

car elle vise uniquement la modification des ontologies dépendantes afin de préserver leur consistance structurelle avec l'ontologie de base et ne touche pas ainsi le référencement sémantique des ressources. (Djedidi et al., 2007) considèrent que l'étape d'évaluation de la qualité doit être intégrée dans le processus avant la mise en opération de l'ontologie après évolution. Concernant IMSE, Rogozan (Rogozan, 2008) considère que les auteurs de cette méthodologie proposent une approche pour supporter la gestion des versions d'une ontologie après son évolution et non pas pour supporter l'évolution proprement dite des ontologies. En effet, ils fournissent un modèle d'analyse de la relation entre les versions de l'ontologie, mais ne se préoccupent pas de la gestion de l'accès aux artefacts dépendants. Rogozan (Rogozan, 2008) définit l'évolution de l'ontologie comme le processus de changement de la version précédente V_N de l'ontologie à une nouvelle version V_{N+1}, afin de rendre compte des modifications dans le domaine de l'ontologie, dans sa conceptualisation ou dans son usage. Cette proposition d'évolution d'ontologie est décrite en neuf étapes. Comme les approches précédentes, son processus contient des étapes principales telles que l'identification et l'édition des changements, la vérification et l'implémentation des changements, l'analyse de la compatibilité entre des versions V_N et V_{N+1}. Cette méthodologie n'aborde pas le problème de la détection d'inconsistance dans l'évolution d'ontologie. Ce problème a été mentionné, mais traite que partiellement la résolution des inconsistances, et leurs résultats reste encore des propositions au niveau théorique.

(Djedidi, 2009) a défini une **méthodologie de gestion de changement Onto-Evoal** (Ontology Evolution-Evaluation) qui s'appuie sur une modélisation à l'aide de patrons et s'inspire largement de celle d'AIFB. Ces patrons spécifient des classes de changements, des classes d'incohérences et des classes d'alternatives de résolution. Sur la base de ces patrons et des liens entre eux, elle propose un processus automatisé permettant de conduire l'application des changements tout en maintenant la cohérence de l'ontologie évoluée. La méthodologie intègre aussi une activité d'évaluation basée sur un modèle de qualité d'ontologie. Ce modèle est employé pour guider la gestion des incohérences en évaluant l'impact des résolutions proposées sur la qualité de l'ontologie et ainsi choisir celle qui préserve la qualité de l'ontologie évoluée.

La recherche actuelle sur les méthodologies d'évolution ne se résume pas toutefois qu'à l'approche de l'AIFB et celle de l'IMSE, d'autres auteurs proposant des éléments méthodologiques à cet effet. (Heflin et Hendler, 2000) et (Heflin et al., 1999) développent SHOE, un langage fondé sur HTML, qui offre des primitives pour la gestion des versions multiples, en permettant d'associer à chaque version ontologique un identifiant unique ainsi qu'une balise « *Backward-Compatible_With* » spécifiant les versions compatibles ou rétrocompatibles. (Oliver et al., 1999) définissent un modèle conceptuel, CONCORDIA, pour la gestion des changements d'une terminologie médicale. Ce modèle associe à chaque concept un identifiant unique, les concepts pouvant ensuite être seulement retirés et non pas physiquement effacés. Ainsi, pour chaque concept, le modèle CONCORDIA est capable de garder la trace de tous ses concepts -parents ou enfants- retirés en utilisant leur identifiant. Enfin, (McGuinness, 2000) fournit des recommandations théoriques pour garantir un processus d'évolution de l'ontologie avec un minimum d'erreurs en utilisant des techniques de fusion pour mettre en évidence ces erreurs.

Du point de vue de l'évolution de l'ontologie dans un environnement multi-acteurs, (Pinto et Martins, 2002), et (Pinto et al., 2004) proposent un modèle pour gérer la négociation des

changements provenant des acteurs distants qui tentent de modifier une ontologie partagée. Dans ce modèle, chaque utilisateur modifie l'ontologie à partir de son poste individuel, en construisant ainsi sa propre ontologie locale. Un comité scientifique analyse ensuite les ontologies locales pour identifier les changements à introduire dans l'ontologie partagée. Une approche similaire est développée par (Sunagawa et al., 2003), la différence étant que les acteurs distants modifient l'ontologie partagée sans aucun intermédiaire. Ainsi, pour assurer la gestion de la dépendance entre les changements effectués par les différents acteurs, ceux-ci peuvent choisir d'accepter les changements des autres, de réduire la dépendance par rapport aux changements des autres ou de rompre la dépendance.

4 CONCLUSION

Les évolutions du web ont pour principale ambition de réduire le problème de l'interopérabilité entre les systèmes hétérogènes. Avec l'apparition du Web 2.0 où tout internaute est contributeur du web, il est nécessaire de résoudre un nouveau type d'hétérogénéité, l'hétérogénéité sémantique. Le web sémantique tente de répondre à cette hétérogénéité. Le web sémantique vise à favoriser l'appropriation du web par les utilisateurs et à réduire les incompréhensions des machines sur la signification de la connaissance. Pour cela, de nombreux standards ont été développés, tels que XML, RDF ou OWL. Des ontologies ont été proposées pour modéliser la sémantique d'un domaine, c'est-à-dire la signification des connaissances du domaine. Néanmoins, peu de réponses ont été apportées à la gestion du changement et au cycle de vie de la connaissance. L'un des principaux enjeux identifiés dans ce chapitre concerne l'évolution d'ontologie au cours du temps et plus spécifiquement l'évaluation de la qualité de l'ontologie au cours de ces changements. La plupart des approches et des outils décrits reposent sur une méthodologie qui suit un processus d'évolution d'ontologies bien déterminé. Un tel processus doit être complété par une phase d'évaluation de la qualité de l'ontologie avant la mise en application de la nouvelle ontologie. Aucun des travaux décrits ne présente une méthode pour évaluer la qualité de l'ontologie évoluée. On remarque aussi que presque tous les outils qui utilisent des techniques pour une détection automatique des changements effectuent seulement l'enrichissement de l'ontologie lors du processus d'évolution et ignorent par conséquent les opérations de modification et de suppression des éléments existants. De plus, les propagations des changements vers les artefacts dépendants (à savoir les ontologies, les référencements sémantiques, les applications) sont rarement abordées et mal traitées. En effet, il n'y a que les outils OntoAnalyseur (Rogozan, 2008) et CoSWEM (Luong et Dieng-Kuntz, 2007) qui assurent seulement une propagation vers les référencements sémantiques des ressources.

Pour l'ensemble de ces raisons, nous avons choisi de consacrer le chapitre suivant à la présentation des outils du model checking que nous souhaitons utiliser pour développer la première méthode de vérification de la cohérence et de la consistance de graphe sémantique lors de l'évolution de l'ontologie.

Chapitre 3

Les Méthodes Formelles et le Model checking

Résumé

Les concepteurs de logiciel savent par expériences que le code du logiciel n'est pas susceptible de fonctionner correctement après une, deux ou trois compilations. Parfois, cela prend un certain temps pour découvrir pourquoi un programme « correct » échoue.Pour répondre à ce besoin, de nombreuses méthodes utilisant des techniques de model checking ont été développées. Ce chapitre présente les méthodes formelles, basées sur des modèles formels. La première partie présentera les différentes méthodes formelles, leurs avantages et leurs inconvénients. Dans la deuxième partie, nous nous intéresserons au model checking qui est une technique utilisant la logique temporelle et un vérificateur de modèle appelé « model checker ». La troisième partie présentera les différents algorithmes de résolution utilisés par le model checking.

Plan

1 Les techniques de vérification formelles 61
1.1 Le Model checking 62
1.2 Techniques basées sur la simulation 63
1.3 Techniques basées sur le test 63
1.4 Techniques basées sur la preuve de théorème 64
2 Procédure du Model checking 65
2.1 Avantages et inconvénients du Model checking 67
2.2 Les model checkers qualitatifs 69
2.2.1 SPIN 69
2.2.2 NuSMV 70
2.2.3 ASTRAL 70
2.3 Les model checkers quantitatifs 70
2.3.1 DCVALID 71
2.3.2 KRONOS 71
2.3.3 UPPAAL 71
2.3.4 HyTech 72

2.3.5 CADP 72
2.4 Les algorithmes de résolution 73
2.4.1 Model checking LTL 74
2.4.2 Model checking CTL 77
3 L'explosion combinatoire 79
3.1 L'abstraction 80
3.2 L'interprétation abstraite 80
3.3 Le Model checking symbolique 80
3.4 Les BDD (Binary Diagram Decision) 81
3.5 Les méthodes utilisant les contraintes 81
3.6 La compression mémoire 81
3.7 Les méthodes à la volée 82
3.8 L'ordre partiel 83
4 Travaux Connexes 83
5 Conclusion 84

La vérification de logiciels devient incontournable avec le développement de la complexité, de la quantité et de l'importance économique des systèmes informatiques dans notre quotidien. Différentes techniques sont mises en œuvre pour la vérification de logiciels. Dans la conception de systèmes logiciels et matériels complexes, plus de temps et d'efforts sont consacrés à la vérification qu'à la construction. Des recherches ont été menées afin de réduire et alléger les efforts de vérifications, tout en augmentant leurs couvertures. Les méthodes formelles offrent un grand potentiel pour obtenir une intégration précoce de la vérification dans le processus de conception, fournir des techniques de vérification plus efficace et réduire le temps de vérification. Leur grand potentiel a conduit à une utilisation croissante par les ingénieurs pour la vérification des systèmes logiciels et matériels complexes. D'ailleurs, les méthodes formelles sont l'une des techniques de vérification hautement recommandées pour le développement de systèmes logiciels à sécurité critique.

La vérification formelle a généralement plus de succès pour la vérification de matériel que la vérification de logiciel (Baier et Katoen, 2008), et ce, principalement par les nombreux exemples de travaux concernant la vérification de la couche "matériel" (Browne et al., 1986). Enfin, les normes de haute qualité en conception "matériel", ainsi que l'utilisation de méthodes de conception assez classiques (composition, agrégation, etc…) dans son cycle de développement ont ouvert la voie à l'introduction en douceur de techniques telles que le model checking et la preuve de théorème. D'ailleurs, le rôle de vérifier l'exactitude de circuits dans le cadre du processus de conception, associé à l'utilisation de modèles à états finis ont été bénéfiques. La preuve de théorème et le model checking, ainsi que leurs combinaisons, ont trouvé leur place dans le processus de développement de matériel de compagnies comme IBM, Intel et Motorola. La preuve de théorème est surtout utilisée pour le contrôle des chemins de données, traitement du signal et des unités de l'arithmétique, alors que le model checking est généralement utilisé pour la logique de contrôle (l'une des principales sources de défauts de conception), les contrôleurs et les circuits combinatoires. Récemment, IBM (*International Business Machines Corporation*) a signalé l'utilisation bénéfique du model checking à plusieurs niveaux dans leur cycle de conception, y compris au niveau le plus abstrait de l'architecture (Schlipf et al., 1997) (Abarbanel-Vinov et al., 2001). Les expériences industrielles ont fourni la preuve que le temps d'exécution des algorithmes de model checking équivaut à ceux dédiés à la simulation aléatoire ; et de plus, le model checking est nettement supérieur en termes de couverture. La conception d'un adaptateur de bus mémoire chez IBM a montré que 24% des défauts détectés ont été trouvés avec le model checking, alors que 40% de ces erreurs n'auraient très probablement pas été trouvées par la simulation.

Le model checking a été appliqué avec succès à une branche particulière du Software, à savoir le développement de protocoles de communication. Dans de tels protocoles, des notions comme l'atomicité, le contrôle de concurrence et le non-déterminisme jouent un rôle crucial. Ces phénomènes peuvent être extrêmement bien manipulés par le model checker. Plusieurs défaillances sérieuses dans les protocoles de communication ont été trouvées en utilisant le model checking. L'exemple le plus frappant est peut-être le bug qui a été exposé au protocole de chiffrement *Needham-Schroeder* (Lowe, 1995) (Lowe, 1996) qui serait passé inaperçu pendant plus de 17 ans.

Par opposition à la conception matérielle, le génie logiciel n'a pas exposé la discipline de « connaissance de vérification » dans le processus de conception. La vérification formelle de

programmes software a commencé dans les années soixante avec les pionniers Floyd (Floyd, 1967) et Hoare (Hoare, 1969). Malgré cet intérêt précoce pour la correction de logiciels, ces techniques de vérification rigoureuse ont été principalement utilisées par les universitaires. Bien que l'approche rigide de vérification à l'aide des axiomes et règles de preuve ne soit jamais devenue populaire parmi le génie logiciel, des concepts tels que les assertions, et, plus important encore, les pré et post conditions ont trouvé leur rôle dans les méthodes de génie logiciel modernes. Dans le populaire « conception par contrat », la philosophie du génie logiciel ainsi que des pré et post conditions constituent le cahier des charges (le contrat) pour lequel le logiciel en cours de développement devrait se conformer.

L'une des principales raisons de l'attitude conservatrice des ingénieurs software à l'égard du model checking a été la nécessité de construire un modèle de leurs logiciels qui se prête à sa vérification. Cet obstacle a récemment conduit à un regain d'intérêt par les grandes entreprises comme Microsoft, La NASA et Compaq pour générer automatiquement des modèles compacts à partir de programmes écrits en langage de programmation tels que C, C++, Java ou similaire. Les premières expériences avec ces techniques sont très prometteuses. Il est prévu que les modèles techniques de contrôle soient rapidement adoptés sur une plus grande échelle par des ingénieurs software dans un futur proche.

Le graphe ci-dessous a été réalisé par (Liggesmeyer et al., 1998), dans le but d'estimer les coûts de réparation d'une faille logicielle lors de la maintenance. Ces coûts sont environ 500 fois supérieurs à ceux d'une correction dans une phase de conception initiale (voir Figure 11**)**. La vérification du système devrait donc prendre place au début de l'étape du processus de conception.

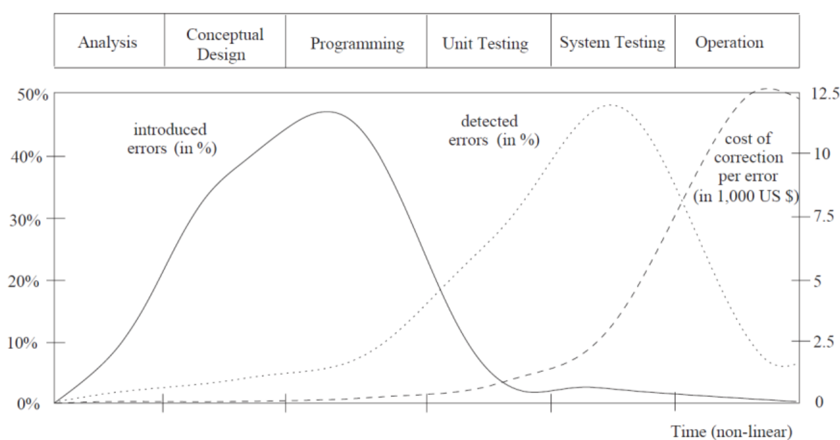


Figure 11. Le coût d'introduction, de détection et de réparation dans le cycle de vie logiciel.

(Liggesmeyer et al., 1998) constate que 50% des défauts sont introduits lors de la programmation, la phase du codage. Alors que seulement 15% des erreurs sont détectées dans les phases initiales de conception, et la plupart des erreurs sont constatées lors de la phase de tests. Les tests unitaires sont destinés à la découverte de défauts dans les modules logiciels individuels qui composent le système. Sans utilisation de ces méthodes, une densité de défauts d'environ 20 défauts par 1000 lignes de code (non commentées) est typique. Cela a été réduit à environ 6 défauts pour 1000 lignes de code en testant le système dès le début. Au lancement d'une nouvelle version du logiciel, le nombre de défauts généralement acceptés du logiciel est d'environ un défaut pour 1000 lignes de code. Les erreurs sont généralement concentrées dans quelques modules du logiciel et se

produisent souvent lors de l'interfaçage de modules. La réparation des erreurs qui sont détectées avant le test peut être bénéfique économiquement. Le coût de la réparation augmente de manière significative d'environ 1000 $ (par réparation d'erreur) dans les tests unitaires à un maximum d'environ 12.500 $ (Liggesmeyer et al., 1998) lorsque le défaut est démontré au cours de l'exploitation du système seulement. Il est d'une importance vitale de rechercher des techniques qui permettent de trouver des défauts le plus tôt possible dans le processus de conception de logiciels : les coûts pour les réparer sont nettement inférieurs, et leur influence sur le reste de la conception est moins importante.

1 LES TECHNIQUES DE VERIFICATION FORMELLES

Au cours de la dernière décennie, la recherche dans les méthodes formelles a conduit au développement de certaines techniques de vérification très prometteuses qui facilitent la détection précoce des défauts. Ces techniques sont accompagnées d'outils puissants qui peuvent être utilisés pour automatiser les diverses étapes de vérifications. Les enquêtes ont montré que les procédures de vérifications formelles auraient révélé les défauts exposés dans le lanceur Ariane-5 (lanceur développé pour placer des satellites sur orbite géostationnaire), la sonde Mars Pathfinder (vaisseau intégrant un robot pour l'exploration de la planète mars), le processeur Intel Pentium 2 et la machine de radiothérapie Therac-25 (Baier et Katoen, 2008), par exemple.

Deux types de méthodes de vérifications formelles peuvent être distingués : le modèle déductif et les méthodes basées sur le modèle. Avec les méthodes déductives, l'exactitude des systèmes est déterminée par les propriétés dans une théorie mathématique. Ces propriétés sont éprouvées avec la plus grande précision possible en utilisant des outils tels que le théorème et des vérificateurs de preuves. Les techniques basées sur le modèle décrivent le comportement du système d'une façon mathématique précise et sans ambiguïté. Il s'avère que – préalablement à toute forme de vérification – la modélisation précise des systèmes mène à la découverte d'inachèvements, d'ambiguïtés et d'incohérences dans les spécifications du système informel. Les problèmes des modèles sont accompagnés par des algorithmes qui explorent systématiquement tous les états du modèle du système. Ceci fournit la base pour toute une gamme de techniques de vérification allant de l'exploration exhaustive (model checking) à des expériences avec un ensemble restrictif de scénarios dans le modèle (simulation), ou dans la réalité (les tests). Le test n'est pas un moyen de vérifier formellement un logiciel. Le programme est soumis à un jeu de données d'entrées correctement construit pour couvrir le plus grand nombre de cas possibles, ou les cas les plus représentatifs en fonction du style de bug que nous souhaitons résoudre, et les valeurs de retour du logiciel sont examinées. Les deux autres méthodes sont des méthodes formelles de vérifications : la preuve de programme et le model checking. La preuve de programme s'attache à démontrer, par exemple, l'invariance réelle d'invariants supposés en se servant directement du code source du programme. Le model checking, lui, demande à ce que nous transformions le programme ou le code source en modèle de comportement, généralement simplifié. Les propriétés que nous souhaitons vérifier sont alors énoncées dans un certain langage (Logique Temporelle) et un programme « vérificateur (model checker) » répond si la propriété est vérifiée par le modèle ou non. Cette section présente un aperçu des principales techniques de vérification formelles.

1.1 LE MODEL CHECKING

Le model checking (Schnoebelen et al., 1999) (Clarke et al., 1999) (Bardin, 2008) est une technique de vérification qui explore tous les états possibles du système. Similaire à un programme d'échecs qui vérifie tous les mouvements possibles, le model checker, l'outil qui accomplit le model checking, examine tous les scénarios possibles du système d'une manière systématique. De cette manière, nous pouvons montrer que le modèle d'un système satisfait vraiment une certaine propriété. Il y a un vrai défi à examiner les espaces d'états les plus larges possible qui peuvent être traités par des moyens actuels, par exemple, des processeurs et des mémoires. Les model checkers peuvent gérer des espaces d'états d'environ 10 exposant 9 états. En utilisant des algorithmes plus performants et des structures de données adaptées, des espaces d'états plus larges (10 exposant 20 jusqu'à 10 exposant 476 états) peuvent être gérés pour des problèmes spécifiques. Même les erreurs subtiles qui ne restent pas encore découvertes en utilisant l'émulation, le test et la simulation peuvent être potentiellement révélées en utilisant la technique du model checking.

Les propriétés qui peuvent être vérifiées en utilisant un model checker sont d'une nature qualitative : le résultat généré est-il correct ? Le système peut-il atteindre une situation de blocage ?, par exemple, quand deux programmes concurrents s'attendent mutuellement, peuvent-ils arrêter le système entier ? Mais aussi des propriétés du moment peuvent être vérifiées : peut-il se produire un blocage du système après une heure de réinitialisation? Ou, une réponse peut-elle toujours être reçue après 8 minutes ? Le model checking demande une déclaration précise et sans ambiguïté des propriétés qui doivent être examinées. Car en faisant un modèle d'un système exact, cette étape conduit souvent à la découverte de plusieurs ambiguïtés et d'incohérences dans la documentation informelle. Le modèle du système est habituellement généré automatiquement par la description d'un modèle qui est spécifié dans un dialecte adéquat des langages de programmation, comme C ou Java ou des langages de description hardware, tels que Verilog ou VHDL (*Very high speed integrated circuits Hardware Description Language*). On doit noter que la spécification du système prescrit ce que le système doit faire, et non ce qu'il ne doit pas faire ; tandis que la description du modèle aborde la façon dont le système doit se comporter. Le model checker examine tous les états relevants du système pour vérifier s'il satisfait la propriété désirée. Si nous rencontrons un état dont la propriété n'est pas vérifiée, le model checker fournit un contre-exemple indiquant à l'utilisateur le moyen d'atteindre l'état indésirable. Le contre-exemple décrit un chemin d'exécution conduisant de l'état initial du système à l'état ignorant la propriété vérifiée. À l'aide d'un simulateur, l'utilisateur peut reproduire le scénario qui a généré l'erreur, nous obtenons, de cette manière, des informations utiles de correction et il est alors possible d'adapter le modèle (ou la propriété) en tant que tel.

Le model checking a été appliqué avec succès à plusieurs systèmes et à leurs applications. Des blocages ont été détectés dans les systèmes de réservation en ligne des compagnies aériennes, des protocoles modernes d'e-commerce ont été vérifiés, et plusieurs études des standards internationaux IEEE pour la communication interne des applications domestiques ont conduit à des modifications significatives des spécifications du système. Ainsi, cinq erreurs non découvertes ont été identifiées dans un module d'exécution du contrôleur *Deep Space 1* (une sonde spatiale), montrant un défaut majeur de conception. Un même défaut, resté invisible pendant la phase de tests et ayant causé un blocage lors un vol expérimental à 96 millions de kilomètres au-dessus de la Terre, a été découvert par le model checking. Aux Pays-Bas, le model checking a révélé plusieurs défauts sérieux

de conception dans le logiciel de contrôles des digues protégeant le port principal de Rotterdam contre les inondations.

1.2 TECHNIQUES BASEES SUR LA SIMULATION

En pratique, l'une des techniques de vérification la plus connue et utilisée est la simulation (Monin, 2000) (Baier et Katoen, 2008). Le simulateur permet à l'utilisateur d'étudier le comportement du système. Ceci passe par la détermination, sur la base du modèle du système, des réactions que doit avoir le système vis-à-vis de certains scénarios spécifiques. Ces scénarios sont fournis par l'utilisateur ou générés par des outils tels que les générateurs aléatoires de scénarios.

La simulation est généralement utile pour une première évaluation rapide de la qualité du prototype (l'étape de conception). Elle est cependant moins adaptée à détecter les erreurs subtiles, car pour le simulateur, il est impossible de générer tous les scénarios possibles du système, et encore moins de tous les simuler. Dans la pratique, seul un petit sous-ensemble de tous les scénarios possibles est effectivement examiné. Par conséquent, il existe un risque réaliste que les défauts subtils restent cachés. Les scénarios inexplorés pourraient révéler l'erreur fatale.

D'ailleurs, il est difficile de quantifier le degré d'exactitude du système lors de l'examen d'un nombre restreint de scénarios. De même, les mesures quantitatives du nombre d'erreurs laissées dans le système sont difficiles à obtenir ; tout comme des indications sur la probabilité que de telles erreurs soient découvertes lorsque le système est en fonctionnement. Dans la pratique, cela signifie souvent que le critère pour arrêter la simulation est simplement au moment où le projet est à court de capitaux.

1.3 TECHNIQUES BASEES SUR LE TEST

Tandis que la simulation et le model checking sont basés sur une description du modèle où tous les états possibles du système peuvent être générés, la technique de vérification basée sur le test (Myers, 1979) (Ammann et Offutt, 2008) (Mathur, 2008) est applicable dans le cas où il est difficile, voire impossible, d'obtenir un modèle du système à vérifier. Avec le test, des séquences d'actions, représentant différents scénarios d'exécutions possibles, permettent de vérifier une réaction spécifique.

Un paramètre important du test est la mesure à laquelle l'accès à l'état interne du système testé peut être obtenu. Deux types de tests peuvent être effectués sur le système. Le test à boite blanche peut accéder totalement à la structure interne d'une implémentation, tandis que dans le test à boite noire, la structure interne est complètement cachée. En pratique, des scénarios intermédiaires sont souvent rencontrés, appelés test à boite grise (gray box). L'avantage principal du test est sa large applicabilité, en particulier pour les produits finaux et il n'est pas restreint seulement aux modèles. Le désavantage est comparable à la simulation, car le test complet est pratiquement impossible. Comme la simulation, le test peut montrer la présence d'erreurs, et non leur absence.

Les tests sont plutôt improvisés et pas très systématiques. Comme résultat, le test est une activité très élaborée, prédisposant aux erreurs, et difficilement gérable. Similaire au model checking, le point de départ de la méthode du test sur des modèles est la spécification précise du système, sans ambiguïté. Avec les méthodes du test traditionnelles, une telle base est souvent absente. Basés sur cette spécification formelle, les algorithmes de la génération du test génèrent des tests

probablement valides, et testent ce qu'il faut tester et pas plus. Les outils du test implémentent ces algorithmes, fournissent une génération de test automatique, plus rapide, et moins source d'erreurs.

Le test de régression implique la vérification du comportement correct d'une version modifiée d'un système existant. Cela implique typiquement l'adaptation, la sélection et la répétition des tests existants. Dans la méthode de test basée sur le modèle, une petite modification du système demande seulement une adaptation de son modèle, et ensuite, un nouveau test peut être automatiquement généré.

En pratique, le modèle basé sur le test a été implémenté dans plusieurs outils logiciels et a démontré sa puissance dans diverses études (logiciel d'application par exemple le e-business, les compilateurs et les systèmes d'exploitation). Pour plusieurs systèmes, comme les systèmes embarqués contrôlant l'échange d'informations entre certaines télévisions et les magnétoscopes, nous avons trouvé des erreurs qui n'ont pas été découvertes avec les techniques de tests conventionnelles.

1.4 TECHNIQUES BASEES SUR LA PREUVE DE THEOREME

La preuve de théorème, (Cook, 1971) (Dingel et Filkorn, 1995) nécessite que le système soit spécifié sous forme d'une théorie mathématique, ou devrait être transformé en une telle forme. En utilisant un ensemble d'axiomes (le théorème de base), un démonstrateur (le logiciel) tente de construire : soit une preuve de théorème en générant les étapes de preuves intermédiaires ; soit de réfuter les axiomes énoncés. Les axiomes sont intégrés ou fournis par l'utilisateur. Les démonstrateurs sont également appelés assistants de preuves. La demande générale de prouver des théorèmes d'un type assez général et l'utilisation de logiques indécidables exige certaines interactions avec l'utilisateur. Il existe différentes variantes : hautement automatisée, les programmes d'assistance de preuves, et interactifs avec des capacités spéciales.

Les vérificateurs de preuves sont des assistants de preuves automatisées qui nécessitent une interaction limitée avec l'utilisateur. Principalement, le vérificateur vérifie si un utilisateur fournit une suggestion de preuve valide ou non. La capacité du vérificateur de preuves à produire les mesures de preuves intermédiaires de façon automatique est plutôt limitée.

Afin de réduire le temps de recherches à démontrer le théorème, l'interaction avec l'utilisateur prend place. L'utilisateur peut aider à trouver la meilleure stratégie à mener sur une preuve. Habituellement, les assistants de preuves interactifs aident à donner une preuve en gardant la trace des choses restantes à faire et en fournissant des conseils sur la façon dont ces théorèmes restants peuvent être prouvés. Le degré d'interactions avec l'utilisateur est généralement assez élevé. Cela couvre non seulement le contenu du théorème, mais aussi la façon dont il est utilisé. En outre, l'utilisation de démonstrateur ou de vérificateur de preuves exige une plus grande habitude des utilisateurs. En général, les êtres humains évitent des petites parties de preuves (triviales ou analogues à), alors que l'assistant de preuves exige explicitement ces étapes.

Le principal avantage de la preuve de théorèmes est qu'il peut agir avec des espaces d'états infinis et peut vérifier la validité des propriétés pour des valeurs de paramètres arbitraires. Par contre, l'inconvénient principal de la preuve de théorèmes est la lenteur du processus de vérifications, le risque d'erreurs et le temps utilisé pour guider la preuve. Aussi, la logique

mathématique utilisée par l'assistant de preuves exige un degré assez élevé d'expertise des utilisateurs.

Le choix de la technique de vérification. Dans cette section, nous savons que les méthodes basées sur des modèles de systèmes sont plus pertinentes et plus précises pour la détection d'erreurs que les méthodes se lançant directement à la recherche de bugs sur les systèmes.

La simulation est l'une des techniques les plus utilisées actuellement, mais son grand désavantage est que le simulateur ne peut générer tous les scénarios possibles et des erreurs fatales peuvent ne jamais être détectées. La preuve de théorème demande une grande interaction avec l'utilisateur ce qui rendrait la technique moins automatique. Le test est une technique très puissante et les tests traditionnels sont moins adaptés que les tests sur les modèles de systèmes. Le test reste toujours limité par la découverte des erreurs et non de leurs absences.

La technique du model checking comme méthode formelle est une méthode très performante et très utilisée dans le domaine des protocoles de communications dans les réseaux informatiques. Cette technique correspond au mieux à notre domaine, car elle pourra révéler des erreurs non détectées par les autres méthodes formelles comme le test et la simulation. La technique du model checking sera déployée pour la vérification de la pertinence des graphes sémantiques.

2 PROCEDURE DU MODEL CHECKING

Le model checking (Schnoebelen et al., 1999) examine tous les états du système afin de vérifier s'ils répondent à la propriété désirée. Le model checker donne un contre-exemple indiquant de quelle façon le modèle peut violer une propriété. Avec l'aide d'un simulateur, l'utilisateur peut trouver l'erreur et d'adapter le modèle ou la propriété pour empêcher la violation de cette dernière.

Le model checking utilise la logique temporelle (Emerson, 1990) (Stirling, 1992) pour décrire les propriétés vérifiant le modèle du système. Les concepts de la logique temporelle utilisée pour la première fois par (Pnueli, 1977) dans la spécification des propriétés formelles sont assez faciles à utiliser. Les opérateurs sont très proches en termes de langage naturel. La formalisation en logique temporelle est assez simple, bien que sous cette apparence simple, elle nécessite une expertise significative.

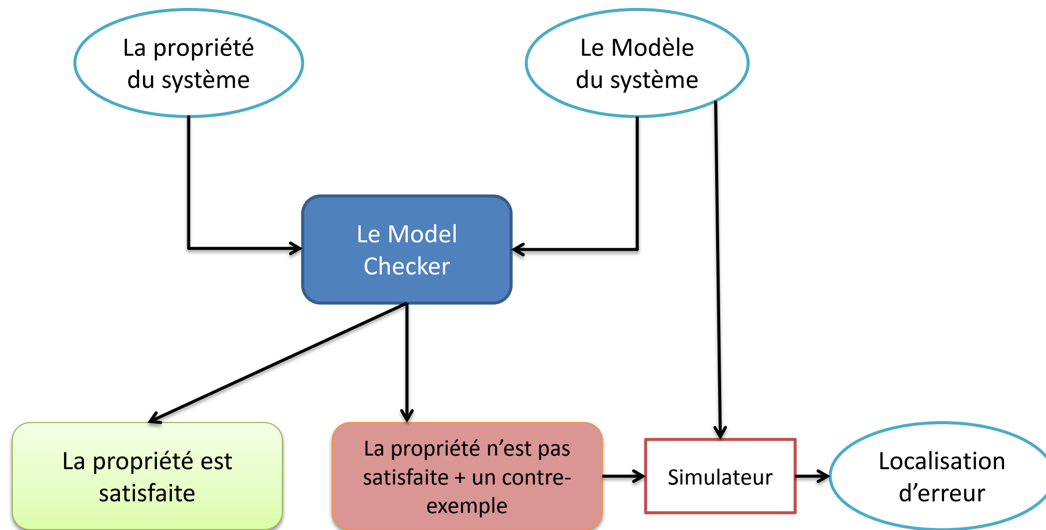


Figure 12. L'approche du model checking.

Le model checking passe par trois différentes étapes : la phase de modélisation, la phase d'exécution et la phase d'analyse (Manna et Pnueli, 1995) et dont l'articulation peut se voir sur la Figure 12.

La phase de modélisation. La modélisation du système est une étape difficile du model checking. Elle est d'ailleurs souvent cruciale pour la pertinence des résultats obtenus par la suite. Il n'existe pas de méthode universelle pour modéliser un système, cela nécessite à la fois une bonne connaissance du système à analyser ainsi que des modèles mathématiques ou informatiques pouvant être utilisés.

- **Les diagrammes de décision binaires** (BDD) ont été introduits par Bryant (Bryant, 1986) pour représenter et surtout manipuler efficacement des expressions booléennes. Ce sont de structures de données pour la logique propositionnelle avec quantificateurs. Un BDD est un arbre binaire basé sur le principe de Shannon de la décomposition d'une fonction booléenne
- **Un automate** est un modèle de machine abstraite permettant de simuler le déroulement d'un calcul. Les automates sont généralement représentés graphiquement sous la forme d'un graphe orienté étiqueté, ce qui permet de clarifier la compréhension que l'on peut avoir du comportement du système. Les états sont souvent représentés par des cercles et les transitions par des flèches.
- **Les Réseaux de Pétri** ont été créés en 1962 par Carl Adam Pétri. Leur formalisme est un outil fondamental permettant l'étude des systèmes dynamiques et discrets. Il s'agit, comme pour les automates, d'une représentation mathématique permettant la modélisation d'un système. D'une certaine façon, les Réseaux de Pétri sont des automates particuliers.

Le model checking a besoin du modèle représentant le système en cours de vérification et d'une caractérisation formelle de la propriété à vérifier. Les modèles décrivent le comportement des systèmes d'une façon précise et sans ambiguïté. Ils sont pour la plupart exprimés en utilisant des automates finis, constitués d'un ensemble fini d'états et d'un ensemble de transitions. Les états comprennent des informations sur les valeurs courantes des variables. Les transitions décrivent l'évolution du système d'un état à un autre.

Pour effectuer une vérification aussi rigoureuse que possible, les propriétés doivent être décrites de manière précise et sans ambiguïté. Cela se fait habituellement en utilisant un langage de spécification de propriétés. La logique temporelle est le langage de spécification de propriétés, une forme de logique modale appropriée pour définir les propriétés pertinentes des systèmes.

La logique temporelle est essentiellement une extension de la logique propositionnelle classique avec des opérateurs faisant référence aux comportements des systèmes au fil du temps. Elle permet la spécification d'un grand nombre de propriétés du système concerné telles que l'exactitude fonctionnelle (Le système fait-il ce qu'il est censé faire ?), l'accessibilité (est-il possible de se retrouver dans un état de blocage ?), la sécurité (quelque chose de mauvais n'arrive jamais), la vivacité (quelque chose de bon finira par arriver), l'équité (sous certaines conditions, un événement peut-il se répéter indéfiniment ?), et des propriétés en temps réel (le système agit-il à temps ?).

La phase d'exécution. Le model checker doit d'abord être initialisé par un réglage approprié des différentes options et les directives pouvant être utilisées pour effectuer la vérification exhaustive. Il s'agit essentiellement d'une approche purement algorithmique dans laquelle la validité de la propriété en cours d'examen est vérifiée dans tous les états du modèle de système.

La phase d'analyse. Il existe essentiellement trois résultats possibles : la propriété spécifiée est soit valide dans le modèle donné ou non, ou le modèle se révèle être trop volumineux pour tenir dans les limites physiques de la mémoire de l'ordinateur. Le model checker examine tous les états du modèle représentant le système, pour vérifier si elles répondent à la propriété désirée. Si un état rencontré viole la propriété en cours d'examen, le model checker fournit un contre-exemple indiquant la façon dont le modèle pourrait atteindre l'état non désiré. Le contre-exemple décrit un chemin d'exécution qui mène de l'état initial du système à un état violant la propriété en cours de vérification. Avec l'aide d'un simulateur, l'utilisateur peut réexécuter le scénario violant la propriété et de cette manière nous pourrons obtenir des informations de débogage utiles, et adapter le modèle (ou la propriété) en conséquence. Si la propriété satisfait le modèle, le model checker va à la suivante, s'il y'en a d'autres, en utilisant le même processus de vérification décrit ci-dessus.

Si le modèle est trop grand pour être manipulé, il existe de nombreuses façons de procéder. Il y a des représentations de l'espace d'états en utilisant des techniques symboliques telles que les diagrammes de décisions binaires ou de réductions d'ordre partiel. Ces techniques permettent ainsi de réduire la taille en mémoire du modèle le rendant alors utilisable sur un ordinateur donné.

2.1 AVANTAGES ET INCONVENIENTS DU MODEL CHECKING

Les avantages du model checking :

- C'est une approche de vérification générale applicable à un large éventail d'applications telles que les systèmes embarqués, le génie logiciel, et la conception matérielle.
- Elle supporte une vérification partielle c'est-à-dire les propriétés peuvent être vérifiée individuellement, permettant ainsi de se concentrer sur les propriétés essentielles dans un premier temps. Aucune nécessité de spécification complète n'est nécessaire.
- Il n'est pas vulnérable à la probabilité avec laquelle une erreur est exposée, ce qui contraste avec les essais et la simulation qui visent à retrouver les défauts les plus probables.

- Il fournit des informations de diagnostic dans le cas où une propriété est invalide, ce qui est très utile pour le débogage.
- C'est une technologie potentielle "Push-button" ; l'utilisation d'un model checker ne nécessite pas un degré élevé d'interaction entre utilisateurs ou d'expertise.
- Il bénéficie d'un intérêt de plus en plus prononcé par les industriels : plusieurs sociétés de matériels ont commencé leurs vérifications internes en laboratoire ; des offres d'emploi avec les compétences requises dans la vérification de modèle apparaissent souvent ; et des modèles de vérificateurs commerciaux deviennent disponibles.
- Il peut être facilement intégré dans les cycles de développement existants : sa courbe d'apprentissage est relativement rapide, et les études empiriques indiquent qu'il peut conduire à des temps de développement plus courts.
- Il est basé sur les éléments des algorithmes de la théorie des graphes, les structures de données et la logique.
- Les logiques temporelles peuvent facilement exprimer un grand nombre des propriétés qui sont nécessaires pour le raisonnement sur les systèmes concurrents.
- Si la spécification n'est pas satisfaite, le model checker produira un contre-exemple (trace d'exécution) qui montre pourquoi la spécification ne tient pas. Les contre-exemples sont très précieux pour le débogage des systèmes complexes. Certaines personnes utilisent le model checking uniquement pour cette fonctionnalité.
- Dans la pratique, le model checker est rapide en comparaison des autres méthodes rigoureuses telles que l'utilisation d'un vérificateur de preuves (*proof checker*), qui peut nécessiter des mois de travail pour un utilisateur en mode interactif.
- Le processus de vérification est automatique. L'utilisateur du model checker n'a pas besoin de construire une preuve de corrections. En principe, l'utilisateur doit entrer une description d'un circuit ou d'un programme et la spécification qui doit être vérifiée.
- Le model checking peut être utilisé lors de la conception d'un système complexe. L'utilisateur n'a pas besoin d'attendre la fin de la phase de conception.

Les inconvénients du model checking :

- Il est surtout approprié pour le contrôle intensif d'applications et moins adapté pour des applications ayant des données variant généralement sur des domaines infinis.
- Son applicabilité est soumise aux problèmes de décidabilité, pour les systèmes à états infinis, ou le raisonnement à propos des types de données abstraites (qui nécessite des logiques indécidables ou semi décidables), alors les algorithmes du model checking ne sont pas, en général, décidables.
- Il vérifie le modèle du système, et non le système actuel (produit ou prototype) lui-même, tout résultat obtenu est donc aussi bon que le modèle du système. Les techniques telles que les essais sont nécessaires pour trouver les défauts de fabrication (pour le matériel) ou des erreurs de codage (pour le logiciel).
- Il vérifie seulement l'exigence (besoin) déclarée c'est-à-dire qu'il n'y a aucune garantie d'exhaustivité (complétude). La validité des propriétés non vérifiées ne peut pas être jugée.
- Il souffre du problème de l'explosion combinatoire (explosion d'espace d'états), le nombre d'états nécessaires pour modéliser le système avec précision peut facilement dépasser la

quantité de mémoire disponible. Malgré le développement de plusieurs méthodes très efficaces pour lutter contre ce problème, les modèles de systèmes réalistes peuvent encore être trop grands pour tenir en mémoire.

- Son utilisation nécessite une certaine expertise dans la recherche d'abstractions appropriées pour obtenir des modèles de systèmes réduits et aux propriétés de l'état dans la logique du formalisme utilisé.
- Il n'est pas garanti pour donner des résultats corrects : comme tout outil, un vérificateur de modèle peut comporter des défauts logiciels.
- Il ne permet pas de vérifier les généralisations : en général, la vérification de systèmes avec un nombre arbitraire de composants, ou des systèmes paramétrés ne peut pas être traitée. Le model checking peut, cependant, suggérer des résultats pour des paramètres arbitraires qui peuvent être vérifiés à l'aide d'assistant de preuves.

Les différents types du model checkers. Il existe deux types de model checkers : les model checkers qualitatifs et quantitatifs.

2.2 LES MODEL CHECKERS QUALITATIFS

Les model checkers qualitatifs sont des outils de vérifications qui se basent sur des logiques qualitatives lors de la description du comportement et des propriétés à vérifier d'un système temps réel. Nous citons à titre d'exemple les outils tels que NuSMV (McMillan, 1993), ASTRAL (Ghezzi et Kemmerer, 1991), et SPIN (Holzmann, 2003).

Ces logiques ont des puissances d'expression limitées à l'aspect qualitatif où nous ne connaissons ni à quel instant apparaît un état, ni sa durée, ni la durée qui le sépare d'un autre état. En fait, le temps est abstrait et il est concrétisé par l'occurrence d'évènements et la succession des états, qui sont matérialisées par les opérateurs d'occurrence.

2.2.1 SPIN

SPIN (http://spinroot.com) est un model checker développé par Gerard Holzmann (Holzmann, 2003) au sein des laboratoires Bell de Murray Hill dans le New Jersey. Il permet à la fois la simulation et la vérification. Il est particulièrement adapté à l'étude des systèmes distribués. Nous nous concentrons sur la partie vérification en laissant de côté la partie simulation se trouvant hors du cadre de nos recherches.

SPIN repose sur le principe des automates communicants qui sont représentés par un réseau de plusieurs automates ayant la particularité de pouvoir communiquer entre eux, c'est-à-dire qu'ils peuvent partager des variables ou encore s'envoyer des messages. Ces automates sont particulièrement adaptés à la modélisation de systèmes de contrôles ou de protocoles de communications. Une fois le modèle représenté sous forme d'un réseau d'automates communicants, il faut regrouper (ou « synchroniser ») ces automates afin de représenter le comportement global du système.

SPIN est un outil qui peut vérifier l'exactitude des logiciels et son élaboration remonte dans les environs des années 80, avec une version gratuite. Il est considéré comme l'un des pionniers, le plus puissant pour la vérification de logiciel. SPIN a été utilisé dans plusieurs domaines, la vérification des

logiciels des systèmes d'exploitation jusqu'aux protocoles de communication pour les systèmes de signalisation ferroviaires.

SPIN utilise le langage PROMELA pour décrire les modèles. PROMELA est, en effet, un langage de programmation très simple, la déclaration des variables et les expressions dans PROMELA sont écrites en utilisant la syntaxe du langage C.

2.2.2 NuSMV

Le model checker NuSMV (Cimatti et al., 2000) (Cimatti et al., 2002) est une amélioration de SMV (McMillan, 1993) (McMillan, 1992). SMV est le premier outil, basé sur le model checking symbolique, qui a permis de vérifier de façon exhaustive des systèmes de taille très élevée. Il a été développé par K. L. McMillan durant sa thèse (McMillan, 1993). Il prend en entrée un modèle représenté par un réseau d'automates sous forme de structures de Kripke. Un langage propre au logiciel est utilisé pour décrire ce modèle (ce langage est présenté par la suite dans cette section). Une fois le modèle décrit complètement, les propriétés à vérifier sont ajoutées à la description et le modèle peut alors être soumis au model checker.

Le langage SMV permet à la fois de modéliser des systèmes synchrones et asynchrones. Un fichier SMV est organisé en modules de façon hiérarchique. Chaque module décrit une machine à états finis et se compose en deux parties. La première sert à déclarer les variables permettant de représenter les états du système. La deuxième, quant à elle, sert à modéliser les transitions possibles. Pour cela, il faut définir les successeurs possibles des variables déclarées dans la première partie. Une fois les modules décrits, il reste à formuler les spécifications en logique temporelle. Une description plus complète est présentée dans la thèse de K. L. McMillan (McMillan, 1993).

2.2.3 ASTRAL

ASTRAL (Ghezzi et Kemmerer, 1991) est un model checker symbolique comme le model checker NuSMV. Le model checker ASTRAL (*Software development Environment*) est un ensemble intégré d'outils de conception et d'analyse basés sur le framework formel d'ASTRAL, qui comprend un éditeur de syntaxe dirigée, un processeur de spécification, un générateur de conditions, de vérification, et un kit de navigation. Le vérificateur de modèle ASTRAL vérifie la satisfiabilité des exigences essentielles d'une spécification ASTRAL en énumérant toutes les pistes possibles de transitions dans un temps donné. Il invite l'utilisateur à l'exécution des messages chaque fois qu'une erreur est rencontrée dans le déroulement d'une spécification (Dang et Kemmerer, 1999).

ASTRAL est un langage de haut niveau pour la spécification formelle des systèmes en temps réel. Il est doté de mécanismes structurants qui permettent de construire des spécifications modulaires de systèmes complexes avec la superposition (Coen-Porisini et al., 1997). Un système en temps réel est modélisé par un ensemble de spécifications de processus et une seule spécification globale. Chaque spécification de processus consiste en une séquence de niveaux, chaque niveau est une vue abstraite du processus étant spécifié.

2.3 Les model checkers quantitatifs

Les model checkers quantitatifs se basent sur des automates temporisés ou hybrides rajoutant des structures permettant de calculer le temps. L'avantage des model checkers quantitatifs est qu'ils

permettent la quantification du temps à l'aide des variables de type horloge. Ils possèdent par conséquent un plus grand pouvoir d'expression.

2.3.1 DCVALID

DCValid a été conçu et programmé par Paritosh K. Pandya au *Tata Institute of Fundamental Research*, *Mumbai*, en Inde (Pandya, 2001). DCVALID est un programme pour vérifier la validité des formules d'intervalle de temps (Duration Calculus) pour les systèmes temps réel. Il peut être utilisé comme un outil pour visualiser les spécifications DC et pour vérifier leur cohérence. Il peut également être utilisé en conjonction avec d'autres outils pour vérifier les propriétés du modèle DC de systèmes. Actuellement, les systèmes écrits en SMV, Verilog et PROMELA[10] (SPIN) sont pris en charge par DCValid.

DCValid est basé sur une procédure de décision d'automates théorique. Pour chaque formule D, un automate à états finis A(D) est construit, précisément en acceptant les séquences d'états finis satisfaisant D. L'automate peut être utilisé pour trouver des modèles et des contres exemples, ou comme un observateur synchrone (ou moniteur) pour le model checking.

2.3.2 KRONOS

KRONOS (Yovine, 1997) est un ensemble d'outils développés à Grenoble, au sein du laboratoire VERIMAG. KRONOS permet l'analyse d'automates temporisés et est disponible sur Internet. C'est un outil de vérification utilisant la technique de model checking symbolique (Burch et al., 1992) (Bryant, 1992) et c'est un environnement offrant à l'utilisateur la possibilité d'assister la vérification. KRONOS permet de vérifier si un système temps réel modélisé par un automate temporisé, sous forme textuelle, satisfait une propriété temporisée décrite par des formules TCTL (Alur et al., 1993).

Il est capable de vérifier des propriétés d'atteignabilité, mais également des propriétés de vivacité. En outre, c'est un outil en ligne de commande facilement invocable par d'autres programmes.

2.3.3 UPPAAL

UPPAAL est issu d'une collaboration entre l'équipe BRICS (Aalbord University, au Danemark) et le Département of Computing Systems (Uppsala University, en Suède). L'outil est disponible, sous conditions, sur Internet (http://www.uppaal.com/).

UPPAAL (Amnell et al., 2001) (Amnell et al., 2012) est un ensemble d'outils permettant de modéliser, simuler et vérifier les systèmes temps réel. Il est particulièrement destiné aux systèmes qui peuvent être modélisés comme un ensemble de processus non déterministes avec des contrôles de structures finies et d'horloges à valeurs réelles, chaque processus pouvant communiquer par des actions de synchronisations (par exemple, les contrôleurs temps réels, les protocoles de communications, ...).

Il est donc principalement basé sur le modèle des automates temporisés. Il a l'avantage de posséder une version texte et graphique, cette dernière permettant même la simulation pas-à-pas de

[10] PROcess Meta LAnguage

systèmes dont le nombre d'états est trop grand pour appliquer les algorithmes de vérifications classiques.

UPPAAL utilise des méthodes très similaires à celles de Kronos. Par contre, il fournit une interface graphique, qui en plus de lancer les algorithmes de vérifications, permet également de simuler pas à pas des systèmes modélisés.

2.3.4 HYTECH

HYTECH (Henzinger et Wong-toi, 1997) est un outil pour la modélisation et la vérification des systèmes temps réel. Il est approprié pour les systèmes qui peuvent être modélisés par des automates hybrides linéaires. Ainsi il permet de vérifier des propriétés exprimées en TCTL à partir d'une spécification du comportement par des automates hybrides linéaires. Intuitivement, un automate hybride linéaire est un multigraphe (V, E) étiqueté avec un ensemble fini X de variables réelles. Les (arcs) liens dans E représentent les actions discrètes du système et elles sont étiquetées avec les assignements de gardes à X. Les sommets (nœuds) représentent les activités continues de l'environnement, et ils sont étiquetés par des contraintes sur les variables dans X et leurs dérivés premiers. L'état d'un automate hybride linéaire change à travers les actions instantanées du système ou pendant des laps de temps, à travers les activités continues du système.

2.3.5 CADP

CADP (*CAESAR/ALDEBARAN Development Package*) (Fernandez et al., 1996) est un ensemble d'outils pour la validation de protocoles. Il fournit des outils de simulation et de vérification formelle. CADP est développé par l'équipe VASY de l'INRIA Rhone-Alpes et le laboratoire de VERIMAG. Il est disponible sur Internet (http://www.inrialpes.fr/vasy/cadp.html).

Les principales fonctionnalités de CADP sont la compilation, la simulation, la vérification formelle et la génération de tests pour les descriptions de systèmes utilisant l'algèbre de processus LOTOS. Il transforme ces spécifications en code C, en passant par une étape intermédiaire de modélisation par des réseaux de Pétri symboliques si nécessaire. Le code C dispose ensuite d'une API permettant son invocation dans le cadre d'une simulation, d'utilisation d'un outil de model checking ou même du développement d'un nouvel outil.

Choix du model checker. Nous avons constaté à travers notre étude des différents model checkers que ceux modélisant les systèmes temps réel sous forme d'automates hybrides ou temporisés sont plus riches que les autres. En fait, ils offrent une modélisation plus fidèle des systèmes temps réel dont le comportement dépend du temps quantitatif ainsi que la possibilité de vérifier des propriétés temporisées qu'il est impossible de vérifier par des outils tels que NuSMV et DCValid.

Si l'on se base sur la richesse de modélisation, HYTECH est le meilleur logiciel du fait qu'il supporte les automates hybrides linéaires. Ensuite, nous trouvons UPPAAL qui permet de modéliser les automates hybrides linéaires ainsi que les automates temporisés. Enfin nous trouvons KRONOS qui se base uniquement sur les automates temporisés. Par contre, pour la vérification des systèmes complexes, UPPAAL apparaît le plus puissant par rapport à KRONOS et HYTECH. Le temps consommé par UPPAAL pour la vérification est nettement plus réduit que celui consommé par KRONOS et HYTECH.

Il est à noter aussi que KRONOS et HYTECH se limitent à la vérification de quelques types de propriétés des systèmes temps réel, notamment les propriétés d'atteignabilité. Alors qu'UPPAAL, de son côté, permet de vérifier les propriétés d'atteignabilité, de sûreté, de non-blocage et de vivacité simple ou bornée.

Ainsi, il est clair que NUSMV et SPIN sont les meilleurs model checker du point de vue de la puissance et de la capacité de vérification des propriétés des systèmes. De ce fait, nous retenons NuSMV et SPIN comme outils de vérification formelle pour la vérification des propriétés sur les graphes sémantiques. Ce choix repose sur le fait que les graphes sémantiques se modélisent par de simples automates et n'ont pas besoin d'automates temporisés, l'évaluation du graphe sémantique se faisant par des ajouts ou des suppressions de nœuds et/ou d'arcs dans le graphe.

2.4 LES ALGORITHMES DE RESOLUTION

Dans cette section, nous présentons les algorithmes du model checking pour deux logiques temporelles : la logique temporelle linéaire LTL et la logique temporelle arborescente CTL (chapitre 3). Le model checking est une technique de vérification de propriétés par exploration de l'ensemble des modèles d'un système représentable par un graphe d'accessibilité. Pour les propriétés exprimables en CTL, qui est une logique d'état, l'algorithme de résolution du model checker est un algorithme de marquage des états du graphe par les propriétés qui les satisfont. Tandis que pour les propriétés exprimables en LTL qui est une logique de chemin, l'algorithme de résolution de base consiste à vérifier que l'intersection de l'ensemble des mots définis par le graphe et du langage défini par la négation de la propriété est vide.

Les structures de Kripke permettent de modéliser les comportements d'un système. La logique temporelle peut exprimer des propriétés sur ces comportements. Une structure de Kripke est utilisée pour représenter les liens entre les états du système à vérifier. Il s'agit d'un graphe orienté dans lequel les nœuds, appelés états, sont étiquetés par les états du système.

Définition 1 (Structure de Kripke). Une structure de Kripke est utilisée pour représenter les liens établis entre les états du système devant être vérifiés. C'est un graphe orienté où les nœuds sont appelés états et sont étiquetés par l'état du système. Une structure de Kripke M est un quintuplet défini par M= (Q, →, P, l, s_0) où :

- Q est l'ensemble des états ou configurations,
- $\rightarrow \subseteq Q \times Q$ est la relation de transition,
- P est un ensemble de propositions atomiques,
- $l : Q \rightarrow 2^P$ est la fonction d'étiquetage des états,
- $s_0 \in Q$ est l'état initial.

Définition 2 (*w*-automate). Un *w*-automate est un automate fini reconnaissant des mots de longueur infinie (*w*-mot). Il a la même structure qu'un automate classique : des états, des transitions entre les états, des étiquettes choisies parmi l'alphabet pour reconnaître les mots, un état initial, mais pas un état final (ou les états d'acceptation).

Un *w*-mot est reconnu par un *w*-automate s'il y a un chemin dans l'automate étiqueté par les lettres du mot, et ce *w*-mot est accepté si le chemin satisfait les conditions d'acceptation de l'automate.

Il y a plusieurs types de *w*-automate, leur différence est dans la façon de définir leurs conditions d'acceptation. Les plus connus sont les automates de Büchi introduit par Büchi en 1962 (Büchi, 1962).

Définition 3 (Automate de Büchi). Un automate de Büchi est un automate fini reconnaissant des mots infinis. Ils sont les plus utilisés en model checking. Un automate de Büchi est un quintuplet A = (*AP*, *Q*, *L*, *I*, *F*) où:

- *AP* est un ensemble de propositions atomiques.
- *Q* est un ensemble d'états finis.
- *L* est la fonction de transition de l'automate.
- *I* est un ensemble des états initiaux.
- $F \in Q$ est un ensemble associé avec les conditions d'acceptation.

Définition 4 (la Dégénéralisation). La dégénéralisation est la transformation d'un automate de *Büchi généralisé* en un automate de Büchi.

Définition 5 (automate de *Büchi généralisé*). Un automate de *Büchi généralisé* est un automate basé sur les transitions. Il se distingue par le fait que les propositions atomiques et les conditions d'acceptation se réfèrent aux transitions plutôt qu'aux états.

2.4.1 MODEL CHECKING LTL

La vérification des formules de la logique temporelle linéaire consiste à interpréter le système et la formule de la logique temporelle à vérifier comme deux *w*-automates, nommés *As* et *Af* respectivement, pour ramener le problème à des comparaisons de langage (Duret-Lutz et Rebiha, 2003). Une telle comparaison n'est pas facile à programmer. Comme la complémentation d'un automate de Büchi est une opération exponentielle, il est plus facile de nier la formule f avant la traduction et de vérifier si le résultat est vide, c'est-à-dire que le langage du produit synchronisé est vide.

