SEMANTIC ENRICHMENT FOR THE AUTOMATED CUSTOMIZATION AND INTEROPERABILITY OF UBL SCHEMAS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

YALIN YARIMAĞAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

MARCH 2008

Approval of the thesis:

SEMANTIC ENRICHMENT FOR THE AUTOMATED CUSTOMIZATION AND INTEROPERABILITY OF UBL SCHEMAS

submitted by **Yalın YARIMAĞAN** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, **Graduate School of Natural and Applied Sciences** _____

Prof. Dr. Volkan Atalay
Head of Department, **Computer Engineering** _____

Prof. Dr. Asuman Doğaç
Supervisor, **Computer Engineering Department, METU** _____

Examining Committe Members:

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Department, METU _____

Prof. Dr. Asuman Doğaç
Computer Engineering Department, METU _____

Prof. Dr. Mehmet Reşit Tolun
Computer Engineering Department, Çankaya University _____

Prof. Dr. Özgür Ulusoy
Computer Engineering Department, Bilkent University _____

Assoc. Prof. Dr. Nihan Kesim Çiçekli
Computer Engineering Department, METU _____

Date: 05.03.2008

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Yalın YARIMAĞAN

Signature :

ABSTRACT

**SEMANTIC ENRICHMENT
FOR THE AUTOMATED CUSTOMIZATION AND INTEROPERABILITY
OF UBL SCHEMAS**

YARIMAĞAN, Yalın

Ph.D., Department of Computer Engineering

Supervisor: Prof. Dr. Asuman DOĞAÇ

March 2008, 167 pages

Universal Business Language (UBL) is an initiative to develop common business document schemas to provide standardization in the electronic business domain. However, businesses operate in different industry, geopolitical, and regulatory contexts and consequently they have different rules and requirements for the information they exchange.

In this thesis, we provide semantic enrichment mechanisms for UBL that (i) allow automated customization of document schemas in response to contextual needs and (ii) maintain interoperability among different schema versions.

For this purpose, we develop ontologies to provide machine processable representations for context domains, annotate custom components using classes from those ontologies and show that using these semantic annotations, automated discovery of components and automated customization of schemas becomes possible. We then provide a UBL Component Ontology that represents the semantics of individual components and their structural relationships and show that when an ontology reasoner interprets the expressions from this ontology, it computes equivalence and class-

subclass relationships between classes representing components with similar content. Finally we describe how these computed relationships are used by a translation mechanism to establish interoperability among schema versions customized for different business context values.

Keywords: UBL, context, ontology, semantics, interoperability

ÖZ

**UBL ŞEMALARININ
OTOMATİK UYARLANABİLMESİ VE BİRLİKTE-İŞLERLİĞİ İÇİN
ANLAMSAL ZENGİNLEŞTİRME**

YARIMAĞAN, Yalın
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Asuman DOĞAÇ

Mart 2008, 167 sayfa

Universal Business Language (UBL), elektronik iş alanında standardizasyon sağlamak için ortak iş dökümanları geliştirme çalışmasıdır. Ancak ticari işletmeler farklı endüstriyel, jeopolitik ve düzenleme koşullarında faaliyet göstermektedirler ve bu nedenle döküman alış-verişi için kullandıkları bilgilerle ilgili olarak birbirinden farklı kural ve gereksinimleri vardır.

Bu tezde, UBL için (i) döküman şemalarının bağlamsal ihtiyaçlara göre otomatik olarak uyarlanması ve (ii) farklı şema sürümleri arasında birlikte-işlerliğin sürdürülmesini sağlayan anlamsal zenginleştirme mekanizmaları sağlanmıştır.

Bu amaçla; bağlam alanları için makinalar tarafından işlenebilir gösterim sağlayan ontolojiler geliştirilmiş, uyarlanmış bileşenleri bu ontolojileri kullanarak etiketlenmiş ve bu anlamsal etiketlerin kullanılmasıyla otomatik bileşen keşfi ve otomatik şema uyarlamasının mümkün olabildiği gösterilmiştir. Daha sonra bireysel bileşenlerin anlamlarını ve aralarındaki yapısal ilişkileri tanımlayan bir UBL Bileşen Ontolojisi sağlanmış ve bu ontolojideki ifadelerin bir ontoloji yorumlayıcısı tarafından yorumlandığı zaman bileşenler arasındaki ilişkilerin derecesine gore onları gösteren

ontoloji sınıfları arasında denklik ve alt-üst sınıf ilişkileri hesaplanabileceği gösterilmiştir. Son olarak da hesaplanan ilişkilerin bir anlamsal tercüme mekanizması tarafından farklı iş bağlamlarına gore uyarlanmış şema versiyonlarının birlikte-işlerliğini sağlamak amacıyla nasıl kullanılabileceği gösterilmiştir.

Anahtar Kelimeler: UBL, bağlam, ontoloji, anlamsallık, birlikte-işlerlik

To my precious wife Evrim and our lovely daughter Defne..

ACKNOWLEDGMENTS

First of all, I would like to express my deepest gratitude to my supervisor Prof. Dr. Asuman Doğaç for all her guidance, insight and continuous support throughout this study. She has been the one encouraging me for starting this study in the first place and without her great advice and help this work would have never been possible.

I wish to express a lot of thanks to Prof. Dr. İsmail Hakkı Toroslu and Prof. Dr. Mehmet Reşit Tolun for their valuable suggestions and comments throughout the steering meetings of this study.

Special thanks to Yıldıray Kabak, Gökçe Banu Laleci and all other fellows at the Software Research and Development Center for the cooperation and support they have provided.

I also want to express my gratefulness to my precious wife Evrim, for all her love, patience and friendship. Without her continuous support, encouragement and especially taking care of everything else in our lives, I would have never had the time and strength to complete this work.

Finally, I would like to thank all members of my extended family for the support, understanding and help they have provided during this period.

This study is supported in part by the European Commission, Project No:ICT-213031 iSURF Project and the Scientific and Technical Research Council of Turkey (TUBITAK), Project No: EEEAG 105E068.

TABLE OF CONTENTS

ABSTRACT iv
ÖZ vi
ACKNOWLEDGMENTS ix
TABLE OF CONTENTS x
LIST OF FIGURES xii
CHAPTER
1. INTRODUCTION 1
1.1 Contributions of the Thesis 7
1.2 Outline 11
2. BACKGROUND ON ENABLING TECHNOLOGIES 12
2.1 Electronic Business Standards 12
2.1.1 Electronic Business using eXtensible Markup Language (ebXML) 13
2.1.2 The ebXML Core Components 14
2.1.3 Universal Business Language (UBL) 17
2.2 Ontologies 24
2.3 Web Ontology Language (OWL) 28
3. USE OF ONTOLOGIES FOR SEMANTIC ANNOTATION 33
3.1 Ontology Development for the Industrial Classification Context 34
3.1.1 Developing Individual Ontologies 36
3.1.2 Aligning Individual Ontologies 39
3.2 Generalization to Other Context Domains 49
4. USE OF SEMANTIC CONSTRUCTS FOR INTEROPERABILITY 51
4.1 Web Ontology Language to Describe UBL Components and Schemas 52
4.2 UBL Component Ontology 53
4.3 Computing Translations through Reasoning 63
4.4 Components with Common Content 69
4.4.1. Structurally Different Components 74
5. SYSTEM ARCHITECTURE AND OPERATION 80
5.1 Knowledge Base 80

5.2 User Tools 82
5.3 Services Layer 84
5.3.1 Reasoning Layer 84
5.3.2 Component Registration Service 85
5.3.3 Component Discovery Service 89
5.3.4 Document Schema Customization Service 93
5.3.5 Component Merge Service 98
5.3.6 Document Instance Translation Service 108
6. IMPLEMENTATION 119
6.1 User Applications 120
6.2 Reasoning Performance 127
7. RELATED WORK 129
7.1 Context Modeling 129
7.2 Ontology Based Interoperability of Information 131
8. CONCLUSIONS AND FUTURE WORK 136
REFERENCES 139
APPENDICES
A. UBL CUSTOMIZATION METHODOLOGY 147
B. WEB ONTOLOGY LANGUAGE 153
C. DESCRIPTION LOGICS 159
VITA 167

LIST OF FIGURES

FIGURES

Figure 1-1 - Developing a Semantic Representation for the Industrial Classification Context 5

Figure 1-2 - The UBL community model in which interoperability between different communities is ensured through a semantic-based translation mechanism 7

Figure 2-1 - Context based specialization example for Core Components [25] 15

Figure 2-2 - The context for the use of UBL business documents [5] 21

Figure 3-1 - An extract from the NAICS ontology 36

Figure 3-2 - Corresponding OWL Definition for Figure 3-1 37

Figure 3-3 - Class Hierarchy of the ISIC ontology 38

Figure 3-4 - Class Hierarchy of the NACE ontology 38

Figure 3-5 - Class Hierarchy of the NAICS ontology 39

Figure 3-6 - OWL definition for the Context Ontology of the Industrial Classification Context 40

Figure 3-7 - Sample OWL definitions to specify equivalence 41

Figure 3-8 - Sample OWL definitions to specify composition 42

Figure 3-9 - Sample OWL definitions to specify subsumption 43

Figure 3-10 - Ontology Alignment Operations 44

Figure 3-11 - Class hierarchy fragments from a sample Context Ontology 46

Figure 3-12 - Inferred Context Ontology corresponding to the Context Ontology fragment in Figure 3-11 47

Figure 3-13 - Inferred Context Ontology in Figure 3-12 with Component Annotations 49

Figure 4-1 - Simplified Order Schema 53

Figure 4-2 - XSD Definitions corresponding to the Order Schema in Figure 4-1 54

Figure 4-3 - Component Ontology Template 55

Figure 4-4 - OWL Definitions corresponding to the UBL Ontology Template in Figure 4-3 56

Figure 4-5 - OWL definitions for type classes of Order schema in Figure 4-1 57

Figure 4-6 - OWL definitions for element classes of Order schema in Figure 4-1 58
Figure 4-7 - OWL definitions for concept classes of Order schema in Figure 4-1 58
Figure 4-8 - OWL Definitions corresponding to the FamilyNameType in Figure 4-2 60
Figure 4-9 - OWL Definitions corresponding to the OrderType in Figure 4-2 61
Figure 4-10 - OWL Definitions corresponding to the Order in Figure 4-2 62
Figure 4-11 - An example custom version of the Order schema in Figure 4-1 63
Figure 4-12 - DL axioms corresponding to the Order schema in Figure 4-2 64
Figure 4-13 - DL axioms corresponding to the CustomOrder schema in Figure 4-11 65
Figure 4-14 - Completion tree for the axiom (Buyer ∩ ¬Customer) 66
Figure 4-15 - Completion tree for axiom (Customer ∩ ¬Buyer) 67
Figure 4-16 - Completion tree for axiom (Order ∩ ¬CustomOrder) 68
Figure 4-17 - Completion tree for axiom (CustomOrder ∩ ¬Order) 69
Figure 4-18 - An alternative component for the Order in Figure 4-1 70
Figure 4-19 - DL axioms corresponding to the NewOrder in Figure 4-18 71
Figure 4-20 - Completion tree for axiom (Buyer ∩ ¬NewBuyer) 72
Figure 4-21 - Completion tree for axiom (NewBuyer ∩ ¬Buyer) 72
Figure 4-22 - Completion tree for axiom (NewOrder ∩ ¬Order) 73
Figure 4-23 - Simplified ProviderParty versions 74
Figure 4-24 - DL axioms corresponding to the Figure 4-23 75
Figure 4-25 - Completion tree for axiom (ProviderParty ⊆ ProviderParty2) 76
Figure 4-26 - Completion tree of axiom (ProviderParty ⊆ ProviderParty2) 77
Figure 4-27 - Processing times for original and modified ontologies 79
Figure 5-1 - System Architecture 81
Figure 5-2 - OWL Classes representing the metadata of standard and custom UBL components 85
Figure 5-3 - UBLComponentMetadata instance for the UBL Item component 86
Figure 5-4 - CustomComponentMetadata instance for a custom component 88
Figure 5-5 - Context class with multiple inheritance 90
Figure 5-6 - Component Discovery Algorithm 91
Figure 5-7 - Influence of component versions on lower level context values 92
Figure 5-8 - Simplified Order Schema 94
Figure 5-9 - Simplified Catalogue Schema 94

Figure 5-10 - Document Schema Customization Algorithm 95
Figure 5-11 - Sample Context Ontology for the Product Classification context 96
Figure 5-12 - Sample Context Ontology for the Industrial Classification context 96
Figure 5-13 - Catalogue Schema in Figure 5-9 customized for the Antibiotics Manufacturing business context value 98
Figure 5-14 - Simplified Item component 98
Figure 5-15 - Sample customized version of the Item component in Figure 5-14 for the Retail Trade context 99
Figure 5-16 - Sample customized version of the Item component in Figure 5-14 for the Drugs and Pharmaceutical Products context 100
Figure 5-17 - Component Merge Algorithm 101
Figure 5-18 - A sample custom version of the Item component in Figure 5-14, generated by merging the versions in Figure 5-15 and Figure 5-16 102
Figure 5-19 - Sample Person component 104
Figure 5-20 - Sample Individual component 104
Figure 5-21 - UBL FinancialInstitution component 105
Figure 5-22 - Sample custom version of the FinancialInstitution component in Figure 5-21 extended with Person component in Figure 5-19 105
Figure 5-23 - Sample custom version of the FinancialInstitution component in Figure 5-21 extended with Individual component in Figure 5-20 106
Figure 5-24 - Semantic Redundancy Elimination Algorithm 107
Figure 5-25 - Sample Order schema 109
Figure 5-26 - Sample Order document conforming to the Order schema in Figure 5-25 110
Figure 5-27 - Sample customized version of the document schema in Figure 5-25 110
Figure 5-28 - Extract from the inferred UBL Component Ontology class hierarchy representing schemas in Figure 5-25 and Figure 5-27 111
Figure 5-29 - Translated version of the document instance in Figure 5-26 112
Figure 5-30 - Component Translation Algorithm 113
Figure 5-31 - findTargetCmp method in Figure 5-30 114
Figure 5-32 - checkForAMatch method in Figure 5-31 115
Figure 5-33 - Simplified Catalogue schema 116
Figure 5-34 - Document instance conforming to the Catalogue schema in Figure 5-33 117
Figure 5-35 - Sample customized version of the Catalogue schema in Figure 5-33 117

Figure 5-36 - Document instance in Figure 5-34 translated to the Catalogue schema in Figure 5-35 118
Figure 6-1 - The Component Customization Tool - Context and Component specification 121
Figure 6-2 - The Component Customization Tool - Customization 122
Figure 6-3 - The Extension Component Definition Tool – The Context Tab 123
Figure 6-4 - The Extension Component Definition Tool - The Type Tab 124
Figure 6-5 - The Extension Component Definition Tool - The Component Tab 125
Figure 6-6 - The Document Schema Customization Tool 126
Figure 6-7 - The Translation Tool 127
Figure 6-8 - Performance of the Translation Engine for computing equivalence and subsumption relationships between Component Ontology classes 128
Figure A-1 - The UBL Customization Flow 148
Figure A-2 - An example UBL Context Chain 149
Figure A-3 - Example complex PartyType 150
Figure A-4 - XSD definitions to extend the PartyType in Figure A-3 151
Figure A-5 - XSD definitions to restrict the PartyType in Figure A-3 151
Figure C-1 - Syntax and semantics for S family of DLs 160
Figure C-2 - Tableau expansion rules for SHIQ 166

CHAPTER 1

INTRODUCTION

Electronic commerce is advancing at a fast pace, replacing conventional means for conducting business throughout the world. The Business-to-Business form of electronic commerce (B2B) includes a broad range of inter-company transactions including wholesale trade, company purchases of services, resources, technology, manufactured parts and components and capital equipment together with various types of financial transactions between companies [1].

Many organizations around the world are structuring their organizations according to B2B requirements to take advantage of the potential of more automation, efficient business processes and global visibility especially in domains such as the healthcare, travel, supply chain management, procurement, shipping and warehousing [2]. A study by the Gartner Group estimates the value of e-commerce market to be $8.5 trillion in 2005 [3].

Nevertheless, transforming a business so that it substitutes computer processing and electronic communication in place of labor services for the production of electronic transactions is not a simple task. Businesses need to sustain substantial expenses in order to adapt to this new form of business environment.

A recent survey on B2B technologies [4] suggest that interactions in a B2B system occur in the following three layers:

- The *communication layer* that provides the protocols for exchanging messages among remotely located partners.

- The *content layer* that provides languages and models to describe and organize information in such a way that all involved parties can understand the semantics of exchanged data and content and types of business documents.

- The *business process* layer that is concerned with the protocols that outline the conversational interactions among different business partners.

Reusing well-understood standard patterns for each of these layers makes the transformation easier to implement, manage, and improve. Adopting common standards reduce development and maintenance costs, improve performance, and enhance business relationships.

The Universal Business Language (UBL) [5] is an OASIS [6] specification addressing this standardization need for the *content layer*. Following is the UBL overview provided by the OASIS:

> *UBL is the product of an international effort to define a royalty-free library of standard electronic XML business documents such as purchase orders and invoices. Developed in an open and accountable OASIS Technical Committee with participation from a variety of industry data standards organizations, UBL is designed to plug directly into existing business, legal, auditing, and records management practices, eliminating the re-keying of data in existing fax and paper based supply chains and providing an entry point into electronic commerce for small and medium-sized businesses.*

Build upon the work and experience in the area, UBL is rapidly being adopted by user communities around the world. Following are some real-world implementation examples using UBL:

- UBL Invoice has been mandated by law for all public-sector business in Denmark and an estimated 1.2 million UBL invoices are currently exchanged in Denmark every month [7].

- A subset of the UBL Invoice has been recommended for all government use by the Swedish National Financial Management Authority [8].

- The Electronic Freight Management project of the U.S. Department of Transportation is developing a UBL based pilot project for a demonstration of electronic commerce in a real-world setting [9].

As these examples suggest, even though UBL is being embraced by governmental and large-scale organizations, it is not on track for meeting the goal of providing an entry point into electronic commerce for small and medium-sized businesses. It should be noted that businesses share many common data requirements, however, they operate in different industry, geopolitical, and regulatory contexts, and because of these differences they have different rules and requirements for the information they exchange in their business documents. Hence, it is not feasible to expect a set of standard document schemas to cover the needs of all kinds of businesses around the world. There is need for complementary mechanisms that can provide seamless tailoring of standard schemas based on individual user needs.

In an effort the address this need, UBL provides a customization methodology [10] to be followed by implementers. This methodology allows the modification of standard UBL schemas in response to contextual needs through XML Schema Language (XSD) [11] derivation operations. Schemas can be extended by adding new components or can be restricted by removing components or by limiting cardinality of components to a subset. Eight context categories are identified for this purpose: Geopolitical, Industry Classification, Product Classification, Business Process, Official Constraint, Business Process Role, Supporting Role and System Constraint. It is required that the context of a document or a component is expressed using a set of name-value pairs, that is, in the form of context category, context value pairs.

This UBL provided customization methodology focuses on the goal of providing syntactic interoperability, in other words, it ensures that customized components do not violate the integrity of UBL schemas and an XSD parser that can interpret standard UBL schemas can also interpret customized UBL schemas. However, it does not address semantic interoperability, that is, when a document schema needs to be customized for a business context, users need to manually discover or provide component versions applicable to that particular context and then replace custom versions in place of their standard counterparts to perform the schema customization. Clearly, this is a non-trivial task as it requires humans to thoroughly understand the

underlying rules and constraints of the UBL specification and modify their systems accordingly, increasing the likelihood of errors and cost. An evaluation work on the successes and challenges of B2B applications [12], states that reducing and even totally eliminating human intervention is crucial for improving automation and reducing costs for B2B applications.

In the work presented by this thesis, we present how to improve the UBL by providing machine processable semantic representations for context domains and describe how these semantics are utilized for automating tasks required for proper customization of UBL schemas. Then, by building on that capability, we provide a semantic translation mechanism that provides interoperability between schemas customized to support different contextual requirements.

Figure 1-1 displays the steps we take for developing semantic representations for context domains. We start by developing specialized converters to derive Web Ontology Language (OWL) [13] ontologies from classification schemas that are currently being used by the industry to represent activities corresponding to context domains. Examples include the International Standard Industrial Classification (ISIC) [14], the North American Industry Classification System (NAICS) [15], the Statistical Classification of Economic Activities in the European Community (NACE) [16]. We then show how similar concepts from these ontologies are related to each other using ontology alignment techniques and process the resulting aligned ontologies using reasoners to infer implicit relationships among their classes. This results in a set of ontologies that formally represent the semantics of context domains.

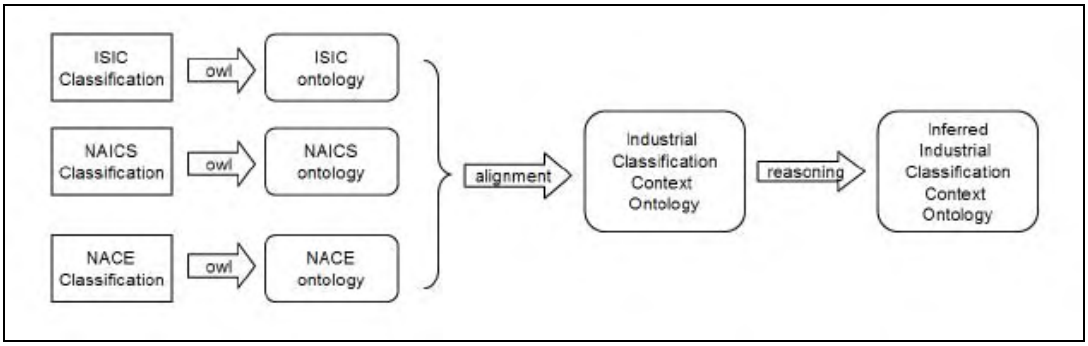

Figure 1-1 - Developing a Semantic Representation for the Industrial Classification Context

Once context ontologies are developed, customized UBL components are annotated using classes from these ontologies and stored in a global custom component repository that is supported by a knowledge base describing the repository content. As specified in the UBL customization methodology, UBL components are only allowed to be customized in response to contextual needs. Annotating custom component versions with classes from context ontologies expresses the context they apply to in a machine processable manner. This allows the development of automated processes that are capable of intelligently searching through custom components by interpreting class-subclass, equivalence and other relationships specified in context ontologies and gather applicable versions of components. These versions then replace their standard counterparts to customize document schemas for particular business context values.

The ability to automate component discovery leads to a significant new flexibility, that is, it becomes possible to merge multiple versions of a component to generate additional versions of that particular component. This greatly simplifies the component customization effort, as it is no longer necessary to manually provide customized components for every single business context. Instead, it is sufficient to customize components for individual context categories since combinations for multiple categories are automatically generated as needed.

The mechanisms described so far has the potential to streamline the customization process especially for those users in small and medium-scale business that do not have

the necessary resources or expertise required for properly tailoring standard UBL schemas for their business needs.

Nonetheless, it may be argued that the capability to seamlessly customize UBL schemas contradicts with the UBL objective for setting the standard for electronic business documents. That is, as user communities deviate from standard schemas to adapt customized schemas, it becomes harder to maintain the interoperability within the UBL community.

In order to provide a solution to this interoperability problem, we provide a semantic translation mechanism that is capable of converting documents between schemas customized for different business contexts. Figure 1-2 displays our approach, in which the UBL community is divided into smaller communities based on their business context. Each community uses schemas tailored for their particular business needs. Parties within a community use the same set of document schemas and communicate directly with each other. When parties from different communities need to make business with each other, they still use their local schemas and the interoperability is provided by translating documents from one schema to another.

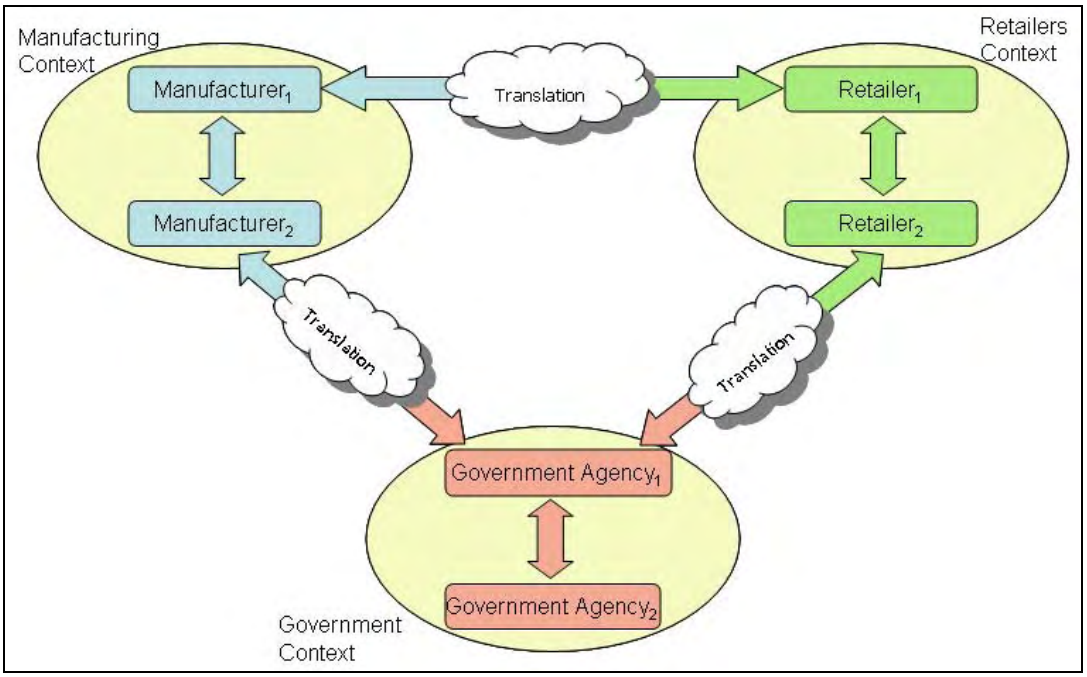

Figure 1-2 – The UBL community model in which interoperability between different communities is ensured through a semantic-based translation mechanism

In order to provide this semantic translation capability, we develop a UBL Component Ontology using OWL. This ontology provides a machine processable representation for UBL components by expressing the structure and the semantics of the components together with the correspondences among different components.

When this UBL Component Ontology is processes by a reasoner, explicitly asserted relationships reveal implicit relationships, which are then evaluated to compute equivalence and class-subclass relationships between ontology classes representing UBL components. By interpreting these equivalence and class-subclass relationships, UBL documents are translated from one schema to another based on the contextual needs.

1.1 Contributions of the Thesis

The specific contributions of the work described in this thesis are as follows:

- *Development of Context Ontologies for the formal representation of business context domains.* Various taxonomies have been widely used by the industry to classify business activities. Yet this approach has two major shortcomings: The taxonomies or classification schemas used are based on very limited formalization, making them unsuitable for machine processing. Next, their content is hardly interchangeable and it is much cumbersome to provide interoperability between systems utilizing different schemas even for a single business domain.

  In our work, we outline an approach to generate ontologies to represent such classification schemas to make them machine processable. This approach makes it possible to formally express the correspondences between similar concepts in different ontologies through ontology alignment operations. Furthermore, reasoners can process ontologies to infer implicit relationships between concepts, that is, given a minimum sufficient set of relationships between ontologies, reasoning can compute the set of all possible relationships among concepts and entities represented by those ontologies. This allows ontologies and in turn the taxonomies they represent to be semantically interoperable, that is, it becomes possible to use their content interchangeably.

- *Annotation of UBL components using classes from context ontologies.* Using classes from context ontologies for the annotation of UBL components allows us to express the applicable context of those components in a machine processable and semantically interoperable manner. This lays the foundation for automating various tasks which otherwise need to be carried out manually:

  - Together with a repository to store custom components and a knowledge base about repository content, our approach allows the development of automated processes that can intelligently search through custom components and are capable of schema customization.

  - The ability to automate component discovery leads to a significant new flexibility, that is, it becomes possible to merge multiple versions of a component to generate additional versions of that particular component. This greatly simplifies the component customization effort, as it is no longer necessary to manually provide customized components for every

single business context. Instead, it is sufficient to customize components for individual context categories since combinations for multiple categories are automatically generated as needed.

- *Development of a Component Ontology to represent the structure and semantics of UBL components.* UBL components are complex hierarchies of considerable depth including other components and even themselves in a recursive manner. As such, representation of correspondences and computation of similarities between different components or between variations of a single component becomes a vastly complex process. In our work, we present a UBL Component Ontology to provide a solution for both problems. By representing components using a formal language such as OWL, it becomes possible to express the relationships among components in a machine processable manner. Based on this capability, it becomes possible to utilize reasoners for the computation of similarities and other implicit relationships between different components.

- *Utilization of the UBL Component Ontology for the computation of similarities between UBL constructs.* The UBL Component Ontology consists of classes to represent the elements, type definitions and business concepts provided by the UBL Specification. For each class, a set of description logics expressions are generated to define the set of relationships distinguishing that particular class. In other words, for every UBL construct, we define the set of expressions that specify the necessary qualifications a class must posses in order to be qualified as a proper representation of that particular construct. When such expressions are interpreted by a description logics reasoner, equivalence and class-subclass relationships are computed among classes. An equivalence relationship between two UBL Component Ontology classes specifies the structural and semantic equivalence of those constructs. Similarly, a class-subclass relationship between two UBL Component Ontology classes specifies that the construct represented by the subclass subsumes the construct represented by the super class.

- *Developing an interoperability mechanism based on the UBL Component Ontology.* With the proliferation of the UBL standard and the mechanisms to

streamline the customization, such as the one presented in our work, it is reasonable to expect that user communities of any size would wish to customize the UBL schemas according to their needs. However, as communities prefer using non-standard schemas, it becomes harder to maintain the interoperability within the UBL community.

In our work, we provide an interoperability mechanism based on the UBL Component Ontology to support the integration of user communities adapting different UBL schema versions. Our integration approach is based on a semantic translation mechanism that converts UBL documents between schemas customized for different business needs. In order to accomplish that, the translation mechanism exploits the UBL Component Ontology for the computation of similarities among different components and uses those similarities for the determination of corresponding components in different context values

The provided approach allows UBL communities to adapt schemas suiting their needs, as whenever parties from different communities need to make business with each other, the interoperability is provided by translating documents from one schema to another.

- *Elimination of structural and semantic redundancy in automatically generated components.* The automated customization capability outlined above is capable of generating new versions for components through merging versions customized for different business context values. However, merely merging multiple components introduces redundancy as similar concepts might be represented differently in versions provided by different users. In order to avoid the creation of components with such redundant content, UBL Component Ontology is utilized for the computation of similarities between merge candidates and redundant content are eliminated.

- *Providing a prototype implementation for the realization of our approach.* Finally, in order to demonstrate the feasibility and effectiveness of the ideas presented in this thesis, we provide a prototype implementation that encompasses described features.

1.2 Outline

The rest of this thesis is organized as follows: Chapter 2 provides background information about technologies and standards that enable the presented work. Chapter 3 discusses our approach for the development of context ontologies and the utilization of those ontologies for the purposes of annotating UBL components. Chapter 4 describes the UBL Component Ontology. Chapter 5 is a presentation of our system architecture and Chapter 6 gives details about its implementation. Chapter 7 is dedicated to a brief survey of similar research from the literature. Chapter 8 concludes the thesis and suggests possible future research directions.

CHAPTER 2

BACKGROUND ON ENABLING TECHNOLOGIES

This chapter provides background information about standards and technologies that enable the work explained in this thesis.

2.1 Electronic Business Standards

There are many standards for structuring information to be electronically exchanged between and within businesses, organizations, government entities and other groups. These standards describe structures that emulate documents, for example purchase orders to automate purchasing. The motive for such standards is reusing well-understood patterns for reducing development and maintenance costs as it becomes easier to implement, manage and improve business processes.

Electronic Data Interchange (EDI) is a set of such standards for computer-to-computer exchange of business data in standard formats. There are two major sets of EDI standards: the United Nations recommended *United Nations/Electronic Data Interchange For Administration, Commerce, and Transport* (UN/EDIFACT) [17] is the international standard and is predominant outside of North America; and the U.S. standard *American National Standards Institute Accredited Standards Committee X12* (X12) [18] is predominant in North America. Both standards are widely used in electronic commerce transactions around the world.

With the proliferation of XML and Internet based technologies, new standards like the RosettaNet [19], the XML Common Business Library (xCBL) [20] and the Electronic Business using eXtensible Markup Language (ebXML) [21] have emerged adapting EDI ideas with contemporary technologies.

RosettaNet is a non-profit consortium of companies especially from Information Technology, Electronic Components, Semiconductor Manufacturing, and Solution Provider areas working to create, implement, and promote industry-wide open e-business process standards to form a common e-business language, aligning processes between supply chain partners on a global basis. The RosettaNet standards define message guidelines, business processes interface and implementation frameworks for interactions between companies. The primary focus is supply chain area, together with a scope of manufacturing, product and material data and service processes.

xCBL is another standard providing a collection of common business elements that underlie all EDI and Internet commerce protocols. xCBL has been developed and modeled after X12 and EDIFACT to preserve and extend the vast amount of existing EDI investments. It is based on a reusable component model to speed the implementation of standards and facilitate their interoperation by providing a common framework.

2.1.1 Electronic Business using eXtensible Markup Language (ebXML)

Electronic Business using eXtensible Markup Language (ebXML) is a family of XML based standards sponsored by OASIS [6] and UN/CEFACT [22] whose mission is to provide an open, XML-based infrastructure that enables the global use of electronic business information in an interoperable, secure, and consistent manner by all trading partners. ebXML recognizes that integration is a complex problem that requires standardization in a number of distinct areas and provides the following set of specifications:

- *ebXML Messaging Services*: Standard protocols like TCP/IP and HTTP are too low-level to serve the needs of electronic business. ebXML messaging addresses this problem by extending the SOAP [23] protocol to add features needed for the exchange of business documents: security, authentication, and non-repudiation.

- *ebXML Registry and Repository*: A standard protocol for accessing central registries and repositories of business data. These data can include such things as trading partner profiles and business document formats.

- *ebXML Collaboration Partner Profile and Collaboration Partner Agreement*: A profile to provide necessary information to do business with a specific trading partner, such as the business processes and document formats that it uses. When two parties trade for the first time, their profiles are combined into a Collaboration Partner Agreement that serves as the basis for their interaction.

- *ebXML Business Processes*: A generic metamodel for business processes with which any business process can be modeled in a machine-readable way to enable companies to deploy software that automatically adapts to the specific business processes of its trading partners.

- *ebXML Core Components*: A set of common business document components for basic business information such as addresses, products, trading parties. A core component used in a particular business context is called a business information entity (BIE). BIEs can be assembled into business document forms (purchase orders, invoices, etc.), and these forms, when populated with data, become interoperable business documents.

2.1.2 The ebXML Core Components

The ebXML Core Components Technical Specification (CCTS) [24], a part of the ebXML family of standards, provides a set of reusable concepts, called “Core Components”. A “Core Component” is a building block for the creation of a semantically correct and meaningful information exchange package and it contains only the information pieces necessary to describe a specific concept. “Core Components” represent common data elements of everyday business documents such as “Address”, “Amount”, or “Line Item” in a syntax neutral way.

Core Components are designed considering that a piece of information might mean different things or have different component parts in different business contexts. As an example, both "patient" and "carrier" are both examples of a "party" object, but in the U.S. a patient uses Social Security Number as an identifier while a shipping carrier might use a Standard Carrier Alpha Code. Carried even further, the overall structure of an invoice in grocery and farm equipment manufacturing might be the same. However, the pieces of information that describe a line item are quite different.

In order to account for this, CCTS also defines the “Business Context” concept as a mechanism for qualifying and refining Core Components according to their use under particular business circumstances. Once Business Contexts are identified, Core Components can be differentiated to take into account any necessary qualification and refinement needed to support the use of the Core Component in the given Business Context.

Figure 2-1 provides an example for this context based specialization taken from [25], in which a simplified Invoice component consisting of Party and Product components is shown. Depending on the applicable business context being Health Care, Motor Freight and Retail Grocery, Party component of Invoice represents Patient, Carrier and ShipTo concepts respectively.

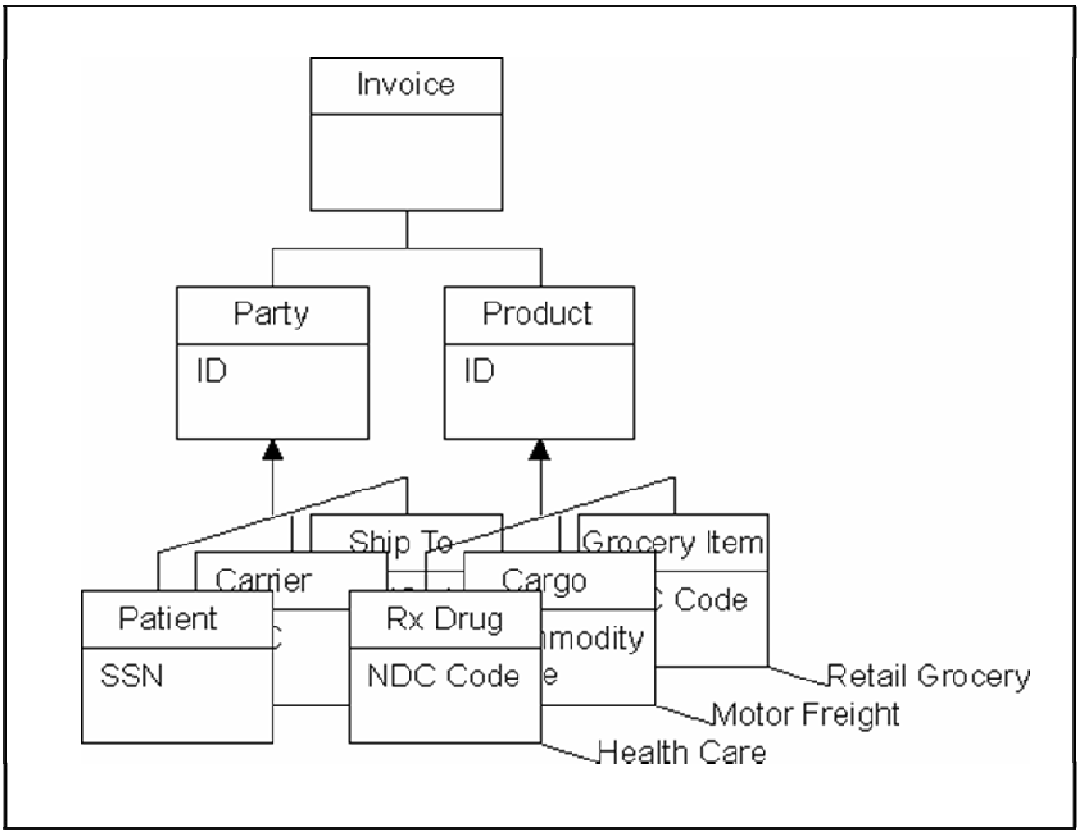

Figure 2-1 – Context based specialization example for Core Components [25]

These concepts allow the modeling of generic business processes, with generic documents exchanged at specific points in the process. The generic documents can identify placeholders for certain required information. For example, an invoice might have buyer, payee, payment information, and one or more line items. When the "context rules" are applied to these generic placeholders, fully described objects are created with all of the detailed information that is appropriate for the context. This produces a full document description that is tailored specifically for the context. Eight context categories are identified for this purpose. These categories and their brief description are provided below:

- *Business Process*: The Business Process name(s) as described using the UN/CEFACT Catalogue of Common Business Processes [26] as extended by the user.

