Mise en contexte des traces pour une analyse en niveaux d'abstraction

Léon Constantin Fopa

▶ To cite this version:

Léon Constantin Fopa. Mise en contexte des traces pour une analyse en niveaux d'abstraction. Web. Université Grenoble Alpes; Université de Yaoundé I, 2015. Français. NNT : 2015GREAM077 . tel-01682808

HAL Id: tel-01682808
https://theses.hal.science/tel-01682808

Submitted on 12 Jan 2018

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

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

UNIVERSITE DE GRENOBLE
UNIVERSITE DE YAOUNDE 1

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

PRÉPARÉ DANS LE CADRE D'UNE COTUTELLE ENTRE L'UNIVERSITÉ DE GRENOBLE ET L'UNIVERSITÉ DE YAOUNDÉ 1.

Spécialité : **Informatique**

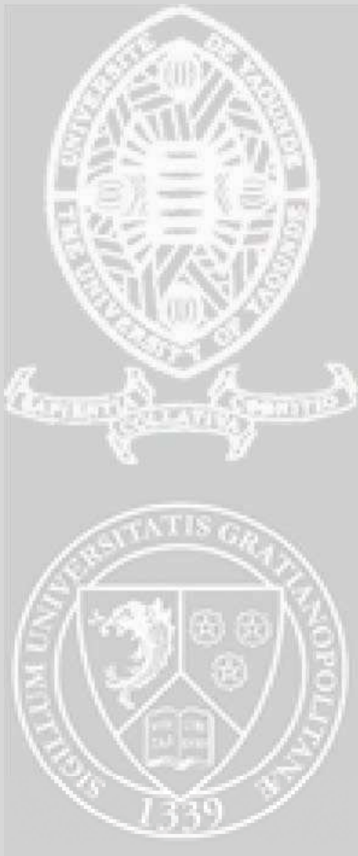
Arrêté ministériel : 6 janvier 2005 - 7 août 2006

Présentée par

Fopa Léon Constantin

Thèse dirigée par **Jean François Méhaut et Maurice Tchuenté**
et codirigée par **Fabrice Jouanot et Alexandre Termier**

préparée au sein **du LIG et du LIRIMA**
et de **l'Ecole Doctorale Mathématique, Sciences et Technologies de l'Information, Informatique (MSTII) et du Centre de Recherche et de Formation Doctorale en Sciences, Technologies et Géosciences (CRD/STG)**



Mise en contexte des traces d'exécution pour une analyse en niveaux d'abstraction

"Exploiting context for an structuration of execution traces in abstraction layers"

Thèse soutenue publiquement le **23 juin 2015**,
devant le jury composé de :

Mme, Marie-Odile Cordier
Professeur émérite à l'Université de Rennes, Présidente
Mr, François Goasdoué
Professeur à l'Université de Rennes, Rapporteur
Mr, Pierre Boulet
Professeur à l'Université de Lille, Rapporteur
Mme, Nadine Cullot
Professeur à l'Université de Bourgogne, Examinatrice
Mr, Jean-François Méhaut
Professeur à l'Université Joseph Fourier, Directeur de thèse
Mr, Maurice Tchuenté
Professeur à l'Université de Yaoundé 1, Co-Directeur de thèse
Mr, Fabrice Jouanot
Maître de conférence à l'Université Joseph Fourier, Encadrant
Mr, Alexandre Termier
Professeur à l'Université de Rennes, Co-Encadrant

Résumé

Les techniques d'analyse et de débogage d'applications sont de plus en plus mises à mal dans les systèmes modernes. En particulier dans les systèmes basés sur des composants embarqués multiprocesseurs (ou MPSoc) qui composent aujourd'hui la plupart de nos dispositifs quotidiens. Le recours à des traces d'exécution devient incontournable pour appliquer une analyse fine de tels systèmes et d'en identifier les comportements divergents. Même si la trace représente une source d'information riche mise à disposition du développeur pour travailler, les informations pertinentes à l'analyse se retrouvent noyées dans la masse et sont difficilement utilisables sans une expertise de haut niveau. Des outils dédiés à l'exploitation des traces deviennent nécessaires. Cependant les outils existants prennent rarement en compte les aspects métiers spécifiques à l'application afin d'optimiser le travail d'analyse.

Dans cette thèse, nous proposons une approche qui permet au développeur de représenter, manipuler et interroger une trace d'exécution en se basant sur des concepts liés à ses propres connaissances métier. Notre approche consiste en l'utilisation d'une ontologie pour modéliser et interroger les concepts métier dans une trace, et l'utilisation d'un moteur d'inférence pour raisonner sur ces concepts métier. Concrètement, nous proposons VIDECOM l'ontologie du domaine de l'analyse des traces d'exécution des applications embarquées multimédia sur MPSoC. Nous nous intéressons ensuite au passage à l'échelle de l'exploitation de cette ontologie pour l'analyse des traces de grandes tailles. Ainsi, nous faisons une étude comparative des différents systèmes de gestion des ontologies pour déterminer l'architecture la plus adaptée aux traces de très grande taille au sein de notre ontologie VIDECOM. Nous proposons également un moteur d'inférence qui adresse les défis que pose le raisonnement sur les concepts métier, à savoir l'inférence de l'ordre temporel entre les concepts métier dans la trace et la terminaison du processus de génération de nouvelles connaissances métier.

Enfin, nous illustrons la mise en pratique de l'utilisation de l'ontologie VIDECOM dans le cadre du projet SoC-Trace pour l'analyse des traces d'exécution réelles sur MPSoC.

Mots clés: Analyse de traces, Web Sémantique, Inférence, Ontologie, RDF.

Abstract

Applications analysis and debugging techniques are increasingly challenging task in modern systems. Especially in systems based on embedded multiprocessor components (or MPSoC) that make up the majority of our daily devices today. The use of execution traces is unavoidable to apply a detailed analysis of such systems and identify unexpected behaviors. Even if the trace offers a rich corpus of information to the developer for her work, information relevant to the analysis are hidden in the trace and is unusable without a high level of expertise. Tools dedicated to trace analysis become necessary. However existing tools take little or no account of the specific business aspects to an application or the developer's business knowledge to optimize the analysis task.

In this thesis, we propose an approach that allows the developer to represent, manipulate and query an execution trace based on concepts related to her own business knowledge. Our approach is the use of an ontology to model and query business concepts in a trace, and the use of an inference engine to reason about these business concepts. Specifically, we propose VIDECOM, the domain ontology for the analysis of execution traces of multimedia applications embedded on MPSoC.

We then focus on scaling the operation of this ontology for the analysis of huge traces. Thus, we make a comparative study of different ontologies management systems (or triplestores) to determine the most appropriate architecture for very large traces in our VIDECOM ontology. We also propose an inference engine that addresses the challenges of reasoning about business concepts, namely the inference of the temporal order between business concepts in the trace and the termination of the process of generating new knowledge from business knowledge.

Finally, we illustrate the practical use of VIDECOM in the SoC-Trace project for the analysis of real execution traces on MPSoC.

Keywords: Trace analysis, Semantic Web, Inference, Ontology, RDF.

Remerciements

J'aimerai exprimer ma gratitude à toutes les personnes qui ont contribué de près ou de loin à l'aboutissement de ce travail.

Je remercie tout d'abord mes directeurs de thèses, le Professeur Jean-françois Méhaut et le Professeur Maurice Tchuenté qui m'ont offert cette opportunité de faire une thèse dans un environnement de cotutelle riche et stimulant.

Mes remerciements vont également à l'endroit de mes encadreurs Alexandre Termier et Fabrice Jouanot qui m'ont patiemment appris la pratique et la rigueur de la recherche. Votre grande disponibilité m'a énormément aidé à recadrer mes efforts et à simplifier mes idées *révolutionnaires*. Je remercie également le Professeur Marie-Christine Rousset pour sa disponibilité lors de nos échanges informels et pour ses précieux commentaires lors de la rédaction de ce manuscrit. Je remercie tous les membres des équipes *HADAS*, *SLIDE* à Grenoble et les membres de l'équipe *LIRIMA* à Yaoundé. Je vous remercie pour nos nombreux échanges (souvent à distance).

J'aimerai également remercier la forte communauté des camerounais de l'Isère pour son soutien tout au long de ma thèse. Je pense à Eric, Thomas, Boddy, Samuel, Léonie, Alain, Juliette, David, et plus particulièrement au *mimbang du mimbang* composé de Jean-pierre, Orléant Njamen et Fred Fabo (mes *mbrah*). Je ne manquerai pas de remercier Serge Emteu et Christiane Kamdem pour la grande expérience que nous avons partagé à Grenoble, sans oublier Eric Michel Fotsing pour l'exigence de qualité du travail qu'il a toujours démontré dans chacun de nos échanges.

Je remercie les grandes familles *Chipowo*, *Tchinda*, *Tadjukem* et *Ngouanat*. Ainsi que toi, maman pour ta persévérance et ta patience qui me guident chaque jour.

Je termine en remerciant mon épouse Kevine ainsi que Ruth Joanna et Jonathan pour la place qu'ils occupent dans mon coeur et la volonté qu'ils me donnent de faire toujours mieux.

Il n'est pas nécessaire d'espérer pour entreprendre
ni de reussir pour persévérer.
– Paul B. Biya

A Chipowo Roger, mon très cher papa

Sommaire

1 Introduction **17**
1.1 Le contexte . . . 17
1.2 La problématique . . . 19
1.3 L'objectif et la contribution . . . 20
1.4 Plan de la thèse . . . 21

2 L'analyse des traces d'exécution des applications embarquées multimédia **23**
2.1 Les traces d'exécution . . . 24
2.2 L'état de l'art de l'analyse *post-mortem* des traces . . . 32
2.3 Conclusion . . . 39

3 VIDECOM, une ontologie pour l'analyse des traces d'applications multimédia sur MPSoC **41**
3.1 L'ontologie pour la représentation des connaissances . . . 43
3.2 La construction d'une ontologie pour l'analyse des traces d'exécution 43
3.3 Les langages de construction des ontologies . . . 45
3.4 L'ontologie VIDECOM . . . 48
3.5 L'interrogation d'une ontologie . . . 58
3.6 Illustration de l'interrogation de VIDECOM pour l'analyse de traces . . . 61
3.7 Conclusion . . . 62

4 Le benchmark des performances de chargement et d'interrogation de VIDECOM **65**
4.1 Le stockage des données *RDF* . . . 67
4.2 Passage à l'échelle pour l'interrogation de VIDECOM . . . 79
4.3 Conclusion . . . 93

5 L'inférence des concepts métier dans VIDECOM **95**
5.1 L'inférence des connaissances pour répondre à des requêtes . . . 97
5.2 La saturation des règles métier . . . 100
5.3 Proposition d'un moteur d'inférence pour les règles métier . . . 103
5.4 Les expérimentations . . . 111
5.5 La saturation à grande échelle . . . 118

5.6 Conclusion 122

6 Intégration de VIDECOM dans le projet SoC-Trace 123
6.1 Contexte académique et industriel de la thèse 123
6.2 L'infrastructure *Framesoc* 124
6.3 Cas d'utilisation de VIDECOM dans le SET 126
6.4 Démonstration de l'utilisation de VIDECOM sur *Framesoc* . . . 127

7 Conclusion 135
7.1 La contribution 135
7.2 Les perspectives 136

A Appendice 141
A.1 Liste des réquêtes d'insertion pour les règles d'inférence RDFS . 141
A.2 Requêtes SQL équivalentes pour le partitionnement vertical . . . 143

Listes des figures

1.1 Les principaux domaines d'application des systèmes embarqués . 18

2.1 Illustration de la dépendance ascendante des niveaux d'abstraction à partir des évènements de type "instructions processeur" d'une trace 26
2.2 Exemple de l'exécution de 4 canaux audio/vidéo d'une application de décodage vidéo dans un simulateur MPSoC de 16 processeurs 27
2.3 Capture d'écran de l'outil Kptrace viewer. 33
2.4 Visualisation d'une trace Kptrace avec Ocelotl 34

3.1 Extrait des classes et propriétés de VIDECOM pour l'analyse de la trace exemple 50

4.1 Représentation des triplets illustratives par trois graphes optimisés (Jena2) 69
4.2 La matrice de bits *SP* et le dictionnaire des objets pour représenter les triplets illustratifs (BitMat) 70
4.3 Graphe bipartie pour répresenter les triplets illustratif ayant `event1` comme sujet (Sesame) 71
4.4 Arbre B^+ représentant l'index *spo* pour l'accès optimisé aux triplets de la table 4.1 à partir du sujet et de la propriété 73
4.5 illustration des modèles physiques de données column-store et row-store 76
4.6 Temps de chargement des traces dans les triplestores 81
4.7 Débit de chargement des traces dans les triplestores du benchmark 82
4.8 Temps d'exécution des requêtes du scénario de description des données 84
4.9 Temps d'exécution des requêtes du scénario de selection temporelle des données 86
4.10 Temps d'exécution des requêtes du scénario d'accès aux séquences d'évènements 87
4.11 Temps d'exécution des requêtes de tri et de limitation des données 89
4.12 Temps d'exécution des requête d'agrégation de données 91
4.13 Temps d'exécution des requêtes d'exploration de l'ontologie . . . 92

5.1 Evolution des temps de saturation 115
5.2 Temps de réponse des moteurs d'inférence aux requêtes *ScenarioQ1* et *ScenarioQ2* avant et après la saturation 117
5.3 Temps de saturation des traces 121

6.1 Architecture de l'infrastructure de trace (Framesoc) et les plugins 125
6.2 Schéma de fonctionnement de l'application ts_record 126
6.3 Interface du Plugin VIDECOM pour le chargement des évènenemts de la trace depuis Framesoc 129
6.4 Interface web de saturation des traces dans VIDECOM 130
6.5 Interface web d'interrogation des traces dans VIDECOM 131
6.6 Visualisation des concepts métier (anomalies et fonctionnalité de ts_record) dans VIDECOM 132
6.7 Visualisation des concepts métier (anomalies et fonctionnalités de ts_record) dans Framesoc et le plugin Ocelotl 133

Listes des tables

2.1 Extrait de trace matérielle 28
2.2 Extrait de trace logicielle *Kptrace* 28
2.3 Extrait de trace logicielle *Lttng* 29
2.4 Extrait de trace logicielle *GStreamer* 29
2.5 Détails des traces issues de 30 secondes de collecte sur différents outils 32
2.6 Exemple des deux occurrences (en bleu) dans la trace de la séquence fréquente résultat $\langle exit_syscall, sys_poll \rangle$ retrouvé par *Profspan* avec une fréquence de 2. 36
2.7 Example de réécriture de trace par Frameminer 37
2.8 Algorithme *Sequitur* appliqué à la trace 38

3.1 Trace exemple. 42
3.2 Liste des contraintes métier pour l'analyse de la trace exemple . 45
3.3 Liste des noms de domaines standards 46
3.4 Ensemble des triplets initiaux produits par la trace exemple . . . 52
3.5 Règles d'inférence de la précédence entre les évènements 54
3.6 Règles d'inférence des quatre propriétés RDFS 56

4.1 Base de triplets *RDF* illustratifs 67
4.2 Requêtes SPARQL illustratives 68
4.3 Partitionnement vertical des triplets de la Table 4.1 76
4.4 Liste des triplestores présentés dans la section 78
4.5 Débits et temps de chargement des triplets et de la trace *T3* . . 83
4.6 Récapitulatif des performances des triplestores sur la trace T3 . . 92

5.1 Triplets pour l'illustration de l'inférence des concepts métier . . . 96
5.2 Résultats de la comparaison des approches TripleTable et verticalPartitioning pour la saturation 120
5.3 Temps de saturation de la trace T3 par *monetdbIE-tt* 121

Chapitre 1

Introduction

Les systèmes qui nous entourent sont de plus en plus complexes. Les traces, d'origine humaine ou système, sont développées comme des outils pour améliorer, comprendre ou prédire le fonctionnement de ces systèmes. Nos travaux se situent dans le domaine de l'analyse des traces d'exécution issues des systèmes embarqués multiprocesseur ou MPSoC.

1.1 Le contexte

Un MPSoC (Multi Processor System-on-Chip) est une petite puce sur laquelle sont gravés plusieurs composants électroniques tels que les processeurs, les mémoires, les bus, les unités de traitement graphiques (GPU) et les ports d'entrée/sortie. Comme l'illustre la Figure 1.1, ces systèmes embarqués sont largement utilisés dans des domaines d'application très variés de notre quotidien tels que le divertissement, la sécurité, la domotique, l'industrie, l'automobile, le médical et les télécommunications. On les retrouve par exemple dans des équipements pour le grand public tels que les smartphones, les set-top box, les systèmes airbag mais aussi dans les automates industriels et les capteurs divers tels que les détecteurs de mouvement et de pression, les capteurs de lumière ou de fumée.

1.1.1 Le développement des applications embarquées

Le développement des applications embarquées sur les MPSoC doit garantir que ces applications s'exécutent correctement sur l'architecture du MPSoC.

Plusieurs aspects rendent difficile l'atteinte de cet objectif par le développeur:

1. Les MPSoC sont généralement de petite taille et sont par conséquent très limités en ressources. Ils ne disposent pas, pour la plupart, d'écran ni de clavier pour faciliter le développement. Ces ressources sont contraignantes

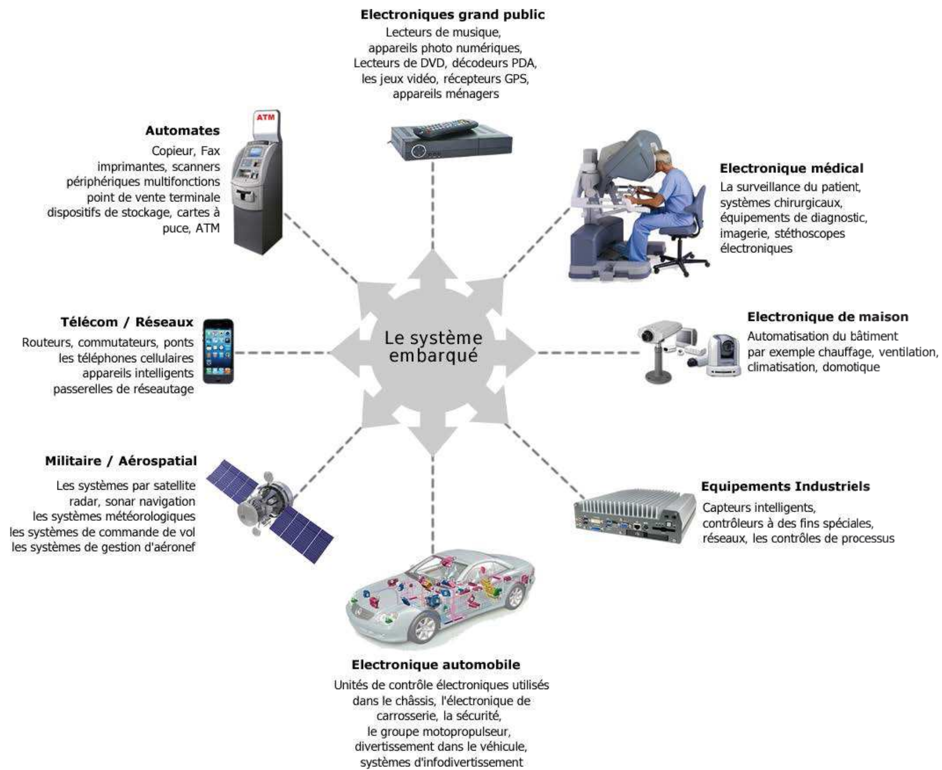


Figure 1.1: Les principaux domaines d'application des systèmes embarqués

pour le développeur. Elles limitent, par exemple, l'utilisation des techniques de développement déjà éprouvées pour les applications destinées aux ordinateurs personnels.

2. Les applications embarquées ont généralement de fortes contraintes de fiabilité et de durabilité. Il est, par exemple, naturel d'attendre de l'application d'appel sur le smartphone ou d'une application de télésurveillance, qu'elle fonctionne correctement 24 heures/24 et 7 jours/7. Ces contraintes obligent le développeur à faire des tests rigoureux dans des conditions identiques (sinon proches) à l'environnement où l'application sera utilisée.

3. Certaines applications embarquées sont dites *temps réelles*. C'est à dire qu'elles doivent respecter des contraintes temporelles spécifiques en plus de retourner des réponses correctes. Ces contraintes temporelles sont critiques et leurs violations ont généralement un impact direct sur le fonctionnement de l'application. Dans le cas d'une application de décodage audio/vidéo, par exemple, le non respect des contraintes temporelles peut entrainer diverses malfonctions telles que des images détériorées ou une désynchronisation entre les images et le son. La vérification du respect de ces contraintes est difficile à cause de la complexité inhérente aux architectures paralléles des MPSoC.

4. Enfin, les systèmes embarqués représentent un marché prospère et très concurrentiel. Le temps de mise sur le marché d'une nouvelle version d'application ou d'une nouvelle architecture, est généralement court. Par exemple, pour rester compétitif, les constructeurs de smartphones mettent un nouveau produit sur le marché tous les 6 à 10 mois. En conséquence, le développement d'une application embarquée ou la correction d'une application déjà en utilisation, doit se faire dans des délais très courts.

Ces aspects ont une influence sur le développement des systèmes embarqués. Selon *Thomas Corbi*, une grande partie du temps de développement d'une application (jusqu'à 60 %) est passée dans la compréhension du fonctionnement de cette application [Cor89]. Cette compréhension sert généralement à identifier et à corriger les malfonctions (*débogage*) et/ou analyser les performances de l'application (*profilage*).

1.1.2 L'analyse de traces d'exécution

Dans le cas des applications embarquées, plus précisement dans le cas des applications multimédia qui sont très répandues dans les smartphones et les set-top box, l'identification et la correction des malfonctions et des contre-performances en se basant uniquement sur les codes sources des applications est difficile. En effet, certaines de ces malfonctions (telles que les problèmes liés aux erreurs de timings) et contre-performance trouvent leur origine et ne peuvent être identifiées que lors de l'exécution de l'application. En conséquence, les techniques de débogage telles que l'exécution pas-à-pas ne sont pas applicables pour ces applications car elles sont intrusives et font disparaitre les problèmes qu'elles sont sensées résoudre.

Ainsi, pour identifier, comprendre et résoudre ces types de malfonctions et contre-performances, les développeurs ont recours à la capture et à l'analyse post-mortem d'une trace d'exécution qui contient des informations collectées pendant l'exécution de l'application sur le système embarqué.

Andreas Zeller décrit le débogage comme un processus itératif où à chaque itération le développeur se pose des questions, formule des hypothèses et fait de multiples observations pour vérifier ses hypothèses [Zel09]. Le processus s'arrête lorsque la vérification des hypothèses formulées sur l'apparition d'une malfonctions ou d'une contre-performance permettent de résoudre le problème qui se posait dans l'application. Dans le cas du débogage par l'analyse des traces, la formulation des hypothèses et leur vérification dépendent de la pertinence des informations sur l'exécution de l'application contenues dans la trace d'exécution. Cette pertinence est fonction des différentes vues (ou niveaux d'abstraction) qu'offrent ces informations, des moyens mis à disposition du développeur pour exploiter ces vues et du problème cible qu'il veut résoudre.

1.2 La problématique

Considérons par exemple une malfonction de désynchronisation entre le son et l'image dans une application de téléconférence. Si le développeur fait l'hypothèse

que la qualité du débit internet est la cause de cette malfonction, alors il s'intéressera aux informations (ou niveaux d'abstraction) liées à l'activité du réseau pendant l'exécution de l'application. S'il fait une seconde hypothèse que l'ordonnancement des tâches parallèles est la cause de cette malfonction, alors il changera de niveau d'abstraction pour s'intéresser aux informations liées à l'activité des tâches d'exécution produites pendant l'exécution de l'application.

L'exploitation des niveaux d'abstraction dans les traces d'exécution des applications embarquées multimédia pose quelques difficultés:

1. **Hétérogéneité des niveaux d'abstraction**. Les informations collectées dans la trace sont généralement à des niveaux d'abstraction prédéfinis. Cependant, en fonction du problème à resoudre, ces niveaux peuvent ne pas être pertinents. En conséquence, le développeur doit se baser sur ses propres connaissances métier pour changer de niveau d'abstraction.

2. **La représentation des niveaux d'abstraction dans la trace**.

   Le développeur ne connait généralement pas les causes du problème à résoudre au moment de la collecte des traces d'exécution. Ainsi, certains niveaux d'abstraction pertinents peuvent être absents de la trace. Cette absence peut également être due au fait que le problème à résoudre est difficilement reproductible (car il se produit dans des conditions d'exécution atypiques créées par l'utilisateur) ou que l'outil de collecte de traces n'est pas capable de capturer ce niveau d'abstraction. En conséquence, le développeur doit exploiter ses propres connaissances métier pour raisonner en parallèle sur différents niveaux d'abstraction afin de déduire des informations pertinentes.

3. **L'interrogation et le raisonnement sur les traces**. Les outils d'analyse de traces basés sur des approches de visualisation, de fouille de données ou de comparaison de traces, ne prennent pas (ou très peu) en compte les connaissances métier du développeur. En conséquence le développeur peut difficilement exploiter les niveaux d'abstraction exposés par ces outils pour, par exemple, construire de nouveaux niveaux d'abstraction pertinents.

4. **La taille des traces**. Quelques minutes d'exécution d'une application embarquée peuvent générer des traces de très grandes tailles. Par exemple, 30 secondes d'exécution d'une application de décodage vidéo peut produire jusqu'à 8 Gigaoctets sur un MPSoC à 16 coeurs. Cette taille pose des problèmes de passage à l'échelle pendant l'analyse de la trace.

1.3 L'objectif et la contribution

1.3.1 L'objectif

Notre objectif est de proposer une approche qui permette au développeur de représenter, manipuler et interroger une trace d'exécution en se basant sur des concepts liés à ses propres connaissances métier. Notre approche est basée sur:

- l'utilisation d'une ontologie RDF pour modéliser les concepts métiers liés aux informations contenues dans les traces d'exécution et aux connaissances métier du développeur,
- l'utilisation d'un moteur d'inférence pour raisonner sur les concepts métier et construire les niveaux d'abstraction pertinents,
- l'utilisation du langage déclaratif *SPARQL* pour interroger la trace en utilisant les concepts métier.

1.3.2 La contribution

Notre contribution est la suivante:

1. Nous proposons l'ontologie VIDECOM pour l'analyse des traces d'exécution des applications embarquées multimédia sur MPSoC.
2. Nous faisons une étude comparative des performances de l'utilisation de VIDECOM sur plusieurs systèmes de gestion des ontologies (ou *triplestores*). Cette étude sert à déterminer les caractéristiques d'un triplestore adapté à l'analyse des traces de grande taille via notre ontologie.
3. Nous proposons un moteur d'inférence, basé sur la saturation, qui s'adapte aux défis que pose la matérialisation des concepts métier dans la trace, à savoir, l'inférence de l'ordre temporel entre les concepts métier et la terminaison du raisonnement lors de l'inférence des concepts métier dans la trace.

1.4 Plan de la thèse

Le reste du document est organisé comme suit:

- Le **chapitre 2** introduit la notion de traces d'exécution et précise l'objectif de l'analyse de traces dans le contexte des systèmes embarqués. La première partie du chapitre présente les types des informations contenues dans les traces et les différentes techniques de collecte de ces informations. La seconde partie du chapitre présente l'état de l'art des différentes méthodes d'analyse de traces ainsi que les défis actuels auxquels font face les concepteurs de ces méthodes.
- Le **chapitre 3** présente notre approche d'analyse de traces en exploitant les connaissances métier modélisées dans une ontologie. Ce chapitre présente les fondements théoriques sur lesquels reposent les ontologies, ainsi que l'interrogation et le raisonnement sur les connaissances qu'elles formalisent. La seconde partie du chapitre présente VIDECOM, notre ontologie pour l'analyse des traces des applications multimédia.

- Le **chapitre 4** s'intéresse au passage à l'échelle de l'interrogation, du stockage et du chargement des traces dans VIDECOM. Ce chapitre présente l'étude comparative des performances de 12 *triplestores* pour le stockage et l'exploitation de VIDECOM sur des traces de grandes tailles.

- Le **chapitre 5** introduit les approches de raisonnement utilisées par les différents moteurs d'inférence de règles. Ce chapitre présente également les problèmes que pose l'inférence des connaissances métier et le modèle que nous proposons pour adresser ces problèmes. Ce chapitre propose enfin une étude comparative des performances de notre moteur en comparaison aux performances des moteurs d'inférence de l'état de l'art.

- Le **chapitre 6** présente l'intégration de *VIDECOM* dans l'infrastrure de gestion construit dans le cadre du projet FUI *SoC-Trace*. Ce chapitre présente le projet *SoC-Trace* et situe notre contribution dans le projet. La dernière partie du chapitre est la présentation d'une démonstration de l'utilisation de l'ontologie VIDECOM dans un cas réel d'analyse d'une application embarquée multimédia.

- Le **chapitre 7** conclut ce document et présente les perspectives pour améliorer le passage à l'échelle de l'interrogation et du raisonnement sur VIDECOM lorsqu'il est appliqué à des volumes de traces de plus en plus importants.

Chapitre 2

L'analyse des traces d'exécution des applications embarquées multimédia

Sommaire

2.1 Les traces d'exécution 24
2.1.1 La collecte des traces 24
2.1.2 Les niveaux d'abstraction dans les traces 25
2.1.3 Quelques exemples de traces d'exécution 26
2.1.4 La taille des traces 31
2.2 L'état de l'art de l'analyse *post-mortem* des traces 32
2.2.1 L'analyse de la trace à l'aide de la visualisation . . . 32
2.2.2 L'analyse des traces à l'aide de la fouille de données 35
2.2.3 L'analyse par comparaison de traces 38
2.3 Conclusion 39

L'objectif de ce chapitre est de présenter les méthodes d'analyse de traces d'exécution des applications multimédia embarquées. Nous allons commencer par définir ce que sont les traces d'exécution, nous allons illustrer comment ces traces sont produites et comment elles contribuent à l'analyse de l'application notamment à travers les niveaux d'abstraction (Section 2.1). Nous allons ensuite présenter les différentes méthodes de l'état de l'art pour l'analyse post-mortem de traces d'execution (Section 2.2). Enfin, nous allons terminer ce chapitre en insistant sur les limites que rencontrent ces méthodes d'analyse de traces dans la gestion des niveaux d'abstraction dans la trace (Section 2.3).

2.1 Les traces d'exécution

Les traces d'exécution sont constituées d'évènements qui contiennent des informations sur l'exécution d'une application [LB94]. La première étape pour l'analyse d'une trace est la collecte des évènements pendant l'exécution de l'application. La prochaine section porte sur la collecte des traces, elle présente les différentes techniques de collecte des évènements et quelques exemples d'évènements de traces.

2.1.1 La collecte des traces

Il existe des techniques matérielles et des techniques logicielles pour collecter les évènements contenus dans une trace d'exécution.

Les techniques matérielles

Ces techniques collectent les informations à partir de mesures qui sont directement faites sur le matériel. On peut citer comme exemple l'outil *Trace32* qui collecte des évènements produits par les écritures et les lectures dans la mémoire et les registres, ainsi que ceux produits par l'exécution pas-à-pas et temps-réel des instructions [dt]. *Cunha et al* propose un simulateur de MPSoC qui permet de collecter des évènements de traces correspondant aux instructions processeurs [CFP15]. Les traces sont produites en continu par les systèmes embarquées qui disposent d'un port spécial prévu pour les collecter. On peut citer le port *BDM* pour les processeurs *Motorola*, et le port *JTAG* pour les processeurs *ARM7* et ceux de la famille *PowerPC*.

Les techniques matérielles de collecte des traces ont l'avantage d'être peu intrusives (pas plus de 5% pour les récents systèmes embarqués). C'est-à-dire qu'elles influencent très peu l'exécution de l'application. Cela est critique par exemple dans le cas des applications de vidéoconférence, de télésurveillance ou de décodage audio/vidéo sur les set-top box.

Etant donné que les informations mesurées sur le matériel représentent un indicateur de ses performances, ce type de trace est utilisé pour l'optimisation des architectures matérielles et des compilateurs. Toutefois, à cause de la fine granularité des évènements collectés par cette technique, les techniques logicielles de collecte sont préférées dans les cas où le développeur veut avoir des vues synthétiques sur l'exécution de l'application.

Les techniques logicielles

Ces techniques consistent en l'instrumentation de l'application, c'est-à-dire à l'ajout (dans le code source ou dans le code binaire de l'application) de segments de codes spéciaux pour calculer et/ou collecter des informations diverses pendant l'exécution de l'application [LB94].

Cette technique permet ainsi de collecter plusieurs types de vues sur l'exécution de l'application. Par exemple, les outils de collecte tels que *Valgrind* permettent de collecter des informations liées à l'utilisation de la mémoire [Dev]. Tandis que d'autres outils, tels que *Pin* [Ber], permettent de collecter des informations en rapport avec l'exécution des instructions de l'application.

Cette technique sert également à collecter des informations sur l'activité du système d'exploitation pendant l'exécution. Dans ce cas on peut citer les *kprobes* qui servent à l'instrumentation des systèmes d'exploitation tels que *Linux* et *STLinux*. Les outils de collecte de traces *Lttng* (pour linux) et *Kptrace* (pour *STLinux*) exploitent ces *kprobes* pour collecter les appels systèmes, les interruptions, les appels de fonctions et les changements de contexte entre les différentes tâches qui surviennent pendant l'exécution de l'application.

De même, certains frameworks de développement d'applications proposent des techniques logicielles de collecte des évènements. On peut citer comme exemple *GStreamer* qui est un framework de développement des applications multimédia [GSt]. Les évènements collectés par *GStreamer* correspondent aux activités de ses différents composants tels que les décodeurs vidéo et audio, ou les démultiplexeurs de flux multimédia.

Contrairement aux techniques matérielles, les techniques logicielles de collecte de traces peuvent sensiblement modifier le comportement de l'application. Toutefois, elles permettent de collecter des informations plus synthétiques sur l'exécution. Les traces ainsi collectées sont très utiles pour le profilage des applications.

Dans les deux prochaines sections, nous allons respectivement définir la notion de niveau d'abstraction dans les traces d'exécution et illustrer cette notion dans quelques exemple de traces d'exécution.

2.1.2 Les niveaux d'abstraction dans les traces

Les évènements d'une trace d'exécution apportent diverses informations sur l'exécution de l'application. Un niveau d'abstraction est une vue constituée d'un ensemble de ces évènements qui portent un intérêt particulier pour le développeur.

Théoriquement, on peut imaginer qu'une trace contient les évènements à la plus fine granularité possible, correspondant par exemple aux instructions du processeur à chaque cycle d'horloge. Ces évènements, bas niveaux, sont regroupés pour constituer des informations plus "intelligibles" selon l'intérêt du développeur.

Considérons par exemple que la trace est issue de l'exécution d'une application de décodage vidéo. Ainsi, comme le montre la figure 2.1, plusieurs de ces instructions processeur sont assemblées pour former des instructions et plusieurs de ces instructions sont regroupées pour former des appels systèmes, ensuite plusieurs appels système sont regroupés pour former une lecture des données et plusieurs lectures des données sont regroupées pour former le décodage d'une image.

Chaque groupe constitue un *niveau d'abstraction*. Certains niveaux d'abstraction sont directement disponibles dans la trace grâce aux outils de collecte de traces d'exécution. On peut citer le niveau *matériel*, *système* et le niveau *applicatif* comme des exemples de tel niveau d'abstraction.

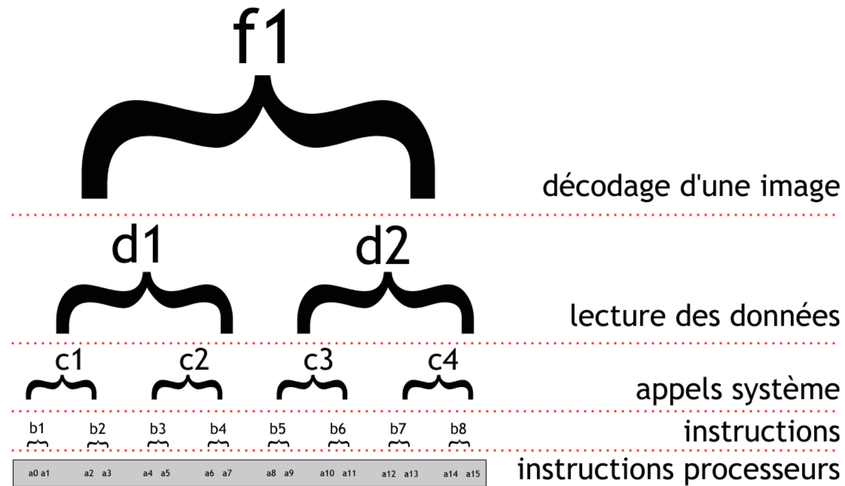


Figure 2.1: Illustration de la dépendance ascendante des niveaux d'abstraction à partir des évènements de type "instructions processeur" d'une trace

Le niveau *matériel* est exprimé par des évènements liés au matériel tels que les instructions processeurs. Le niveau *système* est exprimé par des évènements liés au fonctionnement du système d'exploitation tels que les interruptions, les appels de fonctions, les appels systèmes et les changements de contexte entre les tâches d'exécution. Enfin, le niveau *applicatif* est exprimé par des évènements liés à l'application, tels que les appels de fonctions. Dans la prochaine section nous allons illustrer ces niveaux d'abstraction dans des exemples de traces d'exécution.

2.1.3 Quelques exemples de traces d'exécution

Le niveau matériel

La trace de la Table 2.1 contient des évènements extraits d'une application de décodage vidéo exécutée dans un simulateur de MPSoC [CFP15] à 16 processeurs tous connectés par un réseau ou *NoC* (*Network-on-Chip*) (Figure 2.2).

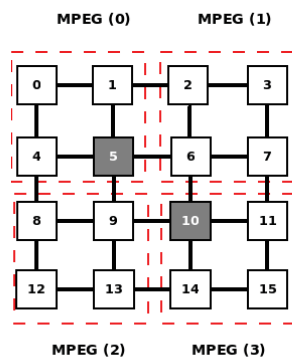


Figure 2.2: Exemple de l'exécution de 4 canaux audio/vidéo d'une application de décodage vidéo dans un simulateur MPSoC de 16 processeurs

	Cycle	Processeur	PC	Instruction	Adresse donnée	Latence
1	212305	1	0x10009d60	fetch	0x10009d60	28
2	212310	2	0x10007e14	load	0x10001a40	40
3	212333	1	0x10009d60	load	0x10001a40	52

Table 2.1: Extrait de trace matérielle

	horodatage	pid	operation	argument
1	93835345.380048	1003	E	0x400ac098 0x00000000 0x000001b6 0x000001b6 "/etc/localtime"
2	93835345.380062	1003	X	0x400ac098 -2
3	93835345.380163	1003	C	1003 0
4	93835345.381993	0	I	168
5	93835345.381999	0	i	
6	93835345.382002	0	S	4037f800
7	93835345.382067	0	s	
8	93835345.382069	0	Ix	
9	93835345.382074	0	C	0 1003

Table 2.2: Extrait de trace logicielle *Kptrace*

	horodatage	durée	opération	cpu_id	argument
1	02:08:13.107099573	(+0.000000951)	sys_recvmsg	3	fd = 16, msg = 0x7F141D941AE0, flags = 256
2	02:08:13.107101626	(+0.000002053)	exit_syscall	3	ret = 4136
3	02:08:13.107102113	(+0.000000487)	sys_poll	3	ufds = 0x7F141D942CB0, nfds = 2, timeout_msecs = -1
4	02:08:13.107102925	(+0.000000812)	exit_syscall	3	ret = 1
5	02:08:13.107103182	(+0.000000257)	sys_poll	2	ufds = 0x7FFF0F749F70, nfds = 1, timeout_msecs = -1
6	02:08:13.107103682	(+0.000000500)	sys_recvmsg	3	fd = 16, msg = 0x7F141D941AB0, flags = 0
7	02:08:13.107104723	(+0.000001041)	exit_syscall	2	ret = 1
8	02:08:13.107105362	(+0.000000639)	sys_writev	2	fd = 5, vec = 0x7FFF0F74A030, vlen = 3
9	02:08:13.107105805	(+0.000000443)	exit_syscall	3	ret = 1
10	02:08:13.107108431	(+0.000002626)	exit_syscall	2	ret = 8
11	02:08:13.107109269	(+0.000000838)	sched_stat_sleep	0	comm = "Xorg", tid = 1233, delay = 74354
12	02:08:13.107109923	(+0.000000654)	sys_recvfrom	2	fd = 5, ubuf = 0x112BB04, size = 4096, flags = 0, addr = 0x0, addr_len = 0x0

Table 2.3: Extrait de trace logicielle *Lttng*

	horodatage	pid	tâche	composant	fichier source	ligne	fonction	objet	arguments
1	3861312076	28988	0x1eb8450	filesrc	gstfilesrc.c	829	gst_file_src_create_read	filesrc0	'Reading 753 bytes at offset 0x1897ab'
2	3861424942	28988	0x1eb8450	queue	gstqueue.c	656	apply_buffer	queue1	'last_stop updated to 0:00:00.160000000'
3	3861747416	28988	0x1eb8400	queue	gstqueue.c	588	update_time_level	queue0	'sink 0:00:00.139319727, src 0:00:00.069659863'
4	3861692180	28988	0x1eb8450	qtdemux	qtdemux.c	3407	gst_qtdemux_combine_flows	demuxer	'combined flow return: ok'
5	3861607337	28988	0x1e4e770	ffmpeg	gstffmpegdec.c	2773	gst_ffmpegdec_chain	ffdec_h2640	'Before (while bsize>0). bsize:0 , bdata:0x7f92f0058b27'
6	3862181038	28988	0x1eb8400	faad	gstfaad.c	756	gst_faad_handle_frame	faad0	'242 bytes consumed, 2048 samples decoded'
7	3862804135	28988	0x1eb8400	queue	gstqueue.c	656	apply_buffer	queue0	'last_stop updated to 0:00:00.092879818'
8	3863209920	28988	0x1eb8400	faad	gstfaad.c	756	gst_faad_handle_frame	faad0	'243 bytes consumed, 2048 samples decoded'
9	3863344143	28988	0x1eb8400	queue	gstqueue.c	656	apply_buffer	queue0	'last_stop updated to 0:00:00.116099773'
10	3865839920	28988	0x1e4e770	queue	gstqueue.c	588	update_time_level	queue1	'sink 0:00:00.160000000, src 0:00:00.080000000'
11	3865915972	28988	0x1e4e770	ffmpeg	gstffmpegdec.c	2607	gst_ffmpegdec_chain	ffdec_h2640	'estimated duration 0:00:00.040000000 1'
12	3865971765	28988	0x1e4e770	ffmpeg	gstffmpegdec.c	2620	gst_ffmpegdec_chain	ffdec_h2640	'Received new data of size 222'
13	3866053776	28988	0x1e4e770	ffmpeg	gstffmpegdec.c	2259	gst_ffmpegdec_frame	ffdec_h2640	'data:0x7f92f0057c80, size:222, id:1'

Table 2.4: Extrait de trace logicielle *GStreamer*

Les évènements correspondent aux instructions d'un processeur *ARM* et portent des informations sur le cycle d'exécution de l'application, le processeur, l'instruction exécutée, l'adresse des données sollicitées et les délais d'accès aux données en mémoire.

