FROM DOMAIN MODELS TO COMPONENTS – A FORMAL TRANSFORMATION APPROACH TOWARDS DEPENDABLE SOFTWARE DEVELOPMENT

AFSOON GHAEMI

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF APPLIED SCIENCE (SOFTWARE
ENGINEERING)
CONCORDIA UNIVERSITY
MONTREAL, QUEBEC, CANADA

APRIL 2011

© AFSOON GHAEMI, 2011

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Afsoon Ghaemi

Entitled: From Domain Models to Components – A Formal Transformation Approach Towards Dependable Software Development

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

____ Chair
Dr. René Witte

____ Examiner
Dr. Nematollaah Shiri

____ Examiner
Dr. Olga Ormandjieva

____ Supervisor
Dr. Vangalur Alagar

Approved by ____
Chair of Department or Graduate Program Director

____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Date ____

Abstract

From Domain Models to Components – A Formal Transformation Approach Towards Dependable Software Development

Afsoon Ghaemi

Many academic, industrial, and government research units have unanimously acknowledged the importance of developing dependable software systems. At the same time they have also concurred on the difficulties and challenges to be surmounted in achieving the goal. The importance of domain analysis and linking domain models to software artifacts were also recognized by various researchers. However, no formal approach to domain analysis was attempted. The primary motivation for this thesis stems from this context. Component-based software engineering offers some attractive mechanisms to tackle the inherent complexity in developing dependable systems. Recently a formal approach has been put forth for such a development. This thesis provides a formal approach for domain analysis, and transforms the domain model to components desired by this development process.

Formal Concept Analysis (FCA) is a mathematical theory for identifying and classifying concepts. This thesis taps its potential to formally analyze the domain in a software development context. It turns out that the approach presented in this thesis cannot be fully automated; nevertheless several useful contributions have been made. These include (1) capturing formal concepts and defining them in FCA; (2) defining composition rules to categorize formal concepts and their trustworthy properties; (3) integrating partial formal context tables to build the concept lattice; (4) specifying and developing a model transformation approach to construct trustworthy OWL ontology; (5) implementing a model transformation technique to generate the TADL specification of the reusable component-based system. The proposed approach is applied to CoCoME, as a benchmark case study in the domain of component-based development.

To my parents.

Acknowledgments

This thesis would never have taken shape without the contribution of those who spent their precious time and shared their valuable knowledge helping me to complete my research.

First and foremost, I would like to express my profound appreciation to my supervisor, Dr. Vangalur Alagar for his supervision and support. Also, my utmost gratitude to Dr. Rokia Missaoui, whose encouragement and help enabled me to hurdle the obstacles in learning challenging but attractive mathematical concepts.

In addition, I would like to take the opportunity to thank my examiners, Dr. Olga Ormandjieva and Dr. Nematollaah Shiri, for their precious time to review my thesis and give me helpful advice.

I also would like to thank Concordia University and the Faculty of Computer Science and Software Engineering for offering me the precious opportunity and excellent academic environment to achieve this work.

Finally, I would like to dedicate my ultimate love to my dearest husband, Nassir and my beloved son, Arian for their encouragement and never ending support.