- *Product Classification*: Factors influencing semantics that are the result of the goods or services being exchanged, handled, or paid for, etc. (e.g. the buying of consulting services as opposed to materials)

- *Industry Classification*: Semantic influences related to the industry or industries of the trading partners (e.g., product identification schemes used in different industries)

- *Geopolitical*: Geographical factors that influence Business Semantics (e.g., the structure of an address).

- *Official Constraints*: Legal and governmental influences on semantics (e.g. hazardous materials information required by law when shipping goods).

- *Business Process Role*: The actors conducting a particular Business Process, as identified in the UN/CEFACT Catalogue of Common Business Processes.

- *Supporting Role:* Semantic influences related to non-partner roles (e.g., data required by a third-party shipper in an order response going from seller to buyer.)

- *System Capabilities:* This Context Category exists to capture the limitations of systems (e.g. an existing back office can only support an address in a certain form).

In an effort to support the business context concept, CCTS also defines a "Core Components Context Constraints Language", which is used to express the relationship between specific Business Contexts and how rules are applied to Core Components to produce context specific entities that can be assembled into larger business documents such as "Order" or "Invoice". Constraint Language depends on described "Context Categories" to allow users to uniquely identify and distinguish between different Business Contexts. For each context category, one or more standard classifications are specified to provide values for the category. Constraint rules are tied to values from these particular set of classifications for identifying and distinguishing Contexts.

2.1.3 Universal Business Language (UBL)

A white paper on UBL [27] states that the XML is often described as the common language of e-commerce. The implication is that by standardizing on XML, enterprises will be able to trade with anyone, any time, without the need for the costly custom integration work that has been necessary in the past. But, as the author suggests, this vision of XML-based "plug-and-play" commerce is overly simplistic. XML can be used to create electronic catalogs, purchase orders, invoices, shipping notices, and the other documents needed to conduct business. But XML by itself does not guarantee that these documents can be understood by any business other than the one that creates them. XML is only the foundation on which additional standards can be defined to achieve the goal of true interoperability.

Most large enterprises have already invested significant time and money in an e-business infrastructure and are reluctant to change the way they conduct electronic business. Furthermore, every company has different requirements for the information exchanged in a specific business process, such as procurement or supply-chain optimization. A standard business language must strike a difficult balance, adapting to the specific needs of a given company while remaining general enough to let different companies in different industries communicate with each other.

The UBL standard specification [5] states that while industry-specific data formats have the advantage of maximal optimization for their business context, the existence of different formats to accomplish the same purpose in different business domains is attended by a number of significant disadvantages:

- Developing and maintaining multiple versions of common business documents like purchase orders and invoices is a major duplication of effort.
- Creating and maintaining multiple adapters to enable trading relationships across domain boundaries is an even greater effort.
- The existence of multiple XML formats makes it much harder to integrate XML business messages with back-office systems.
- The need to support an arbitrary number of XML formats makes tools more expensive and trained workers harder to find.

In [27], Gertner argues that lack of a standard for business documents is not due to a shortage of specifications but rather to an overabundance. A multitude of XML business libraries are already in existence. And this has created a big interoperability problem for both users and system vendors. Gertner further states that a company that adopts one of these specifications is likely to find that many of the companies with which it would like to trade are inaccessible to it because they are using incompatible definitions and XML encodings for many of the same ordinary information elements – product and business descriptions, measurements, dates, locations, and so on. Since use of any e-commerce standard requires significant investment, this greatly increases both the cost of integration and the cost of commercial software.

In [28], Holman states that the objective of a standard business language should be to enable interoperability between dissimilar systems using open standards:

- using XML for all the benefits of platform, vendor and application independence,
- using an agreed-upon vocabulary ensures that all users can identify the same information items using the agreed-upon labels,

- the document composed of a known vocabulary can then be understood by different applications on different platforms,

- various tools can be developed to support document creation, vocabulary validation, and formatting.

The UBL standard is intended to address these needs by defining a generic XML format for business documents that can be extended to meet the requirements of particular industries. Specifically, UBL provides a library of XML schemas for reusable data components such as “Address,” “Item,” and “Payment” — the common data elements of everyday business documents and a set of XML schemas for common business documents such as “Order,” “Despatch Advice,” and “Invoice” that are constructed from the UBL library components and can be used in generic procurement and transportation contexts. As stated by [5], the goal in developing these standard business schemas is to provide a universally understood and recognized commercial syntax for business documents and to operate within a standard business framework such as the ebXML to provide a complete, standards-based infrastructure that can extend the benefits of existing EDI systems to businesses of all sizes with the following advantages:

- Lower cost of integration, both among and within enterprises, through the reuse of common data structures.

- Lower cost of commercial software, because software written to process a given XML tag set is much easier to develop than software that can handle an unlimited number of tag sets.

- An easier learning curve, because users need master just a single library.

- Lower cost of entry and therefore quicker adoption by small and medium-size enterprises.

- Standardized training, resulting in many skilled workers.

- A universally available pool of system integrators.

- Standardized, inexpensive data input and output tools.

- A standard target for inexpensive off-the-shelf business software.

The UBL standard [5] sets the scope of the UBL to cover a supply chain from sourcing to payment, including the commercial collaborations of international trade. The diagram in Figure 2-2 illustrates the process context assumed by UBL documents. The specification also underlines that the UBL library is designed to support the construction of a wide variety of document types beyond those provided by the specification package, that is, it is expected that implementers will develop their own customized document types and components and that other UBL document types will be added as the library evolves.

The primary deliverable of UBL is a set of standard formats for common business documents such as invoices, purchases orders, and advance shipment notices. These formats are designed to be sufficient for the needs of many ordinary business transactions and, more importantly, to serve as the starting point for further customization. To enable this customization, the standard document formats are made up of standard "business information entities," which are the common building blocks (addresses, prices, and so on) that make up the bulk of most business documents. Basing all UBL document schemas on the same core information entities maximizes the amount of information that can be shared and reused among companies and applications.

In [27], the author states his vision for a UBL-enabled world, in which companies publish profiles of their requirements for the business documents involved in specific interactions. These profiles specify the business context of each transaction, that is, specific parameters such as the industries and geographic regions of the trading partners. The context parameters are applied to the standard formats to create new formats specific to a given transactional settings.

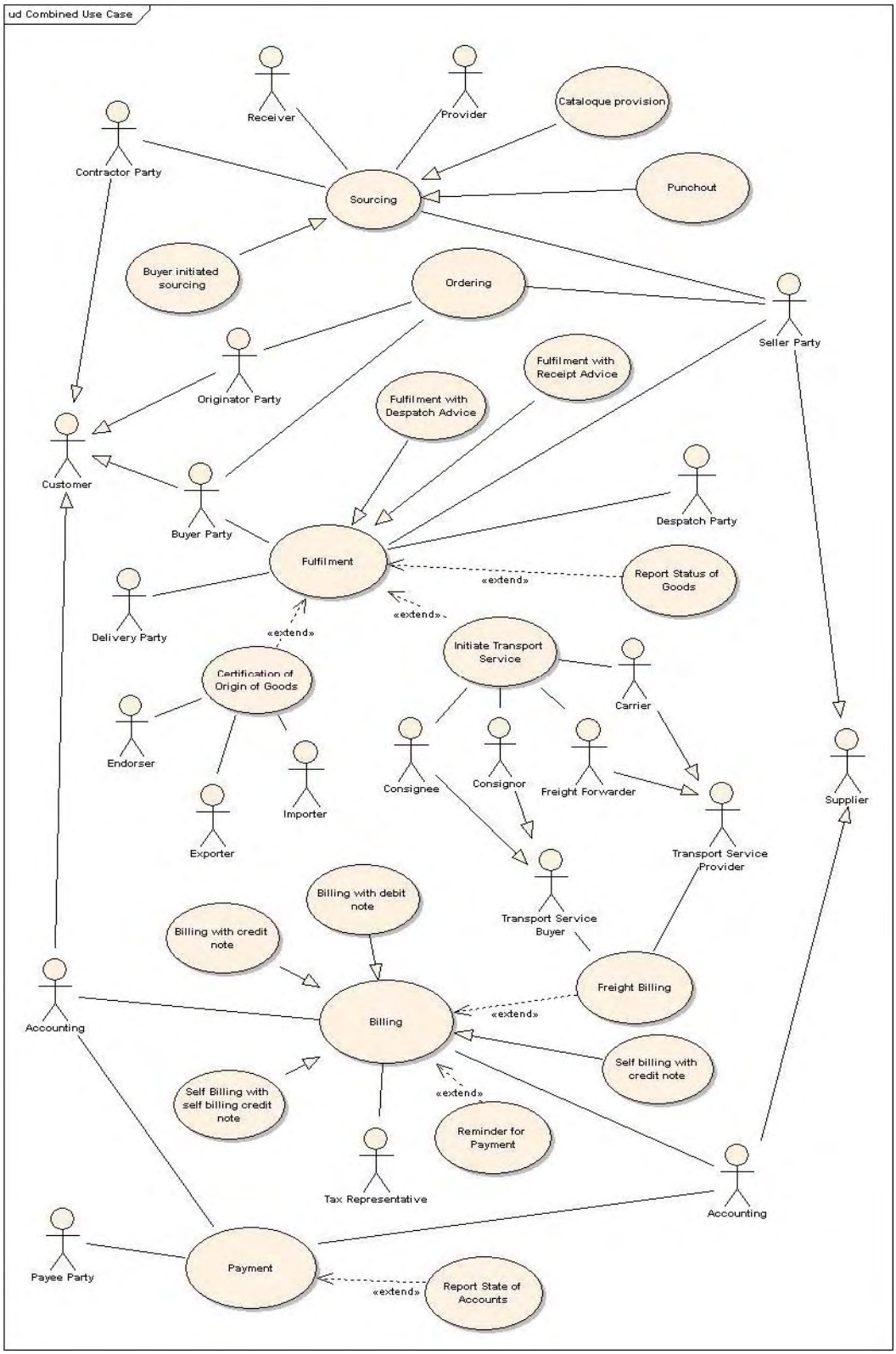

Figure 2-2 - The context for the use of UBL business documents [5]

As an example, a chemical manufacturer might require the specification of hazardous material information when receiving purchase orders from its customers, while an automotive manufacturer buying chemicals might require special satellite positioning (GPS) information in purchase orders to ensure that goods are delivered to exactly the right loading dock. By applying context-specific extensions to the base document formats, a standard format is created that includes fields for both the GPS and the hazardous material related information. The automotive and chemical manufacturers can then trade effectively without the need for long negotiations to settle on document formats that suit them both.

In [29], Gregory et al. summarize the advantages of adapting the UBL approach from an interoperability point of view as follows:

- It is important to understand the vastness of the scale of a global e-commerce standard. There is no centralized control over which version might be supported by existing applications, nor of the mechanisms through which schemas are distributed (aside from making them available). Further, while the mechanism for expressing customizations can be specified, there is no way to control the content of customizations - users will make them to meet their own, non-standard requirements. This means that any built-in mechanism that can assist interoperability in these areas is perhaps more valuable that it might be in more controlled systems.

- It should be recognized that the majority of e-commerce in terms of volume is performed with the exchange of a very small amount of standard information: it is only the abnormal cases that require a large percentage of the constructs reflected in e-commerce schemas, even though their use is occasional. That is to say, in any e-commerce XML vocabulary, there are a lot of optional, needed-but-rarely-used fields allowed in these structures. In the majority of cases, the standard data set is good enough. This is particularly valuable when going across domain boundaries: while it is reasonable to expect trading partners who do business within a single industry domain to support the customization and versions typically used within that domain, this is not the case for many trading partners. For those trading partners whose business is concerned with goods or services that are useful within many different industry

domains, it is not reasonable to expect them to support the customizations and versions across a wide variety of domains. The further down a supply-chain you go, the more likely it is that required goods and services will be generic, rather than specific to that industry supply-chain. A good example of this can be seen in the case of an adhesives manufacturer: their product might be used to glue together airplane seats, car seats, sneakers, and bags for carrying golf-clubs.

- The answer to a lack of version interoperability between trading partners in a non-type-aware scenario is often not to build support for a new version within the processing applications themselves, but to support new interfaces within the gateways, using transformation. This requires development work to identify the mappings between versions, and to implement those mappings. Further, there is the processing cost (and potential for data loss) inherent in performing the transformations themselves. With UBL's type-aware processing (at least for customizations and minor versions), this work is unnecessary.

- As a last consideration, it is reasonable to expect world-wide e-commerce applications as a whole to gradually adopt later - and improved - versions of any standard. But the timing of this adoption will be driven by many factors - supported features, available commercial software, etc. It is reasonable to suppose that only a small number of major versions will be in use at any given point in time. The number of minor versions in use at the same point in time will be far greater. If transformations or other schemes to provide minor-version interoperability are not needed - because type-aware systems make them unnecessary - then the problem of interoperability generally becomes much more tractable.

In [10], Gertner et al. state that one of the most important lessons learned from previous standards is that no business library is sufficient for all purposes; requirements differ significantly amongst companies, industries, countries, and a customization mechanism is therefore needed in many cases before the document types can be used in real-world applications. A primary motivation for moving from the relatively inflexible EDI formats to a more robust XML approach is the possibility of creating formal mechanisms for performing this customization while retaining

maximum interoperability and validation. The UBL acknowledges that the customization of standard schemas will happen and will be done by national and industry groups and smaller user communities; that the changes will be driven by real world needs which will be expressed as context drivers and for that purpose provides a set of guidelines to be followed for the customization of UBL schemas [10]. A brief summary of this UBL Customization Guideline is provided in Appendix A.

2.2 Ontologies

In [30], Genesereth et al. state that a body of formally represented knowledge is based on a conceptualization: the objects, concepts, and other entities that are assumed to exist in some area of interest and the relationships that hold among them. The authors define a conceptualization as an abstract, simplified view of the world that needs to be represented for some purpose. Every knowledge base, knowledge-based system, or knowledge-level agent is committed to some conceptualization, explicitly or implicitly.

Gruber defines an ontology as an explicit specification of a conceptualization [31] and states that for Artificial Intelligent (AI) systems, what "exists" is that which can be represented. When the knowledge of a domain is represented in a declarative formalism, the set of objects that can be represented is called the universe of discourse. This set of objects, and the describable relationships among them, are reflected in the representational vocabulary with which a knowledge-based program represents knowledge. Thus, in the context of AI, the ontology of a program is described by defining a set of representational terms. In such an ontology, definitions associate the names of entities in the universe of discourse (e.g., classes, relations, functions, or other objects) with human-readable text describing what the names mean, and formal axioms that constrain the interpretation and well-formed use of these terms. Formally, an ontology is the statement of a logical theory.

Guarino considers an ontology an *engineering artifact*, constituted by a specific *vocabulary* used to describe a certain reality, plus a set of explicit assumptions regarding the *intended meaning* of the vocabulary words [32] and states that such set of assumptions has usually the form of a first-order logical theory, where vocabulary words appear as unary or binary predicate names, respectively called concepts and relations. In the simplest case, an ontology describes a hierarchy of concepts related by

subsumption relationships; in more sophisticated cases, suitable axioms are added in order to express other relationships between concepts and to constrain their intended interpretation.

Common ontologies are used to describe ontological commitments for a set of agents so that they can communicate about a domain of discourse without necessarily operating on a globally shared theory. An agent is said to commit to an ontology if its observable actions are consistent with the definitions in the ontology. In short, a commitment to a common ontology is a guarantee of consistency with respect to queries and assertions using the vocabulary defined in the ontology.

Noy et al., define the most common uses of ontologies as follows [33]:

- Sharing common understanding of the structure of information among people or software agents is one of the more common goals in developing ontologies. For example, suppose several different Web sites contain medical information or provide medical e-commerce services. If these Web sites share and publish the same underlying ontology of the terms they all use, then computer agents can extract and aggregate information from these different sites. The agents can use this aggregated information to answer user queries or as input data to other applications.

- Enabling reuse of domain knowledge was one of the driving forces behind recent surge in ontology research. For example, models for many different domains need to represent the notion of time. This representation includes the notions of time intervals, points in time, relative measures of time, and so on. If one group of researchers develops such an ontology in detail, others can simply reuse it for their domains. Additionally, if we need to build a large ontology, we can integrate several existing ontologies describing portions of the large domain. We can also reuse a general ontology, such as the UNSPSC ontology, and extend it to describe our domain of interest.

- Making explicit domain assumptions underlying an implementation makes it possible to change these assumptions easily if our knowledge about the domain changes. Hard-coding assumptions about the world in programming-language code make these assumptions not only hard to find and understand but also

hard to change, in particular for someone without programming expertise. In addition, explicit specifications of domain knowledge are useful for new users who must learn what terms in the domain mean.

- Separating the domain knowledge from the operational knowledge is another common use of ontologies. A task can be described to configure a product from its components according to a required specification and implement a program that does this configuration independent of the products and components themselves. Then, an ontology can be developed for PC-components and characteristics and apply the algorithm to configure made-to-order PCs. The same algorithm can also be used to configure elevators if we "feed" an elevator component ontology to it.

- Analyzing domain knowledge is possible once a declarative specification of the terms is available. Formal analysis of terms is extremely valuable when both attempting to reuse existing ontologies and extending them.

As stated in [33], it should be noted that the development of an ontology for a domain is not a goal in itself, but is similar to defining a set of data and their structure for other programs to use. Problem solving methods, domain independent applications and software agents use ontologies and knowledge bases build from ontologies as data.

In [34], Jasper et al. propose a framework for classifying ontology applications and identify the following four main categories:

- Neutral authoring: The basic idea of these applications is to author an artifact in a single language, and to have that artifact translated into a different format for use in multiple target applications. The benefits of this approach include decreased cost of reuse and portability of knowledge across applications, improved application maintainability (because an artifact is only authored in one place, and can be centrally updated) and long term knowledge retention (e.g., via reduced disruption from changes in vendor formats).

- Ontology as a specification: The basic idea of these type of applications is to author an ontology which models the application domain, and provides a vocabulary for specifying the requirements for one or more target applications.

The richer the ontology is in expressing meaning, the less the potential for ambiguity in creating requirements. The software is based on the ontology, which thus plays an important role in the development of the software. The benefits of this approach include documentation, maintenance, reliability and knowledge (re)use.

- Common access to information: The basic idea of these applications is to use ontologies to enable multiple target applications (or humans) to have access to heterogeneous sources of information which is otherwise unintelligible. Benefits of this approach include interoperability, and knowledge reuse. The scenarios in this category differ in a number of ways. First, the direct consumers of the information may be humans or computer applications. Second, the information artifact may play the role of an ontology, or operational data; the latter may be non-computational (e.g., product data) or computational (e.g., services). Another important distinction is whether the target applications agree on the same shared ontology or whether each has its own local ontology. In the former case, the information is made intelligible via translators, and in the latter case, via ontology mapping rules. Finally, access to the information may be via sharing or exchange.

- Ontology based search: The basic idea of these applications is to use an ontology for searching an information repository for desired resources (e.g. documents, web pages, names of experts). The motivation is to improve precision and/or recall as well as reduce the overall amount of time spent searching. The principle actors are knowledge workers (i.e., application users) and ontology authors. Supporting technologies include ontology browsers, search engines, automatic tagging tools, automatic classification of documents, natural language processing, meta-data languages (e.g., XML), natural language ontologies, large general-purpose knowledge bases and thesauri, and knowledge representation and inference systems. In this scenario, an ontology author creates an ontology that assists knowledge workers in identifying concepts that they are interested in. The search engine uses these concepts to locate desired resources from a repository

2.3 Web Ontology Language (OWL)

The Web Ontology Language is an ontology language developed by the World Wide Web Consortium (W3C) Web Ontology Working Group. The overview provided as part of the OWL specification [37] describes a vision for the future of Internet, called the Semantic Web [38] in which information is given explicit meaning, making it easier for machines to automatically process and integrate information available on the Web. Authors further state that the Semantic Web will build on XML's ability to define customized tagging schemes and the flexible approach of the Resource Description Framework (RDF) [39] to represent data. The first level above RDF required for the Semantic Web is an ontology language what can formally describe the meaning of terminology used in Web documents.

In [35], Horrocks et al., claim that in the context of the Semantic Web, ontologies are expected to play an important role in helping automated processes (so called "intelligent agents") to access information. In particular, ontologies are expected to be used to provide structured vocabularies that explicate the relationships between different terms, allowing intelligent agents (and humans) to interpret their meaning flexibly yet unambiguously. For example, a suitable pizza ontology might include the information that Mozzarella and Gorgonzola are kinds of cheese, that cheese is not a kind of meat or fish, and that a vegetarian pizza is one whose toppings do not include any meat or fish. This information allows the term "pizza topped with (only) Mozzarella and Gorgonzola" to be unambiguously interpreted (by, e.g., a pizza ordering agent) as a specialization of the term "vegetarian pizza".

The "OWL Use Cases and Requirements" [36], part of the OWL specification states that ontologies can be used to improve existing Web-based applications, may enable new uses of the Web and provides the following non-exhaustive list of six use cases for the OWL language:

- *Web portals*: In order for a web portal to be successful, it must be a starting place for locating interesting content. Typically, this content is submitted by members of the community, who often index it under some subtopic. A simple index of subject areas may not provide the community with sufficient ability to search for the content that its members require. In order to allow more intelligent syndication, web portals can define an ontology for the community.

This ontology can provide a terminology for describing content and axioms that define terms using other terms from the ontology. For example, an ontology might include terminology such as "journal paper," "publication," "person," and "author." This ontology could include definitions that state things such as "all journal papers are publications" or "the authors of all publications are people." When combined with facts, these definitions allow other facts that are necessarily true to be inferred. These inferences can, in turn, allow users to obtain search results from the portal that are impossible to obtain from conventional retrieval systems.

- *Multimedia collections*: Ontologies can be used to provide semantic annotations for collections of images, audio, or other non-textual objects. It is much difficult for machines to extract meaningful semantics from multimedia. Thus, these types of resources are typically indexed by captions or metatags. However, since different people can describe these non-textual objects in different ways, it is important that the search facilities go beyond simple keyword matching. Ideally, the ontologies would capture additional knowledge about the domain that can be used to improve retrieval of images. Multimedia ontologies can be of two types: media-specific and content-specific. Media specific ontologies could have taxonomies of different media types and describe properties of different media. Content-specific ontologies could describe the subject of the resource, such as the setting or participants. Since such ontologies are not specific to the media, they could be reused by other documents that deal with the same domain. Such reuse would enhance search that was simply looking for information on a particular subject, regardless of the format of the resource.

- *Corporate web site management*: Large corporations typically have numerous web pages concerning things like press releases, product offerings and case studies, corporate procedures, internal product briefings and comparisons, white papers, and process descriptions. Ontologies can be used to index these documents and provide better means of retrieval. Although many large organizations have a taxonomy for organizing their information, this is often insufficient. A single ontology is often limiting because the constituent categories are likely constrained to those representing one view and one

granularity of a domain; the ability to simultaneously work with multiple ontologies would increase the richness of description. Furthermore, the ability to search on values for different parameters is often more useful than a keyword search with taxonomies. An ontology-enabled web site may be used by a salesperson looking for sales collateral relevant to a sales pursuit or a technical person looking for pockets of specific technical expertise and detailed past experience or a project leader looking for past experience and templates.

- *Design documentation*: This use case is for a large body of engineering documentation, such as that used by the aerospace industry. This documentation can be of several different types, including design documentation, manufacturing documentation, and testing documentation. These document sets each have a hierarchical structure, but the structures differ between the sets. There is also a set of implied axes which cross-link the documentation sets: for example, in aerospace design documents, an item such as a wing spar might appear in each. Ontologies can be used to build an information model which allows the exploration of the information space in terms of the items which are represented, the associations between the items, the properties of the items, and the links to documentation which describes and defines them (i.e., the external justification for the existence of the item in the model). That is to say that the ontology and taxonomy are not independent of the physical items they represent, but may be developed/explored in tandem.

- *Agents and services*: The Semantic Web can provide agents with the capability to understand and integrate diverse information resources. A specific example is that of a social activities planner, which can take the preferences of a user (such as what kinds of films they like, what kind of food they like to eat, etc.) and use this information to plan the user's activities for an evening. The task of planning these activities will depend upon the richness of the service environment being offered and the needs of the user. During the service determination / matching process, ratings and review services may also be consulted to find closer matches to user preferences (for example, consulting reviews and rating of films and restaurants to find the "best"). This type of agent requires domain ontologies that represent the terms for restaurants,

hotels, etc. and service ontologies to represent the terms used in the actual services. These ontologies will enable the capture of information necessary for applications to discriminate and balance among user preferences. Such information may be provided by a number of sources, such as portals, service-specific sites, reservation sites and the general Web.

- *Ubiquitous computing*: Ubiquitous computing is an emerging paradigm of personal computing, characterized by the shift from dedicated computing machinery to pervasive computing capabilities embedded in our everyday environments. Characteristic to ubiquitous computing are small, handheld, wireless computing devices. The key issue (and goal) of ubiquitous computing is interoperability under "unchoreographed" conditions, i.e., devices which were not necessarily designed to work together (such as ones built for different purposes, by different manufacturers, at a different time, etc.) should be able to discover each others' functionality and be able to take advantage of it. Being able to "understand" other devices, and reason about their services/functionality is necessary. An ontology language can be used to describe the characteristics of devices, the means of access to such devices, the policy established by the owner for use of a device, and other technical constraints and requirements that affect incorporating a device into a ubiquitous computing network

Horrocks et al. claim that the major extension of OWL over its predecessors is the ability of OWL to provide restrictions on how properties behave that are local to a class [35]. In other words, OWL can define classes where a particular property is restricted so that all the values for the property in instances of the class must belong to a certain class (or data type); at least one value must come from a certain class (or data type); there must be at least certain specific values; and there must be at least or at most a certain number of distinct values.

The OWL language provides three increasingly expressive sublanguages designed for use by specific communities of implementers and users [40]:

- *OWL Lite* which supports those users primarily needing a classification hierarchy and simple constraint features. For example, while OWL Lite supports cardinality constraints, it only permits cardinality values of 0 or 1. It

should be simpler to provide tool support for OWL Lite than its more expressive relatives, and provide a quick migration path for thesauri and other taxonomies.

- *OWL DL* which supports those users who want the maximum expressiveness without losing computational completeness (all entailments are guaranteed to be computed) and decidability (all computations will finish in finite time) of reasoning systems. OWL DL includes all OWL language constructs with restrictions such as type separation (a class can not also be an individual or property; a property can not also be an individual or class). OWL DL is so named due to its correspondence with Description Logics [41], a field of research that has studied a particular decidable fragment of first order logic. OWL DL was designed to support the existing Description Logic business segment and has desirable computational properties for reasoning systems.

- *OWL Full* is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. For example, in OWL Full a class can be treated simultaneously as a collection of individuals and as an individual in its own right. Another significant difference from OWL DL is that a owl:DatatypeProperty can be marked as an owl:InverseFunctionalProperty. OWL Full allows an ontology to augment the meaning of the pre-defined (RDF or OWL) vocabulary. It is unlikely that any reasoning software will be able to support every feature of OWL Full.

Each of these sublanguages is an extension of its simpler predecessor, both in what can be legally expressed and in what can be validly concluded. Ontology developers adopting OWL should consider which sublanguage best suits their needs. The choice between OWL Lite and OWL DL depends on the extent to which users require the more-expressive constructs provided by OWL DL. The choice between OWL DL and OWL Full mainly depends on the extent to which users require the meta-modeling facilities of RDF Schema (e.g. defining classes of classes, or attaching properties to classes). When using OWL Full as compared to OWL DL, reasoning support is less predictable since complete OWL Full implementations do not currently exist.

A brief summary of OWL language constructs is provided in Appendix B for reference.

CHAPTER 3

USE OF ONTOLOGIES FOR SEMANTIC ANNOTATION

This chapter discusses the mechanisms we provide for the semantic annotation of UBL Components, namely the context ontologies, and explains how classes from those ontologies are utilized to automate tasks related to the discovery of components and the customization of document schemas.

The customization mechanism provided by the UBL ensures syntactic interoperability for components customized according to contextual needs. That is, it ensures that customized components do not violate the integrity of UBL schemas and an XSD parser that can interpret standard UBL schemas can also interpret customized UBL schemas. However, it does not address the semantic interoperability; in other words, it does not define necessary mechanisms for discovering and re-using components customized by other users.

Our work builds upon the customization methodology suggested by the UBL and improves it by providing a solution for the semantic interoperability by developing ontologies to represent context domains. Classes from these ontologies are then used for annotating UBL components to express the context value they are applicable for. Expressing context information through a formal ontology language, such as the OWL, provides the following:

- An ontology language is machine processable since it conforms to a formal, well-defined syntax [42]. A description given in an ontology language can be automatically processed to obtain the information it represents. A description in OWL can be parsed into the classes, properties and corresponding values, even when an application knows only the OWL syntax and has no

understanding of a particular domain specific ontology. Furthermore, any program having a prior knowledge of the syntax and semantics of the ontology, can parse the description, extract the represented information and interpret it since the syntax and the semantics are already known by the application using it.

- An ontology describes consensual knowledge, that is, it describes meaning which has been accepted by a group not by a single individual. In other words, it provides a common vocabulary for those who have agreed to use it. Hence when UBL components are annotated with an ontology class, it inherits the well-defined, shared meaning attributed to that class.

- Class-subclass relationships expressed through ontology languages are especially useful for the purposes of representing concepts organized as hierarchies. Considering a component customized for a context class is applicable to hierarchically lower level context classes, the ability to traverse class-subclass relations of ontologies enables the development of automated discovery processes for UBL components and schemas.

- An ontology provides the ability to define relationships among classes, properties and instances which can then be used for reasoning. Especially in the case of independently developed ontologies representing similar concepts, different ontologies may include descriptions for corresponding entities. The reasoning capability is the glue holding different ontologies together for such cases.

The rest of this chapter describes our approach for developing context ontologies, first by providing examples from a specific context domain, namely the *Industrial Classification* context, and then generalizing our ideas to the remaining context domains.

3.1 Ontology Development for the Industrial Classification Context

UBL inherits the business context concept from the ebXML Core Components Specification (CCTS) [24], which defines the *Industrial Classification* context as a means for providing a description of industries or sub-industries in which businesses

takes place. CCTS requires context values for the *Industrial Classification* context to be used from two code lists: the International Standard Industrial Classification (ISIC) [14] and the first two digits of the Universal Standard Product and Service Specification (UNSPSC) [43]. However, in addition to these two, there are other code lists being used by different organizations to classify industrial activities such as the North American Industry Classification System (NAICS) [15], the Classification of Economic Activities in the European Community (NACE) [16].

All these classifications provide detailed taxonomies for industrial activities and disclose considerable amount of semantic information. As such, the taxonomy of these classifications provides context values for UBL components and schemas. However, there is also a need to relate context values from different classifications to each other so that customizations provided for context values specified using a particular classification can be discovered and re-used even when the context is specified using values from different classifications. Furthermore, this semantic interoperability needs to be expressed in a machine processable way so that automated processes can interpret it as well as humans.

In order to fulfill these goals, we develop ontologies to represent the taxonomies of these classifications using a formal ontology language, namely the OWL. Taxonomies reveal limited semantics; however, these semantics proves to be very useful when transformed to ontologies:

- Once taxonomies are expressed through an ontology language like OWL, they become machine processable. Hence, it becomes possible to process class-subclass relationships.

- It becomes possible to formally specify relationships among classes from different ontologies. This, in turn, allows defining machine processable semantic information among different ontologies and relating them.

- Formally expressed relationships in OWL are processed by reasoners to infer additional relationships between ontology classes. Hence, the users continue to use whichever classification they prefer for specifying context values, yet automated processes relate context values from one classification to others.

3.1.1 Developing Individual Ontologies

Within the scope of the work presented in this thesis, specialized converters are developed to derive individual ontologies based on the taxonomy of various classifications. Among those, following three classifications are relevant to the Industrial Classification context:

- NACE
- ISIC
- NAICS

For each of these classifications, the developed converters read and parse the particular classification schemas and generate a corresponding ontology using the OWL language. Figure 3-1 provides an example extract from the developed NAICS ontology and Figure 3-2 provides the corresponding OWL definitions.

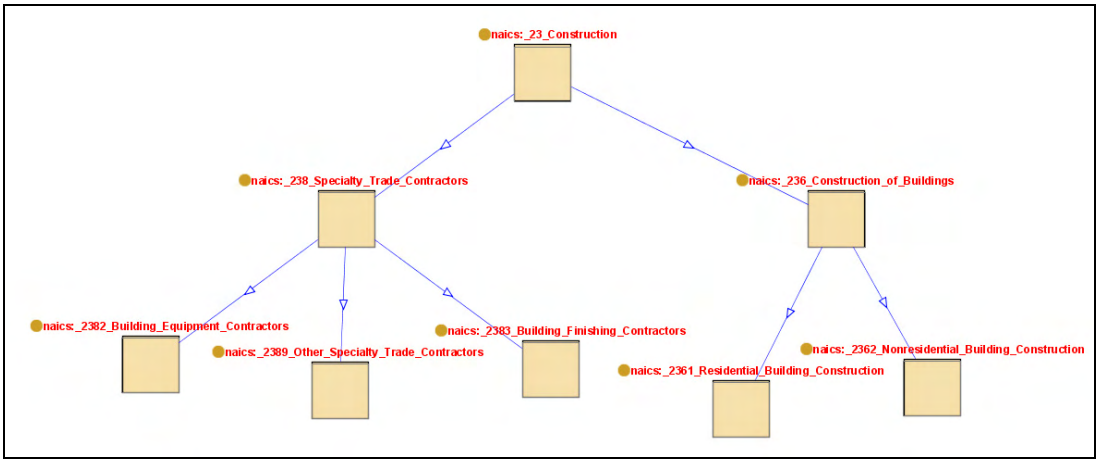

Figure 3-1 – An extract from the NAICS ontology

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns="http://localhost/contextOntology/naics.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#">
  <owl:Ontology rdf:about="NAICS Ontology"/>
  <owl:Class rdf:ID="_2382_Building_Equipment_Contractors">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="_238_Specialty_Trade_Contractors"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="_2362_Nonresidential_Building_Construction">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="_236_Construction_of_Buildings"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="_2383_Building_Finishing_Contractors">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#_238_Specialty_Trade_Contractors"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#_238_Specialty_Trade_Contractors">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="_23_Construction"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:a bout="#_236_Construction_of_Buildings">
    <rdfs:subClassOf rdf:resource="#_23_Construction"/>
  </owl:Class>
  <owl:Class rdf:ID="_2389_Other_Specialty_Trade_Contractors">
    <rdfs:subClassOf rdf:resource="#_238_Specialty_Trade_Contractors"/>
  </owl:Class>
  <owl:Class rdf:ID="_2361_Residential_Building_Construction">
    <rdfs:subClassOf rdf:resource="#_236_Construction_of_Buildings"/>
  </owl:Class>
</rdf:RDF>
```

Figure 3-2 – Corresponding OWL Definition for Figure 3-1

Figure 3-3, Figure 3-4 and Figure 3-5 provide class hierarchies of the developed ontologies representing the NAICS, NACE and ISIC classifications respectively.

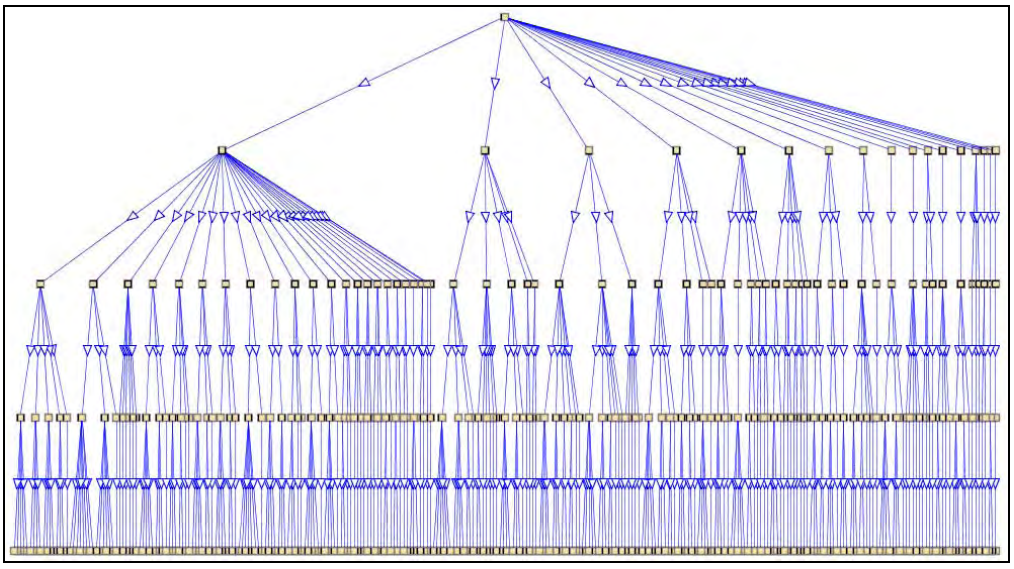

Figure 3-3 - Class Hierarchy of the ISIC ontology

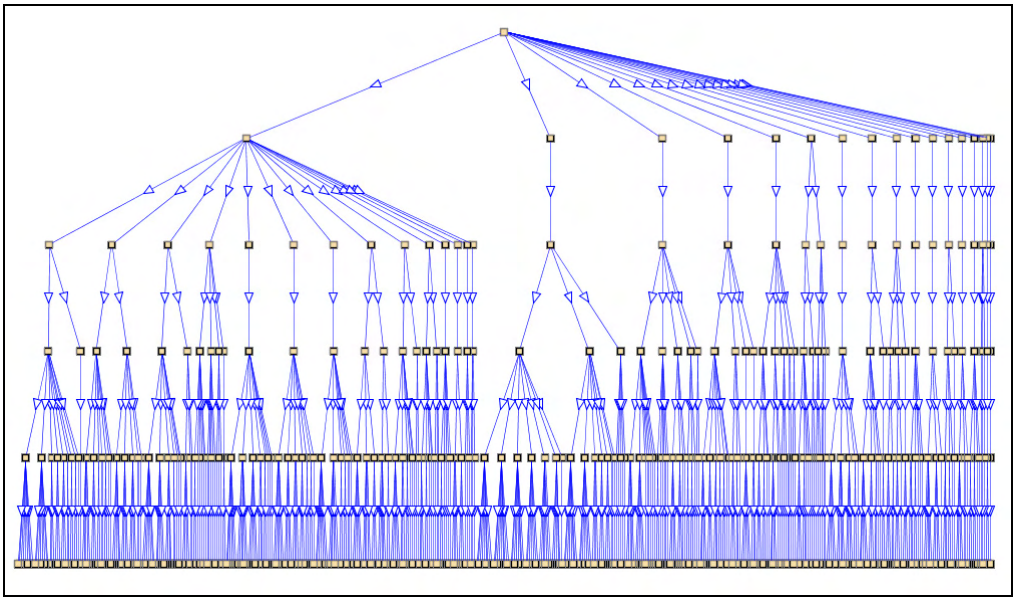

Figure 3-4 - Class Hierarchy of the NACE ontology

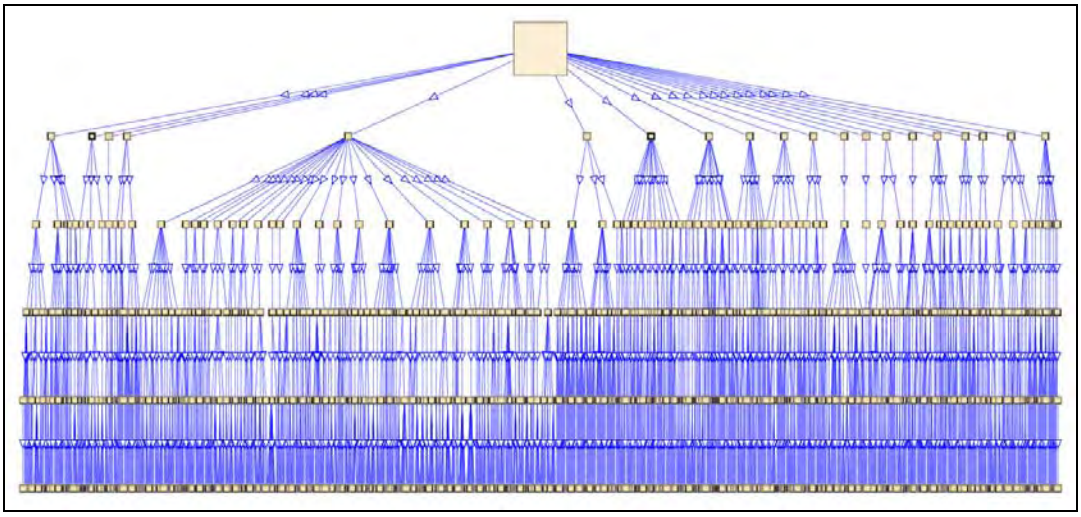

Figure 3-5 - Class Hierarchy of the NAICS ontology

3.1.2 Aligning Individual Ontologies

Once all individual ontologies representing a context domain are developed, it becomes possible to apply ontology reconciliation techniques to specify relationships between classes belonging to different ontologies. This ontology alignment [44][45] task of resolving semantic correspondences between different ontologies can in fact, best be assumed by domain experts and standard issuing bodies. The aim here is not to take on this role but to demonstrate how such correspondences can be exploited once they are specified.

In order to specify correspondences between classes from different ontologies, there is need for a joint ontology including all class definitions and class-subclass relationships from individual ontologies. For this purpose, a unified Context Ontology is generated. This Context Ontology serves two purposes:

- It provides an integrated view of all the ontologies relevant to a context. Instead of duplicating class definitions and class-subclass relationships defined in individual ontologies, Context Ontology includes links to particular ontologies through the `import` construct of OWL. This approach allows for

independent maintenance of individual ontologies since the `import` construct allows resources to be distributed over the Internet.

- It provides a persistent storage for specified ontology alignment expressions. This way, it becomes possible to separate ontology alignment expressions and ontology definitions so that they can be maintained separately.

Figure 3-6 provides an example Context Ontology generated for the *Industrial Classification* context. This example ontology does not contain any ontology alignment expressions but only includes links (import statements) to three individual ontologies, namely to naics.owl, nace.owl and isic.owl.