Le niveau d'abstraction de ces informations permet de scruter finement les performances du *NoC*. Par exemple, les lignes 2 et 3 de la trace illustrent deux accès concurrents à l'adresse de données `0x10001a40`. Ces deux lignes permettent également d'observer que la latence d'accès à cette adresse à partir du processeur 1 est supérieure à la latence d'accès à partir du processeur 2. Cette information peut servir à formuler des hypothèses sur l'implication de la contention d'accès aux adresses mémoires sur le *NoC* comme origine de la contre performance de l'application [LTP13].

Toutefois, il est difficile d'exploiter le niveau d'abstraction matériel pour construire des hypothèses au niveau applicatif. En effet, l'identification de l'exécution d'une étape de décodage vidéo ou d'un mécanisme de gestion du parallélisme entre les tâches nécessite l'observation d'une grande quantité d'évènements. De plus, l'interprétation de ces évènements nécessite une connaissance du système et de l'application.

Le niveau système

La trace de la Table 2.2 provient d'une board ST et contient des évènements extraits à l'aide des *kprobes STLinux* dans l'outil *kptrace*. Les évènements correspondent à l'activité système sur un processeur durant l'exécution de l'application. Ils apportent des informations sur le numéro du processus exécuté et l'opération exécutée dans ce processus.

Les opérations correspondent aux activités du système *STLinux*. On distingue notamment les opérations liées aux changements de contexte entre les processus (opération C), aux interruptions (opérations I, i et Ix), aux *SoftIRQ* (opération S et s), aux appels systèmes (opérations E et X), à la gestion de la mémoire, à l'activité réseau et à la synchronisation des processus [STL].

Les informations système dépendent du système d'exploitation utilisé. Par exemple, la trace de la Table 2.3 représente également des informations système d'un système *Linux*. Ce niveau d'abstraction est plus synthétique que celui de la trace de la Table 2.1. Il permet de mieux appréhender la politique d'ordonnancement des tâches et des processus pendant l'exécution de l'application. Par exemple, on observe que le processus `1003` fait un appel système dont le code est situé à l'adresse `0x400ac098` (lignes 1 et 2). On observe également que le processus est par la suite interrompu (ligne 3) pour permettre l'exécution de l'interruption `168` (lignes 4 et 5) et de la *softIRQ* `4037f800` (lignes 6 et 7). On observe enfin que la fin de toutes les interruptions est signalée (ligne 8) et que le contexte du processus `168` est restauré (ligne 9) pour la poursuite de son exécution. Ce niveau d'abstraction système permet, par exemple, de formuler des hypothèses sur les origines d'un inter-blocage ou des interruptions maladroitement masquées [LCBT+12].

Contrairement à la trace précédente, ce niveau d'abstraction ne contient pas des informations matérielles et ne permet pas, par exemple, de facilement faire une corrélation entre une interruption anormalement longue et la contention d'accès à la mémoire qui se produit au niveau matériel. De même, bien que l'activité système soit perceptible, il reste difficile de vérifier l'intégrité

des contraintes fonctionnelles de l'application. En effet, le niveau d'abstraction applicatif est noyé dans les informations liés aux autres applications en cours d'exécution et au système d'exploitation.

Le niveau applicatif

La trace de la Table 2.4 contient des évènements extraits à l'aide du paramétrage du framework *GStreamer*. Ces évènements correspondent à l'activité des différents composants de décodage audio/vidéo du framework *Gstreamer*. Ils apportent des informations sur le processus et le composant exécuté, le fichier source et la ligne de la fonction exécutée, le nom de la fonction appelée, la référence de l'objet qui appelle la fonction et les arguments permettant de compléter l'interprétation de l'évènement. La trace permet de distinguer les composants *GStreamer* tels que *filesrc* pour la lecture des données depuis le flux d'entrée, *queue* pour la temporisation des données lors de la communication entre deux composants, *qtdemux* pour la séparation des flux audio et vidéo dans le flux d'entrée, *faad* pour le décodage d'un flux audio et *ffmpeg* pour le décodage d'un flux vidéo [GSt].

L'observation des évènements de la trace permet de distinguer les phases de décodage des images de la vidéo et du son associé. Par exemple, l'objet `demuxer` (ligne 4), suivi par les décodeurs vidéo `ffdec_h2640` et audio `faad0` (lignes 5 et 6). Les connaissances métier sur les normes de décodage audio/vidéo *MPEG* [EH00] et *H.264* [ZLC03] permettent de formuler des hypothèses sur les durées des phases de décodage et la taille des données correspondant aux images. Ainsi, les lignes 6 et 8 permettent de vérifier les taille de données décodées par chaque étape de décodage, et la ligne 11 permet de valider les hypothèses sur les temps d'exécution du décodage. Ce niveau d'abstraction applicatif autorise la formulation des hypothèses sur les origines des bugs et des contre-performances au niveau métier et permet par exemple l'analyse des problèmes de désynchronisation des flux audio et vidéo [KKFT+13, KIRT13]. Toutefois, ce niveau n'autorise pas la formulation des hypothèses aux niveaux système et matériel. L'analyste perçoit implicitement ces niveaux grâce à ses propres connaissances. Par exemple, ses connaissances métier peuvent lui permettre de déduire la présence des appels systèmes de lecture de données lors du décodage vidéo.

2.1.4 La taille des traces

La taille des traces dépend du niveau de granularité des évènements collectés et de la durée de la collecte. Considérons les résultats consignés dans la Table 2.5. Ils représentent repectivement les nombres d'évènements et les tailles des traces collectées pendant 30 secondes par differents outils de collecte de traces. On distingue les traces collectées pendant l'exécution d'une application de décodage vidéo sur un simulateur de MPSoC. Puis, les traces collectées sur l'activité d'un système d'exploitation *Linux* à l'aide de l'outil *Lttng*. Ensuite, les traces produites par le framework GStreamer et enfin les traces kptrace.

Les traces issues du simulateur MPSoC sont de granularité très fine (niveau matériel) et sont en conséquence plus importantes que les traces issues de GStreamer ou kptrace qui sont plus synthétiques (niveau applicatif et système). Au vue des résultats de la table 2.5, on a une idée de ce que peut représenter

	# évènements	Taille disque
Simulateur MPSoC	248 214 400	8 000 Mo
Lttng	2 871 131	328 Mo
GStreamer	110 000	13 Mo
Kptrace	90 109	5.6 Mo

Table 2.5: Détails des traces issues de 30 secondes de collecte sur différents outils

une trace issue d'une application de vidéoconférence exécutée pendant 1 heure. En effet, si nous considérons le ratio approximatif de 10 Mo d'espace disque et 0.1 million d'évènements par seconde tel que le montre la trace *Lttng* dans la Table 2.5, alors une collecte de 1 heure produira une trace de 36 Go contenant 360 millions d'évènements.

Les exemples de traces que nous avons présenté dans cette section, montrent la complémentarité des niveaux d'abstraction dans le processus d'analyse d'une application. Ainsi, l'analyste utilise ses propres connaissances du métier et de l'architecture logicielle et matérielle pour effectuer les changements de niveau d'abstraction lui permettant de formuler et/ou vérifier ses hypothèses. A cause des tailles généralement grandes des traces des applications multimédia, les analystes se servent généralement des outils d'analyse de traces pour accéder aux informations contenues dans la trace et à les interpréter au niveau d'abstraction qui les convient. Il existe deux types d'analyse de trace, l'analyse *temps réel* et l'analyse *post-mortem*. L'analyse *temps réel* se fait généralement au moment de la collecte, tandis que l'analyse *post-mortem* se fait une fois que les traces sont entièrement collectées. Dans la prochaine section, nous allons parcourir les différentes approches d'analyse *post-mortem* des traces de l'état de l'art.

2.2 L'état de l'art de l'analyse *post-mortem* des traces

La littérature présente plusieurs approches pour l'analyse *post-mortem* des traces d'exécution [CZVD+09]. On distingue par exemple les approches basées sur la visualisation des informations, sur la fouille de données et sur l'analyse de multiples traces.

2.2.1 L'analyse de la trace à l'aide de la visualisation

L'idée de cette approche est d'exploiter les capacités visuelles de perception de l'analyste qui lui permettent de facilement identifier des corrélations et des motifs graphiques [SGF99]. L'approche consiste donc à visualiser les informations contenues dans la trace. Il existe une multitude d'outils de visualisation de traces. Chacun de ces outils est généralement dédié à un format de trace bien spécifique. Nous allons distinguer les outils de visualisation qui se contentent de représenter les informations contenues dans la trace de ceux qui construisent et proposent de nouveaux niveaux de granularité des informations.

Visualiser les informations de la trace

Cette catégorie d'outils de visualisation permet de représenter de façon temporelle les informations contenues dans la trace. La représentation se fait généralement sous la forme d'un diagramme de *Gantt*. On peut citer comme exemple *Kptrace viewer* pour les traces *Kptrace* [PRSDP+10], *Trace compass* pour les traces *Lttng* [Com]. La Figure 2.3 est une capture de la visualisation d'une trace *Kptrace* (dont un extrait est présenté dans la Table 2.2) sous la forme de pages dans l'outil *Kptrace viewer*.

La capture représente 5% de la première page (sur un total de 60 pages) soit 11 790 microsecondes sur 1 minute et 20 secondes de temps d'exécution de l'application. *KPTrace Viewer* permet de zoomer pour avoir une vue plus précise des informations de la page sans toutefois changer la granularité de ces informations. L'outil visualise les changements de contexte (représentés par des flèches verticales) entre les exécutions des tâches (représentées par des bandes horizontales violettes ou blanches) ou des interruptions (bandes horizontales vertes). Nous pouvons ainsi retrouver visuellement la relation que nous avons identifié dans l'extrait de la trace (Section 2.1.3) entre l'interruption `168` (`gic_eth0`), la *softIRQ* (`net_rx_action`) et le processus `1003` (`syslogd`).

Bien que cette visualisation permette de visuellement identifier, par exemple, des situations d'inter-blocage entre des tâches, l'exploration des traces de grandes tailles devient difficile en pratique. De plus, ce type de visualisation ne permet pas de manipuler ni de sauvegarder les niveaux d'abstraction pertinents construits par l'analyste pendant l'exploration de la trace. En conséquence, l'outil ne propose aucun moyen de découvrir plus efficacement ces niveaux d'abstraction dans de nouvelles traces.

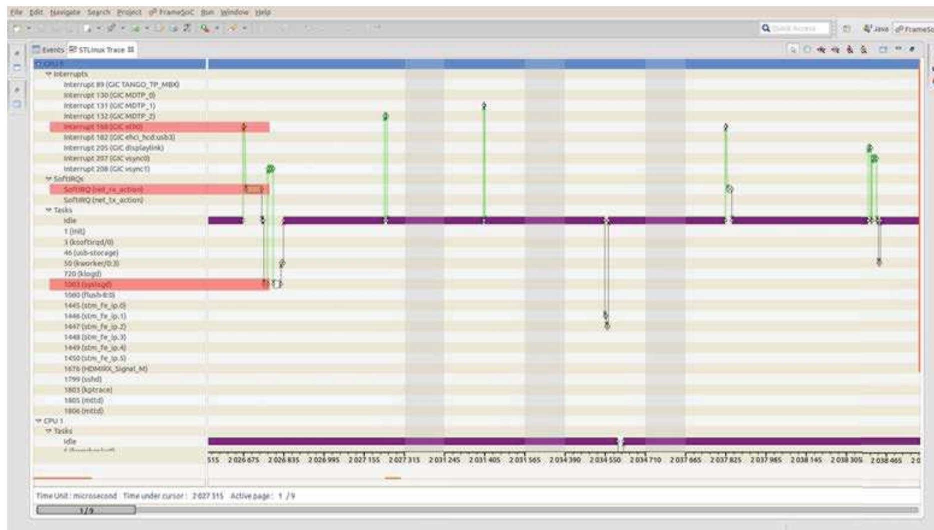


Figure 2.3: Capture d'écran de l'outil Kptrace viewer.

Visualiser les informations en niveaux de granularité

Dans cette catégorie d'outils de visualisation, les informations sont présentées à des niveaux de granularité qui sont plus expressifs que le niveau basique de la trace. On peut citer *Ocelotl* comme exemple d'outil de cette catégorie [DHV+13].

L'outil *Ocelotl* utilise les notions de la théorie de l'informaton telles que l'entropie, pour construire des agrégations temporelles d'évènements pour distinguer les variations de la quantité d'information au cours de l'exécution de l'application. Cette agrégation temporelle permet de distinguer des phases d'exécution dans la trace.

La Figure 2.4 représente une capture d'écran de la visualisation, dans *Ocelotl*, de la même trace *Kptrace* que celle de la Figure 2.3. La capture représente (à droite) des agrégats temporelles des évènements permettant d'identifier 5 phases d'exécution dans toute la trace. La capture présente également (à gauche) l'évolution de la quantité d'information des événement selon differents paramétres d'agrégation.

Les phases d'exécution identifiées par *Ocelotl* sont potentiellement pertinentes pour le développeur. En effet, elles peuvent, par exemple, permettre d'identifier les fluctuations dans les performances de l'application durant son exécution. Ainsi, ces agrégats peuvent représenter des niveaux d'abstraction. Toutefois, *Ocelotl* ne propose pas de moyen d'interpréter les agrégats qu'il construit, ce qui limite leur exploitation sous forme de niveaux d'abstraction.

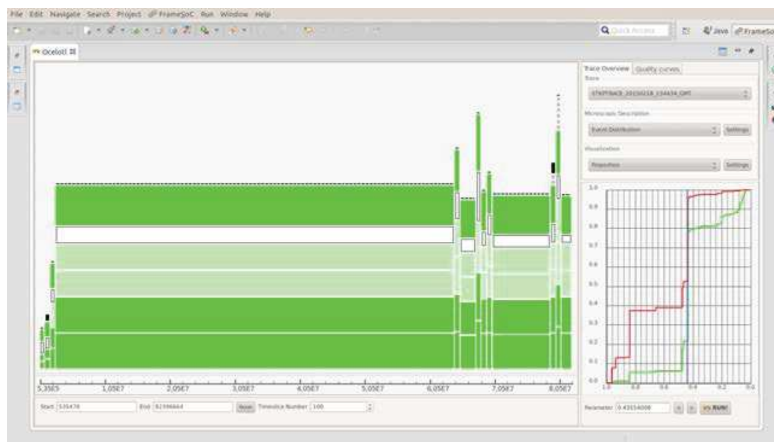


Figure 2.4: Visualisation d'une trace Kptrace avec Ocelotl

Certains outils de visualisation présentés dans cette section ont l'inconvénient d'être non-automatiques. En effet, l'exploration des pages dans *Kptrace viewer* n'est pas automatique. De plus, les outils de visualisation ne proposent pas d'assistance à l'analyste pour l'interprétation des informations graphiques qu'ils présentent. En conséquence, une grande expertise de l'analyste est nécessaire pour les utiliser efficacement. Enfin, bien que les informations représentées par les outils de visualisation peuvent représenter de nouveaux niveaux d'abstraction, ces outils offrent difficilement des moyens au développeur pour exploiter ces

niveaux d'abstraction potentiellement pertinents. Dans la prochaine section, nous allons présenter des approches d'analyse de traces automatiques et semi-automatiques basées sur la fouille de données.

2.2.2 L'analyse des traces à l'aide de la fouille de données

L'idée de cette approche est de trouver automatiquement des motifs d'évènements dont la pertinence est liée à ses apparitions dans la trace. Par exemple, un motif d'évènements peut être pertinent parce qu'il apparaît (ou pas) un nombre de fois spécifique dans la trace, ou parce que ses apparitions respectent une périodicité, ou encore parce que les évènements qui le composent respectent des contraintes spécifiques.

L'approche se base sur des algorithmes de recherche des motifs fréquents, de recherche des motifs représentatifs, de recherche des motifs périodiques et de recherche de motifs sous contraintes.

La recherche des motifs fréquents

A cause de la dimension temporelle des informations dans la trace, les outils basés sur les algorithmes de recherche des séquences fréquentes ont été privilégiés. De manière formelle, soit Σ l'ensemble de toutes les informations possibles apparaissant dans une trace donnée (par exemple les types d'opérations ou les instructions processeur). Un évènement est une information horodatée $e = (t, o)$ avec $t \in \mathbb{N}$ et $o \in \Sigma$. La fonction $\pi(t, o) = o$ retourne l'information portée par un évènement. $\mathcal{T} = \langle e_1, ..., e_n \rangle$ est une trace de taille n. Soit $S = \langle s_1, ..., s_k \rangle, \forall s_i \in \Sigma$, une séquence d'informations de taille k, on dit que S apparaît, ou a une occurrence, dans $\mathcal{T}$ et on note $S \subset \mathcal{T}$ s'il existe une position i dans la trace à partir de laquelle les k prochaines informations consécutives correspondent à celles de S. $\exists\ i \in [1, n] \mid \forall j, i \leq j \leq k, \pi(e_{i+j-1}) = s_j$. Le nombre d'occurrences $\mathcal{O}(S, \mathcal{T})$ de la séquence S, est le nombre de fois qu'elle apparaît dans la trace $\mathcal{T}$. On dit que S est une séquence fréquente dans la trace si $\mathcal{O}(S, \mathcal{T}) \geq \epsilon$ où ϵ est un seuil de fréquence minimal donné.

Un algorithme de recherche des séquences fréquentes dans les traces d'exécution est *Profspan* [ZXHW10]. *Profspan* prend en entrée une trace et le seuil de fréquence minimal et retourne toutes les séquences fréquentes correspondantes.

L'algorithme commence par calculer toutes les séquences fréquentes de taille 1. Puis de façon itérative, il calcule les séquences fréquentes de taille $k + 1$ en étendant chaque séquence fréquente de taille k. L'algorithme s'arrête lorsque aucune séquence fréquente de taille $k + 1$ n'a été trouvée.

La Table 2.6 représente une séquence d'évènement construite en considérant uniquement les appels systèmes comme information dans la trace de la Table 2.3. L'exécution de *Profspan* avec cette trace en entrée et un seuil de fréquence minimal de 2, retourne l'ensemble des séquences d'évènements qui apparaissent au moins deux fois dans la trace. Par exemple, la séquence $\langle exit_syscall, sys_poll \rangle$ apparaît deux fois dans la trace aux positions 2 et 4 (en bleu dans la Table 2.6).

Ces séquences fréquentes permettent, par exemple, d'identifier des séquences d'instructions suspectes pouvant être à l'origine des dégradations des performances de l'application [ZXHW10].

rang	évènement
1	sys_recvmsg
2	exit_syscall
3	sys_poll
4	exit_syscall
5	sys_poll
6	sys_recvmsg
7	exit_syscall
8	sys_writev
9	exit_syscall
10	exit_syscall
11	sched_stat_sleep
12	sys_recvfrom

Table 2.6: Exemple des deux occurrences (en bleu) dans la trace de la séquence fréquente résultat $\langle exit_syscall, sys_poll \rangle$ retrouvé par *Profspan* avec une fréquence de 2.

Cependant les algorithmes de recherche de séquences fréquentes (par exemple *Profspan*) produisent des motifs qui sont potentiellement triviaux ou uniquement utiles pour l'analyse des performances. D'autres algorithmes de fouille de données sont utilisés dans les cas où on recherche des motifs d'évènements rares (discriminants) ou ceux dont les apparitions sont périodiques.

Les motifs fréquents périodiques

La périodicité est une caractéristique des applications multimédia. Les motifs fréquents périodiques sont des motifs fréquents qui se répètent en respectant une période de temps. L'algorithme *PerMiner* est le premier à rechercher des motifs fréquents périodiques dans les traces d'exécution [LCBT+12]. Ce type de motifs fréquents sont efficaces pour caractériser des situations telles que les interruptions périodiques ou les évènements provenant des instruction exécutées en boucle.

La recherche de motifs représentatifs

L'idée dans cette approche est de trouver des motifs d'évènements dont la pertinence est liée au fait que leurs apparitions dans une partie ou toute la trace, permettent de caractériser les parties ou traces où ils n'apparaissent pas. L'approche consiste à rechercher des motifs fréquents puis à les utiliser pour caractériser les zones où ces motifs apparaissent ou pas.

On peut citer *Frameminer* comme exemple d'outil d'analyse de cette catégorie. *FrameMiner* propose une réécriture de la trace en utilisant un ensemble de séquences fréquentes d'évènements garantissant une couverture maximale de la trace [KKFT+13, KFI+12]. Cette réécriture permet d'identifier des zones de régularité (pouvant correspondre à des contraintes fonctionnelles) et de directement mettre en relief des zones non couvertes potentiellement intéressantes pour

l'analyste. L'algorithme est basé sur une heuristique de calcul d'un k-ensemble de séquences d'évènements de couverture maximale.

L'ensemble $\mathcal{S}$ ci-dessous est un résultat que retourne *Frameminer* sur la trace de la Table 2.6 avec une fréquence minimale de 2 et une taille maximale résultat de 3. Ce résultat recouvre 75% de la trace comme le montre la Table de 2.7.

$$\mathcal{S} = \{\langle sys_recvmsg\rangle, \langle exit_syscall, sys_poll\rangle, \langle exit_syscall\rangle\}$$

rang	évènement
1	sys_recvmsg
2	exit_syscall
3	sys_poll
4	exit_syscall
5	sys_poll
6	sys_recvmsg
7	exit_syscall
8	sys_writev
9	exit_syscall
10	exit_syscall
11	sched_stat_sleep
12	sys_recvfrom

Table 2.7: Example de réécriture de trace par Frameminer

L'avantage de *Frameminer* est la limitation de la taille du résultat qui est un paramètre d'entrée de l'algorithme, et la proposition d'une interprétation des motifs fréquents résultats. En effet, ces motifs fréquents sont caractérisés par le fait qu'ils offrent une couverture maximale de la trace. Ces résultats constituent un nouveau niveau d'abstraction qui est automatiquement exploité dans la réécriture de la trace. Ce niveau d'abstraction peut être sauvegardé et exploité dans l'analyse d'une autre trace.

Toutefois, *Frameminer* se base uniquement sur la couverture maximale des séquences fréquentes pour construire ce nouveau niveau d'abstraction. Il ne tient pas compte, par exemple, des connaissances métier de l'analyste. Des approches de recherche de motifs représentatifs dans lesquels les connaissances métier sont pris en compte ont récemment émergé.

On peut citer par exemple les travaux de *Lagraa et al* qui exploitent une modélisation, sous forme de clusters, des connaissances métier de l'analyste sur les latences d'accès aux adresses mémoire [LTP13]. Le modèle permettant ainsi de distinguer les adresses mémoires sur lesquelles il y a de la contention, sert ensuite à filtrer les séquences fréquentes d'évènements que retourne un algorithme de recherche de motifs fréquents appliqué aux traces.

Contrairement à *Frameminer* et à *Profspan*, cette approche produit des motifs fréquents ayant directement un intérêt pour l'analyse. En effet, les motifs fréquents correspondent à des accès aux adresses mémoire qui subissent de la contention sur le réseau de connexion (NoC) des processeurs du MPSoC. Toutefois, le modèle utilisé pour représenter les connaissances métier est rigide et n'offre pas beaucoup de flexibilité à l'analyste. En effet, l'extensibilité et

l'interrogation de ce modèle restent très limitées à la qualité des accès aux adresses mémoires.

Les approches de fouilles de données pour l'analyse de traces ont l'avantage de découvrir de façon automatique des motifs fréquents qui peuvent être considérés comme de nouveaux niveaux d'abstraction. De plus, ces nouveaux niveaux d'abstraction peuvent servir à des analyses ultérieures.

Toutefois, l'interprétation des motifs fréquents résultats par les approches de fouille de données reste limitée. En effet, les moyens qu'offrent ces approches pour l'interprétation des motifs fréquents résultats, sont limités par la taille potentiellement grande des résultats et par le fait qu'elles n'utilisent pas (ou très peu) la large palette des connaissances métier de l'analyste. Dans la prochaine section nous allons présenter une approche d'analyse de traces qui se base sur des analyses ultérieures pour construire un modèle des connaissances métier de l'analyste.

2.2.3 L'analyse par comparaison de traces

L'idée de cette approche est de construire un modèle des connaissances métier de l'analyste à partir des résultats de précédentes analyses de traces. L'approche consiste à exploiter plusieurs traces pour mieux analyser une nouvelle trace.

Amiar et al proposent la construction, par apprentissage, d'un modèle des informations contenues dans une trace. Ce modèle est ensuite utilisé comme une version compressée de la trace ou pour l'analyse d'autres traces [ADFDB13].

Amiar et al utilisent l'algorithme *Sequitur* pour découvrir la structure des évènements dans une trace d'exécution. *Sequitur* est un algorithme récursif qui infère des structures hiérarchiques (correspondant à des grammaires hors contexte) à partir d'une séquence de symboles [NMW97].

La Table 2.8 montre deux règles de grammaire hors contexte (à droite) que retourne *Sequitur* quant il est appliqué sur la trace (à gauche). Ce résultat permet d'établir une relation entre les évènements de la séquence $\langle exit_syscall, sys_poll \rangle$ et montre comment cette relation est exploitée ailleurs dans la trace.

rang	évènement	symbole
1	sys_recvmsg	A
2	exit_syscall	B
3	sys_poll	C
4	exit_syscall	B
5	sys_poll	C
6	sys_recvmsg	A
7	exit_syscall	B
8	sys_writev	D
9	exit_syscall	B
10	exit_syscall	B
11	sched_stat_sleep	E
12	sys_recvfrom	F

$$S_0 \rightarrow A\ S_1\ S_1\ A\ B\ D\ B\ B\ E\ F$$
$$S_1 \rightarrow B\ C$$

Table 2.8: Algorithme *Sequitur* appliqué à la trace

L'approche d'analyse de trace par apprentissage de grammaire est efficace dans les cas où il existe une structure des évènements de trace. Cela n'est pas toujours le cas pour les applications embarquées à cause de la complexité des architectures, et de l'ordonnancement des tâches qui s'exécutent en parallèle sur ces systèmes.

Kamdem et al ont récemment proposé l'outil *TED* qui exploite les niveaux d'abstraction découverts dans de précédentes analyses de traces pour faciliter l'analyse d'une nouvelle trace [KIRT13]. *TED* est un outil de comparaison de trace pour les applications vidéo qui propose des métriques de distances prenant en compte différents types d'anomalies précédemment identifiées dans des analyses de traces de référence. Grâce à ces distances, l'outil peut déterminer la similarité entre une nouvelle trace et une trace de référence par rapport à différentes anomalies connues.

2.3 Conclusion

Dans la première partie de ce chapitre, nous avons introduit les traces et nous avons présenté les différentes techniques de collecte des évènements de traces. Nous avons ensuite présenté la notion de niveau d'abstraction des informations contenues dans la trace et avons montré l'intérêt de ces niveaux d'abstraction pour faciliter la compréhension de l'application à partir des traces. La seconde partie de ce chapitre a porté sur la présentation des différentes approches pour l'analyse de traces.

Il apparaît que, de façon générale, ces approches d'analyse offrent des moyens limités pour construire et/ou interpréter les informations de la trace à plusieurs niveaux d'abstraction différents. De plus ces approches sont généralement non-flexibles et spécialisées à un type de problématique ou à un type d'application précis. Nous avons identifié que ces limites étaient dues au fait que ces approches ne prenaient pas ou très peu en compte les connaissances métier de l'analyste. Ce constat motive la proposition d'une approche permettant de prendre en compte la large palette des connaissances métier de l'analyste tout au long du processus d'analyse de la trace.

Les limitations de l'exploitation des connaissances métier par les outils d'analyse de traces, peuvent provenir du fait que ces connaissances ne sont pas formalisées. La formalisation, la représentation et l'exploitation des connaissances sont des problématiques bien connues dans le Web Sémantique où les ontologies sont utilisées comme modèle de représentation des connaissances.

Dans le prochain chapitre, nous allons présenter les définitions théoriques permettant de comprendre ce que sont les ontologies, comment elles sont construites et comment elles fonctionnent. Nous allons ensuite appliquer ces ontologies au domaine de l'analyse de traces d'applications multimédia sur MPSoC.

Chapitre 3

VIDECOM, une ontologie pour l'analyse des traces d'applications multimédia sur MPSoC

Sommaire

3.1 L'ontologie pour la représentation des connaissances 43
3.2 La construction d'une ontologie pour l'analyse des traces d'exécution 43
3.2.1 Les approches de construction d'une ontologie 43
3.2.2 Les informations explicites 44
3.2.3 Les informations implicites 45
3.3 Les langages de construction des ontologies 45
3.3.1 RDF 46
3.3.2 RDFS 47
3.4 L'ontologie VIDECOM 48
3.4.1 Les classes 48
3.4.2 Les propriétés 48
3.4.3 Les instances 51
3.4.4 Les contraintes 51
3.5 L'interrogation d'une ontologie 58
3.5.1 Les motifs de triplets 58
3.5.2 SPARQL 58
3.6 Illustration de l'interrogation de VIDECOM pour l'analyse de traces 61
3.7 Conclusion 62

Nous avons vu, dans le chapitre précédent, que les connaissances métier étaient très peu formalisées et utilisées dans les outils d'analyse de traces. L'objectif de ce chapitre est de montrer l'usage des ontologies comme des modèles

de représentation de ces connaissances métier dans le contexte de l'analyse des traces.

La première partie du chapitre porte sur la définition des ontologies (Section 3.1). Elle introduit la logique de description qui est la base formelle des ontologies, et présente les différents types d'ontologies. La seconde partie porte sur les méthodes et les langages de construction des ontologies (Section 3.2). Cette partie détaille la construction de VIDECOM, notre ontologie pour l'analyse des traces d'exécution d'applications embarquées sur MPSoC. La troisième partie du chapitre porte sur l'interrogation de VIDECOM pour l'analyse de traces (Section 3.5). La quatrième partie du chapitre présente les capacités de raisonnement de VIDECOM pour d'inférence de connaissances métier (Section 3.4.4). Enfin, la dernière partie conclut le chapitre.

Introduction

Dans chacune des parties de ce chapitre, nous allons illustrer l'ontologie VIDECOM pour analyser la trace de la Table 3.1. La trace, constituée de 5 évènements, représente les informations produites par deux processeurs pendant l'exécution d'une application multimédia, appelée `copyMovie`, pour la copie des données d'un fichier vidéo.

Début	Durée	Opération	Tâche	Processeur	Argument
525323	14	sys_read	copyMovie	0	29
525337	0	switch_to	idle	1	sshd
525337	4	sys_write	copyMovie	0	29
525341	6	sys_read	copyMovie	0	96
525347	20	sys_write	copyMovie	0	56

Table 3.1: Trace exemple.

Cette trace contient deux types d'évènements, les appels systèmes et les changements de contexte. Chaque évènement porte un ensemble d'informations telles que le temps de début, la durée, l'opération exécutée, la tâche et le processeur sur lequel l'opération a été exécutée.

On distingue trois types d'opérations, `sys_read` pour la lecture des données, `sys_write` pour l'écriture des données et `switch_to` pour le changement de contexte entre deux tâches. La sémantique de ces opérations est complétée par la valeur de l'argument de l'évènement correspondant. Par exemple, dans le cas d'un `sys_read` (respectivement `sys_write`), cette valeur représente la taille des données lues (respectivement écrites). Et dans le cas d'un changement de contexte, elle représente la tâche qui sera prochainement exécutée sur le processeur. L'application `copyMovie` respecte la contrainte fonctionnelle suivante: *"l'écriture d'une taille de données est directement précédée par la lecture de la même quantité de données"*.

3.1 L'ontologie pour la représentation des connaissances

L'ontologie est une formalisation des concepts d'un domaine ainsi que leur sémantique [Gru95]. Il s'agit d'une définition, interprétable par un ordinateur, des concepts basiques d'un domaine d'intérêt et des relations qui existent entre eux [NM+01]. Une ontologie sert non seulement à partager une compréhension commune des connaissances d'un domaine, mais aussi à réutiliser et à raisonner sur ces connaissances pour comprendre le domaine ou déduire de nouvelles connaissances.

On distingue deux types d'ontologies, les ontologies générales et les ontologies du domaine [RH06]. Les ontologies générales modélisent des concepts généraux de haut niveaux. Comme exemple d'ontologie générale, on peut citer *DOLCE*, l'ontologie pour l'ingénierie linguistique et cognitive [MBG+02], les ontologies des relations temporelles [HUA12, HP04] et l'ontologie des liens de causalité [Gal12]. Les ontologies du domaine sont destinées à des domaines d'application plus précis. Elles utilisent généralement des taxonomies de concepts et de propriétés pour décrire le domaine. On peut citer par exemple *Wordnet* l'ontologie du lexique de la langue anglaise qui contient les concepts tels que les *mots*, les *verbes* et les règles lexicales [Mil95]. On peut également citer *Yago* [SKW07] et *DBpedia* [LIJ+14] qui sont des ontologies du domaine de l'encyclopédie libre *wikipedia*, ou encore *Gene Ontology* l'ontologie du domaine de la génétique qui contient les concepts tels que les molécules et les gènes [ABB+00].

Les connaissances d'un domaine sont formalisées dans les ontologies par un ensemble de classes, un ensemble de propriétés, un ensemble d'instances et un ensemble de règles d'inférence. Les classes représentent les concepts pertinents du domaine tandis que les propriétés représentent les relations qui existent entre ces différents concepts. Les instances représentent les individus du domaine et sont des instances des classes. Enfin, les règles d'inférence permettent de raisonner sur les concepts de l'ontologie pour préciser leur sémantique dans le contexte du domaine d'intérêt.

Dans la prochaine section nous allons nous intéresser à la construction des ontologies. Nous allons présenter comment se fera la construction des classes, des propriétés, des instances et des règles d'inférence dans le contexte d'une ontologie du domaine de l'analyse des traces.

3.2 La construction d'une ontologie pour l'analyse des traces d'exécution

3.2.1 Les approches de construction d'une ontologie

On distingue les approches *Top-down* et *Bottom-up* pour la construction des ontologies. L'approche *Top-down* est utilisée quand l'ontologie à construire est dédiée à un domaine proche d'un autre domaine plus général déjà défini ou existant déjà sous la forme d'une ontologie générale [KJLW12]. L'approche consiste à étendre cette ontologie existante. Par exemple, une ontologie sur les gènes cancérigènes peut se construire en *Top-down* à partir de l'ontologie générale des

gènes. L'approche *Bottom-up* construit l'ontologie à partir de zéro, elle est basée sur une analyse des concepts pertinents et la sémantique du domaine considéré [VDVM98].

L'analyse d'une trace d'exécution est spécifique à une architecture matérielle, logicielle ou à un environnement d'exécution particulier. De ce fait, il n'existe pas, dans la littérature actuelle, d'ontologie générale pour l'analyse de traces d'exécution. Ainsi, dans le cas particulier des applications multimédia sur MP-SoC, nous allons construire l'ontologie VIDECOM à partir de zéro en suivant l'approche *Bottom-Up* présentée par *Deborah L. McGuinness* qui se décompose en quatre étapes [NM+01]:

1. Identifier les classes pertinentes de la trace

2. Organiser ces classes sous forme de taxonomies

3. Identifier les propriétés entre les différentes classes dans la trace

4. Identifier les instances des classes dans la trace

Dans le contexte de l'analyse des traces d'applications multimédia sur MP-SoC, les classes pertinentes correspondent aux informations qui contribuent à la compréhension du fonctionnement de l'application. Ces informations rassemblent les informations explicites qui sont directement contenues dans les évènements de traces et les informations implicites qui dérivent des contraintes métier de l'application.

3.2.2 Les informations explicites

Les informations explicites sont contenues dans les évènements et peuvent être rassemblées en quatre catégories : *signalétique*, *temporelle*, *spatiale* et *sémantique*. Nous allons les définir en les illustrant avec notre trace exemple de la Table 3.1.

- La catégorie *signalétique* rassemble les informations intrinsèques permettant de décrire l'évènement. Par exemple, l'opération exécutée par l'évènement, le matériel accédé et la fonction appelée par l'opération.

- La catégorie *temporelle* rassemble les informations liées au temps telles que les temps de début ou de fin et la durée des évènements. Elle rassemble également les informations liées à la relation d'ordre entre les évènements.

- La catégorie *spatiale* rassemble les informations permettant de préciser le lieu d'exécution de l'évènement. Par exemple le processeur ou la tâche en cours d'exécution.

- Les informations *sémantiques* correspondent à des informations de plus haut niveau d'abstraction. Par exemple, les fonctionnalités correspondant aux lectures et écritures de données dans l'application `copyMovie`.

3.2.3 Les informations implicites

Les informations implicites dérivent des contraintes métier. La Table 3.2 représente quelques contraintes métier extraites de la description de la trace d'exécution de la Table 3.1. Ces contraintes métier permettent d'enrichir les informations contenues dans la trace.

C_1	"Le temps de fin d'un évènement est la somme de son temps de début et de sa durée"
C_2	"L'opération `sys_read` a comme argument la taille des données lues"
C_3	"L'opération `sys_write` a comme argument la taille des données écrites"
C_4	"L'écriture d'une taille X de données par `copyMovie` doit être précédée, dans la même tâche, par la lecture de cette même taille X de données"

Table 3.2: Liste des contraintes métier pour l'analyse de la trace exemple

Les différentes catégories d'information ainsi que les contraintes métier sont représentées dans VIDECOM par un ensemble de classes, de propriétés, d'instances et de règles d'inférence. Dans la prochaine section, nous présentons les différents langages utilisés pour exprimer ces différents concepts.

3.3 Les langages de construction des ontologies

Trois langages sont issus du standard du Web Sémantique pour décrire les ontologies. Il s'agit de *RDF*, de *RDFS* et de *OWL*. *RDF* permet de représenter les faits. Les deux autres langages permettent d'exprimer des contraintes sur ces faits. Le *RDFS* permet de décrire le schéma hiérarchique de l'ontologie tandis que *OWL* est plus expressif. Nous allons commencer par clarifier les notions de nommage et de noms de domaines du Web Sémantique, qui permettent d'identifier et d'accéder sans ambiguïté aux concepts d'une ontologie.

L'URI et les noms de domaines.

Une ontologie est identifiée par une URL (Uniform Resource Locator). Par exemple, l'URL `http://videcom.imag.fr/vd.rdf#` permet d'identifier l'ontologie VIDECOM. Les instances dans une ontologie sont identifiées par des URI (Uniform Resource Identifier). Un identifiant URI est une URL accessible par tout agent logiciel ou humain. Par exemple, l'instance `event1` qui représente le premier évènement de la trace dans VIDECOM aura l'URI :

`http://videcom.imag.fr/vd.rdf#event1`

Pour éviter ces longues URI, on utilise les noms de domaine. Par exemple, nous pouvons définir le nom de domaine `videcom:` avec l'URL de

VIDECOM. Ainsi l'abréviation de l'URI de l'instance `event1` dans l'ontologie `videcom:` devient `videcom:event1`. Les exemples que nous allons présenter dans la suite étant tous dans l'ontologie VIDECOM, pour parler des individus ou des propriétés de cette ontologie, nous utiliserons la notation `:Name` au lieu de `videcom:Name`, en considérant que `videcom:` est le nom de domaine par défaut.

Il existe des noms de domaines standards qui sont utilisés dans les domaines particuliers. La Table 3.3 donne les URLs où se trouve respectivement une ontologie qui précise les classes et propriétés du domaine considéré.

Noms	URL	domaine d'intérêt
`rdf:`	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`	RDF
`rdfs:`	`http://www.w3.org/2000/01/rdf-schema#`	RDFS
`owl:`	`http://www.w3.org/2002/07/owl#`	OWL
`xsd:`	`http://www.w3.org/2001/XMLSchema.xsd`	les littéraux

Table 3.3: Liste des noms de domaines standards

Les ressources anonymes

Une ressource anonyme (ou *nœud blanc*) est une ressource qui n'est pas identifiée par une URI. Le nœud blanc se distingue des URI par le symbole "_:", par exemple `_:bl`. Les *nœuds blancs* servent à la représentation des données complexes, ou comme nous le verrons dans la Section 3.5, à l'identification des nouvelles instances produites par les règles d'inférence.

3.3.1 RDF

RDF (Resource Description Framework) est un langage simple qui permet de décrire les faits d'une ontologie contenant des instances identifiées par des URIs. Un fait *RDF* est un triplet qui se compose d'un sujet, d'une propriété et d'un objet. En d'autres termes, un triplet `(s P o)` exprime le fait que l'individu `o` est la valeur de la propriété `P` appliquée à l'individu `s`.

Dans un triplet, la propriété et le sujet sont des ressources identifiées par une URI, tandis que l'objet peut être une URI ou un littéral représentant une valeur. Dans ce dernier cas, le triplet exprime le fait que le sujet a cette valeur pour la propriété considérée. Il est également à noter qu'un *nœud blanc* peut être sujet ou objet d'un triplet *RDF*.

RDF permet de faire la distinction entre les individus (objet ou sujet) et les propriétés grâce à deux mots clés `rdf:type` et `rdf:Property`. Par exemple, le triplet `(:isExecutedOnCPU rdf:type rdf:Property)` indique que `:isExecutedOnCPU` est une propriété. La propriété `rdf:type` est également utilisée pour déclarer qu'un individu `i` est une instance de la classe `C` grâce au triplet *RDF* de la forme `(i rdf:type C)`. Par exemple, les deux triplets suivants expriment les faits que `:event1` est une instance de la classe `:SystemRead`, et que `:cpu0` est une instance de la classe `:CPU`.

```
(:event1 rdf:type :SystemRead)
```

(:cpu0 rdf:type :CPU)

3.3.2 RDFS

RDFS (pour *RDF* schema) est un langage qui permet d'ajouter des contraintes sur les faits représentés en *RDF*. Il permet en particulier de définir un objet ou un sujet comme une instance de classe. Il permet également d'exprimer les relations d'inclusion qui peuvent exister entre les classes. Enfin, il permet de préciser la sémantique des propriétés en indiquant les classes sur lesquelles ces propriétés sont applicables.

Les contraintes *RDFS* sont également représentées sous la forme de triplets *RDF* en utilisant quatre propriétés spécifiques ayant chacune une signification particulière: `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain` et `rdfs:range`.