Table of Contents

List of Figures ..... **x**

List of Tables ..... **xiv**

1 Introduction **1**

1.1 Research Context 1

1.1.1 Trustworthiness 2

1.1.2 Component-Based Development (CBD) 3

1.1.3 Domain Analysis 5

1.1.4 Ontology 5

1.1.5 Formal Concept Analysis (FCA) 7

1.2 Difficulties and Drawbacks 8

1.3 Motivations 9

1.4 Research Questions 11

1.5 Proposed Solutions and Contributions 12

1.6 Thesis Outline 13

2 Thesis Background **15**

2.1 Formal Concept Analysis Theory and Tools 15

2.1.1 FCA Theory 17

2.1.2 Impact of FCA 30

2.1.3 FCA and Ontology 32

2.1.4 FCA tools 33

2.1.4.1 Concept Explorer 33

2.1.4.2 ToscanaJ 36

2.1.4.3 Lettice Miner 40

2.2 Ontology 44
2.2.1 Ontology Web Language (OWL)..46
2.2.2 Ontology Tools 47
2.2.2.1 Protégé 48
2.2.2.2 TopBraid Composer 48
2.2.2.3 CMapTools Ontology Editor (COE) 49
2.3 Model Transformation...51
3 Case Study Statement 54
3.1 Common Component Modeling Example (CoCoME) 55
3.2 System Overview..55
3.3 System Requirements Specification 57
3.3.1 Process Sale 57
3.3.2 Manage Express Checkout..59
3.3.3 Order Products 60
3.3.4 Receive Ordered Products 61
3.3.5 Show Stock Reports 61
3.3.6 Show Delivery Reports 62
3.3.7 Change Price..62
3.3.8 Product Exchange 63
3.4 Transformation Tool for Verification Process 64
4 Methodology for Constructing Formal Context Tables 69
4.1 Partial Definition of Context Tables...70
4.1.1 Different Types of Formal Context Tables...71
4.1.2 Definition of Attributes in Formal Context Tables 75
4.1.2.1 Primary Specifications of TADL Component Model 76
4.1.2.2 Keywords to Specify Parameters..79