```
<rdf:RDF
    xmlns="http://www.srdc.metu.edu.tr/IndClsContextOntology.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#">

xmlns:naics="http://www.srdc.metu.edu.tr/contextOntology/naics.owl#"
    xmlns:isic="http://www.srdc.metu.edu.tr/contextOntology/isic.owl#"
    xmlns:nace="http://www.srdc.metu.edu.tr/ctxOntology/nace.owl#">

  <owl:Ontology rdf:about="Industry Context Ontology">

    <owl:imports

rdf:resource="http://www.srdc.metu.edu.tr/ctxOntology/naics.owl"/>

    <owl:imports
      rdf:resource="http://www.srdc.metu.edu.tr/ctxOntology/nace.owl"/>

    <owl:imports
       rdf:resource="http://
www.srdc.metu.edu.tr/ctxOntology/isic.owl"/>
  </owl:Ontology>
```

Figure 3-6 – OWL definition for the Context Ontology of the Industrial Classification Context

Using the unified view provided by context ontologies, it becomes possible for domain experts to specify correspondences between classes from individual ontologies using

one of the various available ontology editors such as the Protégé [46], the Prompt plug-in for Protégé [47], the Swoop [48], and the Chimaera [49].

It should be noted that, since the classification schemas represented by context ontologies are being used for a fair amount of time, a significant amount of such relationships are already known and documented. As an example, [50] provides the correspondences between NACE and ISIC codes. Using such resources together with domain knowledge and expertise, correspondences between classes from different ontologies can be represented in a machine processable way using various OWL constructs including but not limited to the following operations:

- Specifying equivalence: It is very common for multiple classifications to include code for semantically equivalent concepts. As an example, ISIC, NAICS and NACE all specify *Construction* as an industry. For such cases, specification of corresponding classes from different ontologies as equivalent classes is performed using the <owl:equivalentClass> construct.

  Figure 3-7 provides sample OWL definitions that specify an equivalence relationship between the nace:_45_Construction and the naics:_23_ Construction classes.

```
<rdf:Description
  rdf:about=

"http://www.srdc.metu.edu.tr/ctxOntology/nace.owl#_45_Construction">
  <owl:equivalentClass rdf:resource=

"http://www.srdc.metu.edu.tr/ctxOntology/naics.owl#_23_Construction”/>
</rdf:Description>
```

Figure 3-7 – Sample OWL definitions to specify equivalence

- Specifying composition: There are many cases where a concept represented as a single code by one classification corresponds to multiple codes in other classifications. As an example, the NAICS code *11-Agriculture, Forestry, Fishing and Hunting* corresponds to two ISIC codes: *A-Agriculture, Hunting and Forestry* and *B-Fishing*. Such cases are aligned by specifying one class in an ontology as the composition of multiple classes in another ontology by using the <owl:unionOf> construct.

  Figure 3-8 provides sample OWL definitions for specifying a composition relationship between the naics:_11_Agriculture_Forestry_Fishing_and_ Hunting class and the union of isic:_B_Fishing and isic:A_Agriculture_ hunting_and_forestry classes.

```
<rdf:Description rdf:about=
  "http://www.srdc.metu.edu.tr/ctxOntology/naics.owl#
                        _11_Agriculture_Forestry_Fishing_and_Hunting">
    <owl:equivalentClass>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <rdf:Description rdf:about=
          "http://www.srdc.metu.edu.tr/ctxOntology/isic.owl#
                                  _B_Fishing"/>
          <rdf:Description rdf:about=
              "http://www.srdc.metu.edu.tr/ctxOntology/isic.owl#

_A_Agriculture_hunting_and_forestry"/>
        </owl:unionOf>
      </owl:Class>
    </owl:equivalentClass>
  </rdf:Description>
```

Figure 3-8 - Sample OWL definitions to specify composition

- Specifying subsumption: Another common pattern among different classifications is that a code from one classification simply subsumes a code from another classification. As an example, NACE code *CA-Mining and Quarrying of energy producing materials* express a concept that is a superset of the NAICS code *211-Oil and Gas Extraction*. Such cases are aligned by

specifying subsumption relations between classes in different ontologies by using the <owl:subClassOf> construct.

Figure 3-9 provides corresponding OWL definitions for specifying the subsumption relationship between the naics:_211_Oil_and_Gas _Extraction and the nace:CA_Mining_and_quarrying_of_energy_producing_materials classes.

```
<rdf:Description rdf:about=
   "http://www.srdc.metu.edu.tr/ctxOntology/naics.owl#
     _211_Oil_and_Gas_Extraction">
  <rdfs:subClassOf rdf:resource=
     " http://www.srdc.metu.edu.tr/ctxOntology/nace.owl#
         CA_Mining_and_quarrying_of_energy_producing_materials"/>
</rdf:Description>
```

Figure 3-9 - Sample OWL definitions to specify subsumption

Once different ontologies are aligned by specifying all such correspondences, a description logics reasoner computes the inferred class hierarchy for the context domain. This inferred class hierarchy provides a single ontology that represents all individual ontologies and the complete set of relationships among their classes. Figure 3-10 displays the effects of mentioned alignment operations after this reasoning process:

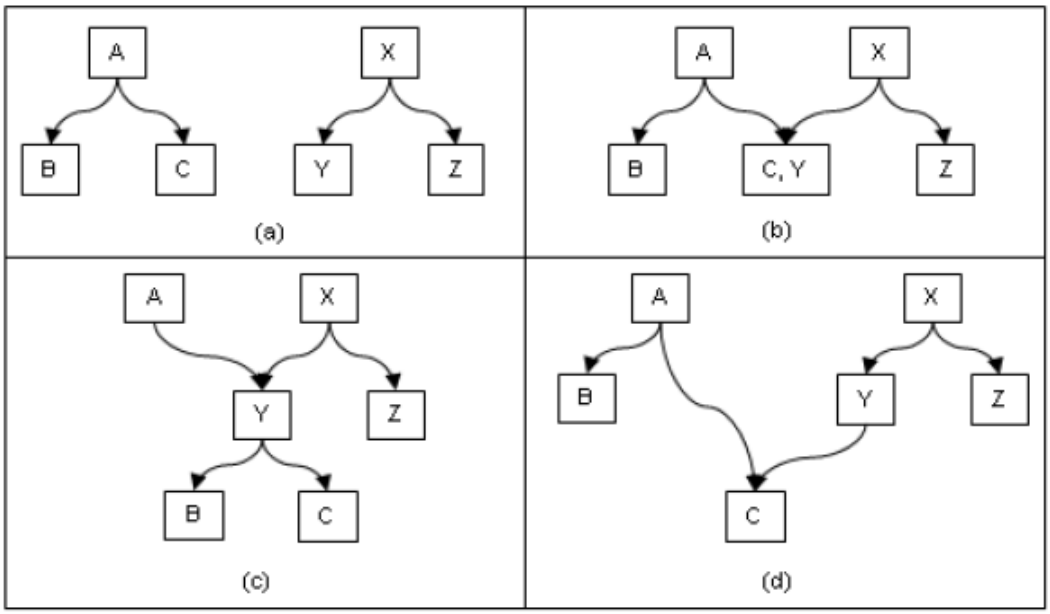

Figure 3-10 - Ontology Alignment Operations

- Figure 3-10 (a) displays sample class hierarchies from two different ontologies.

- Figure 3-10 (b) shows the inferred ontology after (`C ≡ Y`) is expressed through <owl:equivalentClass>. Notice that the axiom provided for alignment cause the reasoner to add additional parents to `C` and `Y`.

- Figure 3-10 (c) gives the inferred ontology after (`(C U B) ≡ Y`) is expressed through <owl:unionOf>. For this case, alignment operation changes direct parents of `B` and `C` and adds a new parent to the `Y`.

- Figure 3-10 (d) depicts the inferred ontology after (`C ⊆ Y`) is expressed through <owl:subClassOf>. This alignment operation results in an additional parent for `C`.

It should be noted that all alignment operations result in classes with multiple inheritance. Since a UBL component customized for a context value is also applicable for context values at hierarchically lower levels, multiple inheritance has significant consequences for the purposes of component discovery. While searching for component versions applicable for a context value, component discovery considers versions applicable to all parents and merges them.

Once the inferred ontology representing the *Industrial Classification* domain is computed, classes from that ontology are used for annotating customized UBL components to express their context, and it becomes possible to develop automated processes that can intelligently discover UBL components by utilizing context ontologies and corresponding annotations.

As an example, consider the Context Ontology fragment in Figure 3-11 in which solid lines represent class-subclass relationships inherited from classifications represented by individual ontologies and the dashed line represents an equivalence relationships specified by a domain expert as part of the ontology alignment process.

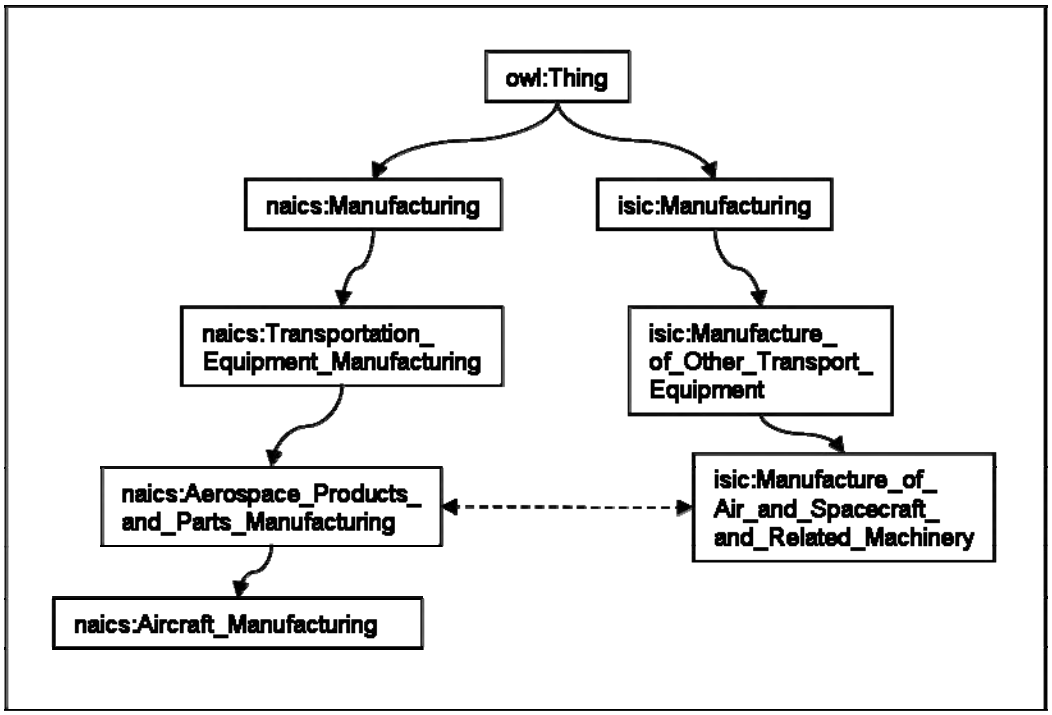

Figure 3-11 – Class hierarchy fragments from a sample Context Ontology

The case of the Context Ontology fragment in Figure 3-11 is similar to the case of Figure 3-10-(b). Consequently, when it is processed by a description logic reasoner, the inferred Context Ontology in Figure 3-12 is computed, in which the equivalent "naics:Aerospace_Products_and_Parts_Manufacturing" and the "isic:Manufacture_ of_Air_and_Spacecraft_and_Related_Machinery" classes are treated like a single class and the combined class inherits the parent and the child classes of its elements.

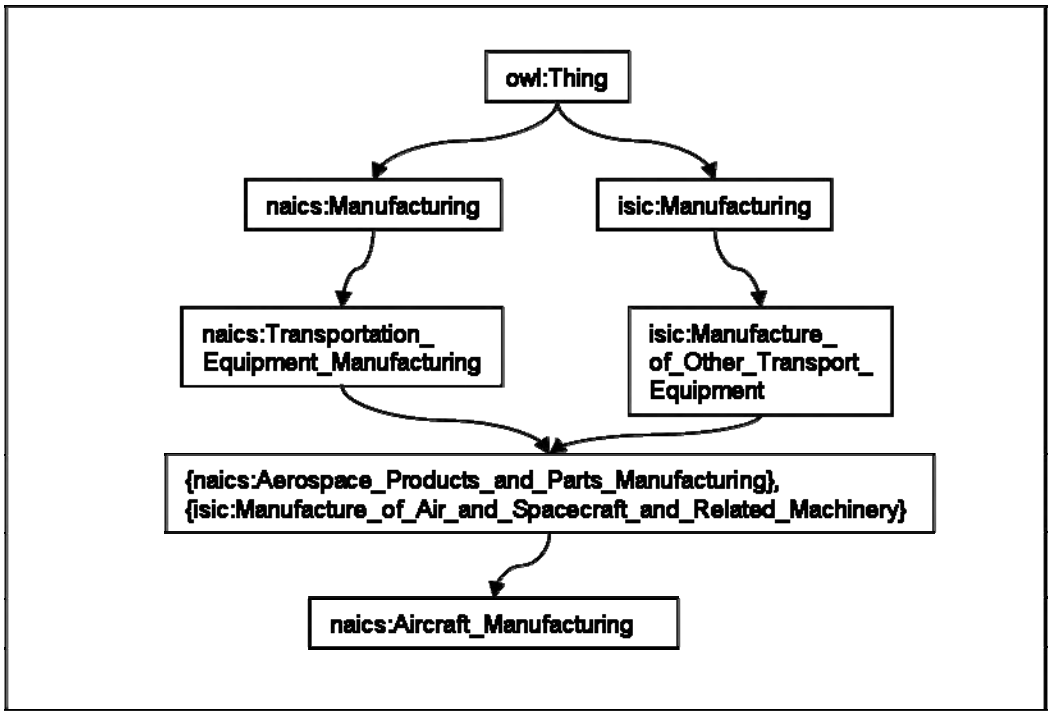

Figure 3-12 – Inferred Context Ontology corresponding to the Context Ontology fragment in Figure 3-11

Following are two example scenarios that demonstrate how the inferred Context Ontology in Figure 3-12 is utilized by an automated process for the purposes of handling requests about gathering component versions applicable to various context values:

- Assume standard UBL Party component is customized for the *naics:Manufacturing* context with the addition of the Manufacturing License component and annotated accordingly as denoted $Party_M$ in Figure 3-13. When an aircraft parts manufacturing company requires the applicable version of the Party component, an automated process through interpreting class-subclass relations defined between *naics:Manufacturing*, *naics:Transport Equipment Manufacturing* and *naics:Aerospace Product and Parts Manufacturing* classes discovers that the version of the Party component customized for the *naics:Manufacturing* context is the one to use rather than the standard UBL

version since the applicable context *naics:Aerospace Product and Parts Manufacturing* is a sub context of the *naics:Manufacturing* context.

- Assume the International Civil Aviation Organization (ICAO) approval is required for all aerospace manufacturers hence the version of the Party component customized for the *naics:Manufacturing* context is further extended with the addition of the ICAOApproval component and annotated with the *isic:Manufacture of Air and Spacecraft and Related Machinery* context, denoted as $Party_{MAS}$ in Figure 3-13. When users accustomed to NAICS classification request components applicable to the *naics:Aircraft Manufacturing* context, through the class-subclass relation between the *naics:Aircraft Manufacturing* and the *isic:Manufacture of Air and Spacecraft and Related Machinery* (which is inferred as a result of the asserted equivalence between the *naics:Aerospace Products and Parts Manufacturing* and the *isic:Manufacture of Air and Spacecraft and Related Machinery* classes), an automated process can discover that the version of the PartyType customized for the *isic:Manufacture of Air and Spacecraft and Related Machinery* context is also applicable to the *naics:Aircraft Manufacturing* context.

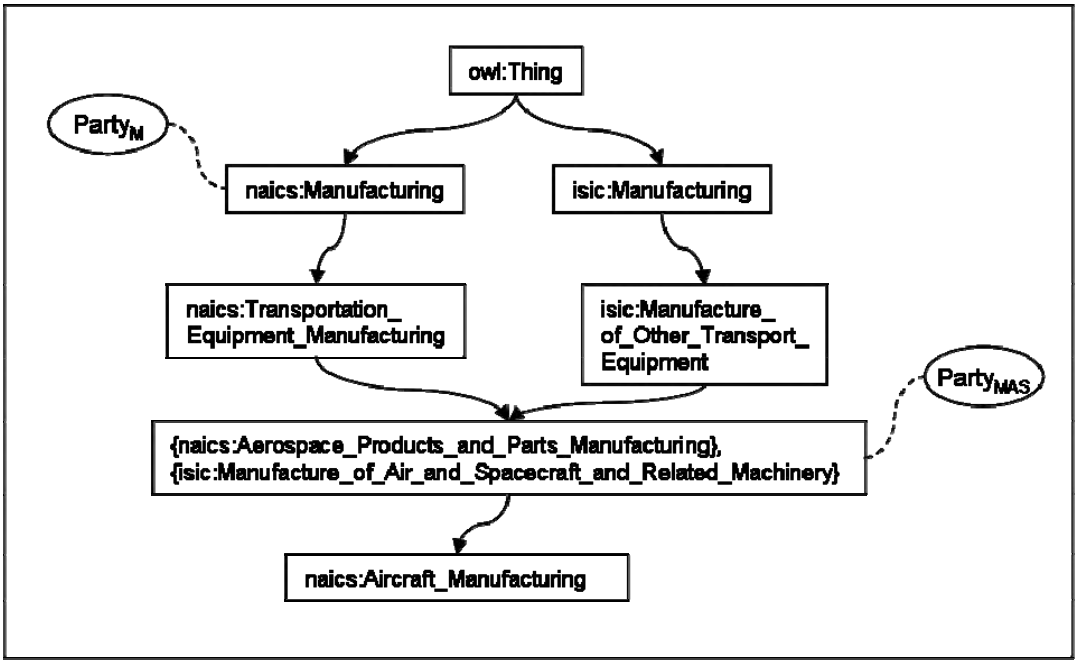

Figure 3-13 – Inferred Context Ontology in Figure 3-12 with Component Annotations

3.2 Generalization to Other Context Domains

The essence of our approach is to provide an open mechanism that is flexible enough to support all classifications applicable for a context, so that users can continue to work with the classifications they are accustomed to. In the meantime, these classifications need to be interpretable by automated processes, therefore individual ontologies are used to reflect the taxonomy of classifications and are related to one another through ontology alignment operations described above.

As in the case of the Industrial Classification context, there are existing classifications related to other context domains as well. For the *Product Classification* context, examples include the Universal Standard Products and Services Classification (UNSPSC) [43], the North American Product Classification System (NAPCS) [51], the Classification of Products by Activity in EU (CPA) [52], Central Product Classification (CPC) [53]. For the *Geopolitical* context, CCTS suggests a four leveled hierarchy consisting of Continent, Economic Region, Country (using ISO 3166.1 codes [54]) and

Region (using ISO 3166.2 codes [54]). For the *Business Process* context, CCTS suggests the use of UN/CEFACT Catalogue of Common Business Processes [26]. In addition to these, it is reasonable to assume the introduction of new classification schemes with time. Similarly, user groups and organizations may suggest different structures for representing domain values.

Following the approach provided in this chapter, taxonomies corresponding to business context domains can be expressed using ontologies and those ontologies can be related to each other using ontology alignment techniques. This way, it becomes possible to express all business context domains in a machine processable manner.

CHAPTER 4

USE OF SEMANTIC CONSTRUCTS FOR INTEROPERABILITY

UBL is being adapted by several communities around the world. Examples include Denmark, where UBL Invoice has been mandated by law for all public-sector businesses [7]; Sweden, where the National Financial Management Authority recommended UBL Invoice for all government use [8], US Department of Transportation which is developing a UBL based pilot project for a demonstration of state-of-the-art electronic commerce in a real-world setting [9]. Denmark and Sweden define custom subsets of UBL by layering on business rules implemented in Schematron [55], a small language for making assertions about the presence or absence of patterns in XML documents.

However, since businesses operate in different industry, geopolitical, regulatory contexts and they have different rules and requirements for the information they exchange, sub setting does not always serve the needs especially in the case of small and medium enterprises. In other words, with the increasing popularity of UBL, it is reasonable to expect that user communities of any size would wish to customize the UBL schemas according to their needs. However, as communities prefer using non-standard schemas, it becomes harder to maintain the interoperability within the UBL community.

In order to provide a solution to this interoperability problem, we provide a semantic translation mechanism that converts documents between schemas customized for different business contexts. Such a translation mechanism supports user communities adapting schema versions that better suits the specific requirements of their businesses. Parties within a community use the same version and can communicate with each other

seamlessly. When parties from different communities need to make business with each other, they still use their schemas and the interoperability is provided by translating documents from one context to another.

4.1 Web Ontology Language to Describe UBL Components and Schemas

The first step to develop such a translation capability is to provide a machine processable mechanism that can express the structure and the semantics of components together with their correspondences in different versions. Considering the complexity of UBL schemas and the physically distributed nature of communities, it is not reasonable to expect manual specification and maintenance of all relationships among practically unlimited number of customized schemas.

As a solution, we developed a "Component Ontology" for UBL that provides a formal representation of components using the OWL language. This eliminates the maintenance burden as it only requires the specification of explicit relationships which can then be used for discovering implicit relationships through reasoning. The Component Ontology serves two major purposes:

- Representing the semantics of UBL components: UBL customization takes place at the level of individual types and elements; hence, translation needs to be done at the same level. When an automated process compares two versions of a schema, it needs to be able to identify corresponding elements in these schemas. When UBL elements are represented as classes of a common component ontology, it becomes possible to utilize that ontology for the computation of similarities between elements from different schemas.

- Representing the structure of UBL schemas: UBL document schemas are complex hierarchies including numerous types and elements any of which might be modified through customization. Even in the case of semantically equivalent components, comparison is a complicated task unless supported by proper tools. By representing the structural layout of types and elements using classes from common component ontology, it becomes possible to utilize semantic reasoners for the comparison of these complex hierarchies.

The Component Ontology consists of classes (i.e. concepts in description logics) representing UBL constructs such as type definitions, element declarations and business concepts and object-type properties (i.e. roles in description logics) that represent various relationships between these constructs. Classes are defined based on their relationship to other classes (through axioms asserting existential restrictions over object-type properties). When reasoners interpret these axioms, they compute equivalence and/or subsumption relationships between classes that are defined through similar restrictions.

4.2 UBL Component Ontology

UBL provides several document schemas and a library of components consisting of elements and type definitions that are used in document schemas. Document schemas are collections of basic and aggregate components which recursively contain other components. This section introduces the Component Ontology developed to represent UBL components.

In order to be able to describe Component Ontology concepts clearer, they are presented through examples. As such, consider the simplified `Order` schema in Figure 4-1 and corresponding XSD definitions in Figure 4-2.

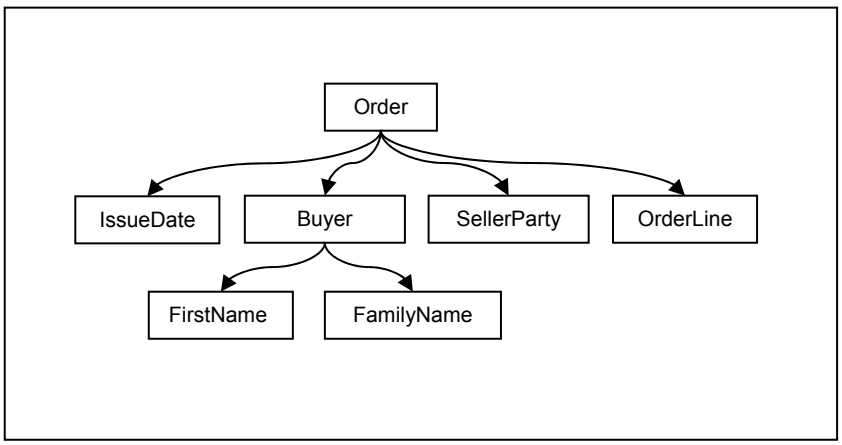

Figure 4-1 - Simplified Order Schema

```
<xsd:element name="Order" type="OrderType" />

<xsd:complexType name="OrderType">
   <xsd:sequence>
      <xsd:element ref="IssueDate" />
      <xsd:element ref="Buyer" />
      <xsd:element ref="SellerParty" />
      <xsd:element ref="OrderLine" />
   </xsd:sequence>
</xsd:complexType>

<xsd:element ref="IssueDate" type="DateType" />
<xsd:element ref="Buyer" type="PersonType" />
<xsd:element ref="SellerParty" type="PartyType" />
<xsd:element ref="OrderLine" type="OrderLineType" />

<xsd:complexType name="PersonType">
  <xsd:sequence>
     <xsd:element ref="FirstName" />
     <xsd:element ref="FamilyName" />
  </xsd:sequence>
</xsd:complexType>

<xsd:element ref="FirstName" type="FirstNameType" />
<xsd:element ref="FamilyName" type="FamilyNameType" />
```

Figure 4-2 – XSD Definitions corresponding to the `Order` Schema in Figure 4-1

In order to represent relationships between entities, types and business concepts, the Component Ontology template shown in Figure 4-3 is developed. DataType, TypeDefinition, ElementDeclaration and Concept are root classes of the Component Ontology. For every type definition, a corresponding TypeDefinition subclass is defined in the Component Ontology. Similarly, for every element declaration, a corresponding ElementDeclaration and a corresponding Concept subclass are defined.

Concept classes represent business concepts and entities represented by the UBL elements. There are no redundancies among elements provided by the UBL and the reuse of existing elements is mandated whenever possible. However, as mentioned earlier, implementations prefer to define their own elements at the expense of redundancy. To be able to avoid semantic redundancy in schemas while giving the implementers the freedom to add elements as they need, the Concept class is defined in our template and a corresponding subclass is added for each unique business concept

and/or entity represented by elements. This allows the definition of multiple elements representing the same business concept/entity and their correspondence is expressed through their relation to the same Concept class. Figure 4-4 provides corresponding OWL definitions for the root classes and properties of the proposed UBL Component Ontology.

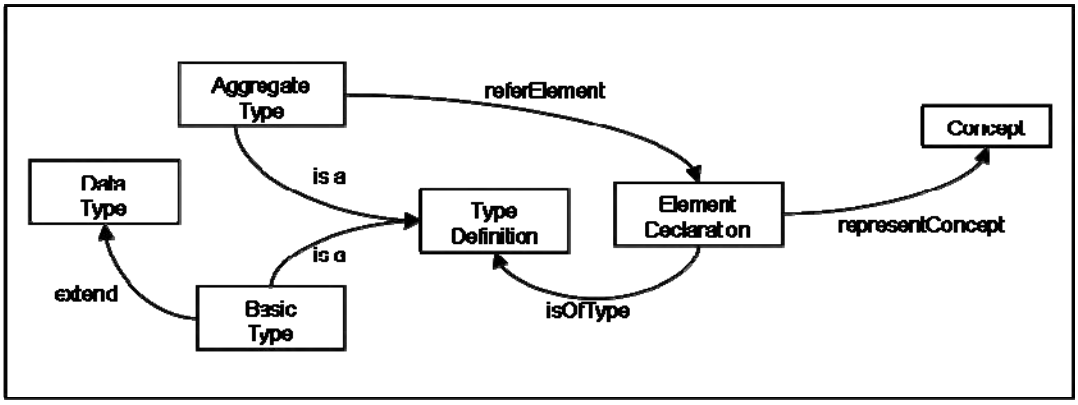

Figure 4-3 - Component Ontology Template

```
<rdf:RDF
      xml:base="http://www.srdc.metu.edu.tr/ublOntology.owl"
      xmlns="http://www.srdc.metu.edu.tr/ublOntology.owl#"
      xmlns:owl="http://www.w3.org/2002/07/owl#"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xmlns:xsd=http://www.w3.org/2001/XMLSchema#>

  <owl:Ontology rdf:about="UBL Component Ontology"/>

  <owl:Class rdf:ID="ElementDeclaration"/>
  <owl:Class rdf:ID="Concept"/>
  <owl:Class rdf:ID="TypeDefinition"/>

  <owl:Class rdf:ID="BasicDataType" >
    <rdfs:subClassOf>
       <owl:Class rdf:resource="#TypeDefinition" />
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:ID="AggregateType" >
    <rdfs:subClassOf>
       <owl:Class rdf:resource="#TypeDefinition" />
    </rdfs:subClassOf>
  </owl:Class>

  <owl:ObjectProperty rdf:ID="referElement">
    <rdfs:domain rdf:resource="#TypeDefinition"/>
    <rdfs:range rdf:resource="#ElementDeclaration"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="isOfType">
    <rdfs:domain rdf:resource="#ElementDeclaration"/>
    <rdfs:range rdf:resource="#TypeDefinition"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="representConcept">
    <rdfs:domain rdf:resource="#ElementDeclaration"/>
    <rdfs:range rdf:resource="#Concept"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="extendBasicType">
    <rdfs:domain rdf:resource="#TypeDefinition"/>
    <rdfs:range rdf:resource="#BasicDataType"/>
  </owl:ObjectProperty>

</rdf:RDF>
```

Figure 4-4 - OWL Definitions corresponding to the UBL Ontology Template in Figure 4-3

Figure 4-5, Figure 4-6 and Figure 4-7 provide respective OWL definitions for type, element declaration and concept classes of the Order schema in Figure 4-1. As described earlier, these classes are straightforward subclasses of corresponding root classes of the UBL Component Ontology.

```
<owl:Class rdf:about="OrderType">
  <rdfs:subClassOf rdf:resource="#AggregateType" />
</owl:Class>
<owl:Class rdf:about="DateType">
  <rdfs:subClassOf rdf:resource="#AggregateType" />
</owl:Class>
<owl:Class rdf:about="PersonType">
  <rdfs:subClassOf rdf:resource="#AggregateType" />
</owl:Class>
<owl:Class rdf:about="PartyType">
  <rdfs:subClassOf rdf:resource="#AggregateType" />
</owl:Class>
<owl:Class rdf:about="OrderLineType">
  <rdfs:subClassOf rdf:resource="#AggregateType" />
</owl:Class>
<owl:Class rdf:about="FirstNameType">
  <rdfs:subClassOf rdf:resource="#BasicDataType" />
</owl:Class>
<owl:Class rdf:about="FamilyNameType">
  <rdfs:subClassOf rdf:resource="#BasicDataType" />
</owl:Class>
```

Figure 4-5 – OWL definitions for type classes of Order schema in Figure 4-1

```
<owl:Class rdf:about="Order">
  <rdfs:subClassOf rdf:resource="#ElementDeclaration" />
</owl:Class>
<owl:Class rdf:about="IssueDate">
  <rdfs:subClassOf rdf:resource="#ElementDeclaration" />
</owl:Class>
<owl:Class rdf:about="Buyer">
  <rdfs:subClassOf rdf:resource="#ElementDeclaration" />
</owl:Class>
<owl:Class rdf:about="Seller">
  <rdfs:subClassOf rdf:resource="#ElementDeclaration" />
</owl:Class>
<owl:Class rdf:about="OrderLine">
  <rdfs:subClassOf rdf:resource="#ElementDeclaration" />
</owl:Class>
<owl:Class rdf:about="FirstName">
  <rdfs:subClassOf rdf:resource="#ElementDeclaration" />
</owl:Class>
<owl:Class rdf:about="FamilyName">
  <rdfs:subClassOf rdf:resource="#ElementDeclaration" />
</owl:Class>
```

Figure 4-6 - OWL definitions for element classes of Order schema in Figure 4-1

```
<owl:Class rdf:about="OrderConcept">
  <rdfs:subClassOf rdf:resource="#Concept" />
</owl:Class>
<owl:Class rdf:about="IssueDateConcept">
  <rdfs:subClassOf rdf:resource="#Concept" />
</owl:Class>
<owl:Class rdf:about="BuyerConcept">
  <rdfs:subClassOf rdf:resource="#Concept" />
</owl:Class>
<owl:Class rdf:about="SellerConcept">
  <rdfs:subClassOf rdf:resource="#Concept" />
</owl:Class>
<owl:Class rdf:about="OrderLineConcept">
  <rdfs:subClassOf rdf:resource="#Concept" />
</owl:Class>
<owl:Class rdf:about="FirstNameConcept">
  <rdfs:subClassOf rdf:resource="#Concept" />
</owl:Class>
<owl:Class rdf:about="FamilyNameConcept">
  <rdfs:subClassOf rdf:resource="#Concept" />
</owl:Class>
```

Figure 4-7 - OWL definitions for concept classes of Order schema in Figure 4-1

In order to relate classes of the UBL Component Ontology to each other and represent the set of correspondences among them, following object properties are defined:

- Basic UBL types are defined through extending simple data types such as text, integer, and date. There may be multiple basic types extending the same data type, such as FirstNameType, FamilyNameType and AddressLineType all being defined through extending the TextType. In order to represent the relationship between classes representing basic types defined by extending the same data type, the extend object property is used.

- UBL defines aggregate types as collections of element declarations. In order to represent the relationship between classes representing aggregate types that refer to a similar set of elements, the referElement object property is used.

- Every UBL element has a type and the isOfType object property is used to represent the relationship between classes representing type definitions and element declarations.

- Definition of multiple elements that represent identical business concepts is allowed and element declaration classes are related to corresponding business concept classes through the representConcept object property.

These properties express the defining characteristics for classes of the UBL Component Ontology. That is, based on the set of properties and values of those properties, classes can be distinguished from and/or related to each other.

In order to be able to distinguish classes from each other and to be able to discover the similarities between classes, existential restrictions are defined over object properties to specify the set of relationships distinguishing that particular class. This allows the utilization of reasoners to classify classes that are described (restricted) through similar relationships. In the following, different types of existential restrictions are introduced:

- Two basic types that are derived from the same data type are eligible for translation to each other. In order to represent the relationship between classes representing such types, following existential restrictions are defined for the `extend` property between BasicType classes and corresponding DataType classes:

```
aType ≡ (BasicType ∩ (∃extend. aDataType))
```

Figure 4-8 provides an example of how this generic description logic (DL) axiom can concretely be expressed in OWL syntax for the FamilyNameType in Figure 4-2.

```
<owl:Class rdf:about="FamilyNameType">
 <owl:equivalentClass>
    <owl:Restriction>
       <owl:onProperty>
         <owl:FunctionalProperty rdf:ID="extendBasicType" />
       </owl:onProperty>
       <owl:someValuesFrom rdf:resource=udt:NameType />
    </owl:Restriction>
 </owl:equivalentClass>
 <rdfs:subClassOf rdf:resource="#TypeDefinition" />
</owl:Class>
```

Figure 4-8 - OWL Definitions corresponding to the FamilyNameType in Figure 4-2

When such expressions are defined for all BasicType classes, reasoners classify those that have `extend` relation with the same DataType class to be equivalent to each other.

- Two aggregate types that refer to the same set of elements are eligible for translation to each other. In order to be able to compute the relationship between classes representing such types, following existential restrictions are defined over referElement properties between AggregateType classes and ElementDeclaration classes:

```
aType ≡ (AggregateType ∩
        (∃referElement. (anElement1 ∩ ... ∩ anElementn)))
```

Figure 4-9 provides an example of how this generic DL axiom can concretely be expressed in OWL syntax for the OrderType in Figure 4-2.

```
<owl:Class rdf:about="OrderType">
 <owl:equivalentClass>
    <owl:Class>
       <owl:intersectionOf rdf:parseType="Collection">
         <owl:Restriction>
           <owl:someValuesFrom>
            <owl:Class>
               <owl:intersectionOf rdf:parseType="Collection">
                  <owl:Class rdf:about="IssueDate" />
                  <owl:Class rdf:about="Buyer" />
                  <owl:Class rdf:about="SellerParty" />
                  <owl:Class rdf:about="OrderLine" />
               </owl:intersectionOf>
            </owl:Class>
           </owl:someValuesFrom>
           <owl:onProperty>
              <owl:ObjectProperty rdf:about="#referElement" />
           </owl:onProperty>
         </owl:Restriction>
         <owl:Class rdf:about="#AggregateType" />
       </owl:intersectionOf>
    </owl:Class>
 </owl:equivalentClass>
</owl:Class>
```

Figure 4-9 – OWL Definitions corresponding to the OrderType in Figure 4-2

When such expressions are defined for all AggregateType classes, reasoners classify those that have referElement relation with the same set of ElementDeclaration classes to be equivalent to each other.

- Every element declaration in UBL represents a business concept. The type of the element defines how the content of the element is structured. Hence, elements that represent the same business concept and have the same structure are eligible for translation to each other. To be able to compute the relationship between classes representing such elements, following existential restrictions are defined for ElementDeclaration classes:

```
anElement ≡ (ElementDeclaration ∩
            (∃representConcept. aConcept) ∩
            (∃isOfType. aType))
```

Figure 4-10 provides an example of how this generic DL axiom can concretely be expressed in OWL syntax for the Order in Figure 4-2.