La propriété `rdfs:subClassOf` est utilisée pour spécifier qu'une classe est sous classe d'une autre classe. Cette propriété permet de spécialiser et de désambiguïser l'interprétation des classes dans l'ontologie. Par exemple dans notre ontologie VIDECOM, on différencie les évènements qui correspondent aux appels système de ceux qui correspondent aux changements de contexte en créant des sous classes de la classe `:Event`.

```
(:SystemCall rdfs:subClassOf :Event)
(:ContextSwitch rdfs:subClassOf :Event)
```

De même, la propriété `rdfs:subPropertyOf` sert à structurer les relations entre les propriétés. Ainsi, le triplet *RDF* suivant indique que la propriété `:isExecutedOnCPU` spécialise la propriété `:isExecutedOn` pour préciser que l'évènement s'exécute sur un processeur.

```
(:isExecutedOnCPU rdfs:subPropertyOf :isExecutedOn)
```

Une propriété peut être vue comme une fonction mettant en correspondance un sujet `s` avec un ensemble d'objet `o`. Cette vue fonctionnelle permet de voir la classe du sujet comme l'ensemble de départ (*domain*) et celle de l'objet comme l'ensemble d'arrivée (*range*). C'est ainsi que les propriétés `rdfs:domain` et `rdfs:range` permettent d'indiquer les classes de départ et les classes d'arrivée d'une propriété. Par exemple, les triplets suivants permettent de préciser que la propriété `:isExecutedOnCPU` porte sur des instances des classes `:Event` (comme sujet) et de la classe `:CPU` (comme objet).

```
(:isExecutedOnCPU rdfs:domain :Event)
(:isExecutedOnCPU rdfs:range :CPU)
```

Le langage *OWL* apporte des propriétés dont les sémantiques sont plus riches que *RDFS*. Toutefois, dans le contexte des ontologies du domaine, les langages *RDF* et *RDFS* permettent déjà de construire les concepts de base du domaine d'intérêt. Nous allons illustrer cela dans la prochaine section en construisant l'ontologie VIDECOM .

3.4 L'ontologie VIDECOM

3.4.1 Les classes

La classe principale de VIDECOM est la classe `:Event` qui représente un évènement. Cette classe peut être spécialisée en plusieurs sous-classes qui correspondent aux types d'évènements qui peuvent apparaître dans une trace. Par exemple, les appels systèmes et les changements de contexte dans la trace de la Table 3.1.

:SystemCall	rdfs:subClassOf	:Event
:ContextSwitch	rdfs:subClassOf	:Event

La classe `:Component` représente les différents composants qui interviennent pendant l'exécution d'un évènement. Elle a deux sous-classes principales: `:Operation` pour les types d'opérations produits dans la trace. Par exemple `sys_read`, `sys_write` et `switch_to`. `:Location` pour les lieux où s'exécute un évènement, par exemple le CPU ou la tâche en cours d'exécution.

:Location	rdfs:subClassOf	:Component
:CPU	rdfs:subClassOf	:Location

La classe `:Semantic` représente les informations sémantiques. Elle a deux sous classes principales qui sont `:Functionality` pour les fonctionnalités et `:Anomaly` pour les anomalies.

:Functionality	rdfs:subClassOf	:Semantic
:Anomaly	rdfs:subClassOf	:Semantic

3.4.2 Les propriétés

La propriété `:eventHas` permet de représenter des valeurs spécifiques liées à un évènement. Les sous propriétés de `:eventHas` permettent de préciser plusieurs types d'informations sur un évènement. Par exemple, son temps de début (`:eventStartAt`), son temps de fin (`:eventEndAt`), sa durée (`:eventHasDuration`).

:eventHas	rdfs:domain	:Event
:eventHas	rdfs:range	xsd:double
:eventStartAt	rdfs:subPropertyOf	:eventHas
:eventEndAt	rdfs:subPropertyOf	:eventHas
:eventhasDuration	rdfs:subPropertyOf	:eventHas

La propriété `:eventIsExecutedOn` sert à localiser l'exécution d'un évènement. Elle peut être spécialisée, par exemple la sous propriété `:eventIsExecutedOnTask` précise la tâche dans laquelle se produit l'exécution de l'évènement.

:eventIsExecutedOn	rdfs:domain	:Event
:eventIsExecutedOn	rdfs:range	:Location
:eventIsExecutedOnTask	rdfs:subPropertyOf	:eventIsExecutedOn

L'ordre entre les évènements est représenté par la propriété `:eventOrdering`. Les sous propriétés de `:eventOrdering`, telles que `:eventPrecede`, permettent de préciser différents types d'ordre temporels entre les évènements. Elles sont inspirées des relations temporelles introduites par *James Allen* [All83].

Le lieu d'exécution des évènements peut également être pris en compte dans ces relations d'ordre. On peut par exemple s'intéresser à l'ordre des évènements dans le fil d'exécution d'une tâche (`:eventPrecedeInTask`).

:eventOrdering	rdfs:domain	:Event
:eventOrdering	rdfs:range	:Event
:eventPrecede	rdfs:subPropertyOf	:eventOrdering
:eventFollow	rdfs:subPropertyOf	:eventOrdering
:eventOverlap	rdfs:subPropertyOf	:eventOrdering
:eventContains	rdfs:subPropertyOf	:eventOrdering
:eventStart	rdfs:subPropertyOf	:eventOrdering
:eventFinishedBy	rdfs:subPropertyOf	:eventOrdering
:eventMeets	rdfs:subPropertyOf	:eventOrdering
:eventEqual	rdfs:subPropertyOf	:eventOrdering
:eventStartedBy	rdfs:subPropertyOf	:eventOrdering
:eventFinishes	rdfs:subPropertyOf	:eventOrdering
:eventMetBy	rdfs:subPropertyOf	:eventOrdering
:eventPrecedeInTask	rdfs:subPropertyOf	:eventOrdering

La propriété `:requestComponent` sert à relier un évènement aux différents composants qui interviennent au cours de son exécution.

:requestComponent:	rdfs:domain	:Event
:requestComponent:	rdfs:range	:Component

La propriété `:running` permet de préciser l'opération exécutée par l'évènement. Par exemple, les sous propriétés `:runningContextSwitch` et `:runningSystemCall` permettent de distinguer l'exécution d'un appel système d'un changement de contexte.

Les faits de l'ontologie peuvent être représentés sous la forme d'un ensemble de triplets, d'une table ou d'un graphe *RDF*. La Figure 3.1 représente l'ontologie VIDECOM sous la forme de graphe *RDF*, Les nœuds en pointillé représentent les littéraux (dans ce cas des entiers) et ceux en trait plein représentent les instances des classes ou les classes. Chaque arc représente un triplet où le nœud sortant est le sujet, le nœud entrant est l'objet et l'arc représente la propriété.

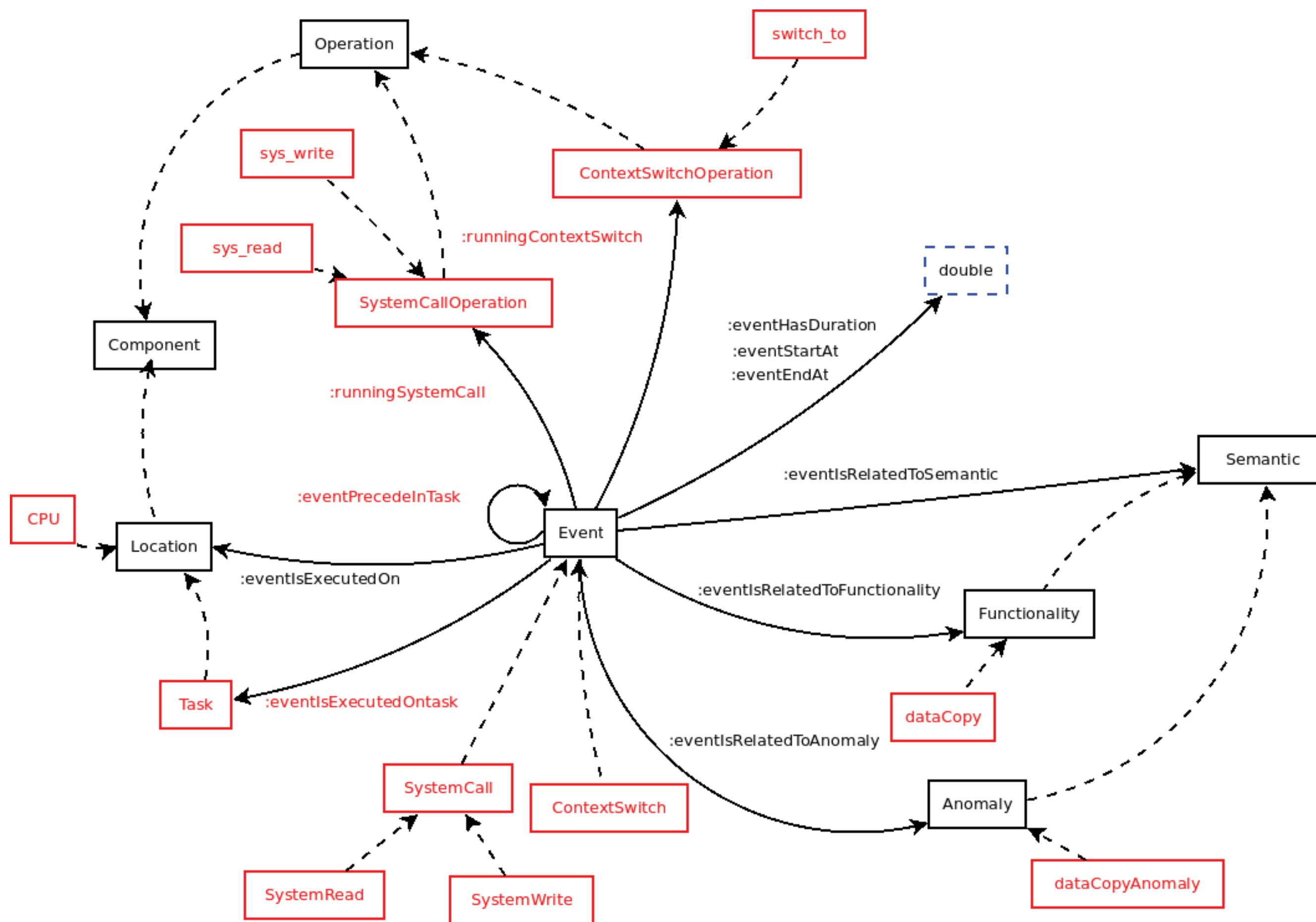


Figure 3.1: Extrait des classes et propriétés de VIDECOM pour l'analyse de la trace exemple

les classes et les propriétés en rouge correspondent aux concepts spécifiques au contexte d'analyse de la trace exemple. Elles sont dérivées sous la forme de sous propriétés ou de sous classes à partir des classes et propriétés principales de VIDECOM (qui correspondent aux classes et propriétés en noir dans la Figure 3.1). Les arcs en pointillé représentent la propriété `rdfs:subClassOf`.

3.4.3 Les instances

Les instances dans VIDECOM représentent les informations contenues dans la trace, elles sont également représenté sous la forme de triplets *RDF*. Les instances de VIDECOM sont construites automatiquement.

Les instances des informations explicites

Les instances qui correspondent aux informations explicites dans la trace sont construites à partir de chaque ligne de la trace en utilisant un *parser*. Le *parser* exploite le format de représentation des informations dans chaque lignes de la trace pour produire identifier les différentes informations portées par la ligne. Des triplets *RDF* sont produits par le parser pour représenter les classes et propriétés identifiées dans chaque ligne. L'ensemble de triplets obtenu forme les *triplets initiaux* et représentent les informations explicites dans la trace.

La Table 3.4 représente les triplets initiaux issus des 5 évènements de la trace de la Table 3.1. Les triplets 1-5, 6-10, 11-15 et 16-20 représentent respectivement les temps de début, les durées, les processeurs et les tâches de chaque évènement. Les triplets 21-24 (respectivement 25-30) représentent les différentes opérations exécutées par chaque évènement (respectivement les arguments de chaque évènement).

Les instances des informations implicites

Les instances qui correspondent aux informations implicites sont automatiquement construites par des moteurs d'inférence à partir des contraintes du domaine exprimées sous la forme de règles d'inférence. Dans la prochaine section nous allons identifier les différentes contraintes dans VIDECOM et nous allons montrer comment elles sont exprimées afin d'être exploitées par les moteurs d'inférence. Dans le chapitre 5 nous allons présenter les moteurs d'inférence plus en détails en nous intéressant particulièrement aux mécanismes qu'ils mettent en œuvre pour automatiquement construire les instances à partir des règles d'inférence.

3.4.4 Les contraintes

L'ontologie supporte une panoplie de contraintes qui permettent de préciser la sémantique des relations entre les instances des classes. Par exemple, `(:e1 :isExecutedOnCPU :cpu0)` nous apporte la connaissance implicite que `:e1` est une instance de la classe `:Event`. Un triplet *RDF* `(s P o)` est interprété en logique du premier ordre comme une formule atomique `P(s,o)`, où `P` est le prédicat et *s* et *o* sont des variables. Ainsi, le triplet `(:e1 :isExecutedOnCPU :cpu0)` qui traduit le fait *"qu'il existe un évènement **e1** et un processeur **cpu0** tels que **e1** s'exécute sur **cpu0**"*, s'exprime par la formule de la logique du premier ordre suivante:

#	s	p	o
1	:event1	:eventStartAt	525319
2	:event2	:eventStartAt	525323
3	:event3	:eventStartAt	525323
4	:event4	:eventStartAt	525327
5	:event5	:eventStartAt	525333
6	:event1	:eventHasDuration	14
7	:event2	:eventHasDuration	0
8	:event3	:eventHasDuration	4
9	:event4	:eventHasDuration	6
10	:event5	:eventHasDuration	20
11	:event1	:eventIsExecutedOnCPU	:cpu0
12	:event2	:eventIsExecutedOnCPU	:cpu1
13	:event3	:eventIsExecutedOnCPU	:cpu0
14	:event4	:eventIsExecutedOnCPU	:cpu0
15	:event5	:eventIsExecutedOnCPU	:cpu0
16	:event1	:eventIsExecutedOnTask	:copyMovie
17	:event2	:eventIsExecutedOnTask	:idle
18	:event3	:eventIsExecutedOnTask	:copyMovie
19	:event4	:eventIsExecutedOnTask	:copyMovie
20	:event5	:eventIsExecutedOnTask	:copyMovie
21	:event1	:runningSystemRead	:sys_read0
22	:event3	:runningSystemWrite	:sys_write0
23	:event4	:runningSystemRead	:sys_read0
24	:event5	:runningSystemWrite	:sys_write0
25	:event1	:eventHasArgument	29
27	:event2	:eventHasArgument	:sshd
28	:event3	:eventHasArgument	29
29	:event4	:eventHasArgument	96
30	:event5	:eventHasArgument	56

Table 3.4: Ensemble des triplets initiaux produits par la trace exemple

$$\exists \text{ e1 } \exists \text{ cpu0 } [\text{ isExecutedOnCPU(e1, cpu0) }]$$

Les contraintes imposées par les propriétés peuvent également être interprétées comme des formules de la logique du premier ordre. Par exemple, la contrainte sur les classes des instances dans la propriété `:isExecutedOnCPU` traduit la contrainte que *"si `e1` s'exécute sur `cpu0` alors `e1` est une instance de la classe `Event`"*. cette contrainte s'exprime par la formule logique suivante:

$$\forall \text{ e1 } \forall \text{ cpu0 } [\text{ isExecutedOnCPU(e1, cpu0)} \Longrightarrow \text{rdf_type(e1, Event) }]$$

Les formules logiques du premier ordre de la forme $\forall$...[.... $\Longrightarrow$ ...] sont appelées des règles d'inférence. Elles servent à inférer les connaissances à droite de la flèche chaque fois que les connaissances à gauche sont vérifiées. Dans le formalisme *RDF*, ces règles d'inférence peuvent être représentées par un tableau à deux colonnes. La première colonne représente une conjonction de motif et la deuxième colonne représente l'ensemble des triplets à inférer si la conjonction retourne au moins un résultat sur la base de triplets.

Si	Alors
`?e1 :isExecutedOnCPU ?cpu0 .`	`?e1 rdf:type :Event`

Il existe plusieurs types de contraintes. On distingue les contraintes dites métier qui sont spécifiques au domaine de l'analyse de traces. Par exemple, la contrainte de correspondance entre la taille des données lues et écrites par l'application `copyMovie`. Ainsi que les contraintes imposées par la sémantique du langage *RDFS*. Toutes ces contraintes sont exprimées dans l'ontologie sous la forme de règles d'inférence.

Les contraintes métier

Les connaissance métier servent à exprimer plusieurs types de contraintes et de nouvelles informations. Nous allons illustrer quelques connaissances métier sous forme de règles d'inférence.

La première information implicite dans la trace de la Table 3.1 est le temps de fin d'un évènement. Cette information est importante car elle sert également à déterminer l'ordre temporel entre les évènements. D'après la connaissance métier, le temps de fin d'un évènement est égale à la somme de son temps de début et de sa durée d'exécution. La règle RM_1 ci-dessous traduit cette connaissance métier sous la forme d'une règle d'inférence.

	Si	Alors
RM_1	`?e :eventStartAt ?s .` `?e :eventHasDuration ?d`	`?e :eventEndAt (?s + ?d)`

L'application de la règle RM_1 aux triplets de la Table 3.4 infère les triplets suivants qui sont rajoutés à la base de données (triplets 31-35).

#	s	p	o
31	:event1	:eventEndAt	525337
32	:event2	:eventEndAt	525341
33	:event3	:eventEndAt	525341
34	:event4	:eventEndAt	525347
35	:event5	:eventEndAt	525367

Grâce à la connaissance des temps de début et de fin de chaque évènement, l'ordre temporel entre les évènements peut être inféré. La règle RM_2 ci-dessous exploite la connaissance métier sur l'ordre temporel, telle que introduit par *Allen et al* [All83], pour inférer l'ordre de précédence entre les évènements dans la trace. Ainsi, l'évènement `?a` précède l'évènement `?b` s'il se termine avant que ne commence `?b`. En ajoutant la contrainte que `?a` et `?b` s'exécutent sur la même tâche, la règle RM_3 spécialise RM_2 pour inférer l'ordre de précédence dans le fil d'exécution d'une tâche.

	Si	Alors
RM_2	?a :eventEndAt ?endA . ?b :eventStartAt ?startB . ?endA < ?startB	?a :eventPrecede ?b
RM_3	?a :eventEndAt ?endA . ?b :eventStartAt ?startB . ?endA < ?startB . ?a :eventIsExecutedOnTask ?task . ?b :eventIsExecutedOnTask ?task	?a :eventPrecedeInTask ?b

Table 3.5: Règles d'inférence de la précédence entre les évènements

Grâce à la présence des triplets 31-35 de la base de données, la règle RM_3 infère les nouveaux triplets 36 - 41 suivants:

#	s	p	o
36	:event1	:eventPrecedeInTask	:event3
37	:event1	:eventPrecedeInTask	:event4
38	:event1	:eventPrecedeInTask	:event5
39	:event3	:eventPrecedeInTask	:event4
40	:event3	:eventPrecedeInTask	:event5
41	:event4	:eventPrecedeInTask	:event5

La règle d'inférence RM_4 (respectivement RM_5) décrit la connaissance métier que l'argument d'un événement de type appel système de lecture (respectivement écriture) de données, correspond à la taille des données lues (respectivement écrites).

	Si	Alors
RM_4	?e :runningSystemRead ?syscall . ?e :eventHasArgument ?data	?e :eventReadData ?data
RM_5	?e :runningSystemWrite ?syscall . ?e :eventHasArgument ?data	?e :eventWriteData ?data

Les règles RM_4 et RM_5 infèrent concepts métier (`:eventReadData` et `:eventWriteData`) qui correspondent les tailles des données lues et écrites par les appels systèmes (triplets 42-45).

#	s	p	o
42	:event1	:eventReadData	29
43	:event3	:eventWriteData	29
44	:event4	:eventReadData	96
45	:event5	:eventWriteData	56

Grâce à ces nouveaux triplets, l'analyste peut désormais exprimer sa connaissance métier sur la fonctionnalité de copie de données dans `copyMovie`.

	Si	Alors
RM_6	?a :eventIsExecutedOnTask :copyMovie . ?a :eventStartAt ?start . ?a :eventPrecedeInTask ?b . ?a :eventReadData ?data . ?b :eventWriteData ?data . ?b :eventEndAt ?end	_:dc rdf:type :dataCopy _:bl :eventIsRelatedToFunctionality _:dc _:bl :eventStartAt ?start _:bl :eventEndAt ?end

La règle RM_6 recherche tous les appels systèmes de lecture de données (`?a`) de la tâche `copyMovie`, qui précèdent des appels systèmes d'écriture de données de la même taille. En cas de succès, la règle infère une nouvelle instance d'évènement (`_:bl`) rattachée à une nouvelle instance (`_:dc`) de la fonctionnalité `:dataCopy` avec les temps de début et de fin qui correspondent aux appels systèmes identifiés.

La règle peut être adaptée pour inférer les cas où les tailles de données lues et écrites sont différentes. La règle RM_7 infère une nouvelle instance d'évènement rattachée cette fois à une instance de l'anomalie identifiée par (*_dca*) qui est une instance de la classe `:dataCopyAnomaly`.

	Si	Alors
RM_7	?a :eventIsExecutedOnTask :copyMovie . ?a :eventStartAt ?start . ?a :eventPrecedeInTask ?b . ?a :eventReadData ?data1 . ?b :eventWriteData ?data2 . ?b :eventEndAt ?end ?data1 != ?data2	_:dca rdf:type :dataCopyAnomaly _:bl :eventIsRelatedToAnomaly _:dca _:bl :eventStartAt ?start _:bl :eventEndAt ?end

Les nœuds blancs dans la partie conclusion de la règle permettent de créer de nouvelles URI en garantissant leur unicité (par rapport aux autres URI de la base de connaissances). Ainsi, grâce aux nœuds blancs `_:bl,` `_:dca` et `_:dc`, les règles RM_6 et RM_7 inférent de nouvelles instances qui correspondent aux fonctionnalités de copie de données ainsi qu'aux anomalies de copie de données (triplets 46-53).

#	s	p	o
46	_:BTH0	rdf:type	:dataCopy
47	_:BXN0	:eventIsRelatedToFunctionality	_:BTH0
48	_:BXN0	:eventStartAt	525323
49	_:BXN0	:eventEndAt	525341
50	_:BTH1	rdf:type	:dataCopyAnomaly
51	_:BNX1	:eventIsRelatedToAnomaly	_:BTH1
52	_:BNX1	:eventStartAt	525327
53	_:BNX1	:eventEndAt	525367

L'expansion infinie

Considérons l'exemple des deux règles d'inférence dépendantes suivant. R_1 infère des triplets liés à `prop_2` à partir des triplets liés à `prop_1`, tandis que R_2 fait l'inverse.

	Si	Alors
R_1	?a prop_1 ?b .	?b :prop_2 _:b
R_2	?a prop_2 ?b .	?b :prop_1 ?a

Toutefois, à cause du nœud blanc (`_:b`) dans R_1, l'inférence de cette règle présente un cas d'*expansion infini*, car les triplets inférés par l'une des règles entraînent l'inférence de nouveaux triplets par l'autre règle et vice-versa. Dans notre cas, nous imposons que les règles métier ne soient indépendantes et non-récursives pour éviter de créer ce type de situation.

Dans la prochaine section, nous allons présenter les règles d'inférence qui proviennent des contraintes des propriétés *RDFS*.

Les règles d'inférence RDFS

Les règles d'inférence permettent également d'exprimer les contraintes impliquées par la sémantique des propriétés *RDFS*. Toutes les contraintes *RDFS* sont collectées sous la forme de 14 règles d'inférence [HPS14]. La Table 3.6 donne la liste des règles d'inférence qui correspondent respectivement à la sémantique des propriétés `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf` et `rdfs:subPropertyOf`.

	Si conjonction	Alors inférer
$RDFS_2$	?a rdfs:domain ?x . ?y ?a ?z .	?y rdf:type ?x
$RDFS_3$	?a rdfs:range ?x . ?y ?a ?z .	?z rdf:type ?x
$RDFS_7$	?a rdfs:subPropertyOf ?b . ?x ?a ?y .	?x ?b ?y
$RDFS_9$	?x rdfs:subClassOf ?y . ?z rdf:type ?x .	?z rdf:type ?y

Table 3.6: Règles d'inférence des quatre propriétés RDFS

Ces règles d'inférence servent à inférer les triplets pour préciser les types des instances. L'application de ces règles à notre base de connaissances permet d'inférer les triplets 54-66.

#	s	p	o
54	:event1	rdf:type	:SystemCall
55	:event2	rdf:type	:ContextSwitch
56	:event3	rdf:type	:SystemCall
57	:event4	rdf:type	:SystemCall
58	:event5	rdf:type	:SystemCall
59	_:BTH0	rdf:type	:Functionality
60	_:BTH1	rdf:type	:Anomaly
61	_:BXN0	rdf:type	:Event
62	_:BXN1	rdf:type	:Event
63	:cpu0	rdf:type	:CPU
64	:cpu1	rdf:type	:CPU
65	:idle	rdf:type	:Task
66	:copyMovie	rdf:type	:Task

L'expression des règles d'inférence

Il existe plusieurs langages pour exprimer les règles d'inférence dans les moteurs d'inférence. *SWRL* (Semantic Web Rule Language) est un langage de règle d'inférence issu du Web Sémantique [IHD04]. Comme nous le verrons dans le chapitre 5, *SWRL* utilise une syntaxe proche de celle de la logique du premier ordre pour exprimer les règles d'inférence. Il propose des prédicats spécialisés pour les fonctions telles que la comparaison (*equal, greaterThan, lessThan*, ...) ou les opérations arithmetiques (*add, substract, multiply*, ...). Par exemple, les règles *SWRL* suivantes correspondent aux règles d'inférence RM_1 et RM_2.

RM_1 ::= `eventStartAt(?a,?s), eventHasDuration(?a,?d), add(?s,?d,?e)`
`-> eventEndAt(?a,?e)`

RM_2 ::= `eventEndtAt(?a,?e), eventStartAt(?b,?s), lessThan(?e,?s)`
`-> eventPrecede(?a,?b)`

Conclusion

En conclusion de cette section, nous retenons que VIDECOM est une ontologie *RDFS* qui contient des hiérarchies de classes et de propriétés. Ces hiérarchies sont extensible en fonction des concepts propres à chaque trace à analyser. Les instances de ces classes et de ces propriétés sont automatiquement construites, soit par un parser à partir des lignes de trace (pour les instances des informations explicites) ou par les moteurs d'inférence à partir des règles d'inférence qui traduisent les contraintes métier de l'analyse de trace (pour les informations implicites). L'ontologie VIDECOM est disponible sur le site web `http://videcom.imag.fr` et contient actuellement 607 classes et 176 propriétés. Dans la prochaine section nous allons nous interresser à l'interrogation des ontologies et plus particulièrement à l'interrogation de VIDECOM pour l'analyse de traces.

3.5 L'interrogation d'une ontologie

Une ontologie sert avant tout à accéder ou à rechercher des informations sur les connaissances. Les motifs de triplets constituent un moyen simple d'accéder à ces triplets en exploitant la structure *RDF*.

3.5.1 Les motifs de triplets

Soit $\mathcal{U}$ l'ensemble, potentiellement infini, des URI qui identifient les resources d'une ontologie, et $\mathcal{V}$ un ensemble de variables (notées avec un "?") toutes distinctes de $\mathcal{U}$. Un *motif de triplet* (ou plus simplement un *motif*) est un élément de $(\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V})$. En d'autres termes, il s'agit d'un triplet *RDF* pouvant contenir des variables aux différentes positions de *sujet*, *propriété* ou *objet*. `(?e :eventIsExecutedOnTask :copyMovie)` est un exemple de motif contenant une seule variable `?e`.

Un triplet *(s p o)* avec $s, p, o \in \mathcal{U}$ *respecte* un *motif* $(s'\ p'\ o')$ avec $s', p', o' \in (\mathcal{U} \cup \mathcal{V})$ si $(s = s') \wedge (p = p') \wedge (o = o')$ avec $\forall u, v \mid u \in \mathcal{U},\ v \in \mathcal{V}\ u = v$ toujours *vraie*. Par exemple le triplet `(:event1 :eventIsExecutedOnTask :copyMovie)` respecte le *motif* précédent (car la propriété et l'objet du triplets sont identique aux ceux du motif et `(?e = :event1)`). Par contre le triplet `(:event2 :eventIsExecutedOnTask :idle)` ne respecte pas le motif (car l'objet du triplet est différent de l'objet du motif `(:copyMovie != :idle)`).

L'évaluation d'un *motif* sur une base de connaissances retourne l'ensemble des triplets qui respectent le *motif*. Par exemple, l'évaluation du motif `(?e :eventIsExecutedOnTask :copyMovie)` sur les triplets de la Table 3.4, retourne les triplets 16,18,19 et 20. De même, l'évaluation du motif `(:event1 ?p ?o)` retournera tous les triplets ayant `:event1` comme sujet (c'est à dire les triplets 1,6,11,16,21 et 25).

Etant données deux motifs $(s'_1\ p'_1\ o'_1)$ et $(s'_1\ p'_1\ o'_1)$, une *conjonction* de motif, notée $(s'_1\ p'_1\ o'_1)\ .\ (s'_2\ p'_2\ o'_2)$ permet de faire des *jointures* entre les variables des motifs. En fonction des positions des variables on distingue six types de jointures *sujet-sujet*, *sujet-propriété*, *sujet-objet*, *propriété-propriété*, *propriété-objet* et *objet-objet*. Par exemple, la conjonction suivante fait la jointure *sujet-sujet* sur les valeurs de la variable `?e` pour obtenir les processeurs sur lesquels s'exécutent tous les évènements produit dans la tâche `:copyMovie`.

```
(?e :eventIsExecutedOnTask :copyMovie) .
    (?e :eventIsExecutedOnCPU ?cpu)
```

Les motifs de triplets sont utilisés dans plusieurs langages d'interrogation des ontologies. Dans la prochaine section nous allons présenter *SPARQL* qui est le langage d'interrogation de triplets *RDF*.

3.5.2 SPARQL

SPARQL est un langage déclaratif de requêtes destiné à interroger les bases de connaissances en *RDF*. Ce langage est standardisé en deux versions par le

Wold Wide Web Consortium : *SPARQL 1.0* [W3C08] et *SPARQL 1.1* [W3C13]. *SPARQL* est pour le *RDF* ce qu'est *SQL* pour les bases de données. Il fonctionne avec une structure proche de celle du *RDF* et exploite les motifs de triplets. Nous nous limiterons à présenter la forme des requêtes `SELECT`, `CONSTRUCT`, la clause `FILTER` et les opérateurs d'agrégation.

Les requêtes SELECT

La forme la plus simple d'une requête *SPARQL* est la requête de sélection `SELECT`. Elle consiste en une clause `SELECT` contenant un ensemble de variables, et une clause `WHERE` contenant des conjonctions de motifs.

La requête *SPARQL* suivante retourne les durées d'exécution et les URI des évènements qui s'exécutent dans la tâche `copyMovie`. La clause `PREFIX` sert à indiquer les noms de domaines.

```
PREFIX videcom:  <http://videcom.imag.fr/vd.rdf>

SELECT ?event ?duration
WHERE {
    ?event videcom:eventIsExecutedOnTask :copyMovie .
    ?event videcom:eventHasDuration ?duration
}
```

L'exécution de la requête sur la base de données de la Table 3.4 retourne le résultat suivant:

?event	?duration
<http://videcom.imag.fr/vd.rdf#event1>	14
<http://videcom.imag.fr/vd.rdf#event3>	4
<http://videcom.imag.fr/vd.rdf#event4>	6
<http://videcom.imag.fr/vd.rdf#event5>	20

La clause FILTER et les opérateurs d'agrégation

De même que le *SQL*, *SPARQL* permet de filtrer les résultats qui respectent une expression booléenne à l'aide de l'opérateur `FILTER`. La requête suivante liste uniquement les évènements de `copyMovie` ayant des durées d'exécution supérieures à 10.

```
PREFIX videcom:  <http://videcom.imag.fr/vd.rdf>

SELECT ?event ?duration
WHERE {
    ?event videcom:eventIsExecutedOnTask :copyMovie .
    ?event videcom:eventHasDuration ?duration
    FILTER (?duration > 10)
}
```

La requête produit le résultat suivant sur la base de connaissances de la Table 3.4

?event	?duration
<http://videcom.imag.fr/vd.rdf#event1>	14
<http://videcom.imag.fr/vd.rdf#event5>	20

La version *SPARQL 1.1* propose des opérateurs d'agrégation tels que MIN, MAX, SUM, AVG. Ces opérateurs ont la même sémantique que leur équivalent en *SQL*. Par exemple, la requête suivante retourne la somme des temps d'exécution des évènements sur chaque processeur.

```
PREFIX videcom:  <http://videcom.imag.fr/vd.rdf>

SELECT ?cpu (SUM(?duration) AS ?workload)
WHERE {
    ?event videcom:eventIsExecutedOnCPU ?cpu .
    ?event videcom:eventHasDuration ?duration .
}
GROUP BY ?cpu
```

Cette requête produit le résultat suivant sur notre base de données.

?cpu	?workload
<http://videcom.imag.fr/vd.rdf#cpu0>	0
<http://videcom.imag.fr/vd.rdf#cpu1>	44

Les requêtes CONSTRUCT

L'idée de la clause CONSTRUCT est de construire des triplets *RDF* à partir des résultats d'une requête SELECT. En effet, les résultats d'une requête *SPARQL* de type SELECT peuvent prendre la forme de triplets *RDF* ce qui permet de les rajouter à la base de connaissances.

La structure des triplets à construire est spécifiée dans la clause CONSTRUCT tandis que la clause WHERE se comporte exactement comme dans le cas d'une requête de type SELECT. Par exemple la requête suivante recherche les évènements ayant une durée d'exécution supérieure à 100 (clause WHERE), et construit les triplets de la clause CONSTRUCT pour chacun des résultats obtenus.

```
PREFIX videcom:  <http://videcom.imag.fr/vd.rdf>

CONSTRUCT { videcom:LongEvent rdfs:subClassOf videcom:Event .
            ?event rdf:type videcom:LongEvent .
}
WHERE {
    ?event videcom:eventHasDuration ?duration .
    FILTER (?duration > 100)
}
```

Dans la requête précédente, les triplets résultat représentent les évènements résultats comme des instances d'une nouvelle classe videcom:EventLong. En conséquence, l'ajout de ces triplets à la base de triplets permet d'enrichir l'ontologie. Toutefois, nous montrerons dans le Chapitre 5 que ce type d'enrichissement est limité. Dans la prochaine section nous allons présenter quelques requêtes *SPARQL* pour interroger VIDECOM dans différentes situations d'analyse de traces.

3.6 Illustration de l’interrogation de VIDECOM pour l’analyse de traces

Considérons un scénario d’analyse de la trace 3.1 dans lequel le devéleppeur veut détecter les éventuels cas de pertes de données lors de l’enregistrement des vidéos par la tâche `copyMovie`. Ces perte de données peuvant conduire à une mauvaise qualité de la vidéo. L’objectif dans ce cas est d’identifier les zones dans la trace pouvant produire cette perte de données. Sachant que les lectures et les écritures de données dans `copyMovie` se font par les appels systèmes, l’idée est donc de vérifier que tous ces appels systèmes respectent la contrainte fonctionnelle de l’application.

La première requête *SPARQL* retourne tous les appels systèmes exécutés par le programme `copyMovie` ainsi que l’argument contenu dans l’évènement.

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
PREFIX videcom:  <http://videcom.imag.fr/vd.rdf>
SELECT ?event ?arg
WHERE {
    ?event rdf:type videcom:SystemCall .
    ?event videcom:eventIsExecutedOnTask videcom:copyMovie .
    ?event videcom:eventHasArgument ?arg .
}
```

En se servant de la connaissance métier que les arguments des appels systèmes correspondent aux tailles des données lues ou écrites et que la lecture et l’écriture des données sont des évènements consécutifs dans la tâche `copyMovie`, l’analyste peut affiner la requête pour distinguer ces deux informations. La requête suivante distingue les évènements de lecture (`?read`) et les évènements d’écriture (`?write`) consécutifs dans la tâche `copyMovie` (`:eventPrecedeInTask`).

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
PREFIX videcom:  <http://videcom.imag.fr/vd.rdf>
SELECT ?read ?data1 ?write ?data2
WHERE {
    ?read  rdf:type videcom:SystemRead .
    ?write rdf:type videcom:SystemWrite .
    ?read  videcom:eventIsExecutedOnTask videcom:copyMovie .
    ?write videcom:eventIsExecutedOnTask videcom:copyMovie .
    ?read  videcom:eventPrecedeInTask ?write .
    ?read  videcom:eventReadData ?data1 .
    ?write videcom:eventWriteData ?data2 .
}
```

En filtrant les cas où les tailles de données lues et écrites sont différents, l’analyste obtient une première vue sur les zones de trace où la contrainte fonctionnelle de l’application `copyMovie` n’est pas respectée.

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
PREFIX videcom:  <http://videcom.imag.fr/vd.rdf>
SELECT ?read ?data1 ?write ?data2
WHERE {
    ?read  rdf:type videcom:SystemRead .
    ?write rdf:type videcom:SystemWrite .
    ?read  videcom:eventIsExecutedOnTask videcom:copyMovie .
    ?write videcom:eventIsExecutedOnTask videcom:copyMovie .
    ?read  videcom:eventPrecedeInTask ?write .
    ?read  videcom:eventReadData ?data1 .
    ?write videcom:eventWriteData ?data2 .
    FILTER ( ?data1 != ?data2 )
}
```

Grâce à la sémantique des concepts de l'ontologie, ces résultats peuvent être retrouvés en utilisant des requêtes plus simples. Par exemple, sachant que le concept `:dataCopyAnomaly` correspond au type d'anomalie que nous avons identifié avec la requête précédente, une requête plus simple pour accéder aux zones d'anomalies est la suivante:

```
PREFIX videcom:  <http://videcom.imag.fr/vd.rdf>
SELECT ?start ?end
WHERE {
    ?anomaly rdf:type videcom:dataCopyAnomaly .
    ?slice   videcom:eventIsRelatedToAnomaly ?anomaly .
    ?slice   videcom:eventStartAt ?start .
    ?slice   videcom:eventEndAt ?end .
}
```

Grâce aux propriétés `:eventEndAt` et `:eventIsRelatedToAnomaly`, inférées respectivement par les règles d'inférence métier RM_1 et RM_7, la requête ci-dessus retournera le résultat suivant qui représentent un cas de perte de données dans la trace.

?start	?end
525341	525367

Cette réquête illustre la nécessité de l'inférence des instances des informations implicites par les moteurs d'inférence avant l'interrogation de l'ontologie. Dans le cas contraire, toutes les requêtes qui intérrogent ces instances ne retourneront pas de résultats compléts.

3.7 Conclusion

L'objectif de ce chapitre était de définir ce que sont les ontologies et de montrer leur applicabilité dans le contexte de l'analyse des trace d'application multimédia sur MPSoC. Nous avons vu les différentes approches de construction des ontologies, ainsi que les différents langages de construction (*RDF* et *RDFS*) et d'interrogation des réquêtes (*SPARQL*). Nous avons également vu que les ontologies, en tant que base de connaissances, utilisent les règles d'inférence exprimées dans des langages tels que *SWRL*, pour raisonner sur les connaissances

du domaine. Enfin, nous avons illustré tous ces aspects dans la constrution de VIDECOM, notre ontologie du domaine de l'analyse des traces d'applications multimédia sur MPSoC.

Dans les deux prochains chapitres, nous allons rechercher le système de gestion des ontologies (encore appelé *triplestore*) qui est le plus adapté à l'analyse des traces via notre ontologie VIDECOM. Nous allons particulièrement nous intéresser aux performances du chargement des triplets, de l'interrogation des triplets et de l'inférence des règles métier qu'offre un tel système.

Ainsi, le prochain chapitre est une étude comparative des performances des triplestores pour le chargement et l'interrogation des connaissances (Chapitre 4). Le chapitre présente les différentes approches de stockage des triplets dans les triplestores et illustre leur impact sur l'interrogation de ces triplets. Ce chapitre est suivi par un chapitre sur l'inférence des règles métier dans les bases de connaissances (Chapitre 5). Dans ce chapitre 5, nous allons introduire les approches d'inférence utilisées par les moteurs de règles. Nous allons également voir comment les limites telles que l'inférence des nœuds blancs dans les règles métier et le passage à l'échelle du raisonnement sur les gros volumes de triplets, sont gérées par ces moteurs d'inférence.

Chapitre 4

Le benchmark des performances de chargement et d'interrogation de VIDECOM

Sommaire

4.1 Le stockage des données *RDF* 67
4.1.1 La stratégie native 68
4.1.2 La stratégie *multi-index* 71
4.1.3 Les stratégies en bases de données 74
4.1.4 Conclusion 78
4.2 Passage à l'échelle pour l'interrogation de VIDECOM 79
4.2.1 Le contexte expérimental 79
4.2.2 Le chargement des triplets 80
4.2.3 Les scénarii d'analyse des traces d'exécution 82
4.3 Conclusion 93

Dans ce chapitre nous allons nous intéresser aux architectures des systèmes de gestion des ontologies et en particulier aux différentes approches de stockage des données *RDF* que proposent ces systèmes. L'objectif du chapitre est de déterminer les caractéristiques de ces architectures qui soient les plus adaptées à l'exploitation efficace de l'ontologie VIDECOM dans le contexte de l'analyse de traces réelles.

Introduction

De manière formelle, un triplet `(s, p, o)` est interprété comme un arc `p` (*ou statement*) entre le sujet `s` et l'objet `o`. Les triplets sont des éléments de type $(\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ où $\mathcal{U}$, $\mathcal{B}$ et $\mathcal{L}$ sont respectivement des ensembles

potentiellement infinis d'URI, de nœuds blancs et de valeurs terminales. Ainsi l'ensemble des triplets d'une base constitue un graphe orienté $G = (V, E)$ avec $V = \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$ l'ensemble des nœuds et $E = \mathcal{U}$ l'ensemble des arcs entre ces nœuds. En conséquence, l'évaluation d'une requête *SPARQL* sur une base de triplets correspond à un isomorphisme de graphe qui est *NP-complet* [MRP+05].

En fonction des co-occurrences des variables dans les motifs de triplets d'une conjonction, on obtient les jointures SS (sujet-sujet), SP (sujet-propriété), SO (sujet-objet), PP (propriété-propriété), PO (propriété-objet) et OO (objet-objet). On distingue les requêtes dites *statement-search* qui recherchent des triplets qui partagent des mêmes sujets, propriétés ou objets et contiennent des jointures SS, PP ou OO, et les requêtes dites *path-search* qui recherchent des chemins dans le graphe *RDF* en utilisant des jointures SO.

L'exploitation des données *RDF* se fait à l'aide des systèmes de gestion des ontologies encore appelés *triplestores*. Ces triplestores offrent des fonctionnalités d'interrogation, de modification, d'ajout et de suppression des triplets *RDF*.

L'objectif des triplestores est de proposer des stratégies de stockage optimisées pour exploiter efficacement les triplets. Tout au long de ce chapitre nous allons utiliser les requêtes *SPARQL Q1* (*statement-search*) et *Q2* (*path-search*) de la Table 4.2, pour illustrer l'interrogation des triplets de la Table 4.1 qui seront stockés selon diverses stratégies dans les triplestores. Ces triplets illustrent cinq types de structuration de l'information des évènements de traces.

1. les triplets (1-5) illustrent les propriétés qui portent sur des valeurs terminales. Ce type de triplet sert à la recherche des sujets liés à des valeurs terminales précises.

2. les triplets (6-10) illustrent les propriétés qui ont des sujets et des objets qui correspondent à des instances de classes différentes.

3. les triplets (11-14) illustrent les propriétés qui portent sur des sujets et objets qui sont des instances de la même classe et qui forment un chaînage de l'information. Ce type de triplet est à la base des requêtes de type *path-search*.

4. les triplets (15-18) illustrent les propriétés qui s'appliquent sur un nombre limité d'évènements de la trace.

5. les triplets (19-23) illustrent les propriétés qui portent sur des instances de classes mais dont les objets sont hétérogènes (elles peuvent être des valeurs terminales ou des URIs d'autres instances).

Nous allons mesurer l'efficacité d'un triplestore dans l'exploitation de VIDECOM par sa capacité à passer à l'échelle en garantissant des temps de réponse rapides aux requêtes de développeur sur VIDECOM.

La première partie du chapitre est consacrée à la présentation des différentes stratégies de stockage de triplets *RDF* (Section 4.1). Dans cette partie, nous allons présenter les modèles logiques ainsi que les modèles physiques de données

#	sujet	propriété	objet
1	:event1	:eventStartAt	525319
2	:event2	:eventStartAt	525323
3	:event3	:eventStartAt	525323
4	:event4	:eventStartAt	525327
5	:event5	:eventStartAt	525333
6	:event1	:eventIsExecutedOnCPU	:cpu0
7	:event2	:eventIsExecutedOnCPU	:cpu1
8	:event3	:eventIsExecutedOnCPU	:cpu0
9	:event4	:eventIsExecutedOnCPU	:cpu0
10	:event5	:eventIsExecutedOnCPU	:cpu0
11	:event1	:eventPrecede	:event2
12	:event2	:eventPrecede	:event3
13	:event3	:eventPrecede	:event4
14	:event4	:eventPrecede	:event5
15	:event1	:runningSystemRead	:sys_read0
16	:event3	:runningSystemWrite	:sys_write0
17	:event4	:runningSystemRead	:sys_read0
18	:event5	:runningSystemWrite	:sys_write0
19	:event1	:eventHasArgument	29
20	:event2	:eventHasArgument	:sshd
21	:event3	:eventHasArgument	29
22	:event4	:eventHasArgument	96
23	:event5	:eventHasArgument	56

Table 4.1: Base de triplets *RDF* illustratifs

sur lesquels se basent ces stratégies. Dans la deuxième partie du chapitre, nous présentons une étude comparative des performances du stockage et de l'interrogation de VIDECOM pour l'analyse de traces sur les triplestores (Section 4.2). Enfin, dans la dernière section de ce chapitre, nous identifions les triplestores les plus adaptés à l'analyse des traces selon les cas d'usage et nous concluons le chapitre (Section 4.3).

4.1 Le stockage des données *RDF*

Il existe trois stratégies pour le stockage des données *RDF*. La première stratégie, dite *native*, utilise des structures de données spécialisées pour stocker les triplets *RDF* en mémoire dans un modèle proche du modèle de graphe. La seconde stratégie, dite *multi-index*, utilise intensivement des index pour accéder aux triplets stockés sur le disque. Quant à la dernière stratégie, elle utilise les structures classiques, telles que les bases de données relationnelles, pour stocker les triplets *RDF* [DOG12].

Dans les trois prochaines sections, nous allons présenter ces stratégies plus en détail et nous allons donner des exemples de *triplestores* qui sont basés sur chacune d'elle.

Q1: Requête de type *statement-search*

SELECT ?event ?start ?cpu ?arg
WHERE {
  ?event videcom:eventStartAt ?start .
  ?event videcom:eventIsExecutedOnCPU ?cpu .
  ?event videcom:eventHasArgument ?arg .
}

Q2: Requête de type *path-search*

SELECT ?e1 ?e3
WHERE {
  ?e1 videcom:eventPrecede ?e2 .
  ?e2 videcom:eventPrecede ?e3 .
  ?e1 videcom:eventIsExecutedOnCPU ?cpu .
  ?e3 videcom:eventIsExecutedOnCPU ?cpu .
}

Table 4.2: Requêtes SPARQL illustratives

4.1.1 La stratégie native

L'objectif dans cette stratégie est de maintenir la structure de graphe sous-jacent aux triplets *RDF* dans le but de proposer une exploration des données basée sur le modèle logique de graphe. Le graphe *RDF* est généralement entièrement maintenu en mémoire afin d'exécuter efficacement les opérations de parcours de graphe. Plusieurs triplestores utilisent cette stratégie de stockage de données *RDF* selon des approches différentes. Nous allons illustrer les principales approches à travers des triplestores suivants: *Jena2*, *BitMat*, *Sesame*, *swiftOWLIM* et *SpiderStore*.

Le triplestore ***Jena2*** propose une API *Java* appelé *Graph* pour effectuer les opérations usuelles d'ajout, de suppression et de recherche dans les graphes. *Jena2* fournit une vue simplifiée des données qui permet de facilement créer, accéder et manipuler les triplets. *Jena2* représente entièrement le graphe *RDF* en mémoire sous la forme d'une liste ordonnée de graphes optimisés et spécialisés à des types précis de triplets. On distingue par exemple, des graphes optimisés pour les triplets inférés, les graphes optimisés pour les triplets de la *TBox* et d'autres optimisés pour les triplets qui correspondent aux données [WSK+03].

La Figure 4.1 montre les triplets de la Table 4.1 représentés par trois graphes. Le premier graphe permet d'optimiser la manipulation des triplets liées aux propriétés `eventStartAt` et `eventHasArgument`, le second graphe optimise la manipulation des triplets pour les propriétés `runningSystemRead`, `runningSystemWrite` et `eventIsExecutedOnCPU`. Et le dernier graphe optimise la manipulation des triplets pour la propriété `eventPrecede`.

Jena2 exploite ce modèle physique de données pour repondre rapidement aux requêtes. En effet, il interroge chaque graphe pour constituer le résultat

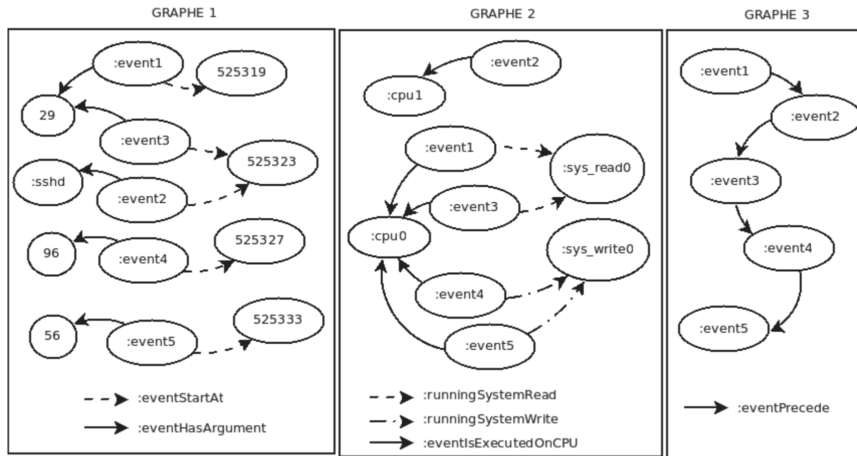


Figure 4.1: Représentation des triplets illustratives par trois graphes optimisés (Jena2)

final et détecte les cas où un graphe ne contribue pas au résultat. Par exemple, le premier graphe de la Figure 4.1 n'est pas sollicité pour l'exécution de la requête *Q2* car il ne contient aucun triplet lié aux propriétés sollicitées dans *Q2*.

Le triplestore ***BitMat*** représente le graphe *RDF* en mémoire avec l'objectif de le rendre compact. Il utilise un modèle physique de données basé sur des matrices de bits. *BitMat* considère chaque triplet comme une entité à trois dimensions lui permettant de représenter le graphe *RDF* comme un cube de bits où chaque valeur indique la présence ou l'absence d'un triplet [ASH09, ACZH10].

Le cube de bits est représenté en mémoire par six matrices à deux dimensions (*SP, SO, PO, PS, OS, OP*) qui servent chacune à déterminer efficacement la troisième valeur d'un triplet étant donné ses deux autres valeurs. Par exemple, la Figure 4.2 illustre la matrice *SP* extraite du cube de bit pour représenter tous les triplets de la Table 4.1. Les colonnes de la matrice correspondent aux propriétés et les lignes correspondent aux sujets. Les valeurs binaires de la matrice correspondent aux codes binaires des objets éventuellement associés au sujet et à la propriété. L'absence de bits à 1 (c'est à dire la valeur `00000` dans la Figure 4.2) sert à indiquer l'absence, dans la base de connaissances, du triplet qui correspond aux valeurs s, p et o associé dans la matrice.

Les requêtes sont exécutées dans *BitMat* à l'aide des opérations de manipulation de bits (*AND* et *OR*) sur les colonnes des matrices en mémoire. Ainsi, l'exécution de *Q1* manipule les bits des colonnes 1, 2 et 6 pour retourner leurs valeurs dans le cas où aucune d'elles ne vaut pas `00000`. *BitMat* est limité pour l'exécution des requêtes de type *path-search*.

Les triplestores ***Sesame*** [BKVH02, Bro05] et ***swiftOWLIM*** [KOM05] utilisent le modèle logique de données, introduit par *Hayes et al*, qui consiste à stocker les triplets *RDF* sous la forme d'un graphe bipartie associant les URI des ressources de l'ontologie aux identifiants des triplets auxquels ils participent

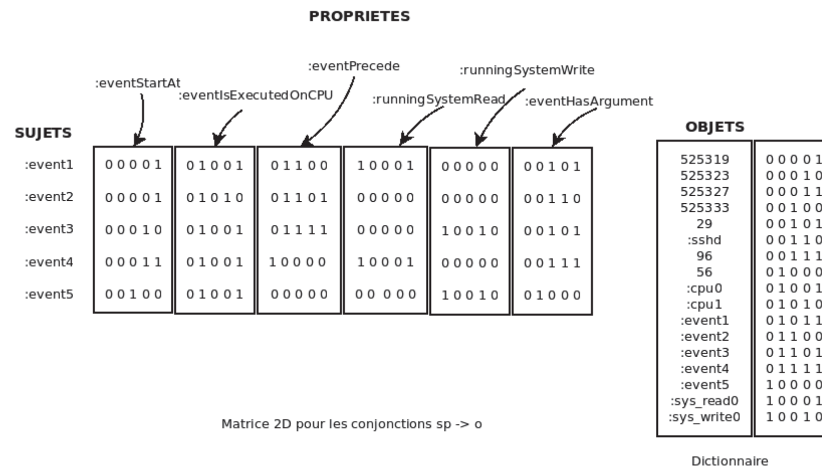


Figure 4.2: La matrice de bits *SP* et le dictionnaire des objets pour représenter les triplets illustratifs (BitMat)

[HG04]. Dans ce modèle, chaque sujet, objet ou propriété est directement reliée aux identifiants des triplets où il apparait et le lien indique le rôle qu'il joue dans le triplet.

La Figure 4.3 représente le graphe biparti construit par *Sesame* pour représenter les triplets (1, 6, 11, 15 et 19) qui ont tous le sujet *:event1* dans la Table 4.1. Avec ce modèle physique de données l'exécution de *Q1* correspond à l'intersection des triplets reliés aux URI des propriétés `eventStartAt`, `eventIsExecutedOnCPU` et `eventHasArgument`.

Le triplestore ***SpiderStore*** utilise un modèle logique de données proche du graphe bipartie. Chaque URI est représenté par une structure de données en mémoire contenant des pointeurs (catégorisés en *outgoing* pour les arcs sortants et *ingoing* pour les arcs entrants) vers d'autres structures de données qui correspondent aux URI avec lesquels il est directement connecté (en tant que sujet, propriété ou objet) [BGZ+10].

Les stratégies natives bénéficient de la localité des données en mémoire pour parcourir efficacement le graphe *RDF*. Cependant, la mémoire limite la taille des données que peuvent gérer les *triplestores* basés sur cette stratégie. De plus, les éventuels index ainsi que les triplets ajoutés ou inférés peuvent être perdus une fois que le *triplestore* est arrêté et doivent dans ce cas être reconstruits à chaque chargement des données dans le *triplestore*. Certains triplestores tels que *SpiderStore* organisent leurs structures de données et les stockent directement sur le disque sous la forme de fichiers binaires qui correspondent à des pages de la mémoire virtuelle. Ces fichiers binaires sont directement chargées en mémoire virtuelle au démarrage du triplestore ce qui permet de récupérer les structures de données. D'autres triplestores tels que *Sesame* et *Jena2* utilisent des bases de données pour la persistance de leurs données.

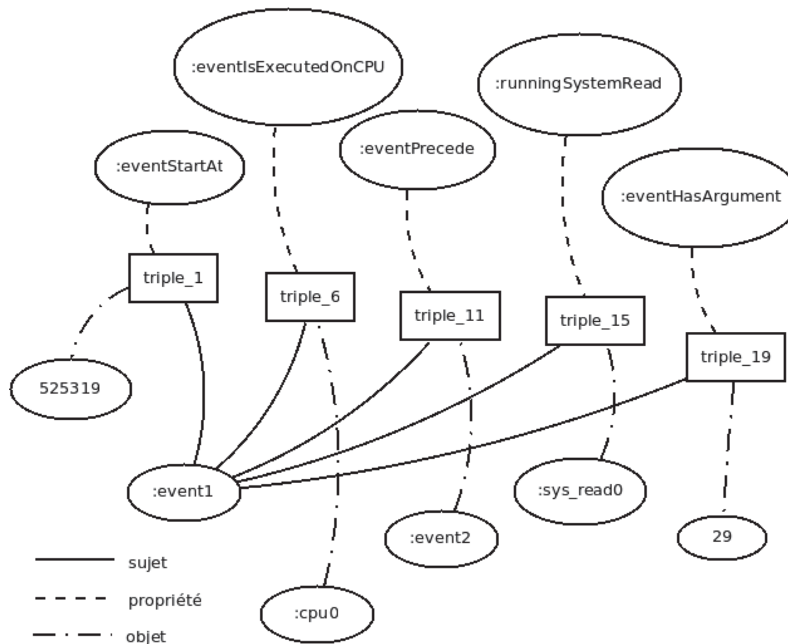


Figure 4.3: Graphe bipartie pour répresenter les triplets illustratif ayant `event1` comme sujet (Sesame)

La stratégie de stockage *multi-index* a pour objectif de permettre le passage à l'échelle des triplestores en utilisant intensivement des index pour optimiser l'accès à de grandes quantités de données.

4.1.2 La stratégie *multi-index*

L'idée de cette stratégie est celle proposée par *Harth* et *Decker* qui consiste à stocker les triplets *RDF* de manière à les accéder efficacement à l'aide des index [HD05]. L'approche est de construire des index qui couvrent toutes les possibilités d'accès aux triplets d'une ou de plusieurs ontologies.

Par exemple, un index est construit pour chacune des 6 permutations `spo, sop, pos, pso, osp` des trois positions d'un triplet `(s,p,o)`. La Figure 4.4 est un exemple d'index (`spo`) pour l'accès aux triplets de la Table 4.1. L'index est construit comme un arbre B^+ où tous les triplets sont triés selon le *sujet* puis stockés dans les feuilles de l'arbre.

Les grandes quantités de triplets sont généralement issues de plusieurs ontologies, ainsi les index sont construits pour les permutations des differentes positions des *quads* `(s,p,o,c)` c'est à dire le triplet `(s,p,o)` ainsi que l'identifiant de son ontologie de provenance c.

Les triplestores tels que ***YARS*** [HD05], ***Sesame-native*** [Bro05], ***OWLIM*** [KOM05] et ***TDB*** [Fou11] sont des exemples de triplestores qui construisent leurs index sous la forme d'arbres B^+. Tandis, que ***kowari*** [WGA05] utilise des arbres *AVL* pour construire ses index. L'utilisation des arbres entraîne la

replication des triplets de la base de connaissance dans chaque index.

Le triplestore ***Hexastore*** utilise une approche similaire à celle des arbres B^+. L'idée est de proposer les index pour toutes les 6 permutations possibles d'accès au triplet `(s,p,o)` [WKB08]. *Hexastore* utilise des structures de données en mémoire pour représenter les index et associe des vecteurs triés à chaque URI pour representer les URI auxquelles il est rattaché. Par exemple, dans un index `spo`, l'URI du sujet est rattaché à un vecteur trié contenant les URI des propriétés où chacune des valeurs fait également référence à un vecteur contenant les URI des objets. Toutefois, ces structures de données exigent beaucoup d'espace mémoire.

BRAHMS est un triplestore spécialement construit pour répondre efficacement aux requêtes de type *path-search*. Il utilise des structures de données spécialisées pour stocker chaque URI dans des blocs mémoires contigus et utilise une table de hachage pour faire les correspondances entre les blocs mémoires et les URI. *BRAHMS* propose ensuite les trois index suivants pour optimiser l'accès au voisinage d'une URI [JK05].

- $S \rightarrow O,P$ accès à l'objet et à la propriété à partir du sujet,
- $O \rightarrow S,P$ accès au sujet et à la propriété à partir de l'objet,
- $P \rightarrow S,O$ accès au sujet et à l'objet à partir de la propriété.

Par exemple, les index de type $P \rightarrow S,O$ permettent d'accéder efficacement au voisinage (`eventPrecede`) d'un évènement dans la requête *Q2*.

Le triplestore ***Virtuoso*** exploite les bases de données pour stocker les *quads* sous la forme `(s,p,o,g)` dans une seule table où g représente le graphe de provenance du triplet [EM09]. Pour optimiser l'accès, *Virtuoso* stocke les *quads* selon deux permutations `(g,s,p,o)` et `(o,g,p,s)` et utilise des index de type bitmap.

Bien que cette approche ait l'avantage de limiter la réplication des triplets, elle transforme les conjonctions des motifs de triplets en des auto-jointures sur la table rendant, l'évaluation de requêtes sur de grands jeux de données coûteux.

Le triplestore ***RDF-3X*** élimine la nécessité de l'utilisation d'une base de données physique par l'utilisation d'index exhaustifs pour toutes les permutations de triples *sujet-propriété-objet* [NW10]. *Neumann et al.* utilisent une table de triplets, potentiellement grande, basée sur leur propre implémentation de stockage et non sur une base de données. Ils adressent le problème du coût des auto-jointures en créant un ensemble approprié d'index. Tous les triplets sont triés dans l'ordre lexicographique et stockés dans un cluster sous la forme d'arbre B^+.

Le triplestore ***GRIN*** stocke les triplets en mémoire et utilise un index pour les accéder. L'idée est de regrouper les triplets directement connectés dans des ensembles et de choisir un triplet (ou centroide) permettant d'accéder à l'ensemble. Ainsi, l'index de *GRIN* est un cluster hiérarchique sous la forme d'un arbre binaire dont les feuilles représentent les triplets et les nœuds internes

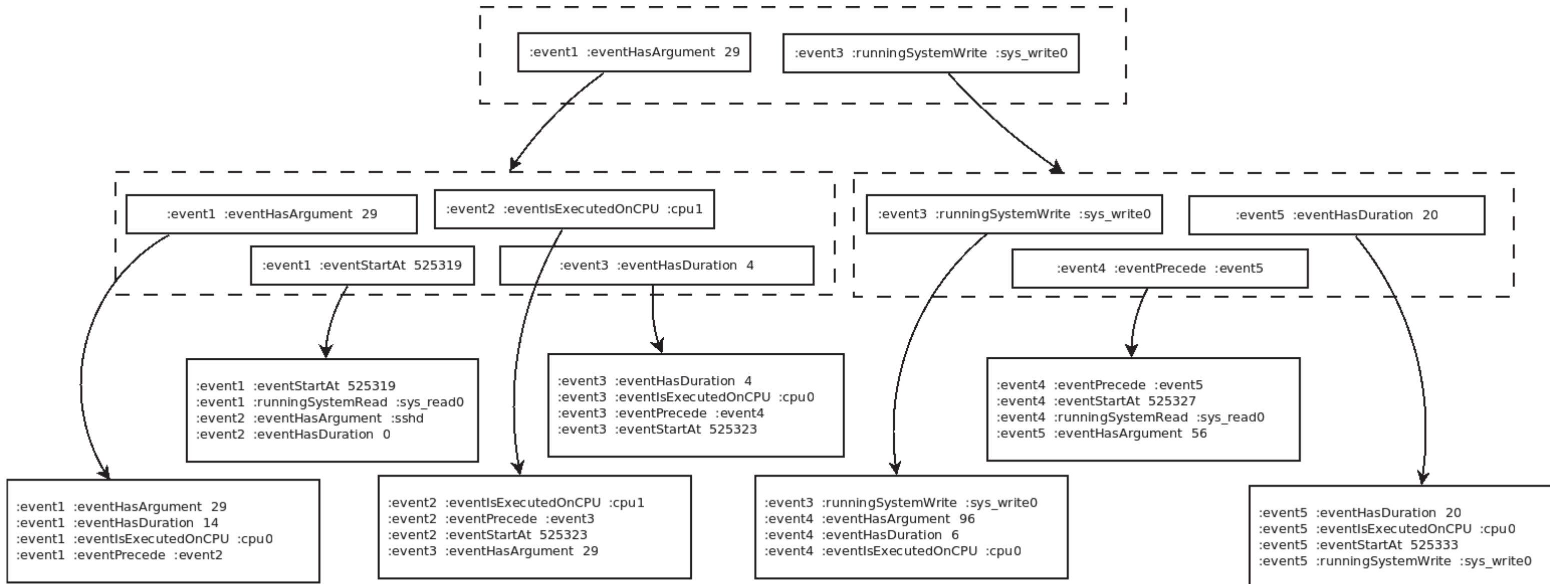


Figure 4.4: Arbre B$^+$ représentant l'index *spo* pour l'accès optimisé aux triplets de la table 4.1 à partir du sujet et de la propriété

représentent les triplets centroides [UPS07]. En conséquence, *GRIN* accède aux triplets en effectuant un parcours en profondeur de l'index pour identifier les feuilles. *GRIN* est adapté pour les requêtes de type *statement-search*.

La stratégie *multi-index* a l'avantage de permettre un accès à une grande quantité de données en utilisant des index. Cependant, pour avoir des index exhaustifs, certains triplestores tels que *TDB* et *virtuoso*, répliquent les données.

4.1.3 Les stratégies en bases de données

La stratégie en bases de données exploite des schémas relationnels spécifiques pour représenter les triplets *RDF* dans une base de données relationnelles.

Ces schémas peuvent être simples comme dans le cas des approches *TripleTable* et *Vertical Partionning*, ou plus complexes comme c'est le cas dans l'approche *Property Table*. Dans la suite nous allons présenter plus en détail ces différentes approches.

La table TripleTable

Cette approche fait usage d'un modèle relationnel basique qui représente un triplet *RDF* sous la forme d'un tuple ayant trois champs pour stocker respectivement le sujet, la propriété et l'objet dans une seule table appelée *TripleTable*.

Un champ supplémentaire (`isLiteral`) permet de distinguer les instances de classes des valeurs terminales à la position *objet* du triplet. Un dictionnaire est construit pour faire la correspondance entre les URI et des identifiants numériques. Ce sont ces identifiants qui sont utilisés dans la table *TripleTable* à la place des URI. Les trois tables suivantes resument la stratégie: `Resource` représente le dictionnaire des URI, `TripleTable` stocke les triplets et `Literal` stocke les valeurs des objets de type litteral.

```
TripleTable ( S INT,  O INT,  P INT, isLiteral BOOLEAN )

Resource ( ID INT, URI VARCHAR )

Literal ( ID INT, VARCHAR )
```

La requête *SQL* ci-dessous est équivalente à la requête *SPARQL Q1*. Les deux conjonctions *SPARQL* de *Q1* sont remplacées par des auto-jointures *SQL* sur la table `TripleTable`. En conséquence, les requêtes contenant un grand nombre de conjonctions sont potentiellement lentes à exécuter par cette stratégie à cause du grand nombre d'auto-jointures qu'elles génèrent sur la table `TripleTable`.

```
SELECT t1.o AS event, t1.o AS start, t3.o AS cpu, t4.s AS arg
FROM TripleTable AS t1, TripleTable AS t2, TripleTable AS t3,
     Resource d1, Resource d2, Resource d3
WHERE {
       d1.uri = 'eventStartAt'
   AND d2.uri = 'eventIsExecutedOnCPU'
   AND d3.uri = 'eventHasArgument'
   AND t1.p = d1.id
   AND t2.p = d2.id
   AND t3.p = d3.id
   AND t1.s = t2.s
   AND t2.s = t3.s
}
```

On peut citer ***3store*** [HG03], ***Jena-SDB*** [WSK+03], ***Sesame-RDBMS*** [BKVH02], ***KAON*** [BEH+02] comme des exemples de triplestores basés sur cette approche. L'avantage de cette stratégie de stockage est sa simplicité de mise en œuvre. Cependant, à grande échelle, les auto-jointures sur la table *Triple Table* deviennent inefficaces. L'idée de l'approche par *partitionnement vertical* est de réduire la taille des tables qui participent aux jointures *SQL* issues des conjonctions *SPARQL*.

Le partitionnement vertical

Cette approche, introduite par *Abadi et al*, consiste à regrouper tous les triplets qui correspondent à une même propriété dans une table dédiée à cette propriété. Le modèle relationnel pour représenter les triplets d'une propriété donnée consiste en deux champs `s` et `o` pour représenter respectivement les sujets et les objets du triplet [AMMH07].

Les auteurs recommandent l'utilisation des bases de données *columns-store* pour stocker les tables issues de cette approche. La Figure 4.5 illustre la différence entre le stockage des tuples en *column-store* et en *row-store*. Contrairement aux bases de données *row-store*, les bases de données *column-store* stockent individuellement les valeurs de chaque colonne de la table de manière contiguë sur le disque [AMH08]. Cela permet de réduire l'espace mémoire nécessaire pour la lecture d'une colonne de la table car, contrairement au *row-store*, cette lecture ne ramène pas les valeurs des autres colonnes de la table. De plus, les valeurs étant contiguës sur le disque leur lecture est plus efficace qu'une stratégie *row-store* consistant à faire des décalages pour éviter les valeurs des autres colonnes du tuple.

Les auteurs recommandent également de trier les valeurs des colonnes `s` pour optimiser les jointures de types *SS* [AMH08]. Ces colonnes ainsi triées permettent de rapidement reconstituer toutes les informations sur les propriétés d'un ensemble de sujets.

Les Tables 4.3 représentent le partitionnement vertical de l'ensemble des triplets de la Table 4.1. Tout comme dans l'approche *TripleTable*, un dictionnaire est utilisé pour stocker des identifiants numériques à la place des URIs dans

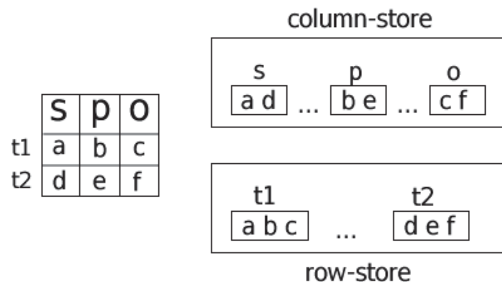


Figure 4.5: illustration des modèles physiques de données column-store et row-store

les tables. Le triplestore ***SW-store*** implémente le partitionnement vertical en utilisant le column-store *c-store* [AMMH09] .

eventStartAt	
s	o
:event1	525319
:event2	525323
:event3	525323
:event4	525327
:event5	525333

eventIsExecutedOnCPU	
s	o
:event1	:cpu0
:event2	:cpu1
:event3	:cpu0
:event4	:cpu0
:event5	:cpu0

runningSystemWrite	
s	o
:event3	:sys_write0
:event5	:sys_write0

eventPrecede	
s	o
:event1	:event2
:event2	:event3
:event3	:event4
:event4	:event5

eventHasArgument	
s	o
:event1	29
:event2	:sshd
:event3	29
:event4	96
:event5	56

runningSystemRead	
s	o
:event1	:sys_read0
:event4	:sys_read0

Table 4.3: Partitionnement vertical des triplets de la Table 4.1

Grâce au partitionnement vertical, une seule table est sollicitée lors de l'accès aux sujets et aux objets d'une propriété. Cette table est potentiellement inférieure en taille (de l'ordre de plusieurs facteurs) à la table `TripleTable`. La requête *SQL* ci-dessous est équivalente à la requête *SPARQL Q1* pour le partitionnement vertical.

```
SELECT t1.o AS event, t1.o AS start, t2.o AS cpu, t3.s AS arg
FROM eventStartAt AS t1, eventIsExecutedOnCPU AS t2,
     eventHasArgument AS t3
WHERE t1.s = t2.s
  AND t2.s = t3.s
```

Dans la requête précédente, les conjonctions de motifs *SPARQL* sont traduites en des jointures *SQL* entre différentes tables. Toutefois, cette approche ne permet pas d'éviter les auto-jointures dans le cas des requêtes de type *path-search*, où une même propriété est utilisée pour construire le chemin entre les sujets.

C'est le cas de la propriété `eventPrecede` dans *Q2*. De plus, l'insertion des triplets peut être lente dans le partitionnement vertical à cause des multiples tables à mettre à jour.

L'approche par regroupement des propriétés

L'idée dans cette approche est de regrouper un ensemble de propriétés dans une même table afin de réduire le nombre de jointures *SQL* nécessaires pour accéder aux triplets de ces propriétés. L'identification des ensembles de propriétés peut se faire automatiquement (ce qui est le cas des triplestores *Jena-PropertyTable* [Wil06], *4store* [HLS09] et *MonetDB/RDF* [PB14]) ou en rapprochant les propriétés en fonction du type d'interrogation à venir sur l'ontologie.

Par exemple, les tables ci-dessous représentent deux ensembles de propriétés {`eventStartAt`, `eventHasArgument`, `eventIsExecutedOnCPU`} et {`runningSystemRead`, `runningSystemWrite`, `eventPrecede`} pour représenter les triplets de la Table 4.1.

PropertyTable_1			
s	eventStartAt	eventHasArgument	eventIsExecutedOnCPU
:event1	525319	10	:cpu0
:event2	525323	0	:cpu1
:event3	525323	4	:cpu0
:event4	525327	6	:cpu0
:event5	525333	20	:cpu0

PropertyTable_2			
s	runningSystemRead	runningSystemWrite	:eventPrecede
:event1	:sys_read0	:sys_write0	:event2
:event2	NULL	NULL	:event3
:event3	NULL	:sys_write0	:event4
:event4	:sys_read0	NULL	:event5
:event5	NULL	:sys_write0	NULL

Dans les tables *PropertyTable_1* et *PropertyTable_2*, la colonne vaut *NULL* quand il n'y a pas de triplet qui associe le sujet à la propriété qui correspond à cette colonne. Ces tables ont été choisis de manière à réduire le nombre de jointures *SQL* nécessaires dans la requête *Q1*. Ainsi, la requête *SQL* ci-dessous permet de retourner le même résultat que la requête *SPARQL Q1* sans aucune jointure *SQL* sur la table *PropertyTable_1*.

```
SELECT t1.s AS event,
       t1.eventStartAt AS start,
       t1.eventHasArgument AS arg,
       t1.eventIsExecutedOnCPU AS cpu
FROM PropertyTable_1 AS t1
```

Toutefois, l'apparition des valeurs *NULL* dans la table rend complexe la réécriture des requêtes *SPARQL* vers *SQL*. Le deuxième inconvénient dans

l'approche est la complexité de sa mise en œuvre et la perte de la flexibilité du modèle *RDF* car l'ajout d'une nouvelle propriété dans l'ontologie peut nécessité une refonte drastique du modèle relationnel.

4.1.4 Conclusion

Dans cette section nous avons présenté les différentes stratégies de stockage des triplets *RDF*. Nous avons distingué la stratégie de stockage native *en mémoire*, la stratégie de stockage *multi-index* et la stratégie de stockage en *bases de données*. La Table 4.4 récapitule les *triplestores* que nous avons présenté tout au long de cette section en précisant pour chaque triplestore, la stratégie de stockage et le support de stockage qu'il utilise.

Dans notre contexte, l'interrogation des concepts métier contenus dans les traces d'exécution à l'aide de *VIDECOM*, pose plusieurs défis aux triplestores. Premièrement, la taille des triplets obtenus à partir des évènements de traces est très importante, ce qui limite le passage à l'échelle de certains triplestores. Ensuite, les requêtes *SPARQL* d'analyse des traces peuvent être autant de type *statement-search* ou *path-search*, ce qui rend certains triplestores efficaces uniquement pour certains types de requêtes. Le choix du triplestore à utiliser pour l'analyse des traces d'exécution apparaît donc comme un compromis à faire sur ces différentes contraintes.

Triplestore	Stratégie	Support
Jena2	Graphe	Mémoire
BitMat	Graphe	Mémoire
Sesame-memory	Graphe	Mémoire
SwiftOWLIM	Graphe	Mémoire
SpiderStore	Graphe	Mémoire
GRIN	Multi-index	Mémoire
BRAHMS	Multi-index	Mémoire
Hexastore	Multi-index	Fichier
RDF-3x	Multi-index	Fichier
Sesame-native	Multi-index	Fichier
OWLIM	Multi-index	Fichier
YARS	Multi-index	Fichier
TDB	Multi-index	Fichier
Kowari	Multi-index	Fichier
Virtuoso	Multi-index	SGBDR
3store	Triple Table	SGDBR
Sesame-RDBMS	Triple Table	SGDBR
Jena-SDB	Triple Table	SGDBR
KAON	Triple Table	SGBDR
SW-Store	Partitionnement Vertical	SGDBR
Jena-property	Property Table	SGDBR
MonetDB/RDF	Property Table	SGDBR
4store	Property Table	SGDBR

Table 4.4: Liste des triplestores présentés dans la section

Dans la prochaine section, nous présentons une étude comparative des performances de l'interrogation de VIDECOM dans plusieurs scénarii d'analyse de traces d'exécution. L'objectif est de déterminer les triplestores qui représentent les meilleurs compromis pour l'analyse des traces d'exécution.

4.2 Passage à l'échelle pour l'interrogation de VIDECOM

Dans la première partie de cette section, nous allons présenter le contexte expérimental dans lequel nous allons faire l'étude comparative. Ensuite, nous allons présenter les performances du chargement des données par les triplestores. Enfin, la dernière partie de la section présente plusieurs comparaisons des temps de réponse des triplestores à différentes requêtes *SPARQL* qui correspondent à des scénarii d'analyse de traces d'exécution.

4.2.1 Le contexte expérimental

Les traces

Nous allons comparer les triplestores sur des traces qui correspondent à différents temps d'exécution d'une application appelée *ts_record* pour l'enregistrement sur disque USB des données multimédia provenant d'un flux internet.

Le tableau ci-dessous récapitule pour chaque trace, le nombre d'évènements dans la trace, la durée d'exécution de l'application à laquelle correspond la trace, la taille sur disque de la trace ainsi que le nombre de triplets et la taille sur disque du fichier au format N-TRIPLE qui contient ces triplets. Nous allons voir dans le Chapitre 5 comment ces triplets sont obtenus en raisonnant sur les concepts métier de VIDECOM instancié à partir des évènements de traces.

	# Evénements	Durée	Taille	# Triplets	Taille triplets
T1	594 803	03 m 06 s	38 Mo	20 060 904	3 433 Mo
T2	1 189 172	06 m 09 s	75 Mo	40 118 282	6 878 Mo
T3	2 975 554	15 m 29 s	189 Mo	100 341 226	17 275 Mo

Le choix des triplestores

Pour cette analyse comparative nous avons choisi au moins un triplestore pour chaque stratégie de stockage illustrée dans la Table 4.4.

Pour la stratégie native, nous avons retenu *Jena2* (version 2.7.5) et *sesame-memory* (version 2.7.8). Pour la stratégie multi-index nous avons retenu *RDF-3x*, *Sesame-native* (version 2.7.14), *OWLIM* (*graphDB-SE* version 6.1), *TDB* (version 1.1.1) et *Virtuoso* (version libre 7.1.0). Et enfin, *4store* pour la stratégie *Property Table*.

Nous n'avons pas inclus certains triplestores pour des raisons d'indisponibilité des codes du triplestore au moment de notre collecte (par exemple *BRAHMS*, *SpiderStore*, *Hexastore*, *GRIN*, *3Store*, *MonetDB/RDF*, *SW-Store*), ou du fait

qu'il soit dans un état de prototype (*BitMat*).

Dans le but de comparer également l'impact des types des bases de données, nous avons utilisé le *column-store* libre *MonetDB* (version 1.7) et le *row-store* libre *PostGreSQL* (version 9.4) pour implémenter des triplestores *monetdb-tt* et *pgsql-tt* (respectivement *monetdb-vp* et *pgsql-vp*) qui correspondent à la stratégie de la table des triplets (respectivement au partitionnement vertical).

Concernant les index, *MonetDB* présente une particularité. En effet, les valeurs contiguës des colonnes des tables sont stockées par *MonetDB* dans des structures de données sous forme de tableaux en mémoire virtuelle, elles peuvent être efficacement stockées et lues sur le disque grâce à des fichiers qui correspondent à des segments de la mémoire virtuelle [Mon08b]. Ce modèle physique de données permet ainsi à *MonetDB* de négliger la clause *SQL* de création d'index. *MonetDB* choisis plutôt de détecter, pendant le chargement, si les valeurs de la colonne sont ordonnées afin de mettre en place des mécanismes optimisés de parcours des tables [Mon08a]. Ainsi, afin de rendre ces systèmes comparables nous allons trier la colonne `s` pour *MonetDB* et nous avons construit des index sur cette même colonne pour *PostGreSQL*.

La machine

Nous exécutons les expériences sur une machine ayant l'architecture *NUMA* (Non Uniform Memory Access) de 4 processeurs de 8 cœurs cadencé chacun à 2.27 GHz *Intel Xeon*. La machine dispose de 64 Gigaoctets de mémoire vive et de 114 Gigaoctets de mémoire virtuelle et d'un disque dur d'une capacité totale de 1 080 Gigaoctets. La machine utilise un système d'exploitation Linux 64-Bits Debian (version 3.16.7)

4.2.2 Le chargement des triplets

Le chargement initial des triplets dans le triplestore se fait à partir du fichier N-TRIPLE qui contient les triplets saturés provenant de la trace.

En fonction du triplestore, le chargement de triplets peut comporter plusieurs étapes intermédiaires telles que la construction des dictionnaires (pour *RDF-3x* et *4store*), la construction des index (notamment les index *ospc*, *posc* et *spoc* pour *sesame-native*) ou encore la détection des ensembles de propriétés (pour *4store*).

Dans le cas des triplestores *monetdb-tt* et *pgsql-tt*, le chargement de triplets commence par la construction du dictionnaire pour associer chaque URI à un identifiant. Ensuite, les URIs sont remplacés dans le fichier N-TRIPLE par leur identifiant respectif pour obtenir un nouveau fichier dit *numérisé*. Après avoir trié les valeurs de la colonne `s`, le fichier numérisé est chargé dans la table `TripleTable` en utilisant une requête *SQL* de type `COPY`. L'index est construit sur la table dans le cas de *PostGreSQL*.

Le processus est le même pour les triplestores *monetdb-vp* et *pgsql-vp*, à l'exception que le fichier numérisé est d'abord séparé en plusieurs fichiers (pour chacune des propriétés) contenant uniquement les champs `s` et `o`. Ensuite, chacun de ces fichiers est trié puis chargé dans la table correspondante et les index sont éventuellement construits sur chacune de ces tables.

La Figure 4.6 représente le temps de chargement des triplets pour chaque triplestore (à l'échelle logarithmique des millisecondes) en distinguant les différents supports de stockage utilisés.

Notre première observation est que la mémoire vive (64 Go) n'est pas suffisante pour que les triplestores *sesame-memory* et *Jena2*, qui utilisent la mémoire comme support, puissent charger la trace *T3*. Bien que sous sa forme de fichier N-TRIPLE, la trace *T3* a une taille brute de 17 Go, cette taille augmente dans le triplestore à cause des structures intermédiaires utilisées. Toutefois, *Jena* est plus rapide dans la construction de ses structures de données que *sesame-memory*. On remarque aussi que *MonetDB* est plus rapide que *PostGreSQL* car contrairement à ce dernier, il ne construit pas obligatoirement l'index.

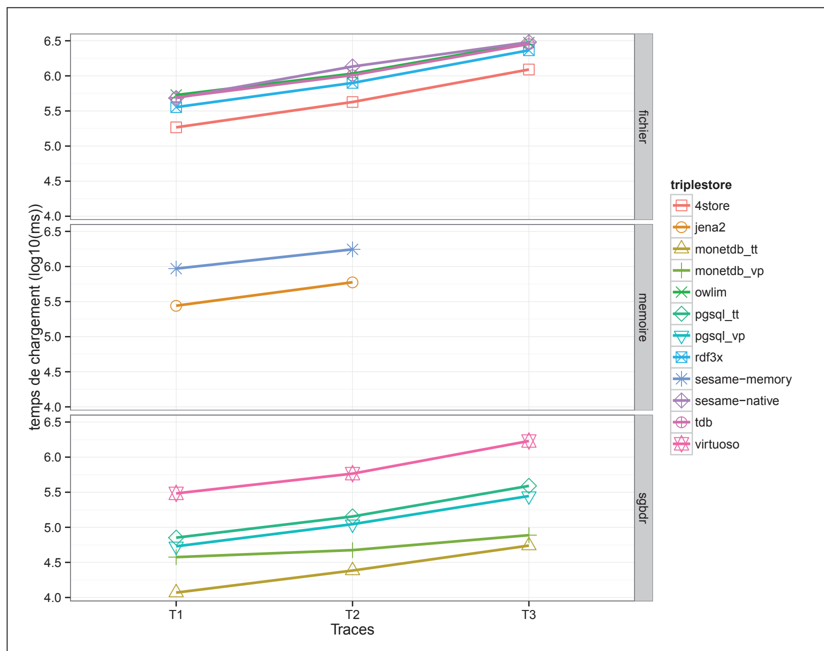


Figure 4.6: Temps de chargement des traces dans les triplestores

La Figure 4.7 représente le débit de chargement en nombre de triplets par seconde de chaque triplestore. Nous observons que ce débit est pratiquement constant pour tous les triplestores à l'exception de *monetdb-vp* et *monetdb-tt* où il est croissant.

Conclusion

Pour conclure cette comparaison des temps de chargement, nous considérons la Table 4.5 qui représente par ordre décroissant, les débits et les temps de chargement des triplets de la trace *T3* par les triplestores.

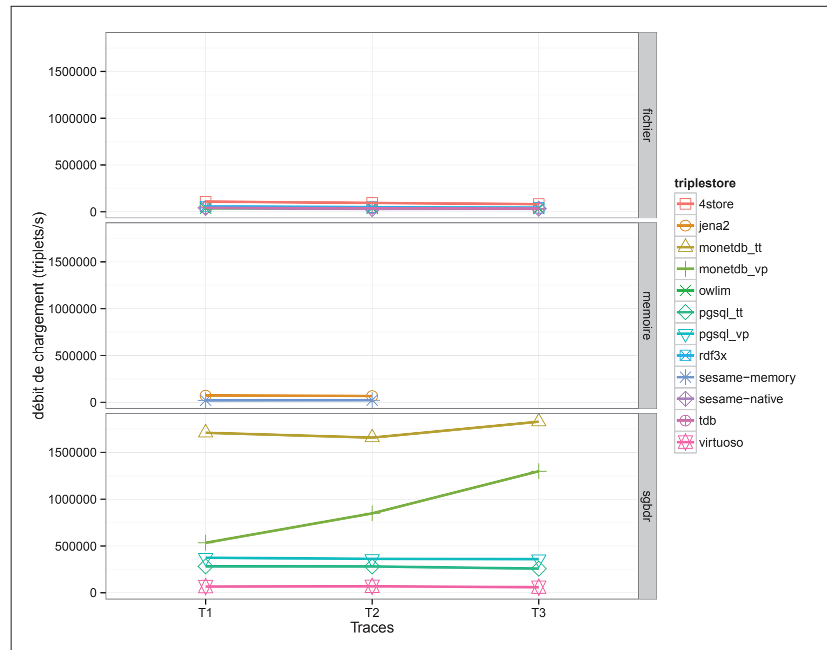


Figure 4.7: Débit de chargement des traces dans les triplestores du benchmark

Les coût de construction des trois index impactent les temps de chargement des triplestores les plus lents. Contrairement à *Sesame-native* qui construit les index (*ospc, posc* et *spoc*) que nous lui avons spécifié, *TDB* n'offre pas à l'utilisateur le choix des index à construire. Ainsi, au chargement il construit 8 index (*SPO, SPOG, GSPO, POS, POSG, GPOS, OSP, OSPG* et *GOSP*) ainsi que deux autres index *node* et *prefix2id* respectivement pour le dictionnaire des URI et les valeurs terminales.

Les approches de la table de triplets et du partitionnement vertical sont plus rapides que l'approche multi-index. Elles sont 10 fois plus rapides sur le *row-store postgresql* et 50 fois plus rapides sur le *column-store MonetDB*. Dans le prochaine section nous allons comparer les temps que mettent ces triplestores pour répondre aux requêtes d'analyse de traces.

4.2.3 Les scénarii d'analyse des traces d'exécution

Dans cette section nous allons comparer les temps de réponse des triplestores à différentes requêtes qui correspondent à des scénarii d'analyse de traces d'exécution. Le protocole expérimental est le suivant: chaque requête est exécutée 3 fois et chaque triplestore exécute toutes les requêtes dans l'ordre sur une trace donnée avant de passer à la suivante.

Le scénario de description des données

C'est le scénario basique où le développeur s'intéresse aux informations portées par chaque évènement de la trace. Ce scénario produit des requêtes de type

Triplestore	Temps	Débits (triplets/seconde)
sesame-native	50 m 20 sec	33 223
OWLIM	49 m 41 sec	33 653
tdb	47 m 02 sec	35 307
rdf3x	38 m 33 sec	43 363
virtuoso	28 m 15 sec	59 190
4store	20 m 32 sec	81 440
pgsql-tt	06 m 29 sec	257 821
pgsql-vp	04 m 38 sec	359 716
monetdb-vp	01 m 17 sec	1 298 831
monetdb-tt	**54 sec**	**1 828 042**

Table 4.5: Débits et temps de chargement des triplets et de la trace *T3*

statement-search contenant des conjonctions *sujet-sujet*. Nous considérons les requêtes suivantes pour ce scénario.