L'approche par automate est simple, claire et l'extension de la logique temporelle par de nouveaux opérateurs est facilitée, car la prise en compte de ces nouveaux opérateurs ne se fait que lors de la traduction de la formule en automate et la suite reste inchangée. (Vardi, 1986)

Enfin, l'approche par automate du model checking (Vardi, 1986) (Vardi, 1996) consiste à interpréter le système et la formule à vérifier comme deux automates, pour ramener le problème à des comparaisons de langages.

La traduction du système en un *w*-automate ne demande pas beaucoup d'efforts lorsque le système est déjà représenté par une structure de Kripke. Il existe plusieurs algorithmes qui permettent la transformation des formules en logique temporelle linéaire vers des w-automates. Un historique des différents algorithmes de transformation est présenté dans la Figure 13 (Duret-Lutz et Rebiha, 2003).

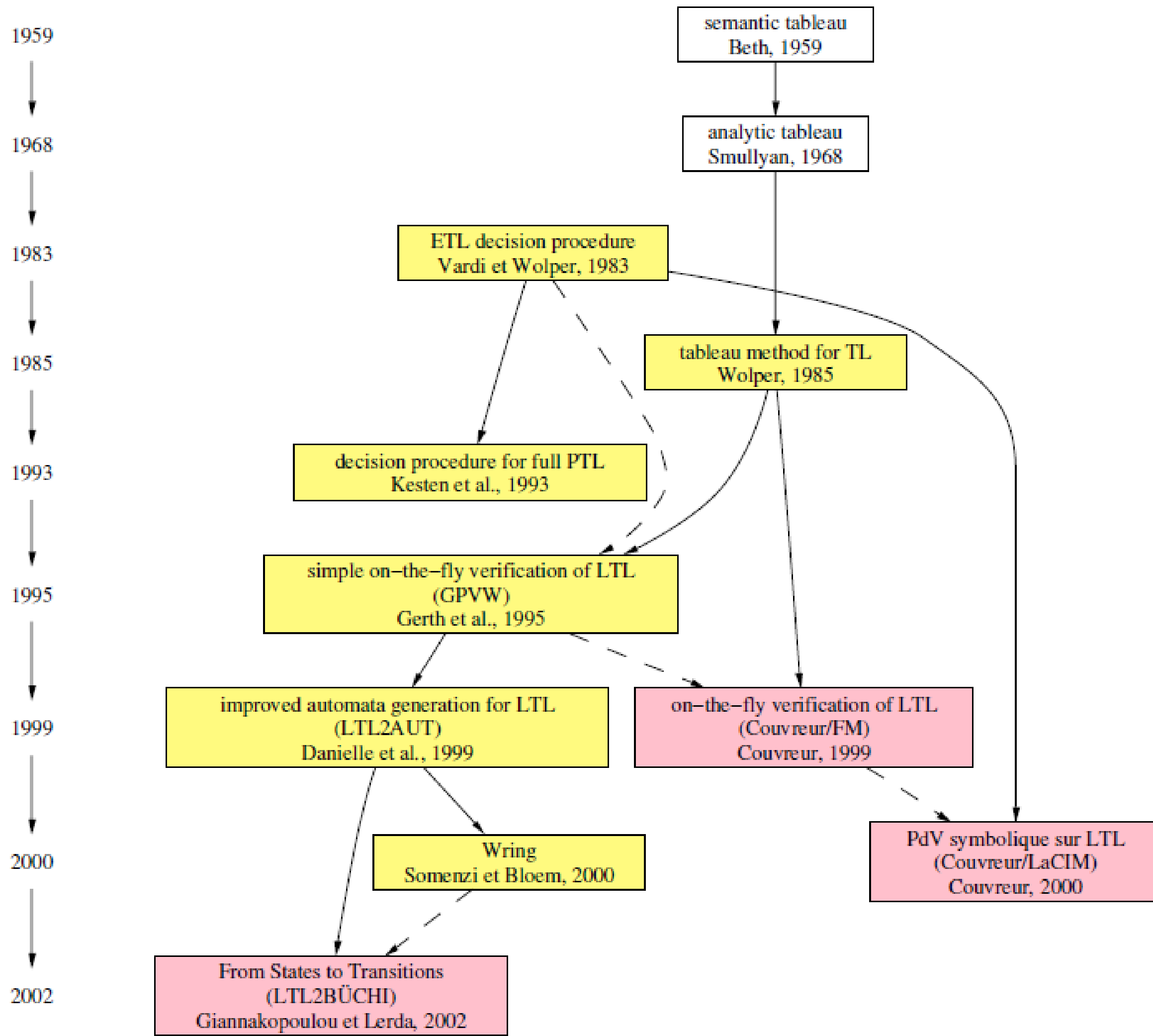


Figure 13. Les différents algorithmes de transformation.

Avant de commencer à présenter les différentes traductions de formules en automates (Beth, 1965) introduit une méthode pour la satisfaction d'une formule en logique temporelle linéaire par tableaux sémantique. Un tableau sémantique est une façon de prouver qu'une formule de logique propositionnelle *f* est satisfiable. Le plus souvent un tableau est utilisé pour des raisonnements par réfutation, c'est-à-dire que pour prouver que *f* est une tautologie, on montre avec un tableau que ¬*f* n'est pas satisfiable.

La preuve par tableau prend la forme d'un arbre dont les nœuds sont étiquetés par un ensemble de formules logiques. Dans le cadre de la logique propositionnelle, cet arbre peut être vu comme une mise sous forme normale disjonctive : les feuilles de l'arbre représentent des conjonctions de sous-formules atomiques à satisfaire, tandis que l'arbre lui-même représente la disjonction de ces conjonctions. Une branche peut être vue comme une suite d'implications, des feuilles vers la racine ; c'est-à-dire que la satisfaisabilité d'une feuille implique celle de la formule.

Une preuve par tableau s'effectue en partant de la formule *f* dont on veut établir la satisfaisabilité, puis en appliquant successivement toutes les règles de tableau possibles jusqu'à obtenir soit des nœuds contenant des formules contradictoires (c'est-à-dire contenant à la fois *p* et ¬*p*), soit des nœuds irréductibles (plus aucune règle ne s'applique).

Une branche dont l'un des nœuds contient des formules contradictoires est dite fermée. La formule *f* est satisfiable s'il existe une branche non fermée. La formule *f* n'est pas satisfiable si toutes les branches sont fermées (dans ce cas, ¬*f* est un théorème).

La première traduction (**l'approche combinatoire**) d'une formule vers un automate de Büchi fut proposée par (Wolper et al., 1983). Cette traduction produit un automate dont le nombre d'états est systématiquement de l'ordre de 2. Il s'agit d'un algorithme dont la correction est facile à établir, mais dont la complexité est toujours celle du pire cas. (Wolper et al., 1983), qui considèrent des automates de Büchi avec un seul ensemble d'états d'acceptation, exprime ces multiples conditions d'acceptions sous la forme d'un second automate, synchronisé par la suite avec celui de la formule. (Wolper, 2000) présente cet algorithme dans un cadre restreint à LTL, et en construisant un automate de Büchi généralisé.

Le raffinement successif dont la taille ne soit pas celle du pire cas est présenté par (Kesten et al., 1993), (Manna et Pnueli, 1995). En fait il ne s'agit pas à proprement parler de la construction d'un automate, mais d'un algorithme pour décider si une formule logique est satisfiable ou non. Il se trouve que cet algorithme construit un graphe orienté, étiqueté sur les états, et dont les chemins ne sont acceptés que si les formules du type ***a* U *b*** ou **F *b*** sont vérifiées. Il peut donc facilement être vu comme la construction d'un automate de Büchi généralisé. La logique est à peu près la même que dans l'algorithme précédent : on construit un ensemble d'états qui sont étiquetés par des ensembles de formules, mais les états sont choisis de façon à couvrir la formule à tester.

L'algorithme de traduction GPVW proposé par (Gerth et al., 1995) construit un automate de Büchi généralisé étiqueté sur les états. Il est basé sur la preuve par tableau. La preuve par tableau s'effectue en partant de la formule f dont on veut vérifier, puis en appliquant successivement toutes les règles de tableau possibles jusqu'à obtenir soit des nœuds contenant des formules contradictoires, soit des nœuds irréductibles.

LTL2AUT (Daniele et al., 1999a) (Daniele et al., 1999b) est un raffinement de l'algorithme précédent. Le cœur de l'algorithme est le même, mais quelques points précis ont été améliorés.

L'algorithme de Couvreur (Couvreur, 1999) aussi appeler Couvreur/ FM résout ce problème en faisant porter les conditions d'acceptation sur les arcs et non sur les états. Ceci permet de bien séparer le présent (l'arc) du futur (la destination de l'arc), et ainsi de partager les futurs.

Dans le cas de l'algorithme de Wring (Somenzi et Bloem 2000), l'automate construit est un automate étiqueté sur les états. Les minterms (un minterm est une conjonction de littéraux dans lequel toutes les propositions atomiques apparaissent) définissent donc des états et non des transitions. Les auteurs de Wring considèrent cette minimisation comme un problème d'optimisation linéaire, qu'ils résolvent avec une heuristique.

La construction d'un automate de Büchi généralisé étiqueté sur les états à partir de cette couverture est sans surprise. L'automate construit est ensuite réduit par une technique de simulation. Cet algorithme produit des automates systématiquement plus petits (au sens large) que ceux de GPVW et LTL2AUT. Il bat parfois celui de Couvreur ce qui semble indiquer que ce dernier pourrait être amélioré.

L'algorithme LTL2BA de traduction que proposent (Gastin et Oddoux, 2001) construit un automate de co-Büchi étiqueté sur les transitions. Il n'est pas basé sur une preuve par tableau. Dans une première étape, l'algorithme génère un automate de co-Büchi alternant très faible (noté VWAA). Dans une seconde étape, il construit un automate de Büchi généralisé étiqueté sur les transitions à partir du VWAA obtenu.

Enfin, l'automate généralisé peut être transformé en un automate de Büchi classique avec au plus n x 2n états (où n est la taille de la formule). Cette seconde étape n'est nécessaire que pour se comparer à d'autres algorithmes, ou pour utiliser un algorithme de vérification ne supportant pas les conditions d'acceptation de Büchi généralisées.

(Giannakopoulou et Lerda, 2001) (Giannakopoulou et Lerda, 2002) ont proposé un algorithme LTL2Büchi de traduction d'une formule LTL en un automate de Büchi classique étiqueté sur les transitions. L'algorithme est une preuve par tableau présentée dans le même formalisme que GPVW, mais décrite comme une version de LTL2AUT produisant des automates de Büchi étiquetés sur les transitions. La formule est tout d'abord réécrite en utilisant les optimisations de (Etessami et Holzmann, 2000) et celles de (Somenzi et Bloem, 2000). L'algorithme de traduction semble similaire à celui présenté par (Couvreur, 1999) deux ans plus tôt, mais visiblement développé de façon indépendante. LTL2BÜCHI utilise des vecteurs de booléens dans chaque nœud pour représenter les conditions d'acceptation qui lui sont associées, et introduit des techniques de comparaison de formules basées sur leur arbre de syntaxe abstrait. Cela diffère de Couvreur/FM qui fait tout cela avec des BDD.

À la différence de Couvreur/FM, qui utilise directement l'automate avec conditions d'acceptation généralisées produit, les auteurs de LTL2BÜCHI ajoutent une phase de dégénéralisation pour obtenir un automate de Büchi étiqueté sur les transitions simples. Cette dégénéralisation est celle de LTL2BA.

Tout comme l'algorithme du couvreur/FM, l'algorithme du Couvreur/ LaCIM (Couvreur, 2000) construit un automate de Büchi généralisé étiqueté sur les transitions puisque de tels automates sont plus compacts que ceux étiquetés sur les états. Mais à la différence du couvreur/FM, la construction n'est pas une méthode du tableau. L'un des avantages de cette traduction est l'utilisation d'une représentation symbolique (des BDD) de l'automate généré. Cette représentation ouvre la porte aux optimisations symboliques. Par exemple avec des techniques de point fixe, il est possible de supprimer de l'automate des états qui ne peuvent appartenir à aucun chemin acceptant. C'est d'ailleurs grâce à cette optimisation que cette méthode produit un automate à un seul état (l'état initial, sans successeur).

2.4.2 MODEL CHECKING CTL

C'est le premier algorithme de model checking à avoir été développé. L'algorithme de base du model checking CTL, dû à Queille, Sifakis, Clarke, Emerson et Sistla (Queille et Sifakis, 1982) (Clarke et al., 1986), est un algorithme de marquage. Son avantage majeur est de tourner en temps linéaire en chacune des entrées (la structure de Kripke et la formule). L'algorithme repose sur le fait que toute formule CTL peut s'exprimer par un nombre restreint de formules sur les états. Cela nous permet de raisonner en termes d'états (satisfaisant la formule) plutôt que d'exécutions.

L'algorithme prend en entrée une structure de Kripke *M* et une formule CTL φ. Il est à base de marquage : pour chaque sous-formule φ 'de φ, en commençant par la plus interne, on va marquer les états s de *M* qui vérifient φ'. On procède ensuite récursivement en réutilisant les marquages des sous-formules plus internes pour une sous formule plus externe. Finalement, *M* satisfait φ ssi l'état initial s_0 est marqué par φ. (Bardin, 2008)

```
Algorithme marquage
  input : formule φ normalisée, M= (Q,→, P, l, s0)

  Case 1 : φ = p
    for all s ∈ Q do
      if p ∈ l(s) then s. φ := true
      else s. φ ':= false
    end for

  Case 2 : φ = ¬ φ '
    do marking(φ ',M) ;
    for all s ∈ Q do s. φ := not(s. φ ') end for

  Case 3 : φ = φ ' ∧ φ ''
    do marking(φ ',M) ; marking(φ '',M) ;
    for all s ∈ Q do s.' := and(s. φ ',s. φ '') end for

  Case 4 : φ = EX φ '
    do marking(φ ',M) ;
    for all s ∈ Q do s. φ:= false end for
    for all (s, s') ∈−→ do
      if s'. φ '=true then s.' := true
    end for

  Case 5 : φ = E φ 'U φ ''
    do marking(φ ',M) ; marking(φ '',M) ;
    for all s ∈ Q do
      s. φ := false ;
      s.seenbefore := false
    end for
    L := Ø
    for all s ∈ Q do if s. φ ''=true then L :=L + {s} end for
    while L 6= Ø do
      choose s ∈ L ; L := L - {s} ;
       s. φ := true ;
      For all (s',s) ∈−→ do // s' predecessor of s
        if s'.seenbefore = false then
          s'.seenbefore := true ;
          if s'. φ ' = true then L := L + {s'} ;
        end if
      end for
    end while

  Case 6 : φ = À φ 'U φ ''
    for marking(φ ',M) ; marking(φ '',M) ;
      L := Ø ;
```

```
for all s ∈ Q do
   s.nb := degree(s) ; s. φ := false ;
   if s. φ '' = true then L := L + {s} ;
end for
while L 6= ∅ do
   choose s ∈ L ; L := L - {s} ;
   s. φ := true ;
   for all (s',s) ∈−→ do // s' predecessor of s
      s'.nb := s'.nb - 1 ;
      if (s'.nb = 0) and (s'. φ ' = true) and (s'. φ = false) do
         L := L + {s'} ;
      end if
   end for
end while
```

Script 9. Algorithme du Model checking CTL par marquage.

La logique CTL est une logique d'état, la vérification d'une formule CTL sur un automate consiste à marquer les états par les formules qu'ils satisfont. Un automate vérifie une propriété P si et seulement si tous les états initiaux sont marqués par P.

Pour résumé, ce qu'il faut retenir de l'algorithme du scipt 9 c'est que l'on peut étiqueter un état **s** par la formule $\boldsymbol{\varphi}$ dans les cas suivants :

- $\boldsymbol{\varphi}$ = ¬*a* : Quand l'état en question n'est pas étiqueté avec *a* (on peut donc l'étiqueter avec 7*a*).
- $\boldsymbol{\varphi}$ = *a* U *b* : Quand l'état en question est étiqueté avec *a* et aussi avec *b*
- $\boldsymbol{\varphi}$ = *a* V *b* : Quand l'état est étiqueté avec soit *a*, soit *b*, soit les deux.
- $\boldsymbol{\varphi}$ = E (*a* U *b*) : Quand l'état est étiqueté avec *b* (on peut donc directement l'étiqueter avec E (*a* U *b*)). Ou bien quand cet état "S" est étiqueté avec *a* et ce même état "S" a comme successeur un autre état S' déjà étiqueté avec E (*a* U *b*). (c'est-à-dire un chemin de S vers S').
- $\boldsymbol{\varphi}$ = EG *a* : On cherche le sous graphe engendré par les états qui sont étiquetés avec *a*, et dans ce sous graphe on cherche les composantes fortement connexes non triviales (triviale = un sommet seul sans boucle) et on les étiquète avec EG *a*. Puis on étiquète aussi avec EG *a* tous les états du sous graphe qui ont comme successeur les états qu'on a déjà étiquetés avec EG *a*.
- $\boldsymbol{\varphi}$ = Ex *a* : Quand l'état S a un successeur (suivant immédiat) qui est étiqueté avec *a*.

Si dans notre formule, nous avons des cas qui ne figurent pas parmi ces 6 règles, on applique des règles de transformations pour les obtenir.

3 L'EXPLOSION COMBINATOIRE

L'explosion combinatoire du nombre d'états est une limite sérieuse à l'utilisation du model checking dans le domaine industriel. De nombreuses recherches ont été effectuées pour trouver une solution à ce problème (Parreaux, 2000).Dans cette section, nous présenterons quelques-unes de ces recherches. L'explosion combinatoire n'a pas été étudiée en détail et ca sera dans nos perspectives.

3.1 L'ABSTRACTION

La technique d'abstraction a pour but d'ignorer certains aspects de l'automate, par exemple en supprimant des variables. Le système est représenté par un modèle dont le comportement est une abstraction du système original permettant de vérifier certaines propriétés.

Les techniques d'abstraction ont été classées selon les effets qu'elles ont sur le modèle. Il y a quatre types d'abstraction :

- **L'abstraction par restriction** : elle consiste à rendre certains comportements impossibles du système en retirant des états ou des transitions directement sur l'automate.
- **L'abstraction par automates observateurs** : elle consiste à restreindre les comportements de l'automate sans toucher à sa structure. Dans la pratique, cette abstraction est peu efficace, car seules les propriétés devant être vérifiées sur le seul état initial ont une réduction de la taille de l'automate produit.
- **L'abstraction par fusion d'états** : cette technique consiste à regrouper un ensemble d'états. La fusion d'états regroupe les états ayant des caractéristiques communes en un super état. Toutes les transitions arrivant ou partant d'un état constituant le super état, arrivant ou partant maintenant de ce dernier. La fusion des états augmente le nombre de comportements possibles du système.
- **L'abstraction sur les variables** : ce dernier type d'abstraction agit sur la partie des données des automates à variables. Cette technique peut se faire d'une façon très simple en supprimant les variables sur lesquelles l'abstraction doit être réalisée. La suppression de certaines variables amène en effet à faire disparaître certaines relations entre les variables, mais il est difficile, lors du choix des variables à supprimer, de tenir compte de toutes les relations entre les variables. Cette technique est considérée comme un cas particulier de l'abstraction par fusion d'états.

3.2 L'INTERPRETATION ABSTRAITE

L'interprétation abstraite (Cousot et Cousot, 1976) (Cousot et Cousot, 1977) (Cousot, 2000) est une théorie de l'approximation issue des travaux sur la sémantique des langages. Elle consiste à déterminer des sémantiques liées par des relations d'abstraction. Son utilisation avec le model checking est très peu automatisée et donc le niveau d'expertise nécessaire pour utiliser cette technique est important.

Les résultats obtenus avec cette technique sont en effet très intéressants parce que c'est une technique complète et qu'il est possible de traiter des problèmes infinis en réalisant une abstraction finie de ceux-ci. Cette technique a d'ailleurs été utilisée dans des problèmes industriels avec succès.

3.3 LE MODEL CHECKING SYMBOLIQUE

Le principe du model checking symbolique est d'utiliser une représentation symbolique de l'automate à vérifier. C'est-à-dire que les états sont manipulés par paquets au lieu d'être considérés un par un. Pour cela, l'algorithme utilise des fonctions booléennes sous la forme de BDDs en tant que représentation interne.

Le model checking symbolique repose sur le calcul itératif d'ensemble d'états et est donc plus adapté à la logique temporelle arborescente que la logique temporelle linéaire. Le fonctionnement d'un model checker symbolique consiste en l'obtention de points fixes pour déterminer les états accessibles ainsi que ceux qui satisfont une ou plusieurs propriétés. Ces points fixes sont calculés à l'aide de deux transformateurs de prédicats associés au franchissement des transitions : le premier (Post) détermine, pour un ensemble d'états donnés, l'ensemble des états successeurs tandis que le second (Pre) est l'opération réciproque du premier. Ces techniques sont souvent plus efficaces pour les systèmes présentant un fort degré de concurrence, mais font intervenir des mécanismes trop lourds pour des systèmes séquentiels.

3.4 Les BDD (Binary Diagram Decision)

Beaucoup de problèmes peuvent être réduits à une séquence d'opérations sur des fonctions booléennes. Les représentations utilisées classiquement sont la forme normale disjonctive (FND), la forme normale conjonctive (FNC), ou même la table de vérité.

Les BDD (*Binary Decision Diagrams*) (Akers, 1978) sont des structures de données particulières permettant une représentation et une manipulation très efficace de fonctions booléennes. La forme la plus commune de BDD a été introduite par Bryant (Bryant, 1986) (Bryant, 1992) et a ensuite trouvé de nombreuses applications au model checking, car elle permet de représenter la relation de transition d'un système de transition sous une formule ensembliste.

3.5 Les methodes utilisant les contraintes

Les contraintes sont utilisées dans des outils du model checking pour implémenter la représentation symbolique des états. Ces techniques permettent de traiter des modèles infinis ou paramétrés. Dans le cas des systèmes paramétrés, la vérification est impossible avec les techniques classiques puisque la modification du paramètre modifie le système sur lequel doivent être vérifiées les propriétés. Une proposition a été faite dans (Fribourg et Peixoto, 1994) et qui repose sur la programmation logique avec contrainte (Jaffar et Lasez, 1987). Le principe de base est de construire, étant donné un automate, un programme logique tel que toute suite d'actions reconnue par l'automate soit la réponse à un but soumis au programme et inversement.

Dans le cas des systèmes infinis, une technique a été développée par l'équipe de recherche de l'institut Max-Planck en Allemagne. Cette technique permet de travailler sur des ensembles infinis d'états en préservant les propriétés de sûreté et de vivacité (chapitre 4, section 2). Les résultats des évaluations confirment l'intérêt de cette technique et vont dans le sens d'une extension de la technique à d'autres types de contraintes.

3.6 La compression memoire

Cette technique, comme son nom l'indique, sert à compresser la mémoire en compressant les états du système. Le but de cette technique est de gagner de la place en mémoire sans trop augmenter le temps de vérification.

- **Méthode classique** : ces méthodes telles que le Byte masking, RLE (Held, 1987) (Lynch, 1985) (Nelson, 1991) (Storer, 1988) ou le codage Huffman (Huffman, 1952) permettent de diminuer la mémoire nécessaire à la vérification sans changer la structure du graphe. Un état

compressé peut être rendu à son état d'origine par l'opération de décompression, ces méthodes préservent donc toutes les propriétés du graphe d'origine. Par contre, le principal défaut de RLE (par exemple) est que, dans le cas où il n'existe pas ou peu de motifs sur l'état, l'état compressé peut être plus grand que l'état original. Le codage Huffmann est basé sur des dictionnaires qui permettent de recoder les octets selon la fréquence de leurs utilisations : le but est d'utiliser moins de bits pour coder les objets les plus utilisés et d'en utiliser plus pour les objets apparaissant le moins souvent. Malgré ces bonnes performances, cette technique est peu utilisée, car pour comparer deux états il faut décompresser ceux-ci entrainant un coût trop important au niveau du temps d'exécution.

- **Le collapse** : cette technique utilise la constitution des états pour réaliser la compression (Holzmann, 1997). Les états possèdent des propriétés non pas sur la structure de ces états, mais entre les états. Il peut sembler intéressant de ne stocker les variables des états ayant la même valeur qu'une seule fois et de transformer les états en tableau de pointeurs vers ces variables. Cette technique de compression donne de bons résultats dans le cas où les états ne sont pas tous atteignables, ce qui est souvent le cas dans les systèmes complexes.
- **Le Bistate Hashing (Super trace)** : cette dernière technique est la plus efficace en terme de compression. L'idée de base développée par G. Holzmann (Holzmann, 1991) pour SPIN (Ben-Ari, 2008) est d'utiliser une table de « hashing » servant à accélérer la recherche des états dans la phase de comparaison pour réaliser la compression. Une amélioration a été apportée à cette technique, car il est possible que deux états différents possèdent la même valeur de « hashing ».

Les méthodes utilisant la compression sont plus efficaces avec les algorithmes du model checking réalisant conjointement la création du graphe d'accessibilité et la vérification des propriétés.

3.7 Les methodes a la volee

L'avantage de cette technique est qu'elle permet de ne pas stocker la totalité du graphe puisqu'il ne sera pas nécessaire de parcourir celui-ci pour vérifier une propriété (Gerth et al., 1995). Avec cette technique, le model checker pourra toujours donner un résultat partiel s'il s'arrête faute de mémoire disponible. Le résultat est suffisant lors d'une détection d'une erreur, mais il est impossible de corriger l'erreur sur le système si le parcours n'est pas terminé. Des heuristiques peuvent essayer d'augmenter la couverture en variant les comportements ou en les favorisant ceux pouvant aboutir à une erreur.

Ces techniques ont un inconvénient important qui impose de reconstruire plusieurs fois les mêmes parties du graphe. Une optimisation a été faite dans (Godefroid et al., 1995) qui repose sur les résultats de la théorie des graphes. Lors du parcours d'un graphe fini, la terminaison de ce dernier est garantie par la seule connaissance de tous les nœuds se trouvant sur le chemin menant de l'état initial du graphe à l'état courant.

L'algorithme de model checking à la volée, présenté dans (Courcoubetis et al., 1992) (Holzmann, 1991), a été implémenté dans le model checker SPIN. Cet algorithme fonctionne en deux phases qui peuvent être réalisées l'une après l'autre ou concurremment. La première phase construit

et stocke les états d'acceptation et la deuxième phase consiste à détecter des cycles passant par ces états d'acceptation.

3.8 L'ORDRE PARTIEL

La technique d'ordre partiel (Godefroid et al., 1996) vise à réduire le nombre d'états du système en utilisant des relations d'équivalence entre les états et les transitions. Cette méthode ne conserve pas les propriétés de la logique temporelle linéaire contenant l'opérateur suivant (***Next***), mais elle conserve toute les autres propriétés.

La technique d'ordre partiel a été utilisée avec succès dans des outils tels que SPIN. Elle donne en général de bons résultats pour un coût en temps d'exécution assez faible. Cette technique peut être classée comme abstraction par restriction, puisque l'on supprime des chemins d'exécution : il faut noter que toutes les propriétés exprimables en logique temporelle sont conservées, excepté celles contenant l'opérateur suivant (***Next***).

4 TRAVAUX CONNEXES

Le model checking a été utilisé dans plusieurs domaines, et surtout dans les systèmes critiques (centrale Nucléaire, machine de traitement des maladies cancéreuses,…etc.). Il y a plusieurs travaux de recherches qui modélisent leurs applications Web en graphes orientés afin d'utiliser la technique du model checking comme méthode formelle (Huang et al., 2004). Dans (Yuen et al., 2005), les auteurs proposent un modèle de comportement à l'application Web, appelé « *Automates Web*» basé sur l'architecture du modèle MVC[11]. Ils modélisent le comportement d'une application Web avec du contenu dynamique. Dans (Haydar et al., 2004), les auteurs présentent une approche formelle pour la modélisation des applications Web en utilisant des automates communicants. Ils observent le comportement extérieur de la partie explorée d'une application Web à l'aide d'un outil de monitoring (outil de surveillance). Le comportement observé est ensuite transformé en automates communicants représentant toutes les fenêtres de l'application en cours de test par l'interception des requêtes HTTP[12] et les réponses en utilisant un serveur proxy.

En ce qui le domaine des réseaux de capteurs, les méthodes et les outils du model checking ont été utilisés, les rares travaux consistaient en effet à vérifier la spécification et l'exactitude d'un protocole (Chiyangwa et Kwiatkowska, 2003) d'une couche bien déterminée ou bien à vérifier la compatibilité des composants utilisés dans TinyOS[13]. (Coleri et al., 2002) représente le seul travail de modélisation formelle d'un réseau de capteurs en tenant compte de la consommation d'énergie. Dans ce travail, les auteurs ont donné une description détaillée d'un nœud décrit avec TinyOS à l'aide des automates hybrides, ce qui leur permet de vérifier des propriétés de type « la couche radio au niveau de paquet ne peut pas être dans l'état de transmission en même temps que la couche radio au niveau 'byte' est dans l'état de réception". D'autre part, ils ont essayé de déterminer les pourcentages des nœuds morts dans tout le réseau (le réseau est partitionné en groupe) et ceci en simplifiant le comportement de chaque nœud. Ce résultat est obtenu par simulation non exhaustive du modèle.

[11] Modèle, Vue et Control
[12] Hypertext Transfer Protocol
[13] Système d'exploitation open-source conçu pour des réseaux de capteurs sans fil

5 CONCLUSION

Le model checking est un modèle de vérification de grand logiciel. Nous avons vu dans ce chapitre les avantages et inconvénients de ce modèle de vérification, ainsi que les autres méthodes de vérification de logiciel (par exemple : la simulation, les méthodes basées sur le test et les méthodes basées sur la preuve de théorème).

De nombreuses approches ont été réalisées pour résoudre le problème de l'explosion combinatoire dont souffre la technique du model checking avec plus ou moins de succès. Le vérificateur de systèmes informatiques pourra choisir l'une de ces solutions selon le résultat espéré, ou il pourra en utiliser plusieurs conjointement.

Nous avons présenté et étudié quelques model checkers. Ces outils sont répertoriés en deux classes, Les model checkers qualitatifs et les model checkers quantitatifs. Notre objectif étant de choisir un model checker puissant pour la vérification des graphes sémantiques. Nous avons retenu deux model checkers qualitatifs SPIN et NuSMV.

Ce choix repose sur le fait que les graphes sémantiques se modélisent par de simples automates, sans besoin d'automates temporisés et l'évaluation du graphe sémantique se fait par des ajouts ou des suppressions de nœuds et/ou d'arcs dans le graphe.

Ce chapitre présente les algorithmes du model checking pour les deux logiques temporelles : la logique temporelle linéaire LTL et la logique temporelle arborescente CTL. Ces deux logiques sont présentées en détail dans le chapitre suivant.

Chapitre 4

La logique temporelle

Résumé

Le model checker vérifie le comportement des systèmes à l'aide de spécifications exprimées en logique temporelle. Cette dernière permet d'exprimer l'évolution de l'état d'un système. Les logiques temporelles permettent de représenter et de raisonner sur certaines propriétés de sûreté des systèmes. En ce sens, elles sont bien adaptées à la spécification et à la vérification des systèmes réactifs et concurrents.Les requêtes en logique temporelle sont introduites pour accélérer la compréhension du système en découvrant des propriétés non connues a priori. Pour aider l'utilisateur à comprendre le comportement du système, les requêtes en logique temporelle utilisent une technique similaire au model checking pour déterminer les propriétés temporelles plutôt que de simplement les contrôler. Le Query Checking est une extension de la technique du model checking.

Plan

1 La logique temporelle 87
1.1 La logique temporelle linéaire 88
1.2 La logique temporelle arborescente 89
2 Classification des propriétés temporelle 91
3 Choix d'une logique temporelle 93
4 Extension de la logique temporelle. 94
5 Logique temporelle et automates 95
5.1 Vérification de formule CTL 95
5.2 Vérification de formule LTL 96
6 Les requêtes en logique temporelle 97
6.1 Types de requête 98
6.2 Résolution des requêtes 99
6.2.1 Approche de Chan 101
6.2.2 Approche de Bruns & Godefroid 104
6.2.3 Approche de Schnoebelen 105
6.2.4 Approche de Chechik 106
7 Conclusion 108

Pour spécifier formellement les exigences de fonctionnement d'un système informatique, il est naturel de raisonner sur la notion de temps. Il s'agit d'exprimer par exemple le droit d'accès à une ressource pendant une certaine durée, l'obligation de la libérer avant un instant donné, ou encore l'obligation qu'une certaine tâche ne soit pas exécutée pendant un temps trop important. Les logiques temporelles apparaissent comme des outils adéquats pour spécifier de telles notions. La notion de logique temporelle, utilisée pour la première fois par (Pnueli, 1977) dans la spécification de propriétés formelles, sont assez facile d'utilisation. Les opérateurs sont très proches des modalités de la langue naturelle. La logique temporelle est une extension de la logique conventionnelle (propositionnelle), elle intègre de nouveaux opérateurs qui expriment la notion du temps. En effet, il n'est pas possible d'exprimer dans la logique classique une assertion liée au comportement d'un programme tel qu'« après l'exécution d'une instruction *i*, le système se bloque ». Dans cette assertion, les actions s'exécutent suivant un axe de temps : à l'instant *t*, exécution de l'instruction *i*, et a *t*+1 blocage du système. Il faut donc une logique qui modélise les expressions du passé et du futur.

Ce chapitre introduit les logiques temporelles, qui seront le formalisme employé pour spécifier les comportements attendus de nos graphes sémantiques. Ces logiques surpassent les autres moyens de spécification du comportement, que ce soit le langage naturel (trop ambigu) ou des formalismes à base de diagrammes (peu expressifs et souvent ambigus). Elles sont plus concises et faciles à lire que des langages de spécification plus généralistes (comme la logique du premier ordre). Enfin, et surtout, leur vérification peut être automatisée, contrairement aux autres logiques. Ce chapitre introduit aussi les requêtes en logique temporelle qui ont été proposées par (Chan, 2000) afin d'accélérer la compréhension de conception en découvrant les propriétés non connues a priori. Le model checking a été initialement proposé comme une technique de vérification, mais il est aussi extrêmement précieux pour la compréhension du modèle. Il est rarement possible de commencer l'étude d'une conception avec une spécification complète disponible. Au lieu de cela, nous commençons avec quelques propriétés clés, et nous tentons d'utiliser le model checker pour les valider. Lorsque les propriétés ne sont pas vérifiées, dans ce cas, qui est en faute : les propriétés ou la conception ? En règle générale, les deux ont besoin d'être modifiés : la conception, si un bug a été trouvé ; et les propriétés si elles étaient trop fortes ou mal exprimées. Ainsi, ce processus ne vise pas seulement à la construction du modèle correct du système, mais aussi à découvrir les propriétés qu'il doit avoir. Les requêtes en logique temporelle seront utilisées dans le but d'apporter de l'expressivité au langage de requêtes standard SPARQL pour les graphes sémantiques.

1 LA LOGIQUE TEMPORELLE

La logique temporelle CTL* (Bardin, 2008) (Emerson et Sistla, 1984) est une logique arborescente, très expressive (pas de limitation à l'utilisation des connecteurs temporels et quantificateurs de chemins) et permet de mélanger des propriétés portant sur une exécution particulière d'un modèle et sur les possibilités d'évolution d'une exécution. On distingue principalement deux classes de logiques. La logique du temps linéaire permettant d'exprimer des propriétés portant sur des chemins individuels du programme et la logique du temps arborescent permettant d'exprimer des propriétés portant sur les arbres d'exécution du programme. La

vérification de propriétés exprimées en CTL* est un problème de grande complexité. Afin de diminuer la complexité, la plupart des outils permettent de vérifier des restrictions exprimées en LTL et CTL.

Les opérateurs temporels permettent d'exprimer des propriétés sur les enchaînements d'états appelés exécutions. Selon les logiques, les exécutions sont soit des séquences d'états, soit des arbres qui représentent l'évolution du système. La différence entre les logiques temporelles provient de l'ensemble d'opérateurs temporels qui peut être utilisé et des objets sur lesquels ils sont interprétés (séquences ou arbres d'états). La Figure 14 montre le pouvoir d'expression des logiques temporelles. CTL et LTL sont moins expressives que CTL*.

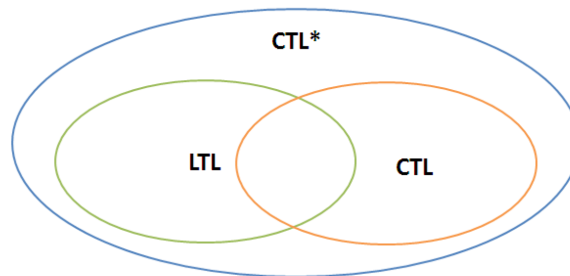


Figure 14. Les logiques temporelles LTL, CTL et CTL*.

1.1 LA LOGIQUE TEMPORELLE LINEAIRE

La logique temporelle linéaire ou LTL « Linear Temporal Logic » (Vardi, 1996) (Manna et Pnueli, 1992) permet de représenter le comportement des systèmes au moyen des propriétés qui décrivent le système pour lesquels le temps se déroule linéairement. En clair, on spécifie le comportement attendu d'un système, en spécifiant l'unique futur possible comme une séquence d'actions qui se suivent.

Soit $\varphi\ et\ \psi \in LTL$. Les formules LTL sont définies par la grammaire suivante :

- **Les propositions atomiques** : φ = *p*, *q*, …, *true*, *false*
- **Les connecteurs booléens** : toutes formules de la forme $\neg\,\varphi,\ \varphi \wedge \psi,\ \varphi \vee \psi,\ \varphi \Rightarrow \psi,\ \varphi \Leftrightarrow \psi$
- **Les connecteurs temporels** : toutes formules de la forme $\mathrm{G}\,\varphi,\ \mathrm{F}\,\varphi,\ \varphi\,\mathrm{U}\,\psi,\ \mathrm{X}\,\varphi$

On définit les opérateurs temporels comme suit :

X (next) : X*p* signifie que *p* est vrai dans l'état suivant le long de l'exécution.

Figure 15. L'opérateur suivant

F (eventually) : F*p* signifie que *p* est vrai plus tard au moins dans un état de l'exécution.

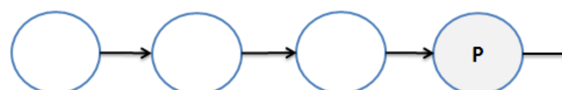


Figure 16. L'opérateur éventuellement.

G (globally ou always) : G*p* signifie que *p* est vrai dans toute l'exécution.

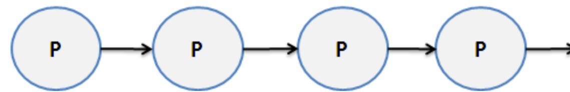


Figure 17. L'opérateur toujours.

U (until) : *p*U*q* signifie que *p* est toujours vrai jusqu'à un état où *q* est vrai.

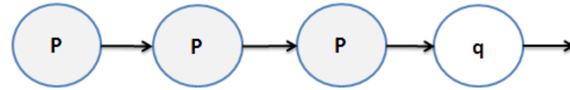


Figure 18. L'opérateur jusqu'à.

1.2 La logique temporelle arborescente

La logique temporelle arborescente ou CTL « Computation Tree Logic » (Clarke et al., 1986) permet de considérer plusieurs futurs possibles à partir d'un état du système plutôt que d'avoir une vue linéaire du système considéré.

Soit $\varphi\ et\ \psi \in CTL$. Les formules CTL sont définies par la grammaire suivante :

- **Les propositions atomiques** : φ = *p*, *q*, …, *true*, *false*
- **Les connecteurs booléens** : toutes formules de la forme $\neg\,\varphi,\ \ \varphi \wedge \psi,\ \ \varphi \vee \psi,\ \ \varphi \Rightarrow \psi,\ \ \varphi \Leftrightarrow \psi$
- **Les connecteurs temporels** : toutes formules de la forme $\mathrm{EF}\ \varphi,\ \ \mathrm{EG}\ \varphi,\ \ \mathrm{E}\ (\varphi\ \mathrm{U}\ \ \psi),\ \ \mathrm{EX}\ \varphi, \mathrm{AF}\ \varphi,\ \ \mathrm{AG}\ \varphi,$ $\mathrm{A}\ (\varphi\ \mathrm{U}\ \ \psi),\ \ \mathrm{AX}\ \varphi$

La logique temporelle linéaire considère une seule exécution du système, et les connecteurs permettent de parler de propriétés le long de cette exécution. C'est une vision linéaire du temps : le futur est fixé. Cependant, on peut aussi voir le temps comme une structure branchante ou arborescente : à chaque état du système, plusieurs futurs sont possibles, selon l'action qui sera effectuée. Les quantificateurs de chemins permettent de quantifier des propriétés sur les exécutions futures possibles à partir d'un état. On parle de propriétés d'états (d'un système). Les opérateurs de la logique temporelle arborescente sont définis en ajoutant l'opérateur **A** (toute exécution) ou l'opérateur **F** (il existe une exécution) devant les opérateurs de la logique temporelle linéaire.

EF *p* : il existe une exécution (**opérateur E**) conduisant à un état ou *p* est vérifiée (**opérateur F**). (Figure 19)

AF *p* : pour toute exécution (**opérateur A**), il existe un état ou *p* est vérifiée (**opérateur F**). (Figure 20)

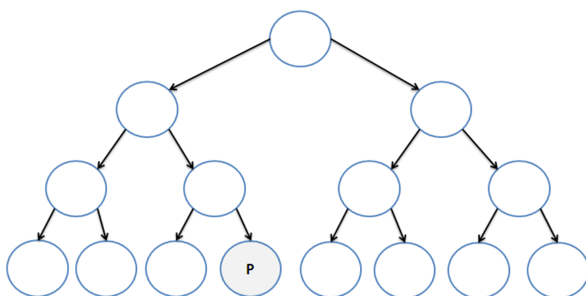


Figure 19. L'opérateur EF.

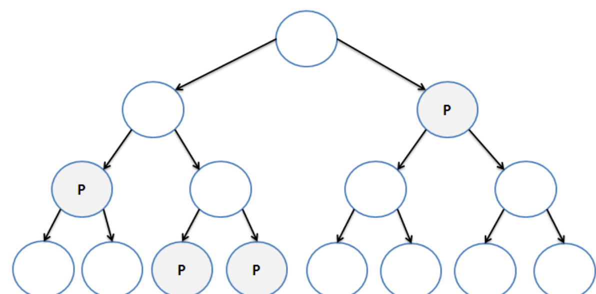


Figure 20. L'opérateur AF.

EG *p* : il existe une exécution (**opérateur E**) ou *p* est toujours vérifié (**opérateur G**). (Figure 21)

AG *p* : pour toute exécution (***opérateur A***), *p* est toujours vérifié (***opérateur G***). (Figure 22)

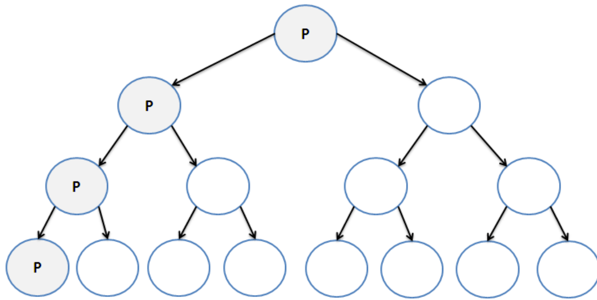


Figure 21. L'opérateur EG.

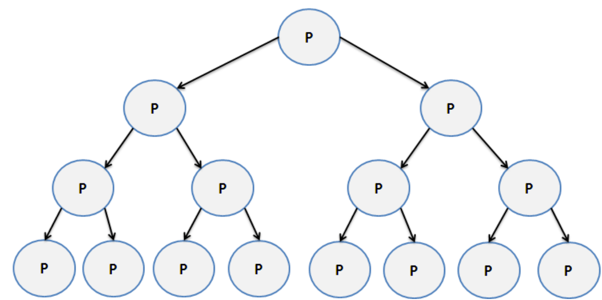


Figure 22. L'opérateur AG.

AX *p* : tous les états immédiatement successeur satisfont *p.* (Figure 23)

EX *p* : il existe une exécution (**opérateur E**) dont le prochain état satisfait *p.* (Figure 24)

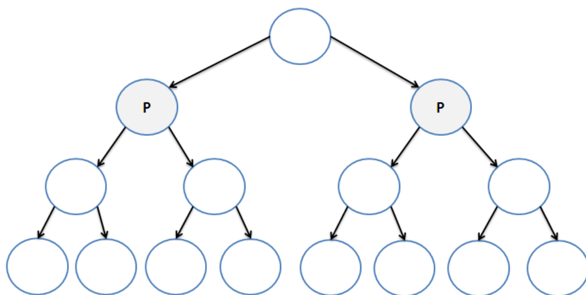


Figure 23. L'opérateur AX.

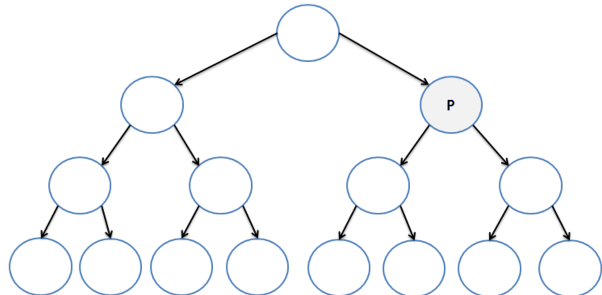


Figure 24. L'opérateur EX.

À *p* U *q* : pour toute exécution (**opérateur A**) *p* est vérifié jusqu'à ce que *q* le soit**.** (Figure 25)

E *p* U *q* : il existe une exécution (**opérateur E**) durant laquelle *p* est vérifié jusqu'à ce que *q* le soit. (Figure 26)

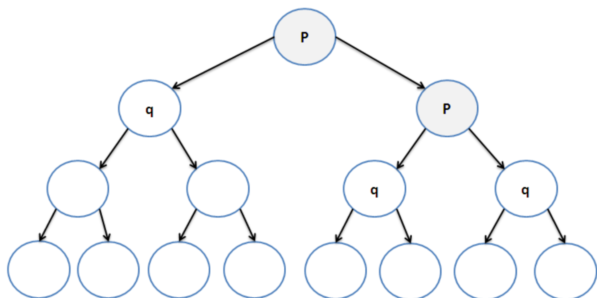


Figure 25. L'opérateur A avec U.

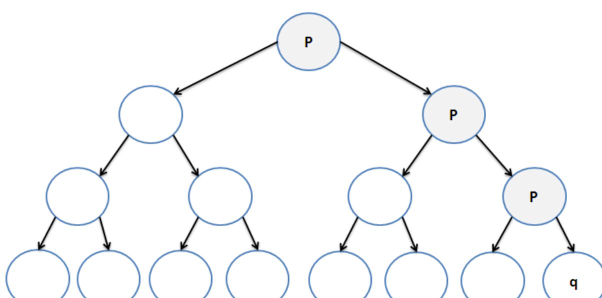


Figure 26. L'opérateur E avec U.

Il existe d'autres opérateurs tels que **AG EF *p*** (à partir de n'importe quel état atteignable, il est possible d'atteindre un état satisfaisant p).

2 CLASSIFICATION DES PROPRIETES TEMPORELLE

Les propriétés exprimables en logique temporelle peuvent être classées en cinq grandes catégories (Parreaux, 2000) : sûreté, vivacité, atteignabilité, équité et absence de blocage. Toute propriété, exprimée en logique temporelle, en vertu du théorème de décomposition (Alpern et Schneider, 1985) peut être réécrite sous la forme d'une conjonction de propriété de sûreté et de vivacité. Cette section présente ces différents types de propriétés puis, pour chacune, elle présente le pouvoir d'expression des logiques temporelles en fonction du type de propriété à vérifier.

Propriété de sûreté (safety). La propriété de sûreté a pour but de dire que quelque chose de mauvais ne se produit jamais. Cette propriété permet de définir les contraintes relatives aux différentes parties du système. La vérification d'une propriété de sûreté est simple et peut être réalisée par tous les model checker. Pour exprimer cette propriété on utilise l'opérateur toujours ***G p*** où *p* est une formule du passé.

Bien qu'il n'existe pas de model checker traitant la logique temporelle avec des opérateurs du passé, le formalisme de logique temporelle intégrant des opérateurs du passé et du futur est plus riche et donc facilite la transcription des énoncés en langue naturelle. Il est possible de vérifier les propriétés en logique temporelle contenant des opérateurs du passé en utilisant une des deux techniques suivantes :

- **Élimination du passé** : cette méthode consiste à transformer la formule en logique temporelle de telle sorte que les opérateurs du passé disparaissent au profit d'opérateur du futur.
- **Utilisation des variables d'histoire** : cette deuxième méthode consiste à transformer la propriété en logique temporelle en une négation de propriété d'atteignabilité en utilisant des variables d'histoires (variable de type booléen). Ces variables sont utilisées pour mémoriser l'occurrence de certains événements sans changer le comportement du système. Cette méthode à tendance à faire exploser le nombre d'états et elle nuit à la lisibilité de spécification rendant plus difficile la génération du code automatique.

Propriété de vivacité (Liveness). La propriété de vivacité énonce que quelque chose finira par avoir lieu. Il s'agit d'une propriété plus forte que l'atteignabilité puisque l'atteignabilité indique qu'il est possible que quelque chose se passe, alors que la propriété de vivacité indique que quelque chose doit se passer.

Cette propriété peut être exprimée aussi bien en LTL qu'en CTL. Lorsque l'on se place dans la logique LTL, l'opérateur ***F*** énonce des propriétés de vivacité et en CTL c'est l'opérateur composé ***AF*** qui le permet. L'opérateur LTL ***U*** lui aussi donne des propriétés de vivacité puisqu'il garantit qu'une propriété aura lieu.

Propriété d'atteignabilité (Reachability). La propriété d'atteignabilité a pour but de déterminer si lors du parcours du graphe d'accessibilité une situation est ou non accessible. Il peut aussi s'agir d'un état ou d'un ensemble d'états ayant en commun une propriété donnée. Nous remarquerons aussi ici qu'un état peut être déterminé exactement grâce à une propriété représentant exactement l'évaluation de toutes les variables du système en cet état.

L'opérateur de possibilité ***EF*** est nécessaire pour pouvoir exprimer les propriétés d'atteignabilité en logique temporelle. Il est donc possible d'utiliser CTL pour exprimer ce type de propriétés. En CTL, cela est possible à l'aide de formules de la forme ***AG EF*** (*p*) ou *p* représente une formule propositionnelle. La non-disponibilité de cet opérateur en LTL rend impossible, sauf pour quelques cas particuliers, l'expression de telles propriétés dans ce formalisme.

Propriété d'équité (Fairness). La propriété d'équité énonce que quelque chose aura lieu un nombre infini de fois. On parle aussi de vivacité répétée ou d'atteignabilité répétée. Ces différentes terminologies se justifient par des utilisations différentes d'un point de vue méthodologique. Ce sont des propriétés courantes des systèmes dès lors que ceux-ci sont obtenus en utilisant le produit de plusieurs automates.

Il existe deux formes d'équité, l'équité faible et l'équité forte [4] respectivement définies par :

- **Equité forte** : si *p* est infiniment souvent demandé, alors *p* sera obtenu (infiniment souvent).
- **Equité faible** : si *p* est continuellement demandé (sans interruption), alors *p* sera obtenu (infiniment souvent).

La différence entre équité faible et forte trouve sa légitimité dans deux propriétés. Premièrement une propriété d'équité forte implique la propriété d'équité faible correspondante et ensuite la propriété d'équité faible est plus facile à obtenir que la propriété d'équité forte. Par contre au niveau de la vérification, il n'y a pas de différences.

Il est possible d'exprimer une propriété d'équité forte et une propriété d'équité plus faible en utilisant soit le quantificateur ***E***, soit le quantificateur ***A*** en utilisant la possibilité d'exprimer des propriétés de chemins avec la CTL. En effet, **AF *p*** est plus fort que **F *p*** avec de l'équité, mais **EF *p*** est moins fort que celui-ci. Nous pouvons donc en quelque sorte encadrer la propriété nécessitant de l'équité par deux propriétés CTL.

Absence de blocage (Deadlock freeness). La propriété d'absence de blocage énonce que le système ne peut pas se trouver dans une situation où il est impossible d'évoluer, si cela n'a pas été explicitement spécifié. Cette propriété est particulièrement importante dans le sens où elle constitue une propriété de correction pour les systèmes supposés ne jamais terminer et constitue une propriété de correction pour tous les systèmes dès lors que l'on a spécifié les états finaux d'un système.

Les propriétés de blocage ne peuvent donc être vérifiées sur les abstractions de systèmes que si celles-ci sont faites explicitement pour conserver ces blocages contrairement aux propriétés de sûreté. Si l'on veut vérifier tous les blocages, il faut absolument ne faire aucune abstraction, mais par contre si le système considéré est une abstraction et que l'on détecte un blocage celui-ci existe nécessairement. Plusieurs techniques peuvent être mise en œuvre pour vérifier l'absence de blocage. Pour cela la propriété ***AG*** (***EX*** true) exprimée en utilisant des quantificateurs CTL suffit pour vérifier cette propriété afin de conclure quant à l'existence de blocage. Une autre solution est de spécifier les états de blocage et de vérifier que ceux-ci ne sont pas accessibles.

3 CHOIX D'UNE LOGIQUE TEMPORELLE

Nous avons vu lors de la présentation de classification des propriétés dynamiques que les logiques temporelles CTL et LTL ne permettent pas d'exprimer toutes les propriétés (Tableau 3).

Propriété	Logique temporelle LTL	Logique temporelle CTL
Sûreté	Exprimable	Exprimable
Vivacité	Exprimable	Exprimable
Atteignabilité	Ce type de propriété n'est exprimable que dans certains cas particuliers.	Il est possible de spécifier toutes les propriétés d'atteignabilité.
Equité	Exprimable	Non exprimable
Absence de blocage	Non exprimable	exprimable

Tableau 3. Pouvoir d'expression des logiques LTL et CTL.

La logique temporelle linéaire LTL s'avère plus naturelle en pratique pour spécifier les comportements attendus du système ou de l'environnement. De plus elle permet d'exprimer des notions utiles comme l'équité, les contre-exemples retournés sont simples (trace d'exécution) et ces logiques sont adaptées à la vérification. Par contre LTL manque parfois d'expressivité.

À l'inverse, la logique temporelle arborescente CTL s'avère parfois contre-intuitive (environnement, équivalence de modèle) et les contre-exemples retournés sont difficiles à interpréter (arbres d'exécution). CTL a cependant l'énorme avantage d'avoir des algorithmes de model checking polynomiaux, alors que ceux de LTL sont PSPACE donc exponentiel en pratique. CTL peut aussi exprimer certaines propriétés utiles absentes de LTL et l'équité peut s'obtenir avec des algorithmes ad hoc.

L'une des façons raisonnables d'utiliser des model checkers semble être d'utiliser CTL pour vérifier les propriétés les plus simples (sûreté) sur tout le modèle et d'en éprouver la validité ; puis après que quelques bugs grossiers aient été trouvés et que le modèle soit validé, utiliser un model checker LTL sur une partie seulement du système. Le Tableau 4 donne quelques résultats de complexité sur les différentes logiques, et la Figure 14 indique les relations d'inclusion entre les logiques LTL, CTL et CTL∗ (Bardin, 2008) (Emerson et Sistla, 1984).

	CTL	LTL	CTL*
MC	P-complet	PSPACE-complet	PSPACE-complet
MC concurrent	PSPACE-complet	PSPACE-complet	PSPACE-complet
MC open	EXPTIME-complet	PSPACE-complet	2-EXPTIME-complet
satisfiablité	EXPTIME-complet	PSPACE-complet	2-EXPTIME-complet

Tableau 4. Complexité des logiques temporelles.

Le Tableau 4, résume pour les trois types de logique temporelle (CTL, LTL, CTL*), leurs complexités par rapport aux critères suivants (MC pour model checking):

- MC : est-ce que $M \vDash \varphi$?
- MC concurrent : est-ce que $M_1, \ldots, M_n \vDash \varphi$?

- MC open : est-ce que $M, E \vDash \varphi$ pour tout E tel que $\vDash \psi$, où ψ est la contrainte d'environnement ?
- Satisfaisabilité : est-ce qu'il existe M tel que $M \vDash \varphi$?

Le M représente le modèle du système à vérifier, φ la propriété en logique temporelle.

Les travaux de recherche menés dans le cadre de cette thèse sont basés sur l'utilisation du model checker NuSMV qui utilise les deux types de logique temporelle, à savoir la logique temporelle linéaire et arborescente.

4 EXTENSION DE LA LOGIQUE TEMPORELLE.

Il existe de nombreux travaux qui essayent d'étendre la logique temporelle LTL et CTL. (Markey, 2003) (Laroussinie et al., 2002) (Gabbay et al., 1989) on introduit des opérateurs du passé dans la logique temporelle linéaire LTL. Si la plupart des logiques temporelles utilisées en model checking pour spécifier les propriétés des systèmes sont des logiques pures futures, l'utilisation d'opérateurs du passé rend les spécifications plus simples et plus naturelles. De nouveaux travaux comme (Gabbay et al., 1980) prouvent que les opérateurs du passé n'ajoutent pas d'expressivité, mais le prix de leur élimination est élevé, causant une explosion non élémentaire sur la taille des formules considérées (Lichtenstein et al., 1985). Plus tard on a prouvé dans les travaux de (Laroussinie et al., 2002) que PLTL est exponentiellement, mais plus condensé que LTL.