```
<owl:Class rdf:about="Order">
 <owl:equivalentClass>
    <owl:Class>
       <owl:intersectionOf rdf:parseType="Collection">
        <owl:Restriction>
           <owl:someValuesFrom rdf:resource="#OrderConcept" />
           <owl:onProperty>
            <owl:ObjectProperty rdf:about="#representConcept" />
           </owl:onProperty>
        </owl:Restriction>
        <owl:Restriction>
           <owl:someValuesFrom rdf:resource="OrderType" />
           <owl:onProperty>
            <owl:ObjectProperty rdf:ID="isOfType" />
           </owl:onProperty>
        </owl:Restriction>
        <owl:Class rdf:about="#ElementDeclaration" />
       </owl:intersectionOf>
    </owl:Class>
 </owl:equivalentClass>
</owl:Class>
```

Figure 4-10 - OWL Definitions corresponding to the Order in Figure 4-2

When such expressions are defined for all ElementDeclaration classes, reasoners classify those that have representConcept and isOfType relation with the same Concept and TypeDefinition classes to be equivalent to each other.

4.3 Computing Translations through Reasoning

Even though individual expressions introduced in Section 4.2 are trivial, their expressiveness increases as they are used together. Relationships between basic constructs trigger the computation of relationships between higher level constructs which recursively trigger the computation of additional relationships.

As an example, consider the CustomOrder schema in Figure 4-11, a possible customization of the Order schema in Figure 4-1, where Customer, Name and Surname components replace Buyer, FirstName and FamilyName respectively. Assume that Customer is defined such that it has a representConcept relationship with the same Concept class as Buyer. Similarly, Name and Surname are defined such that they have representConcept and isOfType relationships with same Concept and TypeDefinition classes as FirstName and FamilyName.

Figure 4-12 provides DL axioms defining the Order schema in Figure 4-2 and Figure 4-13 provides DL axioms defining the CustomOrder schema in Figure 4-11. It should be noted that even though the Component Ontology is defined in OWL, the examples throughout the document are provided using corresponding DL axioms to provide better readability.

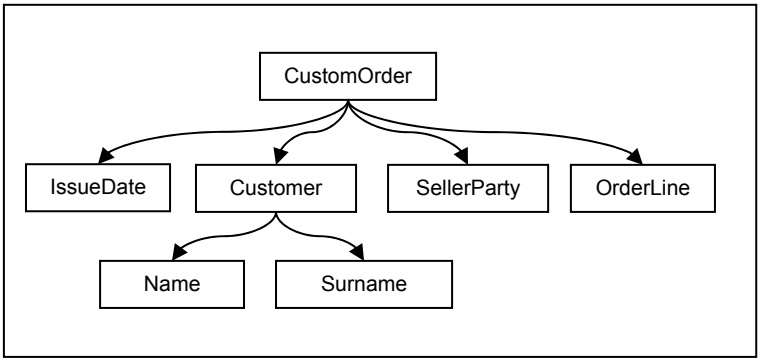

Figure 4-11 – An example custom version of the Order schema in Figure 4-1

```
1. Order ≡           (ElementDeclaration ∩
                     (∃representConcept. OrderConcept) ∩
                     (∃isOfType. OrderType))

2. OrderType ≡       (AggregateType ∩ (∃referElement.
                     (IssueDate ∩ Buyer ∩ SellerParty ∩ OrderLine)))

3. Buyer ≡           (ElementDeclaration ∩
                     (∃representConcept. BuyerConcept) ∩
                     (∃isOfType. PersonType))

4. PersonType ≡      (AggregateType ∩ (∃referElement.
                     (FirstName ∩ FamilyName)))

5. FirstName ≡       (ElementDeclaration ∩
                     (∃representConcept. FirstNameConcept) ∩
                     (∃isOfType. FirstNameType))

6. FirstNameType ≡   (BasicType ∩ (∃extend. TextType))

7. FamilyName ≡      (ElementDeclaration ∩
                     (∃representConcept. FamilyNameConcept) ∩
                     (∃isOfType. FamilyNameType))

8. FamilyNameType ≡ (BasicType ∩ (∃extend. TextType))
```

Figure 4-12 – DL axioms corresponding to the Order schema in Figure 4-2

```
9. CustomOrder ≡          (ElementDeclaration ∩
                          (∃representConcept. OrderConcept) ∩
                          (∃isOfType. CustomOrderType))

10. CustomOrderType ≡     (AggregateType ∩ (∃referElement.
                          (IssueDate ∩Customer ∩SellerParty ∩OrderLine)))

11. Customer ≡            (ElementDeclaration ∩
                          (∃representConcept. BuyerConcept) ∩
                          (∃isOfType. CustomPersonType))

12. CustomPersonType ≡    (AggregateType ∩
                          (∃referElement.(Name ∩ Surname)))

13. Name ≡                (ElementDeclaration ∩
                          (∃representConcept. FirstNameConcept) ∩
                          (∃isOfType. FirstNameType))

14. Surname ≡             (ElementDeclaration ∩
                          (∃representConcept. FamilyNameConcept) ∩
                          (∃isOfType. FamilyNameType))
```

Figure 4-13 - DL axioms corresponding to the CustomOrder schema in Figure 4-11

The rest of this section demonstrates how individual DL axioms cause the computation of equivalances between similarly defined classes based on the following Lemmas. A brief summary of DL rules supporting those Lemmas are provided in Appendix C for reference.

Lemma 1: Two DL concepts C1 and C2 can be considered as equivalent to each other if and only if C1 is subsumed by C2 and C2 is subsumed by C1, that is:

$$(C1 \subseteq C2) \cap (C2 \subseteq C1) \Leftrightarrow C1 \equiv C2$$

Lemma 2: The axiom $(C1 \subseteq C2)$ is satisfiable if and only if the axiom $(C1 \cap \neg C2)$ is unsatisfiable.

Lemma 3: If C1 and C2 are valid concepts, then the axiom $(C1 \cap C2)$ is a valid concept.

Lemma 4: A concept C is satisfiable if and only if it has a contradiction-free completion tree.

Based on these, Buyer can be considered as being equivalent to Customer if following two axioms are satisfiable:

i. Buyer ⊆ Customer

ii. Customer ⊆ Buyer

Figure 4-14 provides the completion tree of the axiom (Buyer ∩ ¬Customer). It should be noted that dashed lines represent alternate expansion options for unions and abbrevations are used in place of object-property names (i.e. DL roles) to simplify the figure:

- t in place of isOfType,
- r in place of referElement,
- c in place of representConcept.

As it can be seen from the figure, the completion tree contains contradictions and proves that the axiom (Buyer ∩ ¬Customer) is unsatisfiable and hence the axiom (Buyer ⊆ Customer) is satisfiable.

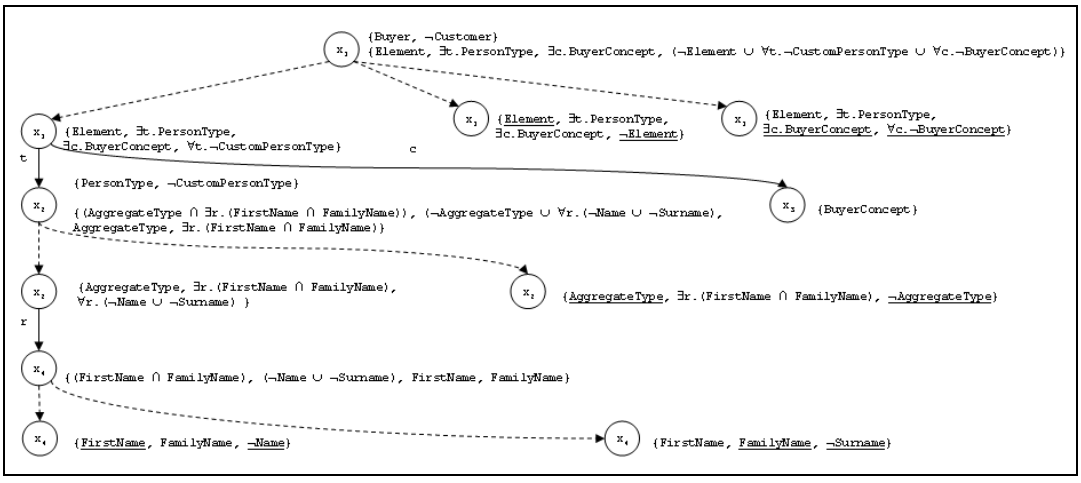

Figure 4-14 - Completion tree for the axiom (Buyer ∩ ¬Customer)

Similarly, Figure 4-15 provides the completion tree for the axiom $(Customer \cap \neg Buyer)$. The tree contains contradictions proving that the axiom is unsatisfiable and hence the axiom $(Customer \subseteq Buyer)$ is satisfiable.

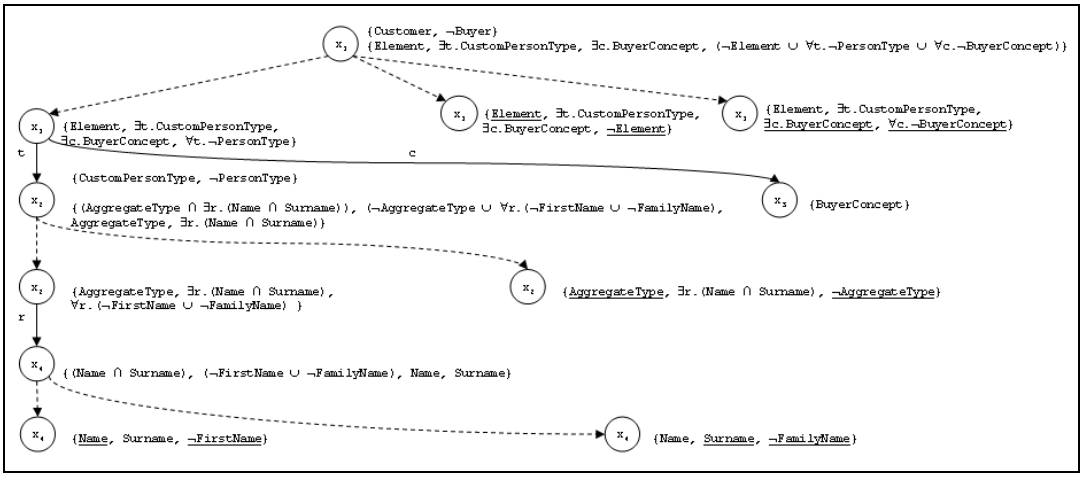

Figure 4-15 - Completion tree for axiom $(Customer \cap \neg Buyer)$

As axioms $(Buyer \subseteq Customer)$ and $(Customer \subseteq Buyer)$ are proven to be satisfiable, it can be concluded that Buyer and Customer are equivalent to each other.

This example demonstrates that, the expressiveness of Component Ontology axioms introduced in Section 4.2 is not limited to identically defined classes. As in the case of Buyer and Customer, which are defined through different axioms (axiom 3 in Figure 4-12 and axiom 11 in Figure 4-13 respectively), Component Ontology definitions allow the computation of equivalences between higher level constructs provided there are sufficient similarities between lower level constructs.

Following similar steps for higher level constructs, Order can be considered as being equivalent to CustomOrder if following two axioms are satisfiable:

i. Order ⊆ CustomOrder

ii. CustomOrder ⊆ Order

Completion trees for axiom (Order ∩ ¬CustomOrder) in Figure 4-16 and for axiom (CustomOrder ∩ ¬Order) in Figure 4-17 prove that axioms (Order ⊆ CustomOrder) and (CustomOrder ⊆ Order) are satisfiable and hence Order and CustomOrder are equivalent to each other.

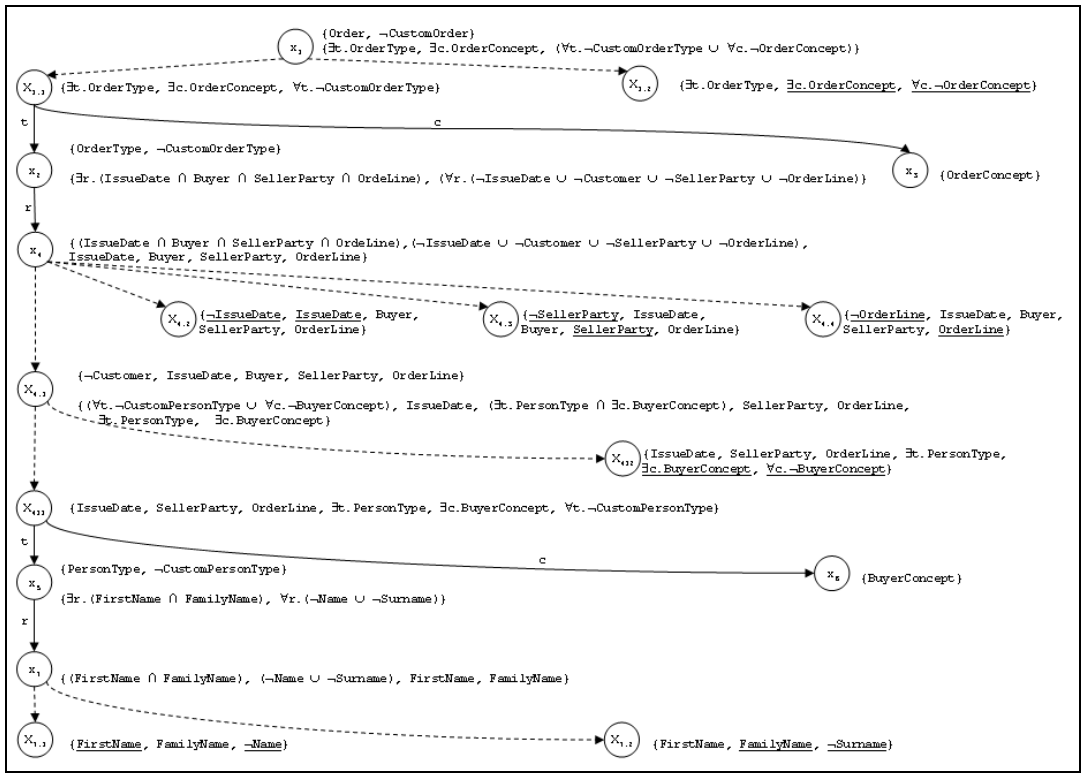

Figure 4-16 - Completion tree for axiom (Order ∩ ¬CustomOrder)

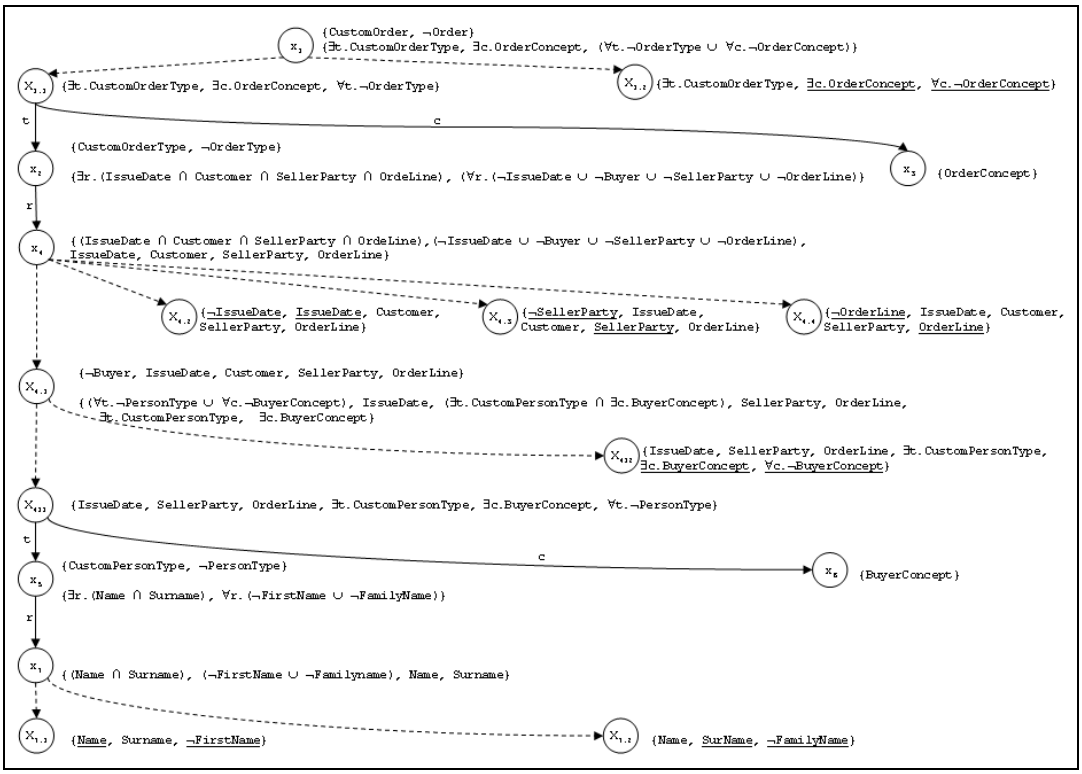

Figure 4-17 - Completion tree for axiom (`CustomOrder ∩ ¬Order`)

This section demonstrated how DL axioms introduced in Section 4.2 allow automated processes to discover equivalances between similarly composed components. Hence, it can be concluded that an equivalence relationship between Component Ontology classes is an indication of structural and semantic similarity between corresponding constructs and those components are considered to be translatable to each other.

4.4 Components with Subsuming Content

Section 4.3 describes how translatability is computed for components with matching content. Nevertheless, real-life scenarios are more complicated. Even though the original UBL specification includes no redundancy, tailored schemas might introduce redundancy as they define additional components resulting in multiple types with the

same or very similar content. Therefore it is not feasible to depend solely on a one-to-one equivalence in order to be able to translate between schemas. This section discusses the expressiveness of the Component Ontology for the case for components that subsume each other.

As an example, consider the `NewOrder` in Figure 4-18 and corresponding DL axioms in Figure 4-19 which is a possible alternative to the Order in Figure 4-1.

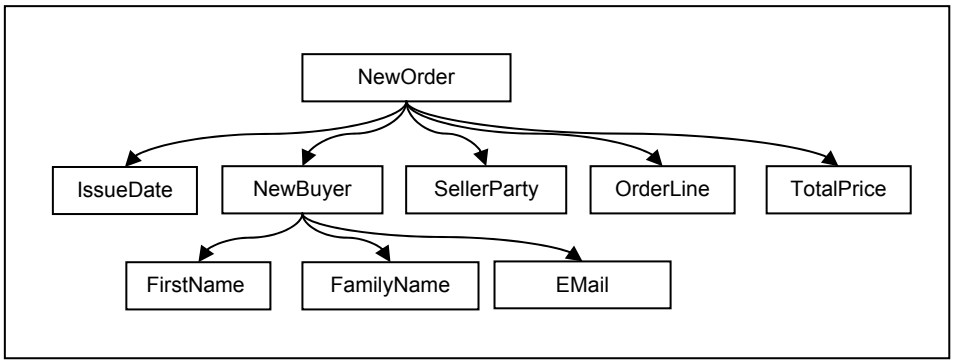

Figure 4-18 – An alternative component for the Order in Figure 4-1

The axiom 3 in Figure 4-12 defines the necessary condition for being a Buyer as "Any ElementDeclaration that represents the BuyerConcept and structured according to PersonType". The axiom 4 in Figure 4-12 defines the necessary condition for being a PersonType as "Any TypeDefinition that refers to FirstName and FamilyName elements". Based on these definitions and axioms 3 and 4 in Figure 4-19, it may be observed that NewBuyer satisfies the necessary condition for being a Buyer however Buyer does not satisfy the condition for being a NewBuyer.

```
1. NewOrder ≡        (ElementDeclaration ∩
                     (∃representConcept. OrderConcept) ∩
                     (∃isOfType. NewOrderType))

2. NewOrderType ≡    (AggregateType ∩ (∃referElement.
                     (IssueDate ∩ NewBuyer ∩ SellerParty
                        ∩ OrderLine ∩ TotalPrice)))

3. NewBuyer ≡        (ElementDeclaration ∩
                     (∃representConcept. BuyerConcept) ∩
                     (∃isOfType. NewPersonType))

4. NewPersonType ≡   (AggregateType ∩ (∃referElement.
                     (FirstName ∩ FamilyName ∩ EMail)))
```

Figure 4-19 - DL axioms corresponding to the NewOrder in Figure 4-18

For the purposes of proving these observations, Figure 4-20 provides the completion tree of the axiom $(Buyer \cap \neg NewBuyer)$. As it can be seen from the figure, the completion tree produces a contradiction-free instance (consisting of nodes $x_{1.1}$, x_2, x_3, $x_{4.3}$), proving the satisfiability of the axiom $(Buyer \cap \neg NewBuyer)$, which in turn proves the unsatisfiability of the axiom $(Buyer \subseteq NewBuyer)$.

On the other hand, the completion tree of the axiom $(NewBuyer \cap \neg Buyer)$ in Figure 4-21 does not have a contradiction-free instance, proving the unsatisfiability of the axiom $(NewBuyer \cap \neg Buyer)$ and satisfiability of the axiom $(NewBuyer \subseteq Buyer)$.

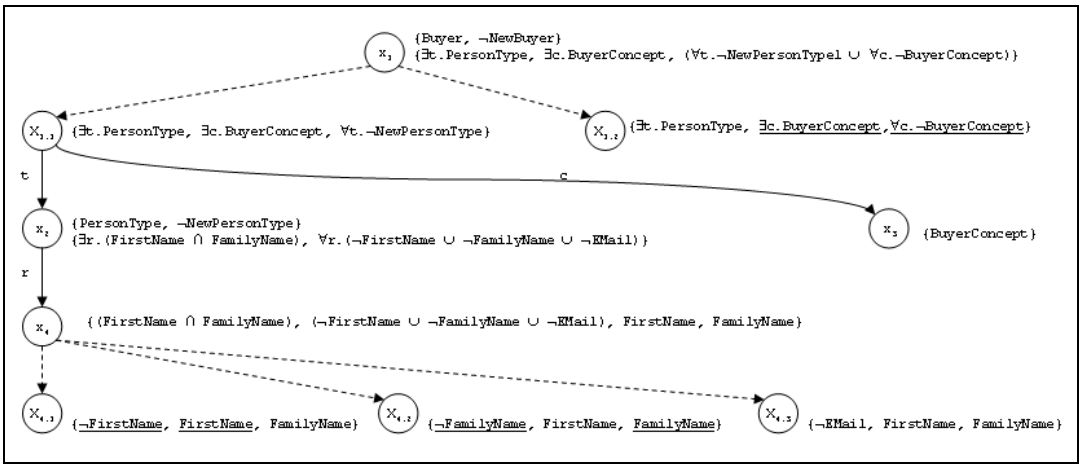

Figure 4-20 - Completion tree for axiom (Buyer ∩ ¬NewBuyer)

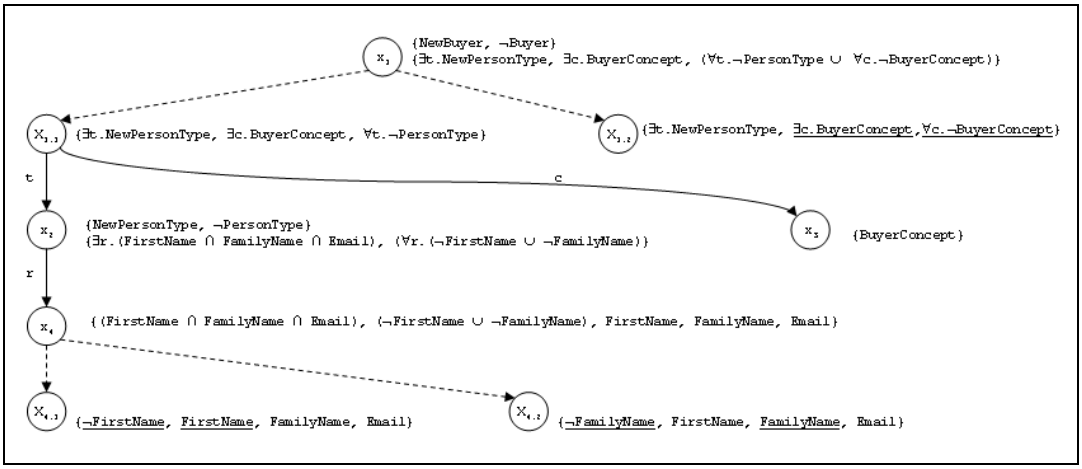

Figure 4-21 - Completion tree for axiom (NewBuyer ∩ ¬Buyer)

Following similar steps for higher level constructs, Figure 4-22 provides the completion tree of the axiom (NewOrder ∩ ¬Order). The tree contains

contradictions and hence proves the satisfiablity of the axiom (NewOrder ⊆ Order).

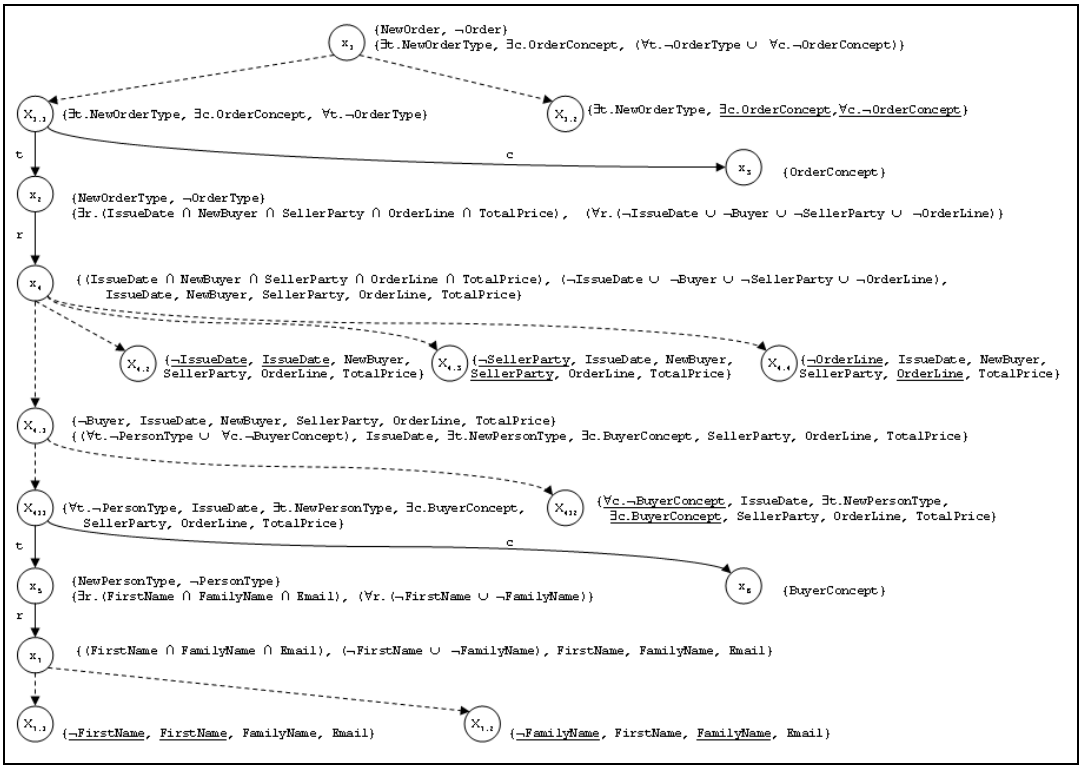

Figure 4-22 – Completion tree for axiom (NewOrder ∩ ¬Order)

This example demonstrates that, Component Ontology axioms introduced in Section 4.2 allow the computation of class-subclass relationships between components that represent identical concepts and are composed of content that subsume each other, that is, include all components (or their equivalents) included by other components. For the purposes of translation, this has the consequence that whenever a component does not have an equivalent component for a context, an applicable sub-class or an applicable super-class can be used for translation as they represent components with common content:

- An equivalence relationship is an indication of symmetric translatability and is the ideal condition. In other words, provided the UBL Component Ontology classes representing components C_1 and C_2 are equivalent to each other, it is possible to fully translate the content of C_1 into C_2 and then back to C_1.

- On the other hand, a class-superclass relationship is an indication of non-symmetric translation. In other words, provided the UBL Component Ontology classes representing the component C_1 is a sub-class of the ontology class representing the component C_2, it is possible to fully translate the content of C_1 into C_2 but the reverse is not true. That is, some of the content in C_2 has no correspondence in C_1 and cannot be translated.

4.4.1 Structurally Different Components

An interesting case to consider about components that subsume each other is structurally different components. Figure 4-23 provides a simple example of this case, in which the content of a component is subsumed by another in a structurally different manner. Figure 4-24 provides DL axioms defining ProviderParty and ProviderParty2.

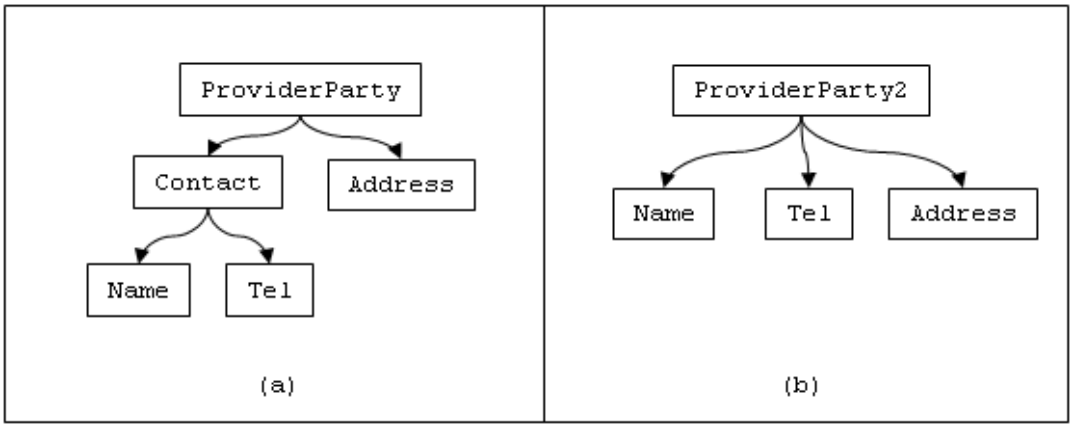

Figure 4-23 – Simplified ProviderParty versions

```
1. ProviderParty ≡  (ElementDeclaration ∩
                    (∃representConcept. ProviderPartyConcept) ∩
                    (∃isOfType. PartyType))

2. PartyType ≡      (AggregateType ∩
                    (∃referElement. (Contact ∩ Address))

3. Contact ≡        (ElementDeclaration ∩
                    (∃representConcept. ContactConcept) ∩
                    (∃isOfType. ContactType))

4. ContactType ≡    (AggregateType ∩ (∃referElement. (Name ∩ Tel))

5. Address ≡        (ElementDeclaration ∩
                    (∃representConcept. AddressConcept) ∩
                    (∃isOfType. AddressType))

6. AddressType ≡    (BasicType ∩ (∃extend. TextType))

7. Name ≡           (ElementDeclaration ∩
                    (∃representConcept. NameConcept) ∩
                    (∃isOfType. NameType))

8. NameType ≡       (BasicType ∩ (∃extend. TextType))

9. Tel ≡            (ElementDeclaration ∩
                    (∃representConcept. TelConcept) ∩
                    (∃isOfType. TextType))

10.TelType ≡        (BasicType ∩ (∃extend. TextType))

11.ProviderParty2 ≡ (ElementDeclaration ∩
                    (∃representConcept. ProviderPartyConcept) ∩
                    (∃isOfType. PartyType2))