```
star.q1
SELECT ?event ?start ?end ?cpu ?duration
WHERE {
    ?event videcom:eventStartAt ?start .
    ?event videcom:eventEndAt ?end .
    ?event videcom:eventIsExecutedOnCPU ?cpu .
    ?event videcom:eventHasDuration ?duration
}
```

```
star.q2
SELECT ?event ?start ?end ?cpu ?duration ?nextInComponent
       ?nextInTrace ?nextInCPU ?nextInOccurrence
WHERE {
    ?event videcom:eventStartAt ?start .
    ?event videcom:eventEndAt ?end .
    ?event videcom:eventIsExecutedOnCPU ?cpu .
    ?event videcom:eventHasDuration ?duration .
    ?event videcom:eventPrecedeInCPU ?nextInCPU .
    ?event videcom:eventPrecedeInTask ?nextInTask .
    ?event videcom:eventPrecedeOccurrence ?nextInOccurrence .
    ?event videcom:eventPrecedeInComponent ?nextInComponent .
    ?event videcom:eventPrecedeInTrace ?nextInTrace
}
```

La requête *star.q1* retourne les temps de début, de fin et la durée de chaque évènement ainsi que le processeur sur lequel il s'exécute. La requête *star.q2* a pour objectif d'augmenter le nombre de conjonctions de *star.q1*. Elle retourne deux fois plus d'informations sur chaque évènement. La requête *star.q3* retourne les mêmes informations que *star.q2* mais uniquement pour les évènements qui correspondent aux interruptions de type *gic_tango_tp_mbx0*. L'objectif de *star.q3* est de rendre le résultat plus sélectif.

```
star.q3
SELECT ?event ?start ?end ?cpu ?duration ?nextInComponent
       ?nextInTrace ?nextInCPU ?nextInOccurrence
WHERE {
    ?event videcom:requestComponent videcom:gic_tango_tp_mbx0 .
    ?event videcom:eventStartAt ?start .
    ?event videcom:eventEndAt ?end .
    ?event videcom:eventIsExecutedOnCPU ?cpu .
    ?event videcom:eventHasDuration ?duration .
    ?event videcom:eventPrecedeInTrace ?nextInTrace .
    ?event videcom:eventPrecedeInCPU ?nextInCPU .
    ?event videcom:eventPrecedeInTask ?nextInTask .
    ?event videcom:eventPrecedeOccurrence ?nextInOccurrence .
    ?event videcom:eventPrecedeInComponent ?nextInComponent
}
```

La Figure 4.8 représente les temps de réponse aux requêtes *star.q1, star.q2* et *star.q3*. De manière générale, les résultats sélectifs de *star.q3* sont efficacement retrouvés par tous les triplestores. Les index *SPOG* de *tdb* sont un ordre de grandeur plus efficaces que les index *spoc* de *sesame-native*. Les triplestores *pgsql_vp* et *monetdb_vp* ont des temps de réponse qui sont dans la moyenne.

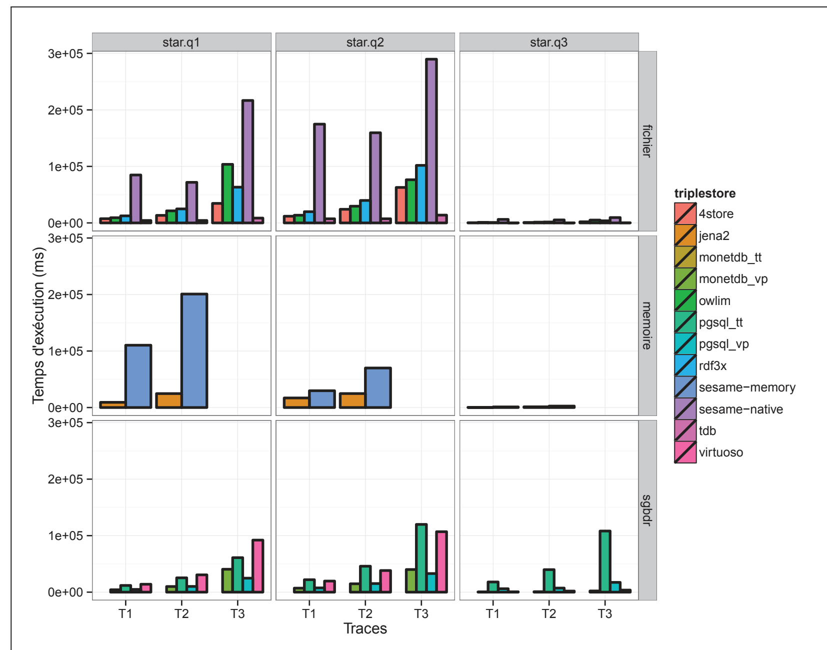


Figure 4.8: Temps d'exécution des requêtes du scénario de description des données

Le scénario de sélection temporelle des données

Du fait de la nature temporelle des données, le développeur peut s'intéresser uniquement à des données qui se trouvent dans un intervalle temporel bien précis. Dans ce scénario nous allons considérer les requêtes suivantes. La requête *filter.q1* applique un filtre sur les résultats de *star.q1* pour ne retenir que ceux qui sont dans l'intervalle temporel [78322418, 156644836]. La requête *filter.q2* quant à elle recherche toutes les tâches qui sont en cours d'exécution durant toutes les fonctionnalités dans l'intervalle de temps [70000000, 70020000].

La Figure 4.9 montre les temps d'exécution des requêtes *filter.q1* et *filter.q2*. Nous avons suspendu l'exécution des triplestores *OWLIM* et *sesame-native* après 1 h d'attente des résultats sans succès. Les triplestores sur fichier et en mémoire sont au moins un ordre de grandeur plus lents à exécuter *filter.q2* par rapport à *filter.q1*. Cette différence n'est pas aussi importante dans le cas des triplestores sur les bases de données.

```
filter.q1
SELECT ?event ?start ?end ?cpu ?duration
WHERE {
    ?event videcom:eventStartAt ?start .
    ?event videcom:eventEndAt ?end .
    ?event videcom:eventIsExecutedOnCPU ?cpu .
    ?event videcom:eventHasDuration ?duration
    FILTER(?start >= 78322418 && ?end <= 156644836)
}
```

```
filter.q2
SELECT ?function ?task
WHERE {
    ?function videcom:eventIsRelatedToFunctionality ?y .
    ?function videcom:eventStartAt ?start .
    ?function videcom:eventEndAt ?end .
    ?event videcom:eventStartAt ?sstart .
    ?event videcom:eventEndAt ?send .
    ?event videcom:runningTask ?task .
    FILTER(?start >= 70000000 && ?end <= 70020000)
    FILTER(?sstart >= ?start && ?send <= ?end)
}
```

Le scénario d'accès à des séquences d'évènements

Ce scénario regroupe les cas où le développeur s'intéresse à des séquences d'évènements. Dans VIDECOM, ce scénario produit des requêtes de type *path-search* contenant des conjonctions *sujet-objet*.

Nous considérons les requêtes suivantes pour ce scénario. La requête *path.q1* retourne toutes les séquences de taille 3 qui se produisent dans la tâche d'exécution *ts_record* et où le premier évènement de la séquence est un appel système. La requête *path.q2* augmente la taille de la séquence à 4. Tandis que la requête

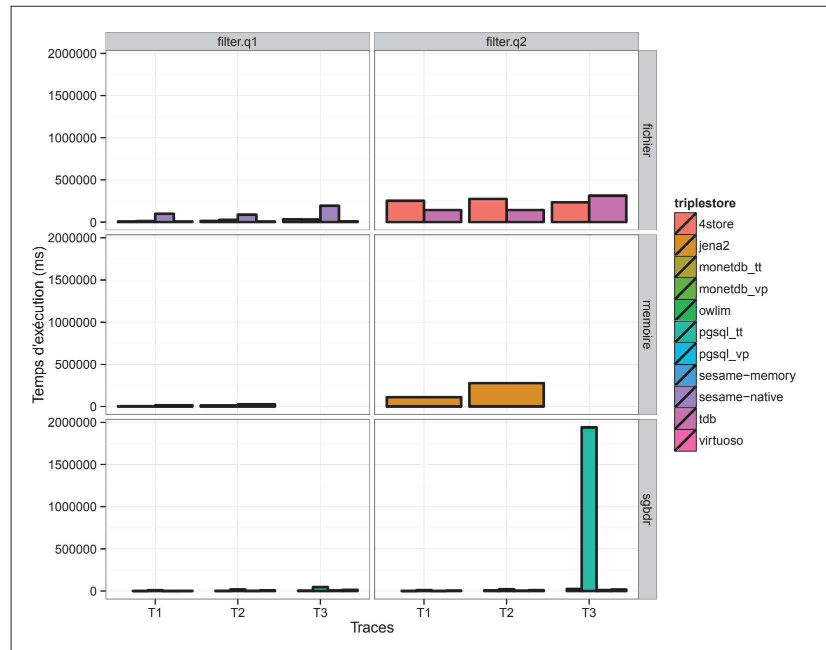


Figure 4.9: Temps d'exécution des requêtes du scénario de selection temporelle des données

path.q3 retourne également les séquences de taille 4 avec la contrainte que le quatrième évènement soit également un appel système.

```
path.q1
SELECT ?event1 ?event2 ?event3
WHERE {
    ?event1 videcom:SystemCallIsExecutedDuringTask videcom:ts_record0 .
    ?event1 videcom:eventPrecedeInTask ?event2 .
    ?event2 videcom:eventPrecedeInTask ?event3 .
}
```

```
path.q2
SELECT ?event1 ?event2 ?event3 ?event4
WHERE {
    ?event1 videcom:SystemCallIsExecutedDuringTask videcom:ts_record0 .
    ?event1 videcom:eventPrecedeInTask ?event2 .
    ?event2 videcom:eventPrecedeInTask ?event3 .
    ?event3 videcom:eventPrecedeInTask ?event4 .
}
```

```
path.q3
SELECT ?event1 ?event2 ?event3 ?event4 ?event5 ?event6
WHERE {
    ?event1 videcom:SystemCallIsExecutedDuringTask videcom:ts_record0 .
    ?event1 videcom:eventPrecedeInTask ?event2 .
    ?event2 videcom:eventPrecedeInTask ?event3 .
    ?event3 videcom:eventPrecedeInTask ?event4 .
    ?event4 videcom:SystemCallIsExecutedDuringTask videcom:ts_record0 .
}
```

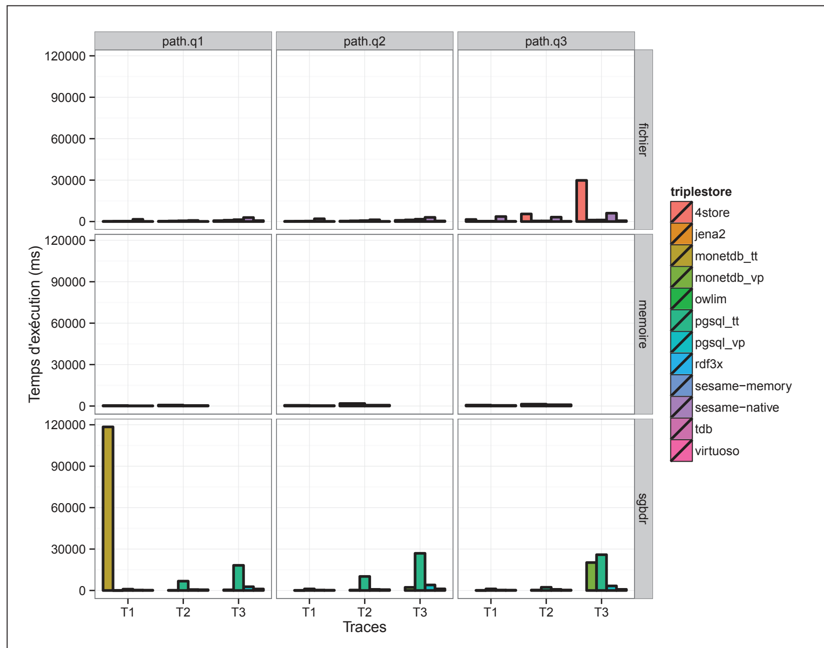


Figure 4.10: Temps d'exécution des requêtes du scénario d'accès aux séquences d'évènements

La Figure 4.10 montre les temps d'exécution requêtes *path.q1, path.q2* et *path.q3*. On observe que *tdb* et *virtuoso* sont les triplestores les plus performants sur ces requêtes. Cela peut être dû au fait que la majorité (plus de 60 %) des conjonctions des trois requêtes concernent la même propriété (`eventPrecedeInTask`). En conséquence, uniquement les index sur cette propriété sont parcourus. *monetdb-vp* est le triplestore le plus rapide juste après *tdb* et *virtuoso*. De même que *tdb* et *virtuoso*, *monetdb-vp* traite uniquement la table qui correspond à la propriété `eventPrecedeInTask` sur laquelle il effectue les auto-jointures. Cependant, le coût de ces auto-jointures devient important pour la trace *T3*. Ce qui explique la chute des performances de *monetdt-vp* sur la trace *T3*. Cette chute des performances s'observe dès la trace *T2* dans le cas de *pgsql-vp*.

Le scénario de tri et de limitation des données

Ce scénario permet de considérer les cas où les résultats des requêtes doivent être triés ou limités pour améliorer leur interprétation. Nous considérons les requêtes suivantes pour ce scénario.

La requête *sort.q1* sert de requête de référence aux autres requêtes qui effectuent des traitements supplémentaires sur les résultats de *sort.q1*.

```
sort.q1
SELECT ?task ?event ?start
WHERE {
    ?event videcom:runningTask ?task .
    ?event videcom:eventStartAt ?start .
}
```

La requête *sort.q2* effectue un tri (*ORDER BY*) sur les résultats. La requête *sort.q3* limite (*LIMIT*) les résultats uniquement aux 100 premiers. La requête *sort.q4* supprime les doublons dans les résultats (*DISTINCT*). Enfin, la requête *sort.q5* regroupe (*GROUP BY*) les résultats par catégorie et calcule leur effectif respectif.

```
sort.q2
SELECT ?task ?event ?start
WHERE {
    ?event videcom:runningTask ?task .
    ?event videcom:eventStartAt ?start .
}
ORDER BY ?start
```

```
sort.q3
SELECT ?task ?event ?start
WHERE {
    ?event videcom:runningTask ?task .
    ?event videcom:eventStartAt ?start .
}
LIMIT 100
```

```
sort.q4
SELECT DISTINCT ?task
WHERE {
    ?event videcom:runningTask ?task .
    ?event videcom:eventStartAt ?start .
}
```

```
sort.q5
SELECT ?task (COUNT(?event) AS ?number)
WHERE {
    ?event videcom:runningTask ?task .
    ?event videcom:eventStartAt ?start .
}
GROUP BY ?task
```

La Figure 4.11 montre les temps d'exécution des requêtes de ce scénario. On observe que la clause *LIMIT* dans la requête *sort.q3* n'impacte pas les performances *monetdb-vp*. En effet, ses temps de réponse varient en fonction de la taille des données ce qui n'est pas le cas des autres triplestores où les temps de réponse restent relativement constants. La raison de ce phénomène se trouve dans le fait que *MonetDB* n'a pas d'index et doit donc parcourir une seconde fois les données pour faire le post-traitement éventuel. On observe également que *monetdb-tt* retourne des résultats dans des temps raisonnables pour ces requêtes. Cela est dû au fait que les réquêtes comportent moins de conjonctions de motifs (uniquement 2) qui portent en plus sur des propriétés sélectives (`runningTask`). En résumé, les tripletstores sur bases de données sont les plus rapides pour ce scénario. En partitulier, *virtuoso* est le plus rapide suivi par *pgsql-vp*.

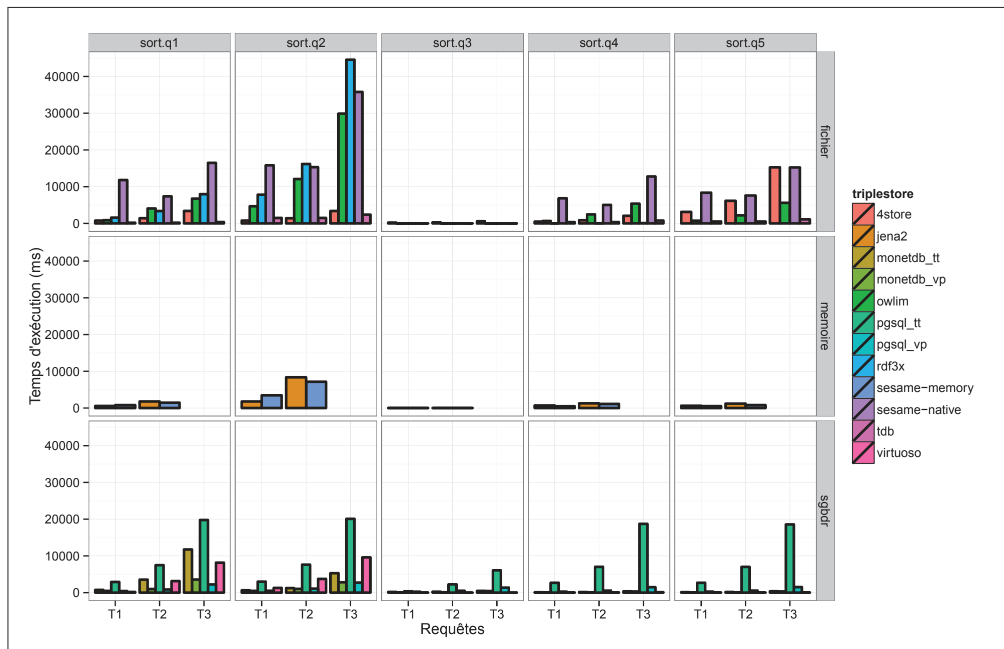


Figure 4.11: Temps d'exécution des requêtes de tri et de limitation des données

Le scénario d'agrégation des données

Ce scénario correspond aux cas où le développeur effectue des calculs sur les données. Les requêtes correspondantes dans ce scénario font usage des opérateurs d'agrégation. La requête *compute.q1* calcule la somme des temps d'exécution, tandis que la requête *compute.q2* calcule les moyennes de ces temps d'exécution pour chaque évènement de type interruption dans la trace.

```
compute.q1
SELECT (SUM(?duration) AS ?workload)
WHERE {
    ?event videcom:eventIsExecutedOn ?cpu .
    ?event videcom:eventHasDuration ?duration
}
```

```
compute.q2
SELECT ?interruption (AVG(?duration) AS ?workload)
WHERE {
    ?event videcom:runningInterruption ?interruption .
    ?event videcom:eventHasDuration ?duration
}
GROUP BY ?interruption
```

La Figure 4.12 représente les temps de réponse aux requêtes *compute.q1* et *compute.q2* où on peut observer que *monetdb-vp* est le triplestore le plus rapide. Cela est lié au nombre limité de motif de triplets dans les réquêtes (uniquement 2), à la sélectvité des propriétés de ces motifs et à l'absence des auto-jointures. Ainsi, les tables raménées en mémoire par *MonetDB* sont petites et sont efficacement interrogées. Dans le cas de la trace *T3*, *monetdb-vp* est plus rapide de deux ordres de grandeur que *sesame-native*.

Le scénario d'exploration du modèle VIDECOM

Ce scénario correspond aux situations où le développeur explore les connaissances contenues dans VIDECOM. Les requêtes de ce scénario permettent généralement d'explorer les propriétés qui peuvent s'appliquer à une instance de classe. Nous considérons les requêtes suivantes pour ce scénario. Les requêtes *model.q1* et *model.q2* retournent toutes les propriétés et respectivement les objets et les sujets reliés à chaque instance d'anomalie dans la trace.

```
model.q1
SELECT ?anomaly ?prop ?object
WHERE {
    ?event videcom:eventIsRelatedToAnomaly ?anomaly .
    ?anomaly ?prop ?object
}
```

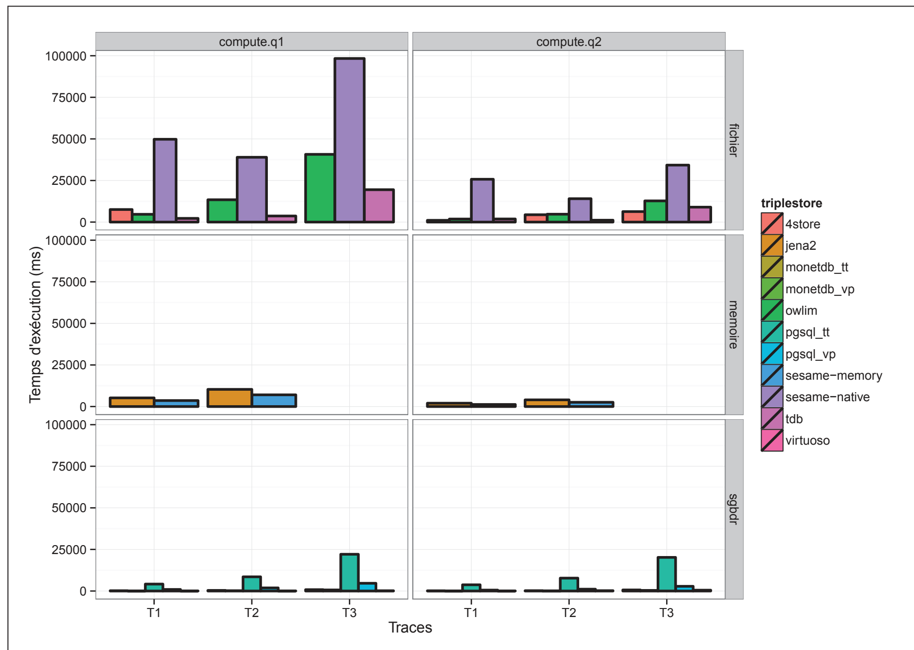


Figure 4.12: Temps d'exécution des requête d'agrégation de données

```
model.q2
SELECT ?subject ?anomaly ?prop
WHERE {
    ?event videcom:eventIsRelatedToAnomaly ?anomaly .
    ?subject ?prop ?anomaly
}
```

La Figure 4.13 représente les performance des requêtes *model.q1* et *model.q2*. La requête *model.q1* est sélective car aucune instance d'anomalie n'apparaît comme sujet d'un triplet, contrairement à *model.q2*. On observe que *monetdb-vp* et *tdb* sont plus performants pour la requête *model.q1*.

La Table 4.6 récapitule pour chaque requête la moyenne de trois temps de réponse (en millisecondes) des triplestores sur la trace *T3*. La somme de ces moyennes est calculée dans la dernière ligne de la table. Les valeurs en gras dans chaque ligne de la table représentent les meilleurs temps de réponse pour la requête correspondante.

Grâce à ses index *tdb* est le triplestore le plus performant autant sur les requêtes *statement-search* que sur les requêtes *path-search*. Toutefois, *tdb* est inefficace pour la requête *filter.q2*. Ainsi, dans un contexte d'analyse de traces où le développeur ne s'intéresse à rechercher des informations complexes dans des intervalles de temps, le triplestore *tdb* apparaît comme le meilleur compromis

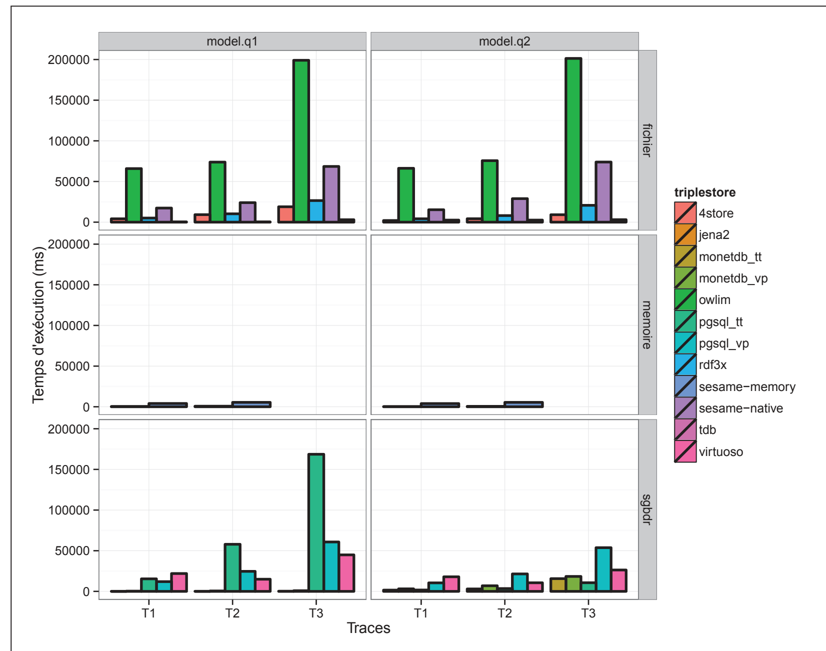


Figure 4.13: Temps d'exécution des requêtes d'exploration de l'ontologie

	tdb	virtuoso	monetdb-vp	pgsql-vp	pgsql-tt
star.q1	**8 576**	92 098	40 539	24 742	61 052
star.q2	**13 833**	106 975	40 044	32 925	119 863
star.q3	**317**	3 634	2 118	17 284	108 167
path.q1	694	1 091	**486**	2 727	18 211
path.q2	**428**	1 180	2 270	4 000	26 912
path.q3	**596**	807	20 227	3 300	25 912
sort.q1	**361**	8 156	3 566	2 231	19 767
sort.q2	**2 395**	9 597	2 812	2 732	20 088
sort.q3	**3**	8	424	1 362	6 074
sort.q4	785	**121**	322	1 481	18 720
sort.q5	1 107	**44**	336	1 512	18 565
filter.q1	10 960	14 028	**5 051**	6 742	46 618
filter.q2	312 642	17 848	24 461	**11 269**	1 939 484
compute.q1	3 106	**176**	645	4 659	22 093
compute.q2	1 052	513	**366**	2 835	20 219
model.q1	3 006	44 899	**829**	60 766	168 616
model.q2	**3 054**	26 300	18 365	53 745	10 650
total	362 915	327 475	**162 861**	234 312	2 651 011

Table 4.6: Récapitulatif des performances des triplestores sur la trace T3

pour l'analyse de trace.

En ce qui concerne le partitionnement vertical, la requête *filter.q2* est exécutée respectivement 13 et 28 fois plus vite par *monetdb-vp* et *pgsql-vp*. Grâce à ses structures de données sous forme de tableaux (éventuellement triées) en mémoire, *monetdb-vp* exploitent mieux la sélectivité des requêtes comme le montrent ses temps de réponse aux requêtes *star.q3*, *model.q1* et *model.q2*. Toutefois, le nombre de jointures pour les requêtes *statement-search* et d'auto-jointures pour les requêtes *path-search* peuvent rapidement dégrader les performances du triplestore à cause de la matérialisation des résultats intermédiaires en mémoire comme l'illustrent les requêtes *path.q2* et *path.q3*. Cependant, la Table 4.6 illustre que monetdb-vp est le seul triplestore dont le temps de réponse n'excède pas 40 secondes.

La table ne contient pas les résultats des triplestores *jena2* et *sesame-memory* car la taille mémoire ne leur permettaient pas de charger la centaine de triplets de la trace *T3*. La table ne contient pas les résultats des triplestores *OWLIM* et *sesame-native* car tous deux n'ont pas terminé l'exécution de la requête *filter.q2* après 1 heure d'exécution. La table ne contient pas les résultats pour *rdf-3x* car il n'exécute pas les requêtes contenant les opérateurs d'agrégation et de filtre. La table ne contient pas les résultat du triplestore *4store* car il produit une erreur pour la requête *compute.q1* sur les traces *T2* et *T3*.

La table ne contient pas les résultats pour le triplestore *monetdb-tt*. En effet, *monetdb-tt* n'a pu exécuter, dans un temps raisonnable, que les requêtes de tri et de modèle car ces requêtes produisent au maximum deux auto-jointures.

4.3 Conclusion

L'objectif de ce chapitre était de déterminer les caractéristiques des triplestores qui sont les plus adaptées à l'analyse de traces via VIDECOM. Nous nous sommes intéressé en particulier aux caractéristiques liées au stockage ainsi qu'à l'interrogation *SPARQL* des triplets. Nous avons ainsi présenté les différentes approches de stockage des triplets dans les triplestores (en mémoire, en multi-index et en bases relationnelles). Ensuite, nous avons effectué une comparaison des performances de 12 de ces triplestores pour le chargement et l'interrogation des triplets sur un benchmark de 17 requêtes *SPARQL* pour l'analyse de traces.

Le chargement des triplets dans les triplestores en mémoire, tels que *Jena2* et *sesame-memory*, est limité par la taille de la mémoire disponible. Nos expérimentations ont montré que dans le cas des triplestores *multi-index*, le débit de chargement des triplets ne dépasse pas les 100 000 triplets/secondes à cause principalement du coût de construction des index. Cependant, grâce au chargement massif en *SQL* des triplets, nos implémentations des approches `TripleTable` et `Vertical Partitioning` de stockage en bases relationnelles, ont des débits 10 fois plus élevés que ceux des triplestores *multi-index*. Ces débits vont jusqu'à 1.3 millions de triplets/secondes pour *monetdb-vp* et 1.8 millions de triplets/seconde pour *monetdb-tt*.

Les triplestores *multi-index* tirent un avantage de leurs multiples index pour

répondre efficacement aux requêtes. C'est notamment le cas de *tdb* qui est le seul triplestore à avoir le meilleur temps de réponse sur 50 % des requêtes de notre benchmark. Toutefois, les performances de *tdb* s'effondrent pour des requêtes très utilisées durant l'analyse pour obtenir des concepts métiers par rapport à d'autres concepts métiers. Cela étant dû au caractère de sous-requête qui apparait dans ce type de requête. Nos expérimentations ont montré que *monetdb-vp* est le triplestore le plus rapide pour ce type de requêtes. Il est 13 fois plus rapide que *tdb* qui met 5 minutes à répondre à la requête sur une base de 100 millions de triplets contre 24 secondes pour *monetdb-vp* sur la même base.

Ainsi, bien que *monetdb-vp* soit 2 à 4 fois moins rapide que *tdb* sur 50 % des requêtes du benchmark, il s'illustre par le fait qu'il est le seul triplestore à répondre à chacune des requêtes du benchmark en au plus 40 secondes sur la base de 100 millions triplets. Toutefois, nos expérimentations ont également mis en évidence que nos implémentations des triplestores en *MonetDB* passent difficilement à l'échelle en fonction du nombre de conjonctions *SPARQL* de la requête. En effet, ces conjonctions *SPARQL* sont éventuellement transformées en des auto-jointures *SQL* qui, dans le cas de *MonetDB*, implique la matérialisations des résultat intermédiaires en mémoire. Ainsi, nos expérimentations ont montré que de nos deux implémentations de *postgresql* (*pgsql-tt*) et *MonetDB* (*monetdb-tt*) de l'approche `TripleTable`, seul *pgsql-tt* répondait à toutes les requêtes du benchmark contrairement à *monetdb-tt*. De plus concernant le `Vertical Partitioning`, *pgsql-vp* est 6 fois plus rapide *monetdb-vp* pour les requête de type *path-search*.

En conclusion, nous retenons que les triplestores *monetdb-vp* est le meilleur compromis en terme de chargement et de d'interrogation des triplets pour l'analyse des traces via VIDECOM. Toutefois, le support d'un moteur d'inférence et plus précisément l'inférence des règles métier est une caractéristique importante d'un triplestore destiné à l'analyse de traces via notre ontologie VIDECOM. Ainsi, dans le prochain chapitre nous allons présenter les moteurs d'inférence et introduire les techniques qu'ils utilisent pour appliquer les règles d'inférence à la base de connaissances. Nous allons également illustrer les limites de ces techniques dans le cas de l'inférence des règles métiers et nous allons présenter notre contribution à la résolution de ces limites.

Chapitre 5

L'inférence des concepts métier dans VIDECOM

Sommaire

5.1 L'inférence des connaissances pour répondre à des requêtes . . . 97
5.1.1 La réécriture de requêtes . . . 97
5.1.2 La saturation . . . 98
5.2 La saturation des règles métier . . . 100
5.2.1 L'algorithme de saturation . . . 100
5.2.2 L'inférence des nouvelles instances dans les règles . . 101
5.2.3 L'inférence de l'ordre temporel entre les évènements 102
5.3 Proposition d'un moteur d'inférence pour les règles métier . . . 103
5.3.1 Définitions formelles . . . 103
5.3.2 Illustration du modèle pour l'inférence des nouvelles instances . . . 106
5.3.3 L'inférence des relations temporelles entre les instances consécutives . . . 109
5.3.4 Illustration du modèle pour l'inférence de l'ordre temporelle . . . 110
5.4 Les expérimentations . . . 111
5.4.1 Le contexte expérimental . . . 112
5.4.2 Comparaison des temps de saturation . . . 115
5.4.3 Comparaison des temps de réponse aux requêtes . . 116
5.5 La saturation à grande échelle . . . 118
5.5.1 Le modèle de saturation sur le partitionnement vertical118
5.5.2 Comparaison des approches TripleTable et Vertical-Partitioning . . . 119
5.5.3 Le passage à l'échelle de la saturation . . . 120
5.6 Conclusion . . . 122

Grâce à notre ontologie VIDECOM, le développeur peut expliciter des concepts métier dans la trace en utilisant des règles d'inférence dites "*métier*" car spécifiques à son domaine d'expertise.

Les moteurs d'inférence servent à inférer les connaissances dans l'ontologie en se basant sur les règles d'inférence définies sur l'ontologie. Cependant, les règles d'inférence métier ont deux spécificités qui les rendent différentes des règles d'inférence usuelles et qui posent plusieurs défis, notamment la terminaison de l'inférence des concepts métiers et l'inférence de l'ordre temporel entre les évènements.

Dans ce chapitre, nous présentons notre contribution, qui est une proposition d'un moteur d'inférence adapté pour la matérialisation des concepts métier dans les traces d'exécution. Dans la première partie du chapitre (Section 5.1), nous introduisons les deux approches pour le raisonnement dans les bases de connaissances et nous déterminons celle qui est la plus adaptée dans le contexte de l'analyse des traces. Dans la seconde partie du chapitre (Section 5.2), nous identifions les problèmes que pose l'inférence des règles métier aux moteurs d'inférence existants. La troisième partie du chapitre (Section 5.3) présente le modèle sur lequel se base notre proposition pour l'inférence des règles métier. La quatrième partie du chapitre (5.4) est une étude comparative des performances de notre proposition de moteur d'inférence et des performance des moteurs d'inférence de l'état de l'art sur des traces réelles d'une application embarquée multimédia. Enfin, la cinquième partie (Section 5.6) conclut le chapitre.

Contexte

Tout au long de ce chapitre, nous allons illustrer la matérialisation de concepts métier dans les triplets de la Table 5.1 qui représentent six triplets issus de deux évènements d'une trace d'exécution.

	s	p	o
1	`:event1`	`:eventStartAt`	`525319`
2	`:event1`	`:eventEndAt`	`525323`
3	`:event1`	`:eventReadData`	`10`
4	`:event2`	`:eventStartAt`	`525325`
5	`:event2`	`:eventEndAt`	`525337`
6	`:event2`	`:eventWriteData`	`10`

Table 5.1: Triplets pour l'illustration de l'inférence des concepts métier

Nous allons illustrer les caractéristiques des règles d'inférence métier en nous servant des règles R_1 et R_2. La règle R_1 est la règle d'inférence de l'ordre de *précédence* entre deux évènements dans VIDECOM. Elle se base sur les temps respectifs de fin et de début des évènements pour inférer le concept de précédence (`eventPrecede`) entre les instances des évènements dans l'ontologie VIDECOM.

R_1	Si	Alors

	`?a :eventEndAt ?endA .`
	`?b :eventStartAt ?startB .`
	`?endA < ?startB`
		`?a :eventPrecede ?b`

La règle R_2 est la règle d'inférence d'une fonctionnalité (`dataCopy`). Elle se base sur la relation de précédence entre les évènements de lecture (`eventReadData`) et d'écriture (`eventWriteData`) des données pour créer une nouvelle instance de la fonctionnalité (`_:dc`). Cette nouvelle instance est rattachée à une nouvelle instance d'évènement (`_:ev`). Les temps de début (`eventStartAt`) et de fin (`eventEndAt`) de cette nouvelle instance d'évènement correspondent respectivement aux temps de début et de fin des évènements de lecture et d'écriture de données.

R_2	Si	Alors

	`?a :eventReadData ?data .`
	`?b :eventWriteData ?data .`
	`?a :eventPrecede ?b .`
	`?a :eventStartAt ?startA .`
	`?b :eventEndAt ?endB`
		`_:dc rdf:type :dataCopy`
		`_:ev :eventIsRelatedToFunctionality _:dc`
		`_:ev :eventStartAt ?startA`
		`_:ev :eventEndAt ?endB`

5.1 L'inférence des connaissances pour répondre à des requêtes

Dans le but de répondre de manière complète aux interrogations posées sur la base de connaissance, les moteurs d'inférence utilisent les règles d'inférence de l'ontologie pour inférer les connaissances qui ne sont par explicitement exprimées dans la base de connaissances.

On distingue deux approches pour répondre à des requêtes sur une base de connaissances (contenant des faits et des règles) à savoir la *réécriture de requêtes* et la *saturation*.

5.1.1 La réécriture de requêtes

Pour chaque requête, l'approche par réécriture de requêtes consiste à exploiter un raisonnement de type *chaînage arrière* (ou *backward-chaining*) pour réécrire les requêtes en remplaçant les atomes de la requête par les prémisses des règles qui permettent de les inférer. Ce remplacement est itéré pour chaque atome qui peut être produit par les règles. Quand les règles sont non recursives, on obtient un ensemble fini de réécritures. L'ensemble des réponses certains de la réquête initiale peut alors être obtenu en prenant l'union des réponses produites en les évaluant de façon standard sur les faits de la base de connaissances.

Considérons par exemple la requête ci-dessous qui recherche tous les événements précédés par `event1` et qui s'exécute sur le même processeur.

```
SELECT ?event
WHERE {
    videcom:event1  videcom:eventPrecede ?event .
    videcom:event1  videcom:eventIsExecutedOnCPU ?cpu .
    ?event  videcom:eventIsExecutedOnCPU ?cpu .
}
```

Cette requête est réécrite en la requête suivante où la connaissance implicite `eventPrecede` est remplacée par la conjonction de la prémisse de la règle R_1.