4.1.2.3 Rules to Compose Partially Defined Context Tables 80

4.2 Construction of Unified Formal Context Table...85

4.2.1 Rules to Integrate Partially Defined Context Tables............................85

4.3 Concept Lattice Derivation...90

4.4 Advantages of the presented method 91

5 Transformation from Context Tables to Concepts Formation..........................93

5.1 Class Hierarchy in Concept Lattice...94

5.2 Class Hierarchy in Ontology 97

5.3 Transformation Rules to Build Ontology from FCA 103

5.3.1 Ontology Overview Definition Rule 104

5.3.2 Class Definition Rules 104

5.3.3 Class Hierarchy Definition Rules 106

5.3.4 Individual Definition Rules 108

5.3.5 Object Property Definition Rules..108

5.3.6 Implication Rules 112

5.4 Name Convention of Ontology Elements 113

5.5 Advantage of Proposed Technique...117

6 Implementation of Transformation from Concepts to Ontology 118

6.1 Input Model...119

6.1.1 Concept Lattice Structural Elements...119

6.1.2 Implication Rules of Concept Lattice...122

6.2 Output Model 125

6.3 Model Transformation..129

6.3.1 Lattice Reducer Procedure 130

6.3.2 Class Definer Procedure..132

6.3.3 Pre-phase Definition Procedure...137

6.3.4 Ontology Builder Procedure 140

7 Model Transformation from Ontology to Components 147

7.1 TADL XML Schemas 148

7.1.1 InterfaceType Schema 148

7.1.2 ComponentType Schema 149

7.1.3 ConnectorType Schema 151

7.1.4 ContractType Schema 152

7.1.5 PackageType Schema 155

7.1.6 RBAC Schema 155

7.1.7 System Schema 157

7.2 Transformation Rules 158

7.3 Model Transformation from OWL Ontology to TADL 163

8 Case Study and Evaluation 171

8.1 Case Study Implementation 172

8.1.1 Context Table Definition and Concept Lattice Derivation 172

8.1.2 Transformation from Concept Lattice to OWL Ontology 187

8.1.3 Transformation from OWL Ontology to TADL Description 191

8.2 Evaluation 195

9 Conclusion 201

9.1 Summary of Results 202

9.2 Assessment 205

9.3 Case Study 209

References 212

Appendix A 222

List of Figures

Figure 1.1: The schema of the whole project and the scope of this thesis 4

Figure 2.1: *Planets* Formal Context Table 18

Figure 2.2: Sample Binary Context Table 19

Figure 2.3: Reduced Labeling Concept Lattice 19

Figure 2.4 (L): Many-valued context 20

Figure 2.4 (R): Conceptual scale 20

Figure 2.5: One-valued context derived from many-valued context 20

Figure 2.6: Concept Lattice of the derived one-valued context 21

Figure 2.7: *Planets* Concept Lattice diagram 24

Figure 2.8: Join-dense of *Planets* Concept Lattice 26

Figure 2.9: Meet-dense of *Planets* Concept Lattice 26

Figure 2.10: *Planets* Formal Context Table in Concept Explorer Tool 34

Figure 2.11: *Planets* Concept Lattice in Concept Explorer Tool 34

Figure 2.12: *Planets* implication sets in Concept Explorer Tool 35

Figure 2.13: Elba's main window while a diagram is edited 38

Figure 2.14: Dialogs for defining the database connection in Elba 39

Figure 2.15: Screenshot of ToscanaJ with nested diagram and highlighting 39

Figure 2.16: Binary Context Editor 41

Figure 2.17: Concept Lattice, tree view structure 41

Figure 2.18: Nested line diagram 43

Figure 2.19: XSLT processor 52

Figure 3.1: Hardware overview of Cash Desk 56

Figure 3.2: Trading system use cases 58

Figure 3.3: Model Checker overview 65

Figure 3.4: Verifying safety and security properties in UPPAAL 66

Figure 4.1: *Planets* Valued Context Table (VCT)....72

Figure 4.2: *Planets* Binary Context Table (BCT)....73

Figure 4.3: *Table1* Valued Context Table (VCT) 73

Figure 4.4: *Table2* Valued Context Table (VCT) 74

Figure 4.5: *Table1* Binary Context Table (BCT)....74

Figure 4.6: *Table2* Binary Context Table (BCT)....74

Figure 4.7: *Table3* Nested Context Table (NCT) of *Table1* and *Table2*....75

Figure 4.8: *Table3* Binary Context Table 75

Figure 4.9 (L): Elements of ComponentType 77

Figure 4.9 (R): Elements of ContractType....77

Figure 4.10: Trustworthy component model....79

Figure 4.11: *Concept1* Valued Context Table (VCT)....84

Figure 4.12: *Concept1* Binary Context Table (BCT) 84

Figure 4.13: *Concept2* Valued Context Table (VCT) 88

Figure 4.14: *Concept2* Binary Context Table (BCT) 88

Figure 4.15: *MergedConcepts* Nested Context Table (NCT) 89

Figure 4.16: *MergedConcepts* Binary Context Table (BCT) 89

Figure 4.17: Merged and pruned *MergedConcepts* context table....90

Figure 4.18: *MergedConcepts* Concept Lattice....91

Figure 5.1: *Planets* Complete Labeling Concept Lattice....95

Figure 5.2: *Planets* Reduced Labeling Concept Lattice 96

Figure 5.3: Various types of OWL properties....98

Figure 5.4: From FCA to OWL Ontology 103

Figure 5.5: Sample hierarchy of object properties in OWL ontology 111

Figure 5.6: Class *FRConcept1-Req3_IFRReq4* and its object 115

Figure 5.7: Properties *hasFRPrivilege* and *hasnotFRPrivilege* 115

Figure 5.8: Properties *hasRolePrivilege* and *hasnotRolePrivilege* 116

Figure 6.1: Lattice specifications in XML format...121

Figure 6.2: The Concept Lattice *MergedConcepts* XML file 121

Figure 6.3: Rules specifications in XML format 124

Figure 6.4: Implication rules of *MergedConcepts* Concept Lattice 124

Figure 6.5: Ontology overview specifications in OWL format 125

Figure 6.6: Ontological class specifications in OWL format..126

Figure 6.7: Individual specifications in OWL format 126

Figure 6.8: Object property specifications in OWL format 127

Figure 6.9: AllValuesFrom Property restriction specifications in OWL format 127

Figure 6.10: HasValue Property restriction specifications in OWL format.....................128