12.PartyType2 ≡     (AggregateType ∩
                    (∃referElement. Name ∩ Tel ∩ Address))
```

Figure 4-24 - DL axioms corresponding to the Figure 4-23

Even though ProviderParty and ProviderParty2 could be considered to be related to each other, axioms in Figure 4-24 do not specify a relation among them; specifically because Component Ontology properties are defined as non-transitive and no relationships are defined between concept classes. Figure 4-25 provides the completion tree for axiom (ProviderParty ∩ ¬ProviderParty2), which is contradiction-free and hence proves the unsatisfiablity of the axiom (ProviderParty ⊆ ProviderParty2).

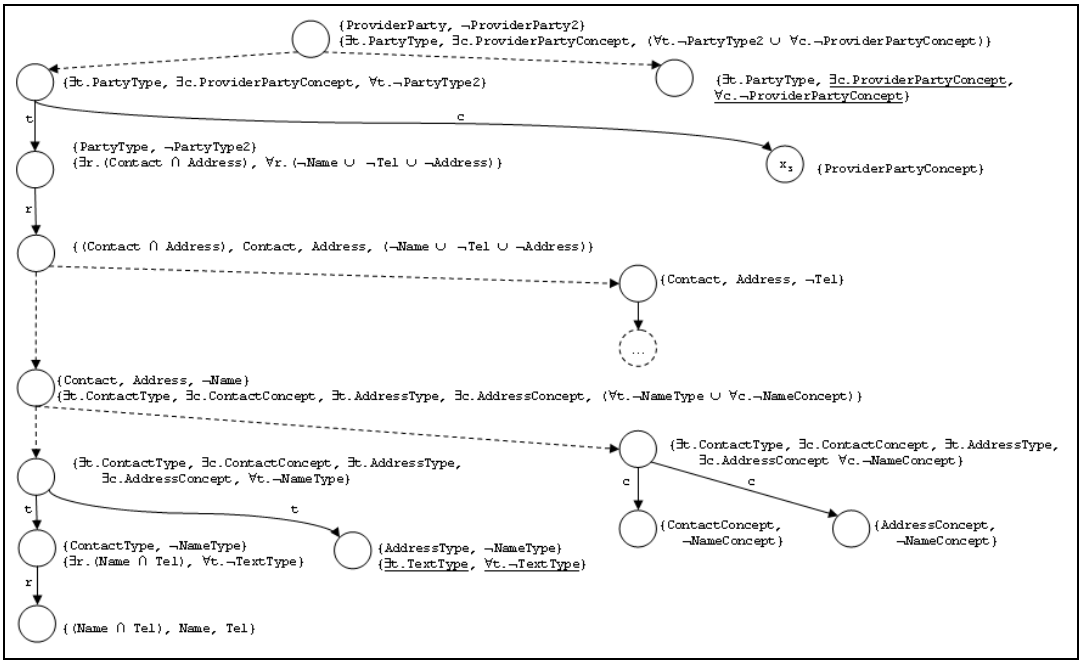

Figure 4-25 – Completion tree for axiom (ProviderParty ⊆ ProviderParty2)

Making Component Ontology properties transitive instead of non-transitive and specifying relationships amongs concept classes changes this. By definition [56], P being a transitive property, P(x, y) and P(y, z) implies P(x, z). Hence, by making isOfType and referElement properties transitive, the relationships between ProviderParty-Contact and Contact-Name pairs implies the same relationship between ProviderParty and Name components.

Defining relationships between concepts that are represented by UBL components is beyond the scope of the study presented in this paper. However, for the sake of observing the outcome, assume (ContactConcept ⊆ NameConcept) and (ContactConcept ⊆ TelConcept) axioms are added to the Component Ontology about concept classes.

Figure 4-26 provides the completion tree for the axiom (ProviderParty ⊆ ProviderParty2) with transitive properties and additional concept axioms. It should be noted that the tree is simplified to improve readability and only shows those axioms that cause contradictions for different branches, including (¬ContactConcept ∪ NameConcept) and (¬ContactConcept ∪ TelConcept) axioms which are added to all nodes yet are only shown where they lead to contradictions. As expected, the completion tree contains contradictions, and proves the satisfiablity of the axiom (ProviderParty ⊆ ProviderParty2).

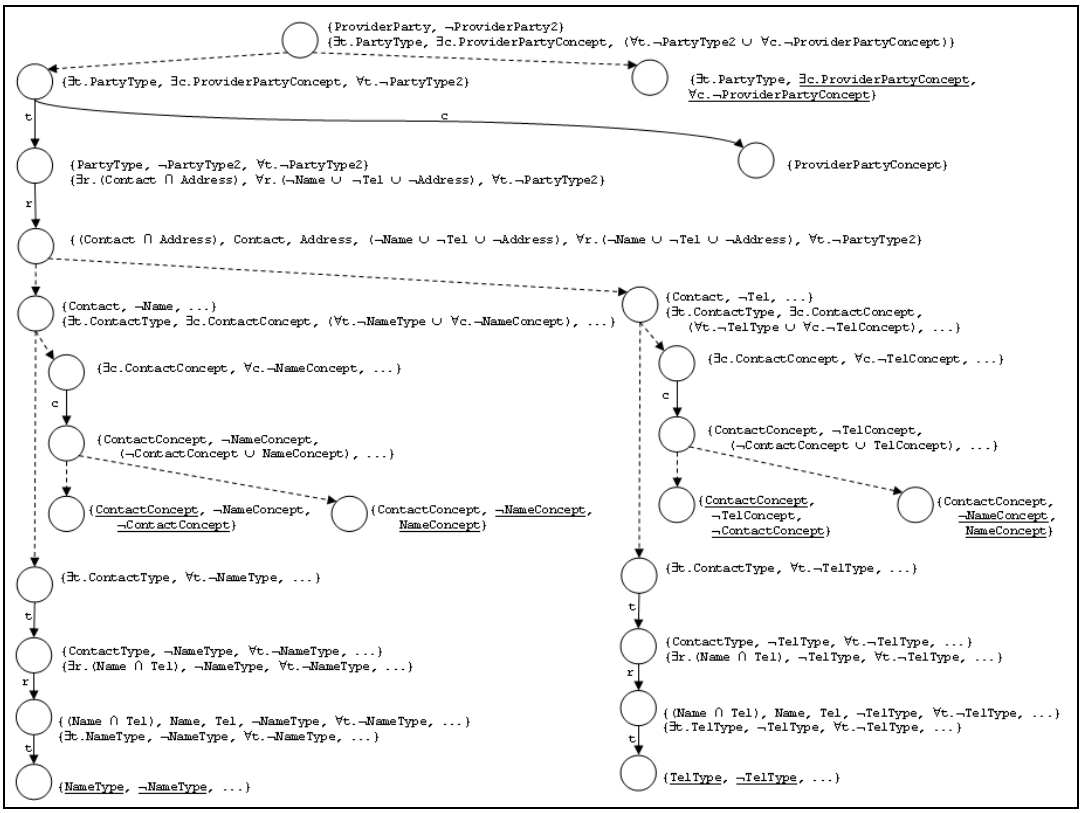

Figure 4-26 – Completion tree of axiom (ProviderParty ⊆ ProviderParty2)

Nevertheless, modifications to the Component Ontology come with a performance penalty. As described in [57], every additional axiom of the form ($A \subseteq B$) defined in Component Ontology causes a corresponding axiom of the form ($\neg A \cup B$) to be added to nodes of the completion tree. This causes an additional branching for every single node and exponentially increases the processing required to construct the completion tree as more axioms are added. When this is combined with the additional processing required for transitive properties, the reasoning performance of the Component Ontology degrades significantly.

Figure 4-27 provides a comparision of processing times required for the two versions of the Component Ontology. All metrics are collected on a Windows XP workstation with Pentium 4 2.8 GHz CPU and 1 GB of RAM. *Original Ontology* denotes the reasoning performance of the Component Ontology described in Section 4.2. *Modified Ontology* denotes the performance of the Component Ontology with transitive properties and additional axioms described in this section. As seen from the figure, for 279 aggregate components of the UBL standard, the reasoning performance jumps from 109 seconds to 2230 seconds. With the exponential increase in processing time, definitions for additional components from customized schemas is likely to cause the processing time of the modified ontology to exceed beyond practically acceptable limits.

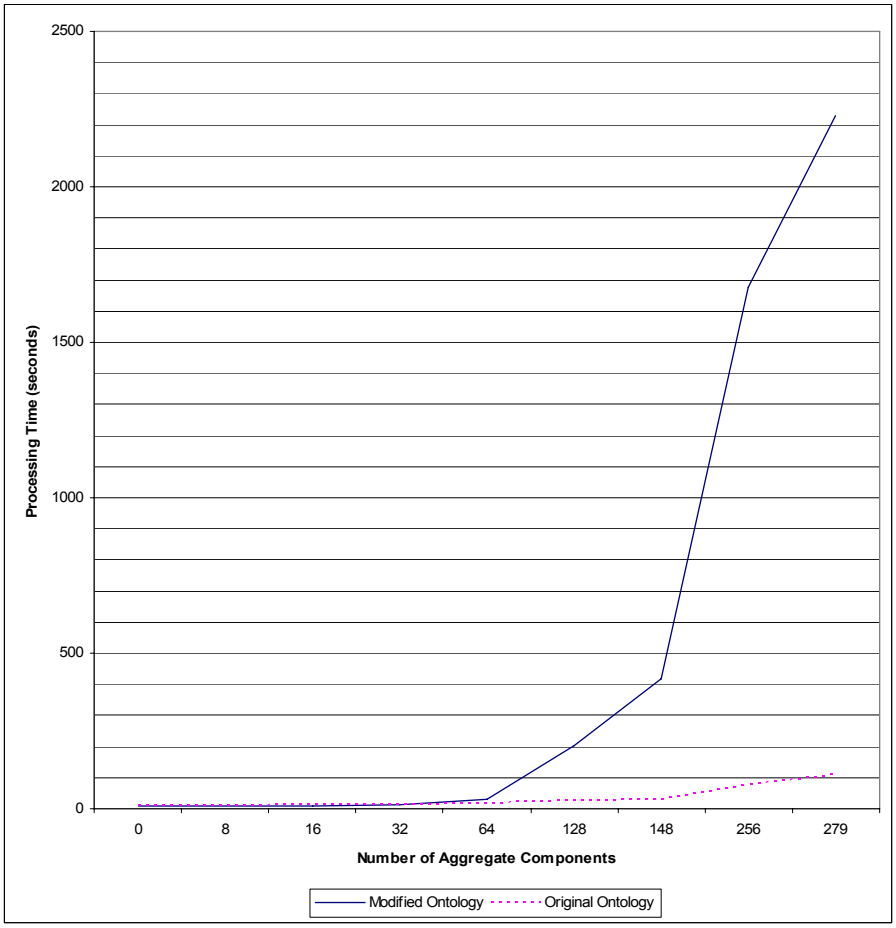

Figure 4-27 - Processing times for original and modified ontologies

As a consequence of these performance figures, the study presented in this document is limited to using the Component Ontology version described in Section 4.2, which is expressive enough to demonstrate the promises of using semantic constructs for computing translatability of components. As faster and more optimized reasoners emerge and more powerful hardware components become available, expressiveness of the Component Ontology could be improved with additional axioms such as the ones discussed in this section.

CHAPTER 5

SYSTEM ARCHITECTURE AND OPERATION

Development of ontologies that represent context domains described in Chapter 3 lays the necessary foundation for implementing automated processes that can intelligently discover components and customize document schemas. When this capability is combined with the UBL Component Ontology representing the semantics of individual components, it becomes possible to develop processes that can translate content between different schemas versions. This chapter describes our system architecture designed to support the development of such processes.

Figure 5-1 displays an overall view of the proposed architecture which provides UBL users a number of tools for component definition, component discovery, document schema customization and document translation. These tools are supported by internal services which interact with a knowledge base through a reasoning layer. The knowledge base stores the UBL Component Ontology and metadata about system artifacts such as context ontologies and customized components.

5.1 Knowledge Base

Context Ontology Metadata shown in Figure 5-1 stores information about context ontologies discussed in Chapter 3. Using these metadata, Reasoning Layer locates individual context ontologies and computes inferred context ontologies representing context domains. All requests from internal services concerning context ontologies are served using these inferred context ontologies.

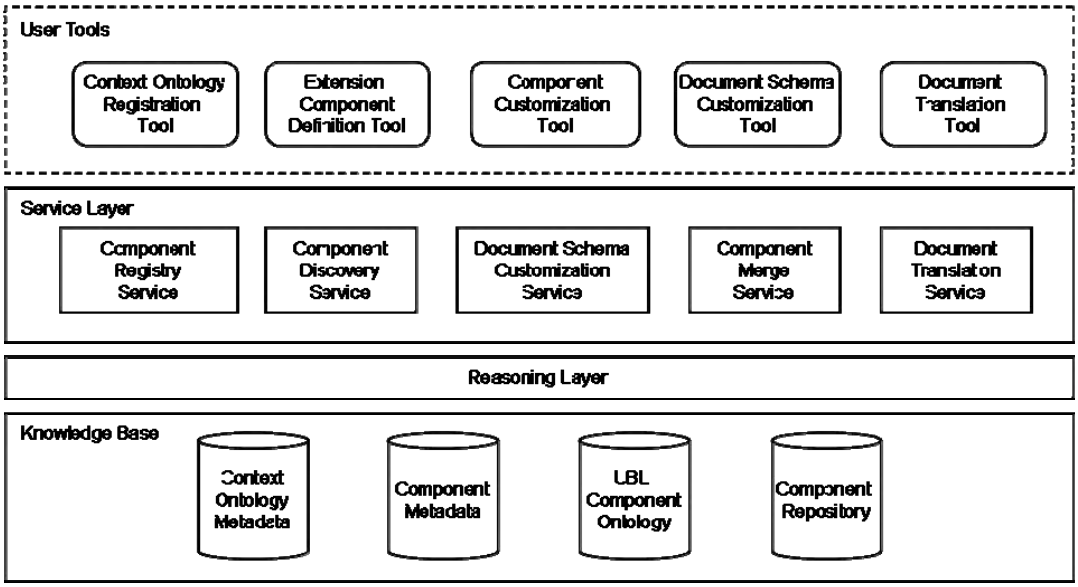

Figure 5-1 - System Architecture

Component Repository stores definitions for standard and custom components. Component metadata provides information about all components, being used specifically for the following purposes:

- When a component is customized for a context, corresponding metadata is created to express the standard UBL component it is derived from and the context it is applicable to by specifying references to classes from inferred ontologies,

- When a new -extension- component is defined for a context, corresponding metadata is created to express the context it is applicable to,

- When a custom version of a component is required for a specific context, component metadata is queried to gather applicable versions with the help of inferred context ontologies,

- When a document schema needs to be customized for a specific context, component metadata is queried to gather applicable versions of components included in that schema and those versions are used to replace original components in the customized document schema.

UBL Component Ontology, as described in Section 4.2, consists of OWL definitions that describe elements, types and concepts included in standard and custom UBL schemas. Corresponding definitions for standard UBL components are generated for the UBL Component Ontology. Whenever an extension component is defined by users, necessary definitions are automatically generated by the Component Registration Service and added to the component ontology.

Reasoning Layer loads the UBL Component Ontology, interprets class definitions and computes equivalence and class-subclass relationships among ontology classes. Internal services access that computed version of the UBL Component Ontology through the Reasoning Layer.

5.2 User Tools

A user accesses the system through a number of tools which allow him to register context ontologies, to customize components, to define extension components, to customize document schemas and to translate document instances between different schemas as described in the following:

- *Context Ontology Registration Tool* provides a GUI for registering new context ontologies and maintaining existing ones. As described in Chapter 3, individual ontologies are developed that represent classifications related to context domains. And additional ontologies may developed by user groups and organizations in the future. Assuming all such ontologies are published on the Internet and maintained by their creators, Context Ontology Registration Tool helps to register those ontologies so that their metadata is maintained by the system.

- *Component Customization Tool* provides Domain Experts a GUI for creating and maintaining custom versions of UBL components. In a typical component customization scenario, domain expert specifies the component to be

customized and the target context. The Component Discovery Service provides a baseline version of the component for the specified context. Domain expert customizes baseline version through XSD extension and restriction operations and submits the resulting component for registration. The Component Registration Service verifies that the submitted version is a valid specialization of the baseline, creates necessary metadata and ensures proper registration of the component.

- *Extension Component Definition Tool* provides Domain Experts a GUI for defining new UBL components to represent concepts that are not included in the standard UBL component library. In a typical extension component definition scenario, domain expert develops a new component through including components from the library of existing components, specifies the concept represented by the component and specifies the business context that the extension component will be applicable for. The Component Registration Service verifies that the submitted component is a valid UBL component, creates necessary metadata, ensures proper registration of the component and updates the UBL Component Ontology with necessary definitions for the extension component.

- *Document Schema Customization Tool* provides UBL users a GUI for generating customized versions of UBL document schemas based on their business context. In a typical document schema customization scenario, the user specifies a UBL document schema to be customized and its business context as a combination of classes from context ontologies. The Document Schema Customization Service, by collaborating with the Component Discovery and Component Merge services, generates a customized version of the UBL document schema for the specified context.

- *Document Translation Tool* provides UBL users a GUI for translating the content of UBL document instances from one context to another. In a typical document translation scenario, the user specifies a UBL document instance and the desired context for translation. The Document Translation Service refers to the UBL Component Ontology to gather corresponding components in the specified context. Then collaborates with the Component Discovery Service to

gather applicable versions of those components for the target context and generates a translation version of the given document instance.

5.3 Services Layer

5.3.1 Reasoning Layer

This layer provides reasoning services to the rest of the system. It consists of a Description Logics reasoner, namely the RacerPro [58], together with supporting service and utility classes. Internal services do not access the knowledge base directly. Instead, they query the knowledge base through the reasoner, which answers queries using inferred ontologies computed from ontologies residing at the knowledge base.

For each business context domain, the knowledge base contains a corresponding Context Ontology, as discussed in Section 3.1.2. Each of these ontologies contains links to individual ontologies representing particular classification schemas together with expressions generated through the ontology alignment steps. The reasoner loads each of these context ontologies, computes additional correspondences between classes from different ontologies and generates the inferred context ontology. All subsequent requests from internal services are handled using this computed ontology.

Whenever a new individual ontology is defined for a business context, the inferred ontology for that particular context domain is re-computed. Similarly, whenever a new alignment operation is defined or an existing one is modified, the corresponding inferred ontology for that context domain is re-computed.

For the case of the UBL Component Ontology, the reasoner loads the UBL Component Ontology and computes inferred relationships among that. Similar to the case of context ontologies, all subsequent requests from internal services are handled using this inferred version of the UBL Component Ontology. Whenever a new extension component is defined, or an existing one is modified and/or deleted, the inferred UBL Component Ontology is re-computed.

5.3.2 Component Registration Service

Given the XSD definition of a customized or extension UBL component and the applicable context information, Component Registration Service ensures proper registration of components.

In order to allow component discovery, knowledge base stores metadata about standard and customized UBL components as instances of `UBLComponentMetadata` and CustomComponentMetadata classes shown in Figure 5-2. Component Registration Service is responsible for the proper creation and maintenance of the metadata instances.

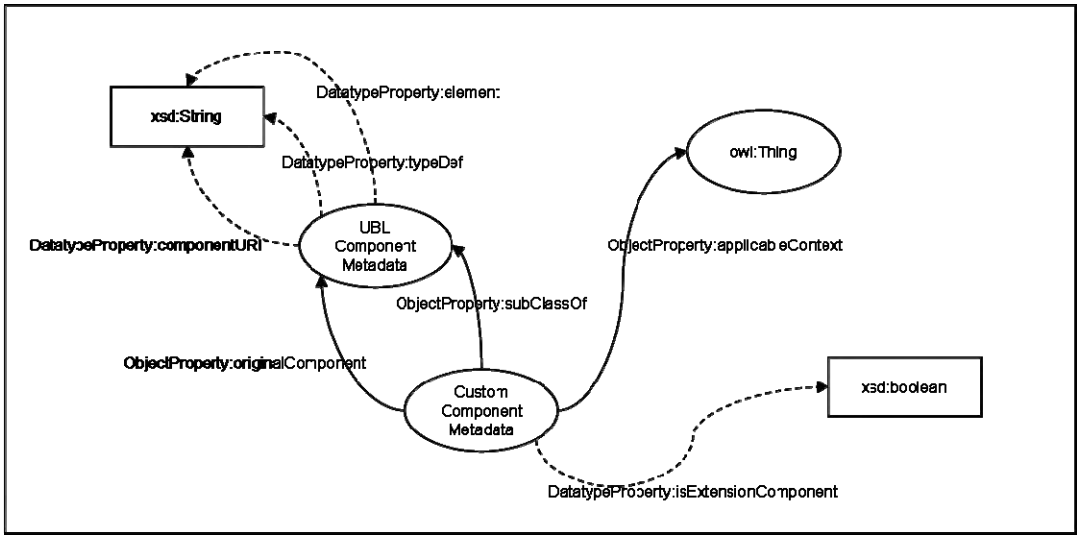

Figure 5-2 - OWL Classes representing the metadata of standard and custom UBL components

As the name suggests, UBLComponentMetadata class represents original UBL components. For every UBL component defined by the UBL standard, a corresponding

UBLComponentMetadata instance is created in the Component Repository. UBL ComponentMetadata class has the following three data type properties:

- *element*: The fully qualified element name of the component including the name and the namespace.
- *typeDef*: The fully qualified type definition name of the component including the name and namespace.
- *componentURI*: The URL of the corresponding XSD file that stores the element and type definitions for the component.

Figure 5-3 provides a sample OWL definition for a standard UBL component, whose element name is Item, type name is ItemType and is located at the URL http://www.srdc.metu.edu.tr/ublschema/common/UBL-CommonAggregate Components -2.0.xsd.

```
<UBLComponentMetadata rdf:ID="cac_Item">

    <element rdf:datatype=”string”>
       urn:oasis:names:specification:ubl:schema:xsd:
          CommonAggregateComponents-2:Item
    <element>

    <typeDef rdf:datatype=”string”>
       urn:oasis:names:specification:ubl:schema:xsd:
          CommonAggregateComponents-2:ItemType
    <typeDef>

    <componentURI rdf:datatype="string">
       http://www.srdc.metu.edu.tr/ublschema/common/
         UBL-CommonAggregateComponents-2.0.xsd
    <componentURI>

</UBLComponentMetadata>
```

Figure 5-3 - UBLComponentMetadata instance for the UBL Item component

The CustomComponentMetadata class represents custom versions of standard UBL Components as well as user defined extension components. For all customized UBL components and user defined extension components, a CustomComponentMetadata instance is created and stored in the Component Repository.

The CustomComponentMetadata class is defined as a subclass of the UBL ComponentMetadata class therefore it inherits the three data type properties described above. In addition to those, CustomComponentMetadata has the following properties:

- *applicableContext*: The fully qualified name of the ontology class representing the applicable context for the custom component. The value of this object type property can be any class from context ontologies therefore its range is defined as owl:Thing.

- *isExtensionComponent:* Boolean flag specifying whether or not the component is a user defined extension component (one that is not provided by the standard UBL specification).

- *originalComponent:* The fully qualified name of the UBLComponentMetadata or CustomComponentMetadata class representing the original component that this particular custom component version is derived from. For the case of extension components defined by users, the value of this property is undefined.

Figure 5-4 provides corresponding OWL definitions for a custom UBL component, which is a customized version of the Item component in Figure 5-3 applicable to the context value naics:23_Construction and is located at the URL http://www.srdc.metu.edu.tr/ublschema/customSchemaRepository/industry_naics_23_cnstrctn.xsd.

```
<CustomComponentMetadata
      rdf:ID="Item_indstry_naics_23_cnstrctn">

  <element rdf:datatype="string">
     srdc:industry:naics:_23_cnstrctn:ubl:Item
  </element>

  <typeDef rdf:datatype="string">
     srdc:industry:naics:_23_cnstrctn:ubl:ItemType
  </typeDef>

  <componentURI rdf:datatype="string">
    http://srdc.metu.edu.tr/ublschema/customSchemaRepository/
      industry_naics__23_cnstrctn.xsd
  </componentURI>

  <applicableContext rdf:resource="string">
     http://srdc.metu.edu.tr/contextOntology/naics.owl#
       _23_Construction
  </applicableContext>

   <isExtensionComponent rdf:datatype="boolean">
     false
   </isExtensionComponent>

  <originalComponent rdf:resource=
    "http://srdc.metu.edu.tr/
       componentRepository/ublInstances.owl#cac_Item">
  </originalComponent>

</CustomComponentMetadata>
```

Figure 5-4 - CustomComponentMetadata instance for a custom component

It should be noted that for a particular context value, there can only be one custom version of a component. Context ontologies can have multiple sibling (semantically equivalent) classes all representing the same context value, however, a custom version registered for a particular context class is considered to be applicable to all sibling classes of that class.

Before registering a new customized version, Component Registration Service verifies that the new version is a valid UBL specialization. In order to perform that, Component Discovery Service is requested to provide the baseline version of the same component for the specified context. Then, the new version is compared against the baseline

version to ensure that it is generated only through UBL conformant XSD derivation operations, namely extension and restriction.

After a component is stored in the repository, it becomes available for discovery and is included in document schema customizations regarding the context it is applicable to.

5.3.3 Component Discovery Service

The purpose of the Component Discovery Service is to search for component versions applicable to a given target context. In order to gather the applicable version of a component for a particular context value, the Component Discovery Service queries the knowledge base for a CustomComponentMetadata instance with the applicableContext property set to any of the semantically equivalent classes representing the specified context value. As described in Section 5.3.2, only one custom version is allowed to be registered for a particular context value.

For cases which knowledge base includes no metadata instance with applicableContext property equal to any of the equivalent classes, Component Discovery queries the reasoner for direct parent classes of the specified context and recursively calls itself to gather versions applicable to parent context values. In the case of single inheritance, component version applicable to the direct parent context is assumed to be applicable to the child context as well.

Nevertheless, as a result of ontology alignment operations described in Section 3.1.2, context ontology classes might have multiple inheritance and consequently there may be multiple parents with applicable custom versions. Figure 5-5 displays possible cases for a context class with multiple inheritance.

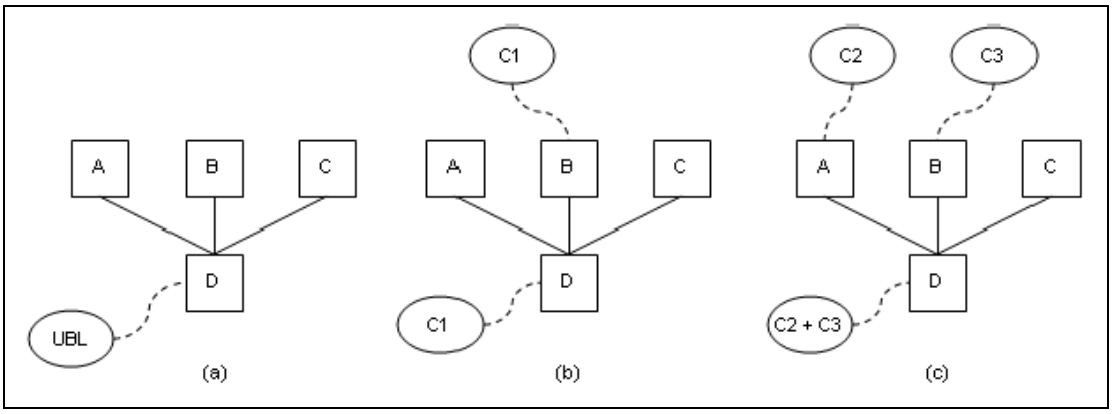

Figure 5-5 - Context class with multiple inheritance

The implications of multiple parents on the component discovery are as follows:

- Figure 5-5-(a): If there is no custom version applicable to any of the parent context values, standard UBL version is assumed to be applicable to the context.
- Figure 5-5-(b): If there is a custom version applicable to only one of the parent context classes, that version is assumed to be applicable to the context.
- Figure 5-5-(c): If there are multiple custom versions applicable to parent context classes, a new version, generated by merging those versions, is applicable to the context.

The algorithm in Figure 5-6 displays the overall steps of how component discovery is executed.

```
UBLcmp  : UBL component being searched
context : target context value

 1. reasoner.loadContextOntologies();
 2. equivalentCtxSet = reasoner.getEquivalentClasses(context);
 3.
 4. for all context class ctx ∈ equivalentCtxSet
 5.   qry =(originalComponent=UBLcmp AND applicableContext=ctx);
 6.   cmp = kBase.componentMetadata.query(qry);
 7.   if (cmp ≠ null) then
 8.     return cmp;
 9.   end if
10. end for
11.
12. //there is no custom version for the exact context value,
13. //check parent context values
14. cmpList = new List();
15. directParentSet = reasoner.getDirectParentClasses(context);
16.
17. for all context class parentCtx ∈ directParentSet
18.   cmp = discoverComponent(UBLCmp, parentCtx);
19.   cmpList.add(cmp);
20. end for
21.
22. //check the number of parents with custom versions
23. if cmpList.size is 0 then            //no parent
24.   return null;
25. else if cmpList.size is 1 then      //only one parent
26.   return cmpList[0];
27. else                                //multiple parents
28.   cmp = mergeComponents(cmpList);
29.   return cmp;
30. end if
```

Figure 5-6 - Component Discovery Algorithm

It should be noted that, since Component Discovery algorithm recursively calls itself for parent context classes, even component versions that are registered for higher level context classes may get merged to generate an applicable version for a particular context. Figure 5-7 displays an example ontology fragment where component versions linked with solid lines represent registered versions and those that are linked with dashed lines represent the influence of registered versions on lower lever context values.

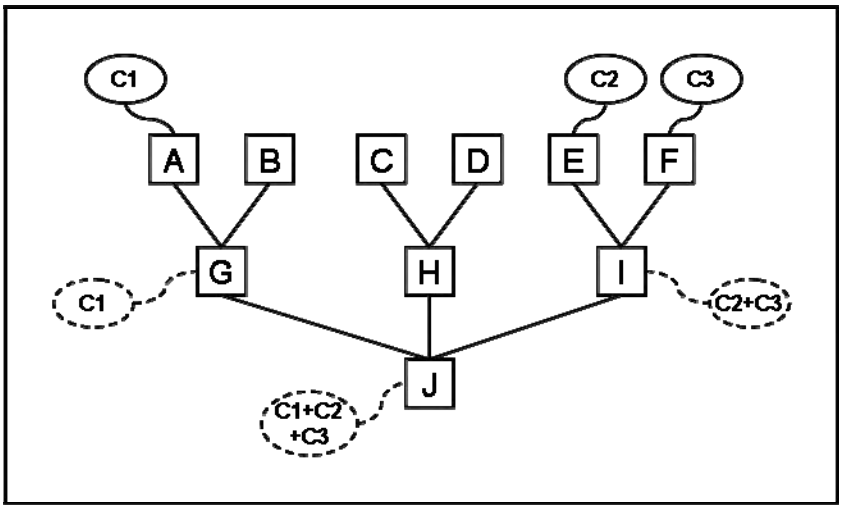

Figure 5-7 - Influence of component versions on lower level context values

Assume the Component Discovery Service is called to gather the applicable version of a component for the context value `J` for which the inferred context ontext ontology is shown in Figure 5-7. There are no components registered for the context value `J`, and the Component Discovery algorithm recursively calls itself for parent context values, namely `G`, `H` and `I`.

There is no registered component version applicable for the context value `G` and the Component Discovery algorithm recursively calls itself for direct parent context values of `G` namely for `A` and `B`. Only context value `A` has a registered component version, namely `C1` which is returned from the recursive call made for context value `G`.

There is no registered component version applicable neither for the context value `H` nor for its parents, therefore the recursive call made for context value `H` does not return any components.

There is no registered component version applicable for the context value `I` and the Component Discovery algorithm recursively calls itself for direct parent context values of `I`, namely for `E` and `F`. Both context value `E` and context value `F` has registered component versions, namely `C1` and `C2`. Component Discovery Service collaborates

with the Component Merge Service to generate a merged version of those two versions, denoted as `C2 + C3` in Figure 5-7 and the recursive call made for the context value `I` returns that merged version.

As a result, the original call to the Component Discovery Service gathers two versions from recursive calls, namely the `C1` and the `C2 + C3`. Collaborating with the Component Merge Service, a merged version of those two versions is generated, denoted `C1 + C2 + C3` in Figure 5-7 and returned to the original service requester.

5.3.4 Document Schema Customization Service

Given a UBL document schema and a business context, the Document Schema Customization Service customizes the document schema by replacing the original UBL components with the customized components applicable for that particular context.

UBL document schemas are composed of several basic and aggregate components. Aggregate components themselves are collections of other basic and aggregate components in a recursive manner. Many aggregate components in those hierarchies are included by other components and UBL document schemas themselves. So customizing an intermediate component for a context implicitly customizes UBL document schemas including that component for the same context.

As an example, consider the greatly simplified Order and Catalogue document schemas in Figure 5-8 and Figure 5-9 respectively. Both document schemas contain the Item component in their hierarchy: the Catalogue schema includes it through the CatalogueLine component and the Order schema includes it through the OrderLine component. Therefore, customizing the Item component for a context, for example by adding the expirationDate component for the *Drugs and Pharmaceutical Products* context, has the effect of implicitly customizing the Order and the Catalogue schemas for the same context. Whenever those document schemas are requested for the *Drugs and Pharmaceutical Products* context or any sub-context such as the *Antibiotics*, the customized Item version replaces the original Item eventually modifying the schema of the relevant documents.

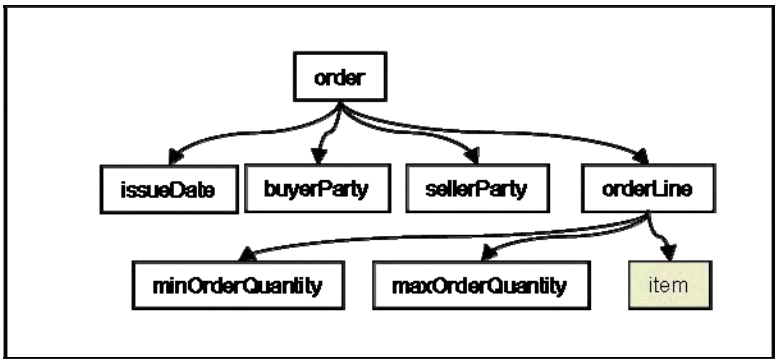

Figure 5-8 Simplified `Order` Schema

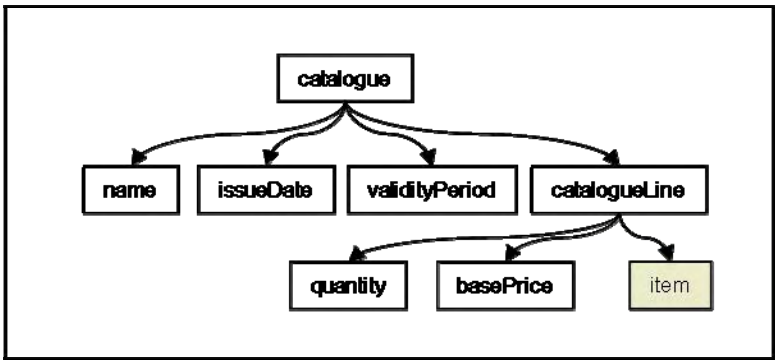

Figure 5-9 – Simplified `Catalogue` Schema

In order to customize a document schema for a particular business context, Document Schema Customization Service traverses the component hierarchy of the schema, gathers applicable custom versions through the Component Discovery Service and replaces them in place of the original UBL components. The algorithm in Figure 5-10 displays the overall steps of how the schema customization is executed.

```
schema : UBL document schema to customize
ctxSet : Set of context classes denoting the target context

 1. customSchema = schema.clone();
 2. for all component cmp ∈ schema
 3.   componentList = new List();
 4.
 5.   for all context class ctx ∈ ctxSet
 6.     tmp = discoverComponent(cmp, ctx);
 7.     componentList.add(tmp);
 8.   end for
 9.
10.   if componentList.size is 1 then
11.     replaceComponent(customSchema, cmp, componentList[0]);
12.   else if componentList.size > 1 then
13.     mergedCmp = mergeComponents(componentList);
14.     replaceComponent(customSchema, cmp, mergedCmp);
15.   end if
16. end for
```

Figure 5-10 - Document Schema Customization Algorithm

Note that a business context value may consist of multiple context categories and for each component included in the document schema hierarchy, the Component Discovery service is called once for every context category. It is possible for each of those calls to return a different version of the component. For such cases, Component Merge Service is called to merge multiple versions into one and that version is used to replace the standard UBL component in the customized document schema.

As an example to demonstrate how Document Schema Customization Algorithm functions, consider the Context Ontology fragment for the *Product Classification* context in Figure 5-11 and the Context Ontology fragment for the *Industrial Classification* context in Figure 5-12.

Figure 5-11 displays that a custom version of the ValidityPeriod component is registered for the *unspsc:51_Drugs_and_Pharmaceutical_Products* context, denoted as the ValidityPeriod_D, and a custom version of the Item component is registered for the *unspsc:511015_Antibiotics* context, denoted as the Item_A. Similarly, Figure 5-12 displays that a custom version of the Item component is registered for the *naics:Manufacturing* context, denoted as the Item_M.

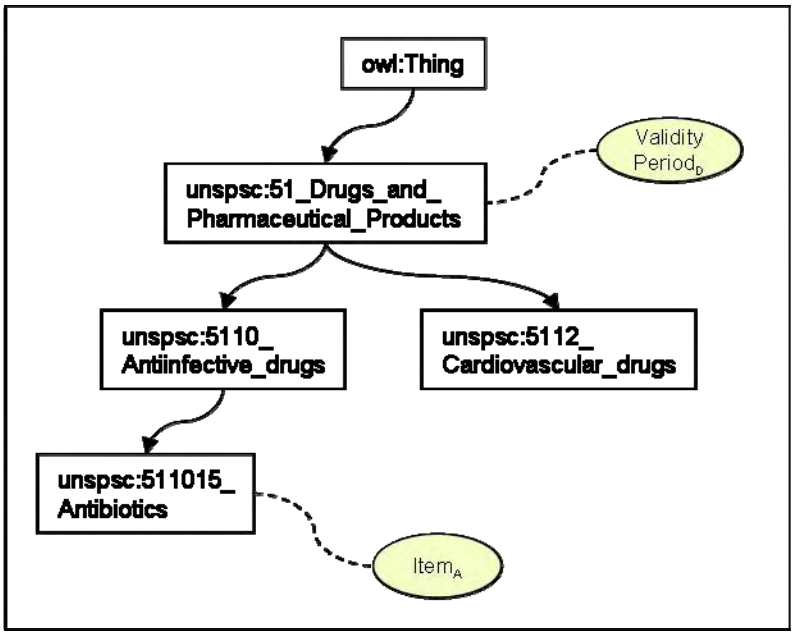

Figure 5-11 – Sample Context Ontology for the *Product Classification* context

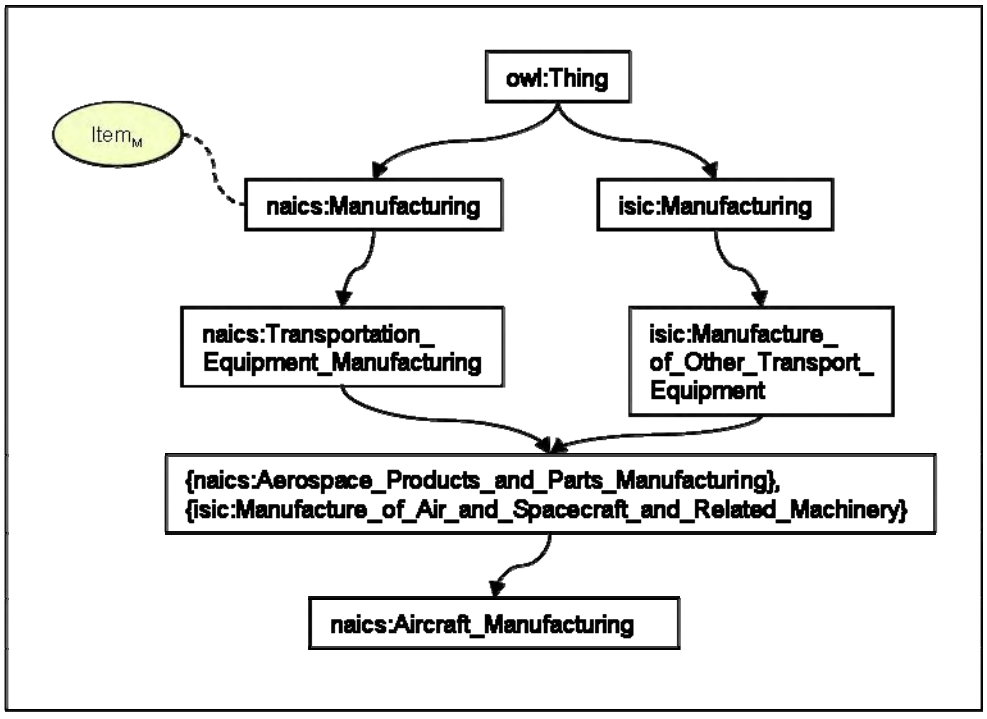

Figure 5-12 – Sample Context Ontology for the *Industrial Classification* context

Now assume that the Document Schema Customization service is called to generate a custom version of the Catalogue document schema in Figure 5-9 applicable for the business context value A*ntibiotics Manufacturing*, specified using a combination of the isic:Manufacturing and the unspsc:511015_Antibiotics classes.

Executing the algorithm in Figure 5-10, standard UBL versions of the Name, IssueDate, CatalogueLine, Quantity and BasePrice components are kept intact in the custom Catalogue schema as there are no registered custom versions for those components.

For the ValidityPeriod component, the Component Discovery Service checking the *Product Classification* context ontology shown in Figure 5-11 returns the ValidityPeriod_D version since there is no custom version registered for the target unspsc:511015_Antibiotics context, but there is one registered for a parent context, namely the unspsc:51_Drugs_and_Pharmaceutical_Products context. On the other hand, the Component Discovery service checking the *Industrial Classification* context ontology shown in Figure 5-12 does not return any components and the ValidityPeriod_D version replaces the original UBL ValidityPeriod in the custom Catalogue schema.

For the Item component, the Component Discovery Service checking the *Product Classification* context ontology in Figure 5-11 returns the Item_A version and the Component Discovery Service checking the *Industrial Classification* context ontology in Figure 5-12 returns the Item_M version. Since Component Discovery resulted in multiple versions for the Item component, Component Merge Service is called to merge those versions to generate a single Item component, denoted as $\text{Item}_A + \text{Item}_{M,}$ and that merged version replaces the original Item component in the custom Catalogue schema.

Figure 5-13 displays the generated version of the Catalogue document schema customized for the *Antibiotics Manufacturing* business context where components that are replaced in place of their standard UBL counterparts are highlighted.

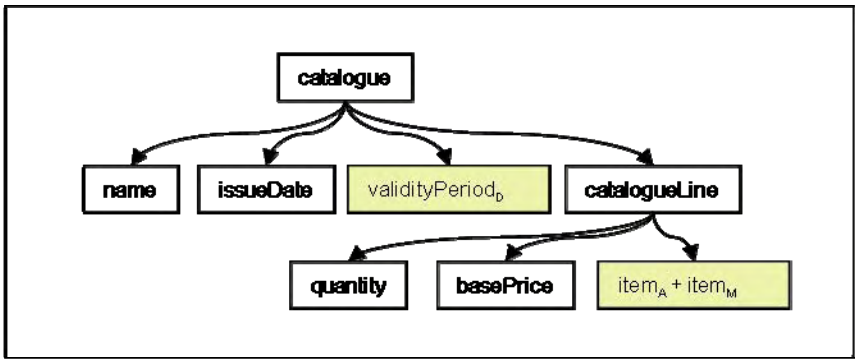

Figure 5-13 – Catalogue Schema in Figure 5-9 customized for the *Antibiotics Manufacturing* business context value

5.3.5 Component Merge Service

Given multiple versions of a UBL component, Component Merge Service generates a combined version of the specified component that represents all different versions. When a target context value includes multiple categories or has multiple parents or a mixture of both, Component Repository may contain multiple applicable versions of a particular component. For such cases, Component Merge Service extracts XSD derivations from individual versions, serializes them and successively applies to the corresponding UBL component. As an example consider the simplified version of the `Item` component in Figure 5-14.

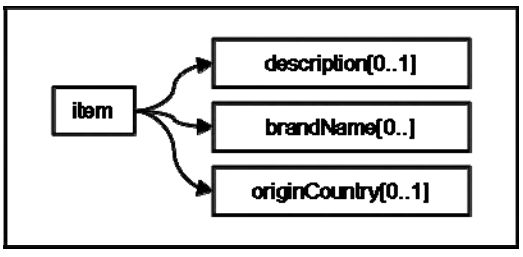

Figure 5-14 – Simplified Item component

Assume Figure 5-15 is a version of the Item component in Figure 5-14 customized for the *Retail Trade* context by restricting the cardinality of the brandName from [0..unbounded] to [1..unbounded] and by removing the originCountry by setting its cardinality to [0..0].

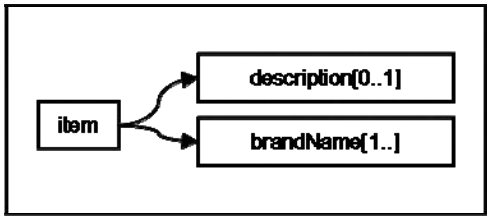

Figure 5-15 – Sample customized version of the Item component in Figure 5-14 for the *Retail Trade* context

Similarly, assume Figure 5-16 is another version of the Item component in Figure 5-14 customized for the *Drugs and Pharmaceutical Products* context by restricting the cardinality of the brandName from [0..unbounded] to [0..5] and by extending through the addition of a new reference for the ID.

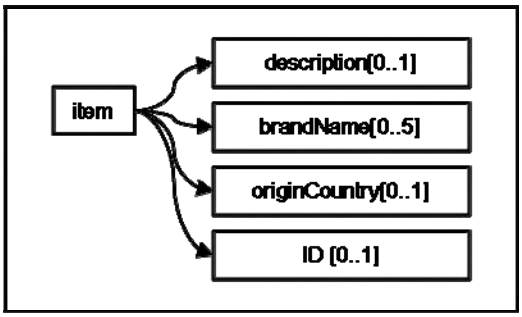

Figure 5-16 - Sample customized version of the Item component in Figure 5-14 for the *Drugs and Pharmaceutical Products* context

When the applicable version of the Item component is required for a context value such as the *Drugs and Pharmaceutical Products* and the *Retail Trade*, neither version can directly replace the standard Item. Instead, a merged version, reflecting all derivations from customized versions needs to be generated. The algorithm in Figure 5-17 displays the overall steps of how component merge is executed:

```
UBLcmp       : standard UBL version from which the custom
               versions are derived from
versionList : list of custom component versions to be merged