Dans (Markey, 2003), l'auteur présente plusieurs extensions de la logique temporelle. La logique temporelle NLTL est une logique temporelle linéaire avec l'opérateur *N* (appelé *now*), la logique NLTL est définie en ajoutant cet opérateur à la logique PLTL. De la même manière que l'ajout des modalités du passé dans la logique LTL, l'ajout de cet opérateur permet de simplifier l'expression de certaines propriétés. La logique NLTL a le même pouvoir d'expression que la logique LTL, c'est-à-dire que toute formule de NLTL peut être traduite en une formule de LTL équivalente. Par contre, en ce qui concerne les problèmes de vérification, leur complexité est exponentiellement plus grande pour la logique NLTL que pour les logiques PLTL ou LTL.

La logique CTL, en particulier, a l'avantage d'être facilement vérifiable : le problème du model checking de cette logique est en effet P-complet (voir tableau 4). Cela a valu à cette logique un succès incontestable dans les applications pratiques. De nombreuses extensions ont été définies à partir de CTL, par exemple CTL+ (Emerson et Halpern, 1985), ECTL+ (Emerson et Halpern, 1986) ou FCTL (Emerson et Lei, 1987).

La logique CTL*, proposée par (Emerson et Halpern, 1986), englobe à la fois CTL et LTL. Par rapport à CTL, elle autorise à écrire, dans la portée d'un quantificateur de chemin, une formule quelconque de LTL. Par rapport à LTL, elle permet de mettre des quantifications sur les chemins. La logique CTL* perd l'intérêt principal de CTL du point de vue de la vérification : la complexité du model checking devient PSPACE-complet, alors qu'elle était polynomiale pour CTL. Par rapport à CTL, la logique CTL+ autorise de faire des combinaisons booléennes des modalités X et U dans la portée d'un quantificateur de chemin.

(Emerson et Halpern, 1986) a défini la logique ECTL. Cette logique est définie de la même façon que CTL, mais avec une modalité en plus. Cette logique est strictement plus expressive que CTL. Cependant [EH86] prouve que cette logique n'est pas encore assez expressive : l'introduction de la modalité $\overset{\infty}{F}$ a pour but, en pratique, de vérifier des propriétés le long de chemins vérifiant des propriétés « d'équité ». Dans cette optique [EH86] définit également une extension de ECTL, appelée ECTL+, qui combine les avantages de ECTL et de CTL+.

Les logiques FCTL et GFCTL sont définies dans le but d'imposer des conditions d'équité aux chemins considérés. Ces deux logiques sont incluses, syntaxiquement, dans la logique ECTL+ parce que du point de vue du model checking, elles ont la même complexité que ECTL+.

(Brun, 1998) avait développé XTL (*eXtended Temporal Logic*) pour éviter la plupart des limitations des logiques temporelles classiques et intégrer plus que les constructions de flots de contrôle disponibles dans tous les langages classiques de programmation. Comparé aux autres logiques temporelles et aux formalismes dédiés à la spécification de systèmes interactifs, comme les machines à états finis (Harel, 1988) ou les ICOs (*Interactive Cooperative Objects*) (Palanque, 1992), XTL a été construit pour permettre de traiter des problèmes rarement pris en compte. En particulier, XTL est adapté à la spécification de problèmes temporels d'un point de vue aussi bien qualitatif que quantitatif.

5 LOGIQUE TEMPORELLE ET AUTOMATES

Pour la vérification de formules en logique temporelle, une structure de Kripke est nécessaire pour représenter le comportement d'un système. Une structure de Kripke est une sorte d'automate fini non déterministe : elle est représentée sous la forme d'un graphe orienté dont les nœuds correspondent aux états accessibles du système et dont les arcs représentent les transitions entre les états. Les logiques temporelles sont généralement interprétées dans des structures de Kripke.

5.1 VERIFICATION DE FORMULE CTL

La vérification de CTL consiste à étiqueter les états d'un système de transitions par les formules qu'il satisfait. On peut utiliser des automates pour le model checking de CTL et CTL*. On utilise alors des automates d'arbres infinis, et là encore, des automates alternants permettent des complexités optimales. On peut aussi définir des logiques arborescentes à base d'automates d'arbres. Cependant ces travaux ont un intérêt plutôt théorique, car CTL* est déjà suffisamment expressive et reste peu utilisée en pratique tandis que CTL possède des algorithmes optimisés beaucoup plus efficaces.

Par exemple pour la formule ¬EX *p*, on procède on trois phases :

1. on marque les états Q_p vérifiant *p*.
2. on marque les états Q_{EXp} vérifiant EX *p*, ce sont ceux dont un successeur par $\longrightarrow$ est dans Q_p.
3. on marque les états Q_φ vérifiant ¬EX *p*, ce sont ceux qui ne sont pas dans Q_{Xp}.

Pour limiter le nombre de cas à traiter dans la vérification de formule, on se restreint aux connecteurs de base *p*, ∧, ¬, EX, EU et AU. Avant d'appliquer la vérification, on passera donc d'abord par une phase de traduction de la formule φ.

Le model checking d'une formule CTL, c'est-à-dire « a-t-on A_m satisfaisant φ ? » peut être résolu en temps $O(|A_m| \times |\varphi|)$ ou A_m représente le modèle du système et φ est la formule CTL.

5.2 VERIFICATION DE FORMULE LTL

Comme on l'a vu dans le chapitre précédent, les formules en logique temporelle sont traduites en automates. Les formules en logique temporelle linéaire doivent être validées sur des exécutions. Une exécution est une suite d'états reliés deux à deux par des transitions. Une exécution est finie si le nombre d'états la constituant est fini et elle est infinie dans le cas contraire. Un automate fini peut donner lieu à des exécutions infinies dans le cas où l'exécution en question parcourt infiniment un nombre fini d'états.

Le modèle M d'un système à vérifier est vu comme un automate A_m reconnaissant des mots de longueur infinie (on parle d'w-mots et d'w-automates). Les lettres de ces w-mots correspondent chacune à une configuration du système. Un w-mot représente alors une séquence d'exécution du système qui traverse chacune de ces configurations. Le langage de l'automate (noté $L(A_m)$) est l'ensemble des w-mots qu'il reconnaît ; il représente l'ensemble des comportements possibles du système. Par ailleurs, la propriété comportementale φ que l'on veut vérifier sur M est elle-même exprimée par un w-automate $A\neg\varphi$ dont le langage est l'ensemble des comportements qui invalident la propriété φ. Un enchaînement de deux opérations permet de déterminer si $L(A_M)$ et $L(A\neg\varphi)$ partagent un w-mot, c'est-à-dire s'il existe une exécution de M qui ne vérifie pas φ. D'abord le produit synchronisé des deux automates est construit, il s'agit d'un w-automate reconnaissant l'intersection des deux langages.

Enfin un *emptiness-check* de ce produit est effectué, cette opération détermine si le langage reconnu par un automate est vide. Le langage du produit est vide lorsqu'il n'existe aucune séquence d'exécution de M qui invalide φ. Dans le cas contraire un contre-exemple, c'est-à-dire un w-mot représentant une exécution du système interdite par φ, peut être retourné.

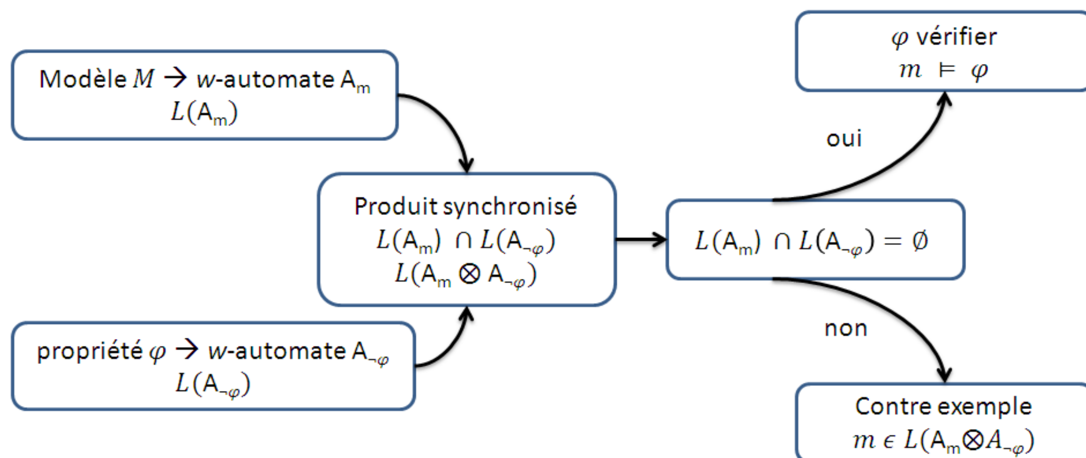


Figure 27. Vérification de formule LTL.

L'emptiness check. Le problème de satisfaction d'un automate se réduit donc à la recherche d'une composante fortement connexe qui soit accessible depuis l'état initial, et qui vérifie les conditions d'acceptation de l'automate. Les méthodes de recherche de composantes fortement connexes peuvent être séparées en deux classes : les approches globales, qui travaillent sur

l'automate dans sa totalité, et les approches à la volée qui sont combinées à la construction de l'automate synchronisé. Le but de ces dernières approches est de ne construire que la partie utile de l'automate (la construction est abandonnée dès qu'un contre-exemple est obtenu).

L'algorithme de Tarjan (Tarjan, 1972) représente l'approche traditionnelle utilisée en recherche de composantes fortement connexes. Il s'agit d'un algorithme de recherche en profondeur, linéaire par rapport à la taille de l'automate. Cet algorithme permet d'énumérer les composantes fortement connexes maximales (au sens de l'inclusion) d'un graphe en effectuant une simple recherche en profondeur.

Une variation de cet algorithme « Couvreur » est proposée dans (Couvreur, 1999) adaptée au model checking, car s'arrêtant dès qu'une composante fortement connexe (pas nécessairement maximale) accessible est trouvée. Les composantes fortement connexes maximales non acceptables peuvent être supprimées de la mémoire. Cela rend la méthode utile dans une vérification à la volée : l'automate produit est construit au fur et à mesure des besoins de l'algorithme de recherche de composantes fortement connexes, et les morceaux de l'automate qui se révèlent être des composantes mortes sont détruits progressivement. La seule information importante à conserver à propos de ces états est le fait qu'ils ont été visités. Ceci peut se faire avec une table de hachage.

Cette approche permet de manipuler des ensembles infinis (les langages) en les représentants par des structures finies (les w-automates). Plusieurs types d'w-automates existent et se distinguent par la forme de leurs conditions d'acceptation, c'est-à-dire la façon d'indiquer quels chemins infinis de l'automate correspondent à des w-mots acceptés. Les plus couramment utilisés, sont appelés automates de Büchi. Cette approche permet de vérifier toute propriété exprimable par un automate de Büchi. Cependant, spécifier des propriétés directement sous cette forme n'est pas toujours aisé. On a recours à des logiques comme la logique temporelle linéaire (LTL) (Vardi, 1996) dont la sémantique est simple et dont on sait transformer les énoncés en automates.

La complexité dans le pire des cas, $A_{\neg\varphi}$ est de taille $O(2^{|\varphi|})$. Le produit synchronisé $A_{\neg\varphi} \otimes A_m$ a une taille en $O(|A_m| \times |A_{\neg\varphi}|)$. Le model checking d'une formule LTL, c'est-à-dire « a-t-on A_m satisfait φ ? » peut être résolu en temps $O(|A_m| \times 2^{|\varphi|})$.

6 LES REQUETES EN LOGIQUE TEMPORELLE

Le model checking a été proposé comme une vérification technique (Clarke et al., 1986), il nous est précieux pour la compréhension du modèle : l'utilisateur émet une hypothèse du comportement du système, il l'exprime comme une formule en logique temporelle, et tente d'utiliser le model checker pour valider cette hypothèse. Le processus est réitéré pendant que l'utilisateur prend connaissance du système. William Chan, dans (Chan, 2000), pense que cette utilisation du model checking n'a pas été assez soulignée dans la littérature. Pour aider l'utilisateur à comprendre le comportement du système, Chan a introduit les requêtes en logique temporelle et utilise une technique similaire au model checking pour déterminer les propriétés temporelles plutôt que de simplement les contrôler.

Le Query Checking est une extension du model checking, qui au lieu de demander « est-ce que le système satisfait une formule en logique temporelle φ ? », nous permet de demander « pour

quelle valeur de X le système satisfait φ (X) ». La valeur X n'est pas un paramètre du système, mais un paramètre de la propriété, que l'on cherche à remplir. Ces requêtes permettent ainsi non pas de vérifier une propriété spécifique du modèle, mais de prendre connaissance du modèle en le questionnant. La technique du Query Checking peut aussi être utilisée pour fournir plus d'informations à l'utilisateur du model checking. Supposons que nous voulons vérifier si $AG\ (x \vee y)$ est juste. Nous pouvons évaluer la requête AG ?. Supposons ensuite que l'on obtienne $AG\ (x \wedge y)$ comme une formule déduite. Notez que cette formule est plus forte que celle que nous avons voulu vérifier. Non seulement nous pouvons conclure que $AG\ (x \vee y)$ est juste, mais on peut aussi informer l'utilisateur de la propriété la plus forte. En outre, dans le cas d'une formule non vérifiée ou fausse, en dehors de l'obtention d'un contre-exemple, l'utilisateur peut poser des questions en vue d'obtenir plus d'informations. Par exemple, si $AG\ (req \ \rightarrow AF\ ack)$ est fausse, si une demande n'est pas toujours suivie d'un accusé de réception, on peut se demander ce qui peut garantir un accusé de réception, $AG\ (? \rightarrow AF\ ack)$.

Nous souhaitons prolonger les travaux innovants de William Chan sur les requêtes en logique temporelle dans le domaine du web sémantique. Le but final des travaux présentés dans cette thèse est d'étendre ces requêtes en langage d'interrogation plus expressive sur des modèles de graphes sémantiques. Nous essayons aussi d'étendre ces travaux à la logique temporelle linéaire « LTL » car les requêtes de Query Checking existent seulement en logique temporelle arborescente « CTL».

Définition 1. Une requête en logique temporelle $\boldsymbol{\gamma}$ est une formule en logique temporelle dans laquelle le symbole « ? », appelé « *palceholder* » (que l'on appellera *joker* dans la suite), apparaît à la place d'une formule propositionnelle.

Définition 2. Une requête est positive (respectivement négative) si le joker apparaît sous un nombre pair (respectivement impair) de négations. Par exemple, la requête en logique temporelle « ? → alarme » est négative parce qu'elle est une abréviation de la requête « ¬? ∨ alarme ».

6.1 TYPES DE REQUETE

Dans cette section, nous passons en revues les requêtes en logique temporelle en utilisant le Query Checking. Dans les travaux de Chan, les requêtes comportent au plus un seul *joker*, alors que les travaux de (Gurfinkel et al., 2003) introduisent des requêtes avec plusieurs *joker*.

Définition 3. Une requête en logique temporelle avec un seul *joker* $?_1$, notée $\varphi[?_1]$, est une expression contenant un symbole $?_1$, où le remplacement $?_1$ par une formule propositionnelle vérifie la formule CTL.

Notez que cette définition permet de multiples occurrences du même *joker* dans la requête. Par exemple, à la fois AG $?_1$, $?_1 \wedge$ AX $?_1$ sont les requêtes valides avec un *joker* unique $?_1$.

Plusieurs *jokers* peuvent être nécessaires afin d'exprimer certaines propriétés d'intérêt. Dans cette section, nous donnons une extension des requêtes en logique temporelle qui permet d'avoir de multiple *joker*, où chaque *joker* peut dépendre d'un ensemble différent de variables propositionnelles.

Une substitution d'une requête avec de multiples *jokers* est un *tuple* de formules propositionnelles, un pour chaque *joker*. Étant donné une requête $\varphi[\ ?_1, \ldots, ?_n]$ avec n *joker*s, avec

PF($?_i$) étant le treillis de formules propositionnelles pour le $_i$ème *joker*, l'ensemble de toutes les substitutions possibles est le produit vectoriel L= PF($?_1$) x … x PF($?_n$).

Définition 4. Une requête γ est mixte si elle n'est ni positive ni négative.

Une requête avec plusieurs *jokers* peut être positive, négative ou mixte dans un *joker* donné. En outre, une requête est positive (négative, monotone) si elle est positive (négative, monotone) dans tous les *joker*s. Les requêtes qui ne sont pas monotones sont considérées comme des non monotones ou mixtes. Par exemple, AG ($?_1 \wedge$ AX $?_2$) est positif (et, par conséquent, monotone), tandis que AG ($?_1 \Longrightarrow$ AG ($?_2 \Longrightarrow$ AX $?_1$)) est mixte.

	Type	Exemple
Un seul joker		
1	Positive, valide	***AG*** $?_1$
2	Positive	***EF*** $?_1$
3	Positive, plusieurs occurrences	***EF*** ($?_1 \wedge$ ***EX*** $?_1$)
4	Négative, valide	***AG*** $\neg ?_1$
5	Négative	***EF*** $\neg ?_1$
6	Négative, plusieurs occurrences	***EF*** ($\neg ?_1 \wedge$***EX*** $\neg ?_1$)
7	Mixte	***AG*** ($?_1 \wedge$ r) $\vee$ ***EF*** ($\neg ?_1$)
Plusieurs jokers		
8	Positive	***AG*** ($?_1 \wedge$ ***AX*** $?_2$)
9	Négative	***EF*** ($\neg ?_1 \wedge$ ***EX*** $\neg ?_2$)
10	Monotone	***AG*** ($?_1 \Rightarrow$ ***AF*** $?_2$)
11	Mixte	***EF*** ($?_1 \Rightarrow$ ***AG*** ($?_2 \Rightarrow$ ***AX*** $?_1$)

Tableau 5. Les différents types de requêtés. (Gurfinkel et al., 2003)

La Tableau 5 résume les différents types de requêtes en logique temporelle. Par exemple, les requêtes valides positives avec un seul *joker* peuvent être résolues à la fois par les approches de Chan et Burns & Godefroid, tandis que les requêtes négatives avec plusieurs occurrences et mixtes ne peuvent être résolues en utilisant ces dernières approches. Ses requêtes et les requêtes avec plusieurs *joker*s sont résolues en utilisant l'approche introduite dans (Gurfinkel et al., 2003). Ces différentes approches seront abordées dans la section suivante.

6.2 RESOLUTION DES REQUETES

Le domaine des requêtes en logique temporelle est une nouvelle extension du model checking. Il y a peu de travaux le concernant et c'est aussi pour cette raison qu'il y a peu d'algorithmes de résolution. Le model checking permet de tester si une requête donnée est ou n'est pas satisfaite par le modèle que l'on « analyse ». Mais, par exemple, remarquant que la propriété $\phi = AG(erreur \longrightarrow F\ alarme)$ n'est pas vérifiée, nous aimerions savoir quelle formule, à la place de « erreur », rendrait la formule ϕ vraie pour le modèle que l'on étudie. Autrement dit, nous voudrions pouvoir répondre à la question $\phi = AG(? \longrightarrow F\ alarme)$ (qu'est-ce qui déclenche une alarme ?). Nous dirons que nous voulons trouver la ou les solutions de la requête γ, où les «solutions» sont des formules propositionnelles. C'est l'objet du *Query-Checking*, introduit par Chan (Chan, 2000). Dans cette section, je présente les différents algorithmes proposés jusqu'à présent pour la résolution des requêtes en logique temporelle.

Les travaux de Chan portent uniquement sur la logique temporelle arborescente CTL. Il définit la notion de solution « exacte » et s'intéresse seulement aux requêtes dont les solutions sont exactes dans tous les modèles. Ces requêtes sont appelées requêtes « valides ». Il montre que décider si une requête CTL est valide est un problème difficile et propose donc un fragment syntaxique de CTL nommée CTL_V, dont les requêtes sont valides. Enfin il donne pour CTL_V un algorithme bilinéaire de résolution du problème du Query-Checking. Il y a peu d'approches pour la résolution du problème du Query Checking. Cette section présente les différentes approches existantes, toutes basées sur l'approche de Chan.

Il y a peu de travaux de recherche sur l'utilisation de la logique temporelle pour l'interrogation de systèmes. La requête en logique temporelle a été introduite par William Chan (Chan, 2000) dans le but d'accélérer la compréhension de conception par la découverte de propriétés inconnues. Les travaux (Gurfinkel et al., 2002) (Gurfinkel et al., 2003) montrent que le Query Checking est applicable à une variété de tâches d'exploration du modèle. Ils l'illustrent à l'aide d'un CCS (*System Cruise Control*), un système qui est responsable de la régulation de la vitesse d'une automobile se déplaçant à une vitesse donnée. Dans leur étude, l'outil qu'ils ont créé recherche toutes les formules propositionnelles vérifiant la requête en logique temporelle, tandis que dans nos travaux de recherche, notre outil recherche uniquement les états qui vérifient la requête en logique temporelle. (Hornus et Schnoebelen, 2002) étudie le problème du calcul de toutes les solutions minimales pour n'importe quelles requêtes temporelles sur n'importe quelles structures de Kripke et dans (Gheorghiu et Gurfinkel, 2006), les auteurs présentent un outil qui calcule des solutions aux requêtes CTL.

Dans l'approche de Chan, une requête en logique temporelle ne peut avoir qu'un seul *joker* tandis que dans (Gurfinkel et al., 2002) (Gurfinkel et al., 2003) (Hornus et Schnoebelen, 2002) (Gheorghiu et Gurfinkel, 2006), elle peut avoir de multiples *joker*s. Les requêtes en logiques temporelles peuvent avoir une représentation unique, en utilisant des opérateurs de la logique arborescente, tandis que les travaux de recherche dans (Hornus, 2001) (Hornus et Schnoebelen, 2002) tentent d'étendre le travail de Chan à d'autres logiques temporelles, comme CTL*, qui comprend la logique temporelle linéaire.

Dans les travaux de recherche présentée dans (Samer et Veith, 2003), les auteurs étudient de façon systématique le programme de Chan sur les langages de requêtes en logiques temporelles, et ont corrigé quelques défauts pour définir une base approfondie pour de futurs travaux. Parmi ceux-ci, ils ont défini deux fragments CTL^D et CTL_v *new*. Dans cet article (Samer et Veith, 2003), ils prouvent qu'une requête CTL^v avec une structure de kripke appropriée, de telle sorte que la requête n'a pas une solution exacte. Cela contredit la revendication principale de Chan. Pour nos travaux, nous considérons ceux de Chan comme une contribution très intéressante et importante (Chan, 2000). Ainsi, il est important de donner une base solide pour obtenir une meilleure compréhension des requêtes en logiques temporelles. Basés sur l'idée obtenue par le contre-exemple, ils définissent deux nouveaux langages de requête :

- CTL^D généralise l'approche de Chan, elle garantit une solution exacte, mais seulement dans les cas où l'ensemble des solutions n'est pas vide. Ce type de requete ajoute de nouveau opérateur.

- CTL^{v} *new* (un fragment syntaxique de LTL) est le sous-ensemble de toutes les requêtes où CTL^{D} nécessite une solution dans chaque structure de Kripke. Ainsi, CTL^{v} *new* répond à des critères originaux par rapport à l'approche de Chan.

(Zhang et Cleaveland, 2005) développe un framework pour la résolution de requête en logique temporelle pour une classe de modèles de systèmes avec des états infinis qui calculent avec des variables à valeurs entières (appelés systèmes de Presburger, dans lesquelles des formules de Presburger sont utilisées pour la modélisation du comportement du système). Cette méthode est basée sur la technique du model checking symbolique qui repose sur la recherche de preuves. La structure de données principale est un système d'équations de prédicats, qui est un codage du produit d'un modèle du système et de la formule de la requête. Les preuves de la validité de l'équation de prédicat sont construites en utilisant les règles de preuve ; des solutions à un *joker* sont déduites des feuilles d'un arbre de preuve afin d'assurer que la preuve obtenue est valable. Ce système de résolution fonctionne avec des *joker*s multiples et des *joker*s avec des événements à la fois positifs et négatifs.

6.2.1 Approche de Chan

Les travaux de Chan (Chan, 2000) portent uniquement sur la logique CTL. Trouver une solution arbitraire à une requête n'est pas très intéressant, trouver toutes les solutions ne semble pas être utile non plus, car il y a des chances d'avoir un trop grand nombre d'entre elles. Au lieu de cela, Chan définit la notion de solution exacte (à partir de laquelle toutes les autres solutions peuvent être déduites) et s'intéresse seulement aux requêtes dont les solutions sont exactes dans tous les modèles. Ces requêtes sont appelées requêtes valides. Chan propose donc un fragment syntaxique de CTL, CTL$_{v}$ dont les requêtes sont valides.

Il y a deux principaux cas dans lesquels une requête n'est pas distributive sur la conjonction. Le premier cas est lorsque le *joker* apparaît dans le champ d'un opérateur temporel qui se trouve sous un nombre impair de négations, tel que AG?. Ces requêtes sont préoccupées par ce qui se passe sur certains chemins. Si $\emptyset_1$ et $\emptyset_2$ sont vérifiés sur certains chemins, nous ne savons pas si $\emptyset_1 \wedge \emptyset_2$ est vrai sur un chemin. (Nous savons que $\emptyset_1 \vee \emptyset_2$ est vrai sur certains chemins, mais ce n'est pas suffisant). Le second cas est lorsque le *joker* apparaît sur le côté droit des opérateurs « until », par exemple, AF? ($F\emptyset$ est équivalent à$\top\ U\ \emptyset$). Ces requêtes demandent ce qui finira par arriver. Même si $\emptyset_1$ et $\emptyset_2$ sont éventuellement vrais, ils ne peuvent être vrais dans les mêmes états le long des chemins. Il existe de nombreuses exceptions au second cas, cependant, comme $A(\emptyset\ W\ \neg\emptyset\ \wedge ?)$ et AFAG?. La stratégie de cette approche est de définir une classe de requêtes qui exclut ces problèmes connus, tout en permettant des exceptions.

Pour cela, Chan définit deux nouveaux opérateurs «until» :

1. *Until disjoint faible* représenté par la lettre $\overline{\mathbf{W}}$ ($\emptyset\ \mathbf{W}\ \psi$ est l'équivalent de $(\phi\ \mathbf{U}\ \psi) \vee \boldsymbol{G}\phi$))
2. *Until disjoint fort* représenté par la lettre $\overline{\mathbf{U}}$

$$\boldsymbol{A}\left(\phi\ \overline{\mathbf{W}}\ \psi\right) \equiv \boldsymbol{A}(\emptyset\ \mathbf{W}\ \neg\emptyset\ \wedge\ \psi) \qquad \boldsymbol{A}\left(\phi\ \overline{\mathbf{U}}\ \psi\right) \equiv \boldsymbol{A}\ (\emptyset\ \mathbf{U}\ \neg\emptyset\ \wedge\ \psi)$$

Formellement, la classe de requêtes CTL^{v} est définie comme le plus petit ensemble de requêtes satisfaisant les contraintes suivantes:

- ? et ¬ ? sont des requêtes CTL^{v}.
- Si ∅ est une formule CTL et γ est une requête CTL^{v}, alors $\varphi \vee \gamma$, $\mathbf{AX\gamma}$, $\mathbf{A}(\gamma\ \mathring{w}\ \phi)$, et $\mathbf{A}(\phi\ \overline{\boldsymbol{W}}\gamma)$ sont aussi des requêtes CTL^{v}.
- Une requête persistance est aussi une requête CTL^{v}.

La classe des requêtes persistances est définie comme suit :

- Si γ est une requête, alors **AG** γ est une requête persistance.
- Si γ est une requête persistance et ϕ est une formule CTL, alors $\phi \vee \gamma$, $\mathbf{AX}\gamma$, $\mathrm{A}(\phi\ \mathbf{W}\ \gamma)$, et $\mathbf{A}(\phi\ \mathbf{U}\ \gamma)$ sont des requêtes persistance.

Remarque :

$\phi\ \mathring{w}\ \gamma \equiv\ \phi\ W\ (\phi \wedge \gamma)$
$\phi\ \mathring{U}\ \gamma \equiv\ \phi\ U\ (\phi \wedge \gamma)$

En d'autres termes, dans les requêtes CTL^{v}:

1. Les négations peuvent être appliquées seulement au *joker* ou aux formules CTL.
2. Le *joker* ne peut pas apparaître de chaque côté des opérateurs $\mathbf{\mathring{U}}$ ou $\overline{\mathbf{U}}$, du côté gauche de l'opérateur $\overline{\mathbf{W}}$ ou **U**, ou sur le côté droit de $\mathbf{\mathring{w}}$.
3. Si le *joker* apparaît sur le côté droit de **W** ou **U**, alors il doit être un *joker* entre l'opérateur **AG** et **U**.

La première restriction sur la négation est d'éviter l'interrogation existentielle sur les chemins. La seconde restriction sur les opérateurs « until » est de veiller à ce que nous n'ayons pas une obligation d'éventualité (qui ne peut être étendue dans tous les modèles). La troisième règle de restriction est de ne pas prendre les requêtes telles que AF? mais permettre des requêtes valides comme AFAG?.

Ces requêtes peuvent être résolues efficacement en appliquant pre∀ et post∃ où :

$$post\exists(S) = \{q' \in Q |\ \exists q. < q, q' > \in\ \Delta\ et\ q\ \in S\}$$

l'ensemble des successeurs des Etats dans S. Pour chaque formule CTL ϕ, l'ensemble $R\phi$ est défini comme :

$$R\phi = \mu Z.\left(\left(S\ \cup post\exists(Z)\right) \cap \left[[\emptyset]\right]\right)$$

ou l'ensemble des états accessibles à partir de S en passant par seulement les états qui satisfont ∅. La figure suivante prise de l'article de Chan, montre une procédure nommée « Solve » qui prend une requête CTL^{v} et un ensemble d'états, et retourne un ensemble d'états. Dans cette figure, $\boldsymbol{\gamma}$ est n'importe quelle requête CTL^{v}, $\boldsymbol{\phi}$ est n'importe quelle formule CTL, $\boldsymbol{S} \subseteq \boldsymbol{Q}$ et est n'importe quel ensemble d'état

$$
\begin{aligned}
Solve(?,\, S) &= S \\
Solve(\neg ?,\, S) &= Q \setminus S \\
Solve(\phi \vee \gamma,\, S) &= Solve(\gamma,\, S \setminus [\![\phi]\!]) \\
Solve(\mathsf{AX}\gamma,\, S) &= Solve(\gamma,\, post_{\exists}(S)) \\
Solve(\mathsf{A}(\gamma\, \mathring{\mathsf{W}}\, \phi),\, S) &= Solve(\gamma,\, S \cup R_{\neg\phi} \cup post_{\exists}(R_{\neg\phi})) \\
Solve(\mathsf{A}(\phi\, \overline{\mathsf{W}}\, \gamma),\, S) &= Solve(\gamma,\, (S \cup post_{\exists}(R_{\phi})) \setminus [\![\phi]\!]) \\
Solve(\mathsf{A}(\phi\, \mathsf{W}\, \gamma),\, S) &= Solve(\gamma,\, B) \\
&\quad \text{where } R = R_{\phi \wedge \neg\gamma(\bot)} \\
&\qquad\quad B = (S \cup post_{\exists}(R)) \setminus ([\![\phi]\!] \cup [\![\gamma(\bot)]\!]) \\
Solve(\mathsf{A}(\phi\, \mathsf{U}\, \gamma),\, S) &= Solve(\gamma,\, B \cup C) \\
&\quad \text{where } C = \nu Z.\ (R \cap post_{\exists}(Z)) \\
&\qquad\quad R \text{ and } B \text{ are the same as above}
\end{aligned}
$$

Figure 28. La résolution des requêtes CTL^{V}.

L'idée est que si γ est n'importe quelle requête CTL^{v}, M est un modèle avec un état initial Q_0, et S est le résultat de « Solve(γ, Q_0) », alors la fonction caractéristique de S est :

$$\bigvee_{q \in S} \left(\bigwedge_{x \in L(q)} x \wedge \bigwedge_{x \in X \setminus L(q)} \neg x \right)$$

est une solution exacte pour γ dans M. La procédure « Solve » s'exécute en temps linéaire de la taille du modèle et linéaire dans la longueur de la requête.

Simplification. Bien que la solution exacte donne des informations complètes, il est susceptible d'être trop complexe à comprendre. Chan propose une stratégie pour faire face au problème en décomposant une proposition dans un ensemble de conjonctions (pour les requêtes positives) ou disjointes (pour les requêtes négatives) à l'aide de projection et ainsi de ne pas se soucier de la minimisation. La décomposition n'est pas un problème nouveau dans le model checking symbolique, mais l'objectif habituel est de produire un petit nombre de petites conjonctions équilibrées ou disjointes afin de réduire le temps ou l'espace pour le calcul de point fixe. Le but, au contraire, consiste à décomposer une proposition dans un nombre potentiellement important de "simples" éléments.

Les requêtes négatives peuvent alors être traitées en utilisant la loi ensembliste de « De Morgan » pour la décomposition disjonctive. La décomposition conjonctive est une approximation conservatrice dans le sens où la conjonction obtenue peut être plus faible que la proposition donnée.

L'algorithme ci-dessous est proposé pour la décomposition conjonctive approximative. La méthode « atomes (*s*) » désigne l'ensemble des propositions atomiques qui apparaissent dans *s*. Avec *j* croissante jusqu'à un *k* donné, l'algorithme trouve des propositions non triviales qui sont plus faibles que *s* et contient seulement *j* propositions atomiques. Les informations redondantes dans le résultat sont réduites en simplifiant le candidat en conjonction avec l'aide d'autres conjonctions déjà calculées. L'algorithme fonctionne en temps exponentiel en *k*. Toutefois, ce n'est pas un problème sérieux dans la pratique, parce que le résultat sera trop compliqué à comprendre pour une valeur de *k* grande. Chan a utilisé seulement $k \leq 4$ pour son expérience préliminaire.

$$
\begin{array}{l}
\{\text{Input: proposition } s \\
\quad \text{and } k \text{ with } 0 < k \leq |atoms(s)| \; \} \\
\mathcal{C} := \emptyset \\
\text{for } j := 1 \text{ to } k \\
\quad \text{for each } Y \subseteq atoms(s) \text{ with } |Y| = j \\
\quad\quad r := (\exists \overline{Y} \,.\, s) \downarrow \mathcal{C} \\
\quad\quad \text{if } r \not\Leftrightarrow \mathit{true} \text{ and } r \not\Leftrightarrow s \\
\quad\quad\quad \mathcal{C} := \mathcal{C} \cup \{r\} \\
\quad\quad \text{fi} \\
\quad \text{end} \\
\text{end} \\
\{\text{Output: } \mathcal{C} \text{ with } s \Rightarrow \bigwedge \mathcal{C}\}
\end{array}
$$

Figure 29. La décomposition conjonctive approximative d'une proposition.

Pour réduire le nombre de résultats, avant de lancer l'algorithme de décomposition, il permet de projeter la proposition *s* sur l'ensemble des propositions atomiques qui intéresse l'utilisateur. Une façon de trouver ces propositions atomiques est d'examiner les formules en logiques temporelles à vérifier. Parfois, le nombre de propositions atomiques pertinentes pourrait paraître important, mais l'utilisateur ne peut vouloir tirer des propriétés d'une forme restreinte.

Application de la méthode chan. Chan a mis en pratique les requêtes en logiques temporelles à deux modèles de fichier SMV (type de fichier pour le model checker NuSMV) (McMillan, 1992) (McMillan, 1993). Il a constaté que la technique est particulièrement utile lorsque des propriétés inattendues sont déduites.

Une direction intéressante pour les travaux futurs est d'étendre les résultats à la logique temporelle linéaire (LTL). Toutes les définitions de cette approche peuvent s'étendent à LTL d'une manière simple. L'avantage des requêtes en LTL est le rajout de l'expressivité aux requêtes en logique temporelle arborescente (Bardin, 2008) (Emerson et Sistla, 1984) (voir section 3 et 4).

Chan a proposé un certain nombre d'applications. La plupart du temps ces applications visent à donner plus d'information à l'utilisateur lors de la vérification de modèle en fournissant une explication partielle lorsque la propriété est vraie et les informations de diagnostic quand elle ne l'est pas. Nous pouvons également utiliser le Query Checking pour recueillir des informations de diagnostic quand une formule CTL ne tient pas sur un chemin (n'est pas vraie).

6.2.2 Approche de Bruns & Godefroid

Cette approche (Bruns et Godefroid, 2001) simplifie le travail de Chan en montrant que le Query Checking peut être accompli en adaptant les algorithmes du model checking existant. En particulier, cette approche montre comment adapter l'approche théorique des automates au model checking de (Kupferman et al., 2000) pour résoudre le problème du Query Checking.

L'approche de Chan fonctionne seulement avec les requêtes exprimées avec des opérateurs de logique CTL, alors que cette approche est définie pour n'importe quelle logique temporelle. Inspiré par l'approche théorique des automates du model checking de (Kupferman et al., 2000). Étant donné une requête en logique temporelle Ø et une structure de Kripke K, nous construisons un automate

alternant représentant ∅, puis nous calculons le produit de cet automate avec k, et enfin nous vérifions si le langage accepté par l'automate produit est vide. Une étape clé dans le développement de cette approche est de découvrir une sorte d'automate alternant approprié pour représenter une requête en logique temporelle. Pour cela, un nouveau type d'automate alternant est utilisé et il s'appelle automate alternant étendu (EAA).

EAA généralise la notion d'une exécution en automate alternant. Une exécution d'un automate alternatif sur un arbre d'entrée est elle-même un arbre, avec chaque nœud de l'exécution marqué par un nœud de l'arbre d'entrée. Les étiquettes sur un nœud de l'exécution et ses enfants doivent satisfaire la fonction de transition de l'automate. Dans un EAA, chaque nœud de l'exécution est en outre marqué avec une valeur du treillis sous-jacente, et les valeurs d'un nœud de l'exécution, et ses enfants doivent satisfaire à nouveau la fonction de transition. Chaque exécution elle-même a une valeur, qui est la valeur d'étiquetage du nœud racine de l'exécution. Une exécution est acceptée si elle satisfait la condition d'acceptation de l'EAA et n'a pas de nœud étiqueté avec l'élément de fond du treillis. Dans un automate alternant standard, chaque nœud d'une exécution est implicitement étiqueté avec la valeur « true ».

Définition 5. Le treillis contient 2 à la puissance $2^{|P|}$nœuds, qui sont les solutions potentielles d'une requête dans une structure de kripke.

6.2.3 Approche de Schnoebelen

Le travail présenté dans (Hornus, 2001) (Hornus et Schnoebelen, 2002) est une extension des travaux de Chan pour la logique temporelle CTL* qui inclut la logique temporelle LTL. Ils présentent un algorithme relativement naïf de recherche de toutes les solutions minimales d'une requête. Cet algorithme parcourt le treillis en partant du haut (de la formule), et en maintenant deux ensembles. Un ensemble **SOL** de solution pseudo minimales qui sera à la fin de l'exécution l'ensemble complet des solutions minimales de γ dans S. Et un ensemble **UKN** des formules du treillis non encore examinées.

$f^{\downarrow}$: ensemble des formules propositionnelles à examiner.
SOL : $\min\{\emptyset, S \vDash \gamma(\emptyset)\}$
UKN : ensemble des formules non examinées.
Initialisation :
Algo_exhaustif(g) {
    Pour toute formule f telle que f ↪ g faire
    Si f ∈ UKN alors {
        Si $S \vDash \gamma(f)$ alors {
            SOL := min(SOL ∪ $\{f\}$)
            UKN :=UKN\ $f^{\uparrow}$
            Algo_exhaustif(f)
        }
        Sinon {
            UKN := UKN\ $f^{\downarrow}$
        }
    }
}

Script 10. Algorithme générique.

Rappelons qu'une **valuation** est une application qui donne une valeur de vérité à chaque proposition atomique.

Une amélioration de l'algorithme précèdent (script 10) a été faite pour restreindre l'ensemble des solutions et savoir si γ admet une unique solution minimale dans S. Cette nouvelle version permet de construire l'ensemble des **valuations** qui peuvent être enlevées (au moins une par une) de l'ensemble fs pour obtenir une solution plus petite. Cette particularité sera utile et permettra de calculer, en temps polynomial, si une requête admet une unique solution dans une structure de Kripke donnée.

Le calcul effectif de solution unique minimale peut se faire en temps $O((f))$ linéaire en fonction de la taille qu'occupe f en mémoire si f est représentée sous forme BDD.

Soit une structure de Kripke S et une requête γ. Nous supposons que $S\ satisfait\ \gamma(fs)$ (dans le cas contraire, γ n'admettrait aucune solution dans S).

Initialisation :
Suppose que $S \models \gamma(fs)$ et $S \nvDash \gamma\ (\bot)$
L : ensemble de valuations
Algo_unique()
{
    L :=algo_minimale(fs)
    Si **L** = ∅ alors
        return (la solution$fs\ est\ minimale\ et\ unique$)
    g := $[[fs]]$**L**
    Si $S \models \gamma(g)$ alors
        return (la solution g est minimale et unique)
    return (il y au moins deux solutions minimales)
}

Script 11. Algorithme générique partiel.

Les deux algorithmes présentés ci-dessus ont une complexité polynomiale. Le second algorithme utilise le premier pour déterminer si une instance du problème du Query Checking admet une unique solution minimale. Cet algorithme est très intéressant dans la mesure où il supprime la nécessité de définir des fragments syntaxiques de logiques temporelles dont les requêtes sont valides. Il généralise en fait la notion de requête valide en la portant localement à tous les modèles individuellement. Cet algorithme permet donc de résoudre en temps polynomial toute instance du problème où la requête admet dans le modèle considéré une unique solution minimale, indépendamment de la logique utilisée.

6.2.4 Approche de Chechik

Cette dernière approche (Gurfinkel et al., 2003) (Gurfinkel et al., 2002) (Chechik et Gurfinkel, 2003) (Gheorghiu et Gurfinkel, 2006) introduit la résolution des requêtes en logique temporelle avec de multiples *joker*s par rapport aux autres approches qui résolvent seulement les requêtes avec un seul *joker*. Dans le Query Checking, on a souvent envie de limiter les propositions atomiques qui sont présentes dans la solution. Étant donné un ensemble fixe de *k* propositions atomiques d'intérêt, le

problème de vérification de requêtes peut être résolu en prenant les 2^{2^k} formules propositionnelles de cet ensemble, vérifier la formule en logique temporelle, et de retourner ensuite l'ensemble de solutions les plus fortes (Chan, 2000) (Bruns et Godefroid, 2001). Cette approche se réfère à la limitation du nombre de résultats en limitant les propositions atomiques. Le nombre *k* de propositions d'intérêt fournit un moyen de contrôler la complexité du Query Checking dans la pratique, en termes de compréhension et en termes de calcul.

Le Query checking est très complexe à résoudre, Chan se concentre sur les requêtes valides, c'est-à-dire des requêtes où toutes les solutions sont déductibles à partir d'une solution unique. La complexité des requêtes valides sur des propositions atomiques *k* est exponentielle en *k*. Chan a montré que, en général, il est coûteux pour déterminer si une requête CTL donnée est valide. Au lieu de cela, il a identifié une série de restrictions syntaxiques qui garantissent que la requête CTL résultante est valide. Il a également mis en place un vérificateur de requête pour les requêtes valides avec un unique *joker*, en utilisant le model checker SMV.

D'autre part, de nombreuses requêtes ne sont pas valides. De toute évidence, de telles requêtes pourraient être utiles pour l'exploration du modèle. (Bruns et Godefroid, 2001) fournissent un mécanisme pour calculer toutes les solutions à n'importe quelle requête avec un unique *joker* (requête positive ou négative), en utilisant des automates alternatifs étendus (Bruns et Godefroid, 2001). (Hornus et Schnoebelen, 2002) étudièrent le problème de la production efficace de certaines des solutions pour les requêtes positives avec un unique *joker*. Leur algorithme calcule une solution en utilisant un nombre linéaire d'appels au model checker, deux solutions utilisant un nombre quadratique d'appels au model checker. Dans les deux méthodes ci-dessus, les requêtes peuvent être spécifiées dans les logiques temporelles autres que CTL, mais à notre connaissance, aucune des méthodes n'a été mis en œuvre.

Le Query Checking peut être étendu plus loin si nous ne limitons pas les requêtes à un seul *joker*. En particulier, les requêtes avec deux *joker*s nous permettent de questionner le modèle sur des paires d'états, c'est ce que cette approche essaye d'introduire. Cette approche est similaire à celle proposée par (Bruns et Godefroid, 2001) où le problème du Query Checking est décidé à l'aide d'automates alternatifs étendus.

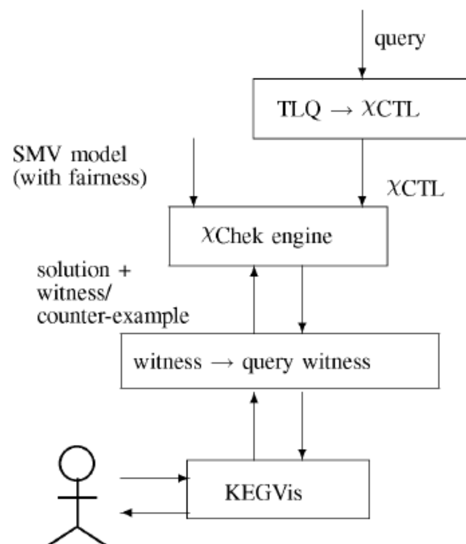


Figure 30. Architecture de l'outil TLQSolver

Étant donné une requête en logique temporelle et un modèle écrit en langage SMV, l'outil TLQSolver (figure 30) convertit d'abord la requête dans une formule χCTL correspondante. Le modèle et la formule qui en résulte sont ensuite transmis à χChek. Enfin, la réponse et une preuve renvoyée par χChek sont transformées à partir de plusieurs valeurs logiques dans des ensembles de formules propositionnelles et présentées à l'utilisateur par le biais KEGVis. KEGVis présente les preuves sous forme graphique en utilisant le *daVinci presenter* (Fröhlich et Werner, 1994) pour la présentation et l'exploration. En plus de l'exploration de preuves, l'utilisateur peut définir un certain nombre de stratégies pour leur exploration (Gurfinkel et Chechik, 2003) : le choix des preuves est basé sur la taille, le facteur de branchement, etc.

La réduction à χCTL est définie de haut en bas. Tous les opérateurs, les constantes et les variables propositionnelles sont converties un à-un : $\varphi \lor \psi$ est mappé à la disjonction de la traduction de φ avec la traduction de ψ, toute variable p est mappée à elle-même ; vrai est mappé sur le symbole de constante $\top$; et ainsi de suite. Pour la traduction du *joker*, il faudra analyser toutes les propositions atomiques de l'état avec les propositions atomiques demandées dans le *joker*.

Application de la méthode. Les exemples de cette approche sont basés sur l'exploration d'une spécification d'un système de contrôle de vitesse. Le système de contrôle de vitesse (CCS) est responsable de la tenue d'une vitesse constante d'une automobile se déplaçant à une certaine vitesse.

7 CONCLUSION

La logique temporelle est une logique utilisée par les model checkers, afin de spécifier les comportements attendus du système à vérifier. Il y deux principaux types de logique temporelle : la logique temporelle linéaire et la logique temporelle arborescente. Il existe de nombreux travaux qui essayent d'étendre ces deux types de logique temporelle, on introduisant des opérateurs du passé dans la logique temporelle linéaire LTL et d'autre logique plus expressive que CTL.

CTL est une logique pour vérifier des propriétés simples (sûreté) sur tout le modèle et d'en éprouver la validité ; puis après que des erreurs ont été trouvées et que le modèle soit validé, la logique LTL peut être utilisé sur une partie seulement du système.

Les requêtes en logique temporelle (le *Query Checking*) introduite par (Chan, 2000), a pour but d'accélérer la compréhension du comportement du système à vérifier. Les requêtes en logique temporelle utilisent une technique similaire au model checking pour déterminer les propriétés temporelles plutôt que de simplement les contrôler. Nous pouvons avoir une requête avec un seul ou plusieurs *jokers*.

Il y a peu de travaux concernant la résolution de ces requêtes et c'est aussi pour cette raison qu'il y a peu d'algorithmes de résolution. Les travaux de Chan (Chan, 2000) portent uniquement sur la logique CTL. L'approche de Bruns & Godefroid simplifie le travail de Chan en montrant que le Query Checking peut être accompli en adaptant les algorithmes du model checking existant. Elle est définie pour n'importe quelle logique temporelle. L'approche de Schnoebelen est une extension des travaux de Chan pour la logique temporelle CTL* qui inclut la logique temporelle LTL. L'approche de Chechik introduit la résolution des requêtes en logique temporelle avec un ou plusieurs jokers par rapport aux autres approches qui résolvent seulement les requêtes avec un seul joker.

Les différentes approches de résolution proposée dans la section 6 sont basées sur l'approche de Chan qui est le pionnier du problème du Query Checking.

On a utilisé la logique temporelle pour spécifier les comportements attendus de nos graphes sémantiques. Ce chapitre présente une première étude sur la logique temporelle. Cette étude a pour objectif d'introduire les grandes notions liées à la logique temporelle et de dégager des pistes de réflexion pour l'application de la logique temporelle à la qualification de graphe sémantique.

Les requêtes en logique temporelle sont utilisées pour interroger nos modèles de graphe sémantique. Les requêtes qu'on a introduites peuvent comporter un ou plusieurs jokers.

Nous souhaitons prolonger les travaux innovants de William Chan sur les requêtes en logique temporelle dans le domaine du web sémantique. Le but final des travaux présentés dans cette thèse est d'étendre ces requêtes en langage d'interrogation plus expressive sur des modèles de graphes sémantiques. Nous essayons aussi d'étendre ces travaux à la logique temporelle linéaire LTL.

Chapitre 5

Qualification de graphes sémantiques

Résumé

L'utilisation croissante des graphes sémantiques et le coût des modifications appuient la nécessité de gérer leur évolution. Une erreur recherchée couramment dans les graphes sémantiques est la résolution de contradictions logiques appelées incohérences et inconsistances. Dans ce chapitre, nous proposons une nouvelle approche pour la vérification de ces différentes contradictions dans les graphes sémantiques. Cette qualification ne se résume pas seulement à la vérification des incohérences dans les ontologies, mais peut également servir à l'interrogation (chapitre 6) de ces modèles afin d'étendre l'expressivité des requêtes SPARQL et une simplicité de l'écriture de ces requêtes. Les domaines d'applications peuvent être nombreux : par exemple le domaine du bâtiment nécessite une vérification de ces graphes sémantiques (vérification des relations entre les différents éléments du bâtiment).

Plan

1 La qualification 113
1.1 Les étapes de transformation 115
1.1.1 Exploration du graphe 115
1.1.2 Détermination d'une racine 116
1.1.3 Génération du modèle 117
1.1.4 RDF2NuSMV 118
1.1.5 RDF2SPIN 120
1.2 La vérification 124
1.3 Comparaison entre RDF2SPIN et RDF2NuSMV 124
2 Cas d'utilisation 126
3 Travaux Connexes 130
4 Conclusion 132

Le Web sémantique vise à organiser et structurer l'énorme quantité d'informations présentes sur le Web. La perspective du Web sémantique est d'enrichir le contenu du Web actuel avec des descriptions formelles de sorte que les agents logiciels puissent être en mesure de traiter les informations fournies par l'homme sur les pages Web (Berners Lee et al., 2001). Les annotations sémantiques décrivent le contenu de ces pages à partir de concepts et/ou de relations représentés dans une ontologie. Dans ce cadre, les ontologies jouent un rôle clé : elles fournissent le vocabulaire et la connaissance du domaine qui seront utilisés pour analyser le contenu des pages Web. L'annotation sémantique basée sur l'ontologie exige que celle-ci soit riche : elle doit représenter non seulement les concepts et relations potentiellement mentionnés dans les documents, mais aussi les termes (et leurs variantes) employés pour dénoter ces entités dans les textes. Depuis 2002, l'évolution des ontologies a fait l'objet de nombreuses recherches. La construction d'ontologies passe par plusieurs étapes. Parmi elles, l'étape d'évolution qui consiste à rendre l'ontologie plus précise et plus appropriée au domaine qu'elle décrit. L'évolution d'ontologie correspond en effet à l'application d'une succession d'opérations de changement. C'est évidemment une tâche critique, car la nouvelle implémentation du changement peut rendre l'ontologie incohérente. Inévitablement, certaines annotations seront une source de contradictions au regard de l'ontologie. La cause réelle d'une contradiction peut venir du texte lui-même, de l'ontologie ou encore des règles d'annotations. Les contradictions dans l'évolution d'ontologie peuvent se produire pour plusieurs raisons telles que : des erreurs de modélisation lors de la correction ou l'adaptation de l'ontologie du domaine, des erreurs de conceptualisation ou de spécification. Après avoir détecté des annotations et des ontologies incohérentes, celles-ci doivent être corrigées en vue de rétablir la cohérence de toute la base d'annotations et de l'ontologie. Ce chapitre propose une nouvelle façon de vérifier les graphes sémantiques en utilisant la méthode du model checking afin de réduire les erreurs dans l'annotation et rendre les données plus pertinentes tout en veillant à la cohérence de l'ontologie évoluée.

Le model checking est un outil puissant pour la vérification du système, car il peut révéler des erreurs qui n'ont pas été découvertes par les autres méthodes formelles telles que les essais et la simulation. Le model checking utilise la logique temporelle pour décrire les propriétés à vérifier sur le modèle du système. Comme nous l'avons vu dans les exemples du chapitre 2, le model checking peut gérer des problèmes complexes avec de grandes quantités d'informations, stockées sous forme de graphes, afin de vérifier les systèmes critiques. En comparaison, dans le web sémantique, l'utilisation de graphes est un problème profond et de graves problèmes d'évolutivité surviennent (Homma et al., 2009). Ainsi, il est approprié d'utiliser les algorithmes développés pour le model checking, dans le domaine du Web sémantique.

1 LA QUALIFICATION

Cette section détaille notre approche (Gueffaz et al., 2011c) qui consiste à transformer des graphes sémantiques en modèles afin d'être vérifiés par le model checker. Cette approche de transformation et de vérification est nommée « ScaleSem ». Pour cela, nous avons développé deux outils appelés « RDF2SPIN » (Gueffaz et al., 2011b) et « RDF2NuSMV » (Gueffaz et al., 2011a) (ces deux outils ont été déposés à l'Agence pour la Protection des Programmes (APP)[14]), qui transforment

[14] http://www.app.asso.fr/focus/le-depot.html

les graphes sémantiques respectivement dans le langage PROMELA (Ben-Ari, 2008) et dans le langage SMV (McMillan, 1993).

Nous avons utilisé le model checker SPIN (Holzmann, 2003) et NuSMV (Cimatti et al., 2000) (Cimatti et al., 2002) pour la vérification de graphes sémantiques. Après avoir présenté plusieurs model checkers, nous avons choisi les model checkers SPIN et NuSMV qui utilisent des automates non temporisés c'est-à-dire que la notion de temps n'est pas prise en compte dans les arêtes de nos modèles. Ce modèle non temporisé est approprié ici, car un graphe sémantique ne prend pas en compte le temps (ou n'a pas besoin de variable de temps) pour passer d'un état du graphe à un autre. Les model checkers Spin et NuSMV sont les plus utilisés dans le domaine de la vérification qualitative.

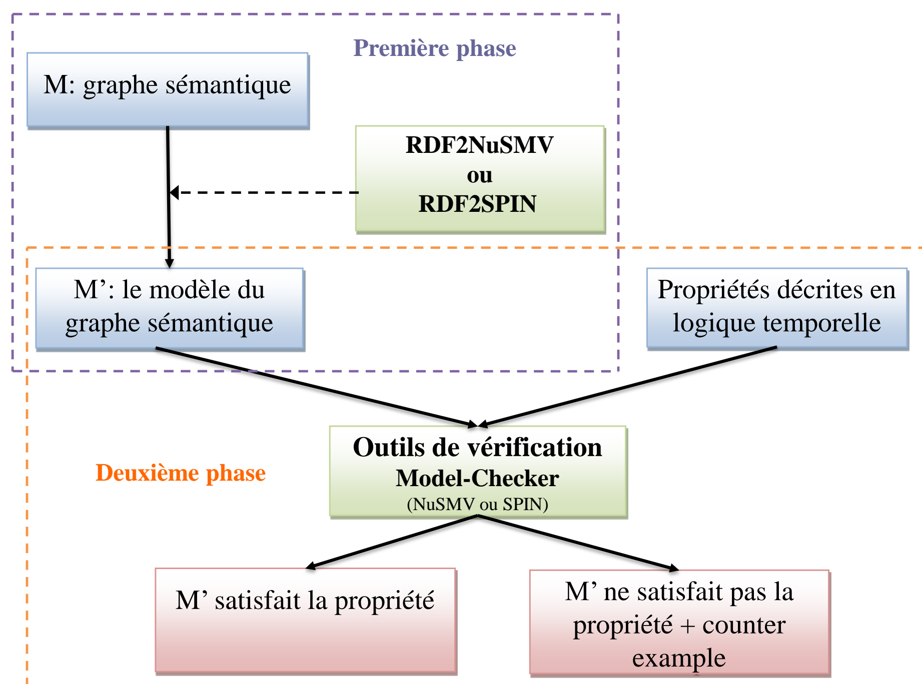


Figure 31. L'approche de la méthode ScaleSem.

Dans la Figure 31 ci-dessus, nous présentons l'architecture de notre approche ScaleSem. Nous pouvons obtenir le graphe sémantique (RDF) et sa description dans la logique temporelle à partir d'une description en langage naturel. Cette architecture est divisée en deux phases. La première phase concerne la transformation du graphe sémantique en un modèle grâce à nos outils RDF2SPIN et RDF2NuSMV disponible sur le lien suivant : http://scalesem.checksem.fr/. Cette phase est composée de trois étapes. La première étape consiste à explorer l'ensemble du graphe sémantique pour obtenir tous les triplets du graphe. La deuxième détermine une racine globale pour le graphe, et la dernière étape quant à elle donne la représentation du graphe sémantique en modèle écrit dans le langage Promela ou le langage SMV (suivant le model checker utilisé). La deuxième phase concerne la vérification de propriétés exprimées en logique temporelle sur le modèle avec un des deux model checker SPIN ou NuSMV.