```
SELECT ?event
WHERE {
    videcom:event1  videcom:eventEndAt ?end .
    ?event  videcom:eventStartAt ?start .
    FILTER(?end > ?start).
    videcom:event1  videcom:eventIsExecutedOnCPU ?cpu .
    ?event  videcom:eventIsExecutedOnCPU ?cpu .
}
```

Toutefois, la réécriture de requêtes est une opération qui peut être coûteuse quand le nombre de règles et le nombre de conjonctions par règle sont importants. La réécriture peut ainsi produire un nombre exponentiel de réécritures en fonction de la taille de la réquête (qui peut être de l'ordre d'une dizaine de conjonctions dans le cas des requêtes d'analyse de traces). De plus, cette opération de réécriture doit être exécutée pour chaque requête.

5.1.2 La saturation

L'idée de cette approche d'inférence des connaissances est de *matérialiser* toutes les connaissances implicites que les règles d'inférence peuvent inférer avant de répondre aux requêtes. L'approche privilégie ainsi la réactivité de la base de connaissances dans la réponse aux requêtes. La matérialisation des connaissances, encore appelée *saturation des données*, est exécutée au chargement initial des triplets et/ou à l'ajout d'une nouvelle règle métier ou de nouveaux triplets.

Les moteurs d'inférence qui implémentent la saturation utilisent une stratégie d'inférence de type *forward-chaining* pour matérialiser les connaissances implicites à partir des règles d'inférence. Cette catégorie de moteurs d'inférence est plus utilisée dans les ontologies légères où la sémantique est généralement limitée aux contraintes des propriétés *RDFS*. Malgré la sémantique limitée des règles d'inférence, la taille des données a un impact important sur la durée de la saturation (qui peut prendre beaucoup de temps avant de terminer).

Pour réduire ce temps de saturation, les auteurs *Jesse Weaver* et *James Hendler* proposent une approche de saturation des règles *RDFS* basée sur un algorithme parallèle [WH09]. Toutefois, leur proposition se limite aux règles *RDFS* et n'inclut pas la possibilité de saturer des règles d'inférence utilisateurs. Certains moteurs d'inférence utilisent des approches ad-hoc pour sa-

turer les règles *RDFS* et les règles utilisateurs. On peut citer les moteurs d'inférence tels que *KAON2* [MS05] qui s'inspirent des techniques issues des bases de données déductives (à base de programmes *Datalog*) pour effectuer la saturation. D'autres moteurs, tels que *Drools* [PNLF08] et *Jena* utilisent une implémentation de *RETE*, le meilleur algorithme de *forward-chaining* [For82]. Les moteurs d'inférence *Sesame* [Ses15], *OWLIM* [Ont14] utilisent des algorithmes spécialisés.

Certains des moteurs d'inférence de cette catégorie ne prennent pas en compte toutes les règles *RDFS* pendant la saturation. Ainsi, les 7 règles *RDFS* ci-dessous sont généralement ignorées car elles matérialisent des concepts tels que `rdfs:Datatype`, `rdfs:member`, `rdfs:Literal` et `rdfs:Resource` qui ont un intérêt limité dans les requêtes posées sur l'ontologie.

	Si	Alors
rdfs1	any IRI ?a in D	?a rdf:type rdfs:Datatype .
rdfs4a	?x ?a ?y .	?x rdf:type rdfs:Resource .
rdfs4b	?x ?a ?y.	?y rdf:type rdfs:Resource .
rdfs6	?x rdf:type rdf:Property .	?x rdfs:subPropertyOf ?x .
rdfs8	?x rdf:type rdfs:Class .	?x rdfs:subClassOf rdfs:Resource .
rdfs10	?x rdf:type rdfs:Class .	?x rdfs:subClassOf ?x .
rdfs12	?x rdf:type rdfs:ContainerMembershipProperty .	?x rdfs:subPropertyOf rdfs:member .
rdfs13	?x rdf:type rdfs:Datatype .	?x rdfs:subClassOf rdfs:Literal .

Tandis que les règles `rdfs2, rdfs3, rdfs5, rdfs7, rdf9` et `rdfs11` sont essentielles pour matérialiser la *subsumption* et la *satisfiabilité* dans les connaissances. Par exemple, la version libre 5.0 du moteur d'inférence *Virtuoso* infère uniquement les contraintes des propriétés `rdfs:subClassOf` et `rdfs:subPropertyOf` en plus des règles utilisateurs [Vir99].

	Si	Alors
rdfs2	?a rdfs:domain ?x . ?y ?a ?z	?y rdf:type ?x .
rdfs3	?a rdfs:range ?x . ?y ?a ?z	?z rdf:type ?x .
rdfs5	?x rdfs:subPropertyOf ?y . ?y rdfs:subPropertyOf ?z	?x rdfs:subPropertyOf ?z .
rdfs7	?a rdfs:subPropertyOf bbb . ?x ?a ?y	?x bbb ?y .
rdfs9	?x rdfs:subClassOf ?y . ?z rdf:type ?x	?z rdf:type ?y .
rdfs11	?x rdfs:subClassOf ?y . ?y rdfs:subClassOf ?z	?x rdfs:subClassOf ?z .

Dans le contexte de l'analyse des traces avec l'ontologie VIDECOM, le moteur d'inférence doit prendre en compte les règles métier du développeur et répondre le plus rapidement possible aux requêtes. Tout cela sur de grands volumes de connaissances issues des évènements des traces. Dans ces conditions, la saturation représente le meilleur compromis pour l'inférence des concepts métier dans la trace. Dans la prochaine section, nous allons présenter les problèmes que pose la saturation des règles métier pour l'analyse des traces d'exécution.

5.2 La saturation des règles métier

Les langages d'expression des règles tels que *Datalog* et *SWRL* permettent de représenter les règles d'inférence dans un formalisme $\{\texttt{P} \rightarrow \texttt{C}\}$. Dans ce formalisme la *prémisse* P est une conjonction de motifs de triplets et la *conclusion* C est un ensemble de motifs de triplets.

La saturation consiste donc à matérialiser les motifs de triplets de la conclusion dans la base de connaissances chaque fois que la conjonction de la prémisse est vérifiée. Ce processus est fait jusqu'à ce qu'aucune nouvelle connaissance (ou nouveau triplet) ne soit inférée.

5.2.1 L'algorithme de saturation

Nous pouvons illustrer la procédure de saturation par l'algorithme 1. L'algorithme prend en entrée la base de connaissances $\mathcal{T}$ et l'ensemble des règles d'inférence $\mathcal{R}$. La saturation est faite en plusieurs itérations. Chaque itération consiste à appliquer chacune des prémisses des règles d'inférence à la base de connaissances (ligne 4). Puis, à matérialiser les connaissances inférées à la base de connaissances $\mathcal{T}$ en exploitant les conclusions des règles (ligne 6). La ligne 9 permet de rajouter les triplets inférés dans la base de connaissances en supprimant les doublons. Enfin, l'algorithme se termine juste après l'itération où aucune nouvelle connaissance implicite n'a été inférée.

Algorithm 1 Saturation

Require: $\mathcal{T}$ `l'ensemble des triplets initiaux,`
Require: $\mathcal{R}$ `l'ensemble des règles d'inférence`
1: **repeat**
2: $\mathcal{T}' \leftarrow \emptyset$
3: **for all** $r \in \mathcal{R}$ **do**
4: $candidats \leftarrow appliquer(premise(r), \mathcal{T})$
5: **for all** $c \in candidats$ **do**
6: $\mathcal{T}' \leftarrow materialiser(conclusion(r), c) \cup \mathcal{T}'$
7: **end for**
8: **end for**
9: $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
10: **until** $\mathcal{T}' - \mathcal{T} = \emptyset$
11: **return** $\mathcal{T}$

Pour illustrer l'exécution de cet algorithme, nous allons considérer la saturation de la base de connaissances de la Table 5.1 par les deux règles métier R_1 et R_2. Comme l'ordre dans lequel les règles sont choisies dans chaque itération n'importe pas, nous considérerons que R_1 est toujours choisie avant R_2.

Première itération: grâce aux triplets 1 et 4 de la table, l'application de la prémisse de R_1 (ligne 4) retourne l'ensemble de candidats suivant.

?a	?eventA	?b	?startB
:event1	525323	:event3	525325

La matérialisation de la conclusion de R_1 sur ce candidat produit le triplet suivant.

7	:event1	:eventPrecede	:event3

L'absence de triplets de la forme {`?a :eventPrecede ?b`} parmi les six triplets dans la base de connaissances, fait que l'application de la prémisse de R_2 ne retourne aucun candidat à cette itération.

L'algorithme rajoute l'unique triplet précédent de la base de connaissances.

Seconde itération: la nouvelle exécution de la prémisse de R_1 produit le même candidat qu'à la précédente itération et produit donc le même triplet.

Grâce aux triplets 1, 3, 5, 6 et le septième triplet inféré à la première itération, l'application de la prémisse de R_2 à la base de connaissances produit l'ensemble des candidats suivant

?a	?startA	?b	?endB	?data
:event1	525319	:event3	525337	10

La matérialisation des motifs de triplets de la conclusion de R_2 sur ce candidat produit les triplets suivants où les URI de la forme `_:BNx` représentent les identifiants des nœuds blancs présents dans les motifs de triplets de la conclusion de R_2:

8	_:BN1	rdf:type	:dataCopy
9	_:BN2	:eventIsRelatedToFunctionality	_:BN1
10	_:BN2	:eventStartAt	525319
11	_:BN2	:eventEndAt	525337

A la fin de l'itération ces nouveaux triplets sont ajoutés à la base de connaissances.

Troisième itération: les applications des prémisses des règles infèrent les mêmes triplets qu'à la première et la seconde itération. Ces triplets sont tous ignorés à la ligne 10 de l'algorithme ce qui vérifie la condition d'arrêt de l'algorithme. La base de connaissances à l'issue de la saturation, contient 11 triplets. Elle a été augmentée de 5 triplets (soit 83% du nombre initial de triplets dans la base de connaissances).

Cependant, la présence de nœuds blancs dans les conclusions des règles métier et l'inférence des relations temporelles posent respectivement des problèmes de terminaison et de performance aux moteurs d'inférence existants.

5.2.2 L'inférence des nouvelles instances dans les règles

Les langages d'expression de règles *Datalog* et *SWRL*, ne proposent pas de syntaxe pour la création de nouvelles instances dans la partie conclusion de la règle. En effet, de manière générale les règles exprimées par ces langages doivent être *safe*, c'est à dire que toutes les variables utilisées dans la conclusion de la règle

doivent être présentes dans la prémisse de la règle. En conséquence, l'inférence consiste généralement à créer des liens entre des instances qui existent déjà.

Pour supporter l'inférence de nouvelles instances, certains moteurs d'inférence étendent ces langages avec des fonctions spécialisées pour considérer les nœuds blancs en conclusion de règle comme des variables. Pendant l'inférence, ces fonctions spécialisées génèrent et proposent des identifiants uniques comme valeurs des nœuds blancs dans la conclusion de la règles. On peut citer la fonction *makeOWLThing* comme un exemple de ce type de fonction pour le langage *SWRL* [Pro08]. Grâce à *makeOWLThing*, la forme suivante de la règle R_2 est considérée comme *safe*.

R_2	Si	Alors

	`?a :eventReadData ?data .`
	`?b :eventWriteData ?data .`
	`?a :eventPrecede ?b .`
	`?a :eventStartAt ?startA .`
	`?b :eventEndAt ?endB .`
	`makeOWLThing(?dc) .`
	`makeOWLThing(?ev)`
		`?dc rdf:type :dataCopy`
		`?ev :eventIsRelatedToFunctionality ?dc`
		`?ev :eventStartAt ?startA`
		`?ev :eventEndAt ?endB`

Le moteur d'inférence *Jena*, étend son langage d'expression des règles (proche de *Datalog*) avec la fonction *makeTemp* qui se base sur le temps système pour créer des identifiants uniques pour les nouvelles instances [Jen].

Toutefois, cette manière de créer les identifiants des nouvelles instances pose le problème de *terminaison*, c'est à dire la création infinie de nouvelles instances pendant la saturation. En effet, chaque fois qu'elle est appliquée, ce type de fonction spécialisée (notamment *makeTemp*) garantit uniquement l'unicité du nouvel identifiant sans vérifier si cette nouvelle instance a déjà été créée à une itération précédente. En conséquence, la règle infère toujours de nouvelles connaissances. Ainsi, la clause d'arrêt de l'algorithme de saturation n'est donc jamais vérifiée et l'algorithme s'exécute à l'infini.

5.2.3 L'inférence de l'ordre temporel entre les évènements

En considérant N instances d'évènements, le nombre de triplets inférés uniquement par la règle R_1 est de :

$$\sum_{i=1}^{N}(N-i) = \frac{N \times (N-1)}{2}$$

Cette quantité de données augmente la taille de la base de connaissances sans apporter une sémantique utile. En effet, le développeur n'est généralement pas intéressé par toutes les relations de précédence d'un évènement avec les autres évènements de la trace. En ne considérant que l'ordre de précédence entre deux évènements consécutifs le nombre de triplets inférés est réduit à $N-1$. Dans ce cas, le triplet (*?a :eventPrecede ?b*) est inféré entre les instances *?a* et *?b*

uniquement si *?b* a le plus petit temps de début supérieur au temps de fin de *?a* comme le précise la règle métier suivante.

R_1	Si	Alors

	`?a :eventEndAt ?endA .`
	`?b :eventStartAt ?startB .`
	`?startB = MIN{ ?start | ?start > ?endA }`
		`?a :eventPrecede ?b`

Cependant, la sémantique de l'opérateur *MIN* ainsi introduit par l'optimisation de l'inférence de l'ordre de précédence, n'est pas supporté par les langage d'expression des règles d'inférence.

Dans la prochaine section, nous présentons l'architecture d'un moteur d'inférence qui adresse ces deux limitations.

5.3 Proposition d'un moteur d'inférence pour les règles métier

Pour utiliser les bases de données relationnelles dans la gestion des bases de connaissances, *Richard Cyganiak* introduit un modèle afin de transformer les requêtes *SPARQL* en des requêtes *SQL* avant de les exécuter sur une table relationnelle `TripleTable`, où chaque tuple {`s p o`} représente un triplet de la base de connaissances [Cyg05].

Nous adaptons ce modèle pour la saturation des bases de connaissances selon l'algorithme 2 qui est une adaptation de l'algorithme 1. La première étape de l'algorithme consiste à créer des requêtes d'insertion SQL à partir des règles d'inférence pris en entrée (lignes 2-4). Ensuite, l'algorithme exécute chacune de ces requêtes sur la table `TripleTable` (lignes 7-10).

La fonction `executeUpdateQuery(q,T)` exécute la requête d'insertion *SQL* `q` sur la table `T` et retourne le nombre de tuples insérés dans la table. La condition d'arrêt de l'algorithme est vérifiée quand aucun nouveau tuple n'est inséré dans la table après une itération sur toutes les requêtes d'insertion issues des règles d'inférence.

5.3.1 Définitions formelles

Dans cette section, nous proposons la définition en algèbre relationnelle (sous forme d'équation où la table `TripleTable` est appelée `TT` pour les raisons de clarté) de la fonction (Ω) qui est utilisée dans l'algorithme 2 pour transformer une règle d'inférence en une requête *SQL*.

Les opérateurs σ, π et $\bowtie$ que nous utilisons dans ces équations correspondent respectivement à la sélection, la projection et à l'équi-jointure de l'algèbre relationnelle. Nous considérons que le renommage des attributs d'une relation est fait par l'opérateur de projection (π). Ainsi, étant donné une relation `R(s,p)`, la projection ci-dessous retourne une relation ayant les attributs `(?a,p,o)` où l'attribut *?s* de la relation R renommé en *?a* et où l'attribut *p* contient uniquement la valeur constante *rdf:type*.

Algorithm 2 saturation sur la table `TripleTable`

Require: `TripleTable`,
Require: $\mathcal{R}$ `l'ensemble des règles d'inférence`
1: $Queries \leftarrow \emptyset$
2: **for all** $r \in \mathcal{R}$ **do**
3: $\quad Queries \leftarrow Queries \cup \{\Omega(r)\}$
4: **end for**
5: **repeat**
6: $\quad n \leftarrow 0$
7: $\quad$ **for all** $q \in Queries$ **do**
8: $\quad\quad n \leftarrow executeUpdateQuery(q, TripleTable) + n$
9: $\quad$ **end for**
10: **until** $n = 0$
11: **return** $TripleTable$

$$\pi_{\substack{?a \leftarrow s \\ p \leftarrow rdf:type}} (R)$$

Nous considérons l'ensemble des identifiants $\Sigma = \{\mathcal{B} \cup \mathcal{V} \cup \mathcal{U}\}$ tel que $\mathcal{B}$, $\mathcal{V}$ et $\mathcal{U}$ sont respectivement les ensembles potentiellement infinis des identifiants des nœuds blancs, des identifiants des variables et des URI.

La réécriture des règles d'inférence en requêtes d'insertion SQL

La fonction (Ω) produit une requête d'insertion pour matérialiser chaque motif de triplet présent dans la conclusion de la règle (Equation (5.1)).

Dans le cas où le motif de triplet à matérialiser ne contient pas de nœuds blancs, la fonction (Ω) effectue une projection sur la relation construite par (δ) à partir de la prémisse de la règle, et renomme les attributs en fonction du motif de triplet à matérialiser (Equation (5.15)).

Dans le cas où le motif de triplet contient des nœuds blancs, la fonction τ est utilisée pour générer des identifiants des nouvelles instances. L'Equation (5.3) représente le cas où nous avons un nœud blanc dans la position de *sujet* du motif de triplet, ce cas se généralise pour toutes les autres positions.

Pour éliminer les doublons, la requête d'insertion ainsi produite effectue une différence entre la relation issue de la prémisse et la table `TripleTable`.

$$\Omega(\{P \rightarrow \{r_1, ..., r_m\}\}) = \begin{cases} \Omega(\{P \rightarrow r_1\}) \\ ... \\ \Omega(\{P \rightarrow r_m\}) \end{cases} \tag{5.1}$$

$$\begin{array}{r} \forall\ x, y, z \in \{\mathcal{V} \cup \mathcal{U}\}, \\ \Omega(\{P \rightarrow \{x\ y\ z\}\}) \end{array} = \begin{array}{l} \textit{insert into TT}\ (s, p, o) \\ \left(\pi_{\substack{s \leftarrow x \\ p \leftarrow y \\ o \leftarrow z}}\ \delta(P) \right) - TT \end{array} \tag{5.2}$$

$$\begin{array}{r} \forall\ b \in \mathcal{B},\ \forall\ y, z \in \{\mathcal{V} \cup \mathcal{U}\}, \\ \Omega(\{p \rightarrow \{b\ y\ z\}\}) \end{array} = \begin{array}{l} \textit{insert into TT}\ (s, p, o) \\ \left(\pi_{\substack{s \leftarrow \tau(b,\ P) \\ p \leftarrow y \\ o \leftarrow z}}\ \delta(P) \right) - TT \end{array} \tag{5.3}$$

La génération des identifiants des nouvelles instances

La fonction $\tau(b, P)$ où P est la prémisse d'une règle et $b \in \mathcal{B}$ est un identifiant d'un nœud blanc dans la règle, est une sorte de fonction de *skolem*. En effet, $\tau(b, P)$ est une fonction injective qui retourne un identifiant unique à partir du nom du nœud blanc (b) et des valeurs de chaque tuple retournées par la requête de sélection issue de la prémisse de la règle. Elle garantit que le même identifiant soit créé pour la même prémisse et le même nom du nœud blanc. Nous avons la garantie que l'algorithme se termine car les règles métier ne contiennent pas des dépendances susceptibles de poser le problème de l'*expansion infinie* (confère Section 3.4.4).

La réécriture des prémisses des règles de sélection SQL

Dans les prochaines équations, nous définissons la fonction (δ) pour transformer la prémisse de la règle en une requête de sélection *SQL* à exécuter sur la table `TripleTable`.

La réécriture d'une conjonction de motifs de triplets correspond à une équi-jointure sur les relations issues des réécritures respectives des motifs de triplets de la conjonction (Equation (5.4)). Les équi-jointures exploitent les renommages des attributs des relations par les opérateurs de projection (π).

La conjonction d'un motif (ou une conjonction de motifs) avec une condition correspond à une sélection sur la relation issue de la réécriture de la conjonction (Equation (5.5)).

$$\delta(m_1\ .\ m_2) = \delta(m_1) \bowtie \delta(m_2) \tag{5.4}$$

$$\delta(m\ .\ (cond)) = \sigma_{cond}(\delta(m)) \tag{5.5}$$

L'équation de (5.6) transforme un motif de triplet de la prémisse, constitué uniquement de variables, en une projection de la table `TripleTable` avec un

renommage des attributs {s,p,o} de la table `TripleTable` pour correspondre aux noms des variables du motif de triplet.

$$\begin{aligned} \forall\ ?s, ?p, ?o \in \mathcal{V},\\ \delta(\{?s\ ?p\ ?o\}) \end{aligned} \quad = \quad \pi_{\substack{?s \leftarrow s\\ ?p \leftarrow p\\ ?o \leftarrow o}} (TT) \tag{5.6}$$

Dans le cas où une ou plusieurs parties du motifs de triplet ne sont pas des variables, ces parties du motif permettent de préciser la condition de la sélection sur la table `TripleTable`. Les équations de (5.7) à (5.13) présentent les requêtes de sélection issues de toutes les combinaisons possibles de motifs de triplets. Par exemple, l'équation (5.13) correspond à la sélection du triplet $\{s_i\ \ p_i\ \ o_i\}$ dans la table.

$$\begin{aligned} \forall\ ?p, ?o \in \mathcal{V},\ \forall\ s_i \in \mathcal{U},\\ \delta(\{s_i\ ?p\ ?o\}) \end{aligned} \quad = \quad \pi_{\substack{s_i \leftarrow s\\ ?p \leftarrow p\\ ?o \leftarrow o}} (\sigma_{s=s_i}(TT)) \tag{5.7}$$

$$\begin{aligned} \forall\ ?s, ?o \in \mathcal{V},\ \forall\ p_i \in \mathcal{U},\\ \delta(\{?s\ p_i\ ?o\}) \end{aligned} \quad = \quad \pi_{\substack{?s \leftarrow s\\ p_i \leftarrow p\\ ?o \leftarrow o}} (\sigma_{p=p_i}(TT)) \tag{5.8}$$

$$\begin{aligned} \forall\ ?s, ?p \in \mathcal{V},\ \forall\ o_i \in \mathcal{U},\\ \delta(\{?s\ ?p\ o_i\}) \end{aligned} \quad = \quad \pi_{\substack{?s \leftarrow s\\ ?p \leftarrow p\\ o_i \leftarrow o}} (\sigma_{o=o_i}(TT)) \tag{5.9}$$

$$\begin{aligned} \forall\ ?o \in \mathcal{V},\ \forall\ s_i,\ p_i \in \mathcal{U},\\ \delta(\{s_i\ p_i\ ?o\}) \end{aligned} \quad = \quad \pi_{\substack{s_i \leftarrow s\\ p_i \leftarrow p\\ ?o \leftarrow o}} (\sigma_{s=s_i \wedge p=p_i}(TT)) \tag{5.10}$$

$$\begin{aligned} \forall\ ?p \in \mathcal{V},\ \forall\ s_i,\ o_i \in \mathcal{U},\\ \delta(\{s_i\ ?p\ o_i\}) \end{aligned} \quad = \quad \pi_{\substack{s_i \leftarrow s\\ ?p \leftarrow p\\ o_i \leftarrow o}} (\sigma_{s=s_i \wedge o=o_i}(TT)) \tag{5.11}$$

$$\begin{aligned} \forall\ ?s \in \mathcal{V},\ \forall\ p_i,\ o_i \in \mathcal{U},\\ \delta(\{?s\ p_i\ o_i\}) \end{aligned} \quad = \quad \pi_{\substack{?s \leftarrow s\\ p_i \leftarrow p\\ o_i \leftarrow o}} (\sigma_{p=p_i \wedge o=o_i}(TT)) \tag{5.12}$$

$$\begin{aligned} \forall\ s_i,\ p_i,\ o_i \in \mathcal{U},\\ \delta(\{s_i\ p_i\ o_i\}) \end{aligned} \quad = \quad \pi_{\substack{s_i \leftarrow s\\ p_i \leftarrow p\\ o_i \leftarrow o}} (\sigma_{s=s_i \wedge p=p_i \wedge o=o_i}(TT)) \tag{5.13}$$

5.3.2 Illustration du modèle pour l'inférence des nouvelles instances

Pour illustrer notre modèle, nous présentons ci dessous les équivalences en *SQL* de la requête d'insertion produite par $(\Omega(R_2))$.

La table suivante anotte les motifs de triplets dans la prémise et la conclusion de R_2. Pour des raisons de clarté, nous allons considérer $P = m_1 . m_2 . m_3 . m_4 . m_5$ comme étant la prémisse de R_2.

R_2	Si		Alors
m_1	`?a :eventReadData ?data .`	m_6	`_:dc rdf:type :dataCopy`
m_2	`?b :eventWriteData ?data .`	m_7	`_:ev :eventIsRelatedToFunctionality _:dc`
m_3	`?a :eventPrecede ?b .`	m_8	`_:ev :eventStartAt ?startA`
m_4	`?a :eventStartAt ?startA .`	m_9	`_:ev :eventEndAt ?endB`
m_5	`?b :eventEndAt ?endB`		

La conclusion de la règle contient 4 motifs de triplets, en conséquence autant de réquêtes d'insertion sont produites:

$$\text{Equation (5.1)} \quad \Omega(R_2) = \begin{cases} \Omega(\{P \rightarrow m_6\}) \\ \Omega(\{P \rightarrow m_7\}) \\ \Omega(\{P \rightarrow m_8\}) \\ \Omega(\{P \rightarrow m_9\}) \end{cases}$$

Considérons la requête d'insertion produite par le premier motif de la conclusion. Pour chaque résultat de réquête de sélection issue de la prémisse, la requête d'insertion génère des identifiants uniques pour les nouvelles instances ($\tau(_:dc, P)$) de la classe `dataCopy` et les ajouter dans la table `TripleTable` en supprimant les doublons:

$$\text{Equation (5.3)} \quad \Omega(\{P \rightarrow m_6\}) = \overset{insert\ into\ TT\ (s,p,o)}{\left(\pi_{\substack{s \leftarrow \tau(_:dc,\, P) \\ p \leftarrow rdf:type \\ o \leftarrow :dataCopy}} \delta(P) \right)} - TT$$

Les quatre conjonctions de la prémisse de la règle produisent une requête de sélection contenant quatre auto-jointures sur la table `TripleTable`:

$$\text{Equation (5.4)} \quad \delta(P) = \delta(m_1) \bowtie \delta(m_2) \bowtie \delta(m_3) \bowtie \delta(m_4) \bowtie \delta(m_5)$$

$$\text{Equation (5.8)} \quad \begin{aligned} \delta(m_1) &= \pi_{\substack{?a \leftarrow s \\ eventReadData \leftarrow p \\ ?data \leftarrow o}} (\sigma_{p=eventReadData}(TT)) \\ \delta(m_2) &= \pi_{\substack{?b \leftarrow s \\ eventWriteData \leftarrow p \\ ?data \leftarrow o}} (\sigma_{p=eventWriteData}(TT)) \\ \delta(m_2) &= \pi_{\substack{?a \leftarrow s \\ eventPrecede \leftarrow p \\ ?b \leftarrow o}} (\sigma_{p=eventPrecede}(TT)) \\ \delta(m_2) &= \pi_{\substack{?a \leftarrow s \\ eventStartAt \leftarrow p \\ ?startA \leftarrow o}} (\sigma_{p=eventStartAt}(TT)) \\ \delta(m_2) &= \pi_{\substack{?b \leftarrow s \\ eventEndAt \leftarrow p \\ ?endB \leftarrow o}} (\sigma_{p=eventEndAt}(TT)) \end{aligned}$$

L'expression algébrique ci-dessous est la requête d'insertion issue de la règle R_2 qui correspond uniquement à la matérialisation du premier motif de la conclusion de la règle. Les trois autres motifs de la conclusion de R_2 produisent des requêtes d'insertion qui ont la même partie sélection des données, mais qui diffèrent au niveau de la construction des données à insérer dans la table `TripleTable`:

$$
\Omega(\{P \to m_6\}) = \begin{array}{l} insert\ into\ TT\ (s, p, o) \\ \left(\pi_{\substack{s \leftarrow \tau(_:dc,\, P) \\ p \leftarrow rdf:type \\ o \leftarrow :dataCopy}} \left(\begin{array}{c} \pi_{\substack{?a \leftarrow s \\ eventReadData \leftarrow p \\ ?data \leftarrow o}} (\sigma_{\ p\, =\, eventReadData}\ (TT)) \\ \bowtie \\ \pi_{\substack{?b \leftarrow s \\ eventWriteData \leftarrow p \\ ?data \leftarrow o}} (\sigma_{\ p\, =\, eventWriteData}\ (TT)) \\ \bowtie \\ \pi_{\substack{?a \leftarrow s \\ eventPrecede \leftarrow p \\ ?b \leftarrow o}} (\sigma_{\ p\, =\, eventPrecede}\ (TT)) \\ \bowtie \\ \pi_{\substack{?a \leftarrow s \\ eventStartAt \leftarrow p \\ ?startA \leftarrow o}} (\sigma_{\ p\, =\, eventStartAt}\ (TT)) \\ \bowtie \\ \pi_{\substack{?b \leftarrow s \\ eventEndAt \leftarrow p \\ ?endB \leftarrow o}} (\sigma_{\ p\, =\, eventEndAt}\ (TT)) \end{array} \right) \right) - TT \end{array}
$$

Dans la pratique, nous utilisons le *SQL* pour exprimer les requêtes d'insertion issues des réécritures des règles. Ainsi, la table suivante représente la version *SQL* de $\Omega(R_2)$ où les valeurs des attributs de la table `TripleTable` sont tous des `VARCHAR`.

On peut observer les cinq instances de la table `TripleTable` qui correspondent respectivement aux cinq motifs de triplets de la prémisse (lignes 4-8). Les données de chaque motif de la prémisse sont retournés par les sélections sur l'attribut `p` (ligne 9-13). Les lignes 14 à 18 représentent les auto-jointures issues des conjonctions entre les motifs de triplets de la prémisse. La suppression des doublons est assurée par la clause `EXCEPT`, elle est appliquée entre les résultats de la sélection et l'ensemble des données de la table `TripleTable` (ligne 19 et 20).

R_2	
1	`INSERT INTO TripleTable (s,p,o)`
2	`SELECT ('_:dc' + (P1.p + P1.o) + (P2.p + P2.o)`
	`+ (P3.p + P3.o) + (P4.p + P4.o) + (P5.p + P5.o)) AS s,`
3	`'rdf:type' AS p, 'dataCopy' AS o`
4	`FROM TripleTable P1,`
5	`TripleTable P2,`
6	`TripleTable P3,`
7	`TripleTable P4,`
8	`TripleTable P5`
9	`WHERE P1.p = 'eventReadData'`
10	`AND P2.p = 'eventWriteData'`
11	`AND P3.p = 'eventPrecede'`
12	`AND P4.p = 'eventStartAt'`
13	`AND P5.p = 'eventEndAt'`
14	`AND P1.o = P2.o`
15	`AND P1.s = P3.s`
16	`AND P2.s = P5.s`
17	`AND P1.s = P3.s`
18	`AND P2.s = P3.o`
19	`EXCEPT`
20	`SELECT s, p, o FROM TripleTable`

L'expression suivante correspond à l'implémentation *SQL* de le fonction τ. En considérant (+) comme un opérateur de concatenation des valeurs des tuples (qui sont de type `VARCHAR`), l'expression construit l'identifiant de la nouvelle instance en concaténant l'identifiant (`_:dc`) du nœud blanc dans la règle aux valeurs des propriétés et des objets de chaque tuple résultat de la requête de sélection.

```
('_:dc' + (P1.p + P1.o) + (P2.p + P2.o) + (P3.p + P3.o) + (P4.p +
                        P4.o) + (P5.p + P5.o))
```

La section A.1 contient les requêtes d'insertion *SQL* issues des règles d'inférence `rdfs2, rdfs3, rdfs5, rdfs7, rdf9` et `rdfs11`.

5.3.3 L'inférence des relations temporelles entre les instances consécutives

L'inférence des relations temporelles entre les événements s'inspire également de l'Algorithme 1. Notre approche consiste à trier les événements de la base de connaissances en fonction de l'ordre temporel désiré; Puis à inférer un triplet pour chaque pair d'événements consécutifs (au sens de l'ordre temporel désiré).

L'algorithme 3 illustre notre approche. L'algorithme prend en entrée: le critère de tri des évènements (*criteria*) qui permet de restreindre les instances des évènements à considérer, une fonction (*order_condition*) qui représente la condition que doivent reunir les instances consécutives pour être liées, la propriété temporelle à inférer entre les événements (p) et la base de connaissances.

La requête de selection *select_event_in_order* retourne les instances des évènements contenus dans la base de connaissances en respectant le critère de tri (ligne 1). La ligne 7 représente l'ajout du triplet construit à partir de deux instances consécutives lorsque la condition est vérifiée entre les deux instances (ligne 6).

Algorithm 3 inférence de l'ordre temporel entre les instances consécutives d'évènements dans `TripleTable`

Require: $criteria$ `ordre entre les événement,`
Require: $order_condition$ `ordre entre les événement,`
Require: p `propriété à inférer,`
Require: $\mathcal{T}$ `la base de connaissances,`
1: $\mathcal{E} \leftarrow select_event_in_order(TripleTable, criteria)$
2: $\mathcal{T}' \leftarrow \emptyset$
3: **repeat**
4: $i \leftarrow 0$
5: **while** $i < (|\mathcal{E}| - 1)$ **do**
6: **if** $ordering_condition(E(i), E(i+1)) = true$ **then**
7: $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{\langle E(i), p, E(i+1) \rangle\}$
8: **end if**
9: $i \leftarrow i + 1$
10: **end while**
11: $\mathcal{T}' \leftarrow \mathcal{T}' - \mathcal{T}$
12: $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
13: **until** $|\mathcal{T}'| = 0$
14: **return** $\mathcal{T}$

5.3.4 Illustration du modèle pour l'inférence de l'ordre temporelle

Nous allons illustrer notre approche pour l'inférence de l'ordre de précédence entre toutes les instances consécutives d'évènements. Nous allons pour cela utiliser la règle R_1 (telle que nous l'avons modifié dans la Section 5.2.3).

R_1	Si		Alors
m_1	`?a :eventEndAt ?endA .`	m_3	`?a :eventPrecede ?b`
m_2	`?b :eventStartAt ?startB .`		
$cond_1$	$?startB = min\{?start \mid ?start > \ ?endA\}$		

L'ordre de précédence dépend des temps de fin et de début des évènements consécutifs. Ainsi, la requête de sélection retourne tous les événements et leur temps de début et de fin. Les évènements sont triés par ordre croissant de temps de fin.

L'équation (5.14) représente l'expression relationnelle de la réquête de sélection des instances d'évènements de la table `TripleTable` pour l'inférence de l'ordre

de précédence. Il s'agit d'une jointure des sélections sur les propriétés `eventStartAt` et `eventEndAt`.

$$\left(\pi_{\substack{?a \leftarrow s \\ ?e \leftarrow o}} \sigma_{\substack{p = eventEndAt \\ orderby\ o}} (TT) \right) \bowtie \left(\pi_{\substack{?b \leftarrow s \\ ?s \leftarrow o}} \sigma_{\substack{p = eventStartAt \\ orderby\ o}} (TT) \right) \tag{5.14}$$

L'équation (5.15) représente la condition à reunir pour inférer la précédence. L'équation considère une pair de tuples consécutifs (t_1, t_{i+1}) provenant de la requête de sélection précédente (Equation (5.14)). L'ordre de précédence, tel que défini par *Allen et al*, est établi lorsque le premier évènement se termine avant le début de l'évènement suivant.

$$\pi_{s \leftarrow ?a} (t_i) < \pi_{s \leftarrow ?b} (t_{i+i}) \tag{5.15}$$

La propriété `eventPrecede` est inférée entre les deux instances quand cette condition est vérifiée.

En pratique l'algorithme peut être implémenté pour chacune des 13 conditions introduites par *Allen et al* pour les relations temporelles [All83]. La modification du critère de sélection permet d'inférer les relations temporelles entre les événements dans diverses conditions. Par exemple, la condition peut préciser le processeur (respectivement la tâche) où l'évènement s'éxécute pour inférer la propriété `eventPrecedeInCPU` (respectivement `eventPrecedeInTask`). De même, pour l'inférence des relations temporelles à des niveaux d'abstraction différents, la condition peut cibler uniquement les évènements liés à des fonctionnalités (respectivement aux anomalies) pour inférer la relation temporelle `eventPrecedeInFunctionality` (respectivement `eventPrecedeInAnomaly`).

Ce dernier type de relations temporelles est essentiel pour construire des règles métier basées sur des niveaux d'abstraction métier issues d'autres règles métier.

Nous avons désormais une approche de saturation qui permet de résoudre les limitations de performance de l'inférence des relations temporelles entre les évènements et qui ne pose pas de problème terminaison. Dans la prochaine section, nous allons comparer des implémentations de cette approche à une stratégie de saturation *ad-hoc* basée sur le moteur *Jena*. L'objectif de cette comparaison est de déterminer le meilleur compromis à faire pour la saturation des règles d'inférence pour l'analyse des traces.

5.4 Les expérimentations

L'objectif de cette section est de comparer deux approches de saturation, l'une basée sur la `TripleTable` et l'autre basée sur le moteur d'inférence *Jena*.

5.4.1 Le contexte expérimental

Les stratégies de saturation

A cause du problème de terminaison que pose la saturation des règles métier au moteur d'inférence *Jena*, nous allons utiliser une stratégie *ad-hoc* pour notre étude comparative des moteurs d'inférence. Cette stratégie consiste à transformer toutes les règles d'inférence métier en des requêtes *SPARQL* de type `CONSTRUCT` et à les exécuter une seule fois afin d'éviter la création des doublons dans les identifiants des nouvelles instances.

Nous ne modifions pas la saturation des règles *RDFS* qui est nativement supportée par *Jena*. Pour être comparable, nous allons également adapter l'algorithme 2 pour exécuter chaque requête issue des règles d'inférence métier une seule fois.

Les moteurs d'inférence

Nous allons comparer le moteur *Jena* à deux implémentations de l'algorithme 2 basées respectivement sur le column-store *MonetDB* et sur le row-store *PostGreSQL*. L'objectif est de comparer également les deux modèles physiques utilisés dans les bases de données à savoir le *column-store* et le *row-store* pour déterminer lequel de ces modèles est le plus adapté à la saturation en base de données.

Nous allons donc comparer *Jena* à *monetdbIE* (l'implémentation de l'algorithme 2 basée sur le *column-store* libre *MonetDB*) et le moteur d'inférence *pgsqlIE* (l'implémentation de l'algorithme 2 basée sur le row-store libre *PostGreSQL*).

Pour optimiser les requêtes de sélection au moment de la saturation sur la table `TripleTable`, une fois que les triplets initiaux sont stockés dans la table, nous créons un index sur la colonne p avec la clause *SQL* suivante :

```
CREATE INDEX index_p ON TripleTable (p);
```

Dans le cas de *PostGreSQL*, cet index est créé sous la forme d'un arbre B^+. *MonetDB* par contre néglige la clause *SQL* de création d'index et choisit plutôt de détecter, pendant le chargement, si les valeurs de la colonne sont ordonnées afin de mettre en place des mécanismes optimisés de parcours des valeurs des colonnes des tables [Mon08a].

Le protocole expérimental

Nous allons faire cette comparaison selon un protocole qui simule les conditions réelles d'un scénario d'analyse des traces d'exécution d'une application embarquée appelée `ts_record`. Cette application sert à l'enregistrement sur disque USB d'un flux vidéo en provenance d'internet.

Dans ce scénario, le développeur construit progressivement de nouveaux niveaux d'abstractions pour matérialiser ses propres connaissances métier sur deux fonctionnalités de l'application, la fonctionnalité d'écriture des données lues sur internet dans des mémoires tampons IP (`dataFromEthernet2IP`) et la fonctionnalité d'écriture de ces données en mémoire tampons dans la mémoire centrale (`dataFromIP2Memory`).

Le protocole se déroule en cinq étapes.

1. L'analyste charge les triplets RDF initiaux issus des évènements de trace dans le triple store.

2. L'analyste exécute les requêtes *SPARQL ScenarioQ1* et *ScenarioQ2* pour rechercher les instances des concepts `dataFromEthernet2IP` et `dataFromIP2Memory`

ScenarioQ1

```
SELECT ?a ?start ?end
WHERE {
    ?a  rdf:type videcom:Event .
    ?a  videcom:eventIsRelatedToFunctionality ?f1 .
    ?f1 rdf:type videcom:dataFromEthernet2IP .
    ?a  videcom:eventStartAt ?start .
    ?a  videcom:eventEndAt ?end
}
```

ScenarioQ2

```
SELECT ?b ?start ?end
WHERE {
    ?b  rdf:type videcom:Event .
    ?b  videcom:eventIsRelatedToFunctionality ?f2 .
    ?f2 rdf:type videcom:dataFromIP2Memory .
    ?b  videcom:eventStartAt ?start .
    ?b  videcom:eventEndAt ?end
}
```

3. Face à l'absence de ce niveau d'abstraction dans la trace, le développeur traduit les connaissances métier qui correspondent à ces niveaux d'abstraction, sous la forme des deux règles d'inférence pour les matérialiser dans la trace.

La fonctionnalité `dataFromEthernet2IP` est assurée par l'exécution de deux interruptions précises (*gic_eth0, net_rx_action*) par l'application `ts_record`. Ainsi, la première règle *R01* lie une nouvelle instance d'évènement (_:s) à une nouvelle instance (_:n) de la classe `dataFromEthernet2IP`, chaque fois qu'elle trouve une des instances consécutives correspondant à ces interruptions.

```
R01
{
  ?e1 videcom:runningInterruption videcom:gic_eth00 .
  ?e2 videcom:runningSoftIRQ videcom:net_rx_action0 .
  ?e1 videcom:eventPrecedeInCPU ?e2 .
  ?e1 videcom:eventStartAt ?start .
  ?e2 videcom:eventEndAt ?end .
} → {
        _:n rdf:type videcom:dataFromEthernet2IP .
        _:s videcom:eventIsRelatedToFunctionality _:n .
        _:s videcom:eventStartAt ?start .
        _:s videcom:eventEndAt ?end .
        _:s videcom:eventOccurredOnCPU ?cpu
}
```

La seconde règle *R02* lie une nouvelle instance d'évènement à une nouvelle instance de la fonctionnalité (_:n) chaque fois quelle trouve une séquence de trois appels systèmes consécutifs (*sys_poll, sys_read, sys_write*).