 1. extensionList = new List();
 2. restrictionList = new List();
 3.
 4. for all component version ∈ versionList
 5.   extensionList.add(UBLcmp.gatherExtensions(version));
 6.   restrictionList.add(UBLcmp.gatherRestrictions(version));
 7. end for
 8.
 9. extensionList = eliminateRedundantExtensions(extensionList);
10.
11. if restrictionList.conflictingCardinalities() is true then
12.   terminate(); //seek assistance to resolve the conflict
13. end if
14.
15. mergedCmp = UBLcmp;
16. derivationList = extensionList ∪ restrictionList;
17. for all derivationOperations drvOperation ∈ derivationList
18.   mergedCmp = mergedCmp.derive(drvOperation);
19. end for
20.
21. return mergedCmp;
```

Figure 5-17 - Component Merge Algorithm

When the Component Merge Algorithm is called for merging the Item component versions in Figure 5-15 and Figure 5-16, it executes through the following steps:

1) For the Item version in Figure 5-15, following two restriction operations are gathered:

   a) brandName-> [1..unbounded]
   b) originCountry->[0..0]

2) For the Item version in Figure 5-16, following derivations are gathered:

   a) one restriction operation: brandName->[0..5]
   b) and one extension operation: ID->[0..1]

3) These derivation operations are then successively applied to the original Item component in Figure 5-14 to generate the merged version of the Item component shown in Figure 5-18.

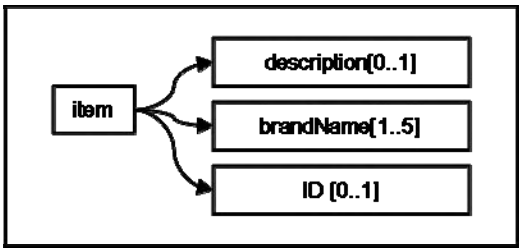

Figure 5-18 – A sample custom version of the Item component in Figure 5-14, generated by merging the versions in Figure 5-15 and Figure 5-16

It should be noted that the version of the Item component in Figure 5-18 is a valid specialization of all the versions shown in Figure 5-14, Figure 5-15 and Figure 5-16 in terms of XSD validation. In other words, any instance document conforming to the Item schema shown in Figure 5-18 also conforms to Item schemas shown in Figure 5-14, Figure 5-15 and Figure 5-16 as mandated by the UBL specification.

Merge algorithm handles all cases except when there are conflicting cardinalities. The most common case is the one in which one customization eliminates an element by specifying a [0..0] cardinality, and another mandates the same element with a [1..1] cardinality. When there is no valid intersection of cardinalities specified for an element, versions can not be automatically merged and require human assistance.

5.3.5.1 Removing Redundancy

The distributed approach presented in our work relies on independent domain experts and users for providing customized versions for components that are tailored for specific business context needs. As a consequence of this distributed approach, it is reasonable to expect the creation of various similar versions of components customized

for different business context values. As mentioned in Section 5.3.3, whenever the target business context consists of multiple context drivers, Component Discovery may return multiple versions of a component and those versions need to be merged before replacing original components in customized schemas. However, when components to be merged consist of similar elements, merely merging them introduce redundancy. That is; when the set of components added in a custom version has a non-empty intersection with the set of components added for another customization, copying all extension components from both versions into the merged component introduce *structural redundancy*. Moreover, the set of added components might be disjoint in terms of component name and type yet there may be semantically equivalent components and adding such components to the merged version introduce *semantic redundancy*. Clearly, it is good practice to avoid both types of redundancy.

Avoiding structural redundancy is rather simple. As shown in steps 4 through 7 of the Component Merge Algorithm in Figure 5-17, all extension operations from component versions are stored in a single list. That list is checked for duplicates before applying included extension operations to the original item, and duplicate extension operations are removed to make sure a component is added only once to the merged version of the component.

On the other hand, semantic redundancy is much harder to detect and resolve. As an example, consider the Person component in Figure 5-19 and the Individual component in Figure 5-20. The only common component that is included both in the Person and in the Individual is the Age. However, there are semantically equivalent components included both in the Individual and in the Person components; namely the Name-FirstName, the Surname-LastName and the Occupation-Job pairs.

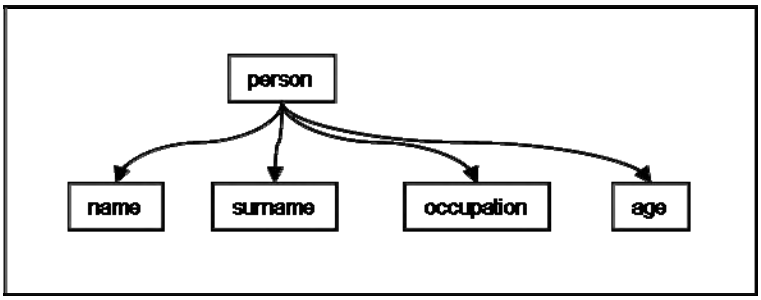

Figure 5-19 – Sample Person component

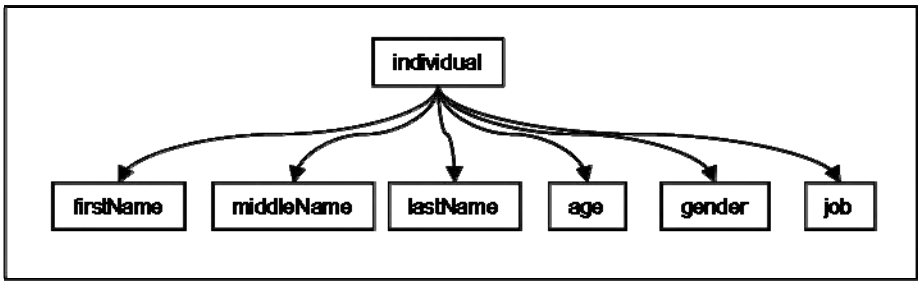

Figure 5-20 – Sample Individual component

Now assume the Component Repository contains two different custom versions for the FinancialInstitution component whose standard UBL version is shown in Figure 5-21.

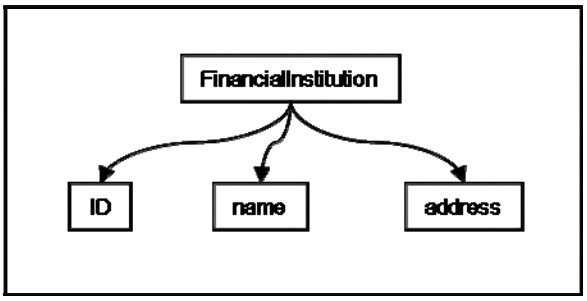

Figure 5-21- UBL FinancialInstitution component

Assume one version is extended for the *Drugs and Pharmaceutical Products* context with the addition of the Person component to generate the version shown in Figure 5-22, and the other version is generated for the *Retail Trade* context with the addition of the Individual component to generate the custom version shown in Figure 5-23.

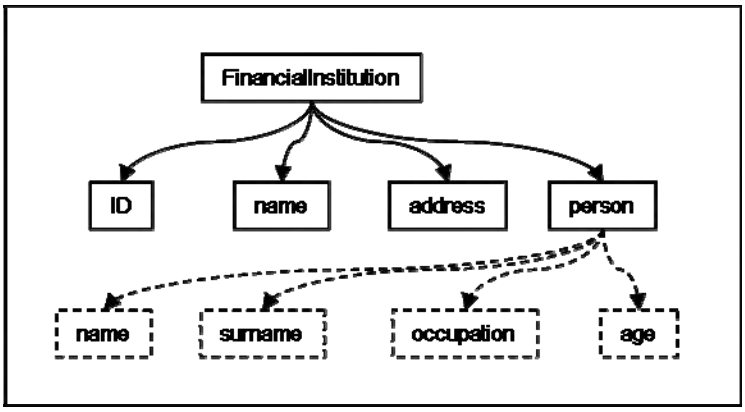

Figure 5-22 – Sample custom version of the FinancialInstitution component in Figure 5-21 extended with Person component in Figure 5-19

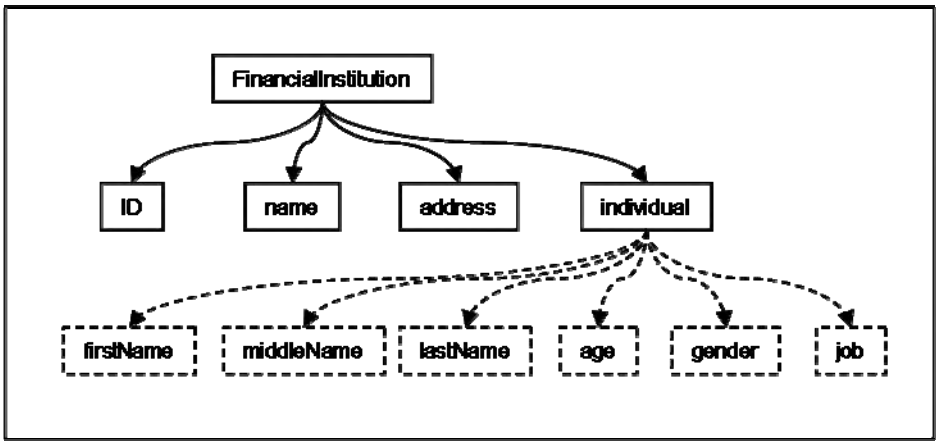

Figure 5-23 - Sample custom version of the FinancialInstitution component in Figure 5-21 extended with Individual component in Figure 5-20

When the applicable version of the FinancialInstitution component is required for the *Retail Trade* and the *Drugs and Pharmaceutical Products* business context, custom versions shown in Figure 5-22 and Figure 5-23 need to be merged. However, as already been discussed, adding both the Person and the Individual components to the merged version introduces redundancy and needs to be avoided.

In order to avoid semantic redundancy, the Component Merge Service utilizes the UBL Component Ontology described in Section 4.2. As discussed in Section 4.2, the UBL Component Ontology is composed of classes representing UBL Components and through reasoning equivalence relationships are computed between classes representing components that consist of semantically equivalent set of elements. Similarly, class-subclass relationships are computed between classes that represent components with subsuming content. Algorithm in Figure 5-24 displays the overall steps of how the semantic redundancy elimination is executed:

```
extensionList : list of extension operations to be checked
                against semantic redundancy

 1. toBeEliminatedList = new List();
 2.
 3. for all extension operation ext1 ∈ extensionList
 4.   for all extension operation ext2 ∈ extensionList
 5.
 6.     //keep only one of the equivalent pairs
 7.     if ext1 ≡ ext2 then
 8.        toBeEliminatedList.add(ext2);
 9.     end if
10.
11.     //keep only sub classes in class-subclass pairs
12.     if ext1 ⊆ ext2 then
13.         toBeEliminatedList.add(ext2);
14.     end if
15.
16.    end for
17. end for
18.
19. extensionList.removeAll(toBeEliminatedList);
20. return extensionList;
```

Figure 5-24 – Semantic Redundancy Elimination Algorithm

Between UBL Component Ontology classes representing the Person and the Individual components in Figure 5-19 and Figure 5-20, a class-subclass relationship is computed. That is, since the set of components included in the Person component is subsumed by the set of components included by the Individual component, the UBL Component Ontology class representing the Person component is computed to be a super-class of the class representing the Individual component.

As a result, when the Semantic Redundancy Elimination algorithm in Figure 5-24 is executed for the list of extension operations including the Person and the Individual components in Figure 5-19 and Figure 5-20, the extension operation for adding the Person component is eliminated in the step 13 of the algorithm and only the Individual component is added to the merged version of the Financial Institution component.

5.3.6 Document Instance Translation Service

Given a UBL document instance and a target business context, the Document Instance Translation Service generates a copy of the given document that conforms to the document schema of the target context and replicates the content of the original document in the generated translation document.

As described in Section 2.1.3, UBL standard provides a set of document schemas in the form of XSD language constructs. Actual UBL Document instances are XML documents structured according to those XSD schemas. With the help of the customization mechanism provided in this work, it is possible to generate custom versions of those schemas tailored for the needs of different business context values and the translation mechanism is designed to support the interoperability of users that structure their business using different versions of UBL schemas.

By definition, every UBL document conforms to a specific UBL document schema. For a given UBL document instance, the scope of the translation is to change the structure of the given document so that it conforms to the corresponding document schema applicable to the target context. It should be noted that the intent of the translation mechanism is not to modify the content of documents, but to adapt the content from one schema to another. In other words, translation mechanism keeps actual values stored in elements of UBL documents intact but structures them according to different schemas required for different context values.

In order to accomplish this, Document Translation Service utilizes the UBL Component Ontology described in Section 4.2, which provides an ontology with classes representing standard and extension UBL components. When the expressions defining those classes are processed by a reasoner, equivalence and class-subclass relationships are computed. Such computed relationships between classes of the UBL Component Ontology indicate the structural and semantic similarities among corresponding UBL components. That is; an equivalence relationship between two UBL Component Ontology classes express that the UBL Components represented by those classes are structurally and semantically equivalent to each other and can be used interchangeably. Similarly, a class-subclass relationship between two UBL Component Ontology classes represent that the UBL Component represented by the child class subsumes the UBL Component represented by the parent class.

Translation works at the component level, that is, in order to translate a document from one schema to another, the element hierarchy of the document instance is traversed in a top-down manner. For every element of the original document, first the corresponding UBL component is gathered. Then the UBL Component Ontology class representing that particular component is located and the corresponding class for the target context is computed. Following that, the UBL component represented by that computed class is gathered and a corresponding element is generated in the translation document.

As an example, consider the greatly simplified Order document schema in Figure 5-25 and the corresponding Order document instance in Figure 5-26.

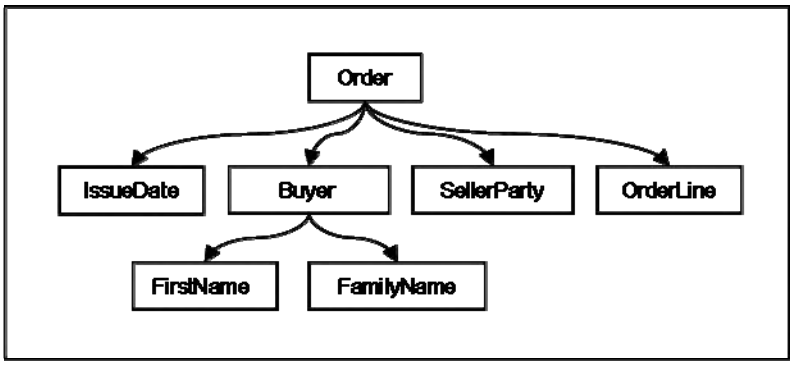

Figure 5-25 – Sample Order schema

```
<Order>
  <IssueDate>31/07/2007</IssueDate>
  <Buyer>
    <FirstName>Asuman</FirstName>
    <FamilyName>Dogac</FamilyName>
  </Buyer>
  <SellerParty>SRDC Ltd.</SellerParty>
  <OrderLine>Windows XP Workstation</OrderLine>
</Order>
```

Figure 5-26 – Sample Order document conforming to the Order schema in Figure 5-25

Now assume it is required to translate the Order document in Figure 5-26 to a different business context for which the Document Schema Customization Service generates the document schema in Figure 5-27.

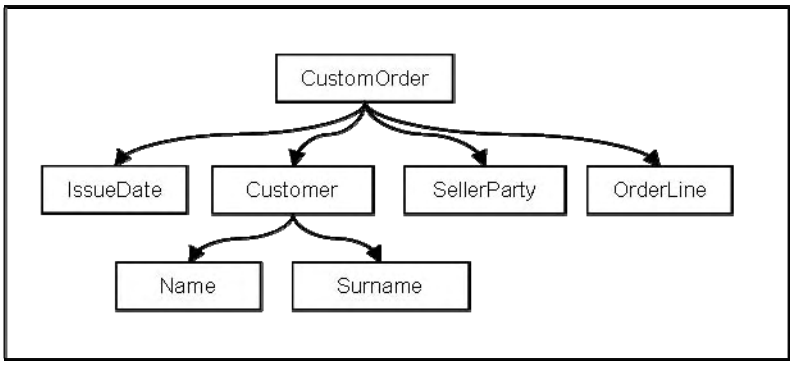

Figure 5-27 – Sample customized version of the document schema in Figure 5-25

Assuming components in Figure 5-25 and Figure 5-27 are defined such that the inferred UBL Component Ontology has the class hierarchy given in Figure 5-28 where multiple classes in a single node express an equivalence relationship between those particular classes.

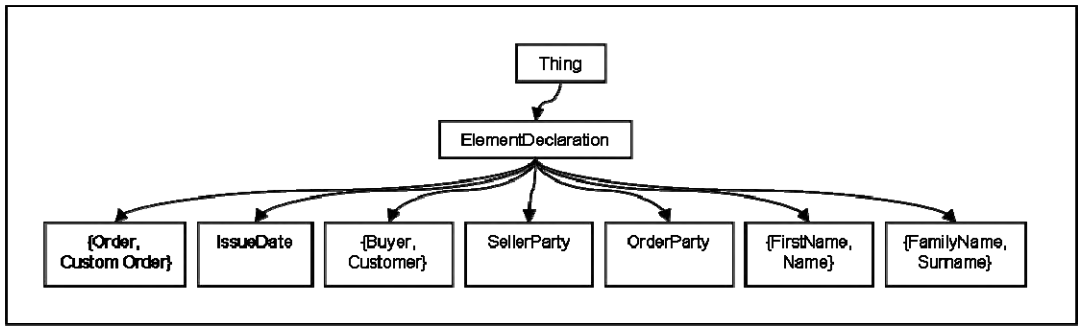

Figure 5-28 – Extract from the inferred UBL Component Ontology class hierarchy representing schemas in Figure 5-25 and Figure 5-27

Using the UBL Component Ontology in Figure 5-28, the Document Translation Service translates the document instance in Figure 5-26 to the document instance in Figure 5-29.

```
<CustomOrder>
  <IssueDate>31/07/2007</IssueDate>
  <Customer>
    <Name>Asuman</Name>
    <Surname>Dogac</Surname>
  </Customer>
  <SellerParty>SRDC Ltd.</SellerParty>
  <OrderLine>Windows XP Workstation</OrderLine>
</CustomOrder>
```

Figure 5-29 – Translated version of the document instance in Figure 5-26

Algorithm in Figure 5-30 displays the overall steps of how the component translation is executed. In order to translate a document instance, the Component Translation algorithm in Figure 5-30 is called for the component corresponding to the root element of that particular document. The Component Translation algorithm traverses the hierarchy of the document by recursively calling itself.

```
currentCmp  : current component to translate from the original
                document
parentCmp   : parent component including the current component
newInstance : the document instance for the translated document
targetCtx   : the context to translate to

 1. targetCmp = determineMatch(currentCmp,parentCmp,targetCtx);
 2.
 3. if (targetCmp ≠ null) then
 4.
 5.   //targetCmp is a match for the currentCmp in the targetCtx
 6.   newInstance.add(targetComponent);
 7.
 8    //recursively call for components included in currentCmp
 9.   childCmpSet = currentCmp.getIncludedComponents();
10.   for all components childCmp ∈ childCmpSet
11.     translateComponent(childCmp, currentCmp, newInstance);
12.   end for
13.
14. else
15.   //there is no match for currentCmp in the target context
16.   //add the component hiearchy to the Extension components
17.  newInstance.addExtensions(currentComponent.getHierarchy());
18. end if
```

Figure 5-30 – Component Translation Algorithm

Details for the determineMatch method referred in step 1 of Figure 5-30 is provided in Figure 5-31 and details for the checkForAMatch method referred in steps 5, 11 and 17 of Figure 5-31 is provided in Figure 5-32.

For a given component, the Component Translation algorithm tries to determine the matching component for the target context. If there is such a component, it is added to the translation document and the Component Translation algorithm is called for each of the components included in the original component. On the other hand, if there is no matching component for the target context, the translation of that particular component is not possible and the component is added to the UBLExtension hierarchy of the translation document. Details of the UBLExtension hierarchy are explained in Section 5.3.6.1.

```
currentCmp     : current component to translate from
                   the original document
parentCmp      : parent component including current component
targetCtx      : the context to translate to
UBLCmpOntology : UBL Component Ontology

 1. currentCls = currentCmp.gatherMatchingClass();
 2.
 3. //first check the set of equivalent classes
 4. equiSet =  UBLCmpOntology.findEquivalents(currentCls);
 5. candidateCmp = checkForAMatch(equiSet,parentCmp,targetCtx);
 6.
 7. if candidateCmp is null then
 8.
 9.   //next check the set of sub-classes
10.   subSet = UBLCmpOntology.findSubClasses(currentCls);
11.   candidateCmp = checkForAMatch(subSet,parentCmp,targetCtx);
12.
13.   if candidateCmp is null then
14.
15.     //finally check the set of super-classes
16.     sprSet = UBLCmpOntology.findSuperClasses(currentCls);
17.     candidateCmp=checkForAMatch(sprSet,parentCmp,targetCtx);
18.   end if
19.
20. end if
21.
22. return candidateCmp;
```

Figure 5-31 – findTargetCmp method in Figure 5-30

When trying to determine the applicable match of a component for a target context, the Component Translation algorithm first checks the set of equivalent classes for the UBL Component Ontology class representing that particular component. If any of the components represented by a class in the set of equivalent classes is a match for the original component, then it is used in the translation document. If none of the classes in the set of equivalent classes provides a match, then the set of sub-classes and then the set of super-classes are checked. If none of those sets provides a match for the original component, it is not possible to translate the original component to the target context.

```
candidateSet   : set of candidate UBL Component Ontology classes
                 to check against the target context and the
                 parent component
parentCmp      : parent component including current component
targetCtx      : the context to translate to

 1. for all classes candidateClass ∈ candidateSet
 2.
 3.   //gather the matching UBL Component for candidate class
 4.   candidateCmp = candidateClass.gatherMatchingComponent();
 5.
 6.   //find the applicable version of the
 7.   //candidateComponent for the target context
 8.   candidateCmp = discoverComponent(candidateCmp, targetCtx);
 9.
10.   //to ensure schema conformity make sure the parent
11.   //component includes the candidate component
12.   if parentCmp.includes(candidateCmp) then
13.     return candidateCmp;
14.   end if
15.
16. end for
17.
18. //none of the components in the given set is
19. //applicable to the target context
21. return null;
```

Figure 5-32 – checkForAMatch method in Figure 5-31

The major constraint of the translation algorithm is to preserve the integrity of document schemas. In other words, the translated version of a document instance has to fully conform to the document schema of the target context. That means, even if the UBL Component Ontology provides an equivalent match for a class, it cannot be used in the translation document unless the corresponding component is conforming with the applicable schema of the target context. In order to ensure this, the Component Translation Algorithm collaborates with the Component Discovery service and for each candidate component computed through the UBL Component Ontology, the version of that particular component applicable to the target context is gathered from the Component Repository. Then the applicable version is checked against the parent component in the target document to ensure that the particular candidate component is included in the schema of the parent component for the target context.

5.3.6.1 Handling Content that can not be Translated Directly

When translating a document from one context to another, it is not always possible to translate all the content to the target context. There may be components defined in the original schema for which no applicable match exists for the target context. Copying such components to translation documents violate the integrity of target schemas and hence is a violation of the UBL specification. Yet the inability to translate some of the content means losing information in the process of translation and undermines the interoperability effort.

In order to provide a solution to this problem, components with no matching peers in target context are added to the UBLExtension hierarchy, a construct specified by the UBL specification as a container for holding any non-schema conformant content. As an example, consider the simplified Catalogue schema in Figure 5-33 and the corresponding sample document in Figure 5-34.

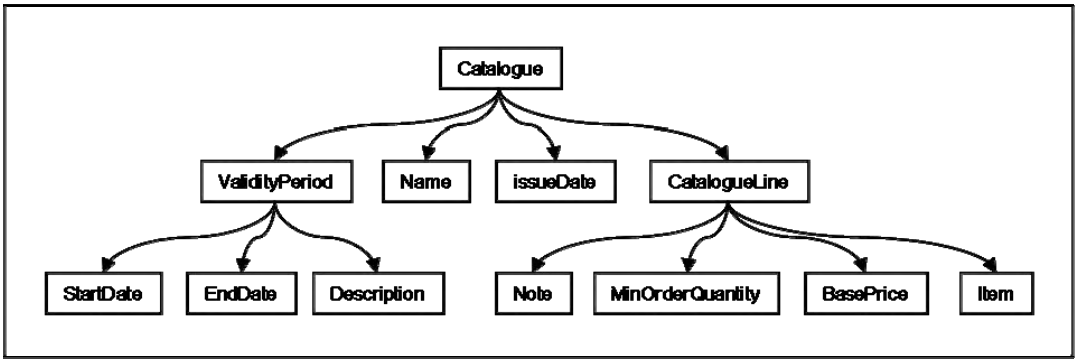

Figure 5-33 – Simplified `Catalogue` schema

```
<Catalogue>
  <ValidityPeriod>
    <StartDate>01/06/2007</StartDate>
    <EndDate>31/08/2007</EndDate>
    <Description>2007 Summer</Description>
  </ValidityPeriod>
  <Name>2007 Summer Catalogue</Name>
  <IssueDate>17/05/2007</IssueDate>
  <CatalogueLine>
    <Note>Just a note</Note>
    <MinOrderQuantity>100</MinOrderQuantity>
    <BasePrice>25 YTL</BasePrice>
    <Item>Swim suits</Item>
  </CatalogueLine>
</Catalogue>
```

Figure 5-34 – Document instance conforming to the Catalogue schema in Figure 5-33

Assume the Catalogue component is customized for a context by removing the ValidityPeriod and the CatalogueLine component is customized for that same context by removing the BasePrice, revealing the Catalogue schema shown in Figure 5-35.

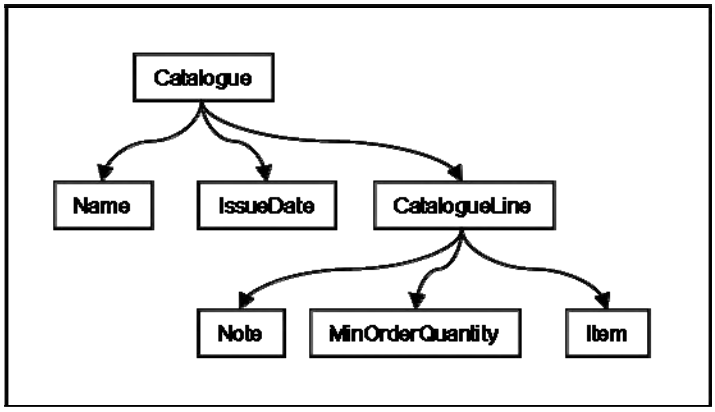

Figure 5-35 – Sample customized version of the Catalogue schema in Figure 5-33

When translation of the document instance in Figure 5-34 to that particular context is required, it is not possible to translate the content of removed components, namely the ValidityPeriod and the BasePrice, to the translation document. Instead, a UBLExtension tag is added to the translation document and all such components are added under that hierarchy, as shown in Figure 5-36. Since UBL Specification does not require schema validation for content located under UBLExtension, this ensures the validity of the translation document in terms of schema conformity. At the same time, the translation document contains all the content of the original document, even those that violate the applicable schema for its context.

```
<Catalogue>
  <UBLExtension>
    <ValidityPeriod>
      <StartDate>01/06/2007</StartDate>
      <EndDate>31/08/2007</EndDate>
      <Description>2007 Summer</Description>
    </ValidityPeriod>
    <CatalogueLine>
      <BasePrice>25 YTL</BasePrice>
   </CatalogueLine>
  </UBLExtension>