1.1 LES ETAPES DE TRANSFORMATION

La transformation de graphe sémantique en un modèle est la première phase de l'approche ScaleSem. Cette phase consiste à représenter le graphe sémantique sous une autre forme, afin de permettre au Model Checher de vérifier le graphe. Les différentes étapes de cette transformation sont présentées dans cette section.

1.1.1 EXPLORATION DU GRAPHE

Afin d'exploiter les graphes sémantiques, nous devons déterminer s'ils ont un sommet racine, et si ce n'est pas le cas, nous devons créer une nouvelle racine, pointant vers chacun des sommets racines précédents, en prenant soin de garder la taille du graphe obtenu, la plus petite possible.

Prenons un graphe sémantique représenté sous forme d'un couple (V, E), où V est l'ensemble des sommets et $E \subset V \times V$ l'ensemble des arêtes. Pour un sommet x, on note $E(x) = \{y \in V | (x, y) \in E\}$ l'ensemble de ses sommets successeurs, et nous supposons que ces sommets sont classés à partir de $E(x)_0$ to $E(x)_{|E(x)|-1}$. Cela correspond à la structure de données classique pour représenter des graphes en mémoire, consistant en un tableau indexé par les sommets et contenant dans chaque entrée la liste des sommets successeurs du sommet correspondant. Il existe plusieurs algorithmes pour parcourir de grands graphes. Parmi eux, la recherche en profondeur d'abord (DFS) et la recherche en largeur d'abord (BFS) sont les plus connus. L'algorithme d'exploration de base que nous utilisons est l'algorithme de recherche en profondeur d'abord (DFS), illustré ci-dessous pour explorer le graphe (voir le Script 12), tout en sachant que l'algorithme en largeur d'abord fonctionne également dans ce contexte.

```
Algorithme : PROCEDURE DFS (x):
  begin
     visited(x) :=  true;  // l'état x est marqué comme visité
     p(x) := 0;  // on commence à visiter ses successeurs
     stack := push(x, nil);
     while stack ≠ nil do
       y := top(stack);
       if p(y) < |E (y)| then // il y a des successeurs non visités
         z := E (y) p(y);
         p(y) := p(y)+1; // prendre le successeur de y
         if ¬ visited (z) then
           visited(z) := true;  // le marquer comme visité
           p(z) := 0;  // commence l'exploration de ses successeurs
           stack := push(z, stack)
         endif
     else //tous les successeurs de y ont été explorés
         stack := pop(stack)
     endif
   end
end
```

Script 12. Algorithme de recherche en profondeur.

Nous avons considéré ici une variante itérative du DFS, utilisant une pile explicite, plutôt que la variante récursive donnée en (Tarjan, 1972). Ceci est nécessaire dans la pratique pour éviter les débordements de la pile d'appels du système lorsque l'algorithme est invoqué pour explorer de grands graphes.

1.1.2 DETERMINATION D'UNE RACINE

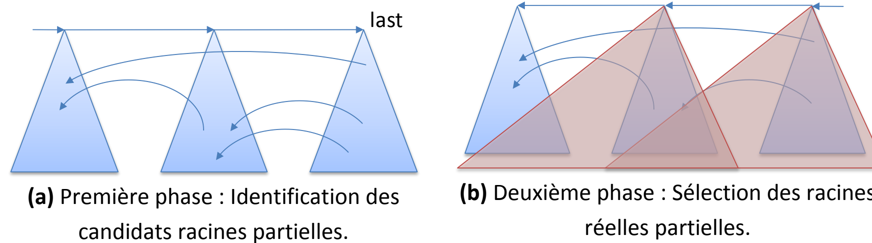
Si le graphe sémantique n'a pas de sommet racine, nous devons en créer un nouveau en tant que successeur ayant tous les sommets du graphe, mais cela augmenterait le nombre d'arêtes. Nous cherchons donc à effectuer cette opération en ajoutant le moins d'arêtes possible. Un sommet x d'un graphe orienté est une racine partielle s'il ne peut pas être atteint à partir de n'importe quel autre sommet du graphe. Si le graphe ne contient qu'une seule racine partielle, tous les autres sommets du graphe peuvent être atteints de la racine, sinon il y aurait d'autres racines partielles dans le graphe. Si le graphe a de multiples racines partielles, la façon la plus économique de fournir une racine est d'en créer une nouvelle avec toutes les racines en tant que successeur partiel : cela permettra d'ajouter au graphe un nombre minimal d'arrêtes.

Nous calculons l'ensemble des racines partielles (voir le script Script **13**) en deux phases, chacune consistant en une exploration successive du graphe. La première phase identifie un ensemble de racines partielles candidates, et la seconde affine cet ensemble afin de déterminer les racines partielles du graphe.

```
Algorithme : PROCEDURE ROOTELECTION():  // pré-condition: ∀ x ∈ V.visited(x) = false
begin
    // première phase
    root_list := nil;
    forall  x ∈ V do
        if ¬ visited(x) then
          DFS(x);
          root_list := cons(x, root_list)
        endif
    endfor;
    // deuxième phase
    if |root_list|= 1 then
        root := head(root_list) // la seule racine partielle est la racine globale
    else
        forall x ∈ V do  visited(x) := false;  endfor;
        forall x ∈ root_list do // explore à nouveau les racines partielles en ordre inverse
          if ¬ visited(x) then
              DFS(x)
          else
              root_list := root_list \ {x} // la racine partielle n'en est pas une en fait
          endif
        endfor;
        if |root_list| = 1 then
          root := head(root_list) // la racine partielle est la racine globale
        else
          root := new_node(); // nouvelle racine, parent des racines partielles
          E(root) := root_list
        endif
    endif
end
```

Script 13. Algorithme de détection de racine.

Remarque : une propriété doit toujours disposer d'une ressource et une valeur, la ressource ne doit jamais être une valeur avec le même prédicat, c'est-à-dire une boucle dans le graphe RDF.



(a) Première phase : Identification des candidats racines partielles.

(b) Deuxième phase : Sélection des racines réelles partielles.

Figure 32. Calcul des racines partielles d'un graphe RDF.

La première phase (voir la Figure 32(a)) explore le graphe jusqu'à ce qu'il soit entièrement exploré, et insère dans *root_list* tous les sommets sans prédécesseur. Si *root_list* contient un seul sommet alors c'est la racine (globale) du graphe puisque tous les autres sommets sont accessibles à partir de lui et il est inutile de passer à la deuxième phase. Sinon, n'importe quel sommet contenu dans *root_list* pourrait également être une racine partielle : le rôle de la deuxième phase consiste à déterminer laquelle de ces racines partielles candidates est en effet la racine globale du graphe.

La deuxième phase (voir la Figure 32(b)) effectue une nouvelle vague d'explorations successives de l'algorithme DFS des racines partielles contenues dans *root_list* dans l'ordre inverse dans lequel elles ont été insérées dans la liste. Si une racine dans *root_list* peut être visitée par une racine partielle de la liste, elle est retirée de la liste parce qu'elle n'est pas une racine partielle. À la fin de cette phase, toutes les racines partielles présentes dans *root_list* sont des racines partielles. En effet, chaque sommet est inaccessible à partir de l'un des candidats racines partielles dans *root_list*. Une racine est créée (Figure 33), ayant en tant que successeur toutes les racines partielles de *root_list*, garantissant ainsi que tous les sommets du graphe sont accessibles à partir de la nouvelle racine. Par conséquent, un tel sommet est inaccessible à partir d'autres nœuds du graphe.

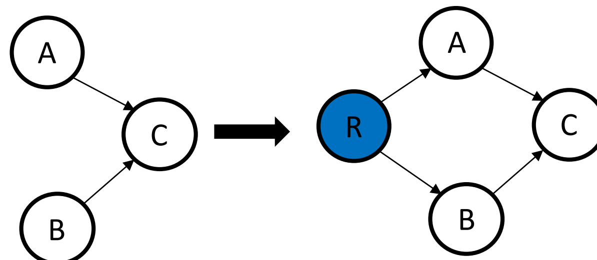


Figure 33. Ajout d'une Racine.

Une racine est un nœud unique qui n'a pas de prédécesseur. Dans ce graphe, nous avons le nœud A et nœud B, deux racines partielles. Nous créons une nouvelle racine globale comme le montre la figure (cercle bleu "R") qui pointe vers les deux racines.L'algorithme pour déterminer une racine dispose d'une complexité $O(|V| + |E|)$, linéaire dans la taille du graphe (nombre de sommets et d'arêtes), étant donné que chaque phase visite chaque état et parcours chaque arrête du graphe qu'une seule fois. Étant donné que le graphe doit être parcouru entièrement, afin de déterminer s'il a une racine ou non.

1.1.3 Generation du modele

La troisième et dernière étape est divisée en trois sous-étapes. La première et la deuxième génèrent deux tables : une table des triplets et une autre table qui dépendra de l'outil utilisé

RDF2SPIN ou RDF2NuSMV. La dernière sous-étape produira le modèle du graphe sémantique écrit dans le langage d'entrée du model checker.

- **Table des triplets** - En parcourant le graphe RDF par des algorithmes de parcours de graphes (parcours en profondeur ou en largeur), nous créons une table constituée de ressources, de propriétés et de valeurs. Dans notre graphe sémantique, la ressource est un sommet, la propriété représente une arête et la valeur est le sommet successeur correspondant à l'arête du sommet. La table de triplets du graphe sémantique est utile pour la prochaine étape.
- Dans cette deuxième sous-étape, RDF2SPIN génère un tableau des ressources et des valeurs, tandis que RDF2NuSMV génère une table de correspondances.
    - **Table des ressources et des valeurs** – Pendant le parcours de la table des triplets, vu dans l'étape précédente, nous attribuons à chaque ressource et à chaque valeur une fonction unique. Ces fonctions sont de type *proctype*. Nous combinons toutes ces fonctions dans une table appelée table des ressources et des valeurs.
    - **Table de correspondance** – dans cette étape, nous attribuons pour chaque ressource ses messages suivants (les prédicats) et ses états suivants (valeurs).
- **Le modèle de graphe** - Dans cette dernière sous-étape, nous écrirons le modèle correspondant au graphe sémantique que nous voulons vérifier. Le modèle sera écrit en langage PROMELA si nous utilisons l'outil de transformation RDF2SPIN et en langage SMV avec l'outil RDF2NuSMV.

1.1.4 RDF2NuSMV

Nous avons développé l'outil RDF2NuSMV qui permet la transformation de graphes sémantiques vers un modèle de graphes en langage SMV, prenant en considération toutes les étapes de transformations décrites précédemment. Cet outil a été développé en langage C++.

Vue d'ensemble du langage SMV. SMV (Symbolic Model Verifier) est le langage utilisé par la model checker NuSMV. Ce langage est conçu pour permettre la description des systèmes à états finis. La description d'un système complexe peut être décomposée en modules, et chacun d'eux peut être instancié plusieurs fois. Cela offre à l'utilisateur une description modulaire et hiérarchique, et prend en charge la définition de composants réutilisables. Chaque module définit une machine à états finis. Les modules peuvent être composés de manière synchrone ou asynchrone à l'aide d'entrelacement. Le module principal nommé « Main » comporte trois portions de code :

- **VAR** identifie une portion de code où les variables sont définies.
- **ASSIGN** identifie une portion de code où les variables sont initialisées et leurs évolutions sont décrites.
- **SPEC** définit les propriétés à vérifier. Ces propriétés peuvent être exprimées en logique temporelle linéaire ou arborescente (chapitre 4).

Le langage d'entrée NuSMV nous permet de décrire des systèmes déterministes et non déterministes. Une des caractéristiques les plus importantes de NuSMV est qu'il est un système ouvert pouvant être facilement modifié, personnalisé ou étendu. Cela est possible par l'architecture de NuSMV structurée et organisée en modules. Chaque module met en œuvre un ensemble de

fonctionnalités et communique avec les autres par l'intermédiaire d'une interface définie avec précision.

NuSMV possède à la fois un mode batch et un mode interactif. Le mode batch offre une méthode intégrée pour traiter avec le système dans lequel les calculs sont activés en fonction d'un algorithme fixe prédéfini. Le mode batch permet une interaction avec le système qui est essentiellement le même fourni par l'ancien model checker CMU SMV.

- **Le mode Interactive** : Dans ce mode, l'utilisateur peut activer les différentes étapes de calcul comme des commandes du système avec des options différentes. Ces étapes peuvent donc être invoquées séparément, éventuellement annulées, ou répétées. Ces mesures comprennent la construction du modèle de différentes façons et les spécifications du model checking. En outre, le mode interactif de NuSMV permet à l'utilisateur d'utiliser un langage de scripts pour définir un modèle différent de contrôle des algorithmes de model checking qui peuvent être invoqués comme une tactique. Ces algorithmes sont fournis via des scripts paramétrables.
- **Le mode Batch** : Dans ce mode, le système se comporte la plupart du temps comme l'original CMU SMV model checker. Il effectue quelques-unes des étapes décrites précédemment dans une séquence fixe. La séquence peut être modifiée via les options de ligne de commande qui permettent différents calculs à des positions fixes.

Le script ci-dessous donne un aperçu du modèle représentant le graphe sémantique.

```
MODULE main
VAR
    state : { };
    msg : {nop, hasNext};
INIT
    state =AddBasicClass_Professor;
INIT
msg = nop;
ASSIGN
next(msg) :=
 case
  state=AddObjectProperty_attends : { hasNext};
TRUE:msg;
esac;
next(state) :=
  case
   state=AddObjectProperty_attends & msg=hasNext: Professor;
TRUE:state;
esac;
```

Script 14. Modèle représentant un graphe sémantique

Ce modèle est composé des parties suivantes:

- **VAR** : cette portion du code définit tous les états du graphe sémantique dans la variable *state*, tandis que la variable *msg* contient tous les prédicats du graphe sémantique.
- **ASSIGN** : cette portion du code est composée de deux parties :
  - **Next(msg)** : cette partie décrit pour chaque état (nœud) du graphe sémantique les différents prédicats (arcs sortant du nœud).

- **Next(state)** : cette partie décrit les états successeurs pour chaque sujet et pour chaque prédicat du graphe sémantique. Elle affiche toute la table des triplets.

- **SPEC** : cette dernière portion du modèle définit les propriétés décrites en logique temporelle linéaire ou arborescente. Pour les logiques temporelles linéaires, il faut les précéder par le mot clé *LTLSPEC* et par *CTLSPEC* pour la logique temporelle arborescente.

Nous avons testé l'outil RDF2NuSMV sur plusieurs graphes sémantiques de différentes tailles. Nous avons calculé la moyenne du temps de transformation de chaque graphe vers le langage SMV. Ces résultats sont présentés dans la Figure 34.

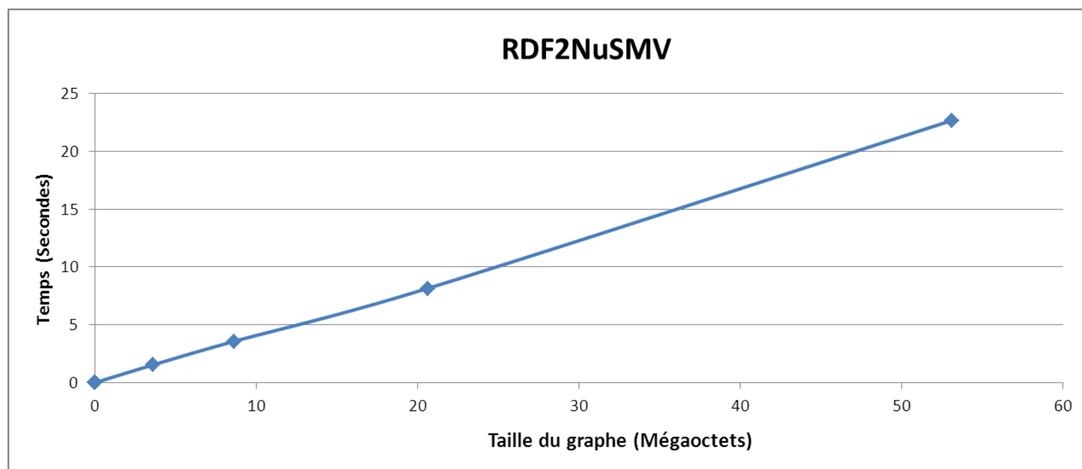


Figure 34. Temps de conversion de graphes sémantiques avec l'outil RDF2NuSMV.

Nous remarquons dans la Figure 34, que le temps de conversion augmente avec l'augmentation de la taille des graphes sémantiques. Nous constatons que la courbe de temps de la transformation des graphes sémantiques vers des modèles NuSMV suit une courbe polynomiale.

1.1.5 RDF2SPIN

Nous avons développé l'outil « RDF2SPIN ». Il transforme des graphes sémantiques vers un autre langage, le langage PROMELA. Il prend en considération toutes les étapes de transformation décrites dans la section 1.1. Cet outil a été développé avec le langage C++, sous l'environnement Linux. RDF2SPIN analyse le graphe sémantique avant de le convertir en modèle SPIN (Langage PROMELA), afin de s'assurer de l'absence de toute erreur dans le graphe sémantique.

Vue d'ensemble du langage PROMELA. PROMELA (PROtocol MEta LAnguage ou PROcess Meta LAnguage) est un langage de spécification de systèmes asynchrones, utilisé pour décrire des modèles en SPIN. Ce qui, en d'autres termes, veut dire que ce langage permet la description de systèmes concurrents, comme les protocoles de communication. Il autorise la création dynamique de processus. La communication entre ces différents processus peut se faire en partageant les variables globales ou alors en utilisant des canaux de communication. On peut ainsi simuler des communications synchrones ou asynchrones.

Le langage PROMELA ne fait pas de différence entre les instructions et les conditions. Une instruction ne peut être passée que si elle est exécutable, une condition que si elle est vraie. Sinon le processus est bloqué jusqu'à ce que la condition devienne vraie.

Le model checker SPIN peut être exécuté en quatre modes :

- **Simulation aléatoire** (Random simulation) : ce mode utilise un générateur de nombres aléatoires pour résoudre le non-déterminisme inhérent dans un programme concurrent ainsi que le non-déterminisme possible dans les commandes gardées d'un même processus. Dans ce mode, SPIN peut remplacer le simulateur de concurrence classique comme un outil pour l'étude des programmes concurrents.
- **Simulation interactive** (Interactive simulation) : permet à l'utilisateur de choisir la prochaine instruction à exécuter.
- **Mode vérification** (Verification mode) : dans ce mode, SPIN recherche un contre-exemple, un calcul qui viole une spécification correcte.
- **Simulation guidée** (Guided simulation) : si un contre-exemple est trouvé, un chemin (une trace) du calcul incorrect peut être utilisé pour récréer le calcul à l'utilisateur afin d'être examiné.

Dans nos travaux de recherches, nous utilisons seulement le mode aléatoire (Random simulation) du model checker SPIN. Il y a deux types de vérification de modèles avec SPIN :

- **La vérification avec des assertions** : cette technique de vérification est assurée par le mot clé **assert.** Par exemple, le rôle de l'instruction « *assert (dividend >= 0 && divisor > 0)* » est de vérifier que les variables *devided* et *divisor* doivent être toujours supérieures à zéro.
- **La vérification avec la logique temporelle** : c'est le type de vérification le plus utilisé et le plus sûr. Avec les assertions, il y a toujours un risque d'erreur, car elles sont limitées dans les propriétés qui peuvent être spécifiées, étant attachées à des points de contrôle spécifiques dans les procédures. Voir dans le livre (Ben-Ari, 2008) à la page 69, un très bon exemple.

La Figure 35 présente l'architecture de la vérification de graphe sémantique avec l'outil RDF2SPIN. Il transforme le graphe sémantique en modèle écrit en langage Promela. Ce dernier est passé en paramètre avec une propriété en logique temporelle linéaire au model checker SPIN. L'un des moyens que SPIN réalise est la génération d'un programme optimisé appelé « vérificateur » pour chaque modèle écrit en langage PROMELA. La vérification en SPIN est un processus en trois étapes :

- Générer le vérificateur à partir du code PROMELA : le vérificateur est un programme écrit en langage C.
- Compiler le vérificateur en utilisant un compilateur C.
- Exécuter le vérificateur. Le résultat de l'exécution du vérificateur est un rapport (Report) si tous les calculs sont corrects ou bien une trace (Trail) s'il contient une erreur.

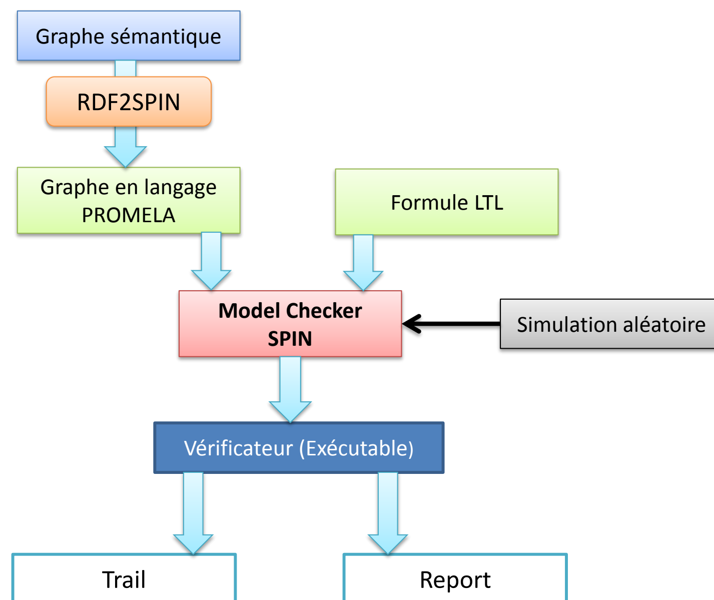


Figure 35. L'architecture de vérification avec RDF2SPIN.

L'outil RDF2SPIN avait plusieurs limites. Dans le choix de notre modélisation, l'impossibilité de décrire des propriétés en logique temporelle était la limite la plus importante, ce qui ne permettait pas au model checker SPIN de vérifier la cohérence du graphe sémantique. Alors RDF2SPIN a été amélioré en passant par trois versions.

La première version était composée de plusieurs processus. Chaque processus représentait un nœud du graphe sémantique. Il renvoyait des messages affichant les noms des différentes valeurs du triplet représenté pasr ce nœud avec les noms des processus lui succédant s'il était un sujet ou seulement son nom si le nœud était une valeur. Nous ne pouvions pas générer de formule en logique temporelle, car les processus ne communiquaient pas entre eux. La limite résidait dans la définition des processus, car l'utilisation des variables dans les logiques temporelles pour représenter un état (ou nœud) du graphe n'était pas permise.

```
…
proctype st_01() {
printf("st_01\n");
if
   :: ch==1 -> printf("->?m1\n"); run st_02();
   :: ch==2 -> printf("->?m2\n"); run st_03();
fi
}
…
```

Script 15. Un processus PROMELA représentant un état sémantique (1ère version).

La deuxième version était basée sur les canaux de communications. Grâce aux canaux de type rendez-vous, le transfert du message de l'expéditeur (un processus avec un envoi déclaration) au récepteur (un processus avec un recevoir cette déclaration) est synchrone et il est exécuté comme une seule opération atomique : la deuxième version de l'outil RDF2SPIN permet donc la communication entre les processus. Encore une fois, elle ne permet pas la génération de formules en logique temporelle à cause du fait qu'on ne peut pas définir les canaux de communications comme des variables afin de représenter les états ou les nœuds.

```
…
proctype st_01() {
printf("st_01\n");
if
   :: ch==1 -> printf("->?m1\n"); run st_02(); glob!m1;
   :: ch==2 -> printf("->?m2\n"); run st_03(); glob!m2;
fi
}
…
```

Script 16. Un processus PROMELA représentant un état sémantique (2e version)

La troisième et dernière version de l'outil RDF2SPIN définit un seul processus avec plusieurs états. Elle permet de générer des formules en logique temporelle, car on peut représenter les états (ou nœuds) par des variables.

```
…
Proctype graphe() {
racine: setState(Book);

if
   :: ch==1 -> printf("auteur1"); goto auteur1;
   :: ch==2 -> printf("auteur2"); goto auteur2;
fi

auteur1: setState(SaintExupéry);

auteur2: setState(VictorHugo);

}
…
```

Script 17. Un processus PROMELA utilisant les variables pour représenter les états sémantiques.

L'outil RDF2SPIN a été testé sur plusieurs graphes sémantiques de taille différente. Nous avons calculé la moyenne du temps de transformation de chaque graphe sémantique en langage PROMELA. Ces résultats sont présentés dans la **Figure 36**.

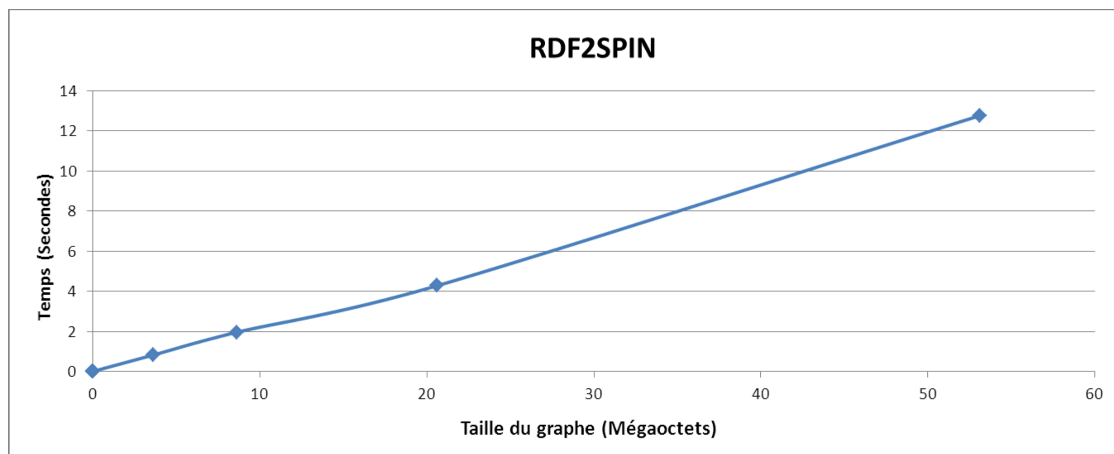


Figure 36. Temps de conversion de graphes sémantiques avec l'outil RDF2SPIN.

Nous remarquons dans la figure 36 que le temps de conversion augmente avec l'augmentation de la taille des graphes. Nous constatons que la courbe de temps de la transformation des graphes sémantiques vers des modèles PROMELA suit une courbe polynomiale.

1.2 La vérification

La deuxième phase de l'approche ScaleSem est la vérification. Cette vérification est assurée par le model checker. Après avoir transformé le graphe sémantique en un modèle compréhensible par le model checker, nous pouvons écrire les propriétés à vérifier en logique temporelle.

Nous passons au model checker le modèle du graphe sémantique et les propriétés à vérifier. Le model checker vérifiera une à une les propriétés et retournera vrai si la propriété est vérifiée, ou faux avec un contre-exemple dans le cas contraire.

1.3 Comparaison entre RDF2SPIN et RDF2NuSMV

Pour vérifier les graphes sémantiques avec les model checkers SPIN et NuSMV, il faut tout d'abord les transformer en modèle écrit dans le langage PROMELA (langage d'entrée du model checker SPIN) et en langage SMV (langage d'entrée du model checker NuSMV) respectivement. Pour cela, nous avons développé deux outils s'occupant de cette conversion, qui sont : RDF2SPIN et RDF2NuSMV.

Cette section compare ces deux outils. La comparaison porte sur les points suivants:

- **Le temps de conversion des graphes sémantique** : compare le temps de conversions des graphes sémantiques pour les deux outils.
- **La taille des fichiers de sorties** : compare la taille des fichiers transformés par les deux outils RDF2SPIN & RDF2NuSMV, c'est-à-dire le fichier écrit en langage PROMELA et en langage SMV.
- **Les model checkers SPIN et NuSMV** : cette dernière comparaison, comparera les avantages et inconvénients des deux model checkers et aussi les deux modèles retournés par les outils RDF2SPIN et RDF2NuSMV.

Le temps de conversion des graphes sémantiques des deux outils RDF2SPIN et RDF2NuSMV est présenté dans la Figure 37 ci-dessous.

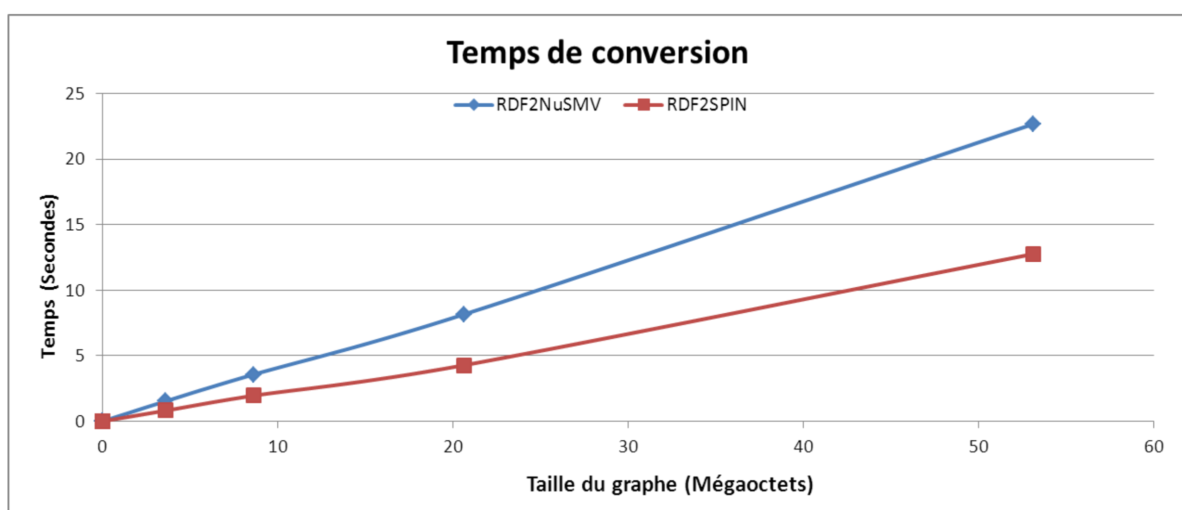


Figure 37. Temps de transformation avec les outils RDF2SPIN et RDF2NuSMV.

Nous remarquons que pour des graphes sémantiques ne dépassant pas 1 Mo, la transformation pour les deux outils est très rapide, elle avoisine les 0 seconde. À l'inverse, pour les graphes sémantiques supérieurs à 1 Mo, l'outil RDF2SPIN est plus rapide que l'outil RDF2NuSMV. Les

deux outils sont très rapides pour la transformation des graphes sémantiques. La transformation d'un graphe de 53,1 Mo qui représente 400 914 triplets ne dépasse pas les 27 secondes.

La Figure 38 présente une comparaison de la taille des fichiers de sorties transformés par les outils RDF2SPIN et RDF2NuSMV. La figure montre que les graphes sémantiques transformés (les modèles) avec l'outil RDF2SPIN ont une taille plus petite que ceux transformés avec l'outil RDF2NuSMV.

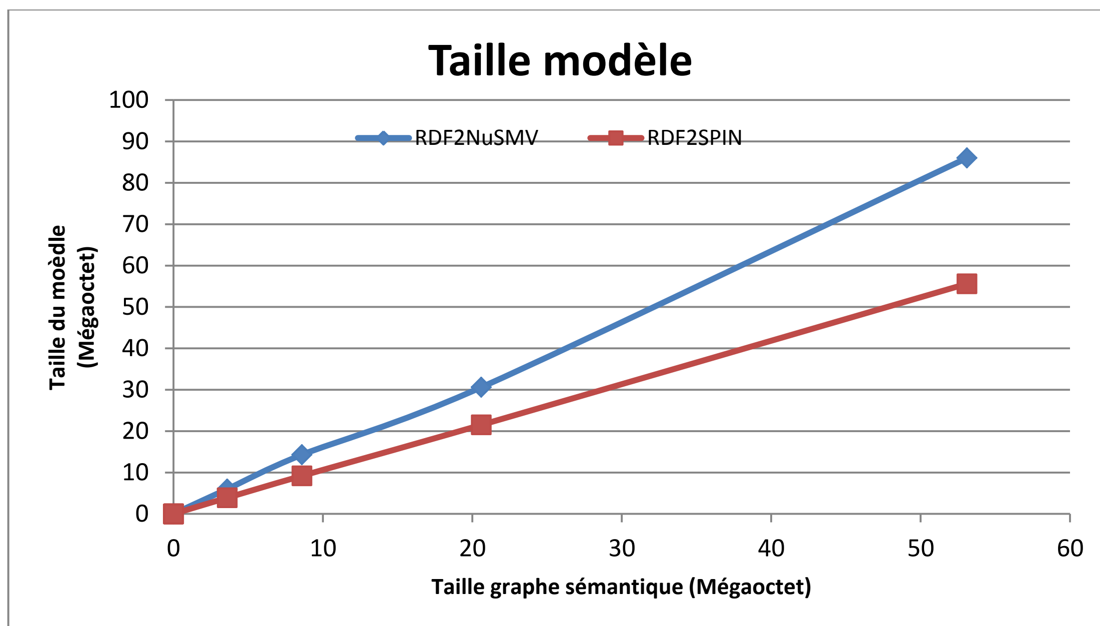


Figure 38. Taille des modèles de graphe sémantique.

RDF2SPIN est plus rapide dans la transformation et la taille de ces modèles est plus petite que ceux du RDF2NuSMV.

On peut déduire que RDF2SPIN est meilleur que RDF2NuSMV, mais le model checker SPIN et les modèles PROMELA ont quelques limitations :

- **Nombre d'états :** Un modèle SMV peut comporter un grand nombre d'états pour décrire le graphe sémantique, alors qu'avec un modèle PROMELA, les états sont définis dans une variable de type *mtype* limitée à 255.
- **Type de logique temporelle** : Le model checker NuSMV utilise deux types de logique temporelle, les logiques temporelles linéaire et arborescente, tandis que le model checker SPIN utilise seulement la logique temporelle linéaire. La différence entre ces deux logiques temporelles est qu'avec la logique temporelle linéaire, la vérification se fait chemin par chemin (unique futur possible) alors qu'avec la logique temporelle arborescente elle se fait sur tout le modèle (plusieurs futurs possibles). Comme NuSMV utilise les deux logiques temporelles, pour la phase de vérification, NuSMV est plus adapté que SPIN pour notre choix. En effet, la logique temporelle linéaire ne nous permet pas de vérifier les propriétés d'absence de blocage et d'atteignabilité (voir chapitre 3).
- **Vérification**. La vérification se fait en logique temporelle pour les deux model checker SPIN et NuSMV. Pour les modèles SMV, l'écriture de la propriété en logique temporelle se fait directement sur le graphe ou sur la commande du model checker NuSMV. Pour effectuer la

vérification avec le model checker SPIN, il faut toujours définir les variables dans le modèle PROMELA car la propriété en logique temporelle utilise des variables pour représenter les états du graphe. Cette opération ralentira la phase de vérification. En NuSMV, les états sont définis par leur nom dans le modèle SMV.

- **Résultat :** Le résultat d'une vérification avec le model checker NuSMV est clair, car, pour chaque formule en logique temporelle, nous obtenons une réponse vraie ou fausse devant la formule et si la formule est fausse alors NuSMV génère un contre-exemple (un chemin composé de nœuds) en dessous de la formule qui viole la propriété. Tandis qu'avec le model checker SPIN, le résultat est ambigu. Pour chaque formule, le model checker SPIN retourne le résultat de l'évaluation de la formule (à savoir vrai ou faux), comme NuSMV, mais également les données requises pour la reconstruction d'un calcul. Ces données sont écrites dans un fichier appelé *Trail*. (Le nom du fichier est le même que celui du fichier code PROMELA avec l'extension supplémentaire trail). Le fichier journal n'est pas destiné à être lu, mais à être utilisé pour reconstruire un calcul en exécutant SPIN en Mode simulation guidée.

Après cette comparaison entre les deux outils de transformation et de vérification, l'outil RDF2NuSMV de transformation de graphes sémantiques et le model checker NuSMV pour la vérification des propriétés en logique temporelle offrent de meilleures fonctionnalités que l'outil RDF2SPIN.

2 Cas d'utilisation

Pour illustrer notre approche, nous prenons la description suivante pour la création d'un graphe sémantique :

« *Le site internet http://checksem.u-bourgogne.fr/mgueffaz/ a une date de création (creation-date) dont la valeur est janvier 2012, a une langue (language) dont la valeur est Français et a un créateur (creator) dont la valeur est http://checksem.u-bourgogne.fr/85740. Le créateur du site internet a un âge (age) dont la valeur est 27 et a un nom (name) dont la valeur est Mahdi Gueffaz.* » La Figure 39 présente le graphe sémantique de la description ci-dessus.

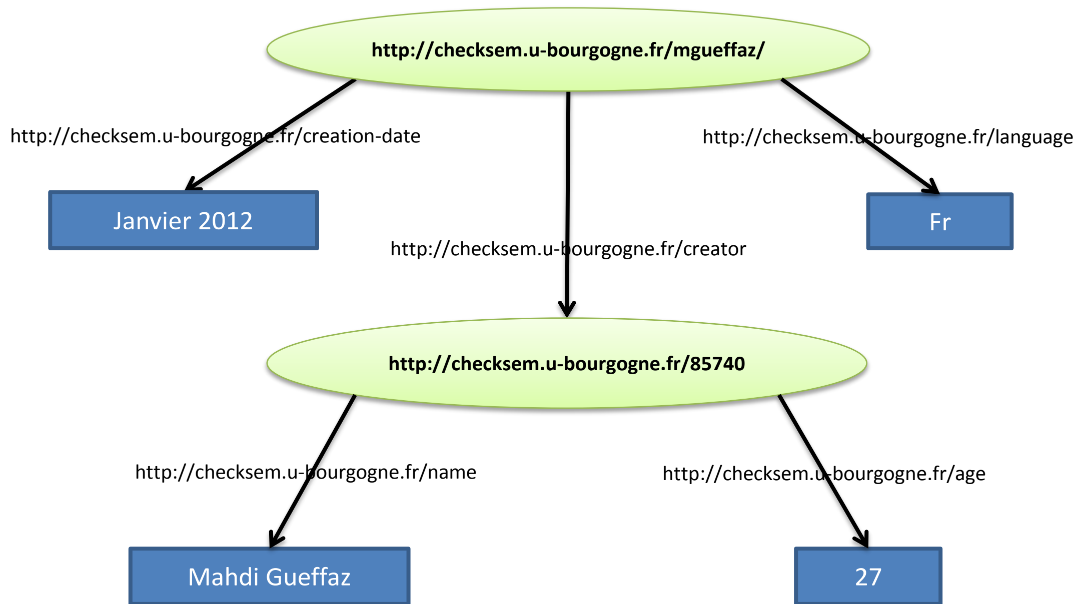


Figure 39. Un exemple de graphe sémantique.

Ce graphe sera représenté par le code en langage RDF/ XML suivant :

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:exterms="http://checksem.u-bourgogne.fr/">

<rdf:Description rdf:about="http://checksem.u-bourgogne.fr/">
      <exterms:creation-date>Janvier 2012</exterms:creation-date>
      <exterms:language>Fr</exterms:language>
      <exterms:creator>
            <rdf:Description                    rdf:about="http://checksem.u-
bourgogne.fr/85740">
                  <exterms:name>Mahdi Gueffaz</exterms:name>
                  <exterms:age>27</exterms:age>
            </rdf:Description>
      </exterms:creator>
</rdf:Description>
</rdf:RDF>
```

Script 18. Le script RDF/XML associé à la Figure 39.

La table des triplets RDF du graphe sémantique sont décrits dans le Tableau 6 ci-dessous :

Sujet	Prédicat	Objet
http://checksem.u-bourgogne.fr/mgueffaz/	http://checksem.u-bourgogne.fr/creation-date	Janvier 2012
http://checksem.u-bourgogne.fr/mgueffaz/	http://checksem.u-bourgogne.fr/language	Fr
http://checksem.u-bourgogne.fr/mgueffaz/	http://checksem.u-bourgogne.fr/creator	http://checksem.u-bourgogne.fr/85740
http://checksem.u-bourgogne.fr/85740	http://checksem.u-bourgogne.fr/name	Mahdi Gueffaz
http://checksem.u-bourgogne.fr/85740	http://checksem.u-bourgogne.fr/age	27

Tableau 6. Tableau des triplets RDF du graphe sémantique.

La transformation du graphe sémantique de la Figure 39 en modèle avec l'outil RDF2SPIN, passe par une étape de génération de modèle en utilisant la table des ressources et des valeurs décrite ci-dessous (Tableau 7) :

Ressource	Fonction PROMELA	Valeur	Fonction PROMELA
http://checksem.u-bourgogne.fr/mgueffaz/	racine: setState(http___checksem_u_bourgogne_fr_mgueffaz);	Janvier 2012	date: setState(Janvier_2012); goto end;
http://checksem.u-bourgogne.fr/85740	creator: setState(http___checksem_u_bourgogne_fr_85740);	Fr	age: setState(Fr); goto end;
		Mahdi Gueffaz	name: setState(Mahdi_Gueffaz); goto end;
		27	age: setState(m27); goto end;

Tableau 7. Table des ressources et des valeurs.

Nous remarquons dans le tableau ci-dessus, que le nombre « 27 » est transformé en « m27 ». En effet, SPIN ne doit pas interpréter ce nombre en tant qu'un entier mais en tant que chaîne de caractères. De même, les symboles tels que « : », « / » et « - » sont interdits.

Le résultat de la transformation du graphe sémantique avec l'outil RDF2SPIN, donne le modèle en langage PROMELA suivant :

```
mtype = {Mahdi_Gueffaz, Fr, janvier_2012,
http___checksem_u_bourgogne_fr_85740,
http___checksem_u_bourgogne_fr_mgueffaz, m27}
mtype state;
int ch;

proctype choice(){
do
      ::ch<3 -> ch++;
      ::ch==3 -> ch=0;
od
}

inline setState(x){
atomic {state=x; printf("state is %e:", state);}
}

proctype graphe(){

racine:setState(http___checksem_u_bourgogne_fr_mgueffaz);
if
      ::ch==1 -> printf("creator"); goto creator;
      ::ch==2 -> printf("date"); goto date;
      ::ch==3 -> printf("language"); goto language;
fi;

creator:setState(http___checksem_u_bourgogne_fr_85740);
if
      ::ch==1 -> printf("name"); goto name;
      ::ch==2 -> printf("age"); goto age;
```

```
fi;

date:setState(janvier_2012);
goto end;

language:setState(Fr);
goto end;

name:setState(Mahdi_Gueffaz);
goto end;

age: setState(m27);
goto end;

end: skip;
}

init{
atomic {
      run choice();
      run graphe();
}
}
```

Script 19. Un script PROMELA représentant un graphe sémantique.

Ce modèle est composé de deux processus. Le premier processus *choice ()* est un processus qui initialisera la variable *ch* à une valeur comprise entre un max et min. La valeur min toujours égale à la valeur 0 et la valeur max égale au nombre maximal d'arcs sortant d'un nœud du graphe sémantique. Dans notre cas, le max est de 3 (voir le nœud « http://checksem.u-bourgogne.fr/mgueffaz/ » du graphe de la Figure 39).

Le deuxième processus *graphe()* contient toutes les valeurs et ressources de notre graphe sémantique. Les valeurs font appel à la fonction prédéfinie *inline setState(x)* pour afficher leur contenu. Quant aux ressources, la fonction utilise la condition *if* pour choisir l'arc (ou prédicat) sortant, afin d'accéder à la ressource / valeur suivant(e).

La dernière partie du modèle *init* sert à lancer les deux processus décrits précédemment en même temps.

La transformation du graphe sémantique en modèle NuSMV avec l'outil RDF2NuSMV, donne le résultat suivant :

```
MODULE main
   VAR
      state :
{http___checksem_u_bourgogne_fr,http___checksem_u_bourgogne_fr_85740,Fr,Jan
vier_2012};
      msg : {nop, http___checksem_u_bourgogne_fr_language,
http___checksem_u_bourgogne_fr_name, http___checksem_u_bourgogne_fr_age,
http___checksem_u_bourgogne_fr_creator,
http___checksem_u_bourgogne_fr_creation_date};
   INIT
      state =http___checksem_u_bourgogne_fr_;
   INIT
      msg = nop;
   ASSIGN
      next(msg)  :=
```