Figure 6.11: Equivalent class specifications in OWL format 128

Figure 6.12: Model Transformation from Concepts to Ontology 130

Figure 6.13: Output XML file of Lattice Reducer procedure...132

Figure 6.14: Output XML file of Class Definer procedure 137

Figure 6.15: Output XML file of Pre-phase Definition procedure 139

Figure 6.16: Output OWL file of Ontology Builder procedure 144

Figure 6.17: *MergedConcepts* OWL Ontology 145

Figure 6.18 (a): The schema of the proposed approach 146

Figure 6.18 (b): FCA part of the proposed approach...146

Figure 7.1: From OWL Ontology to TADL 161

Figure 7.2: Domain Engineering and Component Development 162

Figure 7.3: *MergedConcepts* TADL file in Visual Studio 168

Figure 7.4: *MergedConcepts* TADL XML file (part1) 169

Figure 7.5: *MergedConcepts* TADL XML file (part2) 170

Figure 8.1: *CashBox* Valued Context Table (VCT) 176

Figure 8.2: *CashBox* Binary Context Table (BCT) 176

Figure 8.3: *Cashier* Valued Context Table (VCT) 178

Figure 8.4: *Cashier* Binary Context Table (BCT) 179

Figure 8.5: *CashDesk* Nested Context Table (NCT) 179

Figure 8.6: Merged and pruned *CashDesk* Binary Context Table (BCT) 180

Figure 8.7: *Inventory* Valued Context Table (VCT) 182

Figure 8.8: *Inventory* Binary Context Table (BCT) 183

Figure 8.9: *CoCoME* Binary Context Table (BCT) 184

Figure 8.10: *CoCoME* Concept Lattice 185

Figure 8.11: XML-format *CoCoME* Concept Lattice 186

Figure 8.12: XML-format implication rules of *CoCoME* Concept Lattice 187

Figure 8.13: *CoCoME* OWL Ontology 188

Figure 8.14: *CoCoME* TADL file 192

Figure 8.15: The tag Reactivity of *CoCoME* TADL file 193

Figure 8.16: The tag ConnectorType of CoCoME TADL file 194

List of Tables

Table 4.1: Name Conventions for Attribute Types 83

Table 4.2: Conditions and Integration Rules to merge partial context tables 86

Table 4.3: Integration Rules and Actions to merge partial context tables 87

Table 5.1: Name Conventions for Ontology Elements 114

Table 8.1: Global variables in CoCoME case study 173

Table 8.2: CashBox functional requirement properties 174

Table 8.3: Attribute variables of CashBox context table 175

Table 8.4: Cashier functional requirement properties 177

Table 8.5: Attribute variables of Cashier context table 178

Table 8.6: Newly added attribute variables of CashDesk context table 180

Table 8.7: Inventory functional requirement properties 181

Table 8.8: Attribute variables of Inventory context table 182

Table 8.9: Newly added attribute variables of CoCoME context table 184

Table 8.10: Ontological Attribute/Multi-attribute/Property classes 189

Table 8.11: Ontological trustworthy classes 189

Table 8.12: Cashier Ontological functional requirement classes 190

Table 8.13: Ontological properties 191

Chapter 1

Introduction

In this chapter, we explain the reason behind our interest in the study of *Formal Concept Analysis* (FCA) in order to conduct *domain engineering,* which is the basis for *component-based* development of *dependable* software systems. We state the research problem under consideration, describe our contributions, and present the structure of the thesis.

1.1 Research Context

This thesis is about designing, developing, and verifying a *trustworthy domain model* using Formal Concept Analysis (FCA). The result of FCA will lead to an automatically generated *OWL ontology*. Afterwards, the resulting ontology is automatically transformed into an architecture description language, called *TADL* [49], which is used to develop the trustworthy component-based systems. To reach this goal, a model transformation framework is implemented which produces automatically the detailed specification of reusable components and component-based architecture of the relevant trustworthy

system. The main focus on FCA theory is to compose concept hierarchy and provide a formal basis for *domain analysis*.

During the last four decades, the many challenges in the development of dependable software systems have been addressed by many research and industrial organizations. Since society has come to rely on much software, and much software has direct impact on our daily life, it is important that such software be certified to be dependable. Healthcare domain and safety-critical domain are prime examples of application domains where software should be dependable. Any incorrect execution or service outage in such software systems may lead to catastrophic consequences.

Development of dependable software systems has two significant aspects: the correct implementation of system functionalities, and the selection of the appropriate fault tolerant mechanism to deal with the anticipated failures [61]. Hence, there is a need to design critical systems in such a way that these aspects would be provably correct. Towards this purpose, the credentials of trust should be formally defined along with their level of acceptance [48] while developing these systems.

1.1.1 Trustworthiness

Trustworthiness is the system property that denotes the degree of user confidence that the system will behave as expected [65, 8]. The terms trustworthiness and dependability are used interchangeably in literature [70]. Trustworthiness is a composite concept and the essential quality properties contributing to trustworthiness are *safety, security*, *reliability*, and *availability*. Since many of the current trustworthy systems also involve real-time, we also include *timeliness* to the quality attributes of trustworthiness

[65, 8, 51]. In order to develop trustworthy systems, all the mentioned properties must be combined together in one formal approach [48].

1.1.2 Component-Based Development (CBD)

Component-Based Development (CBD) process is a special type of software development process tailored for developing reusable components and building systems by integrating existing components. Components provide and require services through public interfaces. *Component-Based Software Engineering* (CBSE) promises many advantages to software development including reuse, managing complexity, and reducing development time, effort, and cost. Complexity is effectively managed by dividing the problem into smaller problems of manageable magnitudes, each of which is handled separately in CBSE. The cost of development is reduced by reusing existing solutions to solve these sub-problems [70]. However, the current CBSE practices do not provide the essential needs for developing trustworthy systems, because there is no rigorous formal foundation for the specifications, composition, and verification of non-functional requirements [48].