  <Name>2007 Summer Catalogue</Name>
  <IssueDate>17/05/2007</IssueDate>
  <CatalogueLine>
    <Note>Just a note</Note>
    <MinOrderQuantity>100</MinOrderQuantity>
    <Item>Swim suits</Item>
  </CatalogueLine>
</Catalogue>
```

Figure 5-36 – Document instance in Figure 5-34 translated to the Catalogue schema in Figure 5-35

CHAPTER 6

IMPLEMENTATION

In order to demonstrate the feasibility and effectiveness of the ideas presented in this thesis, a prototype implementation of the architecture described in Chapter 5 is developed. This chapter introduces that implementation.

The implementation is performed using the Java Standard Edition version 6 [60] using the following third party libraries:

- Apache Xerces [61], which is a family of software packages for parsing, generating and manipulating XML documents. The library implements the standard APIs for XML parsing, including the Document Object Model (DOM) [62], the Simple API for XML (SAX) [63] and SAX2. It also provides an implementation of the XML Schema Language (XSD) specification [11].

- OWL-API [64][65], which provides an open source Java interface and implementation for the OWL. The API is focused towards OWL Lite, OWL DL and OWL 1.1 and offers an interface to inference engines and validation functionality.

- Protégé-OWL API [66], which provides another open-source Java library for OWL. The API provides classes and methods to load and save OWL files, to query and manipulate OWL data models, and to perform reasoning.

- JFormDesigner [67], which provides a GUI designer and libraries to support Swing based GUI development for Java.

- JUnit [68] , a regression testing framework that streamlines the development and execution of unit testing in Java.

For ontology development, alignment and testing, Protégé [46] and the Protégé-OWL plug-in [69] are used extensively. For all reasoning tasks, RacerPro [58] from Racer Technologies is used through an academic license.

6.1 User Applications

This section presents user applications described in Section 5.2. These applications constitute the external interface of the system and allow user interaction with the system.

The Component Customization Tool is used to customize components for context values. Figure 6-1 is a snapshot of the user interface of the Component Customization Tool, in which the aggregate Address component is selected for customization for the isic:F_Construction context value of the *Industrial Classification* domain.

When the Discover button is clicked, the Component Discovery service is called to gather the applicable version of the Address component for the isic:F_Construction context and the gathered version is displayed in the Customization dialog.

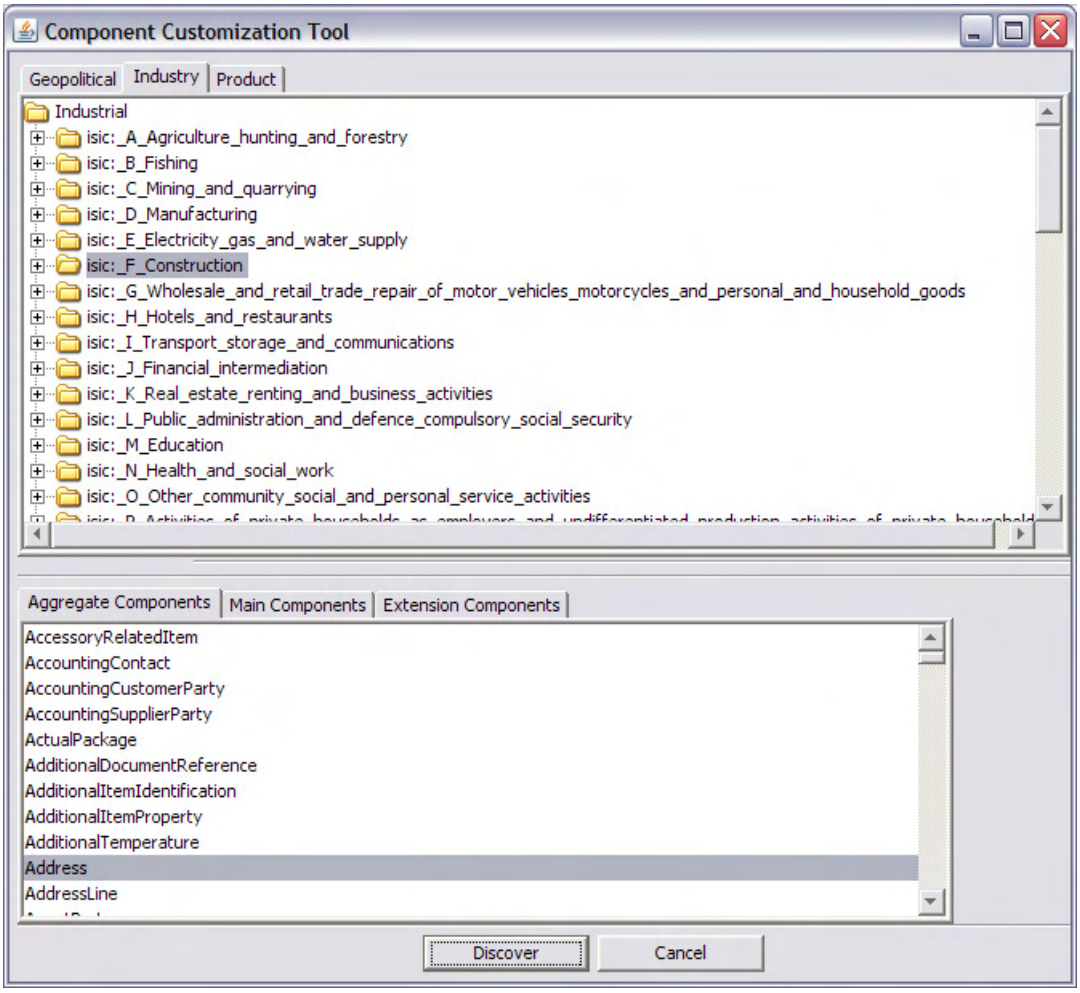

Figure 6-1 - The Component Customization Tool - Context and Component specification

Figure 6-2 displays a snapshot of the Customization dialog of the Component Customization Tool, where components included by the standard UBL Address are shown (denoting the discovery for a custom Address version applicable to the *construction* context did not return a match). Using this dialog, users add and remove components and change the cardinality of already included components.

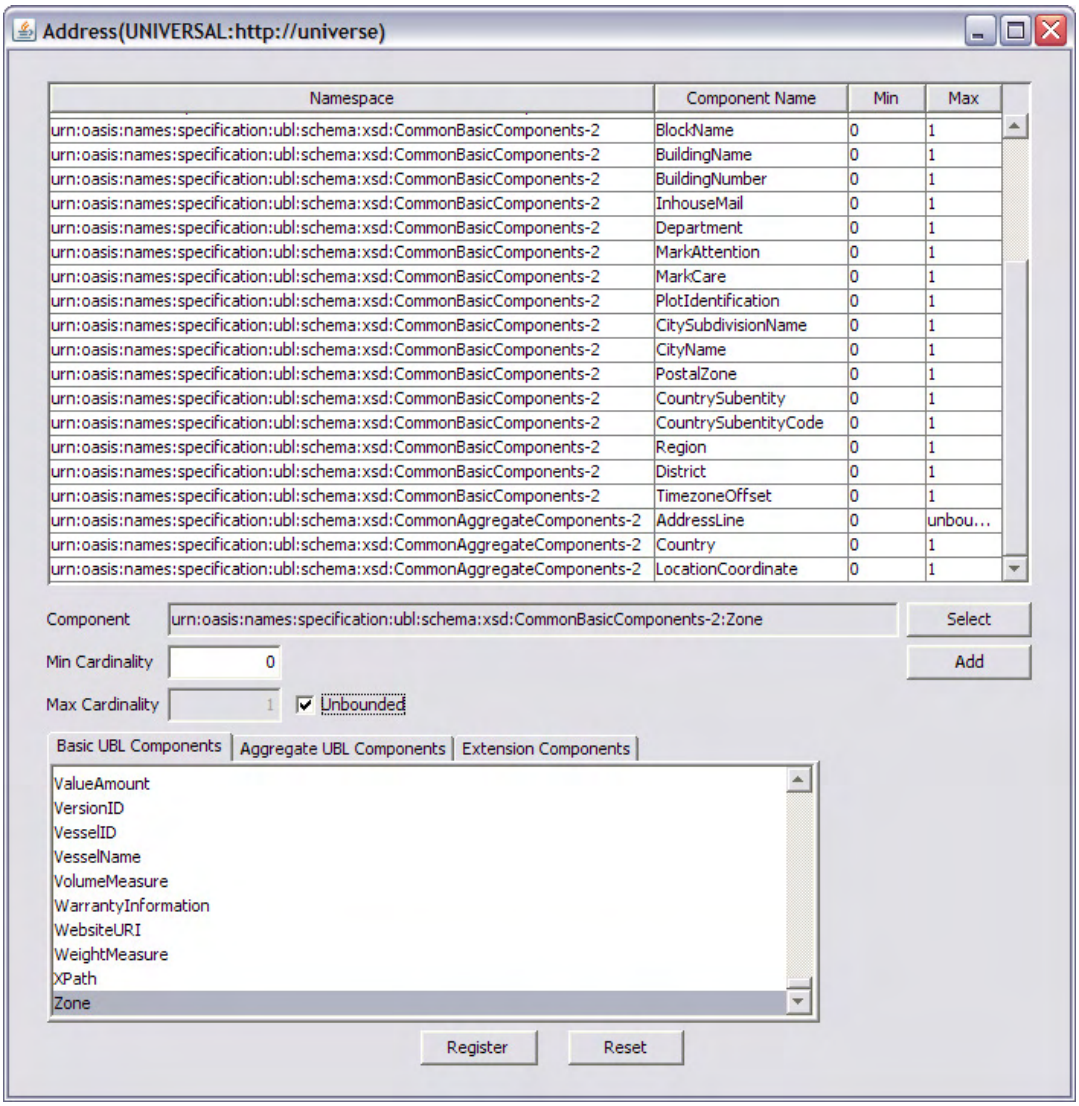

Figure 6-2 – The Component Customization Tool - Customization

The Extension Component Definition Tool is used to create additional components for business entities not covered by the UBL standard. The user interface of the tool consists of three main tabs, namely the *Context*, *Type* and *Component* tabs. Figure 6-3 displays a snapshot of the *Context* tab, which is used to specify the applicable context for the component being defined.

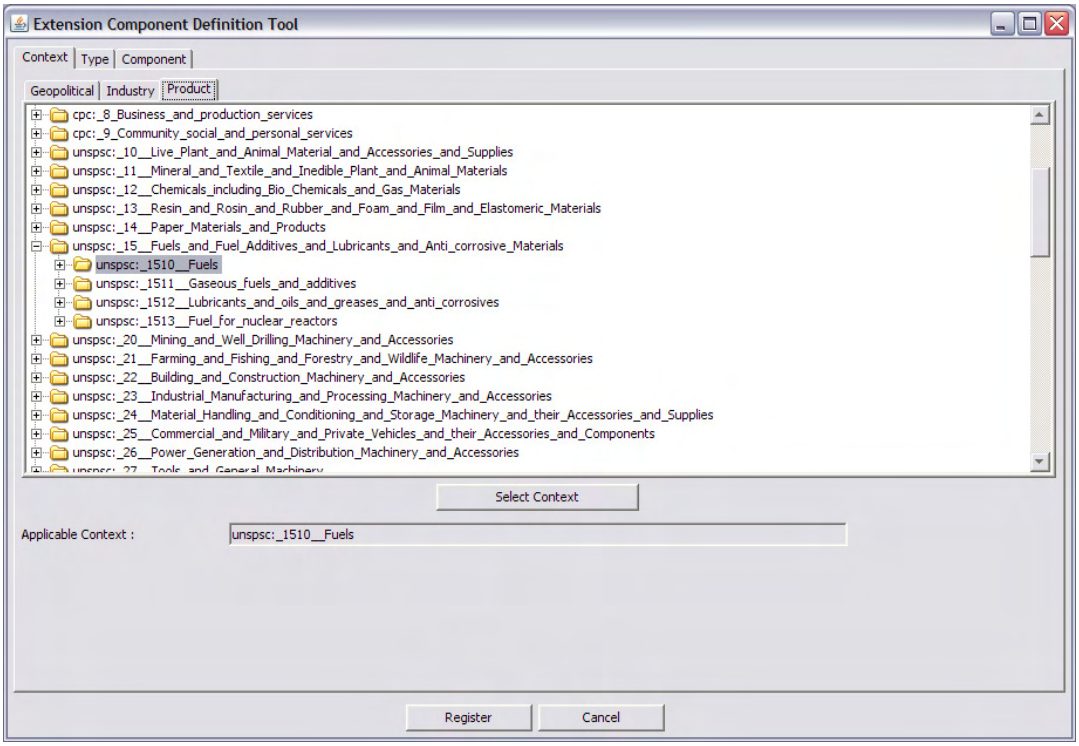

Figure 6-3 – The Extension Component Definition Tool – The Context Tab

The *Type* tab of the Extension Component Definition tool is used to specify type information for the component being defined. The tab itself is further divided into two tabs, *Existing Type* and *New Type*, which are used to specify an existing type or to generate a new type for the component. Figure 6-4 displays a snapshot of the *New Type* tab, where a new type named SampleType consisting of ID, Name and Address components is being defined.

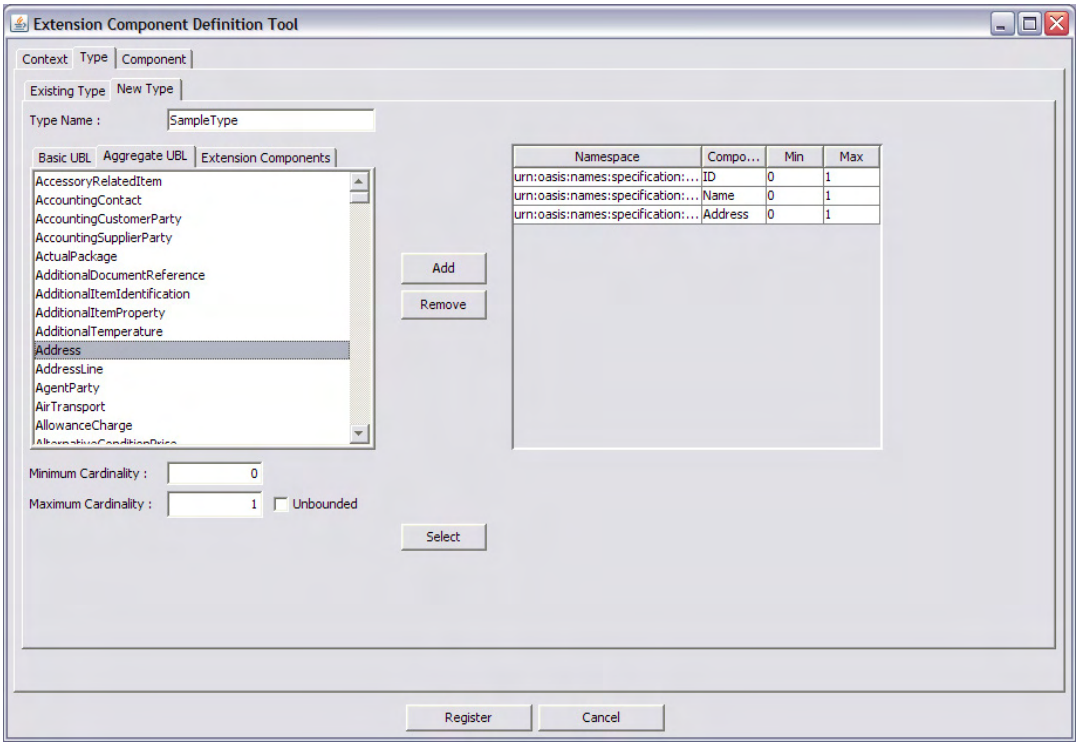

Figure 6-4 - The Extension Component Definition Tool - The Type Tab

The *Component* tab of the Extension Component Definition tool is used to provide component name and to specify the concept for the component being defined. Using the *Component* tab, users can request the generation of a corresponding concept definition for the component or can relate the component to an existing concept. Figure 6-5 displays a snapshot of the *Component* tab, where the component is given the name SampleComponent and generation of a corresponding concept class is requested.

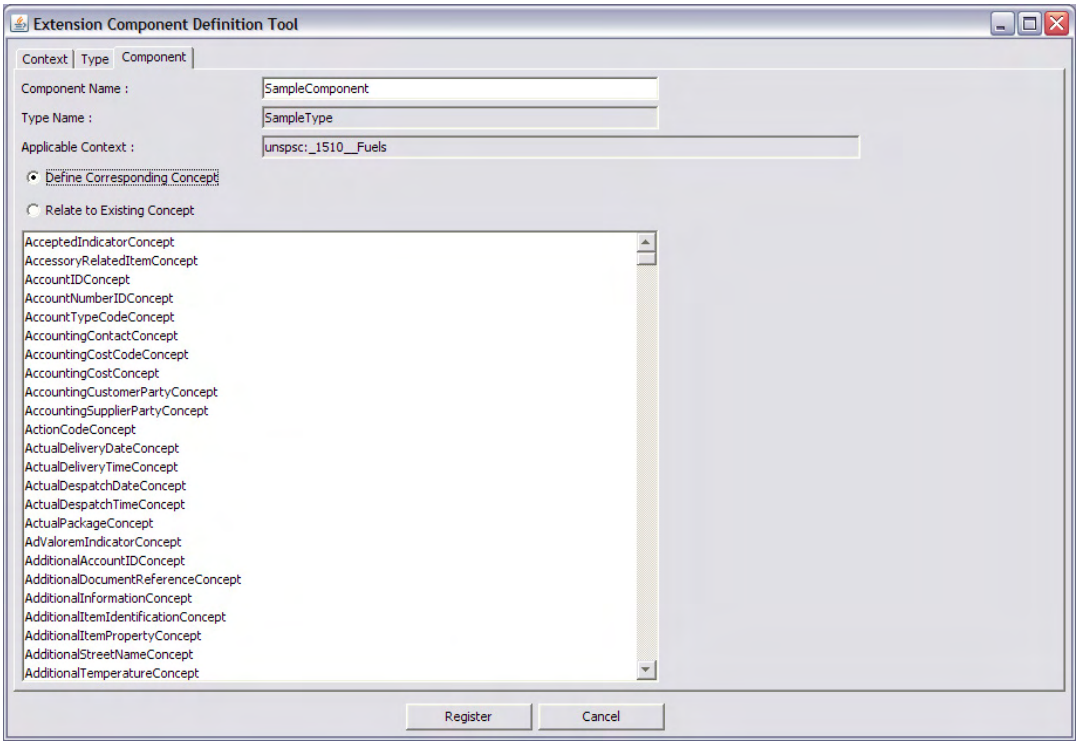

Figure 6-5 - The Extension Component Definition Tool - The Component Tab

Document Schema Customization Tool is used to generate custom versions of UBL document schemas tailored for specific business context values. Using the tool, users specify values for individual context domains to constitute their business context and request the customization of schemas. Figure 6-6 displays a snapshot of the tool, where customization of the Catalogue schema is requested for the *Tools and General Machinery Manufacturing in the North America region* context.

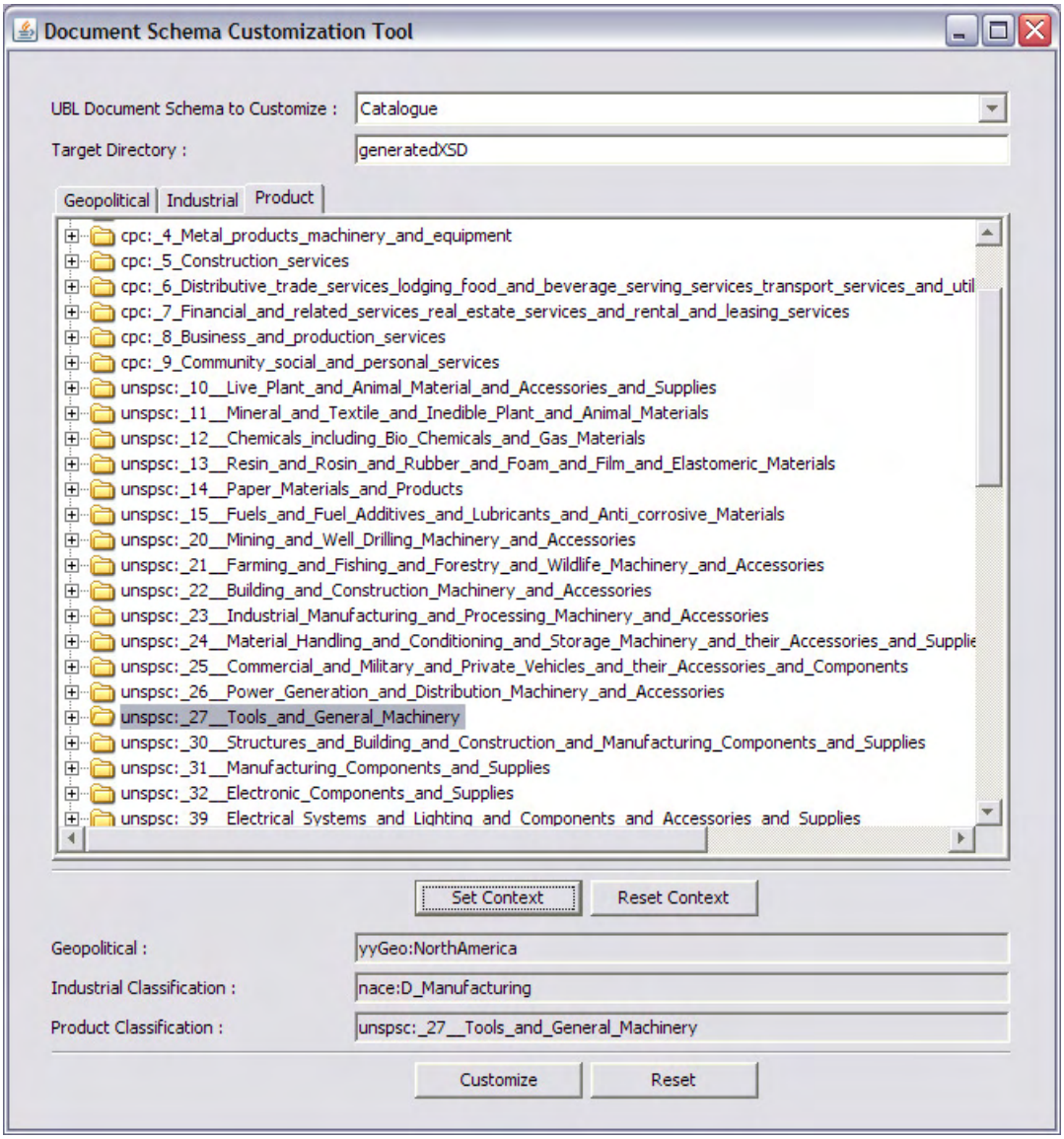

Figure 6-6 - The Document Schema Customization Tool

The Translation Tool is used to generate copies of UBL documents translated to different business context values. Users specify the document to be translated together with the original and the desired context information. Figure 6-7 displays a snapshot of the tool, where translation of a document from the *Manufacturing in the Europe region* context to the *Finance and Insurance in the Asia region* context is requested.

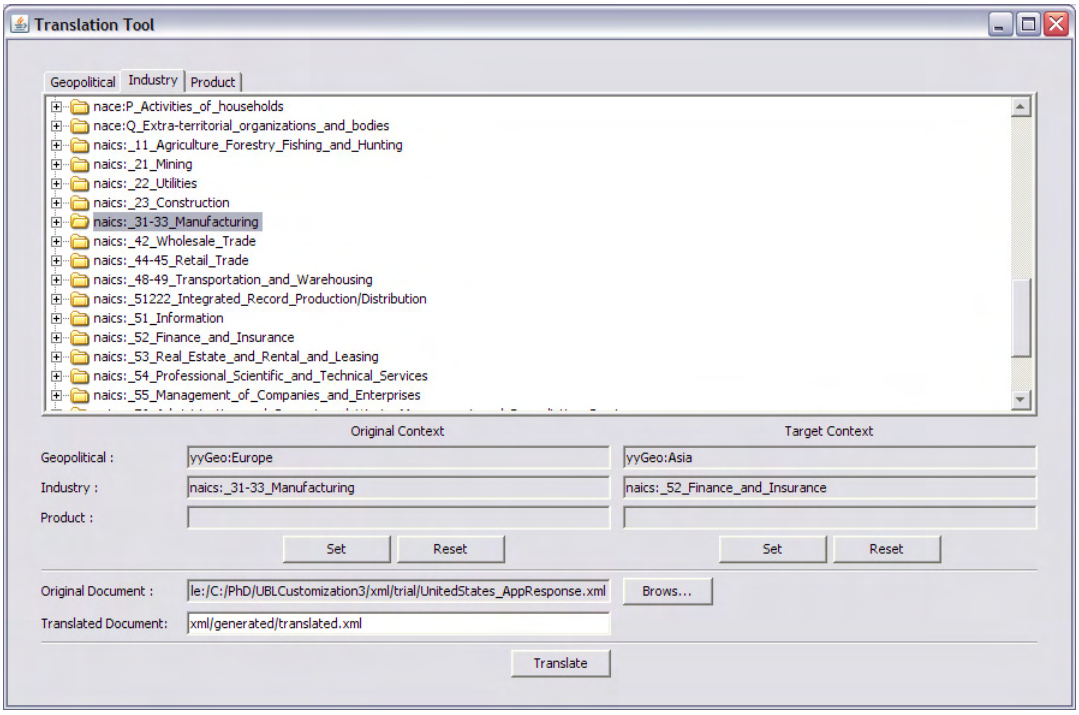

Figure 6-7 - The Translation Tool

6.2 Reasoning Performance

Reasoning performance is an important aspect of the translation process. As described in Section 5.1, whenever users define a new extension component, corresponding classes are automatically generated for the Component Ontology and the ontology is re-processed by the reasoner to compute equivalence and class-subclass relationships between classes.

Figure 6-8 provides performance figures for this computation measured on a Pentium 4 CPU 3.4 GHz Windows workstation with 1GB of RAM. As expected, execution time is a polynomial function of the number of components defined in the ontology, and increases as the number of components defined in the ontology increase.

Nevertheless, considering that the extension component definition is a low frequency operation, possibly in the order of single digits per day, this once per component

penalty is assumed to be within practically acceptable limits even for our modest test setup.

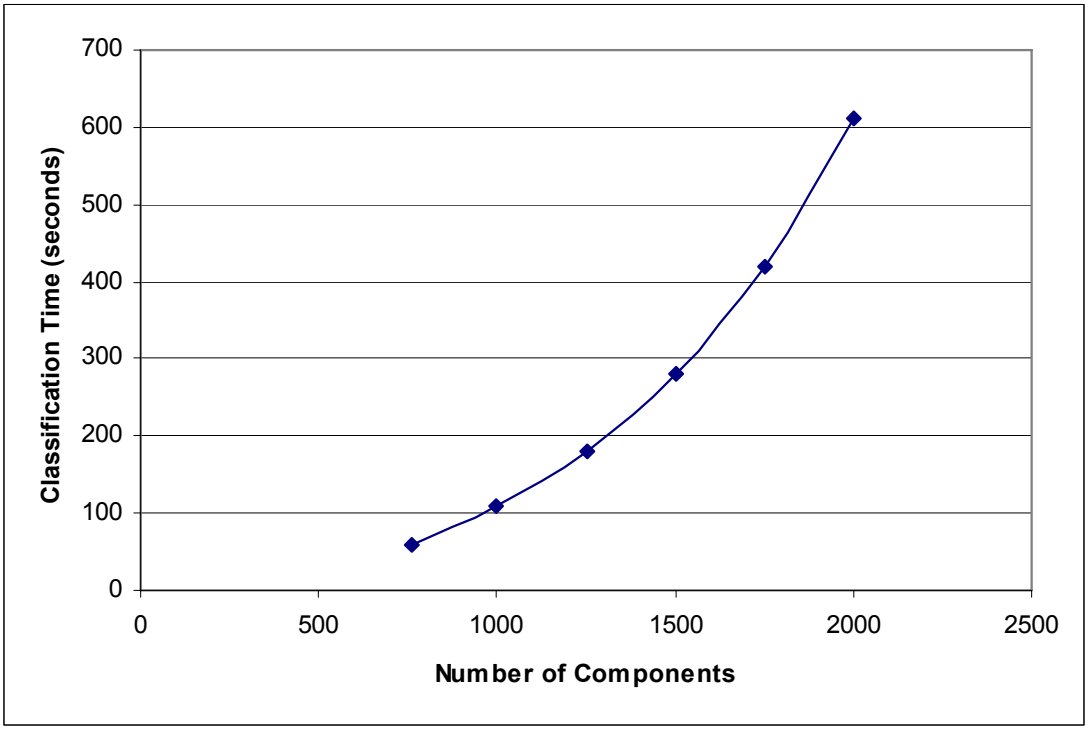

Figure 6-8 - Performance of the Translation Engine for computing equivalence and subsumption relationships between Component Ontology classes

It should be noted that, after this computation all subsequent requests are handled using the inferred ontology until a new component is defined. The actual response time of the translation algorithm using inferred component ontology is in the order of hundred milliseconds.

CHAPTER 7

RELATED WORK

7.1 Context Modeling

As stated in [70], context plays a crucial role in human knowledge representation and reasoning, hence the computer systems which are supposed to act "intelligently" need the ability to represent, utilize and reason about contexts. Authors state that the advent of the web and the ubiquitous connectivity caused context to become a relevant notion in many contemporary application areas such as Information Integration, Distributed Knowledge Management, Semantic Web, Multiagent Systems, Distributed Reasoning, Data Grid and Grid Computing, and Peer to Peer cooperation systems which have acknowledged the need of methods to represent, and reason about, knowledge which is scattered in a large set of local and autonomous inter-related contexts, and have studied various aspects of the context phenomena. Representing and reasoning on context is therefore a research issue spanning several interdisciplinary areas and, in recent years, a number of different aspects of context have been studied, and a number of different approaches to model context representation and reasoning have been proposed.

[71] provides a survey on the subject and classifies existing context modeling approaches based on the schema of data structures they use to exchange contextual information: key value models, markup scheme models, graphical models, object oriented models, logic based models and ontology based models. The work also identifies the requirements that a context modeling approach needs to meet in order to support ubiquitous computing as: distributed composition, partial validation, richness and quality of information, incompleteness and ambiguity, level of formality and applicability to existing environments. Finally a comparative evaluation of the context

modeling approaches based on identified requirements is provided, in which it is concluded that the most promising approaches are ontology based models.

One of the ontology based context modeling approaches is Context Ontology Language (CoOL) [72], which provides a uniform way for specifying the models core concepts as well as an arbitrary amount of sub concepts and facts using ontologies. CoOL is based on an Aspect-Scale-Context (ASC) model where each aspect can have several scales (e.g. kilometer scale or mile scale) to express some context information (e.g. 20). Mapping functions are provided to convert context information from one scale to another. CoOL is considered to be very useful for describing concepts with an inherent metric ordering.

Another ontology based context modeling approach is Context Ontology (CONON) [73], in which Wang et al. present an extensible context ontology for modeling context in pervasive computing environments consisting of a generic upper-ontology and domain specific ontologies for each sub domain to be modeled. The upper-ontology defines entities and concepts like location, user, activity, computational entity which are considered to be most fundamental to represent context information as a part of the work to develop a context aware service infrastructure. Authors claim that due to the evolving nature of context aware computing, completely formalizing all context information is unrealistic and leave context specific details to domain specific ontologies which inherit general classes defined in upper ontologies and are allowed to define whatever sub-classes are required for that particular context.

In [74], Agostini et al. present a specification and an implementation of another ontology based middleware to support context aware service adaptation specifically targeting mobile users. Authors state the following major goals for their middleware: supporting the fusion and reconciliation of context data obtained from distributed sources, supporting context dynamics through an efficient form of reasoning and capturing complex context data that go beyond simple attribute-value pairs. As an example to the latter goal, authors state that in their experimental setup, simple context data about user activity such as the current location (possibly provided by the network operator) and the person user is with (possibly derived through an analysis of user agenda) are defined as assertions to a context ontology which is then processed by a

reasoner to compute complex context data such as whether or not the user is currently engaged in a business meeting.

7.2 Ontology Based Interoperability of Information

Seamless collaboration of distributed and heterogeneous computing resources envisioned by the Semantic Web concept [38], and supporting mechanisms such as the Web Ontology Language triggered a considerable amount of research effort about utilizing ontologies for the establishment of interoperability among distributed and heterogeneous systems.

Firat et al. suggest that applying ontologies in practical semantic interoperability problems has proven to be reducing the amount of work needed to agree on a shared model, to describe the different assumptions made by sources and receivers, and to express (or generate) the mappings required to transform the data when moving it between different sources and receivers [75].

Kasyhyap et al. state that the global interconnectivity fueled by the Internet requires contemporary systems to deal with much more heterogeneous information sources compared to older systems, consisting of a variety of digital data coming from different sources and in different forms such as structured (e.g. Databases), semi-structured (e-mail messages) and unstructured (images) formats [76]. Authors propose an approach that use metadata, context and ontologies to handle semantic interoperability problems and state the two basic components of their approach as using metadata to capture information content of data in the underlying repositories and using terms from domain specific ontologies to characterize contextual descriptions which form the basis of semantic interoperability through relationships between terms across ontologies.

In their survey on approaches to ontology based integration of information [77], Wache et al. state that the information society demands for complete access to available information, which is often heterogeneous and distributed and define the interoperability problem as the problem of bringing together heterogeneous and distributed computer systems. Authors classify problems that might arise due to heterogeneity of the data as structural heterogeneity and semantic heterogeneity, where structural heterogeneity means that different information systems store their data in

different structures and semantic heterogeneity means that content and meaning of an information item is treated differently in different systems. Wache et al. claim that semantic conflicts occur whenever two contexts do not use the same interpretation of the information therefore in order to achieve semantic interoperability in a heterogeneous information system, the meaning of the information that is interchanged has to be understood across the systems.

Goh identifies three main causes for semantic heterogeneity [78] as:

- Confounding conflicts occur when information items seem to have the same meaning, but differ in reality, e.g. owing to different temporal contexts.

- Scaling conflicts occur when different reference systems are used to measure a value. Examples are different currencies.

- Naming conflicts occur when naming schemes of information differ significantly. A frequent phenomenon is the presence of homonyms and synonyms.

Wache et al. state that the use of ontologies for the explication of implicit and hidden knowledge is a possible approach to overcome the problem of semantic heterogeneity [77].

Uschold et al. consider interoperability as a key application of ontologies and state that any information technology environment for business process reengineering or multiagent systems should use integrated enterprise models spanning activities, resources, organization, goals, products and services in which ontologies could be used to support translation between different languages and representations [79].

Wache et al. support this view of Ushold et al. and summarize the major role of ontologies for semantic heterogeneity as follows: “Ontologies are most frequently used in integration tasks to describe the semantics of the information sources and to make the content explicit. With respect to the integration of data sources, they are used for the identification and association of semantically corresponding information concepts by providing a vocabulary for the specification of the semantics of the relevant information systems” [77].

In the ECOIN [75] semantic interoperability framework proposed by Firat et al., ontologies describe both the shared domain model and the ways in which contexts can specialize the shared model by providing a terminology with generic meanings modified in local contexts to express specialized meanings. A context model coupled with the shared model explicitly specifies possible modification dimensions of an ontological term. For example, the meaning of a generic term like airfare can be modified along the currency, coverage, and inclusion dimensions. A context, then, expresses the specific specializations of the shared model that define a given local model which becomes described by the combination of the shared model and a particular context. In the airfare example, for instance, the meaning of airfare objects is made explicit by local sources when they specify the currency used (e.g. USD); and declare whether the coverage is one-way or round-trip and what is included in the airfare (e.g. tax and shipping).

Below is a brief list of example projects and proposals from the literature covering a multitude of sectors in order to demonstrate the versatility of using ontologies for interoperability:

- Smith et al., argue that international efforts towards standardization of biomedical terminology and electronic healthcare records have been focused primarily on syntax, but it is safe to say that the syntactical issues are resolved and it is time for solving the semantic problems relating to biomedical terminology by using ontologies that are able explicitly and unambiguously to relate coding systems, biomedical terminologies and electronic health care records (including their architecture) to the corresponding instances in reality [80]. The Artemis Project [81] is an example to demonstrate the applicability of ontologies to the healthcare domain; where web service messages are annotated using classes from ontologies. These ontologies represent archetypes [82] which are reusable, formal expression of distinct, domain-level concepts such as blood pressure, physical examination, or laboratory result, expressed in the form of constraints on data whose instances conform to some reference information model. Then by providing a mapping between ontologies, Artemis achieves interoperability among web services that operate using different Electronic Healthcare Record standards.

- In [83], Tolksdorf et al. present a coordination middleware to provide semantic support for applications and demonstrate an intelligent Traffic Management System use case to highlight the feasibility of their approach. The middleware coordinates access to a knowledge base for a large number of agents which can be mobile (vehicular) or static (traffic controllers such as traffic lights or message systems), sending and receiving data to and from a central data store. Authors state that the capability of expressing traffic, route data, vehicles, roadways and points of interest unambiguously in terms of a domain ontology permits reasoning over the knowledge base of the traffic management system to deduce information that considerably extends the functionality of a traditional traffic management system such as taking into account restrictions on vehicle type (e.g. that a tractor can not travel on a motorway) and support queries based on reaching some specified type of service (e.g. such as a petrol station) in the most efficient means possible (e.g. inferring when and where traffic conditions are best).

- [84] presents a web services modeling ontology and how it could be applied in the Telecommunication sector to increase the efficiency and reduce the costs of integration among partners of the industry in which Duke et al. claim that benefits are expected in the following areas: improved service discovery, reuse of service interfaces in different products/settings, simpler change management, a browseable-searchable knowledge base for developers and users, semiautomatic service composition, mediation between the data and process requirements of component services to be followed by enterprise level information integration

- Cardoso proposes a semantics based e-Tourism solution which provides dynamic packaging service to customers, where dynamic packaging is defined as the combination of different travel components, bundled and priced in real time from distributed data sources, in response to requests of consumers or booking agents [85]. In order to achieve this, author states that an ontology is developed to represent tourism related information and concepts, then unstructured web sites of different service providers are annotated using classes from this ontology and mediators are developed to provide an integrated view of annotated data.

- In [86], Cuel et al. propose a framework of ontology-driven coordination for the interoperability of processes within organizations, based upon the negotiation of intended meanings.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

Businesses around the world are making significant investments to replace conventional labor and paper based models with computing power and electronic transactions in order to achieve increased automation, more efficient business processes and global visibility. In order to maximize the return for such investments, there is need for standards that streamline the transformation process by making it easier to implement, manage, and improve; adopting common standards reduce development and maintenance costs, improve performance, and enhance business relationships.

UBL specification provides a set of standard electronic business document schemas such as purchase orders and invoices in an effort to standardize electronic business documents and is being adapted by large scale organizations around the world. By adapting UBL standards, involved parties can understand the semantics of exchanged data and the content and types of business documents and can benefit from the experience, know-how and tools developed within the UBL community.

Nevertheless, experience with other electronic business standards has proven that, it is not feasible for a single document schema to meet the needs of all forms of businesses around the world. Instead, there is need for easy to use and inexpensive mechanisms that can be used to tailor document schemas according to specific business needs and clearly such a mechanism should be able to preserve the interoperability achieved by adapting common standards.

Build on the experience of past standards, UBL acknowledges this customization requirement and provides a solution which is mostly concerned with maintaining

syntactic interoperability. That is, by following the customization mechanism provided by UBL, businesses can generate custom versions that can still be processed by software developed to process the standard set of UBL documents. Even though this syntactic interoperability mechanism is a valuable asset, it is not fulfilling important problems such as how users can discover and re-use customizations provided by others and more importantly how the interoperability among different custom versions is to be achieved.

In the work presented by this thesis, we build upon the work provided by UBL and improve it by providing necessary constructs to ensure semantic interoperability among the UBL community.

As customizations are triggered in response to different contextual needs, we start by developing a machine processable representation for context domains using a formal ontology language. In order for context representations to be open and extendible, we allow the development of multiple ontologies for a single domain and describe how correspondences among different ontologies can be established using ontology alignment operations. Then we process these ontologies using a reasoner and compute implicit relationships based on asserted relationships. As the next step, we define a knowledge base to describe custom component versions and annotate components using classes from context ontologies. Together with the machine processable context formalization, this knowledge base allows the automation of several customization related tasks such as the discovery of custom versions provided by other users, automatic merge of multiple custom versions to generate unified representations and finally customization of document schemas for business context values by replacing custom component versions applicable for those context values in place of their standard counterparts.

As the next stage of our work, we provide a semantic translation mechanism to establish interoperability among users adapting different customizations of document schemas. For that purpose, we developed a component ontology consisting of expressions that define the distinguishing characteristics of UBL constructs. We then show that when an ontology reasoner interprets those expressions, based on similarities among different components, it computes equivalence and class-subclass relationships between ontology classes representing such similar components. Finally we present a

translation algorithm, which by exploiting these computed relationships between component ontology classes, is capable of translating UBL documents between schema versions customized for different business context values.

The UBL is structured as an extensible component library and allows both the introduction of new components and the assembly of components to generate new document schemas. Based on the approach presented in this study, the iSURF Project (European Commission, Project No: ICT- 213031) exploits this component library structure to develop representations for documents and messages defined by other (non-UBL) e-business standards. With the capability to translate between different schema versions presented in this work, the project aims to provide interoperability at a much higher level; that is among systems that are based on different e-business standards.

REFERENCES

[1] Lucking-Reiley, D., Spulber, D.F., "Business-to-Business Electronic Commerce". The Journal of Economic Perspectives, Vol. 15, No. 1 (Winter, 2001), pp. 55-68.

[2] Dogac A., Guest Editor Special issue of Journal of Distributed and Parallel Databases on Electronic Commerce Vol. 7, No. 2, April 1999

[3] Gartner Group, "The Economic Downturn is not an excuse to retrench B2B efforts", 2001, http://www.intimex.com/email/3/email_3e.htm; (last accessed on March 2, 2008).

[4] Medjahed B., Benatallah B., Bouguettaya A., Ngu A.H.H., Elmagarmid A.K., "Businessto-business interactions: Issues and Enabling Technologies", *The International Journal on Very Large Data Bases*, Vol. 12, No. 1, May 2003, pp 5985.

[5] The Universal Business Language (UBL) standard v2.0, http://docs.oasis-open.org/ubl/cs-UBL-2.0/UBL-2.0.html; (last accessed on March 2, 2008).

[6] Organization for the Advancement of Structured Information Standards (OASIS), http:// www.oasis-open.org; (last accessed on March 2, 2008).

[7] Adoption of UBL in Denmark - Business Cases and Experiences, http://www.idealliance.org/proceedings/xtech05/papers/03-05-02; (last accessed on March 2, 2008).

[8] Svefaktura (SwedInvoice), http://www.svefaktura.se/SFTIBasicInvoice20051130 EN/index.html; (last accessed on March 2, 2008).

[9] The Electronics Freight Management White Paper, http://www.itsdocs.fhwa.dot.gov/JPODOCS/REPTSTE/14246 files/14246.pdf; (last accessed on March 2, 2008).

[10] Gertner M., Gutentag E., Gregory A., “Guidelines For The Customization of UBL v1.0 Schemas”, http://docs.oasis-open.org/ubl/cd-UBL-1.0/doc/cm/wd-ubl-cmsc-cmguidelines-1.0.html; (last accessed on March 2, 2008).

[11] W3C XML Schema Language, http://www.w3.org/XML/Schema; (last accessed on March 2, 2008).

[12] Damodaran, S., "B2B Integration over the Internet with XML: RosettaNet Successes and Challenges." Pages 188-195, In Proceedings of the Thirteenth World Wide Web Conference (WWW 2004) held in New York City, May 17-22, 2004

[13] OWL Web Ontology Language 1.0 Reference http://www.w3.org/TR/2002/WD-owl-ref-20020729/ref-daml; (last accessed on March 2, 2008).

[14] International Standard Industrial Classification of All Economic Activities (ISIC), Revision 3.1, http://unstats.un.org/unsd/cr/family2.asp?Cl=17; (last accessed on March 2, 2008).

[15] North American Industry Classification System (NAICS), http://www.census.gov/epcd/www/naics.html; (last accessed on March 2, 2008).

[16] Statistical Classification of Economic Activities in the European Community (NACE), Rev. 1.1, http://ec.europa.eu/comm/eurostat/ramon; (last accessed on March 2, 2008).

[17] United Nations Directories for Electronic Data Interchange for Administration, Commerce and Transport (UN/EDIFACT), http://www.unece.org/trade/untdid/welcome.htm; (last accessed on March 2, 2008).

[18] ANSI Accredited Standards Committee (ASC) X12, http://www.x12.org; (last accessed on March 2, 2008).

[19] RosettaNet, http://www.rosettanet.com; (last accessed on March 2, 2008).

[20] XML Common Business Library (xCBL), http://www.xcbl.org; (last accessed on March 2, 2008).

[21] Electronic Business using eXtensible Markup Language (ebXML), http://www.ebxml.org; (last accessed on March 2, 2008).

[22] United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT), http://www.unece.org/cefact; (last accessed on March 2, 2008).

[23] Simple Object Access Protocol (SOAP) Specification v1.2, http://www.w3.org/TR/soap12; (last accessed on March 2, 2008).

[24] UN/CEFACT ebXML Core Components Technical Specification, http://www.unece.org/cefact/ebxml/CCTS V2-01 Final.pdf; (last accessed on March 2, 2008).

[25] Rawlins, M. C, "The ebXML Core Components", http://www.rawlinsecconsulting.com/ebXML/ebXML7.html; (last accessed on March 2, 2008).

[26] The ebXML Catalogue of Common Business Processes Specification, http://www.ebxml.org/specs/bpPROC.pdf; (last accessed on March 2, 2008).

[27] Gertner M., "UBL: The Next Step for Global E-Commerce", http://oasis-open.org/committees/ubl/msc/200112/ubl.pdf; (last accessed on March 2, 2008).

[28] Holman G. K., "Practical Universal Business Language Deployment", First Edition - ISBN 978-1-894049-16-0, 2006.http://www.cranesoftwrights.com/training/ - Sect4.3; (last accessed on March 2, 2008).

[29] Gregory A., Gutentag E., "UBL and Object Oriented XML: Making Type Aware Systems Work", http://www.idealliance.org/papers/dx_xml03/papers/04-04-04/04-04-04.pdf; (last accessed on March 2, 2008).

[30] Genesereth, M. R., Nilsson, N. J., "Logical Foundations of Artificial Intelligence." San Mateo, CA, Morgan Kaufmann Publishers, 1987.

[31] Gruber, T.R.., "Toward Principles for the Design of Ontologies Used for Knowledge Sharing", Int. Journal of Human-Computer Studies, Vol. 43, pp.907-92, 1995.

[32] Guarino N., "Formal Ontology in Information Systems." Proceedings of FOIS'98, Trento, Italy, 6-8 June 1998. Amsterdam, IOS Press, pp. 3-15.

[33] Noy N.F., McGuinness D.L., "A Guide to Creating Your First Ontology". Standford Knowledge Systems Laboratory Technical Report, March 2001.

[34] Jasper, R., Uschold, M., “A Framework for Understanding and Classifying Ontology Applications”. in IJCAI-99 Ontology Workshop. Stockholm, Sweden, July, 1999

[35] Horrocks, I., Patel-Schneider, P. F., Van Harmelen, F., “From SHIQ and RDF to OWL: The Making of a Web Ontology Language”, Journal of Web Semantics, 1(1):7-26, 2003.

[36] Heflin, J., “OWL Web Ontology Language Use Cases and Requirements”, http://www.w3.org/TR/webont-req; (last accessed on March 2, 2008).

[37] McGuinness, D., Harmelen, F., “OWL Web Ontology Language Overview”, W3C Recommendation, February 2004, http://www.w3.org/TR/owl-features; (last accessed on March 2, 2008).

[38] Berners-Lee, T., Hendler, J., Lassila, O., “The Semantic Web” Scientific American, May 2001, pp. 34–43.

[39] Manola F., Miller E., “The Resource Description Framework (RDF) Primer”, http://www.w3.org/TR/rdf-primer; (last accessed on March 2, 2008).

[40] Smith, M. K., Welty, C., McGuiness, D. L., “OWL-Web Ontology Language Guide”, http://www.w3.org/TR/owl-guide; (last accessed on March 2, 2008).

[41] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., Patel-Schneider, P.F., editors, “The Description Logic Handbook: Theory, Implementation and Application” Cambridge University Press, 2003, ISBN 0-521-78176-0.

[42] Staab, S., Studer, R., Handbook on Ontologies, Springer, 2004.

[43] Universal Standard Products and Services Classification (UNSPSC), http://www.unspsc.org; (last accessed on March 2, 2008).

[44] Fridman Noy, N. and Musen, M. A. 1999. An Algorithm for Merging and Aligning Ontologies: Automation and Tool Support. *In Proceedings of the Workshop on Ontology Management at the Sixteenth National Conference on Artificial Intelligence (AAAI-99).* Orlando, FL: AAAI Press.

[45] McGuinness, D. L., Fikes, R., Rice, J., Wilder, S., "An Environment for Merging and Testing Large Ontologies." Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000). Breckenridge, Colorado, USA. April 12-15, 2000.

[46] Protege Ontology Editor and Knowledge Acquisition System, http://protege.stanford.edu; (last accessed on March 2, 2008).

[47] Prompt plugin for Protege, http://protege.cim3.net/cgi-bin/wiki.pl?Prompt; (last accessed on March 2, 2008).

[48] Swoop, The OWL Ontology browser and editor from the University of Maryland, http://www.mindswap.org/2004/SWOOP; (last accessed on March 2, 2008).

[49] McGuinness, D. L., Fikes, R., Rice J., Wilder, S., "The Chimaera Ontology Environment.", Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000). Austin, Texas. July 30 - August 3, 2000.

[50] Statistical Classification of Economic Activities, Complete List and Correspondending ISIC Classes, http://www.fifoost.org/database/nace/naceen 2002c.php; (last accessed on March 2, 2008).

[51] North American Product Classification System (NAPCS), http://www.census.gov/eos/www/napcs/napcs.htm; (last accessed on March 2, 2008).

[52] Statistical Classification of Products by Activity in the European Economic Community (CPA), 2002 version http://ec.europa.eu/comm/eurostat/ramon; (last accessed on March 2, 2008).

[53] Central Product Classification Version 1.1, http://unstats.un.org/unsd/cr/family2.asp?Cl=16; (last accessed on March 2, 2008).

[54] International Organization for Standardization, ISO 3166 Maintenance Agency http://www.iso.org/iso/en/prods-services/iso3166ma/index.html; (last accessed on March 2, 2008).

[55] Schematron, http://www.schematron.com; (last accessed on March 2, 2008).

[56] OWL Web Ontology Language Guide, http://www.w3.org/TR/2004/REC-owl-guide-20040210; (last accessed on March 2, 2008).

[57] Horrocks, I., Scheider, P. F. P., “Optimising Description Logic Subsumption”, Journal of Logic and Computation, 9:3, June 1999, pages 267-293.

[58] RacerPro: Renamed ABox and Concept Expression Reasoner, http://www.racer-systems.com/products/racerpro/index.phtml; (last accessed on March 2, 2008).

[59] Haarslev, V., Moller, R., “Description of the RACER System and its Applications.”, Working Notes of the 2001 International Description Logics Workshop (DL-2001), Stanford, CA, USA, August 1-3, 2001

[60] Java Programming Language, http://java.sun.com; (last accessed on March 2, 2008).

[61] Apache Xerces, http://xerces.apache.org; (last accessed on March 2, 2008).

[62] W3C Document Object Model (DOM), http://www.w3.org/DOM; (last accessed on March 2, 2008).

[63] The Simple API for XML (SAX), http://www.saxproject.org; (last accessed on March 2, 2008).

[64] OWL API, http://owl.man.ac.uk/api.shtml; (last accessed on March 2, 2008).

[65] Bechhofer, S., Lord, P., Volz, R., "Cooking the Semantic Web with the OWL API". 2nd International Semantic Web Conference, ISWC, Sanibel Island, Florida, October 2003

[66] Protégé OWL API, http://protege.stanford.edu/plugins/owl/api; (last accessed on March 2, 2008).

[67] Java Swing GUI Designer, JFormDesigner, http://www.jformdesigner.com; (last accessed on March 2, 2008).

[68] JUnit, Testing Resources for Extreme Programming, http://www.junit.org; (last accessed on March 2, 2008).

[69] The Protégé-OWL Editor, http://protege.stanford.edu/overview/protege-owl.html; (last accessed on March 2, 2008).

[70] Motivations, 2nd International Workshop on Context Representation and Reasoning 2006, http://sra.itc.it/events/crr06; (last accessed on March 2, 2008).

[71] Strang T., Linnhoff-Popien C., “A Context Modeling Survey”, Workshop on Advanced Context Modelling, Reasoning and Management associated with the Sixth International Conference on Ubiquitous Computing, 2004.

[72] Strang T., Linnhoff-Popien C., Frank K., “CoOL: A Context Ontology Language to enable Contextual Interoperability”, LNCS Distributed Applications and Interoperable Systems, Vol. 2893, 2003, pp 236-247.

[73] Wang X.H., Zhang D.Q., Gu T., Pung H.K., “Ontology based context modeling and reasoning using OWL”, Context Modeling and Reasoning Workshop at PerCom 2004, pp 18-22.

[74] Agostini, A., Bettini, C., Riboni, D., "Online Ontological Reasoning for Context-Aware Internet Services". In Proceedings of the Workshop on Context and Ontologies: Theory, Practice and Applications (in conjunction with ECAI 2006), pp. 77-78. Printed by Università di Trento, 2006

[75] Firat, A., Madnick, S., Manola, F., "Multi-dimensional Ontology Views via Contexts in the ECOIN Semantic Interoperability Framework", In Contexts and Ontologies: Theory, Practice and Applications: Papers from the 2005 AAAI Workshop, pp 1-8. Technical Report WS-05-01. American Association for Artificial Intelligence, Menlo Park, California.

[76] Kashyap, V., Sheth, A., “Semantic heterogeneity in global information systems: The role of metadata, context and ontologies”, Cooperative Information Systems, 1998, pp 139-178.

[77] Wache, H., Vögele, T., Visser, U., Stuckenschmidt, H., Schuster, G., Neumann, H., Hubner, S. 2001. “Ontology-based integration of information - a survey of existing approaches”. In Stuckenschmidt, H., ed., IJCAI-01 Workshop: Ontologies and Information Sharing, 108—117.

[78] Goh, C., H., “Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Sources”. Phd, MIT, 1997.

[79] Uschold, M., Gruninger, M., “Ontologies: Principles, methods and applications.” Knowledge Engineering Review, 11(2):93–155, 1996.

[80] Smith, B., Ceusters, W., “Ontology as the Core Discipline of Biomedical Informatics”, http://ontology.buffalo.edu/medo/Recommendations_2005.pdf; (last accessed on March 2, 2008).

[81] The Artemis Project, http://www.srdc.metu.edu.tr/webpage/projects/artemis; (last accessed on March 2, 2008).

[82] Beale, T., Heard, S. “Archetype definitions and principles”, 2003. Australia: Ocean Informatics, the OpenEHR Foundation.

[83] Tolksdorf, R., Nixon, L. J.B., Bontas, E. P., Liebsch, F., Nguyen, D. M., “Enabling Real World Semantic Web applications Through a Coordination Middleware”, In Proceedings of the 2nd European Semantic Web Conference, LNCS 3532, pages 679–693. Springer-Verlag, 2005.

[84] Duke, A., Richardson, M., Watkins, S., Roberts, M., “Towards B2B Integration in Telecommunications with Semantic Web Services”, In Proceedings of the 2nd European Semantic Web Conference, LNCS 3532, pages 710–724. Springer-Verlag, 2005.

[85] Cardoso, J., “E-Tourism: Creating Dynamic Packages using Semantic Web Processes”, W3C Workshop on Frameworks for Semantics in Web Services, Innsbuck, Austria, 2005.

[86] Cuel R., Cristani M., "Ontologies as Intra-Organizational Coordination Tools". In: 5th International Conference on Knowledge Management: Graz, Austria, June 29 - July 1, 2005.

[87] Donini F. M., Lenzerini M., Nardi D., Nutt W., “Reasoning in Description Logics”. In G. Brewka, editor, Foundation of Knowledge Representation. CSLI Publication, Cambridge University Press, 1996.

[88] Horrocks I., Sattler U., Tobies S., “Practical Reasoning for Expressive Description Logics”, In Proceedings of LPAR'99, LNCS, Tbilisi, Georgia, 1999. Springer-Verlag, Berlin.

[89] Horrocks I., Sattler U., “A Tableaux decision procedure for SHOIQ”, Journal of Automated Reasoning, Vol 39, Number 3, pages 249-276, 2007.

[90] Baader F., Sattler U., “An overview of tableau algorithms for description logics”, Studia Logica 69 (2001), 5-40.

[91] Schmidt M. S., Smolka G., “Attributive concept descriptions with complements”, Artificial Intelligence, 48(1):1-26, 1991.

APPENDIX A

UBL CUSTOMIZATION METHODOLOGY

This Appendix provides a brief summary of the UBL Customization Methodology [10].

UBL starts as generic as possible, with a set of schemas that supply all that are likely to be needed in the 80/20 or core case, which is UBL's primary target. Then as displayed in Figure A-1, it allows both sub setting and extension according to the needs of user communities, industries, nations, etc., using the XSD derivation mechanism.

These customizations are based on the eight context drivers identified by the ebXML [24]. Any given schema component always occupies a location in this eight-space, even if not a single one has been identified (that is, if a given context driver has not been narrowed, it means that it is true for all its possible contextual values). For instance, UBL has an Address type that may have to be modified if the Geopolitical region in which it will be used is Thailand. But as long as this narrowing down of the Geopolitical context has not been done, the Address type applies to all possible values of it, thus occupying the "any" position in this particular axis of the eight-space.

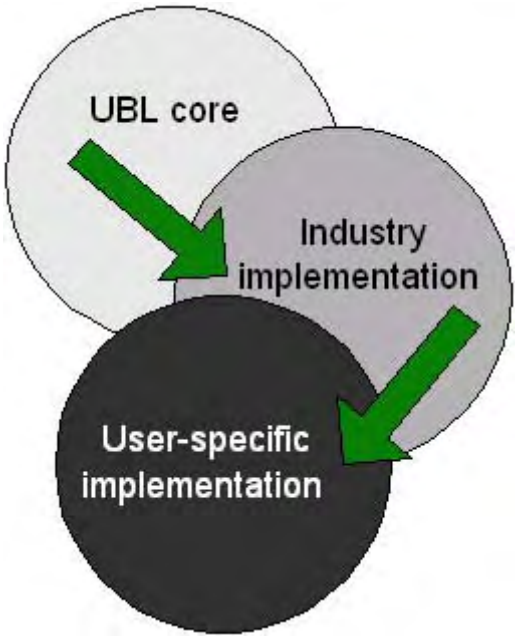

Figure A-1 – The UBL Customization Flow

It is assumed that in many cases specific businesses will use customized UBL schemas. These customized schemas contain derivations of the UBL types, created through additional restrictions and/or extensions to fit more precisely the requirements of a given class of UBL users. The customized UBL Schemas may then be used by specific organizations within an industry to create their own customized schemas.

Due to the extensibility of XSD Schema, this process can be applied over and over to refine a set of schemas more and more precisely, depending on the needs of specific data flows. As such, there is a risk that derivations may form extremely long and unmanageable chains. In order to avoid this problem, the Rule of Once-per-Context was formulated: no context can be applied, at a given hierarchical level of that context, more than once in a chain of derivations. Or, in other words, any given context driver can be specialized, but not reset.

As seen in Figure A-2, if the Geopolitical context driver with a value of "USA" has been applied to a type, it is possible to apply it again with a value that is a subset, or

that occupies a hierarchically lower level than that of the original value, like California or New York, but it cannot be applied with a value equal or higher in the hierarchy, like Japan. In order to use that latter value, one must go up the ladder of the customization chain and derive the type from the same location as that from which the original was derived.

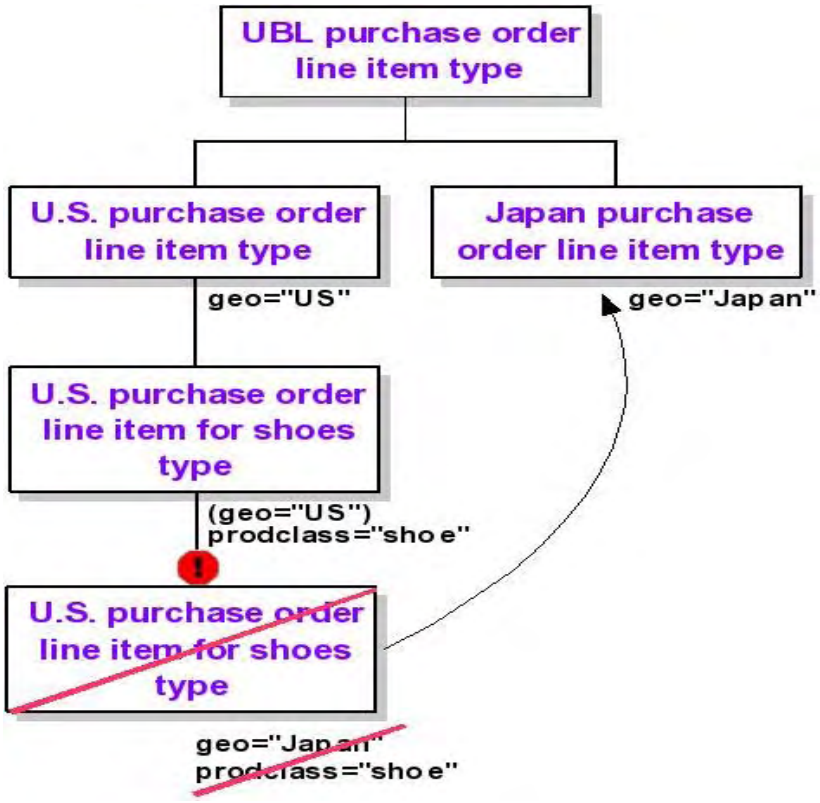

Figure A-2 – An example UBL Context Chain

XSD derivation allows for type extension and restriction. These are the only means by which one can customize UBL schemas and claim UBL compatibility. Any other possible means, even if allowed by XSD itself, is not allowed by UBL. For instance,

although XSD does permit the redefinition of a type, UBL has decided to reject this approach, because by default <xsd:redefine> does not leave any traces of having been used (such as a new namespace, for instance) and because of the danger of circular redefinitions.

XSD extension is used when additional information must be added to an existing UBL type. For example, consider XSD definitions for a sample PartyType in Figure A-3. A company might use a special identification code in relation to certain parties. This code should be included in addition to the standard information used in a Party description (PartyName, Address, etc.) This can be achieved by creating a new type that references the existing type and adds the new information as shown in Figure A-4.