```
R2
{
  ?e1 videcom:SystemCallIsExecutedDuringTask videcom:ts_record0 .
  ?e1 videcom:runningSystemCall videcom:sys_poll0 .
  ?e2 videcom:runningSystemCall videcom:sys_read0 .
  ?e3 videcom:runningSystemCall videcom:sys_write0 .
  ?e2 videcom:eventPrecedeInTask ?e3 .
  ?e1 videcom:eventPrecedeInTask ?e2 .
  ?e1 videcom:eventStartAt ?start .
  ?e3 videcom:eventEndAt ?end .
} → {
  _:n rdf:type videcom:dataFromIP2Memory .
  _:s videcom:eventIsRelatedToFunctionality _:n .
  _:s videcom:eventStartAt ?start .
  _:s videcom:eventEndAt ?end .
  _:s videcom:eventOccurredOnCPU ?cpu
}
```

4. Le moteur d'inférence effectue la saturation en considérant dans l'ordre les règles métier *R01* et *R02* ainsi que les règles *RDFS* (*rdfs2, rdfs3, rdfs5, rdfs7, rdfs9* et *rdfs11*).
5. L'analyste exécute de nouveau les requêtes *ScenarioQ1* et *ScenarioQ2* et accède aux nouveaux niveaux d'abstraction.

Les traces d'exécution

Nous allons exécuter ce protocole sur 3 fichiers au format *N-TRIPLE*[1] contenant chacun des triplets issus de trois traces d'exécution de l'application `ts_record`.

[1] N-Triple est un format simple de représentation des triplets RDF dans un fichier. Le format représente chaque triplet sur une ligne du fichier en séparant les trois URIs associés par des espaces. On distingue d'autres formats tels que RDF/XML, TURTLE et N3

Les fichiers nommés *1m*, *2m* et *5m* contiennent respectivement 1, 2 et 5 millions de triplets issus des évènements de ces traces.

La machine

Nous exécutons les expériences sur la même machine que nous avons utilisé pour le benchmark des triplestores (confère Section 4.2.1).

5.4.2 Comparaison des temps de saturation

La première ligne de courbe de la Figure 5.1 représente le temps de création des instances de la fonctionnalité `dataFromEthernet2IP` par la règle *R01*.

Jena est plus rapide parce qu'il ne construit pas directement les nouveaux triplets créés par la requête `CONSTRUCT` correspondant à la règle métier *R01*. Jena reporte la matérialisation de ces triplets au moment où une requête sollicitera explicitement ces triplets. Cette stratégie permet à *Jena* d'optimiser l'utilisation de la mémoire en ne matérialisant les données que lorsque c'est nécessaire.

PGSQLIE est moins performant que *MonetDBIE* à cause du coût de mise à jour de ses index pendant l'insertion des nouveaux triplets.

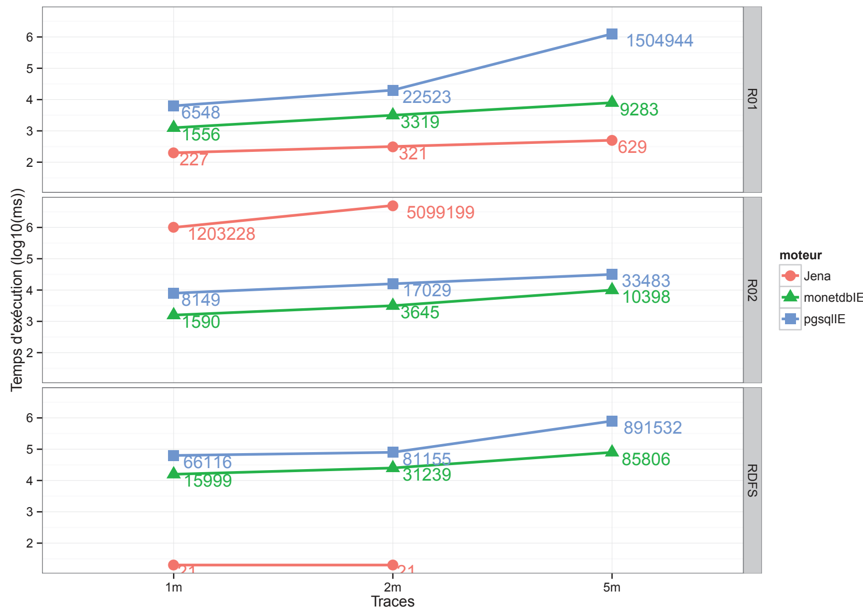


Figure 5.1: Evolution des temps de saturation

La seconde ligne de courbe de la Figure 5.1 représente le temps de création des instances de `dataFromIP2Memory` par la règle *R02*.

Dans le cas des traces *1m* et *2m*, nous observons que l'exécution, par Jena, de la requête `CONSTRUCT` qui correspond à la règle métier *R02*, est plus lente d'un facteur de 4 en comparaison à l'exécution de la requête `CONSTRUCT` de la règle *R01* et de l'inférence des règles *RDFS*.

Nous avons interrompu l'exécution de la requête sur la trace *5m* après 9 heures d'exécution sans résultat à cause de la conjonction des motifs de la règle *R02* ci-dessous, qui déclenche la matérialisation des triplets *implicitement* créés par la règle *R01*.

```
(?e1 videcom:eventStartAt ?start .  ?e3 videcom:eventEndAt ?end)
```

Cette matérialisation prend autant de temps car les propriétés (`:eventStartAt` et `:eventEndAt`), présentes dans ces motifs, ne sont pas sélectives car elles sont produites pour chaque évènement dans la trace.

Nous remarquons également une amélioration, d'un facteur de 2, des performances de *PGSQLIE* sur la trace *5m* en comparaison au temps d'exécution de la règle *R01*. L'explication est que les performances des mises à jour des index se détériorent avec le nombre croissant de données à insérer. En effet, la trace *5m* après saturation contient 87 848 nouvelles instances pour `dataFromEthernet2IP` contre 1 061 pour `dataFromIP2Memory`, soit une écriture en mémoire pour 82 écritures dans les buffers IP (ce qui est normal car la mémoire centrale est plus grande que celle des buffers internet).

Pour l'insertion de grands volumes de données, il est généralement recommandé de faire un chargement massif (ou *bulk load*) des données stockées dans un ou plusieurs fichiers à l'aide des requêtes *SQL* de type `COPY`, ou de procéder par des insertions multiples (requêtes INSERT INTO) en veillant à supprimer les index avant les insertions et à les reconstruire à la fin des insertions pour éviter le coût de leurs mise-à-jour.

Nous avons effectué une expérimentation de cette stratégie en supprimant l'index (de type arbre B^{+}) de *PostGreSQL* avant l'exécution des requêtes d'insertion pour la saturation. Cette stratégie s'est révélée inefficace et nous avons dû arrêter le moteur *pgsqlIE* après 2 heures de saturation sur la trace *1m*. En effet, en l'abscence des index, les accès aux données ont des performances de l'ordre de la taille de la table car ils se font par des *full_scan* complets de la table `TripleTable`.

Dans la troisième courbe de la Figure 5.1, *Jena* décide de nouveau de ne pas matérialiser les triplets inférés par les règles RDFS. Nous observons que *monetdbIE* est resté stable dans les temps d'exécution des règles d'inférence métier et RDFS. Contrairement à *PostGreSQL*, il ne subit pas de coût de mise à jour des index, ce qui lui permet d'avoir de bonnes performances sur la trace *5m* (où il est plus rapide que *pgsqlIE* d'un facteur 2 pour la règle *R01*).

5.4.3 Comparaison des temps de réponse aux requêtes

Les résultats de cette section permettent d'observer la réactivité des moteurs d'inférence dans la réponse aux requêtes avant et après la saturation. La Figure

5.2 présente les temps de réponse des moteurs d'inférence aux requêtes *ScenarioQ1* et *ScenarioQ2* avant (courbe `Execution 1` de la Figure 5.2) et après la saturation (courbe `Execution 2` de la Figure 5.2).

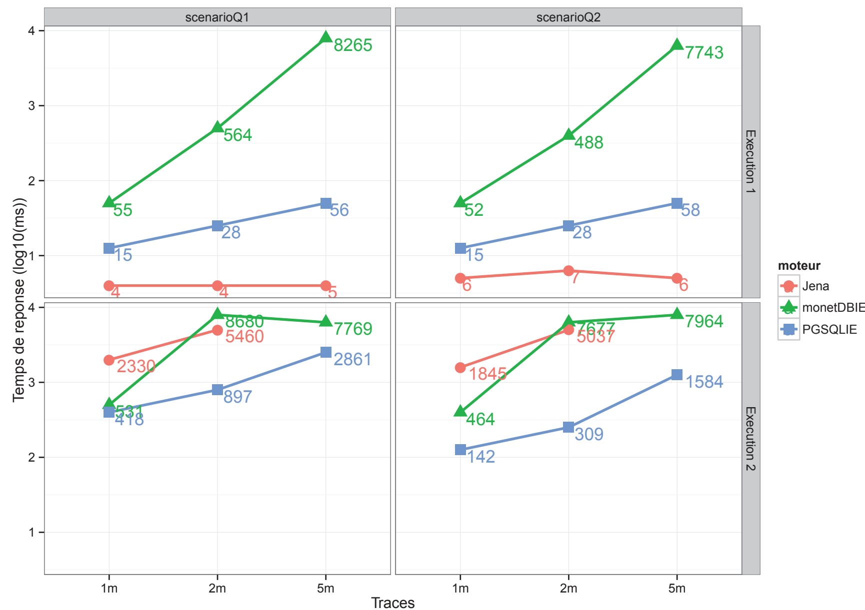


Figure 5.2: Temps de réponse des moteurs d'inférence aux requêtes *ScenarioQ1* et *ScenarioQ2* avant et après la saturation

Avant la saturation, le moteur *Jena* est le plus rapide. La raison est que les triplets sont tous stockés entièrement en mémoire. Grâce à son index, *pgsqlIE* a un temps de réponse plus court que *monetdbIE*.

Après la saturation, le moteur *Jena* perd en efficacité à cause des matérialisations éventuelles de triplets. Le constat selon lequel *monetdbIE* est moins performant que *pgsqlIE* s'explique par le fait que l'accès aux valeurs des tables dans *MonetDB* est uniquement optimisé dans le cas où lesdites valeurs sont triées. Or, l'insertion des nouveaux triplets ne garantit pas le tri des valeurs des colonnes. Ainsi, l'accès aux données de la table `TripleTable` est moins performant sur *MonetDB* que sur *PostGreSQL* où des index sont utilisés.

En conclusion, la saturation à l'aide du modèle `TripleTable` est plus rapide sur *MonetDB* que sur *PostGreSQL*. Cependant, grâce aux index, l'exécution des requêtes sur le modèle `TripleTable` implementé en *PostGreSQL* est plus rapide que l'implémentation de ce modèle en *MonetDB*. Toutefois dans le précédent chapitre, nous avons observé que l'approche de partitionnement vertical est le meilleur compromis pour l'interrogation des connaissances issues de VIDECOM. Dans la prochaine section, nous allons illustrer l'utilisation de l'approche de

partitionnement vertical dans la saturation. Nous allons ensuite illustrer le passage à l'échelle de la saturation sur des traces plus grandes.

5.5 La saturation à grande échelle

Nous avons vu dans le chapitre précédent que le partitionnement vertical consistait à regrouper les triplets dans plusieurs tables correspondant chacune à une propriété de l'ontologie. On peut simplement imaginer que les triplets sont stockés dans plusieurs tables `TripleTable`$_i$ similaire à `TripleTable`, et que chacune de ces tables contient les triplets correspondant à la propriété p_i de l'ontologie.

5.5.1 Le modèle de saturation sur le partitionnement vertical

L'idée de la saturation des connaissances stockées selon l'approche de partitionnement vertical (`VerticalPartitioning`), est la même que la saturation sur l'approche `TripleTable`. La différence se trouve uniquement dans le choix de la table pour les requêtes de sélection de la partie prémisse et la table d'insertion pour les requêtes d'insertion.

De manière formelle, soit $\mathcal{P}$ l'ensemble des propriétés de l'ontologie, et Ω' la fonction qui retourne une requête d'insertion SQL pour insérer les triplets inférés dans les differentes tables du partitionnement vertical. Nous allons illustrer la différence entre les fonction Ω et Ω' sur la règle `rdfs7`.

$$?a\ rdfs{:}subPropertyOf\ ?b\ . \quad \rightarrow \quad ?x\ ?b\ ?y$$
$$?x\ ?a\ ?y$$

$\Omega(rdfs7)$ produit une seule requête d'insertion SQL sur l'unique table `TripleTable` (nommée TT) dans l'équation (5.16).

$$\Omega(rdfs7) = \begin{array}{l} insert\ into\ TT\ (s,p,o) \\ \left(\pi_{\substack{s \leftarrow ?x \\ p \leftarrow ?b \\ o \leftarrow ?y}} \left(\pi_{\substack{?a \leftarrow s \\ rdfs:subPropertyOf \leftarrow p \\ ?b \leftarrow o}} (\sigma_{p = rdfs:subPropertyOf}(TT)) \bowtie \pi_{\substack{?a \leftarrow s \\ ?b \leftarrow p \\ ?y \leftarrow o}}(TT) \right) \right) - TT \end{array} \tag{5.16}$$

A cause de la variable `?b` dans la conclusion, la fonction $\Omega'(rdfs7)$ retourne une requête d'insertion *SQL* pour chaque $TT_{\{?b\}} \in \mathcal{P}$ (Equation (5.17)). De même, étant donné que la table de selection dans le deuxième motif de la prémisse est une variable (`?a`), la requête fait une union des requêtes de selection sur toutes les tables.

$$\Omega'(rdfs7) = \begin{array}{l} \forall\ ?b\ \in \mathcal{P} \\ insert\ into\ TT_{\{?b\}}\ (s,p,o) \\ \left(\begin{array}{ll} \pi_{\substack{s \leftarrow ?x \\ p \leftarrow ?b \\ o \leftarrow ?y}} & \\ \bigcup_{p_i \in \mathcal{P}} & \left(\begin{array}{l} \pi_{\substack{?a \leftarrow s \\ rdfs:subPropertyOf \leftarrow p \\ ?b \leftarrow o}} \\ (\sigma_{o=?b}\ (TT_{subPropertyOf}) \\ \quad \bowtie \\ \pi_{\substack{?a \leftarrow s \\ ?b \leftarrow p \\ ?y \leftarrow o}}\ (TT_{p_i}) \end{array} \right) \end{array} \right) - TT_{\{?b\}} \end{array} \tag{5.17}$$

Cet exemple nous permet d'observer comment la saturation se comporte dans le cas où la règle d'inférence ne précise pas la propriété sur laquelle elle porte. Dans la prochaine section, nous allons comparer la saturation sur les approches `TripleTable` et `VerticalPartitioning`.

5.5.2 Comparaison des approches TripleTable et VerticalPartitioning

Nous allons comparer l'implémentation *monetdbIE-tt* de l'algorithme 2 en utilisant l'approche `TripleTable` à l'implémentation *monetdbIE-vp* du même algorithme en utilisant l'approche du partitionnement vertical.

La Table 5.2 contient les résultats issus de la première itération de la saturation de 170 000 triplets par chacun des moteurs *monetdbIE-tt* et *monetdbIE-vp*. La table représente pour chacun des deux moteurs et pour chaque règle *RDFS*: le nombre de requêtes d'insertion produites, la durée d'exécution de ces requêtes et le nombre de triplets inférés.

Nous observons que l'approche de Partitionnement vertical est plus lente d'un ordre de grandeur en comparaison à l'approche `TripleTable`. L'intérêt du

	monetdbIE-vp		*monetdbIE-tt*		
	# INSERT	Durée (ms)	# INSERT	Durée (ms)	# triplets
rdfs2	1 162	16 636	1	190	5 195
rdfs3	1 162	12 982	1	279	10 055
rdfs5	**1**	**14**	**1**	**64**	15
rdfs7	703	32 099	1	514	123 323
rdfs9	1 893	31 155	1	49	121
rdfs11	**1**	**16**	**1**	**56**	570
TOTAL	4 922	92 902	6	1 152	139 279

Table 5.2: Résultats de la comparaison des approches TripleTable et vertical-Partitioning pour la saturation

partitionnement vertical apparaît uniquement pour les règles `rdfs5` et `rdfs11` où la règle accède à une seule table `rdfs:subPropertyOf` pour `rdfs5` et `rdfs:subClassOf` pour `rdfs11`.

Les conclusions des autres règles contiennent des motifs de triplets qui créent plusieurs requêtes d'insertion. Les caractéristiques de ces requêtes (telles que la table à insérer et les valeurs de sélection) changent d'une requête à l'autre. Ce qui a l'inconvénient de rendre inefficace les stratégies d'optimisation de requêtes dans *MonetDB*. En effet, les plans d'exécution issus de ces requêtes ne sont plus efficaces car les données d'une table sont éventuellement renvoyées sur le disque par l'exécution d'une autre requête qui ramène une autre table en mémoire. Ce phénomène ne se produit pas dans le cas de la saturation `TripleTable` car toutes les requêtes accèdent aux données de la seule `TripleTable`, elle peut donc être modifiée très longtemps en mémoire avant d'être renvoyée sur le disque.

Dans la prochaine section nous allons présenter les résultats de la saturation de plus grands volumes de traces.

5.5.3 Le passage à l'échelle de la saturation

Dans cette section nous représentons les temps de saturation du moteur d'inférence *monetdbIE-tt* sur les triplets initiaux issues des traces *T1, T2* et *T3* (nous avons utilisé les triplets saturées issues de ces traces dans la Section 4.2.1). Ces traces contiennent respectivement 10, 20 et 50 millions de triplets initaux. Nous utilisons les règles d'inférence *RDFS* (*rdfs2, rdfs3, rdfs5, rdfs7, rdfs9* et *rdfs11*), ainsi que les règles métier *R01* et *R02*. Nous considérons également la règle métier (*ordering*) pour l'inférence de l'ordre de précédence entre les nouvelles instances de fonctionnalités créées par les règles métier *R01* et *R02*.

La Figure 5.3 nous permet d'observer que les temps de saturation des règles évoluent de façon linéaire pendant les itérations de l'algorithme de saturation. De façon plus détaillée, la Table 5.3 présente les temps d'exécution des requêtes d'insertion issues de chaque règles d'inférence pendant la saturation de la trace *T3*. On observe que la règle *rdfs7* (pour la subsomption des propriétés) est la plus coûteuse en temps à cause du fait qu'il y beaucoup de sous propriétés dans

VIDECOM qui correspondent à des spécialisations des concepts métier.

règles	Temps de saturation
ordering	13 sec
R01	02 m 53 sec
R02	02 m 28 sec
rdfs11	01 sec
rdfs2	01 m 31 sec
rdfs3	01 m 03 sec
rdfs5	02 sec
rdfs7	**19 m 48 sec**
rdfs9	28 sec
	28 m 27 sec

Table 5.3: Temps de saturation de la trace T3 par *monetdbIE-tt*

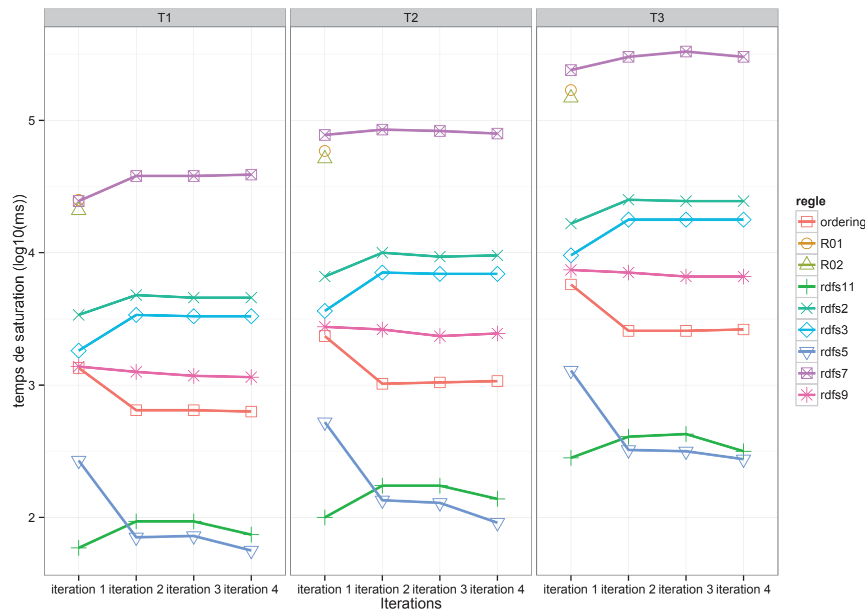


Figure 5.3: Temps de saturation des traces

Le tableau ci-dessous représente le nombre de triplets pour chacune des traces avant et après la saturation. On observe que la saturation double en moyenne le nombre de triplets de l'ontologie VIDECOM issues des événements de traces.

	avant la saturation	après la saturation
T1	10 000 000	20 060 904
T2	20 000 000	40 118 282
T3	50 000 000	100 341 226

Notre moteur *monetdbIE-tt* ne passe pas à l'échelle quand nous essayons de saturer 100 millions de triplets. L'explication est que la table `TripleTable` devient très grande en mémoire ce qui laisse peu d'espace aux résultat intermédaires des auto-jointures pendant la saturation.

5.6 Conclusion

L'objectif de ce chapitre était de présenter les techniques utilisées par les moteurs d'inférence pour la matérialisation des concepts métier à partir des règles métier.

Nous avons présenté les deux approches de raisonnement utilisées par les moteurs d'inférence, à savoir la réécriture de requêtes et la saturation. Il est apparu que la saturation est adaptée pour l'inférence des connaissances dans l'ontologie VIDECOM. En effet, contrairement à la réécriture de requêtes, la saturation minimise le temps de réponse des requêtes en matérialisant, à l'avance, toutes les connaissances que peuvent produire les règles d'inférence. Cette approche rend le moteur d'inférence réactif pendant l'interrogation de l'ontologie.

Nous avons ensuite identifié les défis que pose la saturation des règles métier aux moteurs d'inférence. A savoir, le problème de terminaison posé par l'inférence des nouvelles instances pendant la saturation et le problème du passage à l'échelle de l'inférence des relations temporelles entre les instances des évènements dans la trace. Nous avons proposé un modèle de saturation qui adresse ces deux limitations. Notre contribution consiste en une implémentation d'un moteur qui transforme les règles d'inférence en requêtes d'insertion SQL. Nos expérimentations nous montrent que cette approche de saturation est plus performante sur *MonetDB* que sur *PostGreSQL*. Ceci est dû aux coûts de mise à jour des index dans *PostGreSQL* pendant l'insertion des triplets inférés. Nos expérimentations montrent également que l'implémentation de la saturation en *SQL* ne passe pas à l'échelle lorsque des triplets sont stockés selon l'approche du partitionnement vertical. Cela étant dû au fait que le partitionnement vertical de la table `TripleTable` engendre une multitude de requêtes d'insertion *SQL* inefficaces à exécuter pendant la saturation

Nous pouvons ainsi conclure à la suite de ce chapitre et du précédent, que *MonetDB* est adapté autant pour le stockage et l'interrogation que pour la saturation des triplets issus de l'ontologie VIDECOM pour l'analyse de traces. Toutefois, le partitionnement vertical est adapté à l'interrogation des données, tandis que l'approche `TripleTable` passe à l'échelle pour la saturation. Ainsi, le meilleur compromis est de saturer les triplets en utilisant l'approche `TripleTable`, puis de partitionner la table pour gagner en performance à l'interrogation.

Dans le prochain chapitre nous allons présenter le projet industriel dans lequel nous avons travaillé, puis nous allons présenter l'application de VIDECOM dans le cadre de ce projet.

Chapitre 6

Intégration de VIDECOM dans le projet SoC-Trace

L'objectif de ce chapitre est de présenter le projet industriel dans lequel nous avons effectué cette thèse et de présenter les applications de VIDECOM dans le cadre de ce projet. La première partie du chapitre présente le pojet *Soc-Trace*. La seconde partie du chapitre présente sommairement l'infrastructure de traces *Framesoc*, qui est utilisée par tous les outils d'analyse de trace au sein du projet *Soc-Trace*. La troisième partie du chapitre présente un cas d'utilisation de VIDECOM pour l'analyse de traces réelles sur l'infrastructure *Framesoc*.

6.1 Contexte académique et industriel de la thèse

Nous avons fait cette thèse au sein d'un projet FUI nommé *SoC-Trace*. Le premier objectif du projet *SoC-Trace* est de développer un ensemble de méthodes et d'outils basés sur les traces d'exécution pour permettre une meilleur observabilité et débogabilité d'applications embarquées multicœur. Le deuxième objectif du projet *SoC-Trace* est de fournir un système d'exploitation de traces complet, capable de prendre en charge l'ensemble du flot de traces d'exécution depuis sa collecte jusqu'à son exploitation. Enfin, le troisième objectif du projet est de consolider un pôle d'expertise dans le domaine des traces et du débogage.

Le projet regroupe plusieurs partenaires académiques et industriels. Parmi lesquels on distingue *STMicroelectronics* [STM], *Magillem Design Services* [mag] et *Probayes* [Pro] pour les partenaires industriels ainsi que *INRIA Rhônes-Alpes* [RA] et l'*Université Joseph Fourier* [Fou] pour les partenaires académiques. Nous avons effectué nos travaux au sein de l'équipe *SLIDE* [SLI] du Laboratoire Informatique de Grenoble [dG] et nous avons travaillé avec *STMicroelectronics* .

Les résultats du projet sont intégrés sous la forme d'un produit unique appelé *SET* (Système d'Exploitation de Traces). Le SET est composé d'une infrastructure de gestion des traces appelée *Framesoc* et d'un ensemble d'outils indépendants mais complémentaires, travaillant au dessus du système de gestion de traces.

Les objectifs de notre thèse au sein du projet sont définit par les tâches suivantes:

1. **La définition d'une ontologie de référence pour les traces systèmes** dans le but de définir le socle sémantique pour abstraire les informations brutes contenues dans les traces.

2. **L'organisation en niveaux d'abstraction** pour faciliter l'interprétation de la trace et la navigation au sein de celle-ci en réduisant la trace aux informations utiles dans un contexte donné.

3. **L'interrogation des traces par niveaux d'abstraction** en se basant sur des objectifs définis par les utilisateurs pour lancer un ensemble de règles permettant de guider la recherche de certaines particularités dans les traces. Cette interrogation permet également aux autres outils de disposer d'un niveaux d'abstraction dédié.

Les outils intégrés à l'infrastructure *Framesoc* sont regroupés en deux catégories que sont les outils de visualisation et les outils d'analyse de traces. L'ontologie VIDECOM apparait comme un outil transversal car elle apporte un modèle des connaissances métiers des développeur qui peut être exploité par chacun des outils de deux catégories.

6.2 L'infrastructure *Framesoc*

Framesoc est une infrastructure de gestion de traces disponible sous la forme d'un environnement *Eclipse* dans lequel d'autres outils sont installés sous la forme de *plugins* Eclipse [Pag].

Des plugins Eclipse sont disponibles sur *Framesoc* pour l'importation des traces sous différents formats tels que *CFT*, *OTF2*, *kptrace*, *Paje* et *GStreamer*. Ces importateurs extraient les évènements des traces et les stockent dans un modèle relationnel unifié proposé par *Framesoc* [PMM14]. Grâce à ce modèle unifié les différents outils de visualisation (tels que *kptrace viewer* [PRRR+09] et *Ocelotl* [DHV+13]) et d'analyse de traces (tels que *Frameminer* [KKFT+13] et *Megalog*) peuvent être conjointement utilisés pour l'analyse d'une même trace.

Certains de ces outils stockent leurs résultats d'analyse dans ce modèle relationnel unifié, permettant ainsi aux autres outils de les exploiter. C'est ainsi par exemple, que les blocs résultats de *Frameminer* peuvent être exploités par

Megalog.

La Figure 6.1 présente l'architecture de *Framesoc*. On distingue les différents plugins pour les importateurs de traces ainsi que les outils de visualisation et d'analyse de traces. *Framesoc* utilise une base de données *SQLite* ou *MySQL* pour gérer son modèle relationnel unifié.

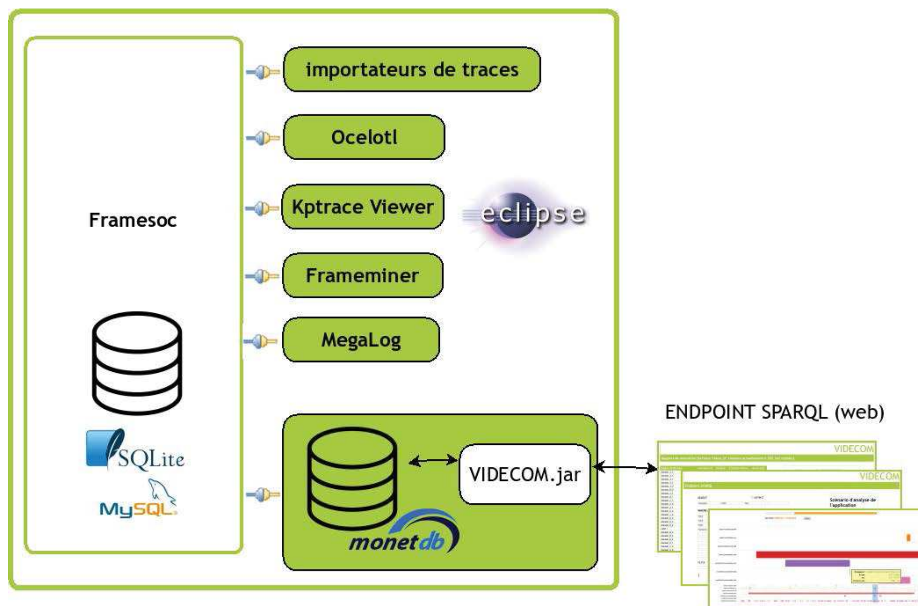


Figure 6.1: Architecture de l'infrastructure de trace (Framesoc) et les plugins

Dans la Figure 6.1 nous pouvons également identifier le plugin Eclipse pour l'ontologie VIDECOM. Le plugin utilise le column-store *MonetDB* pour stocker les triplets issus des évènements de la trace chargée depuis la *Framesoc*. Le plugin VIDECOM permet de faire la saturation des triplets de la trace à tout moment en précisant les règles métier dans un fichier. Les résultats des requêtes *SPARQL* exécutées sur le plugin peuvent être visualisés à partir d'une interface web. Ces résultats peuvent également être stockés dans le modèle relationnel de *Framesoc*.

C'est grâce à cette possibilité de stocker, dans le modèle relationel unifié de *Framesoc*, les résultats des requêtes déclaratives exécutées sur VIDECOM, que les autres outils d'analyse et de visualisation de traces, installés sur *Framesoc*, bénéficient des connaissances métier modélisées dans l'ontologie.

6.3 Cas d'utilisation de VIDECOM dans le SET

Pour illustrer l'utilisation du plugin VIDECOM dans le *Framesoc*, nous allons considérer le cas réel de l'analyse de l'application `ts_record`. `ts_record` est une application embarqué dans les set-top box qui permet d'enregistrer sur un disque USB, un flux vidéo (par exemple une émission) en provenance d'internet.

L'exécution de l'application `ts_record` doit respecter un ensemble de contraintes temporelles (Figure 6.2). Ainsi, la séquence d'interruptions `gic_eth` et `net_rx_action` se produit durant l'exécution pour lire les données depuis le flux internet et les copier dans des buffers IP de faible capacité mémoire (230 KB). Ensuite, toutes les 100 millisecondes, la tâche `ts_record` exécute les appels système `sys_read` et `sys_write` pour respectivement copier des données des buffers IP et les écrire dans la mémoire centrale. Enfin, toutes les 5 secondes, la tâche `flush-8.0` s'exécute pour copier les données de la mémoire vers le disque USB.

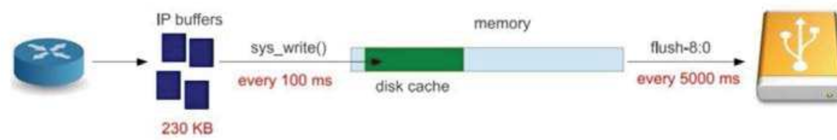


Figure 6.2: Schéma de fonctionnement de l'application ts_record

Si l'utilisateur rencontre des problèmes pendant la lecture du fichier résultat enregistré par `ts_record` (tels que les images détériorées ou la désynchonisation du son et de la vidéo), le développeur peut faire l'hypothèse que les données multimédia copiées par l'application sont corrompues. Il utilise les concepts métier de VIDECOM pour créer des niveaux d'abstraction relatifs aux contraintes temporelles citées précédemment afin d'identifier les zones de la trace où ces contraintes sont vérifiées ou non.

Dans ce cas d'utilisation nous avons utilisé les règles d'inférence métier pour saturer les traces issues de cette application. Le tableau ci-dessous résume les nouveaux concepts métier dont les instances ont été inférées dans VIDECOM pour matérialiser les nouveaux niveaux d'abstraction.

<table>
<tr><td>Concepts métier</td><td>classe principale VIDECOM</td></tr>
<tr><td><code>dataFromEthernet2IP</code></td><td><code>Functionality</code></td></tr>
<tr><td colspan="2">apparition de la séquence d'écriture des données du flux internet vers les buffers IP</td></tr>
<tr><td><code>dataFromIP2Memory</code></td><td><code>Functionality</code></td></tr>
<tr><td colspan="2">apparition de la séquence d'écriture des données des buffers IP vers la mémoire</td></tr>
</table>

sysWriteConstraintNormal	Functionality
respect de la contrainte de 100 ms (appel de sys_write après plus de 99 et avant moins de 110 ms)	

sysWriteConstraintBlocked	Anomaly
non respect de la contrainte de 100 ms (appel de sys_write après plus de 120 ms)	

sysWriteConstraintFailed	Anomaly
non respect de la contrainte de 100 ms (appel de sys_write avant moins de 80 ms)	

flushConstraintNormal	Functionality
respect de la contrainte de 5 sec (exécution de la flush-8.0 après plus de 4 sec et avant moins de 6 sec)	

flushConstraintFailed	Anomaly
non respect de la contrainte de 5 sec (exécution de la flush-8.0 avant moins de 4 sec)	

flushConstraintBlocked	Anomaly
non respect de la contrainte de 5 sec (exécution de la flush-8.0 après plus de 6 sec)	

dataFromMemory2USB	Functionality
apparition de la séquence d'écriture des données de la mémoire vers le disque USB	

Pour conlure ce chapitre, nous allons présenter une démonstration de l'utilisation de VIDECOM sur l'infrastructure *Framesoc* pour l'analyse de l'application `ts_record`.

6.4 Démonstration de l'utilisation de VIDECOM sur *Framesoc*

1. L'utilisation de VIDECOM commence par le chargement de tout ou partie de la trace à analyser. Cette première étape consiste à tranformer chaque évènement de la trace du modèle relationnel de *Framesoc* vers le modèle RDF de l'ontologie. Chaque évènement produit un ensemble de triplets pour représenter les informations basiques qu'il contient. Ces triplets sont directement stockés dans une base de données créée sur *MonetDB*. La Figure 6.3 représente l'interface du plugin VIDECOM qui permet de charger une trace depuis l'infrastructure *Framesoc*.

2. L'étape suivante est la saturation. Grâce à l'interface web (Figure 6.4), le développeur fournit un fichier qui contient ses règles métier dans une forme proche du *Datalog*. Les règles RDFS (`rdfs2`, `rdfs3`, `rdfs5`, `rdfs7`, `rdfs9` et `rdfs11`) sont par défaut prises en compte au moment de la saturation.

L'interface permet de lancer la saturation et de consulter les résultats (temps de saturation et nombre de triplets inférés par règle) des éventuelles saturations déjà exécutées sur cette trace.

3. Une fois l'ontologie saturée, le développeur peut ensuite construire puis exécuter des requêtes déclaratives en utilisant une autre interface web du plugin VIDECOM (Figure 6.5). Cette interface web offre un formulaire pré-rempli avec les noms des concepts de VIDECOM pour faciliter la construction de requêtes SPARQL. L'interface propose également des modèles de requêtes dont le développeur peut s'inspirer. Les résultats des requêtes sont présentés dans la même interface.

4. Dans le cas des requêtes de recherche des concepts métier dans la trace, l'interface permet de visualiser les résultats sous la forme d'un diagramme de Gantt où chaque ligne représente l'activité du concept métier dans la trace. Par exemple, la Figure 6.6 représente le diagramme Gantt qui correspond à toutes les anomalies et les fonctionnalités contenues dans la trace. On peut ainsi distinguer, visuellement, les zones de traces où les contraintes temporelles de l'application `ts_record` n'ont pas été vérifiées dans l'intervalle de temps [8066120 μs, 31252930 μs]. Cette interface permet de confronter plusieurs niveaux d'abstraction et d'observer, par exemple, que les instances de l'anomalie `sysWriteConstraintBlocked` se produisent pendant l'activité des instances de la fonctionnalité `flushConstraintNormal`.

   Il s'est avéré par la suite qu'une mal-fonction de l'application `ts_record` était dûe au fait il arrivait que certaines pages mémoire accédées en lecture par la tâche `flush` soient ensuite sollicitées en écriture par l'interruption `sys_write` de la tâche `ts_record`. Ces interruptions étaient donc mis en attente (ce qui entrainait l'inférence des instances de l'anomalie `sysWriteConstraintBlocked`). Ces anomalies entrainaient à leur tour des retards dans le transfert des données stockées dans les buffers IP vers la mémoire vide. En conséquence, certaines de ces données étaient écrasées par de nouvelles données venant d'Internet. La conséquence de ces données écrasées était des images détériorées à la lecture du fichier résultat.

5. Comme nous l'avons dit précédemment, les résultats des requêtes sur l'ontologie peuvent être stockés dans la base de données Framesoc. Dans la Figure 6.7, on peut distinguer l'interface de visualisation des concepts métiers (camemberg) dans *Framesoc*. La figure représente également les agrégations temporelles construites par le plugin Ocelotl en utilisant les résultats de la requête de recherche des anomalies et des fonctionnalités dans la trace.

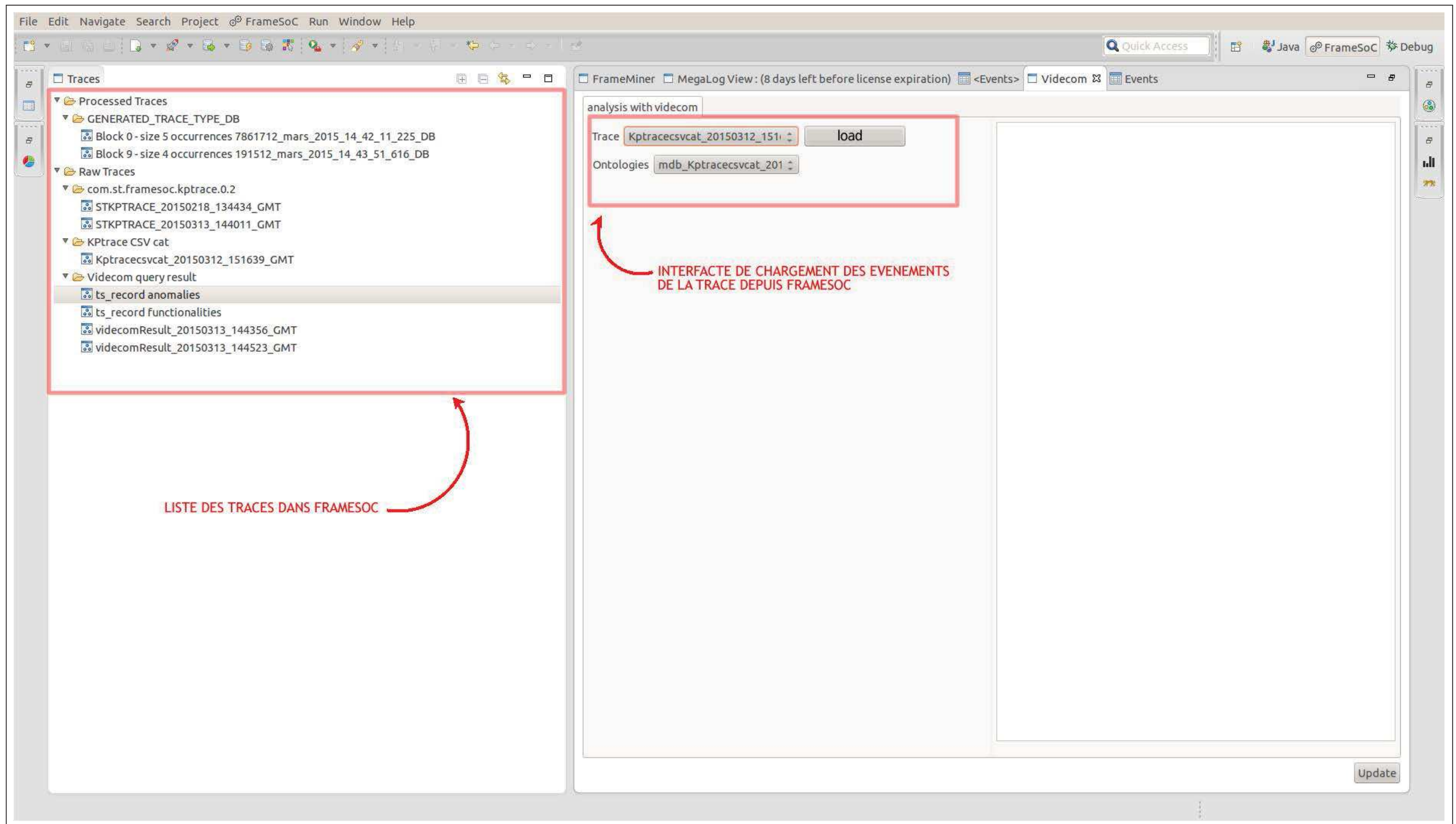


Figure 6.3: Interface du Plugin VIDECOM pour le chargement des évènenemts de la trace depuis Framesoc

VIDECOM

Rapport de saturation [la trace "these_tt" contient actuellement 6 505 342 triplets]

Règles d'inférence	saturation id	itération	# triplets inféres	durée (ms)
domain_3_3	-636363028	1	139	549
domain_3_4	-636363028	1	139	520
domain_3_1	-636363028	1	139	540
domain_3_2	-636363028	1	139	537
domain_3_0	-636363028	1	139	115
domain_8_0	-636363028	1	0	66
domain_2_2	-636363028	1	1360	505
domain_8_1	-636363028	1	0	481
domain_2_3	-636363028	1	1360	512
domain_2_0	-636363028	1	1360	65
domain_2_1	-636363028	1	1360	545
domain_2_4	-636363028	1	1360	528
domain_1_0	-636363028	1	361	98
domain_0_4	-636363028	1	29263	560
domain_0_3	-636363028	1	29263	610
domain_0_2	-636363028	1	29263	575

Les règles d'inférence **rdfs2, rdfs3, rdfs5, rdfs7, rdfs9** et **rdfs11** sont appliquées par défaut. Veuiller attacher le fichier des règles métier

Choisissez un fichier Aucun fichier choisi

Lancer la saturation (*)

Figure 6.4: Interface web de saturation des traces dans VIDECOM

Figure 6.5: Interface web d'interrogation des traces dans VIDECOM

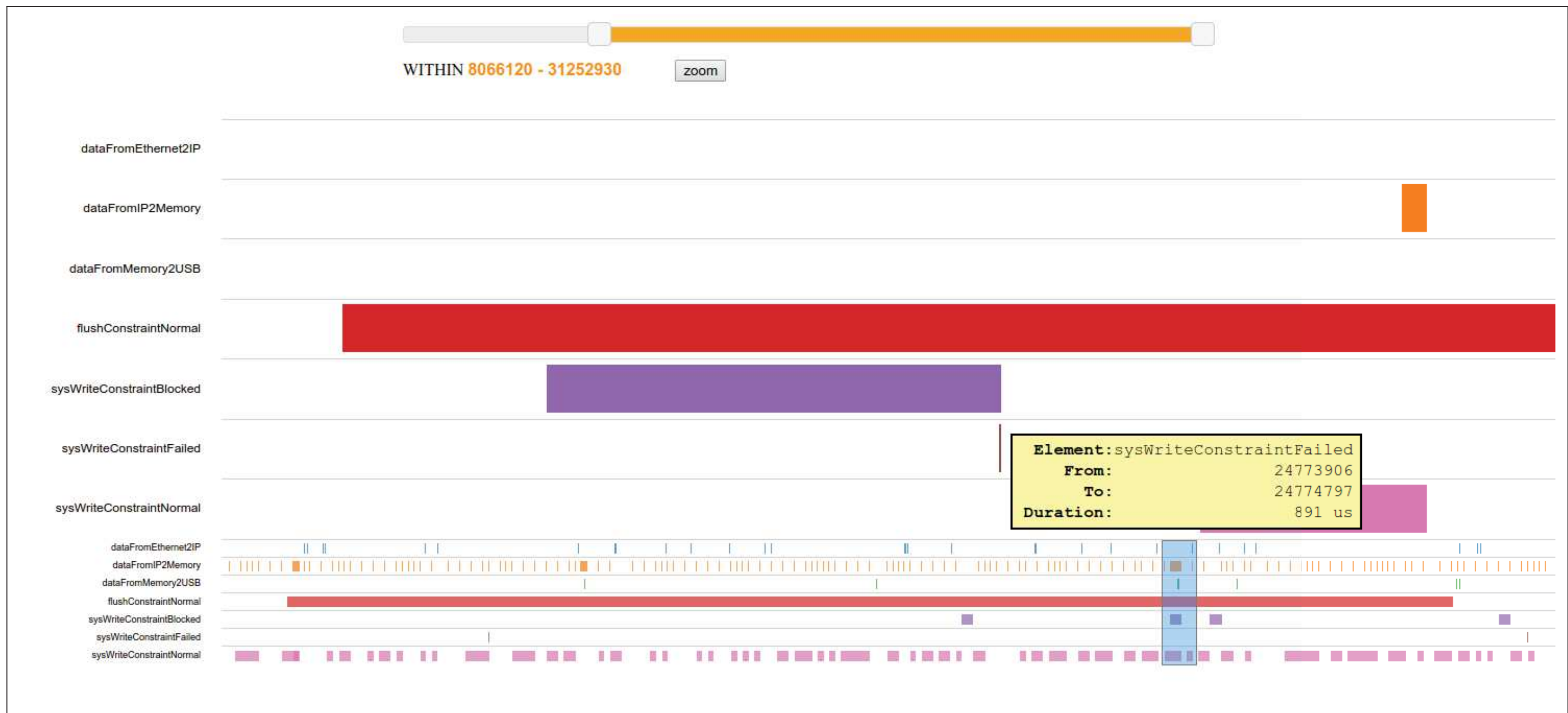


Figure 6.6: Visualisation des concepts métier (anomalies et fonctionnalité de ts_record) dans VIDECOM

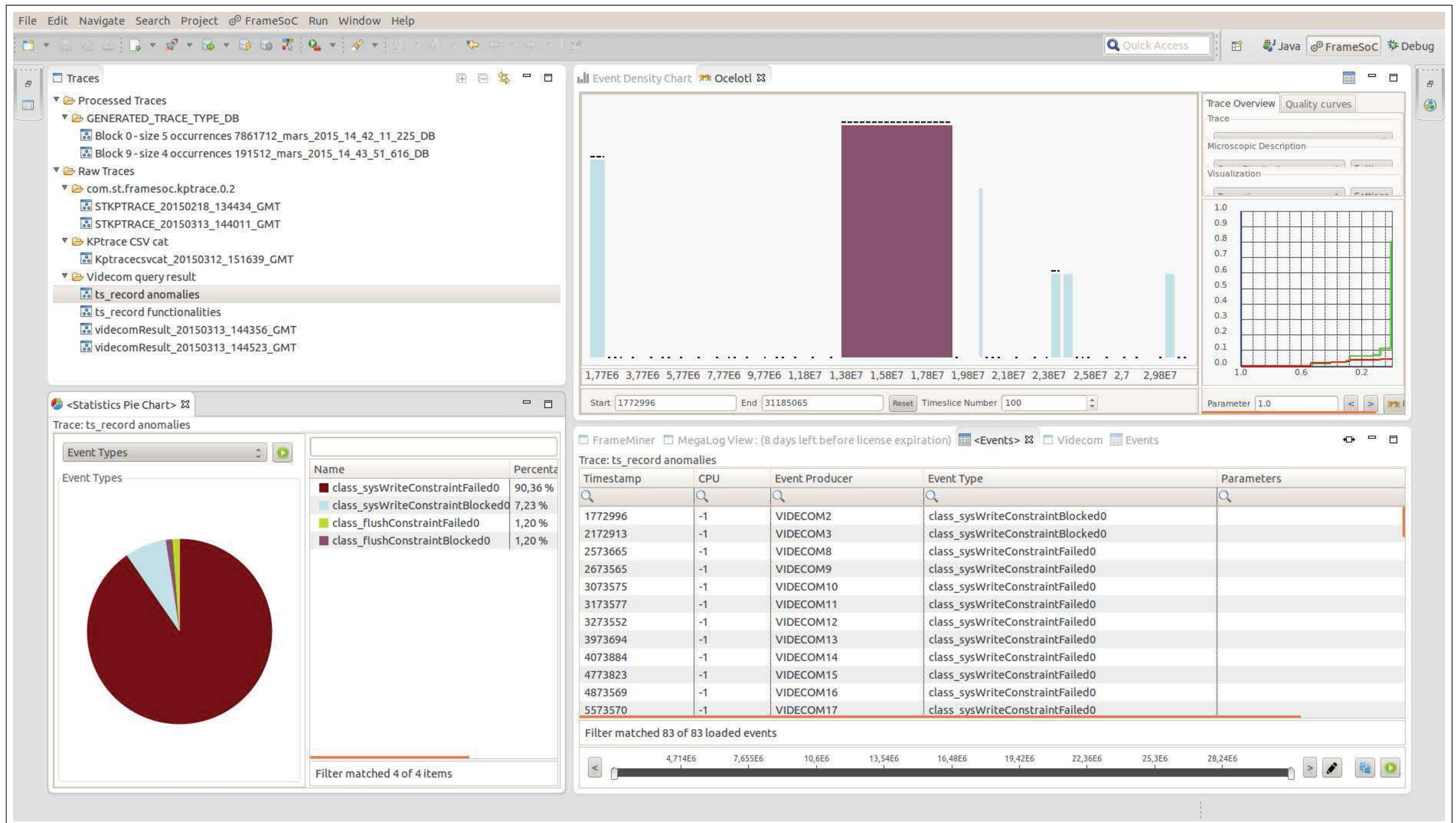


Figure 6.7: Visualisation des concepts métier (anomalies et fonctionnalités de ts_record) dans Framesoc et le plugin Ocelotl

Chapitre 7

Conclusion

Dans cette thèse, nous nous sommes intéressés au domaine de l'analyse des traces d'exécution issues des systèmes embarqués multiprocesseur ou MPSoC. Ces traces contiennent des informations à partir desquelles le développeur extrait des connaissances pour comprendre, améliorer et prédire le fonctionnement des applications embarquées.

Ayant constaté, dans le Chapitre 2, que les outils d'analyse de traces permettent difficilement au développeur d'exploiter ses propres connaissances métier ainsi que celles contenues dans les traces, nous avons proposé une approche qui permet au développeur de représenter, manipuler et interroger ces traces en se basant sur des concepts liés à ses propres connaissances métier.

Nous proposons une ontologie pour modéliser et interroger les concepts et les connaissances métier du développeur et un moteur d'inférence pour raisonner sur ces connaissances. Nous pensons que cette ontologie peut être considéré comme l'ontologie générale du domaine de l'analyse des traces d'exécution et servir de base à la construction d'autres ontologies spécialisées à des cas spécifiques d'analyse de traces.

7.1 La contribution

A cause de la grande diversité et hétérogénéité des traces, le domaine de l'analyse de traces ne dispose pas d'une ontologie générale. Ainsi, dans le Chapitre 3 nous avons proposé VIDECOM pour modéliser les concepts et les connaissances métiers du domaine de l'analyse des traces. Nous avons construit cette ontologie en identifiant les différentes catégories d'informations contenues dans les traces d'applications embarquées multimédia, à savoir les informations signalétiques, temporelles, spatiales et sémantiques. VIDECOM utilise des régles d'inférence pour représenter les connaissances du développeur devant servir à raisonner sur les concepts métier. De plus, les concepts de l'ontologie peuvent être étendus, manuellement ou à l'aide des règles d'inférence, pour représenter de nouveaux

concepts métier.

Ensuite, nous nous sommes intéressé à l'exploitation de VIDECOM pour l'analyse des traces de grande tailles. Dans le but de déterminer les caractéristiques permettant de stocker et d'interroger efficacement l'ontologie VIDECOM instanciée à partir des évènements d'une trace, nous avons comparé les performances de 12 triplestores dans le Chapitre 4. En nous basant sur un benchmark de 17 requêtes d'analyse de traces, nous avons observé que le partitionnement vertical est la stratégie de stockage de l'ontologie qui offre le meilleure compromis sur les performances de chargement et d'interrogation de VIDECOM pour l'analyse de traces. Nous avons également mis en évidence le fait que la stratégie du partitionnement vertical est plus efficace sur le *colomn-store MonetDB* que sur le *row-store PostGreSQL*. Nous avons toutefois observé que le passage à l'echelle du partitionnement vertical était limité par les performances des auto-jointures dans *MonetDB*.

Face à l'efficacité du chargement et de l'interrogation de VIDECOM stockée suivant le partitionnement vertical dans *MonetDB*, nous nous sommes intéressés aux performances de l'inférence des concepts métier. Dans le chapitre 5, nous avons proposé un moteur d'inférence basé sur *MonetDB* qui utilise la méthode de raisonnement des connaissances par la *saturation*. Le moteur d'inférence réécrit les règles d'inférence de l'ontologie en des requêtes *SQL*. Contrairement aux moteurs d'inférence des autres triplestores, notre moteur infère efficacement des informations temporelles entre les événements et termine lors de l'inférence des concepts métier issues des règles d'inférence qui traduisent les connaissances métier du développeur. Nous avons cependant observé que la saturation sur le partionnement vertical est moins efficace que la saturation sur l'ontologie stockée dans une seule table MonetDB. Nous avons également mis en évidence le fait que cette approche de saturation est inefficace lorsqu'elle est implémentée sur le *row-store PostGreSQL* à cause du coût des mises à jour des index.

Enfin, dans le Chapitre 6 nous avons illustré la mise en pratique de VIDECOM, dans le cadre du projet *SoC-Trace*. Nous avons présenté l'intégration de VIDECOM dans l'infrastructure de gestion de traces *Framesoc*, où elle est utilisée dans des contextes réels d'analyse de traces d'applications embarquées multimédia sur MPSoC.

7.2 Les perspectives

Cette thèse ouvre des perspectives quant à l'utilisation des ontologies dans des domaines d'application nouveaux, et ouvre des perspectives intéressantes sur les problèmes de passage à l'échelle d'une part et d'enrichissement semi-automatique d'autre part.

7.2.1 Le passage à l'échelle

Les triplestores distribués

Nous nous sommes limités dans cette thèse au cas des triplestore monolythiques. Les triplestores distribués/parallèle tels que *cliqueSquare* [GKM+13], *Rya* [PCR12]

ou encore *Jena-HBase* [KKTC12], utilisent des techniques *NoSQL* pour interroger efficacement une ontologie dont les triplets stocké sur une architecture distribuée. Nous pensons que ces triplestores permettent le passage à l'échelle l'analyse des traces de plus grande taille. Toutefois ces triplestores devront implémenter la saturation des connaissances métier sur leurs architectures distribuées/parallèles.

Échantillonnage des traces

La taille de la base de connaissances est un facteur de performance important pour l'interrogation et la saturation de VIDECOM. En conséquence, la réduction de cette taille impacte significativement ces performances. Nous pensons que l'échantillonnage des traces est une piste prometteuse pour la réduction de taille des données.

Toutefois, cet échantillonnage doit traiter la problématique de la dépendance des connaissances. En effet, nous avons mené des expérimentations préliminaires sur l'échantillonnage de traces à l'aide du *random Sampling*, de *systematic sampling* et du *stratified sampling* [SM96]. Ces expérimentations nous ont permis d'observer que l'incomplétude des faits de la base de connaissances entraînait l'inférence de connaissances erronées et/ou l'absence d'inférence par certaines règles métier. Ces phénomènes sont plus marqués dans le cas du *random sampling* où les évènements de l'échantillon sont aléatoirement choisis dans la trace. Ces phénomènes sont moins marqués dans le cas du *stratified sampling* où les évènements sont préalablement catégorisés avant d'être aléatoirement choisis dans chaque catégorie.

Il n'en demeure pas moins que l'absence d'inférence et l'inférence de connaissances erronées sont critiques pour l'analyse des traces car ils perturbent l'analyse du problème cible. Pour éviter ces phénomènes nous pensons que l'échantillonnage des données doit privilégier la qualité des connaissances inférées par certaines règles métier en garantissant la présence de tous les faits nécessaires à leur application.

7.2.2 L'enrichissement semi-automatique

L'enrichissement de l'ontologie à l'aide de la fouille de données

Les motifs (fréquents, représentatifs, périodiques ou discriminatifs) qui sont retournés par les algorithmes de fouille de données correspondent à de potentielles connaissances (ou niveaux d'abstraction) pertinentes. Nous pensons que ces motifs peuvent servir à l'enrichissement semi-automatique des concepts métiers dans VIDECOM.

Dans le cas où les résultats de l'algorithme de fouille de données ne sont pas nombreux (comme par exemple *Frameminer* où le nombre de motifs résultats est un paramètre d'entrée de l'algorithme [KKFT+13]), les motifs peuvent être annotés par le développeur puis rattachés comme de nouveaux concepts métier aux autres concepts métier dans VIDECOM. Nous pensons que Frameminer est adapté à cette approche d'enrichissement semi-automatique de VIDECOM car les motifs qu'il retourne sont peu nombreux et ont une couverture maximale de la trace ce qui démontre leur pertinance.

Dans le cas où les résultats de l'algorithme de fouille de données sont nombreux, l'ontologie peut aider à les catégoriser en fonction des concepts métier.

Nous avons mené des expérimentations préliminaires qui consistait à considérer les motifs séquentiels représentatifs retournés par *Frameminer* comme des traces. Grâce à cette approche certains concepts métier de l'ontologie étaient instanciés et certaines règles d'inférence pouvaient s'appliquer. Cette approche permet de rattacher automatiquement un motif à un ou plusieurs concepts métier et/ou niveaux d'abstraction. Cela peut améliorer l'interprétation du motif ou/et aider à formuler de nouvelles règles métier par exemple en exploitant les règles d'association identifiées dans le motif.

Liste des publications

Conférence internationale

1 Christiane Kamdem Kengne, **Fopa Leon Constantin**, Alexandre Termier, Noha Ibrahim, Marie-Christine Rousset, Takashi Washio, Miguel Santana. *Efficiently rewriting large multimedia application execution traces with few event sequences.* **KDD 2013**, pp. 1348-1356

Workshops internationaux

2 **Fopa Leon Constantin**, Fabrice Jouanot, Alexandre Termier, Maurice Tchuente and Oleg Iegorov. *Benchmarking of triple stores scalability for MPSoC trace analysis.* VLDB workshop on benchmarking RDF systems (**Bersys**), 2014

3 Christiane Kamdem Kengne, **Fopa Leon Constantin** , Noha Ibrahim, Alexandre Termier, Marie-Christine Rousset, Takashi Washio *Enhancing the Analysis of Large Multimedia Applications Execution Traces with FrameMiner.* ICDM workshop on Practical Theories for Exploratory Data Mining (**PTDM**), 2012, pp. 595-602

Conférence nationale

4 **Fopa Leon Constantin**, Jouanot Fabrice, Termier Alexandre and Tchuente Maurice. *Le Web sémantique en aide à l'analyste de traces d'exécution.* **BDA'14**-Journées de Bases de Données Avancées

Appendix A

Appendice

A.1 Liste des réquêtes d'insertion pour les règles d'inférence RDFS

RDFS2

$?a\ rdfs{:}domain\ ?x\ .\ \rightarrow\ ?y\ rdf{:}type\ ?x$
$?y\ ?a\ ?z$