In [1] the authors have proposed a component-based software engineering approach for developing trustworthy systems by providing a formal component model that collectively addresses the requirements of trustworthiness and detailing a formal component-based development framework. Also they have introduced an architecture description language, called TADL [49], in which the trustworthy component model can be faithfully described. The TADL specification provides a high level description of systems and makes it easy for software architects to understand the architecture and use it in the proposed formal approach. Many tools have been developed [89, 40] to

practice the formal approach in a practical software development process. *Visual Modeling Tool* (VMT) [89] is one of them. It is used as the user interface to construct trustworthy component models and component-based systems. Also, a model transformation tool [40] is used to analyze the TADL specification, and generate *UPPAAL* and *TIMES* timed automata for verifying the trustworthiness properties of the component. On the other hand, the input to the model transformation tool in [40] is to be generated by the software developer, which in turn demands a complete knowledge of the domain of application from which requirements are extracted.

This thesis is a contribution in this context. The scope of this thesis is in the field of domain engineering to provide a formal approach for domain analysis and to construct a domain model that is automatically transformed to the architectural elements of the trustworthy component model. The schema of the whole project and the scope of this thesis are depicted in Figure 1.1.

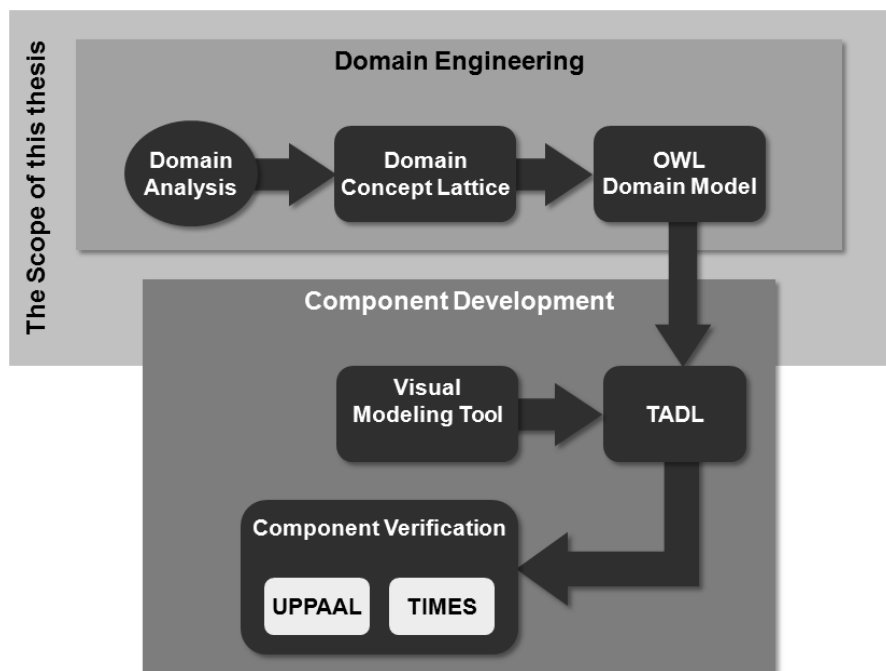

Figure 1.1: The schema of the whole project and the scope of this thesis

1.1.3 Domain Analysis

A domain is a set of applications that share similar requirements, capabilities, and data. Domain engineering is the set of activities that define, model, construct and catalogue a set of artifacts specific to the domain. The artifacts include a model, architectures, components, applications, contexts of operations, and dependability criteria [55]. Domain engineering is an important first step in developing software systems. At the core of domain engineering, domain analysis is used to capture and classify the domain knowledge. It identifies the common and specific requirements that belong to the products in the domain. The collected requirements must include the necessary functionalities, the context of operation for each functionality and the dependability criteria that must be satisfied by the operations. The result of the domain analysis is a domain model which consists of knowledge about the domain by illustrating concepts, associations between concepts, and attributes of concepts. This knowledge can be stored in a knowledge base or ontology which contains vocabulary of the domain anatomy. This knowledge forms the foundation based on which software systems are developed. From the domain model, domain architecture is developed to form the basis for all domain products. Domain applications are designed based on the domain architecture and developed by reusing existing domain components [2].