```
<xsd:complexType name="PartyType">
   <xsd:sequence>

      <xsd:element ref="PartyIdentification"
         minOccurs="0" maxOccurs="unbounded">
      </xsd:element>

      <xsd:element ref="PartyName"
         minOccurs="0" maxOccurs="1">
      </xsd:element>

      <xsd:element ref="Address"
         minOccurs="0" maxOccurs="1">
      </xsd:element>

   </xsd:sequence>
</xsd:complexType>
```

Figure A-3 – Example complex `PartyType`

```
<xsd:complexType name="MyPartyType">
   <xsd:extension base="PartyType">
      <xsd:element ref="MyPartyID" minOccurs="1" maxOccurs="1"/>
   </xsd:extension>
</xsd:complexType>
```

Figure A-4 – XSD definitions to extend the PartyType in Figure A-3

XSD restriction is used when information in an existing UBL type must be constrained or taken away. For instance, the UBL PartyType permits the inclusion of any number of Party identifiers or none. If a specific organization wishes to allow exactly one identifier, this can be achieved by restricting the cardinality of that particular element as shown in Figure A-5.

```
<xsd:complexType name="MyPartyType">
   <xsd:restriction base="PartyType">
      <xsd:sequence>

         <xsd:element ref="PartyIdentification"
            minOccurs="1" maxOccurs="1">
         </xsd:element>

         <xsd:element ref="PartyName"
            minOccurs="0" maxOccurs="1">
         </xsd:element>

         <xsd:element ref="Address"
            minOccurs="0" maxOccurs="1">
         </xsd:element>

      </xsd:sequence>
   </xsd:restriction>
</xsd:complexType>
```

Figure A-5 - XSD definitions to restrict the PartyType in Figure A-3

A very important characteristic of XSD restriction is that it can only work within the limits substitutability, that is, the resulting type must still be valid in terms of the original type; in other words, it must be a true subset of the original such that a document that validates against the original can also validate against the changed one. Thus:

- the number of repetitions of an element can be reduced (that is, its cardinality can be changed from 1..100 to 1..50)
- an optional element can be eliminated (that is, its cardinality can be changed from 0..3 to 0..0)
- a required element cannot be eliminated or cannot be made optional (that is, its cardinality cannot be changed from 1..3 to 0..3)

Following are final remarks on the use of extension and restriction operations:

- Extensions and restrictions can be applied in any order to the same type.
- Derivations can only be applied to types and not to elements that use those types. UBL uses explicit type definitions for all elements, in fact disallowing XSD use of anonymous types that define a content model directly inside an element declaration.
- Derived types can be used anywhere the original type is allowed.
- All derived types should be created in a separate namespace which might be tied to the user organization and reference the UBL namespaces as appropriate.

APPENDIX B

WEB ONTOLOGY LANGUAGE

This appendix provides a summary of [37] including most common OWL constructs and their basic usage.

Following is a list of most basic OWL constructs:

- *Class*: A class defines a group of individuals that belong together because they share some properties. Classes can be organized in a specialization hierarchy using subClassOf. There is a built-in most general class named "Thing" that is the class of all individuals and is a superclass of all OWL classes. There is also a built-in most specific class named Nothing that is the class that has no instances and a subclass of all OWL classes.

- *rdfs:subClassOf*: Class hierarchies may be created by making one or more statements that a class is a subclass of another class. For example, the class Person could be stated to be a subclass of the class Mammal. From this a reasoner can deduce that if an individual is a Person, then it is also a Mammal.

- *rdf:Property*: Properties can be used to state relationships between individuals or from individuals to data values. Examples of properties include hasChild, hasRelative, hasSibling, and hasAge. The first three can be used to relate an instance of a class Person to another instance of the class Person (and are thus occurrences of ObjectProperty), and the last (hasAge) can be used to relate an instance of the class Person to an instance of the data type Integer (and is thus an occurrence of DatatypeProperty). Both owl:ObjectProperty and owl:DatatypeProperty are subclasses of the RDF class rdf:Property.

- *rdfs:subPropertyOf*: Property hierarchies may be created by making one or more statements that a property is a sub property of one or more other properties. For example, hasSibling may be stated to be a sub property of hasRelative. From this a reasoner can deduce that if an individual is related to another by the hasSibling property, then it is also related to the other by the hasRelative property.

- *rdfs:domain*: A domain of a property limits the individuals to which the property can be applied. If a property relates an individual to another individual, and the property has a class as one of its domains, then the individual must belong to the class. For example, the property hasChild may be stated to have the domain of Mammal. From this a reasoner can deduce that if Frank hasChild Anna, then Frank must be a Mammal. Note that rdfs:domain is called a global restriction since the restriction is stated on the property and not just on the property when it is associated with a particular class. See the discussion below on property restrictions for more information.

- *rdfs:range*: The range of a property limits the individuals that the property may have as its value. If a property relates an individual to another individual, and the property has a class as its range, then the other individual must belong to the range class. For example, the property hasChild may be stated to have the range of Mammal. From this a reasoner can deduce that if Louise is related to Deborah by the hasChild property, (i.e., Deborah is the child of Louise), then Deborah is a Mammal. Range is also a global restriction as is domain above.

- *Individual*: Individuals are instances of classes, and properties may be used to relate one individual to another. For example, an individual named Deborah may be described as an instance of the class Person and the property hasEmployer may be used to relate the individual Deborah to the individual StanfordUniversity.

The following OWL constructs are related to expressing the equality or inequality:

- *equivalentClass*: Two classes may be stated to be equivalent. Equivalent classes have the same instances. Equality can be used to create synonymous classes. For example, Car can be stated to be *equivalentClass* to Automobile. From this a reasoner can deduce that any individual that is an instance of Car is also an instance of Automobile and vice versa.

- *equivalentProperty*: Two properties may be stated to be equivalent. Equivalent properties relate one individual to the same set of other individuals. Equality may be used to create synonymous properties. For example, hasLeader may be stated to be the *equivalentProperty* to hasHead. From this a reasoner can deduce that if X is related to Y by the property hasLeader, X is also related to Y by the property hasHead and vice versa. A reasoner can also deduce that hasLeader is a sub property of hasHead and hasHead is a sub property of hasLeader.

- *sameAs:* Two individuals may be stated to be the same. These constructs may be used to create a number of different names that refer to the same individual. For example, the individual Deborah may be stated to be the same individual as DeborahMcGuinness.

- *differentFrom*: An individual may be stated to be different from other individuals. For example, the individual Frank may be stated to be different from the individuals Deborah and Jim. Thus, if the individuals Frank and Deborah are both values for a property that is stated to be functional (thus the property has at most one value), then there is a contradiction. Explicitly stating that individuals are different can be important in when using languages such as OWL (and RDF) that do not assume that individuals have one and only one name. For example, with no additional information, a reasoner will not deduce that Frank and Deborah refer to distinct individuals.

- *AllDifferent*: A number of individuals may be stated to be mutually distinct in one AllDifferent statement. For example, Frank, Deborah, and Jim could be stated to be mutually distinct using the AllDifferent construct. Unlike the differentFrom statement above, this would also enforce that Jim and Deborah

are distinct (not just that Frank is distinct from Deborah and Frank is distinct from Jim). The AllDifferent construct is particularly useful when there are sets of distinct objects and when modelers are interested in enforcing the unique names assumption within those sets of objects. It is used in conjunction with distinctMembers to state that all members of a list are distinct and pair wise disjoint.

OWL allows restrictions to be placed on how properties can be used by instances of a class. These type (and the cardinality restrictions in the next subsection) are used within the context of an owl:Restriction. The owl:onProperty element indicates the restricted property. The following two restrictions limit which values can be used while the next section's restrictions limit how many values can be used:

- *allValuesFrom*: The restriction allValuesFrom is stated on a property with respect to a class. It means that this property on this particular class has a local range restriction associated with it. Thus if an instance of the class is related by the property to a second individual, then the second individual can be inferred to be an instance of the local range restriction class. For example, the class Person may have a property called hasDaughter restricted to have allValuesFrom the class Woman. This means that if an individual person Louise is related by the property hasDaughter to the individual Deborah, then from this a reasoner can deduce that Deborah is an instance of the class Woman. This restriction allows the property hasDaughter to be used with other classes, such as the class Cat, and have an appropriate value restriction associated with the use of the property on that class. In this case, hasDaughter would have the local range restriction of Cat when associated with the class Cat and would have the local range restriction Person when associated with the class Person. Note that a reasoner can not deduce from an allValuesFrom restriction alone that there actually is at least one value for the property.

- *someValuesFrom*: The restriction someValuesFrom is stated on a property with respect to a class. A particular class may have a restriction on a property that at least one value for that property is of a certain type. For example, the class SemanticWebPaper may have a someValuesFrom restriction on the hasKeyword property that states that some value for the hasKeyword property

should be an instance of the class SemanticWebTopic. This allows for the option of having multiple keywords and as long as one or more is an instance of the class SemanticWebTopic, then the paper would be consistent with the someValuesFrom restriction. Unlike allValuesFrom, someValuesFrom does not restrict all the values of the property to be instances of the same class. If myPaper is an instance of the SemanticWebPaper class, then myPaper is related by the hasKeyword property to at least one instance of the SemanticWebTopic class. Note that a reasoner can not deduce (as it could with allValuesFrom restrictions) that all values of hasKeyword are instances of the SemanticWebTopic class.

Both OWL DL and OWL Full use the same vocabulary although OWL DL requires type separation (a class can not also be an individual or property, a property can not also be an individual or class) To imply that restrictions cannot be applied to the language elements of OWL itself (something that is allowed in OWL Full). Furthermore, OWL DL requires that properties are either ObjectProperties or DatatypeProperties: DatatypeProperties are relations between instances of classes and RDF literals and XML Schema data types, while ObjectProperties are relations between instances of two classes. OWL DL and OWL Full vocabulary that extends the constructions of OWL Lite are briefed below:

- *oneOf* (enumerated classes): Classes can be described by enumeration of the individuals that make up the class. The members of the class are exactly the set of enumerated individuals; no more, no less. For example, the class of daysOfTheWeek can be described by simply enumerating the individuals Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday. From this a reasoner can deduce the maximum cardinality (7) of any property that has daysOfTheWeek as its allValuesFrom restriction.

- *hasValue* (property values): A property can be required to have a certain individual as a value (also sometimes referred to as property values). For example, instances of the class of dutchCitizens can be characterized as those people that have theNetherlands as a value of their nationality. (The nationality value, theNetherlands, is an instance of the class of Nationalities).

- *disjointWith*: Classes may be stated to be disjoint from each other. For example, Man and Woman can be stated to be disjoint classes. From this disjointWith statement, a reasoner can deduce an inconsistency when an individual is stated to be an instance of both and similarly a reasoner can deduce that if A is an instance of Man, then A is not an instance of Woman.

- *unionOf*, *complementOf*, *intersectionOf* (Boolean combinations): OWL DL and OWL Full allow arbitrary Boolean combinations of classes and restrictions: unionOf, complementOf, and intersectionOf. For example, using unionOf, we can state that a class contains things that are either USCitizens or DutchCitizens. Using complementOf, we could state that children are not SeniorCitizens. (i.e. the class Children is a subclass of the complement of SeniorCitizens). Citizenship of the European Union could be described as the union of the citizenship of all member states.

- *minCardinality*, *maxCardinality*, *cardinality* (full cardinality): While in OWL Lite, cardinalities are restricted to at least, at most or exactly 1 or 0, full OWL allows cardinality statements for arbitrary non-negative integers. For example the class of DINKs ("Dual Income, No Kids") would restrict the cardinality of the property hasIncome to a minimum cardinality of two (while the property hasChild would have to be restricted to cardinality 0).

- *complex classes*: In many constructs, OWL Lite restricts the syntax to single class names (e.g. in subClassOf or equivalentClass statements). OWL Full extends this restriction to allow arbitrarily complex class descriptions, consisting of enumerated classes, property restrictions, and Boolean combinations. In addition, OWL Full allows classes to be used as instances whereas OWL DL and OWL Lite do not.

APPENDIX C

DESCRIPTION LOGICS

Among three sublanguages of OWL presented in Section 2.3, the work described in this thesis is based on OWL-DL, which itself is based on Description Logics (DL) and particularly the *SHOIN* DL [89]. This appendix provides a brief summary of DL variants from [87][88][89][90].

Preliminary Definitions

Description Logics are a well known family of knowledge representation formalisms. They are based on the notion of concepts and roles and are mainly characterized by constructors that allow complex concepts and roles to be built from atomic ones. Knowledge representation systems based on DL systems provide their users with various inference capabilities that deduce implicit knowledge from the explicitly represented knowledge.

In order to ensure a reasonable and predictable behaviour of a DL system, the subsumption problem for the DL employed by the system should at least be decidable, and preferably of low complexity. Consequently, the expressive power of the DL in question must be restricted in an appropriate way. If the imposed restrictions are too severe, however, then the important notions of the application domain can no longer be expressed. Investigating this trade-off between the expressivity of DLs and the complexity of their inference problems has been one of the most important issues in DL research.

This section presents the syntax and semantics of various DLs. This includes the definition of inference problems (concept subsumption and satisfiability, and both of these problems with respect to terminologies) and how they are related.

The logics to be discussed are based on the extension of the well known DL variant Attributive Concept Description Language with Complements ($\mathcal{ALL}$) [91], denoted using $\mathcal{S}$ from now on. This basic DL is then extended in a variety of ways, Figure C-1 provides an overview.

Construct Name	**Syntax**	**Semantics**	***DL***
Atomic concept	A	$A^I \subseteq \Delta^I$	$\mathcal{S}$
Universal concept	T	$T^I \subseteq \Delta^I$	
Atomic roles	R	$R^I \subseteq \Delta^I \times \Delta^I$	
Transitive roles	$R \in R_+$	$R^I = (R^I)^+$	
Conjunction	$C \cap D$	$C^I \cap D^I$	
Disjunction	$C \cup D$	$C^I \cup D^I$	
Negation	$\neg C$	$\Delta^I \setminus C^I$	
Existential restriction	$\exists R.C$	$\{x \mid \exists y.\langle x,y\rangle \in R^I \text{ and } y \in C^I\}$	
Universal restriction	$\forall R.C$	$\{x \mid \forall y.\langle x,y\rangle \in R^I \text{ implies } y \in C^I\}$	
Role hierarchy	$R \subseteq S$	$R^I \subseteq S^I$	$\mathcal{H}$
Inverse role	R^-	$\{\langle x,y\rangle \mid \langle y.x\rangle \in R^I\}$	$\mathcal{I}$
Number restriction	$\geq nR$	$\{x \mid \#\{y.\langle x,y\rangle \in R^I\} \geq n\}$	$\mathcal{N}$
	$\leq nR$	$\{x \mid \#\{y.\langle x,y\rangle \in R^I\} \leq n\}$	
Qualifying number restriction	$\geq nR.C$	$\{x \mid \#\{y.\langle x,y\rangle \in R^I \text{ and } y \in C^I\} \geq n\}$	$\mathcal{Q}$
	$\leq nR.C$	$\{x \mid \#\{y.\langle x,y\rangle \in R^I \text{ and } y \in C^I\} \leq n\}$	

Figure C-1 – Syntax and semantics for $\mathcal{S}$ family of DLs

Definition 1: Let C be a set of concept names and $\boldsymbol{R}$ a set of role names with transitive role names $\mathbf{R}_+ \subseteq \boldsymbol{R}$. The set of $\mathcal{SI}$-roles is $\mathbf{R} \cup \{R^- \mid R \in \boldsymbol{R}\}$. The set of $\mathcal{SI}$-concepts is the smallest set such that every concept name is a concept and if C and D are concepts and R is an $\mathcal{SI}$-role, then $(C \cap D)$, $(C \cup D)$, $(\neg C)$, $(\forall R.C)$,

$(\exists R.C)$ are also concepts (the last two are called universal and existential restrictions respectively).

The function Inv is defined on roles such that $Inv(R) = R^-$ if R is a role name and $Inv(R) = S$ if $R = S^-$. The function Trans is defined such that Trans(R) returns true iff R is a transitive role.

SHI is obtained from *SI* by allowing, additionally, for a set of role inclusion axioms of the form $R \subseteq S$, where R and S are two roles, each of which can be inverse. A role hierarchy R^+ is a finite set of such role inclusion axioms.

SHIQ is obtained from *SHI* by allowing, additionally, for qualifying number restrictions, i.e., for concepts of the form $(\geq nR.C)$ and $(\leq nR.C)$, where R is a (possibly inverse) role and n is a non-negative integer.

SHIN is the restricted version of *SHIQ* where qualifying number restrictions may only be of the form $(\geq nR)$ and $(\leq nR)$.

SHOIN is derived from *SHIN* with the addition of nominals, which are concepts with a singleton extension. They are used to define concepts through enumerating its members (i.e. through providing a disjunction of nominals).

It should be noted that the work presented in this thesis does not make use of nominals and the remainder of this appendix discusses the simpler *SHIQ* DL. As described in [89], the decision procedure for *SHOIN* is based on the decision procedure for *SHIQ* with additional rules to handle nominals.

An interpretation $I = (\Delta^I, \cdot^I)$ consists of a set Δ^I called the domain of I, and a valuation $\cdot^I$ which maps every concept to a subset of Δ^I and every role to a subset of $\Delta^I \times \Delta^I$ such that, for all concepts C, D and roles R, S and non-negative integers n, the properties in Figure C-1 are satisfied where #M denotes the cardinality of a set M. An interpretation satisfies a role hierarchy R^+ iff $R^I \subseteq S^I$ holds for each $R \subseteq S \in R^+$. Such an interpretation is called a model of the role hierarchy R^+.

A concept C is called satisfiable with respect to a role hierarchy R^+ iff there is some interpretation I such that I is a model of R^+ and $C^I \neq \emptyset$. Such an interpretation is called a model of C with respect to (w.r.t.) R^+. A concept D subsumes a concept C w.r.t. R^+ (written $C \sqsubseteq_{R^+} D$) iff $C^I \subseteq D^I$ holds for each model I of R^+.

Definition 2: For *SHIQ* concepts C and D, $C \sqsubseteq D$ is called a General Concept Inclusion (GCI) axiom and a finite set T of GCIs is called a terminology (TBox).

An interpretation I is said to satisfy a GCI $C \sqsubseteq D$ iff $C^I \subseteq D^I$, and is said to satisfy a TBox T iff it satisfies every GCI in T; such an interpretation I is said to be a model of T.

A concept C is satisfiable w.r.t. T iff there is a model I of T in which $C^I \neq \emptyset$. A concept D is said to subsume C w.r.t. T iff $C^I \subseteq D^I$ holds for every model I of T. Two concepts C and D are said to be equivalent to each other w.r.t. T iff they are mutually subsuming each other.

The following Lemma shows how GCI axioms can be internalized using a universal role U, that is, a transitive super-role of all roles occurring in T and their respective inverses.

Lemma 1: Let T be a TBox, R a set of role inclusion axioms and C, D *SHIQ*-concepts and let

$$C_T = \bigcap_{C_i \sqsubseteq D_i \in T} (\neg C_i \cup D_i)$$

Let U be a transitive role that does not occur in T, C, D, or R,

$R_U = R \cup \{R \sqsubseteq U, \mathrm{Inv}(R) \sqsubseteq U \mid R$ occurs in T, C, D, or $R\}$

Then C is satisfiable w.r.t. T and R^+ iff $C \cap C_T \cap \forall U.C_T$ is satisfiable w.r.t. R^+_U. Moreover, D subsumes C w.r.t. T and R^+ iff $C \cap \neg D \cap C_T \cap \forall U.C_T$ is unsatisfiable w.r.t. R^+_U.

Proof for Lemma 1 can be found in [88].

Definition 3: Let D be a *SHIQ*-concept, R^+ a role hierarchy, and R_D the set of roles occurring in D and $R+$ together with their inverses. Then $T=(S,L,E)$ is a tableau for D w.r.t. R^+ iff S is a set of individuals, $L: S \rightarrow 2^{clos(D)}$ maps each individual to a set of concepts, $E: R_D \rightarrow 2^{S \times S}$ maps each role to a set of pairs of individuals, and there is some individual $s \in S$ such that $D \in L(s)$. Furthermore, for all $s, t \in S$, C, $C1$, $C2 \in clos(D)$, and R, $S \in R_D$, it holds that;

1. if $C \in L(s)$, then $\neg C \notin L(s)$,
2. if $C_1 \cap C_2 \in L(s)$, then $C_1 \in L(s)$ and $C_2 \in L(s)$,
3. if $C_1 \cup C_2 \in L(s)$, then $C_1 \in L(s)$ or $C_2 \in L(s)$,
4. if $\forall S.C \in L(s)$ and $\langle s, t \rangle \in E(S)$, then $C \in L(t)$,
5. if $\exists S.C \in L(s)$, then there is some $t \in S$ such that $\langle s, t \rangle \in E(S)$ and $C \in L(t)$,
6. if $\forall S.C \in L(s)$ and $\langle s, t \rangle \in E(R)$ for some $R \subseteq S$ with $Tran(R)$, then $\forall R.C \in L(t)$,
7. $\langle x, y \rangle \in E(R)$ iff $\langle y,x \rangle \in E(Inv(R))$,
8. if $\langle s, t \rangle \in E(R)$ and $R \subseteq S$, then $\langle s, t \rangle \in E(S)$,
9. if $(\geq nS.C) \in L(s)$, then $\#S^T(s, C) \geq n$,
10. if $(\leq nS.C) \in L(s)$, then $\#S^T(s, C) \leq n$,
11. if $(\leq nS.C) \in L(s)$ and $\langle s, t \rangle \in E(S)$ then $C \in L(t)$ or $\neg C \in L(t)$

where $S^T(s, C)$ is defined as $\{t \in S \mid \langle s,t \rangle \in E(S)$ and $C \in L(t)\}$

Lemma 2: A *SHIQ*-concept D is satisfiable w.r.t. a role hierarchy $R+$ iff D has a tableau w.r.t. $R+$.

A tableau algorithm for *SHIQ*

Based on Lemma 2, whose proof can be found in [88], an algorithm which constructs a tableau for a *SHIQ*-concept D can be used as a decision procedure for the satisfiability of D w.r.t. a role hierarchy R^+. This section presents such an algorithm.

The tableau algorithm works on a finite completion tree (a tree some of whose nodes correspond to individuals in the tableau, each node being labeled with a set of *SHIQ*-concepts), and employs a blocking technique to guarantee termination. If a path contains two pairs of successive nodes that have pair-wise identical labels and whose connecting edges have identical labels, then the path beyond the second pair is no longer expanded, it is said to be blocked.

Definition 4: A completion tree for a *SHIQ*-concept D is a tree where each node x of the tree is labeled with a set L(x) $\subseteq$ clos(D) and each edge <x,y> is labeled with a set L(<x,y>) of (possibly inverse) roles occurring in clos(D).

Given a completion tree, a node y is called an R-successor of a node x iff y is a successor of x and S $\in$ L(<x,y>) for some S with S $\subseteq$ R. A node y is called an R-neighbor of x iff y is an R-successor of x, or if x is an Inv(R)-successor of y.

A node is blocked iff it is directly or indirectly blocked. A node x is directly blocked iff none of its ancestors are blocked, and it has ancestors x', y and y' such that:

1. x is a successor of x' and y is a successor of y' and
2. L(x) = L(y) and L(x') = L(y') and
3. L(<x',x>) = L(<y',y>).

A node y is indirectly blocked iff one of its ancestors is blocked, or it is a successor of a node x and L(<x, y>) = $\emptyset$.

For a node x, L(x) is said to contain a clash (i.e. contradiction) iff {A,$\neg$A} $\subseteq$ L(x) or if, for some concept C, some role S, and some n $\in$ N:($\leq$ n S.C) $\in$

L(x) and there are n+1 S-neighbors y_0, …, y_n of x such that C ∈ L(y_i) and yi ≠ yj for all 0 ≤ i ≤ j ≤ n.

A completion tree is called contradiction-free iff none of its nodes contains a clash; it is called complete iff none of the expansion rules in Figure C-2 is applicable.

For a *SHIQ*-concept D, the algorithm starts with a completion tree consisting of a single node x with L(x) = {D} and applies expansion rules in Figure C-2, stopping when a clash occurs, and answers "D is satisfiable" iff the completion rules can be applied in such a way that they yield a complete and contradiction-free completion tree.

Lemma 3: D being a *SHIQ*-concept, if the expansion rules can be applied to D such that they yield a complete and contradiction-free completion tree, then D has a tableau.

Proof for Lemma 3 can be found in [88].

$\sqcap$-rule	if 1. $C_1 \sqcap C_2 \in L(x)$, x is not indirectly blocked, and 2. $\{C_1, C_2\} \not\subset L(x)$ then $L(x) \rightarrow L(x) \cup \{C_1, C_2\}$
$\sqcup$-rule	if 1. $C_1 \sqcup C_2 \in L(x)$, x is not indirectly blocked, and 2. $\{C_1, C_2\} \cap L(x) = \varnothing$ then $L(x) \rightarrow L(x) \cup \{C\}$ for some $C \in \{C_1, C_2\}$
$\exists$-rule	if 1. $\exists S.C \in L(x)$, x is not blocked, and 2. x has no S-neighbor y with $C \in L(y)$ then create a new node y with $L(\langle x,y\rangle) = \{S\}$ and $L(y) = \{C\}$
$\forall$-rule	if 1. $\forall S.C \in L(x)$, x is not indirectly blocked, and 2. there is an S-neighbor y of x with $C \notin L(y)$ then $L(y) \rightarrow L(y) \cup \{C\}$
$\forall_+$-rule	if 1. $\forall S.C \in L(x)$, x is not indirectly blocked, and 2. there is some R with $Trans(R)$ and $R \subseteq S$ 3. there is an R-neighbor y of x with $\forall R.C \notin L(y)$ then $L(y) \rightarrow L(y) \cup \{\forall R.C\}$
?-rule	if 1. $(\leq nS.C) \in L(x)$, x is not indirectly blocked, and 2. there is an S-neighbor y of x with $\{C, \neg C\} \cap L(y) = \varnothing$ then $L(y) \rightarrow L(y) \cup \{C\}$ for some $C \in \{C, \neg C\}$
$\geq$-rule	if 1. $(\geq nS.C) \in L(x)$, x is not blocked, and 2. there are not n S-neighbors $y_1, \ldots, y_n$ of x with $C \in L(y_i)$ and $y_i \neq y_j$ for $1 \leq i \leq j \leq n$ then create n new nodes $y_1, \ldots, y_n$ with $L(\langle x, y_i\rangle) = \{S\}$, $L(y_i) = \{C\}$ and $y_i \neq y_j$ for $1 \leq i \leq j \leq n$.
$\leq$-rule	if 1. $(\leq nS.C) \in L(x)$, x is not indirectly blocked, and 2. $\#S^T(x, C) > n$ and there are two S-neighbors y, z of x with $C \in L(y)$, $C \in L(z)$, y is not an ancestor of x and not $y \neq z$ then 1. $L(z) \rightarrow L(z) \cup L(y)$ and 2. if z is an ancestor of x then $L(\langle z,x\rangle) \rightarrow L(\langle z,x\rangle) \cup Inv(L(\langle x,y\rangle))$ else $L(\langle x,z\rangle) \rightarrow L(\langle x,z\rangle) \cup L(\langle x,y\rangle)$ 3. $L(\langle x,y\rangle) \rightarrow \varnothing$ 4. Set $u \neq z$ for all u with $u \neq y$

Figure C-2 – Tableau expansion rules for *SHIQ*

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name :	Yarımağan, Yalın
Nationality :	Turkish (TC)
Date and Place of Birth :	1 November 1974, Ankara
Marital Status :	Married
Phone :	+90 312 219 5787
e-Mail :	yalin.yarimagan@gmail.com

EDUCATION

Degree	Institution	Year of Graduation
MS	METU – Computer Engineering	1999
BS	METU – Computer Engineering	1996
High School	Ankara Atatürk Anatolian High School	1992

WORK EXPERIENCE

Year	Place	Enrollment
2003 – present	Havelsan Inc.	Software Group Manager
1998 – 2003	Cybersoft Information Technologies Corp.	Project Technical Leader
1996 – 1998	Management Information Sys. Inc.	Software Engineer

FOREIGN LANGUAGES

Advanced English

PUBLICATIONS

1. Yarimagan, Y., Dogac, A., "Semantics Based Customization of UBL Document Schemas", Journal of Distributed and Parallel Databases, Vol 22, Issue 2-3, pages 107-131, 2007.

2. Yarimagan, Y., Dogac, A., "A Semantic Based Solution for the Interoperability of UBL Schemas", IEEE Internet Computing, submitted for publication.