```
INSERT INTO TripleTable (s,p,o)
SELECT P2.s AS s, 'rdf:type' AS p, P1.o AS o
FROM TripleTable P1,
     TripleTable P2
WHERE P1.p = rdfs:domain
  AND P1.s = P2.p
EXCEPT
SELECT s, p, o FROM TripleTable
```

RDFS3

$?a\ rdfs{:}range\ ?x\ .\ \rightarrow\ ?z\ rdf{:}type\ ?x$
$?y\ ?a\ ?z$

```
INSERT INTO TripleTable (s,p,o)
SELECT P2.o AS s, 'rdf:type' AS p, P1.o AS o
FROM TripleTable P1,
     TripleTable P2
WHERE P1.p = 'rdfs:range'
  AND P1.s = P2.p
EXCEPT
SELECT s, p, o FROM TripleTable
```

RDF5

$?x\ rdfs{:}subPropertyOf\ ?y\ .\quad\rightarrow\quad ?x\ rdfs{:}subPropertyOf\ ?z$
$?y\ rdfs{:}subPropertyOf\ ?z$

```
INSERT INTO TripleTable (s,p,o)
SELECT P1.s AS s, 'rdfs:subPropertyOf' AS p, P2.o AS o
FROM TripleTable P1,
     TripleTable P2
WHERE P1.p = 'rdfs:subPropertyOf'
  AND P2.p = 'rdfs:subPropertyOf'
  AND P1.o = P2.s
EXCEPT
SELECT s, p, o FROM TripleTable
WHERE p = 'rdfs:subPropertyOf'
```

RDFS7

$?a\ rdfs{:}subPropertyOf\ ?b\ .\quad\rightarrow\quad ?x\ ?b\ ?y$
$?x\ ?a\ ?y$

```
INSERT INTO TripleTable(s,p,o)
SELECT P2.s AS s, P1.o AS p, P2.o AS o
FROM TripleTable P1,
     TripleTable P2
WHERE P1.p = 'rdfs:subPropertyOf'
  AND P1.s = P2.p
EXCEPT
SELECT s, p, o FROM TripleTable
```

RDFS9

$?x\ rdfs{:}subClassOf\ ?y\ .\quad\rightarrow\quad ?z\ rdf{:}type\ ?y$
$?z\ rdf{:}type\ ?x$

```
INSERT INTO TripleTable (s,p,o)
SELECT P2.s AS s, 'rdf:type' AS p, P1.o AS o
FROM TripleTable P1,
     TripleTable P2
WHERE P1.p = 'rdfs:subClassOf'
  AND P2.p = 'rdf:type'
  AND P1.s = P2.o
EXCEPT
SELECT s, p, o FROM TripleTable
```

rdfs11

$?x\ rdfs{:}subClassOf\ ?y\ .\quad\rightarrow\quad ?x\ rdfs : subClassOf\ ?z$
$?y\ rdfs{:}subClassOf\ ?z$

```
INSERT INTO TripleTable (s,p,o)
SELECT P1.s AS s, 'rdfs:subClassOf' AS p, P2.o AS o
FROM TripleTable P1,
     TripleTable P2
WHERE P1.p = 'rdfs:subClassOf'
  AND P2.p = 'rdfs:subClassOf'
  AND P1.o = P2.s
EXCEPT
SELECT s, p, o FROM TripleTable
WHERE p = 'rdfs:subClassOf'
```

A.2 Requêtes SQL équivalentes pour le partitionnement vertical

star.q1

```
SELECT t1.s AS event, t1.o AS startAt, t2.o AS endAt,
       t3.o AS cpu, t4.o AS duration
FROM "prop_eventStartAt" t1,
    "prop_eventEndAt" t2,
    "prop_eventIsExecutedOnCPU" t3,
    "prop_eventHasDuration" t4
WHERE t1.s = t2.s
  AND t1.s = t3.s
  AND t1.s = t4.s
```

star.q2

```
SELECT t1.s AS event, t1.o AS startAt, t2.o AS endAt, t3.o AS cpu,
       t4.o AS duration, t5.o AS nextcpu, t6.o AS occurrence,
       t7.o AS component, t8.o AS trace, t9.o AS task
FROM "prop_eventStartAt" t1,
    "prop_eventEndAt" t2,
    "prop_eventIsExecutedOnCPU" t3,
    "prop_eventHasDuration" t4,
    "prop_eventPrecedeInCPU" t5,
    "prop_eventPrecedeOccurrence" t6,
    "prop_eventPrecedeInComponent" t7,
    "prop_eventPrecedeInTrace" t8,
    "prop_eventPrecedeInTask" t9
WHERE t1.s = t2.s
  AND t1.s = t3.s
  AND t1.s = t4.s
  AND t1.s = t5.s
  AND t1.s = t6.s
  AND t1.s = t7.s
  AND t1.s = t8.s
  AND t1.s = t9.s
```

star.q3

```
SELECT t1.s AS event, t1.o AS startAt, t2.o AS endAt, t3.o AS cpu,
       t4.o AS duration, t5.o AS nextcpu, t6.o AS occurrence,
       t7.o AS component, t8.o AS trace, t9.o AS task
FROM "prop_requestComponent" t0,
    "prop_eventStartAt" t1,
    "prop_eventEndAt" t2,
    "prop_eventIsExecutedOnCPU" t3,
    "prop_eventHasDuration" t4,
    "prop_eventPrecedeInCPU" t5,
    "prop_eventPrecedeOccurrence" t6,
    "prop_eventPrecedeInComponent" t7,
    "prop_eventPrecedeInTrace" t8,
    "prop_eventPrecedeInTask" t9
WHERE t0.o = $gic_tango_tp_mbx0$
  AND t0.s = t1.s
  AND t1.s = t2.s
  AND t1.s = t3.s
  AND t1.s = t4.s
  AND t1.s = t5.s
  AND t1.s = t6.s
  AND t1.s = t7.s
  AND t1.s = t8.s
  AND t1.s = t9.s
```

path.q1

```
SELECT t1.s AS event1, t2.o AS event2, t3.o AS event3
FROM "prop_SystemCallIsExecutedDuringTask" t1,
    "prop_eventPrecedeInTask" t2,
    "prop_eventPrecedeInTask" t3,
WHERE t1.o = $ts_record0$
  AND t1.s = t2.s
  AND t2.o = t3.s
```

path.q2

```
SELECT t1.s AS event1, t2.o AS event2, t3.o AS event3,
       t4.o AS event4, t5.o AS event5, t6.o AS event6
FROM "prop_SystemCallIsExecutedDuringTask" t1,
    "prop_eventPrecedeInTask" t2,
    "prop_eventPrecedeInTask" t3,
    "prop_eventPrecedeInTask" t4,
    "prop_eventPrecedeInTask" t5,
    "prop_eventPrecedeInTask" t6
WHERE t1.o = $ts_record0$
  AND t1.s = t2.s
  AND t2.o = t3.s
  AND t3.o = t4.s
  AND t4.o = t5.s
  AND t5.o = t6.s
```

path.q3

```
SELECT t1.s AS event1, t2.o AS event2, t3.o AS event3,
       t4.o AS event4, t5.o AS event5, t6.o AS event6
FROM "prop_SystemCallIsExecutedDuringTask" t1,
    "prop_eventPrecedeInTask" t2,
    "prop_eventPrecedeInTask" t3,
    "prop_eventPrecedeInTask" t4,
    "prop_eventPrecedeInTask" t5,
    "prop_eventPrecedeInTask" t6,
    "prop_SystemCallIsExecutedDuringTask" t7
WHERE t1.o = $ts_record0$
  AND t1.s = t2.s
  AND t2.o = t3.s
  AND t3.o = t4.s
  AND t4.o = t5.s
  AND t5.o = t6.s
  AND t6.s = t7.s
  AND t1.o = $ts_record0$
```

filter.q1

```
SELECT t1.s AS event, t1.o AS startAt, t2.o AS endAt,
       t3.o AS cpu, t4.o AS duration
FROM "prop_eventStartAt" t1,
    "prop_eventEndAt" t2,
    "prop_eventIsExecutedOnCPU" t3,
    "prop_eventHasDuration" t4
WHERE t1.s = t2.s
  AND t1.s = t3.s
  AND t1.s = t4.s
  AND t1.o >= 78322418
  AND t2.o <= 156644836
```

filter.q2

```
SELECT t1.s AS event, t5.o AS task
FROM "prop_eventIsRelatedToAnomaly" t0,
    "prop_eventStartAt" t1,
    "prop_eventEndAt" t2,
    "prop_eventStartAt" t3,
    "prop_eventEndAt" t4,
    "prop_runningTask" t5
WHERE t0.s = t1.s
  AND t1.s = t2.s
  AND t3.s = t4.s
  AND t4.s = t5.s
  AND t1.o >= 78322418
  AND t2.o <= 156644836
  AND t3.o >= t1.o
  AND t4.o <= t2.o
```

sort.q1

```
SELECT t1.o AS task, t1.s AS event, t2.o AS startAt
FROM "prop_runningTask" t1,
    "prop_eventStartAt" t2
WHERE t1.s = t2.s
```

sort.q2

```
SELECT t1.o AS task, t1.s AS event, t2.o AS startAt
FROM "prop_runningTask" t1,
    "prop_eventStartAt" t2
WHERE t1.s = t2.s
ORDER BY startAt
```

sort.q3

```
SELECT t1.o AS task, t1.s AS event, t2.o AS startAt
FROM "prop_runningTask" t1,
    "prop_eventStartAt" t2
WHERE t1.s = t2.s
LIMIT 100
```

sort.q4

```
SELECT DISTINCT t1.o AS task
FROM "prop_runningTask" t1,
    "prop_eventStartAt" t2
WHERE t1.s = t2.s
```

sort.q5

```
SELECT t1.o AS task, COUNT(t1.s) AS number
FROM "prop_runningTask" t1,
    "prop_eventStartAt" t2
WHERE t1.s = t2.s
GROUP BY task
```

```
model.q1
SELECT t2.s AS anomaly, $p̂roperties$ AS prop, t2.o As object
FROM "prop_eventIsRelatedToAnomaly" t1,
    "$p̂roperties$" t2
WHERE t1.o = t2.s
```

```
model.q2
SELECT t1.o AS anomaly, $p̂roperties$  AS prop, t2.s AS subject
FROM "prop_eventIsRelatedToAnomaly" t1,
    "$p̂roperties$" t2
WHERE t1.o = t2.o
```

```
compute.q1
SELECT t1.o AS component, SUM(t2.o) AS workload
FROM "prop_eventIsExecutedOn" t1,
    "prop_eventHasDuration" t2
WHERE t1.s = t2.s
GROUP BY component
```

```
compute.q2
SELECT t1.o AS interruption, AVG(t2.o) AS workload
FROM "prop_runningInterruption" t1,
    "prop_eventHasDuration" t2
WHERE t1.s = t2.s
GROUP BY interruption
```

```
compute.q2
SELECT t1.o AS interruption, AVG(t2.o) AS workload
FROM "prop_runningInterruption" t1,
    "prop_eventHasDuration" t2
WHERE t1.s = t2.s
GROUP BY interruption
HAVING (AVG(t2.o) > 10)
```

Bibliography

[ABB+00] Michael Ashburner, Catherine A Ball, Judith A Blake, David Botstein, Heather Butler, J Michael Cherry, Allan P Davis, Kara Dolinski, Selina S Dwight, Janan T Eppig, et al. Gene ontology: tool for the unification of biology. Nature genetics, 25(1):25–29, 2000.

[ACZH10] Medha Atre, Vineet Chaoji, Mohammed J Zaki, and James A Hendler. Matrix bit loaded: a scalable lightweight join query processor for rdf data. In Proceedings of the 19th international conference on World wide web, pages 41–50. ACM, 2010.

[ADFDB13] Azzeddine Amiar, Mickaël Delahaye, Ylies Falcone, and Lydie Du Bousquet. Compressing microcontroller execution traces to assist system analysis. In Embedded Systems: Design, Analysis and Verification, pages 139–150. Springer, 2013.

[All83] James F Allen. Maintaining knowledge about temporal intervals. Communications of the ACM, 26(11):832–843, 1983.

[AMH08] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 967–980. ACM, 2008.

[AMMH07] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In Proceedings of the 33rd international conference on Very large data bases, pages 411–422. VLDB Endowment, 2007.

[AMMH09] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. The VLDB Journal—The International Journal on Very Large Data Bases, 18(2):385–406, 2009.

[ASH09] Medha Atre, Jagannathan Srinivasan, and James A Hendler. Bitmat: A main memory rdf triple store. Tetherless World Constellation, Rensselar Plytehcnic Institute, Troy NY, 2009.

[BEH$^+$02] Erol Bozsak, Marc Ehrig, Siegfried Handschuh, Andreas Hotho, Alexander Maedche, Boris Motik, Daniel Oberle, Christoph Schmitz, Steffen Staab, Ljiljana Stojanovic, et al. Kaon—towards a large scale semantic web. In E-Commerce and Web Technologies, pages 304–313. Springer, 2002.

[Ber] Sion Berkowits. Pin- A Dynamic Binary Instrumentation Toll. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`. [Online; accessed 24-Febuary-2015].

[BGZ$^+$10] Robert Binna, Wolfgang Gassler, Eva Zangerle, Dominic Pacher, and Günther Specht. Spiderstore: exploiting main memory for efficient rdf graph representation and fast querying. In Proceedings of workshop on semantic data management (SemData@ VLDB), 2010.

[BKVH02] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In The Semantic Web—ISWC 2002, pages 54–68. Springer, 2002.

[Bro05] Jeen Broekstra. Storage, Querying and Inferencing for Semantic Web Languages. `http://www.cs.vu.nl/en/Images/broekstra_2005_thesis_tcm75-92137.pdf`, 2005. [Online; accessed 14-March-2015].

[CFP15] Marcos Cunha, Nicolas Fournel, and Frédéric Pétrot. Collecting traces in dynamic binary translation based virtual prototyping platforms. In Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, page 4. ACM, 2015.

[Com] Trace Compass. Trace Compass. `https://projects.eclipse.org/projects/tools.tracecompass`. [Online; accessed 28-January-2015].

[Cor89] Thomas A Corbi. Program understanding: Challenge for the 1990s. IBM Systems Journal, 28(2):294–306, 1989.

[Cyg05] Richard Cyganiak. A relational algebra for sparql. Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170, page 35, 2005.

[CZVD$^+$09] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. Software Engineering, IEEE Transactions on, 35(5):684–702, 2009.

[Dev] Valgrind Developers. Valgrind. `http://valgrind.org/`. [Online; accessed 24-Febuary-2015].

[dG] Laboratoire Informatique de Grenoble. Laboratoire Informatique de Grenoble. https://www.liglab.fr/. [Online; accessed 29-April-2015].

[DHV+13] Damien Dosimont, Guillaume Huard, Jean-Marc Vincent, R Lamarche-Perrin, Y Demazeau, J Emeras, C Ruiz, O Richard, R Laraki, P Mertikopoulos, et al. La visualisation de traces, support à l'analyse, déverminage et optimisation d'applications de calcul haute performance. In Actes de l'atelier Visualisation d'informations, interaction et fouille de données (VIF) de la 13e Conférence Francophone sur l'Extraction et la Gestion des Connaissances (EGC'2013), pages 55–66, 2013.

[DOG12] C. Faye David, Cure Olivier, and Blin Guillaume. A survey of rdf storage approaches. ARIMA Journal, 15:11–35, 2012.

[dt] Lauterbach development tools. Trace32. http://www.lauterbach.com/frames.html?home.html. [Online; accessed 28-January-2015].

[EH00] Touradj Ebrahimi and Caspar Horne. Mpeg-4 natural video coding–an overview. Signal Processing: Image Communication, 15(4):365–385, 2000.

[EM09] Orri Erling and Ivan Mikhailov. Rdf support in the virtuoso dbms. In Networked Knowledge-Networked Media, pages 7–24. Springer, 2009.

[For82] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial intelligence, 19(1):17–37, 1982.

[Fou] Université Joseph Fourier. Université Joseph Fourier Grenoble, Science, Technologies, Santé. https://www.ujf-grenoble.fr/. [Online; accessed 29-April-2015].

[Fou11] The Apache Software Foundation. TDB Architecture. http://jena.apache.org/documentation/tdb/architecture.html, 2011. [Online; accessed 18-March-2015].

[Gal12] Antony Galton. States, processes and events, and the ontology of causal relations. In FOIS, pages 279–292, 2012.

[GKM+13] François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge Quiané-Ruiz, and Stamatis Zampetakis. Cliquesquare: efficient hadoop-based rdf query processing. In BDA'13-Journées de Bases de Données Avancées, 2013.

[Gru95] Thomas R Gruber. Toward principles for the design of ontologies used for knowledge sharing? International journal of human-computer studies, 43(5):907–928, 1995.

[GSt] GStreamer. GStreamer: Open source multimedia framework. http://gstreamer.freedesktop.org/. [Online; accessed 24-Febuary-2015].

[HD05] Andreas Harth and Stefan Decker. Optimized index structures for querying rdf from the web. In Web Congress, 2005. LA-WEB 2005. Third Latin American, pages 10–pp. IEEE, 2005.

[HG03] Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk rdf storage. 2003.

[HG04] Jonathan Hayes and Claudio Gutierrez. Bipartite graphs as intermediate model for rdf. In The Semantic Web–ISWC 2004, pages 47–61. Springer, 2004.

[HLS09] Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The design and implementation of a clustered rdf store. In 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009), pages 94–109, 2009.

[HP04] Jerry R Hobbs and Feng Pan. An ontology of time for the semantic web. ACM Transactions on Asian Language Information Processing (TALIP), 3(1):66–85, 2004.

[HPS14] Patrick J. Hayes and Peter F. Patel-Schneider. RDF 1.1 Semantics. `http://www.w3.org/TR/rdf11-mt/`, 2014. [Online; accessed 27-february-2015].

[HUA12] M Hemalatha, V Uma, and G Aghila. Time ontology with reference event based temporal relations (retr). International Journal of Web & Semantic Technology, 3(1), 2012.

[IHD04] Harold Boley Said Tabet Benjamen Grosof Ian Horrocks, Peter F. Patel-Schneider and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. `http://www.w3.org/Submission/SWRL/`, 2004. [Online; accessed 29-March-2015].

[Jen] Apache Jena. Class MakeTemp. `https://jena.apache.org/documentation/javadoc/jena/com/hp/hpl/jena/reasoner/rulesys/builtins/MakeTemp.html`. [Online; accessed 30-March-2015].

[JK05] Maciej Janik and Krys Kochut. Brahms: A workbench rdf store and high performance memory system for semantic association discovery. In The Semantic Web–ISWC 2005, pages 431–445. Springer, 2005.

[KFI+12] Christiane Kamdem Kengne, Leon Constantin Fopa, Noha Ibrahim, Alexandre Termier, Marie-Christine Rousset, and Takashi Washio. Enhancing the analysis of large multimedia applications execution traces with frameminer. In Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on Data Mining, pages 595–602. IEEE, 2012.

[KIRT13] Christiane Kamdem Kengne, Noha Ibrahim, Marie-Christine Rousset, and Maurice Tchuente. Distance-based trace diagnosis for multimedia applications: Help me ted! In Semantic Computing (ICSC), 2013 IEEE Seventh International Conference on, pages 306–309. IEEE, 2013.

[KJLW12] Daniel Kless, Ludger Jansen, Jutta Lindenthal, and Jens Wiebensohn. A method for re-engineering a thesaurus into an ontology. In FOIS, pages 133–146, 2012.

[KKFT+13] Christiane Kamdem Kengne, Leon Constantin Fopa, Alexandre Termier, Noha Ibrahim, Marie-Christine Rousset, Takashi Washio, and Miguel Santana. Efficiently rewriting large multimedia application execution traces with few event sequences. In Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 1348–1356. ACM, 2013.

[KKTC12] Vaibhav Khadilkar, Murat Kantarcioglu, Bhavani Thuraisingham, and Paolo Castagna. Jena-hbase: A distributed, scalable and efficient rdf triple store. In Proceedings of the 11th International Semantic Web Conference Posters & Demonstrations Track, ISWC-PD, volume 12, pages 85–88. Citeseer, 2012.

[KOM05] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. Owlim–a pragmatic semantic repository for owl. In Web Information Systems Engineering–WISE 2005 Workshops, pages 182–192. Springer, 2005.

[LB94] James R Larus and Thomas Ball. Rewriting executable files to measure program behavior. Software: Practice and Experience, 24(2):197–218, 1994.

[LCBT+12] Patricia López Cueva, Aurélie Bertaux, Alexandre Termier, Jean François Méhaut, and Miguel Santana. Debugging embedded multimedia application traces through periodic pattern mining. In Proceedings of the tenth ACM international conference on Embedded software, pages 13–22. ACM, 2012.

[LIJ+14] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, et al. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web, 2014.

[LTP13] Sofiane Lagraa, Alexandre Termier, and Frédéric Pétrot. Data mining mpsoc simulation traces to identify concurrent memory access patterns. In Proceedings of the Conference on Design, Automation and Test in Europe, pages 755–760. EDA Consortium, 2013.

[mag] magillem. Magillem Design Services. `http://www.magillem.com/`. [Online; accessed 29-April-2015].

[MBG+02] Claudio Masolo, Stefano Borgo, Aldo Gangemi, Nicola Guarino, Alessandro Oltramari, Ro Oltramari, Luc Schneider, Lead Partner Istc-cnr, and Ian Horrocks. Wonderweb deliverable d17. the wonderweb library of foundational ontologies and the dolce ontology. 2002.

[Mil95] George A Miller. Wordnet: a lexical database for english. Communications of the ACM, 38(11):39–41, 1995.

[Mon08a] MonetDB. MonetDB Index definition. https://www.monetdb.org/Documentation/Manuals/SQLreference/Indices, 2008. [Online; accessed 02-April-2015].

[Mon08b] MonetDB. MonetDB Storage model. https://www.monetdb.org/Documentation/Manuals/MonetDB/Architecture/Storagemodel, 2008. [Online; accessed 13-April-2015].

[MRP+05] William H Milnor, Cartic Ramakrishnan, Matthew Perry, Amit P Sheth, John A Miller, and Krzysztof Kochut. Discovering informative subgraphs in rdf graphs. 2005.

[MS05] Boris Motik and R Studer. Kaon2–a scalable reasoning tool for the semantic web. In Proceedings of the 2nd European Semantic Web Conference (ESWC'05), Heraklion, Greece, volume 17, 2005.

[NM+01] Natalya F Noy, Deborah L McGuinness, et al. Ontology development 101: A guide to creating your first ontology, 2001.

[NMW97] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical strcture in sequences: A linear-time algorithm. J. Artif. Intell. Res.(JAIR), 7:67–82, 1997.

[NW10] Thomas Neumann and Gerhard Weikum. The rdf-3x engine for scalable management of rdf data. The VLDB Journal, 19(1):91–113, 2010.

[Ont14] Ontotext. GraphDB-Lite Reasoner. https://confluence.ontotext.com/display/GraphDB6/GraphDB-Lite+Reasoner, 2014. [Online; accessed 31-March-2015].

[Pag] Generoso Pagano. Framesoc: Trace Management and Analysis Infrastructure. http://soctrace-inria.github.io/framesoc/. [Online; accessed 26-Febuary-2015].

[PB14] Minh-Duc Pham and Peter A Boncz. Monetdb/rdf: Discovering and exploiting the emergent schema of rdf data. ERCIM News, 96:41–42, 2014.

[PCR12] Roshan Punnoose, Adina Crainiceanu, and David Rapp. Rya: a scalable rdf triple store for the clouds. In Proceedings of the 1st International Workshop on Cloud Intelligence, page 4. ACM, 2012.

[PMM14] Generoso Pagano and Vania Marangozova-Martin. Framesoc workbench: Facilitating trace analysis through a consistent user interface. Technical Report RT-0447, Inria, 2014. hal-00977887.

[PNLF08] Mark Proctor, Michael Neale, Peter Lin, and Michael Frandsen. Drools documentation. JBoss. org, Tech. Rep, 2008.

[Pro] Probayes. Probayes, Mastering Uncertainty. http://www.probayes.com/fr/. [Online; accessed 29-April-2015].

[Pro08] Protégé. SWRLExtensionsBuiltIns. http://protege.cim3.net/cgi-bin/wiki.pl?SWRLExtensionsBuiltIns, 2008. [Online; accessed 30-March-2015].

[PRRR+09] Carlos Prada-Rojas, Frederic Riss, Xavier Raynaud, Serge De Paoli, and Miguel Santana. Observation tools for debugging and performance analysis of embedded linux applications. In Conference on System Software, SoC and Silicon Debug-S4D, 2009.

[PRSDP+10] Carlos Prada-Rojas, Miguel Santana, Serge De-Paoli, Xavier Raynaud, et al. Summarizing embedded execution traces through a compact view. In Conference on System Software, SoC and Silicon Debug S4D, 2010.

[RA] INRIA Grenoble Rhône-Alpes. INRIA, Inventeur du monde numérique. http://www.inria.fr/centre/grenoble. [Online; accessed 29-April-2015].

[RH06] Francisco Ruiz and José R Hilera. Using ontologies in software engineering and technology. In Ontologies for software engineering and software technology, pages 49–102. Springer, 2006.

[Ses15] Sesame. System documentation for Sesame 2. http://rdf4j.org/sesame/2.8/docs/system.docbook?view#chapter-introduction, 2015. [Online; accessed 31-March-2015].

[SGF99] K. Smith-Gratto and M. Fisher. "gestalt theory: A foundation for instructional screen design. In Journal of Instructional Technology Systems, pages 361–371, 1999.

[SKW07] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In Proceedings of the 16th international conference on World Wide Web, pages 697–706. ACM, 2007.

[SLI] SLIDE. Scalable Information Discovery and Exploitation. http://slide.liglab.fr/. [Online; accessed 21-April-2015].

[SM96] Ravindra Singh and Naurang Singh Mangat. Elements of survey sampling, volume 15. Springer Science & Business Media, 1996.

[STL] STLinux. STLinux. http://www.stlinux.com/. [Online; accessed 26-Febuary-2015].

[STM] STMicroelectronics. STMicroelectronics. http://www.st.com/web/en/home.html. [Online; accessed 21-April-2015].

[UPS07] Octavian Udrea, Andrea Pugliese, and VS Subrahmanian. Grin: A graph based rdf index. In AAAI, volume 1, pages 1465–1470, 2007.

[VDVM98] Paul E Van Der Vet and Nicolaas JI Mars. Bottom-up construction of ontologies. Knowledge and Data Engineering, IEEE Transactions on, 10(4):513–526, 1998.

[Vir99] OpenLink Virtuoso. RDF Data Access and Data Management. http://docs.openlinksw.com/virtuoso/rdfsparqlrule.html, 1999. [Online; accessed 31-March-2015].

[W3C08] W3C. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/, 2008. [Online; accessed 27-february-2015].

[W3C13] W3C. SPARQL 1.1 Overview. http://www.w3.org/TR/sparql11-overview/, 2013. [Online; accessed 27-february-2015].

[WGA05] David Wood, Paul Gearon, and Tom Adams. Kowari: A platform for semantic web storage and analysis. In XTech 2005 Conference, pages 05–0402. Citeseer, 2005.

[WH09] Jesse Weaver and James A Hendler. Parallel materialization of the finite rdfs closure for hundreds of millions of triples. Springer, 2009.

[Wil06] Kevin Wilkinson. Jena property table implementation, 2006.

[WKB08] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. Proceedings of the VLDB Endowment, 1(1):1008–1019, 2008.

[WSK+03] Kevin Wilkinson, Craig Sayers, Harumi A Kuno, Dave Reynolds, et al. Efficient rdf storage and retrieval in jena2. In SWDB, volume 3, pages 131–150. Citeseer, 2003.

[Zel09] Andreas Zeller. Why programs fail: a guide to systematic debugging. Elsevier, 2009.

[ZLC03] Xiaosong Zhou, Eric Q Li, and Yen-Kuang Chen. Implementation of h. 264 decoder on general-purpose processors with media instructions. In Electronic Imaging 2003, pages 224–235. International Society for Optics and Photonics, 2003.

[ZXHW10] Jia Zou, Jing Xiao, Rui Hou, and Yanqi Wang. Frequent instruction sequential pattern mining in hardware sample data. In Data Mining (ICDM), 2010 IEEE 10th International Conference on, pages 1205–1210. IEEE, 2010.