1.1.4 Ontology

Ontology is a "content theory about the sorts of concepts, their properties, constraints, and the relations between concepts that are possible in a specified domain of knowledge" [14]. It includes machine-interpretable definitions of basic concepts in the

domain and relations among them. Ontology development is necessarily an iterative process and is a major approach for capturing and representing reusable knowledge [52]. Since concepts, relationships and their categorizations in a real world can be represented with domain-specific ontologies, they can be used as resources of domain knowledge for domain analysis. Nowadays, ontology technologies are frequently applied to many problem domains such as (1) communications, (2) computational inferences, and (3) reuse and organization of knowledge [34]. Ontology defines a common vocabulary for researchers who need to share information in a domain. In order to allow sharing and reusing ontologies, a common ontology language was developed and named *ontology Web Language* (OWL) [67, 32].

OWL is a standard development language that is based on logical models. Therefore, it can benefit from the use of the *reasoning* about ontologies. Reasoning involves: (1) syntax checking, (2) consistency checking, (3) subsumption, checking whether a class description is more general than another class description, and (4) query answering.

Many different tools are available for building and maintaining ontologies. The most well known and widely used ontology tools available on the market are Protégé, TopBraid Composer, CMapTools Ontology Editor (COE), Altova SemanticWorks, and SMORE/SWOOP. We review these tools in Chapter 2. Ontology editors mostly define the components of OWL ontologies such as individuals, properties, and classes. Also they use ontology reasonors providing automated inference services on OWL-DL ontologies. Some of them support query languages like SPARQL on ontologies.

1.1.5 Formal Concept Analysis (FCA)

Formal Concept Analysis is a branch of applied mathematics based on the formalization of concept hierarchy and lattice theory [28]. FCA is able to reveal and visualize conceptual structures inherent in data while neither adding nor removing information from the underlying data [11]. This formalism is capable to approach the conceptual structure of an application domain and it may have the potential to provide a formal basis for domain analysis and domain modeling. During domain analysis, FCA techniques are used to extract *extent* objects and *intent* attributes. Then, formal contexts are built, where each *formal context* is a triple (G, M, I) such that G is the set of objects, M is the set of attributes, and $I \subseteq G \times M$ is a binary relation. On the set of all formal concepts of a formal context the sub-/super-concept relation $\leq$, defines the ordering relation that forms a *complete lattice* called the *concept lattice* $B(G, M, I)$ [28]. As an ordered set, a concept lattice can be visualized by a line diagram. The nodes in the line diagram represent the formal concepts of the domain [18].

Over the last two decades, a collection of tools have emerged to help FCA users visualize and analyze concept lattices [75, 38]. They range from the earliest DOS-based implementations (e.g., ConImp and GLAD) to more recent implementations in Java like ToscanaJ [11], Galicia [81], ConExp [15], Coron [73], and Lattice Miner [59]. A main issue in the development of FCA tools is to visualize large concept lattices and provide efficient mechanisms to highlight patterns (e.g., concepts, associations) that could be relevant to the user [13]. FCA software tools such as ConExp, Lattice Miner and ToscanaJ implement basic functionalities needed to develop formal concepts, i.e., they define and store extent objects, intent attributes and their binary relationships in the

formal context table, and then produce the relevant concept lattice. Afterwards, the obtained concept lattice may be saved as an XML-format Meta model.

1.2 Difficulties and Drawbacks

The development of dependable software systems is a difficult task. Also the current solutions have some drawbacks.

Difficulties: The number of the assorted parameters influencing the quality of dependability, combined with the miscellaneous cases of failure events that should be considered, increases enormously the complexity of system development. The other problem is to be localized in the selection of the appropriate fault tolerant mechanism at the final development artifacts [61]. However, the trustworthiness should be provided from the primary steps of software development process, i.e., requirements analysis phase. Finally, lack of standard requirements specification languages and models is another obstacle to obtain accurate analysis and inference over large complex artifacts [85].

Drawbacks: Based on formal foundations and deep theoretical results, methods and tools have been developed to support specification, design, validation and verification of software systems. Many other formal specification and verification techniques have been applied to non-trivial case studies and are used in practice, e.g., for the development of safety critical systems.

However, actual practice shows that the techniques for engineering software-intensive systems suffer from many severe drawbacks in quality and from methodological shortcomings [85]:

- Pragmatic modeling languages and techniques have no clean scientific foundations which inhibit the construction of powerful analysis and development tools;
- Formal approaches are not well-integrated with pragmatic methods and do not scale up to complex software-intensive systems;
- The proposed solutions are too general to deal with the problem contexts so that domain analysis is failed to carry out and this has confronted us with the lack of standard domain-specific software components.