```
        case
          state=Fr : nop ;
          state=Janvier_2012 : nop ;
          state=http___checksem_u_bourgogne_fr_85740 : {
http___checksem_u_bourgogne_fr_age, http___checksem_u_bourgogne_fr_name};
          state=http___checksem_u_bourgogne_fr_ : {
http___checksem_u_bourgogne_fr_creator,
http___checksem_u_bourgogne_fr_language,
http___checksem_u_bourgogne_fr_creation_date};
          TRUE:msg;
        esac;
      next(state) :=
        case
          state=http___checksem_u_bourgogne_fr_85740 &
msg=http___checksem_u_bourgogne_fr_age : Fr;
          state=http___checksem_u_bourgogne_fr_85740 &
msg=http___checksem_u_bourgogne_fr_name : Janvier_2012;
          state=http___checksem_u_bourgogne_fr_ &
msg=http___checksem_u_bourgogne_fr_creator :
http___checksem_u_bourgogne_fr_85740;
          state=http___checksem_u_bourgogne_fr_ &
msg=http___checksem_u_bourgogne_fr_language : Fr;
          state=http___checksem_u_bourgogne_fr_ &
msg=http___checksem_u_bourgogne_fr_creation_date : Janvier_2012;
          TRUE:state;
        esac;
```

Script 20. Un script NuSMV représentant un graphe sémantique.

La section **VAR** du modèle définit tous les états du graphe sémantique dans la variable *state*, tandis que la variable *msg* contient tous les prédicats du graphe sémantique. La section **ASSIGN** est composée de deux parties :

- **Next(msg)** décrit pour chaque état (nœud) du graphe sémantique, ses différents prédicats (arcs sortant du nœud).
- **Next(state)** décrit les états successeurs pour chaque état et pour chaque prédicat du graphe sémantique c'est-à-dire que cette partie affiche toute la table des triplets décrite dans le Tableau 6.

Nous remarquons aussi comme pour le modèle PROMELA, que les symboles comme « : », « / » et « - » ne sont pas autorisés.

Le but de cette transformation est de pouvoir vérifier les graphes sémantiques représentant des bâtiments et aussi de pouvoir utiliser les méthodes du model checking pour l'interrogation des graphes sémantiques qui sont réalisés par le langage SPARQL souffrant de plusieurs limites (voir chapitre 5).

La limite de notre approche réside dans le nombre d'états pour l'outil RDF2SPIN (255 états) et l'explosion combinatoire pour l'outil RDF2NuSMV (espace mémoire insuffisant).

3 Travaux Connexes

Dans cette section, nous présenterons brièvement les recherches liées à la vérification des graphes sémantiques en utilisant le model checking. Peu de travaux existent sur l'utilisation de méthodes formelles pour la qualification de graphes sémantiques. À l'inverse, plusieurs recherches sont effectuées permettant la vérification d'applications Web.

Le travail dans (Mateescu et al., 2009) propose une nouvelle façon de convertir un graphe sémantique en un format nommé BCG (*Binary Coded Graph*) utilisé dans la boîte à outils CADP (*Construction and Analysis of Distributed Processes*). CADP est une boîte à outils de vérifications de systèmes concurrents asynchrones. La boîte à outils accepte en entrée plusieurs langages et ceux-ci sont compilés vers LTS (*Labeled Transition System*), qui sont des graphes d'états/transitions représentant le comportement de systèmes concurrents. CADP fournit plusieurs types de représentations des LTS ; l'une d'entre elles est le format BCG.

Cependant, il est à noter que le format BCG souffre de nombreuses restrictions. Ces restrictions sont tout à fait justifiées dans l'usage de ce format avec les LTS qui est son but premier. Le détournement de son utilisation sur des graphes sémantique devient problématique. Nous l'avons vu plus tôt, les données d'un LTS sont portées par l'étiquette (label) d'une transition (une arrête), il a donc fallu s'adapter aux limites du format. Les données du graphe sémantique sont donc enregistrées dans l'étiquette d'une transition en suivant une syntaxe qui n'impose pas de limites supplémentaires sur les données stockables, qui soit facile à manipuler et enfin, le plus important, qui peut être utilisé par CADP.

Par contre, un graphe NuSMV garde toutes les informations du graphe sémantique et stocke tous les triplets RDF dans le fichier NuSMV, ainsi il n'y a pas de pertes d'informations.

Un autre inconvénient de la transformation de graphes sémantiques en graphe BCG par la boite CADP est la taille. Pour un graphe sémantique de 30 Mégaoctets, nous aurons comme résultat, un graphe d'une taille de 1,2 Giga octets en BCG, soit un facteur multiplicateur de 40. Alors qu'avec notre approche ScaleSem, la taille du graphe est très similaire à la taille du graphe sémantique.

La conversion de graphe sémantique en graphe BCG, utilise deux outils. À l'inverse, notre approche ScaleSem en utilise un seul. Cela augmentera le risque de perte d'informations et d'erreurs dans le modèle de graphe sémantique.

Il y a plusieurs recherches dans lesquelles les applications Web sont modélisées comme un graphe orienté. Dans (Sciascio et al., 2003), (Sciascio et al., 2002), les composants d'une fenêtre (par exemple, une page, une trame et un lien) sont modélisés en tant qu'états. L'accessibilité entre les états est définie comme exigence de l'application Web et ils sont vérifiés à l'aide du model checking. (Ricca et Tonella, 2001) (Ricca et Tonella, 2000), (Han et Hofmeister, 2006) ont pris une approche pour modéliser les applications Web en utilisant la composition parallèle des diagrammes UML. Le travail dans (Haydar et al., 2005) propose la voie à la discrimination des états d'intérêt par l'introduction d'un opérateur spécialisé pour LTL. Ils l'utilisent pour vérifier les applications Web.

Dans (Yuen et al., 2005), les auteurs proposent un modèle de comportement de l'application Web, appelé « Automates du Web » basé sur l'architecture du modèle MVC. Ils modélisent le comportement d'une application Web avec du contenu dynamique comme un prolongement de liens-automates avec la fonction de contrainte logique des automates finis étendus (EFA). Le Framework de test des applications Web basées sur le comportement du modèle est également présenté dans leurs recherches. Dans (Haydar et al., 2004), les auteurs présentent une approche formelle pour les applications Web en utilisant la modélisation en automates communicants. Ils observent le comportement externe d'une partie explorée d'une application Web en utilisant un outil

de surveillance. Le comportement observé est ensuite converti en automates communicants représentant toutes les fenêtres et le cadre de l'application en cours de test en interceptant les demandes et les réponses HTTP en utilisant un serveur proxy.

Le model checker Solibri (http://www.solibri.com/) analyse les modèles d'informations pour la sécurité de l'intégrité et la qualité du bâtiment. Le système propose des services faciles à utiliser de visualisation avec des fonctionnalités intuitives. Avec un simple clic de souris, un système de rayons X révèle les défauts et les faiblesses potentiels dans la conception du modèle de bâtiment, met en évidence les éléments se chevauchant et vérifie que le modèle est conforme aux codes du bâtiment.

Le model checker Solibri est disponible avec des règles spécifiques qui peuvent détecter ce qui manque dans le modèle. Trouver ce qui est effectivement absent de la conception d'un bâtiment est un défi très difficile. Avec notre approche de qualification, nous pouvons faire la même vérification que le model checker Solibri, et en plus, nous pouvons vérifier les relations entre les objets du bâtiment.

4 Conclusion

La taille importante des graphes sémantiques pose le problème de leur vérification, leur stockage et leur parcours. Notre but dans cette recherche est la vérification des graphes sémantiques afin de faciliter leurs interrogations et parcours, avec les méthodes formelles et plus précisément avec les algorithmes du model checking. Pour cela, nous utilisons les model checkers SPIN et NuSMV, parmi les plus utilisés et les mieux documentés de nos jours.

Ce chapitre présente un aperçu de deux outils de transformation de graphes sémantiques. L'outil RDF2NuSMV transforme le graphe sémantique en modèle représenté sous le format NuSMV, tandis que l'outil RDF2SPIN le transforme en modèle PROMELA. Ces modèles servent au model checker pour vérifier des propriétés sur les graphes sémantiques.

Une comparaison entre les deux outils de transformation et de vérification a été réalisée dans ce chapitre. L'outil RDF2NuSMV de transformation de graphes sémantiques et le model checker NuSMV pour la vérification des propriétés en logique temporelle offrent de meilleures fonctionnalités que l'outil RDF2SPIN limité à 255 états.

Chapitre 6

Interrogation de graphes sémantiques

Résumé

Les requêtes SPARQL restent limitées, car elles ne permettent pas d'exprimer des requêtes avec une séquence non bornée de relations (par exemple, "Existe-t-il un itinéraire d'une ville A à une ville B qui n'utilise que les trains ou les bus?"). Le model checking a été proposé comme une technique de vérification. Il est utile pour la compréhension du modèle. L'utilisateur émet l'hypothèse d'un comportement du système, puis il l'exprime en formule de logique temporelle, et enfin, il tente d'utiliser le model checker pour valider l'hypothèse. Le processus est réitéré pendant que l'utilisateur prend connaissance du système. Cette utilisation du model checking n'a pas été assez poussée d'après (Chan, 2000). Pour aider l'utilisateur à comprendre les différents comportements du système, Chan introduit les requêtes en logique temporelle (Le *Query Checking*) et utilise une technique similaire au model checking symbolique pour déterminer les propriétés temporelles plutôt que de simplement les contrôler. Dans ce chapitre, nous proposons une nouvelle approche en utilisant les requêtes en logique temporelle pour l'interrogation des graphes sémantiques.

Plan

1 Requête en logique temporelle 136
1.1 Syntaxe 137
2 Complexité de SPARQL 137
3 SPARQL et Logique Temporelle 144
3.1 Expressivité 144
3.2 La Simplicité 147
3.3 Langage de vérification 148
3.4 Extension des requêtes 150
4 Le STL-Resolver 151
5 SPARQL vers RLT 152
5.1 Selection 153
5.2 Projection 153

5.3 Jointure 153
5.4 Union.. 154
6 Travaux Connexes 160
7 Conclusion 162

Dans le domaine du Web sémantique, plusieurs approches ont été proposées pour l'interrogation des graphes, la plupart d'entre elles ayant conduit à des langages standardisés par le W3C, tels que XPath, XQuery et SPARQL. Diverses extensions de SPARQL ont été proposées afin d'augmenter son expressivité. Plusieurs langages d'interrogation de données RDF ont été proposés et mis en œuvre. Une étude sur les langages de requêtes RDF, menée par le W3C, a identifié plus de 20 langages en cours de développement ou mis en place (Angles et Gutierrez, 2005). Pour certains, ils sont dans la lignée des langages d'interrogations de bases de données traditionnels (comme par exemple SQL (*Structured Query Language*) ou encore OQL (*Object Query Language*)). Pour les autres, ils sont basés sur des langages logiques et de règles. Parmi ces derniers nous pouvons citer :

- RQL (*RDF Query Language*) (Karvounarakis et al., 2002) qui est un langage typé pour l'interrogation des référentiels RDF ;
- SquishQL (*Simple Query Language RDF*) qui est un langage de requêtes style SQL permettant une navigation simple dans les sources du graphe RDF ;
- RDQL (*RDF data Query Language*) (Seaborne, 2004) qui est une implémentation de SquishQL ;
- RDFQL (*RDF Query Language*) qui est un langage de requêtes basé sur SQL pour effectuer des requêtes, des opérations d'inférence et la construction des vues sur les données RDF structurées ;
- TRIPLE (Sintek, et Decker, 2002) qui est un langage qui permet la définition de règles, l'inférence et la transformation des modèles RDF ;
- notation 3 (N3) (Berners-Lee, 2001) qui fournit une syntaxe textuelle pour RDF ;
- VERSA qui est un langage à base de graphes avec un certain soutien pour les règles ;
- SeRQL (*Sesame RDF Query Language*) qui combine les caractéristiques de langages comme RQL, RDQL, N-Triple, N3 ainsi que quelques nouvelles fonctionnalités ;
- RXPath (*Regular XPath*) qui est un langage de requêtes basé sur XPath (*XML Path Language*) (Magkanaraki et al., 2002) (Haase et al., 2004).
- SPARQL (Prudhommeaux et Seaborne, 2005) qui est un langage de requêtes RDF conçu pour répondre aux exigences et aux objectifs de conceptions mentionnés précédemment. Il définit un langage de requêtes avec un style similaire à SQL, où une simple requête est basée sur les modèles de requêtes et le traitement des celles-ci consistent à lier des variables pour générer des patrons de solutions (*graph pattern matching*). SPARQL est toujours en cours d'amélioration. (Gueffaz et al., 2012a).

Le langage SPARQL est devenu le langage standard pour l'interrogation des graphes sémantiques, mais ses limites ont été démontrées (Schmidt, 2009) (Pérez et al., 2006). Notre objectif de recherche principal est de définir un langage de requêtes puissant et expressif pour les graphes sémantiques, tout en gardant ce langage assez simple pour qu'il puisse être facilement intégré et compris. Dans ce chapitre, nous proposons une extension ou une amélioration du langage de requêtes SPARQL avec les opérateurs des propriétés en logique temporelle, permettant de combler les lacunes de SPARQL dans les graphes sémantiques. Pour atteindre cet objectif, nous avons utilisé la technologie disponible dans l'approche ScaleSem pour la vérification des graphes sémantiques.

1 REQUETE EN LOGIQUE TEMPORELLE

Les requêtes en logique temporelle sont une généralisation de la vérification de modèle. Cette vérification de modèle permet aux propriétés du système non seulement d'être vérifiées, mais aussi d'être générées d'une manière systématique, sous forme de logique temporelle. Une requête en logique temporelle est un cas particulier d'une propriété en logique temporelle. Elle contient un symbole substituable spécial "?" appelé *joker* (***placeholder*** en anglais). Intuitivement, la requête interroge les propriétés du système qui donnent une spécification correcte lorsque le *joker* est remplacé par une proposition dans la requête.

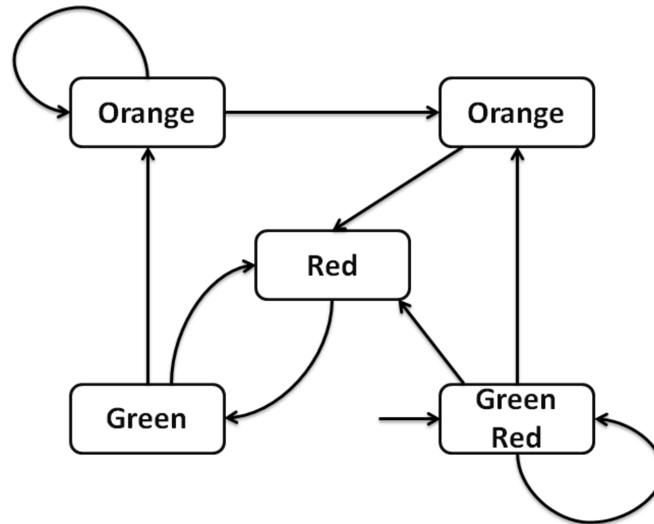


Figure 40. Exemple d'une structure de Kripke: le feu tricolore.

Pour mieux comprendre, nous illustrons cela par un exemple. Supposons que nous concevons un système de feux de circulation, comme montré dans la structure de Kripke de la **Figure 40** (en supposant que les feux de circulation alternent dans le bon ordre et sans blocage). On s'appuie sur le modèle de contrôle pour vérifier si la formule en logique temporelle est satisfaite, par exemple :

$$G(\text{orange} \rightarrow F\ \text{red}) \quad (S)$$

Cette formule indique que « toutes les lumières orange sont finalement suivies par des feux rouges ». Le model checking fournira :

- une réponse positive : le système satisfait (S)
- ou une réponse négative : le système ne satisfait pas (S).

Les requêtes temporelles conduisent à une analyse plus fine du système. La requête suivante interroge le système pour les couleurs du feu qui mènent toujours au rouge :

$$G(? \rightarrow F\ \text{red}) \quad (Q)$$

Calculer les solutions pour la requête (Q), dans le système, nous indiquera, entre autres choses, si le système satisfait sa spécification : (S) est satisfaite si et seulement si la lumière orange est une solution pour (Q). Par exemple, si (S) n'est pas satisfaite, une réponse à la requête (Q) peut conduire à la découverte qu'en réalité, la propriété qui satisfait le système est G (vert $\rightarrow$ F orange).

1.1 SYNTAXE

Soit *p* une formule de proposition atomique. Soit φ et ψ des requêtes en logique temporelle CTL. Les requêtes en logique temporelle sont définies ainsi :

- *p* contenant une ou plusieurs variables (*joker*) est une requête en logique temporelle CTL
- $\varphi \wedge \psi$, $\varphi \vee \psi$, $\neg\varphi$ sont des requêtes en logique temporelle CTL
- **AX** φ, **EX** φ, **AF** φ, **EF** φ, **AG** φ, **EG** φ, **A** $(\varphi U \psi)$, **E** $(\varphi U \psi)$ sont des requêtes en logique temporelle CTL.

2 COMPLEXITE DE SPARQL

En 2004, le Groupe de travail RDF Data Access, qui fait partie de l'activité Web sémantique du W3C, a publié un premier projet de travail public d'un langage de requêtes pour RDF, appelé SPARQL (Prudhommeaux et Seaborne, 2005). Depuis, SPARQL a été rapidement adopté comme le standard pour l'interrogation des données du Web sémantique. En janvier 2008, SPARQL est devenu une recommandation du W3C. Les requêtes SPARQL (Chebotko et al., 2006) sont des requêtes de correspondance de modèle (*pattern matching*) basées sur les triplets qui constituent un graphe RDF de données. La requête SPARQL introduit quatre types de requêtes :

- la requête **SELECT** renvoie la valeur de la variable. Elle peut être liée par un modèle correspondant de requête ;
- la requête **ASK** renvoie vrai si la requête correspond, et faux dans le cas contraire ;
- la requête **CONSTRUCT** retourne un graphe RDF en substituant les valeurs dans les modèles de données ;
- la requête **DESCRIBE** retourne un graphe RDF qui définit la ressource correspondante.

SPARQL est un langage de requêtes RDF, proche de SQL. Une requête SPARQL simple est basée sur les modes de recherches. Le traitement des requêtes se compose de liaisons de variables pour générer des modèles de solutions (filtrage). SPARQL est toujours en cours d'amélioration. Il est défini par le groupe de travail Data Access (DAWG) du W3C, et devient, ainsi, le langage de requêtes le plus populaire pour les graphes sémantiques. C'est pour l'ensemble de ces raisons que nous avons choisi de construire notre travail sur le langage SPARQL que nous allons présenter dans la suite de ce chapitre.

L'analyse de la complexité SPARQL n'est pas nouvelle : l'enquête préliminaire de la complexité combinée de SPARQL dans (Pérez et al., 2006) montre que le problème de l'évaluation des expressions complètes SPARQL est PSPACE-complet. Une amélioration conséquente de cette première analyse (Schmidt, 2009) explore la complexité de toutes expressions et fragments de la requête, où un fragment désigne une classe d'expressions ou des requêtes qui peuvent être construites en utilisant un sous-ensemble fixe des opérateurs SPARQL. Le problème d'évaluation de l'opérateur « Optionnel » seul est déjà PSPACE-difficile. L'auteur montre, en outre, que cette grande complexité est causée par une imbrication illimitée des expressions « Optionnel ». Pourtant, l'opérateur « Optionnel » est loin d'être la construction la plus compliquée en SPARQL. Cette

observation suggère que des soins dans l'optimisation de requêtes doivent être pris dans les requêtes contenant l'opérateur « Optionnel » et servira de ligne directrice pour l'optimisation de SPARQL.

Dans (Neumann et Weikum, 2009), l'optimiseur de requêtes peut choisir les jointures optimales, même pour les requêtes complexes, avec un modèle de coût qui comprend des résumés statistiques pour les chemins de jointures entiers. L'absence d'un schéma global et la diversité des noms de prédicats posent de problèmes majeurs pour la conception de base de données physique. En principe, nous pouvions compter sur un autoréglage pour matérialiser les chemins de jointures fréquentes, mais, dans la pratique, la structure évolutive des données, la variance et la dynamique de la charge de travail transforment ce problème en une tâche complexe. La modélisation des requêtes de données RDF avec un grand nombre de jointures de façon inhérente forme une grande partie de la charge de travail, mais les attributs de jointures sont beaucoup moins prévisibles que dans un cadre relationnel. Cela nécessite des choix spécifiques d'algorithmes de traitement de requêtes, et un soin pour l'optimisation des requêtes de jointures complexes.

Nous constatons, à partir de (Schmidt, 2009), que le langage de requêtes SPARQL utilisé pour interroger les graphes RDF a de nombreuses limitations. Étant donné que SPARQL est une technologie jeune, la spécification W3C actuelle a encore quelques limitations, qui deviennent évidentes lorsque nous le comparons à des langages de requêtes tels que SQL ou XQuery. La liste suivante présente les caractéristiques importantes manquantes dans SPARQL :

- **Agrégation** : La spécification actuelle ne supporte pas les fonctions d'agrégations, comme l'addition des valeurs numériques, le comptage ou le calcul de la moyenne.
- **Mises à jour** : Alors que la norme SPARQL supporte l'extraction des données de graphes RDF, la construction pour l'insertion de nouveaux triplets dans un graphe RDF et la manipulation de graphe (dans le style de SQL, Insertions et Mise à jour des clauses) sont manquants.
- **Expression de chemin** : SPARQL ne prend pas en charge la spécification de contraintes ou d'expressions de chemin. Par exemple, en utilisant une seule requête SPARQL, il est impossible de calculer la fermeture transitive d'un graphe ou d'en extraire tous les nœuds accessibles à partir d'un nœud fixe. Cette lacune a maintes fois été identifiée dans les publications précédentes et les différentes propositions pour l'intégration des expressions de chemin.
- **Vues** : Dans les langages de requêtes traditionnels comme SQL, des vues logiques sur les données jouent un rôle important. Ils sont essentiels à la conception de base de données et la gestion des accès. SPARQL ne prend pas en charge la spécification des vues logiques sur les données. Cependant, il est à noter que les vues matérialisées sur les données peuvent être extraites à partir du graphe d'entrée en utilisant la forme de requête CONSTRUCT.
- **Support pour les contraintes** : les mécanismes pour vérifier les contraintes d'intégrité dans la base de données RDF ne sont pas couverts dans la spécification SPARQL actuelle. En SQL, les contraintes d'intégrité sont dérivées implicitement de la clé primaire et étrangère qui sont spécifiées comme établies dans la phase de conception du schéma. Au-delà, il est possible d'appliquer des contraintes définies par l'utilisateur en utilisant l'instruction CREATE.

Depuis janvier 2012, le W3C a introduit une nouvelle version du langage d'interrogation SPARQL, SPARQL 1.1 (Harris et Seaborne, 2012) qui prend en compte l'agrégation et les mises à jour décrites précédemment. Mais il reste toujours limité pour les expressions de chemin (Distance entre deux nœuds, diamètre, etc…).

Plusieurs langages d'interrogations de données RDF ont été proposés et mis en œuvre, certains dans les lignées traditionnelles des langages de requêtes de bases de données (par exemple, SQL, OQL), d'autres basés sur la logique et les langages de règles (Angles et Gutierrez, 2005).

- **RQL** est un langage typé pour interroger les répertoires RDF ;
- **SquishQL** est un langage de requêtes style SQL qui permet une navigation graphique simple dans les sources RDF ;
- **RDQL** est une implémentation de SquishQL ;
- **RDFQL** est un langage de requête style SQL pour effectuer des requêtes, des opérations d'inférence, et la construction de vues sur les données RDF structurées ;
- **TRIPLE** est un langage qui permet de définir la règle, l'inférence et la transformation de modèles RDF ;
- **Notation 3 (N3)** fournit une syntaxe à base de texte pour RDF ;
- **Versa** est un langage basé sur les graphes avec un certain soutien pour les règles ;
- **SeRQL** combine les caractéristiques des langages comme RQL, RDQL, N-Triple, N3 et quelques nouvelles fonctionnalités ;
- **RXPath** est un langage de requêtes basé sur XPath, et le langage GOOD.

Il y a plusieurs propositions de langages de requêtes pour les modèles qui représentent l'information avec une structure de graphes, explicite ou implicite.

Le langage de requêtes graphiques G (Cruz et al., 1987) sert à interroger des données représentées par un graphe étiqueté. Il introduit la notion d'interrogation graphique, qui est basée sur un modèle graphique utilisant des expressions régulières pour représenter les requêtes récursives. Le langage G a évolué en un langage plus puissant appelé G+ (Cruz et al., 1988) où la requête graphique est le bloc de construction de base. Les nœuds de la requête graphique peuvent être étiquetés avec des variables et les arcs avec les expressions régulières. Une simple requête comporte deux éléments, une requête graphique qui spécifie les modèles de classe à chercher et un graphique récapitulatif qui représente la façon de restructurer la réponse obtenue par la requête graphique.

GraphLog est un langage de requêtes visuel, basé sur une représentation graphique des données et des requêtes pour l'hypertexte[15], qui a évolué à partir du langage G+ (Cruz et al., 1988). Une requête est un seul modèle graphique contenant un arc distingué. L'effet de la requête est de trouver toutes les instances du modèle qui apparaissent dans le graphe de base de données et définir pour chacun d'eux un lien virtuel représenté par l'arc distingué. Les requêtes GraphLog demandent des motifs qui doivent être présents ou absents dans le graphe de base de données. Chaque motif, appelé une requête graphique, définit de nouvelles arêtes qui sont ajoutées au graphe chaque fois

[15] Un système hypertexte est un système contenant des nœuds liés entre eux par des hyperliens permettant de passer automatiquement d'un nœud à un autre.

que le motif est trouvé. Le langage prend également en charge le calcul des fonctions d'agrégations et de chemins.

Gram présente une algèbre d'interrogation où les expressions régulières sur les types de données sont utilisées pour sélectionner des chemins dans un graphe. Il utilise un modèle de données où les chemins sont les objets de base. Une expression de chemin est une expression régulière sans union, dont le langage ne contient que des séquences alternées de type de nœuds et d'arcs, commençant et terminant par un type de nœud. Le langage de requêtes est basé sur une algèbre hyperwalk[16] avec les opérations fermées sous l'ensemble des hyperwalks. C'est un langage de requêtes avec une syntaxe style SQL. Il définit deux types d'opérations algébriques : des opérateurs unaires (projection, sélection, renommage) et binaires (jointure, concaténation, opérations sur les ensembles) qui sont fermés sous l'ensemble des hyperwalk.

PaMal (Gemis et Paredaens, 1993) est un modèle graphique pour décrire des schémas et des instances d'objets de la base de données et un langage de manipulation de données graphiques basé sur le filtrage.

GraphDB (Güting, 1994) est un modèle de données orienté objet et est un langage d'interrogations des bases de données graphiques. Une base de données en GraphDB est une collection de classes d'objets réparties en : classes simples (objets simples qui représentent des nœuds), les classes lien (liens entre les nœuds qui représentent les arcs) et les classes chemin (représentant plusieurs chemins dans la base de données). Une requête comporte plusieurs étapes. GraphDB soutient d'autres opérations, par exemple pour les fonctions de tri, regroupement, et les agrégats (par exemple Somme).

Lorel (Abiteboul et al., 1997) est un langage de requêtes pour les données semi-structurées conçues pour l'objet Modèle d'échange (OEM). Lorel est une extension de OQL, en étendant ses caractéristiques pour la manipulation de données semi-structurées. Il utilise les modèles d'expressions régulières pour parcourir les chemins hiérarchiques d'objets, limités à la sémantique de chemins simples (ou chemins acycliques). Lorel présente un langage de requêtes style SQL qui prend en charge deux types d'expressions de chemin :

- les expressions de chemin simples, qui permettent d'obtenir l'ensemble des objets accessibles par la suite d'une séquence d'étiquettes à partir d'un objet nommé dans le graphe OEM ;
- une syntaxe plus puissante pour les expressions de chemin, appelées les expressions de chemin générales fondées sur les caractères génériques et les expressions régulières.

Corese (Corby et al., 2004) est un moteur de recherche web sémantique basé sur les graphes conceptuels offrant des fonctionnalités pour l'interrogation des graphes RDF. Au moment de la rédaction, il ne supporte que les requêtes de longueurs du chemin fixes et pas d'autres expressions de chemin comme les chemins de longueurs variables ou de contraintes sur les nœuds internes, bien que cela semble être prévu.

[16] Un système hyperwalk est un système contenant des nœuds liés entre eux par des chemins permettant de passer automatiquement d'un nœud à un autre.

SPARQLeR (Kochut et Janik, 2007) étend SPARQL en ajoutant aux requêtes la possibilité de chercher des motifs de graphes impliquant des variables de chemin. Chaque variable de chemin est utilisée pour la capture simple (c'est-à-dire acyclique) des chemins dans les graphes RDF, et est comparée arbitrairement de triplets RDF entre deux nœuds donnés. Cette extension offre de bonnes fonctionnalités comme de tester la longueur des chemins et si un nœud donné se trouve dans les chemins trouvés. SPARQLeR n'est pas défini avec une sémantique formelle, d'où l'utilisation de variables de chemin (notamment dans la position du sujet). Ces variables ne sont pas forcément intuitives à comprendre, en particulier, quand elles ne sont pas liées. Même lorsque c'est le cas, l'utilisation multiple de la même variable de chemin à plusieurs reprises n'est pas entièrement formalisée : il n'est pas précisé quel chemin doit être retourné ou s'il doit retourner le même. Les effets des variables de chemins dans la clause DISTINCT ne sont pas traités non plus. Enfin, plusieurs problèmes sont soulevés dans l'évaluation dans les motifs de graphes d'une telle extension. En particulier, la stratégie d'obtention des chemins, puis ensuite de les filtrer, est inefficace, car elle peut générer un grand nombre de chemins.

SPARQ2L (Anyanwu et al., 2007) permet également d'utiliser des variables de chemins dans les modèles graphiques et offre de bonnes caractéristiques telles que les contraintes dans les nœuds et les arêtes, c'est à dire, tester la présence ou l'absence de nœuds et/ou d'arêtes, les contraintes dans les chemins, par exemple, chemins simples ou non-simples, la présence d'un motif dans un chemin. Cette extension n'est pas non plus décrite sémantiquement. On peut seulement essayer de deviner ce qui est la sémantique intuitive des concepts. Il semble que les algorithmes ne soient pas complets à l'égard de leurs sémantiques intuitives, puisque l'ensemble des réponses peut être infini en l'absence de contraintes pour l'utilisation de chemins les plus courts ou acycliques. En outre, cette extension souffre de la généralité, c'est à dire, qu'elle ne permet pas d'utiliser plus d'un motif de triplet ayant une variable de chemin. Un assouplissement de cette restriction nécessite une adaptation radicale de l'algorithme d'évaluation qui est par ailleurs inopérant. Ceci est dû à la fonction de compatibilité qui ne prend pas en compte l'utilisation de la variable de chemin même dans de multiples motifs de triplet. En ce qui concerne SPARQLeR, l'ordre d'évaluation est très complexe lors de l'utilisation de la construction PATHFILTER pour le filtrage des chemins. De plus, le résultat du modèle de graphe dépend de la construction de tous les chemins (qui peuvent ne pas être exhaustifs en raison du nombre infini de chemins pouvant être construits pour le cycle de graphes RDF), en prenant, de plus, en compte ceux qui correspondent à un motif régulier.

nSPARQL est une récente extension de SPARQL. Ce langage a un fragment restreint de RDFS et est proposé dans (Pérez et al., 2008). Cette extension permet d'utiliser des expressions régulières imbriquées, c'est à dire, des expressions régulières étendues avec des axes de branchements sous la forme empruntée de XPath. Les auteurs ont présenté une syntaxe et une sémantique formelle de leur proposition. Les expressions régulières dans SPARQL (comme dans le cas de CPSPARQL) ont la capacité de capturer la sémantique du fragment RDFS utilisé. En particulier (C) PSPARQL peut exprimer tous les exemples fournis dans (Pérez et al., 2010) pour démontrer l'expressivité du langage proposé. D'une part, nSPARQL a un axe pour la navigation sur les nœuds et les arêtes. D'autre part, CPSPARQL a des contraintes sur les arrêtes et les nœuds traversés. Il peut être utile de mettre les deux extensions ensemble.

D'autres extensions à SPARQL existent comme : **SPARQL-DL** (Sirin et Parsia, 2007) qui étend SPARQL pour supporter des requêtes en logiques de descriptions sémantiques, **SPARQL++** (Polleres et al., 2007) étend SPARQL avec des fonctions externes et d'agrégations, qui servent de base pour la description déclarative des mappages d'ontologie, et **iSPARQL** (Kiefer et al., 2007) étend SPARQL pour permettre de mesurer les jointures similaires.

PSPARQL (synonyme de *Path SPARQL*) correspond au langage SPARQL en y ajoutant des motifs d'expressions régulières pour surmonter cette limitation fournissant un large éventail de paradigmes d'interrogations (Alkhateeb et al., 2008). Les graphes PRDF (*Path RDF*) sont utilisés pour définir les patrons de graphes PSPARQL. La syntaxe de PSPARQL est définie en remplaçant les patrons de graphes de SPARQL par des graphes PRDF. PRDF est un langage qui étend la syntaxe et la sémantique du langage RDF de façon à pouvoir étiqueter les arcs d'un graphe par des expressions régulières.

Le langage de requêtes **CPSPARQL** (*Constrained Path SPARQL*) est une extension de PSPARQL qui permet, en outre, d'exprimer d'autres constructions dans les requêtes SPARQL telles que des contraintes sur les sommets des chemins traversés. Cette extension a plusieurs avantages, parmi eux, elle ajoute plus d'expressivité à PSPARQL et améliore l'efficacité en utilisant des contraintes prédéfinies servant à couper les chemins inutiles. Les graphes CPRDF (*Constrained Path RDF*) sont aussi une extension des graphes PRDF pour le langage de requêtes CPSPARQL, permettant d'exprimer des contraintes sur les sommets des chemins traversés.

Les limites du langage SPARQL et les autres langages. Nous avons comparé ces différents langages d'interrogation de graphes RDF en nous basant sur différents travaux (Haase et al., 2004) (Hutt, 2005) (Angles et Gutierrez, 2005) et (Alkhateeb, 2008).

Dans le Tableau 8, les colonnes représentent les langages de requêtes et les lignes représentent des caractéristiques des types de requêtes. En outre, nous utilisons "-" pour indiquer que la fonction n'a pas de support dans le langage de requête, "+" pour désigner qu'il existe un support partiel, et "√" pour désigner un support complet de la fonction. Le Tableau 8 résume les principales différences entre les extensions actuelles de SPARQL et le langage SPARQL lui même. Comme il est possible de le remarquer, beaucoup de fonctionnalités dans SPARQL et ses extensions ne peuvent pas être exprimées dans les langages actuels comme G +, GraphLog, et d'autres.

	SPARQL	SPARQ2L	SPARQLeR	Corese	(P/CP)SPARQL	RQL	SeRQL	RDQL	Triple	N3	Versa	RxPath	G+	GraphLo	Gram	GraphDB	Lorel	F-G
Nœuds adjacents	+	√	√	√	√	+	+	+	+	+	+	-	√	√	√	+	√	+
Arcs adjacents	+	√	√	√	√	+	+	+	+	-	-	-	√	√	√	+	√	+
Degré d'un nœud	-	-	-	-	-	+	-	-	-	-	-	-	√	√	-	?	-	-
Chemin	-	+	+	-	+	-	-	-	-	-	-		√	√	√	√	√	√
Chemin de longueur fixe	-	+	+	√	√	+	+	+	+	+	-	+	√	√	√	√	√	√
Distance entre deux nœuds	-	-	-	-	-	-	-	-	-	-	-	-	√	√	-		-	-
Diamètre	-	-	-	-	-	-	-	-	-	-	-	-	√	√	-		-	-
Quantifi-cation	-	-	-	-	-	√	√	-			-		-	-			√	
Agrégation	-	+	-	+	+	√	-	-			√		-	-			√	
Chemin inverse	-	-	√	-	-/√	-	-	-			-		√	√				
Chemin incluant les cycles	-	√	-	-	√	-	-	-			-		-	-			-	

Tableau 8. Comparaison des langages de requêtes

"√" indique un support complet, "+" support partiel et "-" pas de support.

Il y a plusieurs caractéristiques des langages de requêtes pour les graphes RDF, mais le tableau ci-dessus prend en considération les caractéristiques suivantes :

- **Arcs adjacents** : Les langages ne possedent pas tous cette caractéristique. Le problème est que celle-ci peut être exprimée uniquement comme une association de deux requêtes : une pour les arcs sortants et une autre pour les arcs entrants. Quelques langages ne soutiennent pas l'opérateur d'association.
- **Nœuds adjacents** : cette caractéristique ressemble à la précédente sauf qu'elle est pour les nœuds au lieu des arcs.
- **Degré d'un nœud** : Le degré d'un nœud est le nombre d'arêtes ayant une extrémité en ce nœud (chaque boucle augmente de deux le degré). La plupart des langages ne soutiennent pas l'agrégation à ce niveau.
- **Distance entre nœuds** : n'est pas soutenue par plusieurs langages, cette fonctionnalité cherche à compter le nombre de ressources ou nœuds séparant deux nœuds du graphe.

- **Diamètre** : cette caractéristique est basée sur la distance et les chemins. Elle n'est pas soutenue par plusieurs langages.
- **Chemin de longueur fixe** : cette caractéristique trouve tous les chemins de longueur fixe entre deux ressources (ou nœuds). Soutenue partiellement par plusieurs langages, utilisant une association de tous les modèles de chemins possibles (combinaison des directions de l'arc) de longueur x entre les ressources initiales et finales. Dans le cas général, ceci demande l'évaluation de 2^n sous-requêtes pour un chemin de longueur n.
- **Agrégation** : Des fonctions d'agrégations, comme MIN, MAX et COUNT, ont été soutenues longtemps dans le monde structuré du langage de requêtes SQL et pourraient fournir des bénéfices dans la communauté RDF. De toute façon, très peu de langages courants soutiennent l'agrégation.
- **Quantification** : Un prédicat existentiel sur un ensemble de ressources est satisfait si au moins l'une des valeurs satisfait le prédicat. De façon analogue, un prédicat universel est satisfait si toutes les valeurs satisfont le prédicat. Par exemple, une requête qui retourne toutes les personnes qui sont auteurs de toutes les publications.
- **Chemin inverse** (*inverse path*): comme son nom l'indique, c'est le chemin inverse entre deux ressources.
- **Chemin incluant les cycles** : c'est un chemin où les nœuds ne sont pas tous distincts, c'est-à-dire des chemins comportant des cycles.

Dans cette étude, nous nous sommes intéressés aux fonctionnalités et lacunes de SPARQL, donné comme le langage standard du W3C. Mais nous avons également comparé celui-ci avec les langages les plus connus d'interrogations de graphes du Web sémantique. Ces derniers s'inspirent bien sûr de SPARQL, l'étendent ou la complètent.

3 SPARQL ET LOGIQUE TEMPORELLE

Nous avons vu, précédemment, que le langage de requêtes SPARQL est le standard pour l'interrogation des données du Web sémantique. Mais il se caractérise par sa complexité et a plusieurs limites. Le but des travaux de recherches présentés ici dans le domaine de l'interrogation des données du web sémantique est d'améliorer certaines lacunes identifiées précédemment. Pour cela, nous utilisons les requêtes en logique temporelle pour simplifier les requêtes SPARQL et obtenir une plus grande expressivité avec les opérateurs en logiques temporelles. Cette section est divisée en deux parties. La première partie présente l'avantage de l'utilisation des requêtes en logique temporelle par rapport au langage de requêtes SPARQL et une deuxième partie présente la simplicité de l'utilisation des requêtes en logique temporelle.

3.1 EXPRESSIVITE

Un langage de requêtes des données du web sémantique, robuste, est caractérisé par son pouvoir d'expressivité sur le graphe sémantique. L'expressivité dans un langage de requêtes est très importante. Il faut prouver que les requêtes en logique temporelle sont, idéalement, plus expressives qu'en SPARQL :

- **Diamètre du graphe** : le diamètre d'un graphe est la plus grande distance entre deux sommets quelconques de ce graphe.
- **Calcule du chemin entre deux nœuds** : retourne le chemin (liste des nœuds) entre deux nœuds/ressources.
- **Nœuds adjacents** : les nœuds adjacents d'un nœud sont l'ensemble des nœuds qui précèdent et succèdent ce nœud.
- **Degré d'un nœud** : le degré d'un nœud est le nombre d'arcs ayant pour extrémité ce nœud.
- **Arcs adjacents** : cette caractéristique liste les arcs sortants et les arcs entrants d'un nœud/ressource.
- **Chemin de longueur fixe** : retourne le chemin (liste des nœuds) entre deux ressources d'une longueur définie par l'utilisateur.
- **Autres opérateurs** : par exemple, l'opérateur UNTIL de la logique temporelle n'a pas d'équivalent en requête SPARQL.

Pour comprendre les différentes caractéristiques décrites ci-dessus, nous nous basons sur la portion du graphe RDF suivante (Figure 41) (Angles et Gutierrez, 2005) :

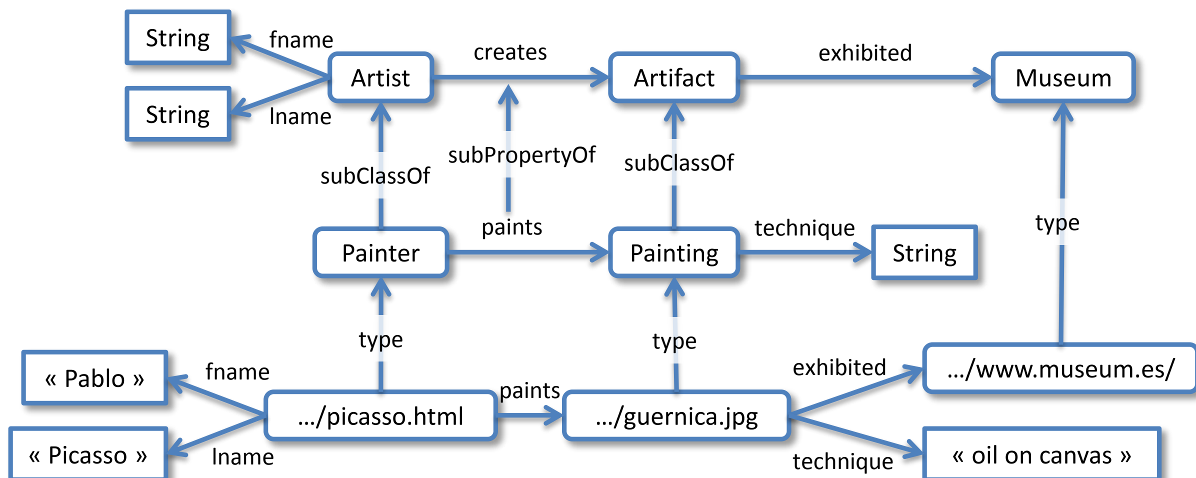


Figure 41. Le sous-graphe RDF représentant un musée.

La structure d'un document RDF est un graphe orienté étiqueté complexe. Il est composé d'un ensemble de triplets <sujet, prédicat, objet>. En outre, le prédicat (appelé aussi propriété) relie le sujet (ressource) à l'objet (valeur). Ainsi, le sujet et l'objet sont des nœuds du graphe reliés par un arc orienté étiqueté à partir du sujet vers l'objet.

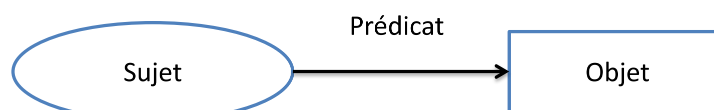


Figure 42. Triplet RDF.

Les **nœuds** sont des ressources ou des valeurs, et les **arcs** des propriétés. Les ressources sont représentées dans le graphe par des cercles, les propriétés par des arcs orientés et les valeurs par un rectangle, voir Figure 42. Voici l'analogie avec les opérateurs de la logique temporelle :

- pour accéder à l'objet, à partir du sujet : deux opérations **Next** sont nécessaires ;
- pour accéder au prédicat, à partir du sujet : une seule opération **Next** est nécessaire.

Diamètre d'un graphe. Avec l'option **–r** du model checker NuSMV (Tutorial NuSMV[17]), nous obtenons le diamètre du modèle représentant le graphe sémantique et aussi le nombre d'états atteignables dans le graphe.

Arcs adjacents. Cette caractéristique peut être obtenue avec une requête en logique temporelle. En réalité, c'est une requête composée de deux sous-requêtes. La première pour la récupération des arcs sortant de la ressource et la deuxième pour obtenir les arcs entrant à la ressource :

Eventually (ressource → **Next** ?prédicat1 & ?prédicat2 → **Next** ressource)

Par exemple, dans la Figure 41, les arcs adjacents de la ressource *guernica.jpg* sont *paints*, *type*, *exhibited* et *technique*.

Chemin entre deux ressources. Le chemin entre deux ressources peut se traduire en formule de logique temporelle. Pour cela, nous utilisons la négation de la formule permettant de vérifier qu'un état est un bien successeur d'un autre état. Ainsi, si ce n'est pas le cas, il nous retournera un contre-exemple (chemin en l'occurrence).

! **Eventually** (ressource1 → **Next** ressource2)

Le model checker NuSMV retournera le chemin complet de la ressource *ressource1* jusqu'à la ressource *ressource2*, s'il existe.

Le chemin entre la ressource *picasso.html* et *Museum* de la **Figure 41** est le chemin composé des nœuds et ressources suivantes : *paints*, *geurnica.jpg*, *exhibited*, *www.museum.es*, *type* et *Museum*. Le model checker NuSMV retournera un seul chemin aléatoire à la fois.

Chemin de longueur fixe. Le chemin entre deux ressources d'une longueur fixe donne la liste de toutes les ressources (nœuds) qui existent entre les deux ressources de cette longueur. Pour exprimer cette caractéristique avec une requête en logique temporelle, nous utilisons seulement la négation d'une formule en logique temporelle des deux ressources en les séparant avec un nombre fini d'opérateurs **Next**. Ce nombre représente le nombre de nœuds qui séparent les deux ressources. Par exemple, la logique temporelle ci-dessous, cherche le chemin de longueur deux qui sépare les deux ressources (resource1 et Resource2):

! **Eventually** (ressource1 → **Next Next Next Next** ressource2)

Il y quatre opérateurs **Next** parce que le chemin allant de la ressource *ressource1* à la ressource *ressource2*, passe par deux prédicats et deux objets. Les algorithmes du model checking retournent un contre-exemple qui contient le chemin entre les deux ressources, comme la formule en logique temporelle a été niée.

Dans la figure 42, le chemin de longueur deux entre la ressource *picasso.html* et la ressource *Painting* est le suivant : *type*, *Painter* et *paints*. Alors qu'un chemin de longueur trois entre ces mêmes ressources n'existe pas.

---

[17] http://nusmv.fbk.eu/NuSMV/tutorial/index.html

Nœuds adjacents. On peut avoir la liste de tous les nœuds adjacents d'une ressource donnée d'un modèle représentant un graphe sémantique. Pour cela il faut juste chercher avec une requête en logique temporelle les nœuds qui précèdent et succèdent la ressource en question. Comme pour les arcs adjacents, cette requête est composée de deux requêtes. Une pour la récupération des nœuds successeurs et une autre pour les nœuds prédécesseurs.

Eventually (ressource → **Next Next** ?ressource1 & ?ressource2 → **Next Next** ressource)

Toujours dans la même figure 42. Les nœuds adjacents de la ressource *picasso.html* sont : *Pablo*, *Picasso*, *Painter* et *guernica.html*.

Autres opérateurs. Une autre caractéristique peut se faire avec les formules de la logique temporelle. On peut aussi vérifier l'ordre des ressources dans une branche de notre modèle de graphe sémantique.

! **Eventually** (ressource1 **& Eventually** ressource2)

Cette formule vérifie si la ressource *ressource2* est située après la ressource *ressource1* dans une branche du graphe. Cette formule était vraiment très importante dans la détection des incohérences (contradictions) dans le log d'évolution des ontologies, ce qui nous a permis de mettre en œuvre la méthodologie **CLOCK**[18] (chapitre 7).

Par exemple, la ressource picasso.html ne se trouve pas après la ressource Painting, parce qu'il n'existe pas de chemin menant de la ressource *Painting* à la ressource *picasso.html*. Mais le contraire est vrai, il y a un chemin menant de la ressource *picasso.html* à la ressource *Painting* dans le graphe.

D'autres opérateurs peuvent aussi être utilisés, par exemple l'opérateur *until* qui cherche à vérifier si un état est toujours présent dans un chemin jusqu'à l'apparition d'un autre état sur son arborescence.

Remarque : Les requêtes en logique temporelle décrites ci-dessus sont écrites d'une façon générale c'est-à-dire ni en logique temporelle linéaire, ni en logique temporelle arborescente. Pour l'écrire dans une de ces logiques, il faut utiliser les opérateurs adéquats à la logique temporelle choisie.

3.2 LA SIMPLICITE

Dans (Neumann et Weikum, 2009) le problème en discussion est de choisir des jointures optimales pour des requêtes complexes SPARQL. Par exemple, notez la requête SPARQL ci-dessous:

```
SELECT ?x
where {
? x hasName? y.
? y hasAdress? z.
? z hasAge "26"
}
```

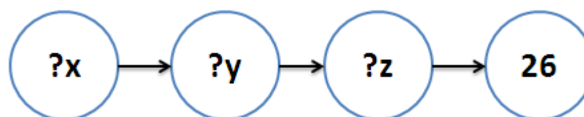


Figure 43. Requête SPARQL et sa représentation graphique.

---

[18] Change Log Ontology ChecKing

Cette requête sélectionne le sujet ?x qui a l'âge de 26 ans. Dans la requête en logique temporelle, la requête SPARQL devient :

Eventually (?x -> **Next Next Next Next Next Next** 26 & ?x -> **Next Next Next Next Next** hasAge)

Notez que la requête devient plus simple que dans le langage de requêtes SPARQL, en raison de l'utilisation des opérateurs de la logique temporelle. Nous utilisons six fois l'opérateur suivant **Next** dans la requête, car il y a trois nœuds et trois arcs pour accéder au nœud *26* représentant l'âge (Figure 44) et aussi cinq **Next** pour le prédicat *hasAge* situé à une distance de cinq (comptant le nombre de nœuds et arcs) du nœud ?x.

3.3 LANGAGE DE VERIFICATION

Les sections précédentes s'intéressent seulement à la forme **SELECT** des requêtes SPARQL. Cette forme permet de combiner les opérations de projection des variables requises avec l'introduction de nouvelles liaisons de variables dans la solution de la requête. La suite de ce chapitre présente un autre type de requêtes SPARQL : la forme **ASK**. Cette forme de requête permet de vérifier ou de tester si un motif d'une requête a une solution ou non. Aucune information n'est retournée sur les solutions possibles de la requête, seulement une réponse *vrai /oui* s'il existe une solution ou *faux/non* sinon.

Les algorithmes du model checking sont des algorithmes très puissants pour la vérification des propriétés sur des graphes ou modèles de systèmes complexes (voir chapitre 3). Beaucoup de travaux ont montré leur puissance par rapport aux autres méthodes formelles (Baier et Katoen, 2008). Une autre solution et amélioration des requêtes en langage SPARQL est l'utilisation de ces algorithmes pour la vérification des patrons du graphe sémantique (les triplets RDF). Pour pouvoir arriver à cette simplification et amélioration des requêtes SPARQL, il suffit de transformer les requêtes de la forme **ASK** en formules de la logique temporelle.

La forme d'une requête SPARQL de la forme ASK est la suivante :

ASK {modèle de graphe}

Les requêtes sont basées sur la notion de modèles de graphes que nous cherchons à retrouver dans une base de triplets. Dans un modèle de graphe, un triplet peut comporter des variables (i.e. des identificateurs précédés par ?) qui seront instanciées lorsqu'un triplet concorde au modèle en affectant une variable.

Always (modèle de graphe)

Dans le modèle de graphe, les prédicats de tous les triplets seront remplacés par des flèches et s'il y a plusieurs triplets, ils seront séparés par l'opérateur « **et** » de la logique temporelle. Par exemple, pour vérifier que la ressource *artificat* est suivie par la ressource *Museum* dans la figure 42, la requête SPARQL et sa requête en logique temporelle équivalente sont :

ASK { *artifact* exhibited *Museum* }	**Always** (*artifact* → **next** *Museum*)

Pour cet exemple, le model checker retournera la valeur vrai car la ressource *Museum* est vraie après la ressource *artifact*. Le model checker est plus performant que les requêtes SPARQL de la forme ASK parce qu'il a été conçu pour la vérification des systèmes complexes.

Nous avons testé notre outil de résolution sur un graphe sémantique composé de 10 116 triplets. Nous l'avons comparé avec le moteur de recherche sémantique CORESE[19] (Corby et al., 2004). Les requêtes décrites ci-dessus ont été composées avec la logique temporelle linéaire.

Pour une requête de type sélection :

SPARQL	SELECT ?x WHERE
	{
	?x ?t
	<http://www.le2i.com/ifc2rdf/ifcIndividual#ifcownerhistory_33>
	}
		0.1s

Requête en logique temporelle	F(?x->X Xshttp://www.le2i.com/ifc2rdf/ifcIndividual#ifcownerhistory_33)	12.292s

Pour une requête de type jointure :

SPARQL	SELECT ?k WHERE
	{
	<http://www.le2i.com/ifc2rdf/ifcIndividual#ifcstyleditem_365> ?t ?z
	.
	?z ?v ?k
	}
		0.1s

Requête en logique temporelle	F(shttp_--www_le2i_com-ifc2rdf-ifcIndividual_ifcstyleditem_365->X X X X ?x)	13.927s

Nous remarquons que le temps d'exécution est supérieur à celui du moteur de recherche CORESE. De plus, l'explosion combinatoire est encore présente, comme le montre les résultats. La figure suivante montre la limite de notre approche.

[19] COnceptual REsource Search Engine

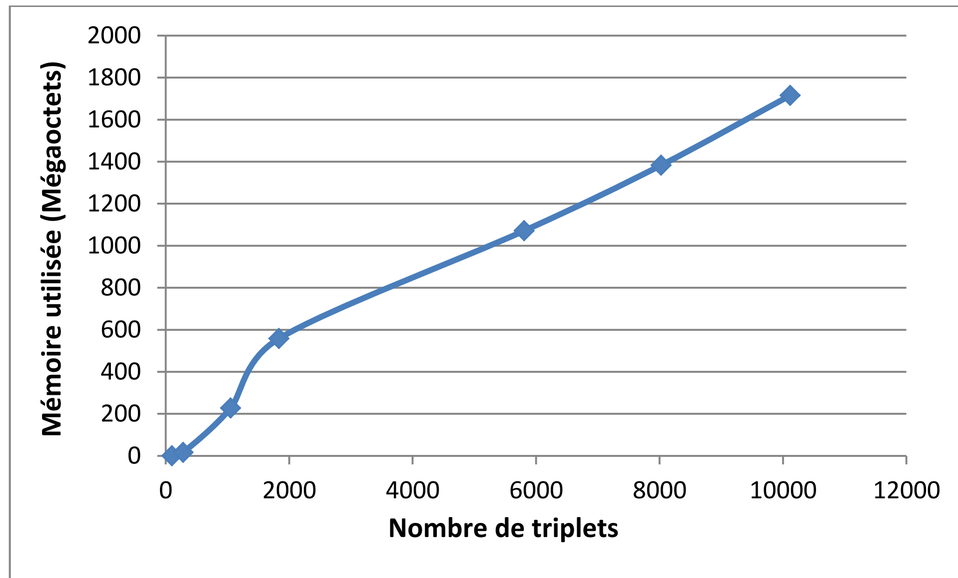


Figure 44. Mémoire utilisée par l'outil STL Resolver.

La mémoire utilisée par l'ordinateur (Figure 44) augmente avec l'augmentation du modèle représentant le graphe sémantique (en nombre de triplets). L'expérience a été arrêtée lorsque l'outil n'a pas eu assez de mémoire pour lancer le calcul (à l'étape 17 603 triplets). Le probleme de l'explosion combinatoire n'es pas traité dans ce travail de recherche et sera dans nos perspectives.

3.4 Extension des requetes

Nous avons vu la transformation des requêtes en logique temporelle utilisant les opérateurs de la logique temporelle CTL. Nous pouvons étendre la transformation des requêtes à la logique temporelle linéaire qui utilise des opérateurs du passé. Dans cette section, nous expliquons comment arriver à cette extension.

Nous avons vu dans le chapitre 3, les formules en logique temporelle linéaire sans les opérateurs du passé. Voici les différents opérateurs du passé introduits dans le model checker NuSMV :

- *p* V *q* est vrai au temps *t* si *q* est vraie à toutes les étapes de temps $t' \geq t$ jusqu'à et incluant le temps *t''* où *p* est également vrai. Alternativement, *p* peut être jamais vérifiée, dans ce cas, *q* doit être vraie dans toutes les étapes de temps $t' \geq t$.
- Y *p* est vrai au temps *t*> 0 si *p* est vrai à l'instant *t*-1. Y *p* est faux à l'instant t_0.
- Z *p* est équivalente à Y *p*, à l'exception que l'expression est vrai à l'instant t_0.
- H *p* est vrai à l'instant *t* si *p* est vrai dans toutes les étapes à l'instant t_0 précédentes $t' \leq t$.
- O *p* est vrai à l'instant *t*, si *p* est vrai dans au moins une des étapes précédentes de temps $t' \leq t$.
- *p* S *q* est vrai à l'instant *t* si *q* est vrai à l'instant $t' \leq t$ et *p* est vrai dans toutes les étapes de temps de *t'* à l'instant *t* compris.
- *p* T *q* est vrai à l'instant *t* si *p* est vrai à l'instant $t' \leq t$ et *q* est vrai dans toutes les étapes de temps de *t'* à *t* compris. Par ailleurs, si *p* n'a jamais été vrai, alors *q* doit être vrai dans toutes les étapes de temps de *t'* à *t*.

4 LE STL-RESOLVER

L'outil STL-Resolver a fait l'objet d'un dépôt APP, suite à son développement par l'équipe et moi-même pendant ma thèse. C'est un moteur de requêtes pour les modèles de graphes sémantiques qui utilise les requêtes en logiques temporelles en se basant sur les algorithmes de parcours du model checker NuSMV. NuSMV a été notre choix pour l'implémentation du STL-Resolver par rapport au model checker SPIN en raison de ses avantages multiples :

- nos modèles de graphes sémantiques en langage SMV ne sont pas limités en nombre d'états ;
- la vérification se fait directement, sans passer par un compilateur intermédiaire;
- il implémente les deux types de logiques temporelles : la logique temporelle linéaire et la logique temporelle arborescente pour la vérification (chapitre 4) ;

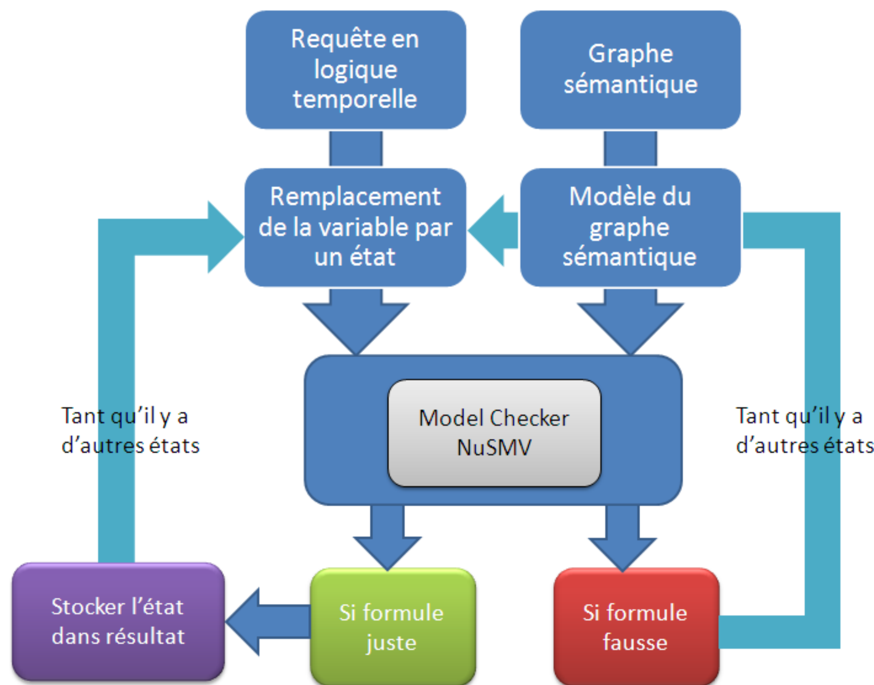


Figure 45. Schéma général du fonctionnement de l'outil STL-Resolver.

L'architecture de la figure 45 explique le fonctionnement de l'outil STL-Resolver. Dans un premier temps, le modèle du graphe sémantique (ou graphe) est écrit en langage SMV par l'outil RDF2NuSMV. Dans un second temps, l'outil STL Resolver prend en entrée le modèle du graphe sémantique et une requête en logique temporelle. Cet outil récupérera tous les états du modèle et transforme la requête en logique temporelle en une liste de formules de logiques temporelles (c'est-à-dire sans variable ou *joker* dans la requête) en remplaçant à chaque fois la variable de la requête en logique temporelle par un état de notre graphe. Ensuite, toutes les formules seront testées, en les donnant en paramètre au model checker NuSMV. Si la formule est juste, donc l'état remplaçant la variable de la requête est une solution, sinon un autre état sera cherché (s'il en reste) et le processus décrit ci-dessus sera réitéré.

```
Algorithme : STL RESOLVER
Input: TEMPORAL LOGIC QUERY + NUSMV MODEL
Output: SET OF STATES
Begin
   For all triplet in the query do
       If the triplet_i contain a variable then
             If the triplet_i.Object = next variable then
                   Stack = push (Successor (Triplet_i.Subject));
             End if

             Else if the triplet_i.Object = next next variable then
                   Stack1 = push (Successor (Triplet_i.Suject));
                       While stack1 not nil do
                           y = top (stack1);
                           Stack = push (Successor (y));
                       End do
             End if

             Else if the triplet_i.Object = next next …variable then // next operator more than two
                   Stack = push (Successor (….Successor (Triplet_i.Suject));
             End if

             Else if the triplet_i.Subject =variable then
                   For all vertex x of the graph do
                         Stack1= push (Successor (x));
                         If Triplet_i.Object ∈ stack1 then
                               Stack = push (x);
                         End if
                   End do

             End if

       Else
             i++;
   End do
   Return stack;
End
```

Script 21. Algorithme de résolution de requêtes en logique temporelle.

L'algorithme de résolution (Script 21) traite chaque triplet de la requête en logique temporelle. Si la variable dans le triplet est l'objet, alors la solution sera le successeur du Sujet. Mais cela dépendra également du nombre d'apparitions de l'opérateur **Next**. Si celui-ci apparaît deux fois, alors la solution sera le successeur du successeur du Sujet. Dans le cas où la variable est le sujet, alors la solution sera les prédécesseurs de l'objet. Enfin, la fonction *successeur* retourne les nœuds successeurs au nœud donné en paramètre.

5 SPARQL VERS RLT

Les opérateurs algébriques qui sont définis dans SPARQL ressemblent aux opérateurs algébriques définis dans l'algèbre relationnelle :

- en particulier, l'opérateur **AND** est transformé à une jointure algébrique,
- le **filtre** à un opérateur de sélection algébrique,
- l'**Union** à un opérateur d'union,
- l'**option** à une jointure externe gauche (ce qui permet le remplissage facultatif de l'information),

- et **Select** à un opérateur de projection.

La forme de requête **Select** est la forme la plus utilisée dans les requêtes SPARQL. Dans cette section, nous ne présentons que la requête SPARQL avec la forme Select. Une requête de base SPARQL a la forme suivante :

Select ?variable1,
?variable2,… **Where**
{pattern1. pattern2. …}

où chaque motif se compose d'un sujet, prédicat, objet, et chacun d'eux est une variable ou un littéral. La requête spécifie les littéraux connus et laisse les inconnues comme variables. Les variables peuvent apparaitre dans plusieurs motifs et impliquent donc la jointure. Le processeur de requêtes doit trouver toutes les liaisons possibles de variables qui satisfont les modèles donnés et ainsi retrouver les liaisons de la clause de projection, à la demande. Notez que toutes les variables ne sont pas nécessairement liées.

L'algèbre relationnelle (Cyganiak, 2005) est introduite pour faciliter la cartographie de la requête SPARQL qui nous permettra de l'utiliser dans les applications de la logique temporelle. Nous définissons les opérateurs dans les relations RDF (Gueffaz et al., 2012b).

4.1 SELECTION

La Sélection σ, parfois appelée la restriction, est un opérateur unaire qui ne sélectionne que les *tuples* d'une relation pour laquelle une formule propositionnelle est juste.

SPARQL	SELECT ?x WHERE { ?x nom "Paul"}
Requête en logique temporelle	Eventually (?x → Next "Paul")
Algèbre relationnelle	$\pi_{?x} \rightarrow \sigma_{\substack{?predicate=Name\wedge \\ ?object=Paul}} \rightarrow Triples$

4.2 PROJECTION

L'opérateur de projection π restreint une relation à un sous-ensemble de ses attributs. Voir l'exemple de la section 5.1.

4.3 JOINTURE

La jointure interne ⋈ joint deux relations sur leurs attributs partagés. La jointure $A \bowtie B$ contient toutes les tuples de A et de B, privé de ceux ou les attributs communs ne sont pas égaux.

La jointure externe gauche (: ⋈) contient en outre tous les *tuples* de la première relation qui n'ont pas de *tuples* appartenant à la seconde.

SPARQL	SELECT ?x WHERE { ?x composed_by ?y. ?y name "Bob Dylan"}
Requête en logique temporelle	Eventually (?x → Next Next "Bob Dylan")

4.4 UNION

L'union ∪ de deux relations A et B est l'ensemble d'union des tuples de A et de B. Contrairement à l'algèbre relationnelle régulière, les ensembles A et B n'ont pas besoin d'être identiques.

SPARQL	SELECT ?x ?y WHERE { {?x name "John"} UNION { ?y name "Paul"}}
Requête en logique temporelle	Eventually (?x → Next John & ?y → Next Paul)

Le but d'une requête OPTIONAL en langage SPARQL est de compléter la solution avec des informations supplémentaires. Si le modèle dans une clause OPTIONAL correspond, les variables définies dans ce modèle sont liées à une ou à plusieurs solutions. Si le modèle ne correspond pas, la solution reste inchangée. La requête SPARQL représentée ci-dessous sélectionne le sujet ?x qui a *Paul* et / ou *paul@yahoo.fr* comme objet.

SPARQL	SELECT ?x WHERE { ?x name "Paul" OPTIONAL {?x email paul@yahoo.com"}}
Requête en logique temporelle	Eventually (?x → Next "Paul" & Eventually ?x → "paul@yahoo.com")

```
Algorithme: SPARQL2RLT
Input: SPARQL QUERY Q
Output: TEMPORAL LOGIC QUERY
Begin

    If there is a PROJECTION then
        For all triplet do
            Triplet ();
            Transition = +"&";
        End do
    End if

    If there is a JOIN then
        For all triplet do
            Triplet ();
            Transition = +"&";
        End do
    End if

    If there is a UNION then
        For all triplet do
            Triplet ();
            Transition = + "&";
        End do
    End if

    If there is an OPTIONAL then
        For all triplet do
            Transition = + "& Eventually";
            Triplet ();
        End do
    End if

    Return "Eventually (" transition ")";
End
```

Script 22. Transformation SPARQL vers requêtes avec opérateurs de la logique temporelle.

L'algorithme du Script 22, transforme les requêtes en langage SPARQL en requête utilisant les opérateurs de la logique temporelle linéaire et arborescenete. La transformation dépend de la requête SPARQL. Si c'est une requête de type *projection*, *jointure* ou *union*, nous transformons chaque triplet de la requête en un triplet utilisant les opérateurs de la logique temporelle avec la fonction **Triplet** définie dans le Script 23 ci-dessous et nous les séparerons avec l'opérateur **&**. Par contre, si la requête SPARQL est de type *optional*, comme cette forme ajoutera des informations au résultat de la requête SPARQL, la requête en logique temporelle fera précéder tous les triplets contenus dans la partie OPTIONAL de la requête par le mot clé *Eventually*. Ce mot clé ajoutera les informations si elles existent dans le modèle du graphe sémantique.