1.3 Motivations

Domain analysis plays a key role in developing dependable software systems. Types and number of trustworthy attributes such as safety, security, reliability, and availability vary from one domain to another domain. Besides, the dependability criterion is to be composed from attributes which are related to concepts in a domain and in turn concepts which belong to a specific application domain. Therefore, we learn that the dependability criterion is domain-dependent and should be formulated from domain concepts. As a result, finding an effective method for domain analysis becomes a necessary task for building dependable systems [2]. Domain analysis enables software engineers acquire or infer implicit knowledge that the stakeholders do not articulate, or assess the trade-offs that will be necessary between conflicting requirements [43]. Fortunately the experts in different domains in conjunction with the standardization bodies have recently started to make ontologies that more or less narrow this gap. Ontology not only is understandable by machines and humans, it also has inference rules that can automatically check for consistency. Although the importance of domain

analysis was recognized in the literature [14, 45] quite early, no formal domain analysis method was put forth for constructing dependable component-based systems [2]. So, our first motivation is to do a domain analysis by capturing domain components and the properties to be specified as part of the dependability criteria in order to establish a standard OWL ontology with the quality attributes. The target ontology can be utilized as a shared knowledge containing reusable components, and the queries and assertions are exchanged with ontology among domain experts. Besides, the consistency checking can be done using its inference rules. Moreover, the retrieved "trustworthy" ontology when applied to component-based development will produce a detailed specification of reusable components and a component-based architecture.

To reach this goal, we prefer to go through a mathematical theory such as FCA. The use of a formal method for conceptual clustering and rule mining brings many advantages. First, formal models built using formal methods provide us concept classifications that can be formally analyzed. Conceptual hierarchies can be formed using precedence relations. Besides, derived implication and association rules of formal methods can be utilized to explore the conceptual structure, their constraints and their relationships. Applying accurate mathematical theories and using structured methodologies may prevent the resulting domain model from probable deficiencies such as redundancies, inconsistencies, and contradictions. Moreover, formal models facilitate the model transformation process and the implementation of automated tools. Hence, the employment of formal methods in domain analysis is our second motivation.

Using formal methods leads to the construction of a formal model that remains in a high-level of abstraction and needs to be transformed to a more practical level and a convenient model. Since the transformation process is complicated enough to be achieved using manual methods, developing automated procedures for model transformation becomes an essential need for this thesis. Moreover, the employment of

automatic methods minimizes the human intervention to avoid errors in the transformation process. Hence, our third motivation is to automate the transformation process by using a Model Transformation approach.

In this research, two transformation tools are developed, one of which automatically generates the OWL domain model from captured domain semantics implementing our defined transformation rules. The second tool automatically transforms OWL model into TADL model performing the specified transformation rules in [48]. It is important that the specified trustworthy properties be considered in transformation process of both mentioned transformation tools.

1.4 Research Questions

Domain analysis is a challenging task that involves identification and analysis of the applications, their detailed requirements, and the relations and data that exist in a specific domain. This research provides an ontology-based approach for domain engineering, and investigates how trustworthy criteria can be handled using formal methods. The derived domain model is transformed into the formal specifications of a component-based system containing trustworthy properties.

The research questions addressed are the following:

- What kind of formal model is suitable for domain analysis? Can Formal Concept Analysis (FCA) help us in formal modeling? How FCA can be used to help in dependable software development?
- Since FCA is an abstract mathematical model for defining the formal concepts, how to define the complex artifacts of component-based systems like services

and interfaces in formal context tables so that finally, they could be transformed into component constructs in TADL?

- How to recognize the trustworthy attributes in the software requirements specifications and transform them into the corresponding properties for formal analysis [48]?
- How to minimize the human interference or guide it in such a manner that would lead to have consistent software specifications [48]?
- How the system requirements, which are collected by domain analysis, are going to be transformed and represented in to the ontology [48]?
- Can FCA be a "semantic basis" for OWL? What are the transformation rules from FCA to OWL? How OWL can help in consistency checking?