```
Algorithme : TRIPLET()
Input: SPARQL QUERY
Output: TRANSITION
Begin
    If there is a SELECTION then
        If there selection of the subject or the object then
            Subject = Subject;
            Predicate =→;
            Object = "Next Next Object";
            Transition = triplet;
        End if

        Else if there is selection of the predicate then
            Subject = Subject;
            Predicate =→;
            Object = "Next Predicate";
            Transition = triplet;
        End if
    End if

    //Optimization des triplets
    For all triplet in the Transition do
        If there is a triplet_i.Object = triplet_i+1.Subject then
            Triplet_i.Object = +"triplet_i+1.Object";
            Delete triplet_i+1;
        End if
        Else if there is a triplet_i.Subject= triplet_i+1. Object then
            Triplet_i+1.Object = +"triplet_i.Object";
            Delete triplet_i;
        End if
    End do
  Return Transition;
End
```

Script 23. Algorithme pour l'extraction et la transformation des triplets RDF.

L'algorithme **Triplet** du Script 23, traite chaque triplet de la requête SPARQL. Le prédicat de chaque triplet sera remplacé par →, le sujet restera toujours le même et l'objet aussi sauf si c'est une variable. Dans ce cas, il sera remplacé par **Next Next variable.** Dans le cas où nous avons une sélection sur le prédicat, nous remplacerons le prédicat par → et l'objet par un **Next variable**.

La dernière partie de cet algorithme est la partie optimisation. Elle sert à réduire le nombre de triplets dans la requête en logique temporelle.

L'outil SPARQL2RLT (dépôt APP) transforme les requêtes SPARQL en requêtes utilisant les opérateurs de la logique temporelle. Pour le développement de cet outil, nous avons décidé d'utiliser LEX & YACC pour la décomposition de la requête SPARQL en arbre afin de faciliter la transformation. LEX sert à reconnaître les entités lexicales et à les remplacer par des mots clé qui seront reconnus dans la grammaire du langage défini dans la grammaire écrite en syntaxe YACC (*Yet Another Compiler Compiler*). Par la suite, YACC va reconnaître et vérifier si les expressions respectent ou non cette grammaire. LEX & YACC sont deux outils très performants, facilitant l'analyse lexicale et syntaxique, respectivement deux étapes difficiles de la compilation.

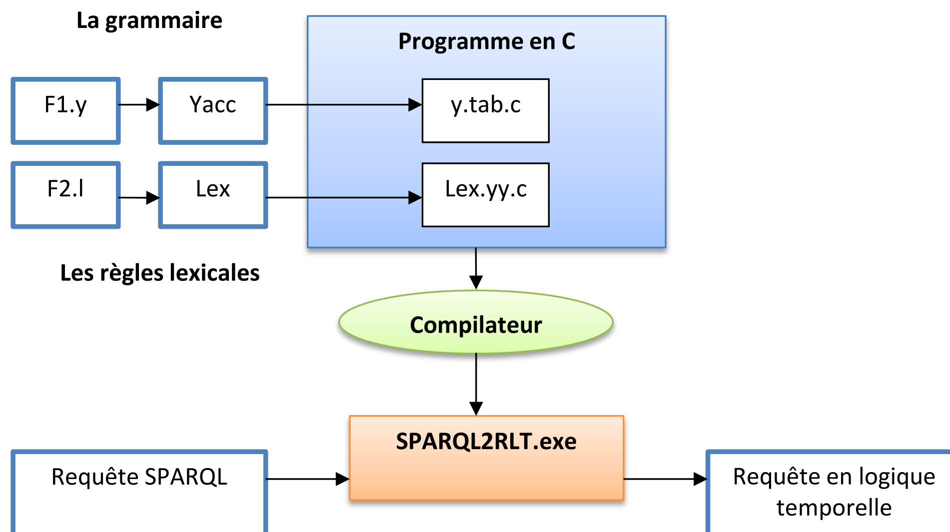


Figure 46. Architecture de l'outil de conversion SPARQL2RLT.

L'analyse lexicale repère, dans le texte source, des éléments significatifs appelés « unités lexicales » ou « Token ». Les unités lexicales peuvent être des ponctuations, mots clés, constantes, identificateurs sans tenir compte des commentaires, et des espacements. Dans notre cas, la requête SPARQL peut être composée par :

- **Les mots clés** : PREFIX, SELECT, ASK, WHERE, UNION, OPTIONAL, FILTER, REGEX.
- **Les opérateurs** : on distingue deux types :
  - Les opérateurs arithmétiques : + - / *
  - Les opérateurs booléens : = != < > <= >=
- **Les séparateurs** : caractères de ponctuation tels que : , ; . : { } ( )
- **Les identificateurs** : ce sont des mots formés par l'utilisateur pour les variables, les URL, et pour la définition des triplets RDF.
- **Les constantes** : entiers et réels.

Remarque 1 : il n'existe pas de commentaires dans les requêtes SPARQL

La procédure de développement d'un analyseur lexical consiste en la description des entités lexicales à l'aide d'expressions régulières. Ensuite, il suffit de transformer ces expressions régulières en automates équivalents qui seront implémentés sur machine par des matrices appelées « Tables de transitions ».

Remarque 2 : l'analyseur lexical supprime les blancs automatiquement.

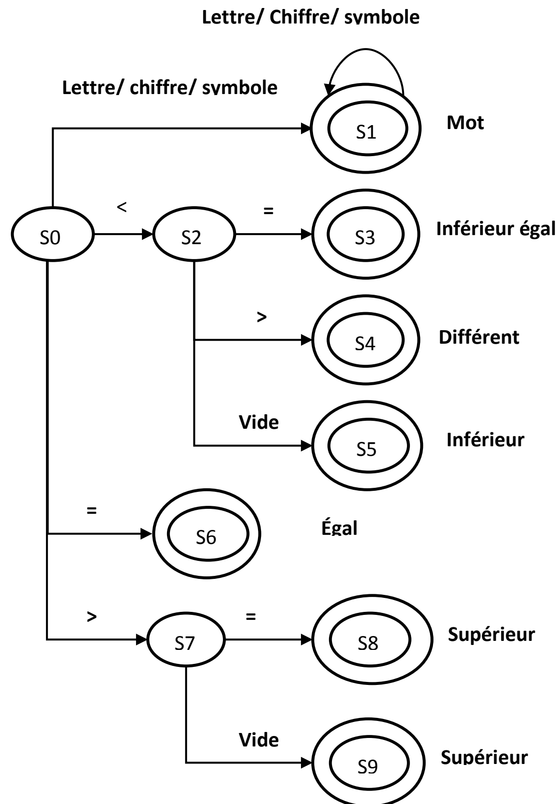


Figure 47. Automate à états finis reconnaissant les opérateurs booléens et les mots.

Les symboles peuvent être des : un point, un anti-slache, un slache, un tiret, un souligné.

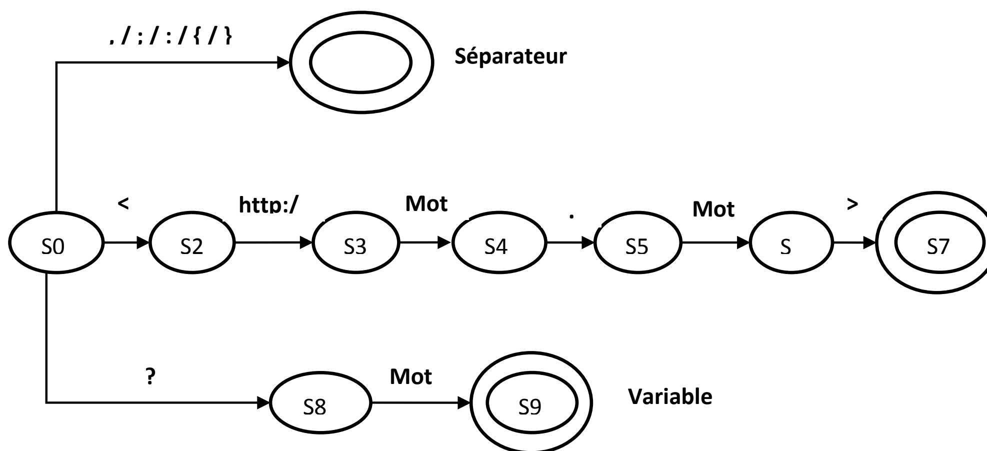


Figure 48. Automate à états finis de l'analyseur lexical.

Notre fichier *lex* est le suivant :

```
blancs [ \t]
ask [Aa][Ss][Kk]
select [Ss][Ee][Ll][Ee][Cc][Tt]
where [Ww][Hh][Ee][Rr][Ee]
prefix [Pp][Rr][Ee][Ff][Ii][Xx]
filter [Ff][Ii][Ll][Tt][Ee][Rr]
```

```
regex [Rr][Ee][Gg][Ee][Xx]
URL      "<"("http://"|"https://"){0,1}[A-Za-z0-9][A-Za-z0-9\-\.]+[A-Za-z0-
9]\.[A-Za-z]{2,}[\43-\176]*">"
var [a-zA-Z0-9]+
varSparql [?][a-zA-Z0-9]+

%%

{blancs} {}
{ask} {return ASK;}
{select} {return SELECT;}
{where} {return WHERE;}
{prefix} {return PREFIX;}
{filter} {return FILTER;}
{regex} {return REGEX;}
{URL} { yylval=strndup(yytext+1,strlen(yytext)-2); return URL;}
{var} { yylval=strdup(yytext); return var;}
{varSparql} {yylval=strdup(yytext+1); return varSparql;}

"{" return(AC_OUV);
"}" return(AC_FER);
"\n" return(FIN);
"," return(VIRG);
":" return(PP);
"\"" return (DC);
```

Script 24. Analyse Lexicale des requêtes

La structure générale d'un fichier *Lex* est formée de trois parties délimitées par le symbole %% comme suit :

1. **Partie déclaration** : vous pouvez mettre deux types de déclarations. Soit des déclarations ou des #define pour votre code C, ou bien des définitions d'expressions régulières pour le Lex. Lex différencie les deux types de définitions par la mise en forme que vous donnez à vos lignes. Par défaut, les définitions pour Lex sont des définitions qui commencent au premier caractère de chaque ligne. Les définitions pour le code C, peuvent être des #include, des déclarations de variables, types, ou constantes (#define) doivent être mis en forme différemment. Cette partie a servi à définir les mots clés du langage SPARQL en se basant sur les automates de la figure 47 et la figure 48.
2. **Partie règles** : dans cette section, vous pouvez définir une action pour chacune des expressions régulières que vous recherchez.
3. **Partie fonctions** : cette dernière partie permet de faire de simples calculs et affiche le résultat à la fin de l'analyse. Dans notre cas, cette partie n'est pas nécessaire.

Une fois l'analyse lexicale terminée, l'analyse syntaxique permet de vérifier si un texte est conforme à une grammaire et de construire l'arbre syntaxique correspondant. La grammaire de notre analyseur syntaxique est la suivante (les mots en gras sont des non-terminaux) :

```
Expression   →   Pr Type Suite
Pr           →   PREFIX Mot : < URL > Pr
                 | ε
Type         →   SELECT varSparql WHERE
                 | ASK
varSparql    →   ? Mot varSparql
                 | ? Mot
Suite        →   { Triplet Type1 }
Triplet      →   Type2 Type3 Type2 Triplet
                 | Type2 Type3 Type2
type1        →   UNION Triplet
                 | OPTIONAL Triplet
                 | FILTER
                 | ε
type2        →   varSparql
                 | " Mot "
                 | URL
type3        →   Mot : Mot
                 | URL
URL          →   http:// Mot . Mot
Mot          →   Lettre Mot  | lettre Fin
                 | Chiffre Mot | chiffre Fin
                 | symbole Mot | symbole Fin
Fin          →   ε
```

Script 25. Grammaire des requêtes.

5 TRAVAUX CONNEXES

L'approche présentée dans (Mateescu et al., 2009) s'appuie sur l'environnement BCG pour une représentation graphique compacte dotée avec XTL, un langage général en mesure de décrire les graphes, d'extraire et de manipuler les valeurs des données contenues dans les étiquettes de transitions. De ce point de vue, XTL offre des manipulations graphiques de la même nature que celles disponibles dans XQuery pour la recherche dans les documents XML, et permet de librement intégrer les opérations relationnelles et celles basées sur les graphes.

La boîte à outils CADP (*Construction and Analysis of Distributed Processes*), est un ensemble d'outils conçus pour pratiquer des opérations de vérification formelle sur des processus distribués, comme des programmes complexes ou des protocoles réseau, représentés sous la forme de graphes appelés LTS (*Labeled Transition System*). La boite à outils contient un langage nommé XTL (*eXecutable Temporal Language*) qui est un métalangage adapté à l'expression des algorithmes d'évaluation et de diagnostic pour les formules de logiques temporelles telles que CTL (Clarke et al., 1986), Hml (Hennesy et Milner, 1985), Actl (Nicola et Vaandrager, 1990), etc. D'inspiration fonctionnelle, ce métalangage offre des primitives d'accès à toutes les informations contenues dans les graphes BCG : états, étiquettes des transitions, fonctions successeurs et prédécesseurs, ainsi

qu'aux types et fonctions du programme source. Il permet la définition de fonctions récursives servant à calculer des prédicats de base et des modalités temporelles portant sur les ensembles d'états et de transitions. Le format BCG utilise des techniques efficaces de compression permettant de stocker des graphes (sous forme explicite) sur disque de manière très compacte. Ce format est indépendant du langage source et des outils de vérification. En outre, il contient suffisamment d'informations pour que les outils qui l'exploitent puissent fournir à l'utilisateur des diagnostics précis dans les termes du programme source. Pour exploiter ce format, un environnement logiciel est disponible, qui se compose de bibliothèques C et de plusieurs outils, notamment: *Bcg_Io* (qui effectue des conversions de format), *Bcg_Open* (qui permet d'appliquer à des graphes BCG les outils de l'environnement Open/Caesar pour la vérification à la volée), *Bcg_Draw* (qui permet d'afficher en PostScript une représentation 2D d'un graphe), et *Bcg_Edit* (qui permet de modifier interactivement la représentation graphique produite par *Bcg_Draw*).

CADP n'a pas été conçu pour le web sémantique afin de manipuler les graphes sémantiques (graphe RDF ou graphe OWL). Pour cela, CADP à développer l'outil *RDF.Open* afin de convertir un graphe sémantique en graphe BCG en utilisant l'API Open/ CAESAR. La transformation de graphe sémantique en modèle de graphe sous le format BCG passe par une étape intermédiaire ce qui augmentera le risque de perte d'informations et d'erreurs dans le modèle de graphe sémantique. Un autre inconvénient de la transformation est la taille des graphes BCG. Un graphe RDF de 30 mégaoctets donne un modèle de graphe équivalent avec une taille de 1,2 giga-octet octets en format BCG, soit un facteur multiplicateur de 40. Alors qu'avec notre approche ScaleSem, la taille du modèle du graphe sémantique est presque identique à celle du graphe sémantique (en RDF).

Il est à noter que le format BCG souffre de nombreuses restrictions qui deviennent problématiques pour la manipulation d'un graphe sémantique. Nos graphes NuSMV représentent les graphes sémantiques en gardant toutes les informations des triplets du graphe donc il n'y a pas de risque de perte d'information. Ci-dessous une autre limite de la boite à outils CADP pour l'interrogation des graphes sémantiques (voir Tableau 9).

Requête SPARQL	Langage XTL	Requête en logique temporelle
Select ?nom **Where** { Bureau19 occupant personne Personne nom ?nom}	**ITER_BGP_S**(!''Bureau19'', !''occupant'', ?personne :String, **ITER_BGP_P**(!personne, !''nom'', ?nom :String, **SPARQL_PRINTLN**(nom)))	F ((state=Bureau19 → X X state=?nom) & (Bureau19 → X X msg=nom))

Tableau 9. Comparaison des langages d'interrogation.

Ce tableau compare trois types de langages d'interrogations des données du web sémantique. Ces requêtes vont nous permettre de connaitre le nom de la personne occupant le bureau numéro 19 en utilisant une jointure. On remarque que la requête XTL est très compliquée à comprendre et difficile à écrire aussi par rapport à une requête en logique temporelle. Cette simplicité vient des opérateurs de la logique temporelle qui sont très proches des modalités de la langue naturelle.

	Nombre d'outils utilisés pour la traduction d'un graphe sémantique	Taille des graphes de sortie	Risque de perte de l'information	Langage d'interrogation
L'approche ScaleSem	1	Le modèle a presque la même taille que le graphe sémantique	Non	Claire et simple
La boite à outils CADP	2	La taille du modèle est très grande, soit un facteur multiplicateur de 40	oui	Très ambiguë

Tableau 10. Comparaison de la boite à outils CADP et l'approche ScaleSem.

Le Tableau 10 présente une synthèse des différences qui existent entre la boite à outils CADP et l'approche ScaleSem que nous avons mise en œuvre. Notre approche pour l'interrogation de graphe sémantique utilise un modèle de graphe sémantique et un langage basé sur les opérateurs de la logique temporelle CTL et LTL afin de simplifier les requêtes SPARQL et aussi pour augmenter leurs pouvoirs d'expressivité. Les travaux de (Neumann et Weikum, 2009), n'ont pas essayé d'inventer un nouveau langage d'interrogation de graphes sémantiques, mais ils se sont orientés dans la résolution du problème des jointures en proposant un système d'indexation des triplets RDF.

Les travaux (Schmidt, 2009) portent sur l'optimisation sémantique et algébrique des requêtes SPARQL. Il a comparé les performances de plusieurs moteurs de requête SPARQL et a découvert le déficit et les potentiels d'optimisation qui pourraient être considérés dans une future mise à jour de SPARQL.

6 CONCLUSION

Ce chapitre permet d'identifier les limites dans l'expressivité des requêtes en langage SPARQL (chemin de longueur fixe, chemin entre deux ressources, etc…). Il a démontré l'intérêt de l'utilisation des requêtes en logique temporelle pour l'interrogation des graphes sémantiques. Le model checking est une méthode qui a été conçue pour la vérification des systèmes en utilisant des propriétés en logique temporelle. Les algorithmes du model checking sont des algorithmes très puissants pour la vérification des propriétés sur des graphes ou modèles de systèmes complexes. Beaucoup de travaux ont montré leurs puissances par rapport aux autres méthodes formelles (Baier et Katoen, 2008). La solution que nous proposons avec notre outil STL Resolver est une amélioration des requêtes en langage SPARQL grâce à l'utilisation de ces algorithmes pour la vérification des patrons du graphe sémantique (les triplets RDF). L'avantage des formules en logique temporelle est leur facilité d'utilisation et leur puissance d'expression des formules de vérification sur les modèles de graphes sémantiques. Nous avons aussi introduit un nouvel outil SPARQL2RLT pour transformer les requêtes en langage SPARQL en requêtes utilisant les opérateurs de la logique temporelle linéaire (LTL) ou arborescente (CTL). L'objectif final de ces transformations est de résoudre les lacunes des requêtes SPARQL sur les gros graphes et plus précisément sur les jointures. Ce problème fait actuellement l'objet de recherches intensives pour réduire le temps d'interrogation des gros graphes.

Les algorithmes du model checking sont très performants pour le parcours des graphes, mais ils sont confrontés au problème de l'explosion combinatoire. Notre approche pour l'interrogation des graphes sémantiques, n'a pas donné de très bons résultats en termes de rapidité, mais elle permet d'ajouter de l'expressivité et la simplicité aux requêtes en langage SPARQL.

Chapitre 7

La méthodologie CLOCk

Résumé

Depuis moins de 10 ans, le domaine de l'évolution du concept de l'ontologie a connu un engouement croissant. Le développement des solutions d'interopérabilité à base d'ontologies et le coût des modifications requièrent de nouvelles approches pour prendre en compte le changement, donc l'évolution des ontologies. Tout changement peut mener à des contradictions au sein de l'ontologie. Il est indispensable de vérifier la consistance de l'ontologie lors de sa construction, de sa migration ou encore de sa fusion avec d'autres ontologies. Dans ce chapitre, nous proposons une nouvelle méthodologie basée sur l'approche ScaleSem permettant de prévoir et d'identifier les contradictions lors de l'évolution des ontologies. Cette approche est basée sur la logique temporelle et les patrons de conception d'ontologies. Nous mettons en œuvre l'approche proposée en utilisant le model checker NuSMV. Sur la base de ces modèles, nous proposons un processus automatisé pour guider et surveiller la mise en œuvre du changement tout en veillant à la cohérence de l'ontologie à faire évoluer.

Plan

1 L'évolution des ontologies 167
1.1 La méthodologie AIFB 168
1.2 La méthodologie IMSE 168
1.3 La méthodologie de Rogozan 169
1.4 La méthodologie de Djedidi 171
1.5 Les autres méthodologies 173
2 La détection des incohérences 175
3 La méthodologie CLOCk 178
3.1 OntoVersionGraph 179
3.2 Notre approche 180
4 Application de la méthode 184
5 Conclusion 187

Le web sémantique est un environnement dynamique, multi-acteurs et distribué. La conception d'ontologie est un travail lent et difficile. Néanmoins, comme tout modèle basé sur une conceptualisation du monde réel, l'ontologie doit s'adapter au changement et évoluer. Les raisons du changement sont multiples (Rogozan, 2008). Par exemple : le domaine de définition peut changer, l'ontologie peut être utilisée pour des tâches différentes que celles définies initialement... Les travaux sur l'évolution des ontologies, cruciaux pour le développement du web sémantique, sont récents. L'évolution d'une ontologie passe par plusieurs étapes dont l'étape de changement qui consiste à modifier l'ontologie pour prendre en compte les évolutions du domaine qu'elle modélise. L'adaptation de l'ontologie au changement peut faire apparaitre deux types de problèmes : l'incohérence et l'inconsistance. L'incohérence est détectée lorsqu'il existe une interprétation possible de cette ontologie qui soit un modèle, mais que ce modèle sous-jacent associe à certaines classes un ensemble d'instances qui sera toujours vide. C'est-à-dire qu'aucun objet ne pourra jamais être une instance de cette classe. Ce type de classes est appelé classe insastifiable. L'inconsistance est détectée lorsqu'il n'existe aucune interprétation possible de l'ontologie qui soit un modèle pour cette ontologie. Conduire l'application des changements tout en maintenant la cohérence de l'ontologie est une tâche cruciale et coûteuse en termes de temps et de complexité. Un processus automatisé est donc essentiel.

Dans ce chapitre, nous nous intéressons aux problèmes de la gestion des changements d'une ontologie. Pour répondre à ces problèmes, nous proposons une nouvelle méthodologie CLOCk (**C**hange **L**og **O**ntology **C**hec**k**er) basée sur une méthode formelle et le model checking. Cette méthode s'appuie sur une modélisation à l'aide des patrons d'inconsistances pour la détection et la correction de l'incohérence provoqués pendant le processus d'évolution d'ontologie.

1 L'EVOLUTION DES ONTOLOGIES

D'une façon générale, l'évolution d'ontologies est une notion complexe qui fait référence à des facteurs structurels, fonctionnels et également logiciels. Plusieurs définitions ont été proposées dans la littérature. Dans (Maedche et al., 2003) et (Stojanovic, 2004), les auteurs pensent l'évolution comme « la modification appropriée d'une ontologie et la propagation des changements dans les autres ontologies qui en dépendent. » En parallèle, les auteurs de (Klein, 2004) et (Noy, 2004) ont proposé une autre définition plus précise : « l'évolution d'une ontologie est la capacité de gérer les changements apportés lors de l'évolution en créant et en maintenant différentes versions d'une ontologie. Cette capacité consiste à identifier et à différencier les versions, à modifier les versions, à spécifier des relations qui rendent explicites les changements effectués entre les versions. » La deuxième définition suppose qu'une ontologie peut avoir plusieurs états (appelés versions) qui peuvent être explicitement différenciés.

Plusieurs recherches ont montré l'importance majeure de l'évolution d'ontologie ainsi que le manque presque total des approches pour gérer cette évolution. Dans la littérature, il existe deux méthodologies principales sur l'évolution d'ontologies : la méthodologie AIFB et la méthodologie IMSE. Dans cette section, nous allons les décrire brièvement ainsi que les autres méthodologies qui s'en inspirent.

1.1 LA METHODOLOGIE AIFB

La méthodologie AIFB (*Institute of Applied informatics and Formal Description Methods*) proposée par l'université de Karlsruhe (Allemagne) repose sur six étapes décrites par (Stojanovic et al., 2002).

- **Détection des changements**, cette étape se base sur les besoins explicites (ajout, suppression d'une entité ontologique) exprimés par les développeurs d'ontologies et sur l'application de méthodes heuristiques sur trois niveaux différents : la structure, les instances et les usages.
- **La représentation des changements**, elle vise l'édition des changements élémentaires, composites ou complexes de façon à les adapter au langage ontologique utilisé.
- L'étape de **sémantique des changements** qui a pour objectif de conserver la consistance de l'ontologie après son évolution en résolvant systématiquement des changements additionnels et en proposant à l'utilisateur des stratégies d'évolution.
- La quatrième étape est **l'implémentation des changements** qui consiste à notifier, à appliquer et à garder trace de tous les changements appliqués.
- L'avant-dernière étape est la **propagation des changements**, cette étape assure la consistance des objets dépendants après qu'une mise à jour de l'ontologie ait été effectuée. Ces objets peuvent inclure des ontologies et des applications fonctionnant avec l'ontologie.
- l'étape de **validation**, dans cette phase, les utilisateurs évaluent les résultats de l'évolution. Si la qualité de l'ontologie est préservée ou améliorée, alors les changements peuvent être validés ; sinon les annuler ou reprendre les phases d'évolution selon un processus cyclique.

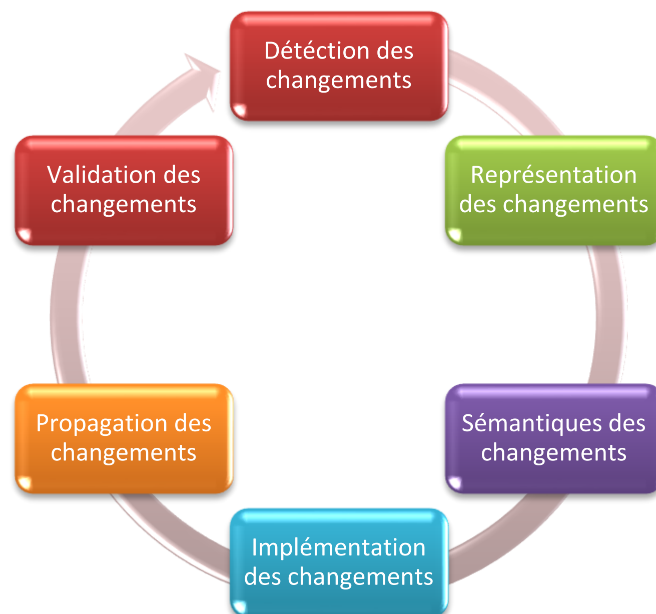


Figure 49. Les étapes de la méthodologie AIFB.

1.2 LA METHODOLOGIE IMSE

La méthodologie IMSE (*Information Management and Software Engineering*) de l'université d'Amsterdam (Klein, 2004) (Noy, 2004) (Noy et Musen, 2004), permet d'identifier et de différencier les versions, de modifier les versions, de spécifier des relations qui rendent explicites les

changements effectués entre les versions et d'utiliser des mécanismes d'accès pour les artefacts dépendants (*Artefacts dépendants* : il s'agit objets qui dépendent de l'ontologie à évoluer comme les ontologies, les applications, etc.). Ce processus se base sur deux modules fondamentaux pour une méthodologie de « *versionnage* » des ontologies :

- Le **module d'analyse de relation entre deux versions d'ontologies** permet les fonctionnalités suivantes : (i) mise en évidence des changements effectués dans la définition des entités ontologiques, (ii) spécification de la relation sémantique entre les entités ontologiques qui ont subi un changement dans leurs définitions, (iii) description des changements effectués par un ensemble de métadonnées qui décrivent l'auteur, la date et le but de chaque changement.
- Le **module d'identification des versions d'ontologies sur le Web** permet de rendre opérationnelle la nouvelle version de l'ontologie après évolution. Il se base sur deux principes : (i) Un changement dans la définition d'une entité ontologique produit une nouvelle version, ayant un nouvel URI (Uniform Resource Identifier) alors qu'un changement dans l'annotation textuelle d'une entité produit seulement un nouveau fichier avec un nouvel URL (Uniform Resource Locator), (ii) la forme d'URI indique si la version est compatible avec la version antérieure.

1.3 LA METHODOLOGIE DE ROGOZAN

Selon Rogozan (Rogozan, 2008), la méthodologie AIFB ne propose aucune étape d'analyse des effets des changements sur la relation entre l'ontologie évoluée et les artefacts dépendants. De plus, l'étape de propagation des changements est essentiellement unidirectionnelle, car elle vise uniquement la modification des ontologies dépendantes afin de préserver leur consistance structurelle avec l'ontologie de base et ne touche pas ainsi le référencement sémantique des ressources. Concernant la méthodologie IMSE, (Rogozan, 2008) considère que les auteurs de cette méthodologie proposent une approche pour supporter la gestion des versions d'une ontologie après son évolution et non pas pour supporter l'évolution proprement dite des ontologies. En effet, ils fournissent un modèle d'analyse de la relation entre les versions de l'ontologie, mais ne se préoccupent pas de la gestion de l'accès aux artefacts dépendants.

La méthodologie proposée par Rogozan s'inspire des deux méthodologies AIFB et IMSE. Cette méthodologie suit un processus composé de neuf étapes (voir figure 50). La première étape permet d'accéder à l'ontologie à modifier. Les six étapes qui suivent correspondent à celles de la méthodologie AIFB suivies par l'étape d'analyse des changements qui permet de fournir une analyse des effets des changements afin d'identifier ceux susceptibles de provoquer des problèmes de dysfonctionnement de systèmes, des inconsistances d'ontologies dépendantes ou encore des pertes d'accès aux ressources référencées sémantiquement. La dernière étape du processus consiste en la mise en opération de la version V_{N+1}, avec la prise en compte de la propagation des changements vers les artefacts dépendants. L'objectif de cette étape est de préserver les rôles de l'ontologie pour la nouvelle version V_{N+1} soit en utilisant seulement la nouvelle version, soit des versions multiples présentant des liens de correspondances. Le choix de l'une de situation se fait selon la possibilité des changements exécutés pour passer de V_N à V_{N+I}.

- **Accéder, identifier et télécharger la version de l'ontologie à modifier**. Dans cette première étape, les développeurs doivent accéder à l'ontologie à modifier et la télécharger sur leur poste de travail. L'ontologie récupéré par les développeurs est notée V_N.
- **Identifier les changements**. Les développeurs identifient les changements à apporter à l'ontologie d'une manière descendante, c'est-à-dire à partir des modifications dans le domaine de définition ou d'utilisation de l'ontologie, ou ascendante, c'est-à-dire à partir de l'analyse de l'ontologie elle-même.
- **Éditer les changements**. Dans cette étape, les développeurs éditent les changements identifiés précédemment. Pour permettre l'édition des changements, une taxonomie des changements élémentaire spécifiant l'ajout, l'effacement et la modification des entités ontologiques est définie. Par exemple, si un concept à l'intérieur de la hiérarchie des concepts est supprimé, ses sous-concepts peuvent être, soit supprimés, soit rattachés au concept-parent, soit rattachés au concept-racine de la hiérarchie. Le résultat de l'étape d'édition consiste alors dans une séquence des changements à apporter à l'ontologie, chaque changement ayant la forme d'un couple de type (changement édité, scénario de changements additionnels).
- **Vérifier les changements**. Cette étape vérifie les effets des changements sur la consistance de l'ontologie. Si des changements invalident les contraintes de l'ontologie, alors les développeurs doivent être informés et une méthode pour les supporter dans la résolution d'inconsistances doit leur être fournie. Pour les changements qui invalident les axiomes de l'ontologie, les développeurs peuvent aussi modifier directement les axiomes invalidés afin de contourner les inconsistances.
- **Approuver les changements**. Dans cette étape, la séquence des changements est validée conjointement par les développeurs de l'ontologie. Si cette validation n'est pas possible dû au caractère distribué de l'environnement d'occurrence du processus d'évolution, alors des mécanismes de gestion des conflits doivent être prévus.
- **Implémenter les changements**. Cette étape vise l'implémentation des changements avec une préservation de la trace et des métadonnées associées, par exemple dans un journal de changements, et avec une gestion de l'identification des versions, sachant que seulement les changements ontologiques produisent une nouvelle version de l'ontologie, que nous notons V_{N+1}, ayant un nouvel URI.
- **Identifier les changements et analyser la relation de compatibilité entre V_N et V_{N+1}**. Le processus d'évolution peut causer des incompatibilités pouvant provoquer l'altération des rôles de l'ontologie. Le but de cette étape est de (1) identifier les changements exécutés pour passer de V_N à V_{N+1}, (2) caractériser leurs effets sur la relation sémantique entre les entités ontologiques appartenant à V_N et celles appartenant à V_{N+1} et (3) analyser les effets des changements sur la compatibilité de V_{N+1} du point de vue de la préservation des rôles de l'ontologie.
- **Valider V_{N+1} dans la communauté**. Les développeurs approuvent ou désapprouvent collectivement la nouvelle version de l'ontologie avant de la rendre opérationnelle.
- **Opérationnaliser V_{N+1}**. Cette dernière étape consiste à préserver les rôles de l'ontologie pour la nouvelle version Vn+1 en fonction des résultats de l'analyse de la relation de compatibilité entre V_N et V_{N+1}.

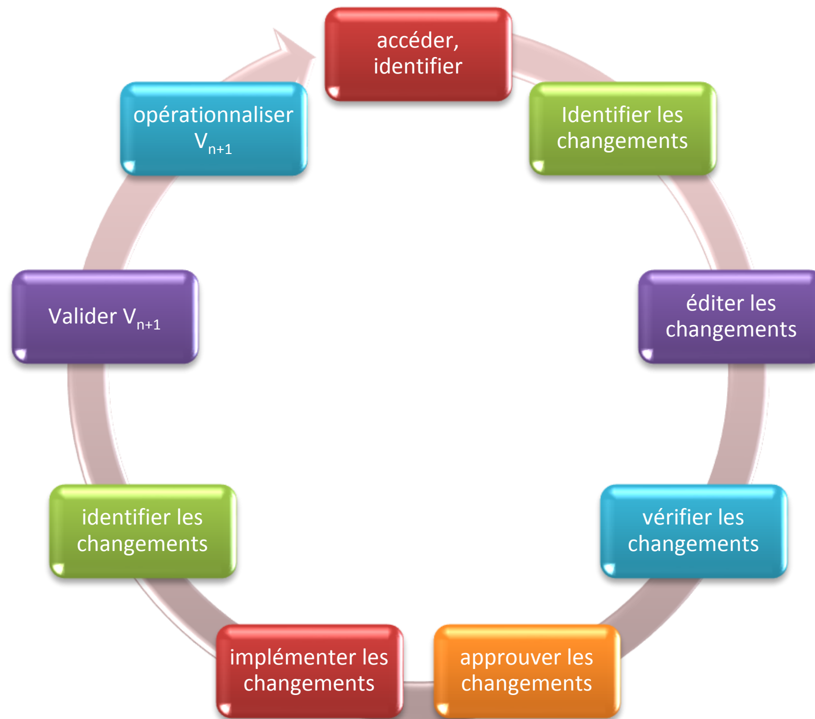


Figure 50. Les étapes de la méthodologie Rogozan.

1.4 LA METHODOLOGIE DE DJEDIDI

Rim Djedidi (Djedidi et Aufaure, 2009) définit une méthodologie de gestion de changements s'inspirant largement de la méthodologie AIFB, appelé **Onto-Evo^{a}l** (Ontology Evolution-Evaluation) qui s'appuie sur une modélisation à l'aide de patrons. Ces patrons spécifient des classes de changements, des classes d'incohérences et des classes d'alternatives de résolution. Sur la base de ces patrons et des liens entre eux, un processus automatisé permet de conduire l'application des changements tout en maintenant la cohérence de l'ontologie évoluée. La méthodologie intègre aussi une activité d'évaluation basée sur un modèle de qualité d'ontologies. Ce modèle est employé pour guider la résolution des incohérences en évaluant l'impact des alternatives de résolution proposées sur la qualité de l'ontologie et ainsi choisir celles qui préservent la qualité de l'ontologie évoluée.

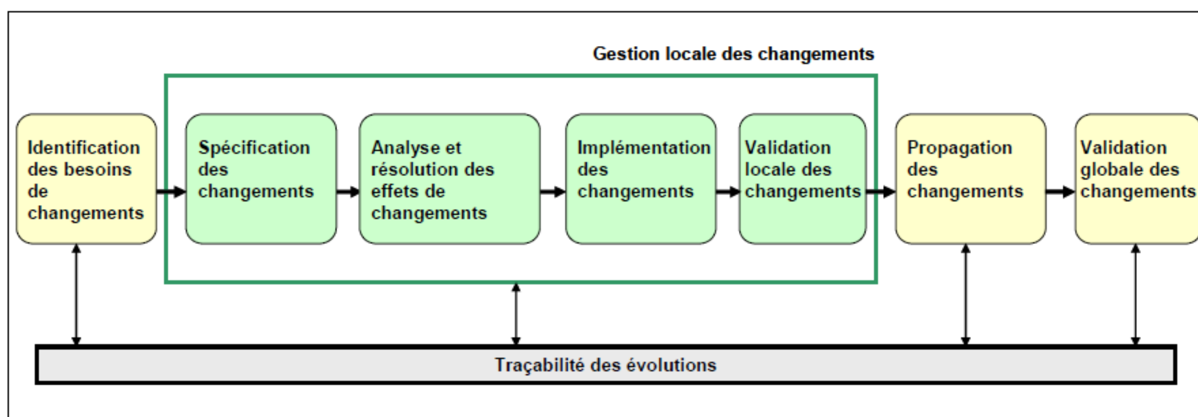


Figure 51. Méthodologie de gestion de changements Onto-Evo^{a}l.

La méthodologie **Onto-Evo^{a}l** est composée de quatre étapes (voir figure 51). L'étape d'identification des besoins de changement, la gestion des changements qui est une étape

importante et qui sera détaillée ci-dessous, l'étape de propagation des changements et la dernière étape, la validation globale des changements. Notons que tout au long du processus, les différents résultats sont sauvegardés dans le journal d'évolution qui est une structure permettant de conserver l'historique des évolutions de l'ontologie et les détails de traitement de ces évolutions sous forme de séquences chronologiques d'informations. Le but principal du journal d'évolution est la facilité du suivi de l'évolution, le retour arrière, la justification des changements, la gestion des versions.

Le processus de gestion de changements est conduit à travers quatre phases (figure 51): spécification du changement, analyse du changement, résolution du changement et application du changement.

- La **spécification du changement** est lancée suite à la demande d'un changement à appliquer sur une ontologie initiale supposée cohérente. L'utilisateur demande un changement élémentaire ou composé sans avoir à définir et ordonner les opérations de changements intermédiaires nécessaires à son application.
- **Analyse du changement**. Le changement formellement spécifié est appliqué à une version temporaire de l'ontologie pour analyser ses impacts.
- La **résolution du changement** comprend deux principales activités :
  - La **proposition de résolutions**. À partir des instances d'incohérences détectées, les patrons d'alternatives sont instanciés pour générer les alternatives potentielles de résolution. Chaque alternative représente des opérations de changements additionnels à appliquer pour résoudre une incohérence. Cependant, elle ne doit pas causer d'autres incohérences. Pour cela, toutes les alternatives sont vérifiées et résolues selon un mécanisme récursif et seules les alternatives cohérentes sont retenues.
  - **L'évaluation des résolutions** permet de guider le choix de l'alternative à appliquer en évaluant l'impact de chacune des alternatives sur la qualité de l'ontologie. L'évaluation se base sur un modèle de qualité considérant les aspects structure et usage de l'ontologie à travers un ensemble de critères et de métriques. L'alternative qui préserve la qualité de l'ontologie peut être automatiquement choisie. L'évaluation de l'impact sur la qualité participe à l'automatisation du processus en guidant le choix des résolutions à travers des alternatives annotées et évaluées.
- **L'application du changement** correspond à l'application finale du changement. Elle emploie des techniques d'évaluation permettant de guider la résolution des incohérences et de minimiser la dépendance à l'utilisateur. Ainsi, si les résolutions préservent la qualité, le changement requis et ses changements dérivés seront directement validés et appliqués et l'ontologie va évoluer. Par contre, si les alternatives ont un impact négatif sur la qualité, les résultats des différentes phases du processus seront présentés au responsable de la conception de l'ontologie en complément de son expertise pour qu'il décide du changement.

L'ensemble des traitements d'analyse et de maintenance de la cohérence est appliqué à une version temporaire de l'ontologie qui peut être abandonnée si les changements sont annulés pour

une restitution de l'ontologie initiale. Dans le cas contraire, la version modifiée de l'ontologie est définie.

La méthodologie **Onto-Evo^{a}l** se base sur une modélisation de patrons. Ces patrons de gestion de changements CMP (Change Management Patterns) sont proposés comme une solution permettant de ressortir des invariances observées répétitivement lors d'un processus d'évolution d'ontologies. De la modélisation des CMP en ressort trois catégories de patrons : des patrons de changements, des patrons d'incohérences et des patrons d'alternatives de résolution (Djedidi, 2009).

- Les **Patron de changement** sont définis sur la base du modèle OWL DL. L'idée est de catégoriser les changements, définir formellement leur signification, leur portée et leurs implications potentielles.
- **Patron d'incohérence** : Durant la phase d'analyse, le système vérifie les contraintes spécifiées dans la signature du changement, les liens entre le patron de changement instancié et les patrons des incohérences qu'il peut potentiellement causer. Durant ce processus, il tient compte de toutes les incohérences logiques concrètement détectées par le raisonneur Pellet. Toutes les incohérences constatées sont alors classées selon les patrons d'incohérences pour être résolues. Le module de classification se base sur l'interprétation des résultats de Pellet ainsi que les compositions d'axiomes et les dépendances entre axiomes pour identifier les axiomes causant les incohérences.
- **Patron de résolution** : Dans la phase de résolution du changement, la proposition de résolutions se base sur les liens entre le patron d'incohérence instancié et les patrons d'alternatives qui peuvent potentiellement le résoudre. Un patron d'alternative représente un changement additionnel à appliquer pour résoudre une incohérence logique. Il est décrit comme un changement (basique ou complexe) et hérite des propriétés d'un patron de changement, ce qui implique qu'il peut lui-même causer des incohérences. D'autres informations peuvent aussi décrire les patrons d'alternatives telles que les préconditions à satisfaire pour choisir le patron comme solution de résolution.

1.5 LES AUTRES METHODOLOGIES

D'autres auteurs proposent des éléments méthodologiques sur l'évolution des ontologies. (Heflin et Hendler, 2000) et (Heflin et al., 1999) développent SHOE, un langage fondé sur HTML, qui offre des primitives pour la gestion des versions multiples, en permettant d'associer à chaque version ontologique un identifiant unique ainsi qu'une balise « Backward-Compatible_With » spécifiant les versions compatibles ou rétrocompatibles. (Oliver et al., 1999) définissent un modèle conceptuel appelé CONCORDIA, pour la gestion des changements d'une terminologie médicale. Ce modèle associe à chaque concept un identifiant unique, les concepts pouvant ensuite être seulement retirés et non pas physiquement effacés. Ainsi, pour chaque concept, le modèle CONCORDIA est capable de garder la trace de tous ses concepts parents ou enfants retirés en utilisant leur identifiant. Enfin, (McGuinness, 2000) fournit des recommandations théoriques pour garantir un processus d'évolution de l'ontologie avec un minimum d'erreurs en utilisant des techniques de fusion pour mettre en évidence ces erreurs.

Du point de vue de l'évolution de l'ontologie dans un environnement multi-acteurs, (Pinto et Martins, 2002), et (Pinto et al., 2004) proposent un modèle pour gérer la négociation des

changements provenant des acteurs distants qui tentent de modifier une ontologie partagée. Dans ce modèle, chaque utilisateur modifie l'ontologie à partir de son poste individuel, en construisant ainsi sa propre ontologie locale. Un comité scientifique analyse ensuite les ontologies locales pour identifier les changements à introduire dans l'ontologie partagée. Une approche similaire est développée par (Sunagawa et al., 2003), la différence étant que les acteurs distants modifient l'ontologie partagée sans aucun intermédiaire. Ainsi, pour assurer la gestion de la dépendance entre les changements effectués par les différents acteurs, ceux-ci peuvent choisir d'accepter les changements des autres, de réduire la dépendance par rapport aux changements des autres ou de rompre la dépendance.

Rogozan (Rogozan, 2008) a proposé un outil « ontoanalyseur » fondé sur un modèle uniformisé de représentation des changements. Ce modèle est sémantiquement riche, car il est basé sur une ontologie des changements. Cet outil se focalise sur la propagation des changements vers les artefacts dépendants et plus précisément vers les référencements sémantiques des ressources de l'ontologie. Pour améliorer la propagation des changements vers le référencement sémantique des ressources (Luong et Dieng-Kuntz, 2007) ont développé l'outil CoSWEM (Corporation Semantic Web Evolution Management). Ce système permet à l'utilisateur, grâce à ses interfaces graphiques, de comparer les différences entre deux versions de l'ontologie.

L'approche de (Tho et al., 2008) propose une technique d'évolution d'ontologies qui peut enrichir une ontologie construite manuellement avec des connaissances supplémentaires découvertes d'une manière automatique. Les connaissances extraites sont formalisées en des entités liées formant une petite ontologie qui sera intégrée, grâce à des mesures de similarité floues, dans l'ontologie à évoluer.

L'approche proposée par (Chrisment et al., 2006) permet de mettre à jour une ontologie légère du domaine en analysant de nouveaux documents appartenant au domaine. La mise à jour de l'ontologie se réalise à partir de l'analyse d'un corpus et de la gestion de types abstraits (concepts de haut niveau d'abstraction). La détection de nouveaux termes est effectuée grâce à deux règles. La première permet d'extraire les termes généraux qui n'existent pas dans l'ontologie, la deuxième permet d'extraire des termes spécifiques du corpus. Les termes extraits sont ensuite intégrés dans l'ontologie de base moyennant deux règles. La première vise à intégrer les nouveaux termes dans la hiérarchie des concepts existants et la seconde permet de créer de nouvelles relations associatives (relations sémantiques) entre les concepts existants.

L'outil de (Trousse et al., 2008) se focalise précisément sur la détection des changements au niveau des usages. Il intègre des algorithmes de fouille de données qui permettent d'extraire des connaissances concernant le profil et les préférences des utilisateurs grâce à l'analyse de leur trace.

Le système Evolva proposé par (Zablith, 2009) s'appuie sur des heuristiques d'extraction de connaissances qui permettent d'établir, grâce à des bases de connaissances, des relations entre les connaissances extraites et les concepts de l'ontologie initiale.

OntoVersionGraph (Pittet et al., 2011) est un outil pour l'évolution et le versionning d'ontologies basé sur la méthodologie AIFB. Spécialement conçu pour le domaine de gestion des connaissances. Cet outil peut être utilisé avec n'importe quel type d'ontologie basé sur les logiques

de description et en particulier le formalisme OWL-DL. Nous avons choisi d'adapter notre approche a cette proposition pour le caractère universel de cette solution (indépendant du type d'ontologie) et ces relations avec la méthode AFIB.

2 LA DETECTION DES INCOHERENCES

Avant de présenter notre approche de détection d'incohérences dans le processus d'évolution d'ontologie, nous allons présenter les risques et les limites des approches existantes par rapport à la détection des incohérences. L'objectif de cette détection est de conduire de manière automatisée l'application d'un changement tout en assurant la cohérence de l'ontologie.

Les opérations de peuplement et d'enrichissement intègrent toutes les deux une activité de validation de la cohérence. Dans le processus de peuplement, la maintenance de la cohérence consiste à identifier et éliminer les informations redondantes et a vérifier que les instances ne causent pas de contradictions, en utilisant des services standards de raisonnement. Dans le processus d'enrichissement, la maintenance de la cohérence est limitée à la détection des incohérences liées aux modifications appliquées.

Deux approches de vérification de cohérence sont présentées dans (Stojanovic, 2004) pour les ontologies KAON (KArlsruhe ONtology):

- La **vérification a posteriori** consiste en une seule vérification et est exécutée pour tous les changements appliqués.
- La **vérification a priori** est faite avant l'application d'un changement.

À chaque changement est associé un ensemble de préconditions à satisfaire pour qu'il puisse être appliqué. Par ailleurs, avant toute application de changement, l'ontologie est supposée cohérente. La maintenance de la cohérence est aussi basée sur les principes de résolution présentés dans (Haase et Stojanovic, 2005). Lorsque les axiomes transformés en OWL mènent à une ontologie incohérente, l'incohérence est localisée et l'axiome ayant le degré de confiance le plus faible est identifié et supprimé pour la résoudre.

La vérification a posteriori est une approche très coûteuse. Elle est appliquée à l'ontologie dans son ensemble. De plus, elle ne permet pas de préciser lequel(s) de(s) changement(s) appliqué(s) a causé les incohérences c'est la seconde approche qui a été adoptée dans (Stojanovic, 2004).

Deux approches sont proposées pour une résolution automatique des incohérences (Stojanovic et al., 2002) (Stojanovic, 2004):

- **Approche procédurale** : la maintenance se base sur les contraintes du modèle de cohérence et sur un ensemble prédéfini de règles à appliquer pour les satisfaire.
- **Approche déclarative** : la maintenance se base sur un ensemble complet d'axiomes inférés, formalisant l'évolution.

Il a été démontré qu'aussi bien l'approche procédurale que l'approche déclarative permettent d'appliquer le même ensemble de changements et offrent les mêmes possibilités de contrôle et d'adaptation des résolutions. Leur comparaison ne peut que se baser sur des critères subjectifs tels que l'efficacité du système d'évolution qui les implémente (Stojanovic, 2004).

La cohérence n'est vérifiée que pour des changements d'ajout d'axiomes, ce choix étant basé sur la monotonicité logique d'OWL. L'objectif de la vérification est de localiser la sous-ontologie incohérente minimale soit un ensemble minimal d'axiomes contradictoires (Haase et Stojanovic, 2005).

Pour les incohérences structurelles (se réfère aux contraintes du langage de l'ontologie), une résolution à base de règles de réécriture est proposée afin de transformer les axiomes en expressions compatibles avec les contraintes du modèle OWL Lite (Haase et Stojanovic, 2005). Pour la résolution des incohérences logiques (se réfère à la sémantique formelle de l'ontologie et à sa satisfiabilité) des stratégies de résolution d'incohérences sont introduites, elles se basent sur les contraintes de OWL Lite (Haase et Stojanovic, 2005). Elle vérifie que l'ontologie est sémantiquement correcte et ne présente pas de contradictions logiques.

Dans (Plessers et De Troyer, 2006), les auteurs ont défini une approche et un algorithme de localisation des axiomes causant les incohérences et ont proposé un ensemble de règles que le responsable de l'ontologie peut appliquer pour résoudre les incohérences. Le modèle de cohérence se base sur les contraintes de OWL DL.

Dans (Luong et Dieng-Kuntz, 2007), la maintenance de la cohérence tient compte uniquement du niveau structurel. Par ailleurs, il n'y pas une réelle phase de maintenance de l'ontologie évoluée avec détection et résolution des incohérences. La proposition de résolution d'incohérences consiste en une bibliothèque de résolutions associant des stratégies de résolution aux différents changements modélisés par l'ontologie d'évolution. Elle s'inspire des stratégies de résolution proposées dans (Stojanovic L., 2004).

Dans (Flouris et al., 2007), la maintenance de la cohérence logique a été prise en compte. Elle est basée sur l'application des principes de changement de croyance. Dans (Qi et al., 2006), les auteurs proposent une approche de gestion d'incohérences OWL DL basée sur la révision des croyances en complément aux travaux de (Flouris et al., 2005).

(Castano., 2006) propose une résolution des incohérences structurelle et logique en éliminant les redondances et les incohérences structurelles dans le cas du peuplement de l'ontologie. Cette résolution a été appliquée sur l'approche BOEMIE (*Bootstrapping Ontology Evolution with Multimedia Information Extraction*).

Dans (Cimiano, 2007), la maintenance de la cohérence logique est résolue à partir des degrés de confidence. L'algorithme de transformation des axiomes en OWL tient compte des degrés de confiance les plus faibles pour éviter la génération d'incohérences logiques. (Haase et Stojanovic, 2005)

(Klein, 2004) (Maedche et al., 2003) (Noy, et al., 2006) ne proposent pas de résolution d'incohérence, mais proposent une vérification de compatibilité entre les différentes versions de l'ontologie.

(Djedidi, 2009), défini une méthodologie de gestion de changements **Onto-Evo^{a}l** (Ontology Evolution-Evaluation) qui s'appuie sur une modélisation à l'aide de patrons. Ce modèle est employé pour guider la résolution des incohérences en évaluant l'impact des alternatives de résolution

proposées sur la qualité de l'ontologie et ainsi choisir celles qui préservent la qualité de l'ontologie évoluée. Seule la cohérence logique est considérée, car la cohérence structurelle – se référant aux contraintes du langage et l'utilisation de ses constructeurs – est vérifiée automatiquement au début du processus. Pour la vérification de la cohérence logique, le raisonneur Pellet est employé en interfaçage avec le système de gestion de changements. Pellet supporte aussi bien le niveau terminologique TBox (classes et propriétés de l'ontologie) que le niveau assertionnel ABox (instances de l'ontologie) de OWL DL (Sirin et al., 2007). Cependant, certaines incohérences logiques, notamment celles se référant aux propriétés, ne sont pas détectées par Pellet et sont prises en charge par le système de gestion. Par ailleurs, Pellet ne permet pas de préciser les axiomes qui ont causé les incohérences ni comment résoudre les incohérences détectées. L'identification des axiomes causant les incohérences est basée sur les travaux de (Plessers et De Troyer, 2006).

La méthodologie proposée par (Rogozan, 2008) ignore la vérification de la qualité de l'ontologie après évolution et avant la mise en opération de la nouvelle version de l'ontologie. C'est une méthodologie qui peut détecter les changements effectués entre une version de base et une version évoluée de l'ontologie.

Le tableau 11 présente une synthèse des différentes propositions de vérification de la qualité de l'ontologie.

	Niveau	Vérification	Proposition de résolution	Résolution automatique
Approche d'apprentissage d'ontologie basée sur l'identification des besoins de changement (Cimiano, 2007) (Cimiano et Völker, 2005)	Logique	À partir des degrés de confidence	Plusieurs solutions triées par une fonction d'évaluation	Suppression des axiomes incertains
Approche BOEMIE (Castano, 2006) (Castano et al., 2007) (Petasis, 2007)	Structure et logique	Oui		En cas de peuplement : élimination des redondances et incohérences logiques
Approche de gestion de changement pour des ontologies distribuées (Klein, 2004)(Maedche et al., 2003) (Noy et al., 2006)		Compatibilité entre différentes versions	Dériver des changements additionnels	Résoudre des problèmes spécifiques
Processus global d'évolution d'ontologie KAON (Stojanovic, 2004)	- Structurel (KAON) - Structurel et logique (OWL Lite) (Haase et Stojanovic, 2005)	- A priori (KAON) - Localisation de la sous-ontologie Incohérente minimale après des opérations d'ajout (OWL Lite) (Haase et Stojanovic, 2005)	- stratégies de résolution proposant les axiomes à supprimer (niveau logique, OWL Lite)	-Approche déclarative et procédurale (KAON) - Règles de réécriture (niveau structurel OWL Lite)
Approche d'évolution d'ontologie basée sur les principes de croyances (Flouris, 2006) (Flouris & Plexousakis, 2006) (Flouris et al., 2006a)	Logique	Révision de la satisfaction des axiomes		Basée sur les principes de changement de croyance
Approche de détection de changement basée sur l'historique des versions (Plessers et De Troyer, 2005) (Plessers et al.,2007)	Logique	Algorithme de localisation	Proposition de règles pour diminuer les axiomes les plus restrictifs	
Approche de gestion de l'évolution d'un web sémantique d'entreprise (Luong et Dieng- Kuntz, 2007)	- Structure de l'ontologie - Cohérence des annotations	- Vérification (pour les annotations) basée sur le modèle de cohérence des annotations (exprimé en RDF) ou sur la comparaison de versions d'ontologie	Bibliothèque de stratégies de résolution par type de changements (pour les ontologies) - Sélection (basée sur des règles) des entités cohérentes dans les annotations (si sans trace de changements d'ontologie)	Approche procédurale de résolution des annotations (si trace de changements d'ontologie)
Approche Onto-Evo^{a}l (Djedidi, 2009)	Logique	Raisonneur Pellet. Identifications des axiomes causant les incohérences sont basées sur les travaux de (Plessers et De Troyer, 2006).	s'appuie sur une modélisation à l'aide de patrons.	

Tableau 11. Synthèse des approches pour la maintenance de la cohérence

3 LA METHODOLOGIE CLOCK

La plupart des approches et des outils que nous avons décrits reposent sur une méthodologie qui suit un processus d'évolution d'ontologies bien déterminé. Un tel processus doit être complété par une phase d'évaluation de la qualité de l'ontologie avant la mise en production de la nouvelle

ontologie. Cette étape sert à juger la qualité de l'ontologie du point de vue de l'utilisateur. Aucun des outils décrits dans la section précédente ne présente une méthode pour évaluer la qualité de la nouvelle ontologie au niveau des usages d'une façon générale avec n'importe quelle application interagissant avec l'ontologie en question. On remarque aussi que presque tous les outils qui utilisent des techniques pour une détection automatique des changements effectuent seulement l'enrichissement de l'ontologie en évolution et ignorent par conséquent les opérations de modification et de suppression des éléments existants. De plus, les propagations des changements vers les artefacts dépendants (à savoir les ontologies, les référencements sémantiques, les applications) sont rarement abordées et mal traitées.

Cette section est divisée en deux parties. La première partie détaille l'approche OntoVersionGraph d'évolution des ontologies proposée par (Pittet et al., 2010). La seconde partie présente notre méthodologie **CLOCk** et son rôle dans le processus d'évolution OntoVersionGraph. Cette partie présente aussi les futures extensions de **CLOCk** pour la détection d'incohérences.

3.1 ONTOVERSIONGRAPH

L'approche OntoVersionGraph a été proposée par (Pittet et al., 2010) pour la gestion de l'évolution des ontologies. Elle regroupe deux ontologies plus petites et connectées qui sont l'ontologie de domaine et l'ontologie de versioning (ontologie contenant un ensemble de concepts et de propriétés permettant de représenter les évolutions de l'ontologie du domaine). Le fonctionnement de OntoVersionGraph est décrit dans la figure 52. L'idée générale est de créer une nouvelle ontologie à partir d'une ontologie de base en passant par plusieurs versions à l'aide d'opérateurs (ces opérateurs peuvent être des ajouts ou des suppressions) avec la possibilité de revenir à une version antérieure. Le versioning d'ontologies est la création de plusieurs versions successives de l'ontologie initiale. Il permet de gérer ces différentes versions et de détecter les incompatibilités (entre versions, les instances et les applications liées) qui peuvent apparaitre dans le processus d'évolution.

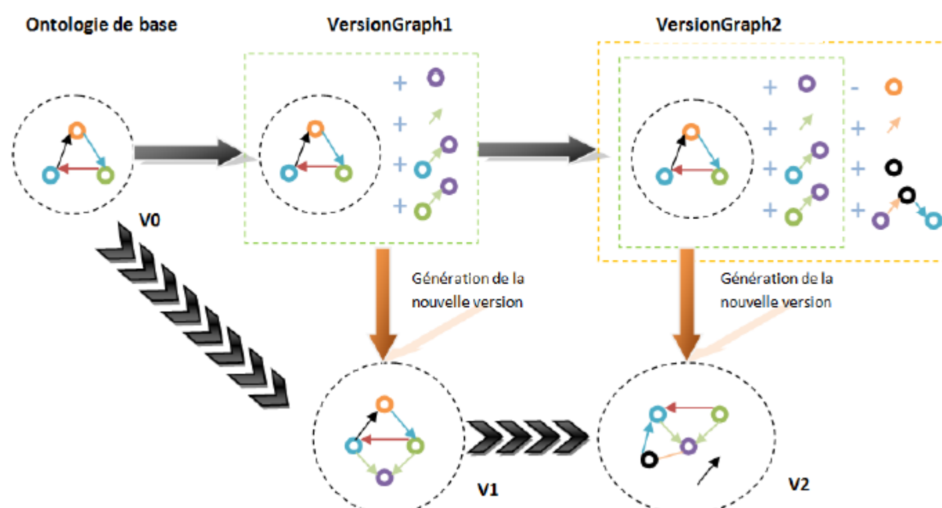


Figure 52. Le fonctionnement de OntoVersionGraph.

Le processus est décrit de la manière suivante : l'utilisateur effectue de nouveaux changements dans l'ontologie de base V_i. Les versions sont stockées dans un nouveau fichier VersionGraph. Les changements peuvent être une suite d'opérations simples et/ou complexes que

l'utilisateur souhaite appliquer sur le schéma ou les données d'une ontologie. Les opérations simples (ou élémentaires ou atomiques) représentent des opérations comme des ajouts ou des suppressions de relations ou de concepts. Les changements complexes (ou composés) sont une exécution de plusieurs opérations simples à la suite. Lorsque l'utilisateur valide ces changements, une nouvelle version V_{i+1} est créée. Cette nouvelle version devient l'ontologie de base pour des modifications futures.

L'ontologie de versioning est fixée dès la création de l'ontologie de domaine. Cette ontologie contient un ensemble de concepts et de propriétés permettant de représenter les évolutions de l'ontologie du domaine. Les principaux concepts de cette proposition son `VersionGraph` et `VersionContext`.

Le concept `VersionGraph` permet de représenter les évolutions de l'ontologie. Chaque `VersionGraph` peut être en relation avec une instance de la classe `VersionContext` qui permet de donner des informations sur la version telle que l'auteur, le titre ou encore la date de création. Chaque `VersionGraph` est également en relation avec une version précédente de l'ontologie.

L'approche OntoVersionGraph est basée sur la méthodologie AIFB qui comporte 6 étapes vu dans la section 1.1 de ce chapitre. Elle consiste à générer une nouvelle version de l'ontologie de base à l'aide d'une opération de validation (COMMIT).

3.2 NOTRE APPROCHE

Cette section présente l'adaptation de nos travaux dans le processus d'évolution OntoVersionGraph. Notre objectif est d'améliorer la phase d'identification des inconsistances d'ontologie dans le processus d'évolution d'ontologies. Plus précisément, nous souhaitons prédire et l'identifier des patrons de succession de changements incompatibles dans le journal ou le log d'évolutions. Plusieurs études ont montré l'importance de l'évolution d'ontologie sans prendre réellement en compte la gestion du changement.

La modélisation par patron propose des guides de bonnes pratiques et fournit des catalogues de composants ontologiques réutilisables. La notion de patrons de conception d'ontologies a été introduite par (Gangemi et al., 2004), (Rector et Roger, 2004), (Svatek, 2004). D'un point de vue théorique, les CMP (*Change Management Patterns*) se rapprochent des ODP (*Ontology Design Patterns*), plus particulièrement des patrons logiques LOP (*Logical Ontology Patterns*). Tous les deux sont appliqués dans le cycle d'ingénierie ontologique. Tout comme les CMP, les LOP sont indépendants du domaine modélisé par l'ontologie et peuvent être appliqués plus d'une fois dans une ontologie pour résoudre un problème d'évolution (CMP) ou de conception (LOP). L'implémentation des CMP et les expressions formelles des LOP sont toutes les deux issues du langage OWL-DL. Néanmoins, ils sont assez différents : les CMP proposent des solutions réutilisables pour guider la gestion des changements dans un processus d'évolution locale d'une ontologie. Les ODP sont conçus comme des guides de bonnes pratiques réutilisables dans un contexte de conception collaborative d'ontologies en réseau. Les LOP représentent des compositions de constructions logiques pour résoudre des problèmes d'expressivité (Presutti et al., 2008). Enfin, nous avons développé un processus complémentaire d'identification d'incohérences présentées dans la figure 53.

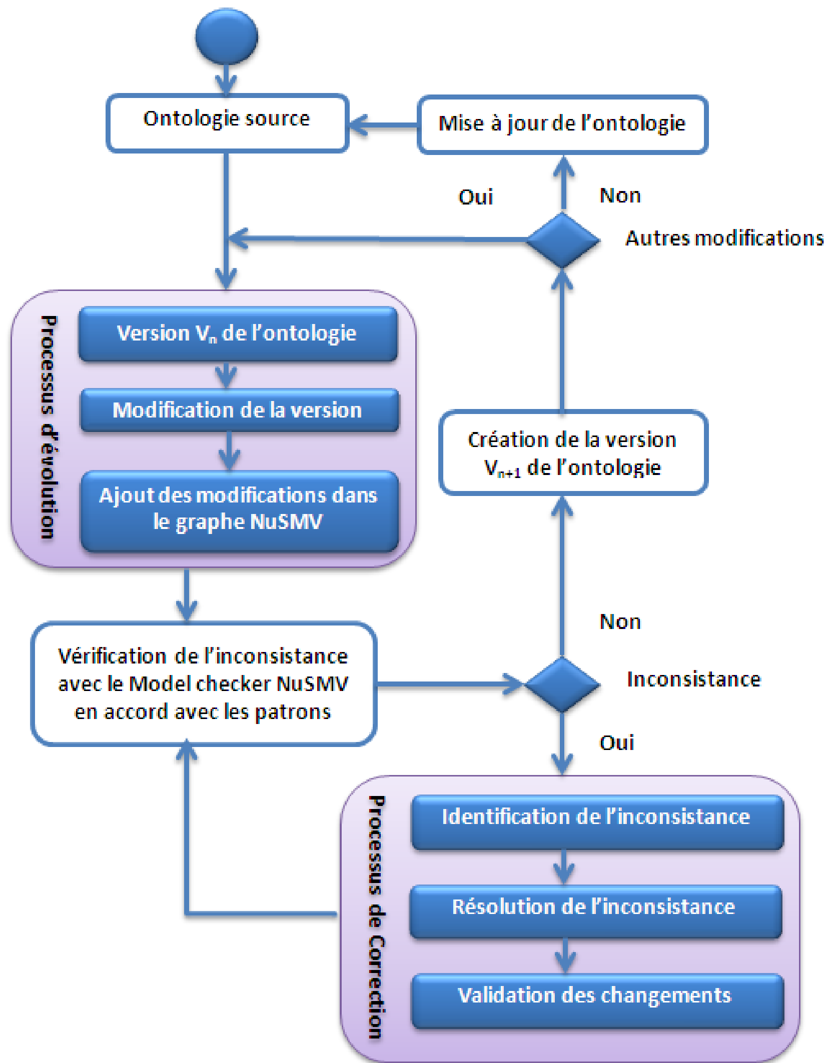


Figure 53.Le processus d'identification d'incohérences.

La figure 53 décrit le processus d'évolution, d'identification et de correction d'incohérence d'ontologie. La première étape (processus d'évolution) est la modification de l'ontologie de base V_n en ajoutant ou en supprimant des propriétés ou des concepts. L'étape suivante (processus d'identification) gère la cohérence de l'ontologie en utilisant le model checker NuSMV. Si une incohérence est détectée, le model checker retournera un contre-exemple affichant la séquence des modifications dans le graphe NuSMV provoquant l'incohérence. Sinon le model checker retournera « vrai » et le processus passera à la dernière étape : la validation. La troisième étape (processus de correction) est utile lorsque le model checker détecte une incohérence, cette étape corrige l'incohérence avant la mise en œuvre de la nouvelle version de l'ontologie et repassera par l'étape deux du processus pour s'assurer de l'inexistence de l'incohérence. Si après vérification, l'incohérence a disparu, on passe à l'étape de validation qui est la dernière étape de ce processus afin de valider la nouvelle version V_{n+1} de l'ontologie. Sinon le processus boucle sur l'étape trois pour corriger l'incohérence jusqu'à ce que l'incohérence disparaisse.

La taille et la complexité des ontologies rendent impossible une gestion manuelle des effets des changements. Pour la détection des inconsistances, nous avons besoin d'un mécanisme de

détection automatique. Les travaux de (Luong, 2004) proposent deux approches pour l'évolution de l'annotation sémantique :

- **Approche procédurale** : cette approche est appliquée dans le cas où il existe un journal de trace qui capture des changements ontologiques effectués.
- **Approche basée sur des règles** : cette approche est dédiée aux cas où l'on ne garde plus la trace de changement entre les versions de l'ontologie.

Dans notre approche de détection des inconsistances lors de l'évolution de l'ontologie on utilisera l'approche procédurale. Cette approche garde la trace de changement entre deux versions d'ontologie qui permet de vérifier les modifications effectuées pendant l'évolution de l'ontologie. Les modifications effectuées constituent une suite d'opérations simple et/ou composite que l'utilisateur souhaite appliquer sur le schéma (voir les parties « processus d'évolution » et « processus de correction » de la figure 53). Les opérateurs simples représentent des opérations comme des ajouts ou des suppressions de relation ou de concepts. Les opérateurs complexes sont une suite d'opérations simples.

En se basant sur les techniques du model checking et des formules de la logique temporelle, nous détecterons les inconsistances dans l'évolution d'ontologie provoquées par les opérateurs simples et/ou complexes en vérifiant juste les changements (des logs) effectués pendant le processus d'évolutions de l'ontologie de base. La méthodologie CLOCk intervient dans l'étape « sémantique des changements » du processus d'évolution. L'ontologie doit évoluer d'un état consistant vers un autre état consistant, c'est-à-dire l'état où les contraintes du modèle ontologique sont respectées. Afin de résoudre les inconsistances introduites par les changements, d'autres changements additionnels peuvent être nécessaires, la tâche de cette étape étant alors de permettre la résolution de tous les changements additionnels d'une manière systématique.

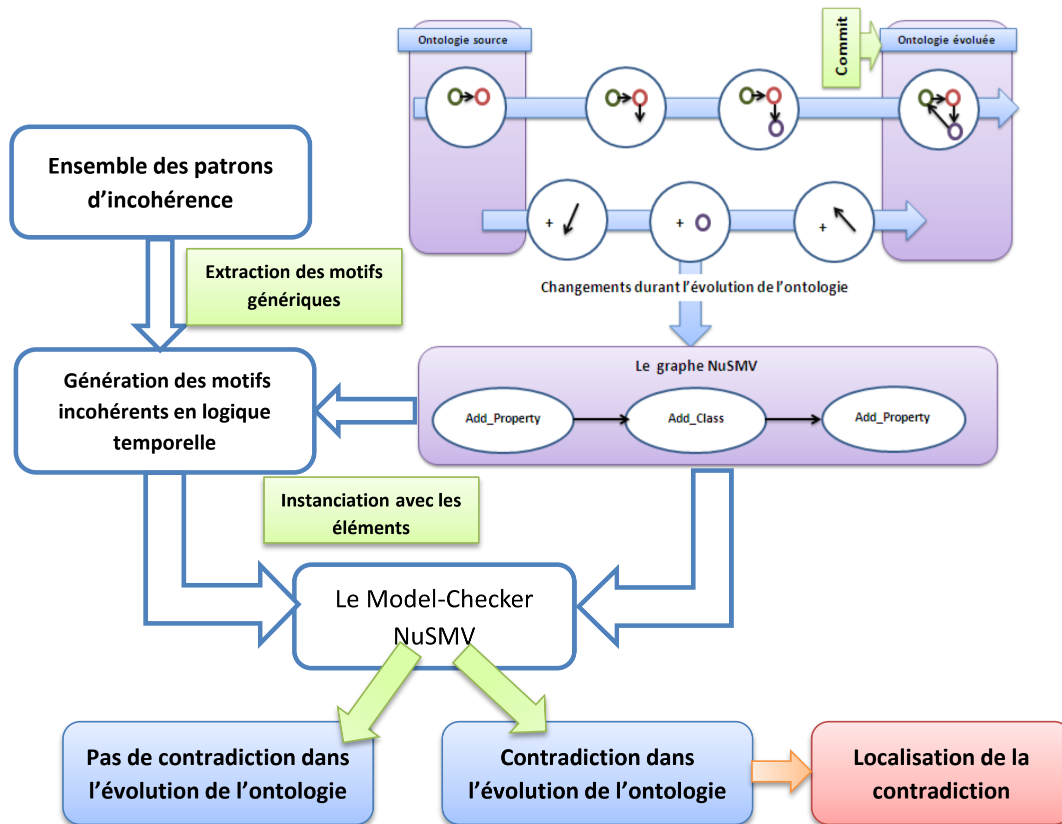


Figure 54. L'architecture de la méthodologie CLOCk.

La figure 54 présente les différentes étapes que nous proposons pour identifier l'incohérence dans l'évolution d'ontologie. Ce processus d'identification s'articule en trois phases. La première phase consiste à transformer le log évolution (un fichier pour la traçabilité des évolutions), spécifié en OWL DL, en un graphe NuSMV permettant au model checker de le parcourir (Gueffaz et al., 2011a). Le graphe NuSMV est composé de nœuds représentant les axiomes tracés dans le log et les arcs définissant les relations de succession entre eux. Les modifications sont représentées chronologiquement dans le graphe NuSMV.

La deuxième phase génère des motifs incohérents potentiels. Tout d'abord, ces motifs sont extraits des patrons d'incohérences correspondants aux constructeurs utilisés dans les axiomes. Puis, ils sont spécifiés en logique temporelle. Ensuite, un algorithme les instancie avec les éléments du graphe NuSMV (concepts, propriétés, etc.) afin de générer tous les motifs incohérents pouvant figurer dans la chronologie des axiomes. Par exemple, le patron d'incohérence correspondant au constructeur « AddDisjointClass ». Ce constructeur prend en entrée deux paramètres classes « Class1 » et « Class2 » qui sont des classes disjointes. Ces patrons d'incohérences définissent des règles d'instanciation spécifiées par l'attribut « hasInstanciationRule » pour chaque paramètre. Ce sont ces règles qui vont permettre de guider l'instanciation des motifs avec les éléments de la chronologie d'axiomes.

Enfin, la troisième phase utilise le model checker NuSMV pour vérifier si un des motifs définis précédemment en logique temporelle peut être localisé dans le graphe NuSMV. Pour cela les motifs générés en logique temporelle lui sont fournis en entrée de même que le graphe NuSMV. Le model checker parcourt le graphe chronologiquement de nœud en nœud afin de trouver une succession de

nœuds correspondant à l'un des motifs incohérents. Aussi, les nœuds ne doivent pas nécessairement être directement voisins successifs pour correspondre à un motif, ils ont juste besoin d'apparaître chronologiquement dans le même ordre.

Notre stratégie de vérification est automatique et se base sur la proposition décrite précédemment, qui consiste à utiliser des patrons de changements pour la vérification d'incohérences (Figure 54). La méthodologie CLOCk peut être appliquée sur l'ontologie pour une vérification a posteriori ou a priori en assurant la cohérence logique et structurelle de l'ontologie.

Une des tâches des raisonneurs comme Fact++ ou Pellet est de valider la cohérence du modèle associé à l'ontologie. Notre méthodologie a été testée avec différents raisonneurs. RacerPro et Pellet ne permettent pas d'identifier les axiomes impliqués dans l'inconsistance soulevée. Ils donnent simplement un indice afin d'aider l'utilisateur à déboguer l'ontologie. Cependant cette explication est trop vague pour que l'utilisateur comprenne entièrement la raison de l'inconsistance. Hermit et Fact++ identifient l'incohérence comme un problème au niveau des individus (« Bad Individuals »), ce qui signifie que la contradiction se trouve dans les instances de l'ontologie. Ils donnent une liste d'axiomes liés à l'incohérence. Cependant, comme il n'y a pas de log d'évolution permettant de récupérer la chronologie des axiomes, les axiomes ne peuvent pas être énumérés dans un ordre chronologique. Il n'est donc pas possible d'identifier lequel d'entre eux a causé la contradiction. La méthodologie CLOCk identifie l'incohérence provoquée lors de l'évolution de l'ontologie grâce au model checker et grâce au log d'évolution traçant la chronologie des axiomes.

4 APPLICATION DE LA METHODE

Cette section présente un exemple simple d'une ontologie inconsistante. La première partie de cette section décrit l'exemple d'ontologie incompatible et présente une comparaison des résultats des différents raisonneurs. La deuxième partie montre l'identification des successions incompatibles d'axiomes en utilisant notre approche CLOCk.

Nous considérons une ontologie (cohérente) O = (T, A), où T est la Tbox et A est la ABox. Dans les logiques de description, le domaine d'intérêt est représenté par l'intermédiaire d'une base de connaissances, où une séparation est faite entre les connaissances intensionnelles générales sur les concepts et les rôles, stockées dans un TBox (pour "Boîte terminologique"), et les connaissances spécifiques sur chacun des objets dans le domaine modélisé, stocké dans une ABox (pour "Boîte assertionnel").

Cette ontologie est composée des concepts *étudiant* et *professeur* définis comme suit:

T :

$$Etudiant \sqsubseteq \neg Professeur$$
$$Professeur \equiv \exists enseigne.Cours$$
$$Etudiant \equiv \exists assiste.Cours$$

Cette TBox définit *étudiant* et *professeur* en tant que deux classes disjointes. L'équivalence (≡) entre professeur et ∃enseigne.Cours signifie que l'ensemble des individus de professeur correspond à l'ensemble des individus définis par la quantification existentielle. Cela signifie qu'un professeur

doit enseigner un cours. La même chose pour la quantification existentielle ∃assiste.Cours. Un étudiant doit assister à un cours. La disjonction (⊑¬) entre *étudiant* et *professeur* implique sémantiquement qu'il est impossible pour un individu d'enseigner et d'assister à un cours s'il est membre de l'ensemble des individus *étudiant* ou *professeur*.

À :

Professeur(Luke)
Cours(Maths)

La ABox A insère deux individus *Luke* de type professeur et *Maths* de type Cours. Ajoutons un nouveau concept nommé *Doctorant* à la TBox T définie ci-dessus, que nous souhaitons être un *étudiant* (Doctorant ⊑ étudiants). Dans la ABox A, nous ajoutons aussi un autre individu appelé *Paul* de type *Doctorant* et l'assertion *enseigne(Paul,Maths)* qui signifie Paul enseigne un cours de Maths. Cet exemple montre vite que le système permet l'ajout de ce type d'assertion, mais que cette assertion provoque une incohérence dans l'ontologie. Ci-dessous, nous trouvons les différentes réponses des raisonneurs sur cet exemple.

Reasoner type	**Inconsistency Identification**	
	Involved Axioms	**Explanation**
RacerPro	∅	Individual Paul is forced to belong to class Student and its complement
Pellet	∅	Individual Paul is forced to belong to class Student and its complement

Hermit 1.3.6	`Bad Individuals:`
	
	`Paul Type PhDStudent,`
	`Paul teaches Maths,`
	`PhDStudent SubClassOf Student,`
	`Professor DisjointWith Student,`
	`Teaches Domain Professor`
		∅

Fact++	`Bad Individuals:`
	
	`Paul Type PhDStudent,`
	`Paul teaches Maths,`
	`PhDStudent SubClassOf Student,`
	`Professor DisjointWith Student,`
	`Teaches Domain Professor`
		∅

Figure 55. Résultats des différents raisonneurs.

RacerPro et Pellet n'arrivent pas à identifier les axiomes impliquant la cohérence soulevée. Néanmoins, Ils fournissent des indications afin d'aider l'utilisateur à déboguer l'ontologie. Cependant cette explication est incomplète pour arriver à comprendre la raison de l'incompatibilité (l'incohérence). Si Paul est obligé d'appartenir à la classe étudiant, on peut s'attendre à une erreur d'héritage entre Paul et une classe différente de Student, mais c'est tout ce que nous savons.

Hermit and Fact++ identifient l'incohérence comme un type «Bad individuals », ce qui signifie que la contradiction se trouve dans la ABox. Ils donnent chacun une liste de cinq axiomes liés à l'incohérence. L'absence de log d'évolution permettant de récupérer la chronologie d'évolution, provoque l'incapacité d'énumérer les axiomes dans un ordre chronologique. Il est donc impossible d'identifier les changements causant la contradiction. Par la suite, l'axiome de disjonction entre étudiant et professeur et l'axiome d'héritage entre *étudiant* et *doctorant* figurent parmi ces axiomes,

mais en fait, leur présence ne cause pas la contradiction. Les trois autres axiomes sont bien impliqués et ils peuvent expliquer de manière exhaustive la contradiction. En effet la cohérence soulevée n'est pas un problème de disjonction ou d'héritage, mais simplement un non-respect de la restriction de domaine de la propriété "enseigne".

Dans notre approche CLOCk, la première étape est l'extraction des axiomes du log d'évolution et leur traduction dans un graphe NuSMV. Ces axiomes sont présentés en utilisant une syntaxe non formelle abstraite:

```
A1::='AddBasicClass('Professeur')'
A2::='AddInstance('Luke Professeur')'
A3::='AddBasicClass('Etudiant')'
A4::='AddBasicClass('Cours')'
A5::='AddInstance('Maths Cours')'
A6::='AddDisjointClass('Professeur Etudiant')'
A7::='AddObjectProperty('enseigne')'
A8::='AddObjectProperty('assiste')'
A9::='AddPropertyDomain ('enseigne Professeur')'
A10::='AddPropertyDomain ('assiste Etudiant')'
A11::='AddObjectPropertyRange ('enseigne Cours')'
A12::='AddObjectPropertyRange ('assiste Cours')'
```

L'ajout du concept *Doctorant*, son lien d'héritage avec le concept *étudiant*, l'ajout de l'individu *Paul* et son assertion donne l'addition des axiomes suivants à la chronologie d'axiomes (également ajoutés au graphe NuSMV):

```
A13::='AddBasicClass('Doctorant')'
A14::='AddInstance('Paul Doctorant')'
A15::='AddClassInheritance('Doctorant Etdiant')'
A16::='AddInstanceObjectProperty ('Paul enseigne Maths')
```

La deuxième étape consiste à extraire les modèles génériques d'incohérence et leur instanciation avec les éléments de l'ontologie. Tout d'abord, les modèles génériques d'incohérences sont extraits des motifs d'incohérence correspondant au type de changement de chaque axiome. Dans la chronologie des axiomes, nous trouvons huit différents types de changement ("AddBasicClass", "AddInstance", "AddDisjointClass", "AddObjectProperty", "AddPropertyDomain", "AddObjectPropertyRange", "AddClassInheritance" et "AddInstanceObjectProperty"). Deuxièmement, les règles d'instanciation sont également extraites à partir des modèles d'incohérence. L'algorithme d'horloge applique ces règles à instancier et génère tous les modèles possibles incohérents, susceptibles de figurer dans la chronologie d'axiomes. Ces modèles sont spécifiés en formules de logique temporelle comme ci-dessous :

```
LTLSPEC F (state=AddDisjointClass(Professeur Etudiant) | state=AddEquivalentClass(Professeur Etudiant))
LTLSPEC F (state=AddDisjointClass(Etudiant Professeur) | state=AddEquivalentClass(Etudiant Professeur))
LTLSPEC F (state=AddDisjointClass(Professeur Doctorant) | state=AddEquivalentClass(Professeur Doctorant))
...
```

Script 26. Formules de logiques temporelles générées par l'algorithme.

Dans la troisième et dernière étape, les modèles sont donnés au model checker NuSMV qui vérifie leur existence dans le graphe NuSMV. Pour notre exemple, le model checker donne le résultat ci-dessous:

```
-- specification F (
state = AddPropertyDomain_enseigne_Professeur |
( state = AddInstance_Paul_Doctortant & F state = AddInstanceObjectProperty_Paul_enseigne_Maths)
) is true
```

Script 27. Résultat de NuSMV sur les formules du script 26.

La succession incohérente d'axiomes « AddPropertyDomain ('enseigne Professeur')' → 'AddInstance('Paul Doctorant')' → 'AddInstanceObjectProperty(Paul enseigne Maths) » a été trouvé dans le graphe NuSMV. Cette séquence contient trois axiomes; le dernier est celui qui provoque la contradiction et les deux autres axiomes illustrent les contraintes ne respectant pas cet axiome. Nous pouvons voir que l'assertion "Paul enseigne Maths" (dernier axiome) ne respecte pas la restriction de domaine de la propriété objet "enseigne" que l'individu «Paul» appartient à la classe "Doctorant" alors que le domaine de l'enseignement est pour les individus de type "professeur".

En outre, CLOCk affiche le résultat suivant pour aider l'utilisateur à comprendre l'incohérence:

<table>
<tr><td rowspan="4">CLOCk</td><td colspan="2">Inconsistency Identification</td></tr>
<tr><td>Involved Axioms</td><td>Explanation</td></tr>
<tr><td rowspan="2">ABox inconsistent succession of axioms:
AddPropertyDomain('enseigne Professeur')
→AddInstance('Paul Doctorant')
→AddInstanceObjectProperty('Paul enseigne Maths')</td><td>Axiom causing inconsistency:
AddInstanceObjectProperty(Paul enseigne Maths)</td></tr>
<tr><td>Unrespected Constraints:
AddPropertyDomain('enseigne Professeur')
AddInstance('Paul Doctorant')</td></tr>
</table>

Tableau 12. Résultats retournés par CLOCk montrant la source de l'incohérence.

5 CONCLUSION

Nous avons présenté dans ce chapitre une étude critique sur les différentes approches existantes supportant l'évolution des ontologies. De cette étude, nous avons pu dégager que l'étape sémantique des changements de l'ontologie après évolution possède certaines lacunes. Notre contribution consiste à répondre à ces lacunes en faisant évoluer l'ontologie et en maintenant sa cohérence et sa qualité tout au long de son processus d'évolution.

L'incohérence ontologique est étroitement liée à l'évolution d'ontologie. Dans ce chapitre, nous avons proposé une nouvelle méthodologie appelée **CLOCk** (Change Log Ontology Checker) basé sur l'approche ScaleSem pour identifier les incohérences dans l'évolution d'ontologie en utilisant le model checker NuSMV et en s'inspirant des patrons de conception d'ontologies. Nous avons montré les lacunes liées à la détection et l'identification de l'incohérence logique dans les solutions

existantes de gestion du changement. Nous avons présenté les avantages des techniques employées dans notre approche pour compléter la solution dans l'approche OntoVersionGraph.

Pour nos travaux futurs, nous visons à intégrer cette méthodologie comme la base d'une gestion complète du changement pour les ontologies OWL-DL accompagnées de log d'évolution. Une telle base est d'une importance capitale pour le déploiement d'applications basées sur des ontologies OWL-DL dans le contexte du Web sémantique. Bien que ce document ne traite que de l'identification d'incohérence, nous projetons de traiter la correction de l'incohérence d'ontologies, c'est-à-dire la résolution des contradictions qui provoquent l'insatisfiabilité d'une ontologie.

Plusieurs outils de débogage existent : OWLDebugger (Horridge et al., 2008), SWOOP (Kalyanpur et al., 2006), RepairTab (Lam et al., 2008). Leur but principal est d'isoler le plus petit ensemble d'axiomes menant à l'insatisfiabilité d'une classe. Les outils comme RepairTab (Lam et al., 2008) proposent aussi des solutions pour résoudre les contradictions identifiées avec une estimation de leurs conséquences sur les autres classes. (Roussey, 2009) décrit une stratégie pour déboguer les ontologies OWL-DL qui utilise un ensemble de services existants : raisonneurs, outils de détection d'anti-patrons, outils de résolution d'incohérences. Néanmoins les solutions sont toujours limitées soit à la suppression d'une partie des axiomes existants ou du remplacement d'une classe par une de ses superclasses, soit à l'amélioration de la détection des anti-patrons et l'ordonnancement des classes à corriger dans le débogage. De plus, les ontologies utilisées dans leurs expérimentations ont été écrites par des experts en logiques de description dans le but d'identifier des conflits particuliers alors que notre approche est applicable sur n'importe quelle ontologie.

Chapitre 8

Conclusions et Perspectives

Le W3C[20] a pour fonction principale de standardiser la représentation et l'échange d'informations sur le Web. Cet objectif devrait contribuer à rendre l'information compréhensible pour les procédés automatisés et les utilisateurs. De nouvelles normes ont été développées afin de permettre la représentation sémantique de l'information sous forme de langages dérivés du XML. Cette base est appelée web sémantique. Ce dernier est généralement représenté comme un empilement de langages allant des langages orientés des processus automatiques aux langages représentant des concepts plus abstraits de la sémantique formelle.

Plusieurs langages ont été développés dans le cadre du web sémantique et la plupart de ces langages sont basés sur le langage XML. Le langage OWL et le langage RDF sont des langages très importants du web sémantique. Le langage OWL permet de représenter les ontologies, et il propose aux machines une grande capacité d'exécution du contenu Web. Le RDF est le premier standard du W3C pour l'enrichissement des ressources sur le Web avec des descriptions détaillées. Ces langages sont utilisés pour représenter la sémantique associée à l'information, quelles que soient sa forme et sa structure sous la forme de graphes. Pour permettre la construction de graphes sémantiques, de nombreux outils ont été développés comme *Annotea*, qui est un projet du W3C qui spécifie l'infrastructure pour l'annotation des documents Web. Le principal format utilisé dans l'annotation est RDF et les types de documents peuvent être annotés sont des documents basés sur HTML ou XML. Cependant, aucune n'offre la possibilité de vérifier la cohérence de la sémantique et de réduire les erreurs d'annotations.

Le web sémantique, calqué sur le monde réel, est un environnement dynamique en constante évolution. Cette évolution requiert de répercuter les changements sur les ontologies du web sémantique. De par sa nature, une ontologie est en constante évolution. Tout changement peut mener à des contradictions au sein de l'ontologie. Il est donc indispensable de vérifier l'ontologie. Ces contradictions apparaissent pour plusieurs raisons telles que des erreurs de modélisation lors de la construction d'une ontologie ou lors de sa migration ou lors de fusion d'ontologies. L'évolution d'une ontologie est, de l'avis de la communauté, une démarche composée de plusieurs étapes. L'une d'elles, l'étape de changement, consiste à modifier l'ontologie pour la rendre plus précise et plus adéquate au domaine qu'elle modélise. Ce processus d'enrichissement, tout comme le processus de peuplement d'ontologie et une source et un révélateur d'incohérences. Conduire l'application des changements tout en maintenant la cohérence de l'ontologie est une tâche cruciale et coûteuse en termes de temps et de complexité. Un processus automatisé est donc essentiel.

L'objectif de cette thèse de doctorat est de considérer les graphes sémantiques comme des graphes d'état sur lesquels des techniques de model checking peuvent être appliqués. Notre ambition était de développer une nouvelle approche et des outils, tout d'abord, pour parcourir plus rapidement les graphes sémantiques, ensuite, pour proposer un système de requêtes plus simple dans son écriture, mais offrant une plus grande couverture fonctionnelle que les langages de requêtes existants, enfin, pour vérifier la cohérence des ontologies lors de leur évolution.

Pour répondre à ces attentes, nous avons développé un ensemble basé sur l'utilisation d'une méthode formelle : le model checking. Le model checking est une technique très puissante et très

[20] World Wide Web Consortium

précise dans la détection d'erreurs dans les systèmes complexes. Il peut révéler des erreurs qui n'ont pas été découvertes par les autres méthodes formelles telles que les essais et la simulation. Il peut aussi gérer des problèmes complexes avec de grandes quantités d'informations, stockées sous forme de graphe.

1 RAPPELS

Dans ce travail, nous avons tout d'abord développé des outils pour transformer les graphes sémantiques dans des modèles afin d'être vérifiés par le model checker. Les outils RDF2SPIN et RDF2NuSMV transforment les graphes sémantiques en langage PROMELA et en langage NuSMV respectivement. Nous utilisons le model checker SPIN et le model checker NuSMV pour vérifier les modèles des graphes sémantiques. SPIN est un outil logiciel pour la vérification de modèles de systèmes. Le système est décrit dans un modèle en langage appelé PROMELA. NuSMV est l'amélioration du model checker SMV, il travaille sur les mêmes principes que SMV. SPIN vérifie l'exactitude des propriétés exprimées en logique temporelle linéaire, alors que NuSMV vérifie les propriétés à la fois en logique temporelle linéaire et arborescente. Une étude sur le comportement de ces outils a démontré que l'outil RDF2SPIN a plusieurs limites par rapport à RDF2NuSMV. Le model checker SPIN n'utilise que la logique temporelle linéaire et les modèles des graphes sémantiques qui peuvent être traités ne doivent pas dépasser une taille limite de 255 nœuds (états). Suite à cette étude, la suite de nos travaux s'est basée sur le model checker NuSMV. Les graphes RDF traités dans nos travaux sont présentés sous forme de fichiers XML. Leur format n'est pas toujours compatible avec l'organisation attendue par le model checker NuSMV (absence de racine). Nous avons développé un algorithme de conversion qui s'articule en trois étapes : une étape d'exploration du graphe RDF, une étape pour la détermination d'un sommet racine et une étape pour la génération du modèle représentant le graphe RDF.

Une fois que nous avons obtenu un système de traitement des graphes sémantiques par le model checker NuSMV, nous avons travaillé sur la conception d'un langage de requêtes spécifique à notre système. Notre objectif de recherche dans cette partie a été de définir un langage de requêtes simple, puissant et expressif pour les graphes sémantiques. Nous avons utilisé comme base le langage de requêtes SPARQL. Nous avons étendu ce langage avec les opérateurs des propriétés en logique temporelle permettant de combler certaines limites de SPARQL dans les graphes sémantiques (l'expression des chemins entre deux états). Sur ce point notre approche est satisfaisante. Par contre, sur de nombreux points nos performances restent tributaires de la structure des graphes et des types de requêtes. Pour l'instant, nos performances ne sont pas satisfaisantes au regard de celles de SPARQL.

Le dernier point de notre travail concerne l'étude des problèmes de la gestion des changements d'une ontologie. Sur ce point, nous avons proposé une nouvelle méthodologie nommée CLOCk (*Change Log Ontology Checker*) basée sur le model checking. Elle s'appuie sur une modélisation à l'aide de patrons d'inconsistances pour la détection et la correction de l'incohérence provoqués pendant le processus d'évolution d'ontologie. Pour être plus précis, notre approche permet la prédiction et l'identification des patrons de succession de changement incompatibles dans le journal ou le log d'évolutions. Plusieurs études ont montré l'importance de l'évolution d'ontologie. Néanmoins, presque toutes ces approches ne gèrent les changements. Ce processus d'identification

s'articule en trois phases. La première phase consiste à transformer le log d'évolutions (un fichier pour la traçabilité des évolutions), spécifié en langage OWL-DL, en un graphe NuSMV. Le graphe NuSMV est composé de nœuds représentant les axiomes tracés dans le log et les arcs définissant les relations de succession entre eux. Les modifications sont représentées chronologiquement dans le modèle NuSMV. La deuxième phase génère des motifs d'incohérence. Tout d'abord, ces motifs sont extraits des patrons d'incohérences correspondants aux constructeurs utilisés dans les axiomes et sont spécifiés en logique temporelle. Ensuite, un algorithme les instancie avec les éléments du graphe NuSMV (concepts, propriétés, etc.) afin de générer tous les motifs d'incohérences pouvant figurer dans la chronologie des axiomes. Ces patrons d'incohérences définissent des règles d'instanciation spécifiées par l'attribut « *hasInstanciationRule* » pour chaque paramètre. Ce sont ces règles qui vont permettre de guider l'instanciation des motifs avec les éléments de la chronologie d'axiome. La troisième et dernière phase utilise le model checker NuSMV pour vérifier si un des motifs définis précédemment en logique temporelle peut être localisé dans le graphe NuSMV. Pour cela les motifs générés en logique temporelle lui sont fournis en entrée, ainsi que le graphe NuSMV. Le model checker parcourt le graphe chronologiquement de nœud en nœud afin de trouver une succession de nœuds correspondant à l'un des motifs incohérents. Les nœuds ne doivent pas nécessairement être directement voisins successifs pour correspondre à un motif, ils ont juste besoin d'apparaître chronologiquement dans le même ordre.

Notre stratégie de vérification est automatique et consiste à utiliser des patrons de changements pour la vérification d'incohérences. La méthodologie CLOCk peut être appliquée sur l'ontologie pour une vérification a posteriori ou a priori en assurant la cohérence logique et structurelle de l'ontologie.

2 PERSPECTIVES

L'utilisation d'un model checker pour la gestion de graphes sémantiques est une approche innovante qui n'a, à notre connaissance jamais été traitées par d'autres chercheurs. Pendant ces trois années, nous avons identifié plusieurs limites à cette approche.

En pratique la limitation majeure du model checking est la taille gigantesque des systèmes de transitions due au phénomène de l'explosion combinatoire du nombre d'états du système. Ce phénomène est dû, d'une part à la taille du système de transitions qui augmente exponentiellement avec le nombre de variables, et d'autre part au nombre de composants du système dans le cas où le système est concurrent. Pour limiter chacune de ces sources potentielles d'explosion combinatoire nous avons utilisé le model checker symbolique NuSMV. Le principe du model checking symbolique est d'utiliser une représentation symbolique de l'automate à vérifier. C'est-à-dire que les états sont manipulés par paquets au lieu d'être considérés un par un. Pour cela, l'algorithme utilise des fonctions booléennes sous la forme de BDD[21] en tant que représentation interne. Le model checking symbolique limite le nombre de variables et les ordres partiels pour les entrelacements de composants. Ceci augmentera les ressources de calcul permettant l'utilisation industrielle du model checking pour certains types d'application. Malgré cette utilisation de NuSMV, la taille des graphes sémantique est souvent une limite. Pour répondre à cette limite et gagner de la place en mémoire,

[21] Binary Decision Diagram

nous souhaitons d'une part réduire le nom des variables définies dans le modèle NuSMV et d'autre part développer un outil de fragmentation du graphe sémantique avant sa transformation pour traiter en parallèle les graphes sémantiques.

Dans notre travail, nous avons cherché à étendre le langage SPARQL. SPARQL est un langage de requête pour le Web sémantique qui a été standardisé par le W3C. Évaluer les requêtes SPARQL est connu pour être un problème de complexité NP-difficile (Pérez et al., 2009). Nous avons utilisé les algorithmes du model checking pour interroger les graphes sémantiques afin d'augmenter l'expressivité du langage de requête SPARQL. Pour cela, nous avons proposé un nouveau langage d'interrogation de graphes sémantiques basés sur l'approche ScaleSem. Ce langage utilise les opérateurs de la logique temporelle pour nous permettre un déplacement dans le graphe sémantique. Nous avons montré que les requêtes en logique temporelle sont plus simples et ont une plus grande expressivité en comparaison du langage de requête SPARQL. Néanmoins, le temps d'exécution de ces requêtes reste supérieur à celles équivalentes en SPARQL. La définition d'une syntaxe des requêtes en logique temporelle pour le web sémantique avec des temps de réponse acceptables est un challenge pour les prochaines années. Nous pourrons améliorer notre outil STL Resolver pour répondre à ces objectifs.

La méthodologie CLOCk proposée dans ce travail fonctionne sur de petits graphes. L'augmentation de la taille de l'ontologie provoque automatiquement l'augmentation du nombre d'instanciations des patrons d'incohérences. Ces patrons correspondent aux constructeurs utilisés dans les axiomes et sont spécifiés en logique temporelle. Nous avons décrit une stratégie pour détecter des incohérences au cours de son évolution. Notre stratégie améliore l'efficacité du processus d'évolution d'ontologie en proposant un ensemble d'actions prédéfinies à exécuter par le concepteur d'ontologie. Cette méthodologie est une mise en œuvre de notre approche ScaleSem dans le domaine de l'évolution d'ontologie. Notre objectif futur est de faire de cette méthodologie un débuggeur d'ontologie plus précis et plus rapide que les raisonneurs d'ontologie connu comme Pellet, Fact++.

ANNEXE 1

Cette annexe contient un petit exemple qui illustre la méthodologie CLOCk mise en œuvre dans l'outil ontoVersionGraph. Notre exemple se base sur l'ontologie présentée dans la Figure 56. Cette ontologie est composée de par quatre classes de base *Course*, *Professor*, *Student* et *PhD Student* et trois instances de classe *Maths*, *Luke* et *Paul*. Les classes *Professor* et *Student* sont deux classes disjointes c'est-à-dire qu'ils ne peuvent pas avoir d'instances en commun. La classe *PhD Student* hérite de la classe *Student*. Paul est un Doctorant (*PhD Student*), Luke est un professeur (*Professor*) et Maths est un cours (*Course*). La classe *Student* assiste à (*attends*) un cours (*Course*), la classe *Porfessor* enseigne (*teaches*) un cours (*Course*), et l'instance Paul enseigne (*teaches*) un cours de Math (Maths).

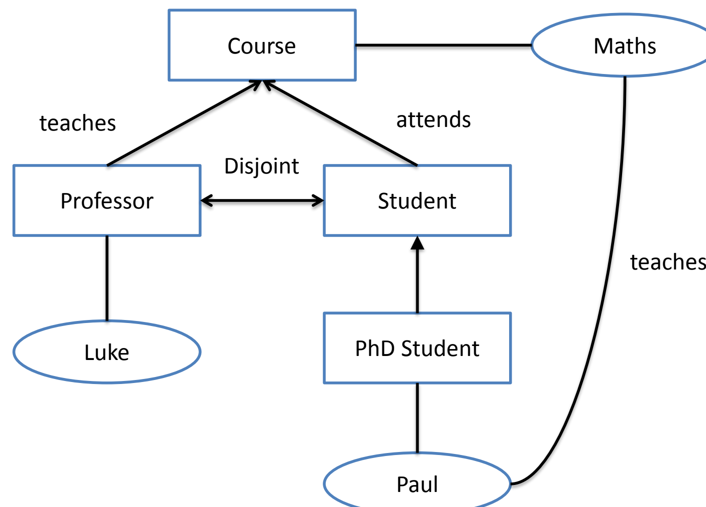


Figure 56. Exemple d'ontologie.

Apres évolution de cette ontologie en une ontologie décrite dans la figure 56, on obtient le log d'évolution global représenté en graphe NuSMV décrit dans le script 28.

```
MODULE main
	VAR
	state	:	{AddDisjointClass_Professor_Student,AddObjectProperty_teaches,	AddBasicClass_PhDStudent,
AddBasicClass_Student,AddObjectPropertyRange_teaches_Course,	AddInstance_Paul_PhDStudent,
AddClassInheritance_PhDStudent_Student,	AddInstanceObjectProperty_Paul_teaches_Maths,
AddInstance_Maths_Course,AddBasicClass_Course,	AddPropertyDomain_teaches_Professor,
AddPropertyDomain_attends_Student,	AddObjectPropertyRange_attends_Course,
AddInstance_Luke_Professor,AddBasicClass_Professor, AddObjectProperty_attends };
		msg : {nop, hasNext};
	INIT
		state =AddBasicClass_Professor;
	INIT
		msg = nop;
	ASSIGN
```

```
                    next(msg) :=
                              case
                                        state=AddObjectProperty_attends : { hasNext};
                                        state=AddObjectProperty_teaches : { hasNext};
                                        state=AddPropertyDomain_teaches_Professor : { hasNext};
                                        state=AddBasicClass_Student : { hasNext};
                                        state=AddInstance_Luke_Professor : { hasNext};
                                        state=AddBasicClass_Professor : { hasNext};
                                        state=AddObjectPropertyRange_teaches_Course : { hasNext};
                                        state=AddPropertyDomain_attends_Student : { hasNext};
                                        state=AddBasicClass_Course : { hasNext};
                                        state=AddObjectPropertyRange_attends_Course : { hasNext};
                                        state=AddInstance_Paul_PhDStudent : { hasNext};
                                        state=AddBasicClass_PhDStudent : { hasNext};
                                        state=AddInstance_Maths_Course : { hasNext};
                                        state=AddClassInheritance_PhDStudent_Student : { hasNext};
                                        state=AddDisjointClass_Professor_Student : { hasNext};
                                        TRUE:msg;
                              esac;
                    next(state) :=
                              case
          state=AddObjectProperty_attends & msg=hasNext : AddPropertyDomain_teaches_Professor;
          state=AddObjectProperty_teaches & msg=hasNext : AddObjectProperty_attends;
          state=AddPropertyDomain_teaches_Professor & msg=hasNext : AddPropertyDomain_attends_Student;
          state=AddBasicClass_Student & msg=hasNext : AddBasicClass_Course;
          state=AddInstance_Luke_Professor & msg=hasNext : AddInstance_Maths_Course;
          state=AddBasicClass_Professor & msg=hasNext : AddBasicClass_Student;
          state=AddObjectPropertyRange_teaches_Course & msg=hasNext : AddObjectPropertyRange_attends_Course;
          state=AddPropertyDomain_attends_Student & msg=hasNext : AddObjectPropertyRange_teaches_Course;
          state=AddBasicClass_Course & msg=hasNext : AddDisjointClass_Professor_Student;
          state=AddObjectPropertyRange_attends_Course & msg=hasNext : AddBasicClass_PhDStudent;
          state=AddInstance_Paul_PhDStudent & msg=hasNext : AddInstanceObjectProperty_Paul_teaches_Maths;
          state=AddBasicClass_PhDStudent & msg=hasNext : AddClassInheritance_PhDStudent_Student;
          state=AddInstance_Maths_Course & msg=hasNext : AddInstance_Paul_PhDStudent;
          state=AddClassInheritance_PhDStudent_Student & msg=hasNext : AddInstance_Luke_Professor;
          state=AddDisjointClass_Professor_Student & msg=hasNext : AddObjectProperty_teaches;
                                        TRUE:state;
                              esac;
```

Script 28. Le graphe NuSMV du log d'évolution.

Après extraction et instanciation des patrons de contraintes (Script 29) en formule de la logique temporelle, ils seront ajoutés automatiquement dans le fichier NuSMV représentant le log d'évolution global de l'ontologie.

```
LTLSPEC F (state=AddPropertyDomain_teaches_Professor & F state=AddInstance_Paul_PhDStudent & F state=AddInstanceObjectProperty_Paul_teaches_Maths)
LTLSPEC F (state=AddInstance_Paul_PhDStudent & F state=AddPropertyDomain_teaches_Professor & F state=AddInstanceObjectProperty_Paul_teaches_Maths)
LTLSPEC F (state=AddInstance_Paul_PhDStudent & F state=AddInstanceObjectProperty_Paul_teaches_Maths & F state=AddPropertyDomain_teaches_Professor)
```

Script 29. Les patrons d'incohérences en logique temporelle

En passant en paramètre le fichier NuSMV avec les propriétés en logique temporelle représentant les patrons d'incohérences, le model checker NuSMV retournera le résultat d'exécution suivant :

```
*** This is NuSMV 2.5.2 (compiled on Fri Oct 29 11:33:56 UTC 2010)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>
*** Copyright (c) 2010, Fondazione Bruno Kessler
*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado
*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification F ((state = AddPropertyDomain_teaches_Professor &  F state = AddInstance_Paul_PhDStudent) &  F state = AddInstanceObjectProperty_Paul_teaches_Maths)  is true
-- specification  F ((state = AddInstance_Paul_PhDStudent &  F state = AddPropertyDomain_teaches_Professor) & F state = AddInstanceObjectProperty_Paul_teaches_Maths)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = AddBasicClass_Professor
  msg = nop
-> State: 1.2 <-
  msg = hasNext
-> State: 1.3 <-
  state = AddBasicClass_Student
-> State: 1.4 <-
  state = AddBasicClass_Course
-> State: 1.5 <-
  state = AddDisjointClass_Professor_Student
-> State: 1.6 <-
  state = AddObjectProperty_teaches
-> State: 1.7 <-
  state = AddObjectProperty_attends
-> State: 1.8 <-
  state = AddPropertyDomain_teaches_Professor
-> State: 1.9 <-
  state = AddPropertyDomain_attends_Student
-> State: 1.10 <-
  state = AddObjectPropertyRange_teaches_Course
-> State: 1.11 <-
  state = AddObjectPropertyRange_attends_Course
-> State: 1.12 <-
  state = AddBasicClass_PhDStudent
-> State: 1.13 <-
  state = AddClassInheritance_PhDStudent_Student
-> State: 1.14 <-
  state = AddInstance_Luke_Professor
```

```
-> State: 1.15 <-
  state = AddInstance_Maths_Course
-> State: 1.16 <-
  state = AddInstance_Paul_PhDStudent
-- Loop starts here
-> State: 1.17 <-
  state = AddInstanceObjectProperty_Paul_teaches_Maths
-> State: 1.18 <-
-- specification F ((state = AddInstance_Paul_PhDStudent & F state = AddInstanceObjectProperty_Paul_teaches_Maths) & F state = AddPropertyDomain_teaches_Professor) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 2.1 <-
  state = AddBasicClass_Professor
  msg = nop
-> State: 2.2 <-
  msg = hasNext
-> State: 2.3 <-
  state = AddBasicClass_Student
-> State: 2.4 <-
  state = AddBasicClass_Course
-> State: 2.5 <-
  state = AddDisjointClass_Professor_Student
-> State: 2.6 <-
  state = AddObjectProperty_teaches
-> State: 2.7 <-
  state = AddObjectProperty_attends
-> State: 2.8 <-
  state = AddPropertyDomain_teaches_Professor
-> State: 2.9 <-
  state = AddPropertyDomain_attends_Student
-> State: 2.10 <-
  state = AddObjectPropertyRange_teaches_Course
-> State: 2.11 <-
  state = AddObjectPropertyRange_attends_Course
-> State: 2.12 <-
  state = AddBasicClass_PhDStudent
-> State: 2.13 <-
  state = AddClassInheritance_PhDStudent_Student
-> State: 2.14 <-
  state = AddInstance_Luke_Professor
-> State: 2.15 <-
  state = AddInstance_Maths_Course
-> State: 2.16 <-
  state = AddInstance_Paul_PhDStudent
-- Loop starts here
-> State: 2.17 <-
  state = AddInstanceObjectProperty_Paul_teaches_Maths
-> State: 2.18 <-
```

Script 30. La détection des incohérences par le model checker.

Dans le Script 30, on remarque que le model checker a détecté deux incohérences surlignées en jaune. Pour la première incohérence, l'instance *Paul* qui est un *doctorant* ne peut pas enseigner des cours de *math*, ce qui est totalement logique parce que la classe *Professeur* et la classe *étudiant* sont des classes disjointes. La deuxième erreur est que la propriété *enseigné* (*teaches*) est pour les *professeur* (Classe *Professor*) et pas pour l'instance Paul. Cette deuxième erreur confirme la première.

Bibliographie

(Abarbanel-Vinov et al., 2001) Abarbanel-Vinov, Y., Aizenbud-Reshef, N., Beer, I, Eisner, C., Geist, D., Heyman, T. Reuveni, I. Rippel, E., Shitsevalov, I., Wolfsthal, Y., Yatzkar-Haham, T. (2001). *On the effective deployment of functional formal verification*. Formal Methods in System Design, 19:35–44.

(Abiteboul et al., 1997) Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J. (1997). *The Lorel Query Language for Semistructured Data*. Int. Journal on Digital Libraries 1, pp. 68–88.

(Akers, 1978) S. B. Akers. (1978). Binary Decision Diagrams. IEEE Transactions on Computers archive. Volume 27 Issue 6, June 1978. Pages 509-516

(Alkhateeb, 2008) Alkhateeb, F. (2008). *Querying RDF(S) with Regular Expressions*. Thèse soutenue le 30 juin 2008. Université joseph Fourier, Grenoble.

(Alkhateeb et al., 2008) Alkhateeb, F., Baget, J. F. et Euzenat, J. (2008). *Constrained regular expressions in SPARQL*. In Proceedings of the 2008 International Conference on Semantic Web and Web Services (SWWS'08).

(Alkhateeb et al., 2009) Alkhateeb, F., Baget, J. F., et Euzenate, J. (2009). *Extending SPARQL with regular expression patterns (for querying RDF)*. Web Semantics, 7(2):57–73.

(Alpern et Schneider, 1985) Alpern, B. et Schneider, F. B. (1985). *Defining liveness*. Information Processing Letters. Volume 21, Issue 4, Pages 181–185.

(Alur et al., 1993) Alur, R., Courcoubetis, C. et Dill, D. (1993). *Model-Checking in Dense Real-Time*. Information and Computation, 104(1) : 2–34.

(Amnell et al., 2001) Amnell T., Behrmann, G., Bengtsson, J., D'argenio, P.R., David, A., Fehnker, A., Hune, T., Jeannet, B., Larsen, K.G., Möller, O., Pettersson, P., Weise, C., Yi, W. (2001). *UPPAAL – Now, Next, and Future*, Proc. Modelling and Verification of Parallel Processes (MOVEP2k), vol. 2067 de Lecture Notes in Computer Science, p. 99-124, Springer.

(Amnell et al., 2012) AMNELL T., BEHRMANN G., BENGTSSON J., D'ARGENIO P.R., DAVID A., FEHNKER A., HUNE T., JEANNET B., LARSEN K.G., MÖLLER O., PETTERSSON P., WEISE C., YI W. (2012). « UPPAAL », http://www.uppaal.com.

(Ammann et Offutt, 2008) Ammann, P. et Offutt, J. (2008). *Introduction to Software Testing*. Cambridge University Press.

(Angles et Angles, R. et Gutierrez, C. (2005). *Querying RDF Data from a Graph Database*

Gutierrez, 2005) *Perspective*. 2nd. European Semantic Web Conference (ESWC2005), May 2005, Heraklion, Greece. Lecture Notes in Computer Science, Volume 3532, pp. 346-360.

(Anyanwu et al., 2007) Anyanwu, K., Maduko, A. et Sheth, A. P. (2007). *SPARQ2L: towards support for subgraph extraction queries in RDF databases*. In Proceedings of the 16th international conference on World Wide Web (WWW'07), pages 797–806.

(Aristote, 2012) Aristote, définition de l'ontologie. (2012). http://fr.wikipedia.org/wiki/Ontologie_%28philosophie%29

(ARQ, 2012) *ARQ SPARQL Processor for Jena*. http://jena.sourceforge.net/ARQ/ (2012).

(Baader et Nutt, 2003) Baader, F. et Nutt, W. (2003). Basic description logics. Dans Baader, F., Calvanese, D., McGuinness, D., Nardi, D. et Patel-Schneider, P. (éditeurs), *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, pp. 47-100.

(Baader et al., 2003) Baader, F., Calvanese, D., McGuinness, D., Nardi, D. et Patel-Schneider, P. (2003). *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press.

(Baier et Katoen, 2008) Baier C. et Katoen, J. P. (2008). *Principles of Model Checking*. ISBN-10: 0-262-02649-X ISBN-13: 978-0-262-02649-9

(Baget et al., 2005) Baget, J. F., Canaud, E., Euzenat, J., Hacid, M. S. (2005). *Les langages du web sémantique*. Information – Interaction-Intelligence (I3) Une Revue en Sciences du Traitement de l'Information.

(Bardin, 2008) Bardin, S. (2008). *Introduction au Model Checking*. École Nationale Supérieure de Techniques Avancées.

(Bastide, 2008) Bastide, A. (2008). *Six sites pour tout savoir sur la RFID*. Indexel.

(Becket et McBride, 2004) Becket, D., McBride, B. (2004). *RDF/ XML Syntax Specification (Revised). W3C recommendation*. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/

(Ben-Ari, 2008) Ben-Ari, M. (2008). *Principles of the Spin model checker*. Editeur : Springer London Ltd, ISBN-10: 1846287693, ISBN-13: 978-1846287695

(Berners-Lee, 2001) Berners-Lee, T. (2001). *Notation 3 - An RDF Language for the Semantic Web*. http://www.w3.org/DesignIssues/Notation3

(Berners-Lee et al., 2001) Berners-Lee, T., Hendler, J. et Lassila, O. (2001). *The Semantic Web*. Scientific American. pp. 34–43.

(Beth, 1965) Beth, E. W. (1965). *The Foundations of Mathematics*. North Holland, 1959. Second edition revised in 1965.

(Blakeley, 2007) Blakeley, C. (2007). Virtuoso RDF Views. http://www.openlinksw.com

(Bray et al., 2006) Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, M., Yergeau, E., Cowan, F. (2006). *Extensible Markup Language (XML) 1.1 (second edition) W3C recommendation*.

(Brun, 1998) Brun, P. (1998). *XTL : une logique temporelle pour la spécification formelle des systèmes interactifs*. Thèse de doctorat à l'université paris XI Orsay, France.

(Bruns et Godefroid, 2001) Bruns, G. et Godefroid, P. (2001). *Temporal Logic Query-Checking*, Proc. 16th Ann. IEEE Symp. Logic in Computer Science (LICS '01), pp. 409-417.

(Burch et al., 1992) Burch, J. R., Clarke, E. M., Mcmillan, K. L., Dill, D. L., Hwang, L. J. (1992). *Symbolic Model Checking: 10^{20} States and Beyond*. *Information and computation*. 142- 170.

(Brickley and Guha, 2000) Brickley, D. et Guha, RV. (2000), *Resource Description Framework (RDF)* Schema Specification 1.0: W3C Candidate Recommendation 27 Mars 2000.

(Brickley et al., 2004) Brickley, D., Guha, R.V., McBride, B. (2004). *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation 10 February 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/

(Browne et al., 1986) Browne, M. C., Clarke, E. M., Dill D. L. et Mishra, B. (1986). *Automatic verification of sequential circuits using temporal logic.* IEEE Transactions on Computers, 35(12):1035–1044.

(Bry et al., 2008) Bry, F., Furche, T., Ley, C., Linse, B. et Marnette, B. (2008). *RDFLog: It's Like Datalog for RDF*. In WLP.

(Bryant, 1986) Randal E. Bryant. (1986). *Graph-Based Algorithms for Boolean Function Manipulation,*'' IEEE Transactions on Computers, Vol. C - 35, No. 8, pp. 677 – 691.

(Bryant, 1992) Bryant, R. E. (1992). *Symbolic Boolean manipulation with ordered binary-decision diagrams.* ACM Computing Surveys (CSUR) 24 (3), 293-318.

(Büchi, 1962) Büchi, J. R. (1962). *On a decision method in restricted second order arithmetic.* In Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science, Berkley, 1960, pages 1–11. Standford University Press.

(Castano, 2006) Castano, S. (2006). *Ontology evolution: The BOEMIE approach*. BOEMIE Workshop, at the conference (EKAW 06).

(Castano et al., 2007) Castano, S., Espinosa, S., Ferrara, A., Karkaletsis, V., Kaya, A., Melzer, S. (2007). *Ontology dynamics with multimedia information: The BOEMIE evolution methodology*. In G. Flouris, & M. d'Aquin (Eds.) Proceedings of the International Workshop on Ontology Dynamics (IWOD-07) at ESWC 07 Conference (pp. 41-54).

(CBS, 2009) CBS Interactive. (2009). *Cloud Computing : vers la dématérialisation des salles informatiques* - ZDNet.fr.

(Chan, 2000) Chan, W. (2000). *Temporal-Logic Queries*,” Proc. 12th Conf. Computer Aided Verification (CAV ’00), pp. 450-463.

(Champeau, 2007) Champeau, G. (2007). *Le Web 3.0 : l'alliance du P2P et du Web 2.0 ?* Futura-Techno.

(Chebotko et al., 2006) Chebotko, A., Lu, S., Jamil, H. M. et Fotouhi, F. (2006). *Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns*. Technical Report TR-DB-052006-CLJF. http://www.cs.wayne.edu/»artem/main/research/TR-DB-052006-CLJF.pdf

(Chechik, et Gurfinkel, 2003) Chechik, M. et Gurfinkel, A. (2003). TLQSolver: *A Temporal Logic Query Checker*. In Proceedings of 15th International Conference on Computer Aided Verification (CAV'03), LNCS 2725, pp. 210-214.

(Chiyangwa et Kwiatkowska, 2003) Chiyangwa, S. et Kwiatkowska, M. Z. (2003). *Modelling Ad hoc On-Demand Distance Vector (AODV) Protocol with Timed Automata*. In Proceedings of the Third Workshop on Automated Verification of Critical Systems (AVoCS’03), Southampton, UK.

(Chrisment et al., 2006) Chrisment, C., Hernandez, N., Genova, F., Mothe, J. (2006). *D’un thesaurus vers une ontologie de domaine pour l’exploration d’un corpus*. AMETIST, INIST, Vol. 0, p. 59-92.

(Cimatti et al., 2000) Cimatti, E. M., Clarke, E., Giunchiglia, F. et Roveri, M. (2000). *NuSMV: a new symbolic model checker.* International Journal on Software Tools for Technology Transfer (STTT), 2(4).

(Cimatti et al., 2002) Cimatti, E. M, Clarke, E., Giunchiglia, F., Giunchiglia, M., Pistore, M., Roveri, R., Sebastiani, R. et Tacchella, A. (2002). *NuSMV 2: An OpenSource Tool for Symbolic Model checking*. *Proceedings of Computer Aided Verification (CAV 02)*, 2002.

(Cimiano et Völker, 2005) Cimiano, P., & Vlker, J. (2005). *Text2Onto - a framework for ontology learning and data-driven change Discovery.* In A. Montoyo, R. Munoz, & E. Metais (Eds.), LNCS: Vol. 3513. Natural Language. Processing and Information Systems (pp. 227-238). Berlin, Germany: Springer. doi: 10.1007/b136569

(Cimiano, 2007) Cimiano, P. (2007). *On the relation between ontology learning, engineering, evolution and expressivity*. Invited talk at 7th Meeting on Terminology and Artificial Intelligence TIA 2007. Sophia Antipolis, France.

(Clarke et al., 1986) Clarke, E. M., Emerson, E. A. et Sistla, A. P. (1986). *Automatic verification of finite-state concurrent systems using temporal logic specifications*. ACM Transactions on Programming Languages and Systems, 8(2):244–263.

(Clarke et al., 1999) Clarke, E. M., Grumberg, O. et Peled, D. A. (1999). *Model checking*. MIT Press.

(Coen-Porisini et Coen-Porisini, A., Ghezzi, C. et Kemmerer, R. (1997). *Specification of realtime*

al., 1997) *systems using ASTRAL*, IEEE Transactions on Software Engineering.

(Coleri et al., 2002) Coleri, S., Ergen, M. et Koo, T. (2002). *Lifetime analysis of a sensor network with hybrid automata modeling*, in 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA).

(Cook, 1971) Cook, S. (1971). *The complexity of theorem-proving procedures*. In 3rd Annual ACM. Symposium on Theory of Computing, pages 151–158. ACM Press.

(Corby et al., 2004) Corby, O., Dieng-Kuntz, R. et Faron-Zucker, C. (2004). *Querying the Semantic web with Corese Search Engine*. In Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'2004), sub-conference (PAIS'2004), Valencia (Spain), pages 705–709.

(Courcoubetis et al., 1992) Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M. (1992). *Memory-efficient algorithms for the verification of temporal properties*. FORMAL METHODS IN SYSTEM DESIGN, pp. 275-288.

(Cousot et Cousot, 1976) Cousot, P. et Cousot, R. (1976). *Static Determination of Dynamic Properties of Programs.* In B. Robinet, editor, Proceedings of the second international symposium on Programming, Paris, France, pages 106—130, Dunod, Paris.

(Cousot et Cousot, 1977) Cousot, P. et Cousot, R. (1977). *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* In Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238—252, Los Angeles, California, 1977. ACM Press, New York, NY, USA.

(Cousot, 2000) Cousot, P. (2000). *Interprétation abstraite*. Technique et Science Informatique, Vol. 19, Nb 1-2-3. Hermès, Paris, France. pp. 155-164.

(Couvreur, 1999) Couvreur, J. M. (1999). *On-the-fly verification of temporal logic*. In Jeannette M. Wing, Jim Woodcock, et Jim Davies, editors, Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99), volume 1708 of Lecture Notes in Computer Science, pages 253–271, Toulouse, France. Springer-Verlag.

(Couvreur, 2000) Couvreur, J. M. (2000). *Un point de vue symbolique sur la logique temporelle linéaire. I*n Pierre Leroux, editor, Actes du Colloque LaCIM 2000, volume 27 of Publications du LaCIM, pages 131–140. Université du Québec à Montréal.

(Cyganiak, 2005) Cyganiak, R. (2005). *A relational algebra for SPARQL*. Digital Media Systems Laboratory. HP Laboratories Bristol. HPL-2005-170.

(Cruz et al., 1987) Cruz, I.F., Mendelzon, A.O., Wood, P.T. (1987). *A Graphical Query Language Supporting Recursion*. SIGMOD Rec. 16, pp. 323–330.

(Cruz et al., 1988) Cruz, I. F., Mendelzon, A. O. et Wood, P. T. (1988). *G+: Recursive queries without recursion.* In Proceedings of Second International Conference on Expert Database

Systems, pages 355–368.

(Dang et Kemmerer, 1999) Dang, Z. et Kemmerer, R. A. (1999). *Using the ASTRAL model checker to analyze Mobile IP*. Software Engineering, 1999. Proceedings of the 1999 International Conference on Software Engineering.

(Daniele et al., 1999a) Daniele, M., Giunchiglia, F. et Vardi, M. Y. (1999a). *Improved automata generation for Linear Temporal Logic.* Technical report, ITC-IRSC.

(Daniele et al., 1999b) Daniele, M., Giunchiglia, F. et Vardi, M. Y. (1999b). *Improved automata generation for Linear Temporal Logic.* In N. Halbwachs et D. Peled, editors, Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99), volume 1633 of Lecture Notes in Computer Science, pages 249-260. Springer- Verlag.

(Nicola et Vaandrager, 1990) De Nicola R. et Vaandrager, F. W. (1990). *Action versus State based Logics for Transition Systems.* Semantics of Systems of Concurrent Processes: 407-419.

(Dingel et Filkorn, 1995) Dingel, J. et Filkorn, T. (1995). *Model Checking for infinite state systems using data abstraction, assumption commitment style reasoning and theorem proving.* In 7th International Conference on Computer Aided Verification (CAV), volume 939 of Lecture Notes in Computer Science, pages 54–69. Springer-Verlag.

(Djedidi, 2009) Djedidi, R. (2009). *Approche d'évolution d'ontologie guidée par des patrons de gestion de changement*, Thèse de doctorat.

(Djedidi et Aufaure, 2009) Djedidi, R. et Aufaure, M. A. (2009). *Patrons de gestion de changements. Ingénieries des connaissances*.

(Djedidi et al. 2007) Djedidi, R., Abboute,H., Aufaure M.A. (2007). *Évolution d'ontologie : Validation des changements basée sur l'évaluation*. Actes de premières journées francophones sur les ontologies (JFO).

(Duret-Lutz et Rebiha, 2003) Duret-Lutz, A. et Rebiha, R. (2003). *SPOT : une bibliothèque de vérification de propriétés de logique temporelle à temps linéaire*. Mémoire de DEA.

(Emerson et Sistla, 1984) Emerson E. A. et Sistla, A. P. (1984). *Deciding Full Branching Time Logic. Information and Control*, 61(3), pages 175–201, Academic Press

(Emerson et Halpern, 1985) Emerson, E. A. et Halpern, J. Y. (1985). *Decision Procedures and Expressiveness in the Temporal Logic of Branching Time*. Journal of Computer and System Sciences, 30(1), pages 1–24, Academic Press.

(Emerson et Halpern, 1986) Emerson, E. A. et Halpern, J. Y. (1986). *"Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic*. Journal of the ACM, 33(1), pages 151–178, ACM Press.

(Emerson et Lei, 1987) Emerson E. A. et Lei, C. L. (1987). *Modalities for Model Checking : Branching Time Logic Strikes Back.* Science of Computer Programming, 8(3), pages 275–306,

Elsevier Science.

(Emerson, 1990) Emerson, E. A. (1990). *Temporal and Modal Logic*. Dans Jan van Leeuwen, éditeur, Handbook of Theoretical Computer Science, volume B, chapitre 16, pages 995–1072. Elsevier Science.

(Etessami et Holzmann, 2000) Etessami K. et Holzmann, H. J. (2000). *Optimizing Büchi automata*. In C. Palamidessi, editor, Proceedings of the 11th International Conference on Concurrency Theory (Concur'2000), volume 1877 of Lecture Notes in Computer Science, pages 153–167. Springer-Verlag.

(Fernandez et al., 1996) Fernandez, J. -C., Garavel, H., Kerbrat, A., Mounier, L., Mateescu, R. et Sighireanu, M. (1996). *CADP : a protocol validation and verification toolbox.* In Rajeev Alur and Thomas A. Henzinger, editors, Proceedings of the Eighth International Conference on Computer Aided Verification CAV, volume 1102, pages 437–440, New Brunswick, NJ, USA, / 1996. Springer Verlag.

(Flouris et al., 2005) Flouris, G., Plexousakis, D., et Antoniou, G. (2005). *On applying the AGM theory to DLs and OWL.* In Y.Gil, E. Motta, V. Benjamins, & M. Musen, (Eds.), LNCS: Vol. 3729. The Semantic Web – ISWC 2005 (pp. 216-231). Berlin, Germany: Springer. doi: 10.1007/11574620

(Flouris, 2006) Flouris, G. (2006). *On belief change and ontology evolution*. Ph.D. Thesis, University of Crete, Department of Computer Science, Heraklion, Greece.

(Flouris et al., 2006a) Flouris, G., Plexousakis, D., & Antoniou, G. (2006a). *Evolving ontology evolution.* Invited Talk at the 32nd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM-06).

(Flouris & Plexousakis, 2006) Flouris, G., & Plexousakis, D. (2006). *Bridging ontology evolution and belief change.* LNCS: Vol. 3955, Advances in Artificial Intelligence (pp. 486-489). Berlin, Germany: Springer. doi: 10.1007/11752912_51.

(Flouris et al., 2007) Flouris, F., Manakanatas, D., Kondylakis, H., Plexousakis, D., Antoniou, G. (2007). *Ontology Change: Classification & Survey - The Knowledge Engineering Review*, 1–29, Cambridge University Press.

(Floyd, 1967) Floyd, R. W. (1967). *Assigning meanings to programs.* Proceedings of the American Mathematical Society Symposia on Applied Mathematics. Vol. 19, pp. 19–31.

(Fribourg et Peixoto, 1994) Fribourg, L. et Veloso Peixoto, M. (1994). *Automates Concurrents à Contraintes*. Technique et Science Informatiques 13(6), pages 837-866.

(Fröhlich et Werner, 1994) Fröhlich, M. et Werner, M. (1994). *The Graph Visualization System daVinci—A User Interface for Applications*, Technical Report 5/94, Dept. of Computer Science, Bremen Univ., 1994, citeseer.nj.nec.com/fr94graph.html.

(Gabbay et al., Gabbay, D., Pnueli, A., Shelah, S. et Stavi, J. (1980). *On the temporal analysis. of*

1980) *fairness*. In Proceedings of the 12th ACM Symposium on Principles of Programming Languages (PoPL'80), pages 163-173, Las Vegas.

(Gabbay et al., 1989) Gabbay, D. M. (1989). *The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems*. Dans Behnam Banieqbal, Howard Barringer, et Amir Pnueli, éditeurs, Proceedings of the 1st Conference on Temporal Logic in Specification, Avril 1987, volume 398 de Lecture Notes in Computer Science, pages 409–448. Springer-Verlag.

(Gagnon, 2007) Gagnon, M. (2007.) *Brève introduction à RDF*. Ecole polytechnique de Montréal.

(Gai, 2007) Gai Anh-Tuan. (2007). *Web 3.0 : une autre branche pour l'arbre des possibles*. Le Monde.

(Gandon, 2008) Gandon. F. (2008). *Le "futur" du web à la lecture des recommandations du W3C.*

(Gangemi et al., 2004) Gangemi, A., Catenacci, C. et Battaglia, M. (2004). *Inflammation ontology design pattern: an exercise in building a core biomedical ontology with descriptions and situations*. In D.M. Pisanelli (Ed.) Ontologies in Medicine. IOS Press, Amsterdam.

(Gastin et Oddoux, 2001) Gastin, P. et Oddoux, D. (2001). *Fast LTL to Büchi automata translation.* In G. Berry, H. Comon, et A. Finkel, editors, Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01), volume 2102 of Lecture Notes in Computer Science, pages 53-65. Springer-Verlag.

(Gemis et Paredaens, 1993) Gemis, M., Paredaens, J. (1993). *An Object-Oriented Pattern Matching Language*. In: Proc. 1th ISOTAS, Springer-Verlag, pp. 339–355.

(Gerth et al., 1995) Gerth, R., Peled, D., Vardi, M. Y., Gerth, R., Eindhoven, D. D., Peled, D., Vardi, M. Y., Wolper, P. (1995). *Simple On-the-fly Automatic Verification of Linear Temporal Logic*. In Protocol Specification Testing and Verification 1995. Pages 3-18,Warsaw, Poland, 1995. Chapman & Hall.

(Gheorghiu et Gurfinkel, 2006) Gheorghiu, M. et Gurfinkel, A. (2006). *TLQ: A Query Solver for States*. In Tools and Posters track of Formal Methods 2006, 4 pages.

(Ghezzi et Kemmerer, 1991) Ghezzi, C. et Kemmerer, R. (1991). *ASTRAL: An assertion language for specifying real-time systems*, Proceedings of the 3rd European Software Engineering Conference, pp.122-146.

(Giannakopoulou et Lerda, 2001) Giannakopoulou, D et Lerda, F. (2001). *Efficient translation of LTL formulae into Büchi automata*. Technical Report 01.29, Research Institute for Advanced Computer Science.

(Giannakopoulou et Lerda, 2002) Giannakopoulou, D. et Lerda, F. (2002). *From states to transitions: Improving translation of LTL formulae to Büchi automata*. In D.A. Peled et M.Y. Vardi, editors, Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02), volume 2529 of

Lecture Notes in Computer Science, pages 308–326, Houston, Texas. Springer-Verlag.

(Godefroid et al., 1995) Godefroid, P., Holzmann, G. J., Pirottin, D. (1995). *State-Space Caching Revisited*. Formal Methods in System Design.

(Godefroid et al., 1996) Godefroid, P., Peled, D., Staskauskas, M. G. (1996). *Using Partial-Order Methods in the Formal Validation of Industrial Concurrent Programs*. ISSTA 1996: 261-269

(Gruber, 1993) Gruber, T., R. (1993*). À translation approach to portable ontologies-Knowledge Acquisition* 5(2):199-220. http://tomgruber.org/writing/ontolingua-kaj-1993.pdf

(Gueffaz et al., 2011a) Gueffaz, M., Rampacek, S., Nicolle, C. (2011a). *RDF2NµSMV: Mapping semantic graphs to NµSMV model checker*, The Third International Conference on Advances in Future Internet (AFIN 2011), Nice/Saint Laurent du Var, France.

(Gueffaz et al., 2011b) Gueffaz, M., Rampacek, S., Nicolle, C. (2011b). *RDF2SPIN: Mapping semantic graphs to SPIN model checker*, The International Conference on Digital Information and Communication Technology (DICTAP), Dijon, France, Springer-Verlag.

(Gueffaz et al., 2011c) Gueffaz, M., Rampacek, S., Nicolle, C. (2011c). SCALESEM: *Evaluation of semantic graph based on Model Checking,* Webist 2011- The 7th International Conference on Web Information Systems and Technologies, Noordwijkerhout, Hollande, INSTICC, ACM SIGMIS.

(Gueffaz et al., 2011d) Gueffaz, M., Rampacek, S., Nicolle, C. (2011d). *Qualifying semantic graphs using Model Checking.* 7th International Conference on Innovations in Information Technology (Innovations'11), Abu Dhabi, Emirats arabes Unis, Sponsored by IEEE, April 2011.

(Gueffaz et al., 2011e) Gueffaz, M., Rampacek, S., Nicolle, C. (2011e). *A new approach based on NµSMV model to query semantic graph.* The International Conference on Digital Information Processing and Communications (ICDIPC 2011), Ostrava, République Tchèque, Springer-Verlag, Juillet 2011.

(Gueffaz et al., 2011f) Gueffaz, M., Rampacek, S., Nicolle, C. (2011f). *Mapping SPARQL Query to temporal logic query based on NµSMV model checker to Query Semantic Graphs.* International Journal of Digital Information and Wireless Communications (IJDIWC) 1, 1 (2011) 64-74, 2011.

(Gueffaz et al., 2012a) Gueffaz, M., Rampacek, S., Nicolle, C. (2012a). *Temporal Logic To Query Semantic Graphs Using The Model checking Method*. Journal of Software, Vol 7, No 7, pp. 1462-1472, Juillet 2012.

(Gueffaz et al., 2012b) Gueffaz, M., Rampacek, S., Nicolle, C. (2012b). *Mapping SPARQL Query to temporal logic query based on NµSMV model checker to Query Semantic Graphs*. International Journal of Digital Information and Wireless Communications

(IJDIWC) 1, 2 (2012) 366-380.

(Gueffaz et al., 2011c) Gueffaz, M., Perrine, P., Rampacek, S., Cruz, C., Nicolle, C. (2012c). *Inconsistency identification in dynamic ontologies based on Model Checking.* The 8th International Conference on Web Information Systems and Technologies, Porto : Portugal, 2012

(Guillaud, 2008) Guillaud, Hubert. (2008). *Web sémantique : y aura-t-il une application phare ? Futura-Techno.*

(Gurfinkel et al., 2002) Gurfinkel, A., Devereux, B. et Chechik, M. (2002). *Model Exploration with Temporal Logic Query Checking*, in Proceedings of SIGSOFT Conference on Foundations of Software Engineering (FSE'02), November 2002, pp. 139-148

(Gurfinkel et Chechik, 2003) Gurfinkel A. et Chechik, M. (2003). *Proof-Like Counterexamples, Proc. Ninth Int'l Conf.* Tools and Algorithms for the Construction and Analysis of Systems (TACAS '03), pp. 160-175.

(Gutierrez et al., 2004) Gutierrez, C., Hurtado, C., Mendelzon, A., O. (2004). Foundations of Semantic Web Databases, Proceedings ACM Symposium on Principles of Database Systems (PODS), Paris, France, pp. 95 - 106.

(Gurfinkel et al., 2003) Gurfinkel, A., Chechik, M., Devereux, B. (2003). *Temporal Logic Query Checking: A Tool for Model Exploration*. IEEE Trans. Software Eng. 29(10): 898-914.

(Güting, 1994) Güting, R.H. (1994). *GraphDB: Modeling and Querying Graphs in Databases.* In: Proc. of 20th VLDB Conference, Morgan Kaufmann, pp. 297–308.

(Haase et al., 2004) Haase, P., Broekstra, J., Eberhart, A., et Volz, R. (2004). *A comparison of RDF query languages*. In Proceedings of the International Semantic Web Conference (ISWC), 502–517.

(Haase et Stojanovic, 2005) Haase, P., Stojanovic, L. (2005). *Consistent Evolution of OWL Ontologies.* In A.Gomez-Perez, J. Euzenat (Eds.), LNCS, vol.3532. The Semantic Web: Research and Applications (pp. 182-197). Berlin, Germany: Springer. doi: 10.1007/b136731.

(Han et Hofmeister, 2006) Han M. et Hofmeister, C. (2006). *Modeling and Verification of Adaptive Navigation in Web Applications*" in Proc. International Conference on Web Engineering (ICWE'06), pp. 329-336.

(Harel, 1988) Harel, D. (1988). *On visual formalisms*, Communications of the ACM, 31(5):514-530, may 1988.

(Harris et Seaborne, 2012) Harris, S. et Seaborne, A. (2012). *SPARQL 1.1 Query Language*. W3C Working Draft 24 July 2012. http://www.w3.org/TR/sparql11-query/

(Haydar et al., 2004) Haydar, M., Petrenko, A. et Sahraoui, H. (2004). *Formal Verification of Web Applications Modeled by Communicating Automata*, in Proc. International Conference on Formal Techniques for Networked and Distributed Systems (FORTE2004), LNCS 3235, pp. 115-132.

(Haydar et al., 2005) Haydar, M., Boroday, S., Petrenko A. et Sahraoui, H. (2005). *Properties and Scopes in Web Model Checking,* in Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE '05), pp. 400-404.

(Heflin et al., 1999) Heflin, J., Hendler, J., & Luke, S. (1999). *Coping with changing ontologies in a distributed environment*. Proceedings of the Workshop on Ontology Management of the 16th National Conference on Artificial Intelligence (AAAI-99), WS-99-13, 74-79, AAAI Press.

(Heflin et Hendler, 2000) Heflin, J., et Hendler, J. (2000). *Dynamic Ontology on the Web*. Paper presented at the AAAI, 17th National Conference on artificial Intelligence.

(Held, 1987) Held, G. (1987). *Data Compression: Techniques and Applications, Hardware and Software Considerations*, second edition, John Wiley & Sons, New York, NY.

(Hennesy et Milner, 1985) Hennesy, M., Milner, R. (1985). *Algebraic laws for nondeterminism and concurrency.* Journal of the ACM.

(Henzinger et Wong-toi, 1997) Henzinger, T.A., Ho, P.H., Wong-toi, H. (1997). *HYTECH: A Model-Checker for Hybrid Systems*, Journal on Software Tools for Technology Transfer, vol. 1, n° 1-2, p. 110- 122, Springer.

(Herman, 2007) Herman, I. (2007). "SPARQL is a Candidate Recommendation". Semantic Web Activity News. World Wide Web Consortium. http://www.w3.org/blog/SW/2007/06/15/sparql_is_a_candidate_recommendation/

(Hoare, 1969) Hoare, C. A. R. (1969). *An axiomatic basis for computer programming*. Communications of the ACM, 12(10):576–580,583. doi:10.1145/363235.363259

(Holzmann, 1991) Holzmann, G. J. (1991). *Design and Validation of Computer Protocol,* Prentice Hall, New Jersey, 1991, ISBN 0-13-539925-4. Disponible sous forme électronique sur le site web de l'auteur.

(Holzmann, 1997) Holzmann, G. J. (1997). *State Compression in Spin. Proc*. Third Spin Workshop, Twente University, The Netherlands.

(Holzmann, 2003) Holzmann, G. J. (2003). *The SPIN Model Checker* - Primer and Reference Manual. Addison-Wesley, Pearson Education.

(Homma et al., 2009) Homma, K., Takahashi, K., Togashi, A. (2009). *Modeling and Verification of Web Applications Using Formal Approach*. IEICE Tech. Rep., vol. 109, no. 40, SS2009-8, pp. 43-48.

(Hornus, 2001) Hornus, S. (2001). *Requêtes en logique temporelle*. Mémoire DEA algorithmique, Master thesis in computer aided verification, in French. Stage effectué au LSV de l'ENS Cachan, sous la direction de Philippe Schnoebelen.

(Hornus et Schnoebelen, Hornus, S. et Schnoebelen, P. (2002). *On solving temporal logic queries*. International Conference on Algebraic Methodology And Software Technology

2002) (AMAST), september Lecture Notes in Computer Science (LNCS 2422), Springer.

(Horridge et al., 2008) Horridge, M., Parsia, B., Sattler, U. (2008). *Explanation of OWL Entailments in Protégé 4*, Poster et démonstration dans le cadre de la conférence ISWC 2008.

(Huang et al., 2004) Huang, Y., Yu, F., Hang, C., Tsai, C., Lee, D. T., Kuo, S. (2004). *Verifying web applications using bounded Model Checking,* In DSN 04 Proceedings of the International Conference on Dependable Systems and Networks Page 199.

(Huffman, 1952) Huffman, D. A. (1952). *A method for the construction of minimum-redundancy codes*, Proceedings of the I.R.E., pp 1098-1102.

(Hutt, 2005) Hutt, K. (2005). A *Comparison of RDF Query Languages*. 21st Computer Science Seminar SE1-T4-1.

(Jaffar et Lasez, 1987) Jaffar, J., Lassez, J. L. (1987). *Constraint Logic Programming*. Principal Of Programming Language (POPL). pp 111-119.

(Kalyanpur et al., 2006) Kalyanpur, A., Parsia, B., Sirin, E., Grau, B.C., Hendler, J.A. (2006). *Swoop: A web ontology editing browser*. Journal of Web Semantics 4(2), 144–153.

Karvounarakis et al., 2002 Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M. (2002). RQL: A Declarative Query Language for RDF. In: Proc. of the 11th WWW conference, ACM Press. Pages 592–603.

(Kesten et al., 1993) Kesten, Y., Manna, Z., McGuire, H. et Pnueli, A. (1993). *A decision algorithm for full propositional temporal logic*. In C. Courcoubertis, editor, Proceedings for the 5th Conference on Computer Aided Verification (CAV'93), volume 697 of Lectures Notes in Computer Science, pages 97–109. Springer-Verlag.

(Kiefer et al., 2007) Kiefer, C., Bernstein, A., Lee, H. J., Klein, M. et Stocker, M. (2007). *Semantic process retrieval with iSPARQL*. In Proceedings of the 4th European Semantic Web Conference (ESWC ’07). Springer.

(Kifer et al., 1995) Kifer, M., Lausen, G. et Wu, J. (1995). *Logical Foundations of Object-Oriented and Frame-Based Languages*. J. ACM, 42(4):741–843.

(Klein et Fensel, 2001) Klein, M. et Fensel, D. (2001). *Ontology versioning for the Semantic Web*. Journal of the First International Semantic Web Working Symposium (SWWS), pages 75-91.

(Klein, 2002) Klein, M. (2002). *Supporting evolving ontologies on the internet*; Journal of the EDBT 2002 PhD Workshop.

(Klein, 2004) Klein, M. (2004), *Change Management for Distributed Ontologies*. PhD thesis, Vrije University of Amsterdam.

(Kochut et Janik, 2007) Kochut, K. et Janik, M. (2007). *SPARQLeR: Extended SPARQL for semantic association discovery.* In Proceedings of 4th European Semantic Web Conferenc (ESWC’07), pages 145–159.

(Kupferman et al., 2000) Kupferman, O., Vardi, M. Y., Wolper, P. (2000). *An automata-theoretic approach to branching-time model checking*. Journal of the ACM 47(2): 312-360.

(Lacot, 2005) Lacot, X. (2005). *Introduction à OWL, un langage XML d'ontologies Web*. Juin 2005.

(Lam et al., 2008) Lam, J., Sleeman, D., Pan, J., et Vasconcelos, W. (2008). *A Fine-Grained Approach to Resolving Unsatisfiable Ontologies*. In Journal on Data Semantics X (Vol. 4900/2008, pp. 62-95): Springer.

(Laroussinie et al., 2002) Laroussinie, F., Markey, N. et Schnoebelen, P. (2002). *Temporal logic with forgettable past*. In Proc. 17th IEEE Symp. Logic in Computer Science (LICS'2002), Copenhagen, Denmark, July 2002, pages 383-392. IEEE Computer Society.

(Laublet et al., 2002) Laublet, P., Reynaud, C., Charlet, J. (2002). *Sur quelques aspects du Web sémantique.* Assises du GDR I3, Editions Cépadues, Nancy.

(Laublet et al., 2004) Laublet P., Charlet J. et Reynaud, C. (2004). *Introduction au Web sémantique. Information Interaction Intelligence*, Hors-série 2004, p 7-20.

(Lausen et al., 2008) Lausen, G., Meier M. et Schmidt, M. (2008). *SPARQLing Constraints for RDF.* In EDBT, pages 499–509.

(Lichtenstein et al., 1985) Lichtenstein, O., Pnueli, A. et Zuck, L. (1985). *The glory of the past. In Proceedings 3rd Workshop on Logics of Programs*, Brooklyn, NY, USA, 17-19 June 1985, volume 193 of Lecture Notes in Computer Science, pages 196{218. Springer Verlag, Berlin.

(Liggesmeyer et al., 1998) Liggesmeyer, P., Rothfelder, M., Rettelbach, M. et Ackermann, T. (1998). *Qualit¨atssicherung Software-basierter technischer Systeme*. Informatik Spektrum, 21(5):249–258.

(Liotard, 2008) Liotard, C. (2008). *Web 1.0 - Web 2.0 - Web 3.0.* L'Atelier Informatique.

(Lowe, 1995) Lowe, G. (1995). *An Attack on the Needham-Schroeder Public-Key Authentication Protoc*ol. Information Processing Letters, volume 56, number 3, pages 131-133.

(Lowe, 1996) Lowe, G. (1996). *Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR*. In Tools and Algorithms for the Construction and Analysis of Systems, Margaria and Steffen (eds.), volume 1055 of Lecture Notes in Computer Science, Springer Verlag, pages 147-166, 1996. Also in Software Concepts and Tools, 17:93-102.

(Luong, 2004) Luong, P. H. (2004). *Gestion de l'évolution d'un web sémantique d'entreprise*. Thèse de doctorat, école doctorale sciences et technologie de l'information et de la communication, école des mines de Paris.

(Luong et Dieng-Kuntz, 2007) Luong, P-H. et Dieng-Kuntz, R. (2007). *A Rule-based Approach for Semantic Annotation Evolution*. Journal of The Computational Intelligence, 23(3):320-338. Blackwell Publishing, Malden, MA 02148.

(Lynch, 1985) Lynch, T. D. (1985). *Data Compression Techniques and Applications, Lifetime Learning Publications*, Belmont, CA.

(Maedche et al., 2003) Maedche, A., Motik, B., et Stojanovic, L. (2003). *Managing Multiple and Distributed Ontologies in the Semantic Web*. VLDE Journal -Special Issue on Semantic Web, 12,286-302,

(Magkanaraki et al., 2002) Magkanaraki, A., Karvounarakis, G., Anh, T.T., Christophides, V., Plexousakis, D. (2002). *Ontology Storage and Querying.* Tech. Report 308, ICS-FORTH – Hellas.

(Manach, 2009) Manach, J. (2009). *RFiD 2.0 : des puces ou des ordinateurs* ? Internet Actu.

(Manna et Pnueli, 1992) Manna Z. et Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems*, volume I (Specification). Springer Verlag, 1992.

(Manna et Pnueli, 1995) Manna, Z. et Pnueli, A. (1995). *Temporal Verification of Reactive Systems - Safety.* Springer-Verlag.

(Marin, 2004) Marin, D. (2004). *RDF Formalization. Technical report, Universidad de Chile*, TR/DCC-2006-8. http://www.dcc.uchile.cl/~cgutierr/ftp/draltan.pdf.

(Markey, 2003) Markey, N. (2003). *Logiques temporelles pour la vérification : expressivité, complexité, algorithmes.* Thèse de doctorat à l'université d'Orléans.

(Mateescu et al., 2009) Mateescu, R. Meriot, S. et Rampacek, S. (2009). *Extending SPARQL with Temporal Logic*, Rapport technique.

(Mathur, 2008) Mathur, A. P. (2008). *Foundations of Software Testing*. April 17, 2008 | ISBN-10: 8131716600 | ISBN-13: 978-8131716601 | Edition: 1.

(McGuinness, 2000) McGuinness, D. (2000). *Conceptual Modeling for Distributed Ontology Environments.* Paper presented at the ICCS 2000, Darmstadt, Germany.

(McMillan, 1992) McMillan, K. L. (1992). *The SMV system -- DRAFT*. Available at http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.r2.2.ps.

(McMillan, 1993) McMillan, K. L. (1993). *Symbolic Model checking - An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

(Minsky, 1981) Minsky, M. (1981*). A Framework for Representing Knowledge.* Mind Design. MIT Press. Pp. 95-128.

(Monin, 2000) Monin J. F. (2000). *Introduction aux méthodes formelles*. Hermès, préface de G. Huet. 2-7462-0140-2.

(Munoz et al., 2007) Munoz, S., Pérez, J. et Gutierrez, C. (2007). *Minimal Deductive Systems for RDF*. In ESWC, pages 53–67.

(Myers, 1979) Myers, G. J. (1979). *The Art of Software Testing*. John Wiley & Sons.

(Nardi et Nardi, D. et Brachman, R. J. (2003). *An introduction to description logics*. Dans

Brachman, 2003)	Baader, F., Calvanese, D., McGuinness, D., Nardi, D. et Patel-Schneider, P. (éditeurs), The Description Logic Handbook : Theory, Implementation and Applications. Cambridge University Press, pp. 544.
(Nelson, 1991)	Nelson, M. R. (1991). *The Data Compression Book*, M&T Books, Redwood City, CA.
(Neumann et Weikum, 2009)	Neumann T. et Weikum, G. (2009). *Scalable Join Processing on very large RDF graphs.* SIGMOD'09.
(Noy, 2004)	Noy, N. (2004). *Tools for Mapping and Merging Ontologies*. In S. Staab et R. Studer (Eds.), Handbook on Ontologies: Springer Verlag,
(Noy et Musen, 2004)	Noy, N.F. et Musen, M.A. (2004). *Ontology Versioning in an Ontology Management Framework*. IEEE Intelligent Systems, Vol. 19, No. 4.
(Noy et al., 2006)	Noy, N. F., Chugh, A., Liu, W., Musen, M. A. (2006*). A framework for ontology evolution in collaborative environments*. LNCS: Vol. 4273. The Semantic Web - ISWC 2006 (pp. 544-558). Berlin, Germany: Springer. doi: 10.1007/11926078
(Oliver et al., 1999)	Oliver, D. E. Y., Shahar, M., Musen, A., & Shortliffe, E. H. (1999). *Representation of change in controlled medical terminologies*, AI in Medicine, 15(1):53–76.
(Palanque, 1992)	Palanque, P. (1992). *Modélisation par objets coopératifs interactifs d'interfaces homme-machine dirigées par l'utilisateur*. PhD Thesis, Université Toulouse I, France.
(Pandya, 2001)	Pandya, P. K. (2001). *Specifying and Deciding Qauntified Discrete-time Duration Calculus formulae using DCVALID,* in Proc. Real-Time Tools, RTTOOLS'2001, Aalborg, August, 2001 (affiliated with CONCUR 2001). Technical Report TCS-00-PKP-1, Tata Institute of Fundamental Research, Mumbai.
(Parreaux, 2000)	Parreaux, B. (2000). *Vérification de systèmes d'événements B par Model checking PLTL -Contribution à la réduction de l'explosion combinatoire en utilisant de la résolution de contraintes ensemblistes*. Thèse soutenu le 08 décembre 2000.
(Pérez et al., 2008)	Pérez, J., Arenas, M., et Gutierrez, C. (2008). nSPARQL: *A Navigational Language for RDF*. In ISWC, pages 66–81.
(Pérez et al., 2009)	Pérez, J., Arenas, M., et Gutierrez, C. (2009). *Semantics and complexity of SPARQL.* ACM Trans. Database Syst., 34 :16 :1–16 :45.
(Pérez et al., 2006)	Pérez, J., Arenas, M., Gutierrez, C. (2006). *The Semantics and Complexity of SPARQL*, Best Paper Award , 5th International Semantic Web Conference, ISWC 2006, Athens, Georgia, USA.
(Pérez et al., 2010)	Pérez, J., Arenas, M., Gutierrez, C. (2010). *nSPARQL: A Navigational Language for RDF Journal of Web Semantic*s 8(4):255-270.
(Pinto et al., 2004)	Pinto, H., Staab, S., Tempich, C. (2004). *Diligent: Towards a fine-grained methodology for distributed, loosely-controlled and evolving engineering of*

ontologies. In Proceedings of the 16th EuropeanConference on Artificial Intelligence (ECAI 2004), Valencia, Spain.

(Pinto et Martins, 2002) Pinto, S., et Martins, J. (2002). *Evolving Ontologies in Distributed and Dynamic Settings*. Paper presented at the 8th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR2002), San Francisco.

(Pittet et al., 2011) Pittet, P., Cruz, C., Nicolle, C. (2011). *Guidelines for a Dynamic Ontology - Integrating Tools of Evolution and Versioning in Ontology*. Proceedings of the International Conference on Knowledge Management and Information Sharing (KIMS 2011), Paris, France, 26-29 October, 2011: 173-179.

(Pittet et al., 2010) Pittet, P., Cruz, C., Nicolle, C. (2010). *Towards Dynamic Ontology - Integrating Tools of Evolution and Versionning in Ontology*. SEMAPRO 2010, The Fourth International Conference on Advances in Semantic Processing 2010.

(Plessers et De Troyer, 2005) Plessers, P, & De Troyer O. (2005). *Ontology change detection using a version log*, In Y.Gil, E. Motta, V.R. Benjamins, & M. Musen (Eds.), LNCS: Vol. 3729. The Semantic Web – ISWC 2005 (pp. 578-592). Berlin, Germany: Springer-Verlag. doi: 10.1007/11574620

(Plessers et De Troyer, 2006) Plessers, P., et De Troyer, O. (2006). *Resolving inconsistencies in evolving ontologies*. In Y. Sure, & J. Domingue (Eds.), LNCS: Vol.4011. The Semantic Web: Research and Ap-plications, Proceedings of the 3rd European Semantic Web Conference ESWC 2006 (pp. 200-214). Berlin, Germany: Springer.

(Plessers et al., 2007) Plessers, P., De Troyer, O., & Casteleyn, S. (2007). *Understanding ontology evolution: A change detection approach.* Journal of Web Semantics: Science, Services and Agents on the World Wide Web, Elsevier Publication, 5, 39-49.

(Polleres et al., 2007) Polleres, A., Scharffe, F. et Schindlauer, R. (2007). *SPARQL++ for Mapping Between RDF Vocabularies.* On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, pages 878–896.

(Pnueli, 1977) Pnueli, A. (1977). *The temporal logic of programs*. In proc. 18th IEEE Symp. Foundations of Computer Science (FOCS'77), Providence, RI, USA. pages 46-57.

(Presutti et al., 2008) Presutti, V., Gangemi, A. (2008). *Content Ontology Design Patterns as Practical Building Blocks for Web Ontologies*. In Proceedings of the 27th International Conference on Conceptual Modeling (ER 2008), Berlin, Springer.

(Prudhommeaux et Seaborne, 2005) Prudhommeaux, E. et Seaborne, A. (2005). *SPARQL Query Language for RDF*. http://www.w3.org/TR/rdf-sparql-query/

(Qi et al., 2006) Qi, G., Liu, W., et Bell, D. A. (2006). *A Revision-Based Approach for Handling Inconsistency in Description Logics*. In Proceedings of the 11th International Workshop on Non-Monotonic Reasoning (NMR-06).

(Queille et Sifakis, 1982) Queille, J. et Sifakis, J. (1982). *Specification and verification of concurrent systems in CESAR.* International Symposium on Programming In International Symposium on Programming, pp. 337-35.

(Rector et Roger, 2004) Rector, A., Rogers, J. (2004). *Patterns, properties and minimizing commitment: Reconstruction of the Galen upper ontology in OWL.* In A. Gangemi and S. Borgo (Eds.), EKAW'04 Workshop on Core Ontologies in Ontology Engineering. CEUR.

(Ricca et Tonella, 2001) Ricca F. et Tonella, P. (2001). *Analysis and Testing of Web Applications*, in Proc. International Conference on Software Engineering (ICSE2001), pp. 25-34.

(Ricca et Tonella, 2000) Ricca F. et Tonella, P. (2000). *Web Site Analysis: Structure and Evolution*, in Proc. International Conference on Software Maintenance (ICSM2000), pp. 76-86.

(Rogozan, 2008) Rogozan, D. C. (2008). *Gestion de l'évolution des ontologies : méthodes et outils pour un référencement sémantiques évolutif fondé sur une analyse des changements entre versions d'ontologie.* Rapport de thèse de recherche doctorale en informatique cognitive (DIC 9410). Télé-Université du Québec.

(Rosnay, 2009) Rosnay, J. (2009). *L'environnement cliquable : une virtualité bien réelle* - AgoraVox le média citoyen.

(Roussey, 2009) Roussey, C., Corcho, O., et Vilches-Blázquez, L. M. (2009*). A catalogue of OWL ontology antipatterns.*, K-CAP, USA. 205-206.

(Roynette, 2009) Roynette, B. (2009). *Le futur du web, des interfaces et des usages en 2019*. Vidéo You Tube.

(Samer et Veith, 2003) Samer, M., et Veith, H. (2003). *Validity of CTL Queries Revisited*. CSL 2003: 470-483.

(Sattler et al., 2003) Sattler, U., Calvanese, D. et Molitor, R. (2003). *Relationships with other formalisms.* Dans Baader, F., Calvanese, D., McGuinness, D., Nardi, D. et Patel-Schneider, P. (éditeurs), The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, pp. 142-183.

(Schlipf et al., 1997) Schlipf, T., Buechner, T., Fritz, R., Helms, M. et Koehl, J. (1997). *Formal verification made easy*. IBM Journal of Research and Development, 41(4–5):567–576.

(Schmidt, 2009) Schmidt, M. (2009). *Fondations of SPARQL query optimization*. PhD Thesis, Albert-Ludwigs-Universität Freiburg (Germany).

(Schnoebelen, et al., 1999) Schnoebelen, P., Bérard, B., Bidoit, M., Laroussinie, F., Petit, A. (1999). *Vérification de logiciels : Techniques et outils de model-checking*, Vuibert, Paris.

(Sciascio et al., 2002) Sciascio, E. D., Donini, M. F., Mongiello, M. et Piscitelli, G. (2002). *AnWeb: a System for Automatic Support to Web Application Verification,* in Proc. International Conference on Software Engineering and Knowledge Engineering (SEKE '02), pp. 609-616.

(Sciascio et al., 2003) Sciascio, E. D., Donini, M. F., Mongiello, M. et Piscitelli, G. (2003). *Web Applications Design and Maintenance using Symbolic Model checking*,"in Proc. Seventh European Conference on Software Maintenance and Reengineering (CSMR '03), pp. 63-72.

(Seaborne, 2004) Seaborne, A. (2004). RDQL - *À Query Language for RDF,* W3C Member Submission 9 January 2004. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/

(Sintek, et Decker, 2002) Sintek, M. et Decker, S. (2002). *TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web*. Proc. of the 1th ISWC.

(Sirin et Parsia, 2007) Sirin E. et Parsia, B. (2007). *SPARQL-DL: SPARQL query for OWL-DL*. In Proceedings of the 3rd OWL Experiences and Directions Workshop (OWLED 2007).

(Sirin et al., 2007) Sirin, E., Parsia, B., Cuenca Grau, B., Kalyanpur, A. et Katz, Y. (2007). *Pellet: A practical OWL DL reasoner*. Journal of Web Semantics, 5(2):51-53.

(Somenzi et Bloem, 2000) Somenzi, F. et Bloem, R. (2000). *Efficient Büchi automata for LTL formulae*. In Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00), volume 1855 of Lecture Notes in Computer Science, pages 247-263. Springer-Verlag.

(Stirling, 1992) Stirling, C. (1992). *Modal and Temporal Logic.* Dans Samson Abramsky, Dov M. Gabbay, et Thomas S. E. Maibaum, éditeurs, Handbook of Logic in Computer Science, volume 2, pages 477–563. Oxford University Press.

(Stojanovic, 2004) Stojanovic, L. (2004). *Methods and Tools for Ontology Evolution*. PhD thesis, University of Karlsruhe.

(Stojanovic et al., 2002) Stojanovic, L., Maedche, A., Motik, B., et Stojanovic, N. (2002). *User-driven ontology evolution management.* LNCS: Vol. 2473, Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web, Proceedings of the 13th International Conference EKAW2002 (pp. 285-300). Berlin, Germany: Springer-Verlag. doi: 10.1007/3-540-45810-7

(Storer, 1988) Storer, J. A. (1988). *Data Compression: Methods and Theory, Computer Science Press*, Rockville, MD.

(Sunagawa et al., 2003) Sunagawa, E., Kozaki, K., Kitamura, Y., et Mizoguchi, R. (2003). *An Environment for Distributed Ontology Development Based on Dependency Management*. Paper presented at the International Semantic Web Conference (ISWC), Florida, USA.

(Svatek, 2004) Svatek, V. (2004). *Design patterns for semantic web ontologies: Motivation and discussion*. 7ème conférence Business Information Systems, Poznan.

(Tarjan, 1972) Tarjan, R. E. (1972). *Depth-first search and linear graph algorithms.* SIAM Journal on Computing, vol. 1, no 2, p. 146–160.

(Tho et al., Tho T.Q et Thai D. N. (2008). *Ontology Evolution for Customer Services*. Journal of The Knowledge Representation Ontology Workshop (KROW 2008), Sydney,

2008)	Australia. Conferences in Research and Practice in Information Technology, Vol. 90.
(Trousse et al., 2008)	Trousse, B., Aufore, M. A., Le grand, B., Lechevakier, Y. et Masseglia, F. (2008). *Web Usage Mining for ontology management in Data mining Mining with ontologies imlementations*, *finding, and frameworks*, chapter 3, page 37-64.
(Tsarkov et Horrocks, 2003)	Tsarkov, D. et Horrocks, I. (2003). *DL reasoner vs. first-order prover*. Dans Proc. of the 2003 Description Logic Workshop (DL 2003) volume. pp. 152-159.
(Tsarkov et Horrocks, 2006)	Tsarkov, D. et Horrocks, I. (2006) *FaCT++ Description Logic Reasoner: System Description*. In: Proc. of the Int. Joint Conf. on Automated Reasoning IJCAR 2006, Vol. 4130 Springer, pp. 292--297.
(Vardi, 1996)	Vardi, M. Y. (1996). *An automata-theoretic approach to linear temporal logic*. In Faron Moller and Graham M. Birtwistle, editors, Proceedings of the 8th Banff Higher Order Workshop, volume 1043 of Lecture Notes in Computer Science, pages 238–266. Springer-Verlag.
(Vardi, 1986)	Vardi, M. Y. (1986). *An automata-theoretic approach to automatic program verication.* In Proceedings of the 1st IEEE Symposium on Logic in Computer Science (LICS'86), pages 332–344. IEEE Computer Society Press.
(Wolper et al., 1983)	Wolper, P., Vardi, M. Y. et Sistla, A. P. *Reasoning about infinite computation paths*. In Proceedings of the 24th IEEE Symposium on Foundations of Computer Science (FOCS'83), pages 185.194. IEEE Computer Society Press.
(Wolper, 2000)	Wolper, P. (2000). *Constructing automata from temporal logic formulas: A tutorial.* In Brinksma, E., Hermanns, H. et Katoen, J.-P., editors, Proceedings of the FMPA 2000 summer school, volume 2090 of Lecture Notes in Computer Science, pages 261–277, Nijmegen, the Netherlands. Springer-Verlag
(Yovine, 1997)	Yovine, S. (1997). *KRONOS: A Verification Tool for Real-Time Systems*, Journal of Software Tools for Technology Transfer, vol. 1, n° 1-2, p. 123-133, Springer.
(Yuen et al., 2005)	Yuen, S., Kato, K., Kato, D. et Agusa, K. (2005). *Web Automa A Behavioral Model of Web applications based on the MVC model*, Computer Software, vol. 22, no. 2, pp. 44-57.
(Zablith, 2009)	Zablith, F. (2009). *Evolva: A Comprehensive Approach to Ontology Evolution*, European Semantic Web Conference (ESWC) PhD Symposium, Crete, Greece, Journal of the 6th European Semantic Web Conference, LNCS 5554, eds. L. Aroyo et al., pp. 944-948, Springer-Verlag, Berlin, Heidelberg.
(Zeldman, 2006)	Zeldman, Jeffrey. (2006). *Web 3.0.*
(Zhang et Cleaveland, 2005)	Zhang, D. et Cleaveland, R. (2005). *Efficient temporal-logic query checking for presburger systems*. ASE 2005: 24-33.

Publications

Revues Internationales

(Gueffaz, et al., 2012a) Gueffaz, M., Rampacek, S. et Nicolle, C., Temporal Logic to Query Semantic Graphs Based on the NuSMV model checker, Journal of Software (JSW), Selected Best papers on IIT, 2012

(Gueffaz, et al., 2012b) Gueffaz, M., Rampacek, S. et Nicolle, C., Mapping SPARQL Query to Temporal Logic Query Based on NµSMV model checker to Query Semantic Graphs, International Journal of Digital Information and Wireless Communications (IJDIWC) 1, 2 (2012) 366-380, 2012

(Gueffaz, et al., 2011f) Gueffaz, M., Rampacek, S. et Nicolle, C., Verifying Semantic Graphs With the model checker SPIN, International Journal of Digital Information and Wireless Communications (IJDIWC) 1, 1 (2011) 64-74, 2011

Conférences internationales

(Gueffaz, et al., 2011c) Gueffaz, M., Rampacek, S. et Nicolle, C., Inconsistency Identification In Dynamic Ontologies Based On Model checking, Webist 2012- Proceedings of the the 8th International Conference on Web Information Systems and Technologies , Porto, Portugal, INSTICC, ACM SIGMIS, May 2011

(Gueffaz, et al., 2011a) Gueffaz, M., Rampacek, S. et Nicolle, C., RDF2NµSMV: Mapping Semantic Graphs to NµSMV model checker, Proceedings of the Third International Conference on Advances in Future Internet (AFIN 2011), Nice/Saint Laurent du Var, France, Août 2011

(Gueffaz, et al., 2011e) Gueffaz, M., Rampacek, S. et Nicolle, C., A New Approach Based on NµSMV Model to Query Semantic Graph, Proceedings of the International Conference on Digital Information Processing and Communications (ICDIPC 2011), Ostrava, République Tchèque, Springer-Verlag, Juillet 2011

(Gueffaz, et al., 2011b) Gueffaz, M., Rampacek, S. et Nicolle, C., RDF2SPIN: Mapping Semantic Graphs To Spin model checker, Proceedings of the International Conference on Digital Information and Communication Technology (DICTAP), Dijon, France, Springer-Verlag, 21 Juin 2011

(Gueffaz, et al., 2011c) Gueffaz, M., Rampacek, S. et Nicolle, C., Scalesem: Evaluation Of Semantic Graph Based On Model checking, Webist 2011- Proceedings of the 7th

International Conference on Web Information Systems and Technologies , Noordwijkerhout, Hollande, INSTICC, ACM SIGMIS, May 2011

(Gueffaz, et al., 2011d) Gueffaz, M., Rampacek, S. et Nicolle, C., Qualifying Semantic Graphs Using Model checking, Proceedings of the of the 7th International Conference on Innovations in Information Technology (Innovations'11), Abu Dhabi, Emirats arabes Unis, Sponsored by IEEE, April 2011

Séminaires

(Gueffaz, et al., 2011g) Gueffaz, M., Rampacek, S. et Nicolle, C., La logique temporelle pour interroger et qualifier des graphes sémantiques, Réseau Grand Est (RGE), Strasbourg, France, 13 octobre 2011

(Gueffaz, et al., 2011h) Gueffaz, M., Qualification et interrogation de graphes sémantiques à l'aide du Model checking, Assemblé général du laboratoire LE2I, Creusot (Centre universitaire du Condorcet), France, 1 juillet 2011

(Gueffaz, et al., 2011i) Gueffaz, M., Rampacek, S. et Nicolle, C., Qualification de graphes sémantiques à l'aide du Model checking, 17eme Forum des jeunes chercheurs (FJC 2011), Dijon, France, 16 et 17 Juin 2011

(Gueffaz, et al., 2010) Gueffaz, M., L'interrogation des graphes sémantiques à l'aide de la logique temporelle, Séminaire Université de Franche-Comté., Besançon, France, Juillet 2010

Dépôt APP

SPARQL2RLT: IDDN FR.001.240031.000.S.P.2012.000.10800. Outil de transformation de requêtes SPARQL en requête utilisant la logique temporelle.

STL-Resolver: IDDN FR.001.240032.000.SP.2012.000.10800. Outil de résolution des requêtes en logique temporelle.

RDF2SPIN : IDDN FR.001.240034.000.SP.2012.000.10800. Outil de transformation de graphes RDF en modèles de graphe SPIN.

RDF2NuSMV: IDDN FR.001.240040.000.S.P.2012.000.10800. Outil de transformation de graphes RDF en modèles de graphe NuSMV.

Rapports techniques

Gueffaz, M., *Introduction à la logique Temporelle*, Le2i - Université de Bourgogne, Dijon, France, Novembre 2009.

Gueffaz, M., *Introduction au Model-Checking*, Le2i - Université de Bourgogne, Dijon, France, Novembre 2009.

Gueffaz, M., *Introduction au Web sémantique*, Le2i - Université de Bourgogne, Dijon, France, Janvier 2010.

Gueffaz, M., *RDF et SPARQL*, Le2i - Université de Bourgogne, Dijon, France, Janvier 2010.

Gueffaz, M., *L'outil SPIN*, Le2i - Université de Bourgogne, Dijon, France, Février 2010.

Gueffaz, M., *L'outil NuSMV*, Le2i - Université de Bourgogne, Dijon, France, Février 2010.

Gueffaz, M., *Introduction aux graphes de données*, Le2i - Université de Bourgogne, Dijon, France, Février 2010.

Gueffaz, M., *Transformation de SPARQL vers LTL*, Le2i - Université de Bourgogne, Dijon, France, Avril 2010.

Gueffaz, M., *RDF2SPIN et RDF2NuSMV*, Le2i - Université de Bourgogne, Dijon, France, Mai 2010.

Gueffaz, M., *Rapport de doctorat 2010*, Le2i - Université de Bourgogne, Dijon, France, Juin 2011.

Gueffaz, M., *Requête en logique temporelle*, Le2i - Université de Bourgogne, Dijon, France, Septembre 2010.

Gueffaz, M., *La boite à outil ScaleSem*, Le2i - Université de Bourgogne, Dijon, France, Octobre 2010.

Gueffaz, M., *Transformation des requêtes SPARQL en requête utilisant la logique temporelle*, Le2i - Université de Bourgogne, Dijon, France, Octobre 2010.

Gueffaz, M., *L'outil STL Resolver*, Le2i - Université de Bourgogne, Dijon, France, Janvier 2011.

Gueffaz, M., *CADP vs ScaleSem*, Le2i - Université de Bourgogne, Dijon, France, Janvier 2011.

Gueffaz, M., *Etat de l'art sur les différents algorithmes de résolution Model checking*, Le2i - Université de Bourgogne, Dijon, France, Février 2011.

Gueffaz, M., *SPARQL vs Temporal Logique Query*, Le2i - Université de Bourgogne, Dijon, France, Juin 2011.

Gueffaz, M., *Rapport de doctorat 2011*, Le2i - Université de Bourgogne, Dijon, France, Juillet 2011.

Gueffaz, M., *La qualification sémantique dans le versioning d'ontologies*, Le2i - Université de Bourgogne, Dijon, France, Septembre 2011.

Gueffaz, M., *Détection d'inconsistance dans l'évolution d'ontologie*, Le2i - Université de Bourgogne, Dijon, France, Octobre 2011.

Gueffaz, M., *La méthodologie CLOCk*, Le2i - Université de Bourgogne, Dijon, France, Décembre 2011.

Gueffaz, M., *L'interface graphique ScaleSem*, Le2i - Université de Bourgogne, Dijon, France, Décembre 2011.

Gueffaz, M., *Nouvelle version de l'outil RDF2SPIN*, Le2i - Université de Bourgogne, Dijon, France, Janvier 2012.