1.5 Proposed Solutions and Contributions

In order to answer the above questions, we introduce Formal Concept Analysis (FCA) as a mathematical theory for conceptual clustering and rule mining which is applied in requirements analysis and component retrieval. Also, OWL is used as a common ontology language to formally represent the results of domain analysis which allows reasoning about ontologies. Lattice Miner FCA tool and TopBraid Composer ontology tool are adopted as application platforms. TADL, an architecture description language, is applied as a high level specification for trustworthy component models. Since all the above tools use XML format to represent their input and output models, Extensible Stylesheet Language Transformation (XSLT) is adopted to perform model transformations.

The main concern of this thesis is developing an OWL ontology derived from the software requirements specifications by applying Formal Concept Analysis. Besides, the resulting trustworthy domain model is transformed to TADL specification in order to be used in dependable component-based software development. Indeed, the main contributions of this thesis are the following:

1. Specification of the formal concepts, captured during domain analysis, to define formal context tables using FCA. To satisfy the trustworthiness, the quality properties safety, security and timeliness are also deliberated.
2. Development of guidelines to define the formal context tables according to the component-based artifacts and trustworthy properties.
3. Development of guidelines to integrate the partially defined formal context tables, to construct a unified and consistent formal concept lattice.
4. Specification and implementation of a model transformation approach to generate a standard OWL ontology containing the trustworthy criteria.
5. Implementation of the “transformation rules” defined in [48] to generate the TADL specification of the reusable components and the component-based architecture which are relevant to the obtained OWL ontology.

1.6 Thesis Outline

The rest of the thesis is structured as follows: Chapter 2 discusses the state of the art theory and tools related to this research. Chapter 3 introduces the problem statement of a benchmark case study that will be illustrated throughout the thesis. Chapters 4 to 7 contain our main contributions. Chapter 4 states the research methodology for domain analysis. It discusses the presentation of formal context tables that lead to the

construction of the formal concept lattice. Chapter 5 provides an automated model transformation technique for generating OWL ontology from formal concept lattice. The implementation of the transformation rules which are presented in Chapter 5 is explained in Chapter 6. Chapter 7 includes the transformation algorithm of ontology to TADL and its implementation. In Chapter 8 the case study introduced in Chapter 3 is fully explained with the techniques and tools presented in this thesis. Also the results are critically discussed, comparing what is done in this research with what has been done in previous works [89, 40]. Finally, in Chapter 9 we conclude the thesis by summarizing our contributions and identifying directions for future work.

Chapter 2

Thesis Background

This chapter reviews basic concepts on which the rest of the thesis depends. This review will help the reader to understand the rationale behind the objectives, and to appreciate and judge the contributions of this thesis. This chapter is organized in three sections. Section 2.1 presents the Formal Concept Analysis (FCA) and its impact on software engineering. Then, the interactions between FCA and ontology and also some FCA tools are illustrated in this section. The formal definition of ontology and the different types of OWL ontology languages are explained in Section 2.2 in which the initial objectives of ontology and its utilized tools are also introduced. Section 2.3 discusses the model transformation approach and proposes its implementation using XSLT Stylesheets and XPath [74, 47].

2.1 Formal Concept Analysis Theory and Tools

Formal Concept Analysis [28], that is also named Galois Graphs, is introduced by WILLE [84] and WOLFF [87]. It is a mathematical theory of concept hierarchies based

on Lattice theory. FCA provides a conceptual framework for structuring, analyzing and visualizing concepts and concept hierarchies. In FCA, application domains are organized and structured according to Concept Lattices [23]. In other words, Formal Concept Analysis can capture the conceptual structure of an application domain [19]. It starts with an analysis of the formal context given by use cases and the relevant "things" involved in these cases. It produces a lattice visualized by a line diagram which is used as a design and decision aid for building an appropriate class/object structure. This structure is a prerequisite for further modeling steps, e.g. modeling of processes by sequence diagrams. The formal context and the concept lattice represent two different views on the same information. Usually a line diagram of the concept lattice is computed from the formal context and further investigation of the context data is done with the help of the diagram [18].

FCA has been applied in various fields of science, such as Psychology, Sociology, Medicine, Linguistics, and Computer Science. In each domain FCA makes the concepts and their relations explicit and precise [23]. In the domain of software engineering, FCA has typically been applied to support software maintenance activities [20, 21, 68, 69], the refactoring or modification of existing code [10, 24], and the identification of object-oriented structures [60, 66, 79]. There is also a body of literature [17, 63, 64] reporting the application of FCA to the identification and maintenance of class hierarchies in database schemata. Beyond the identification of classes, FCA has also been applied to other areas of software engineering including requirements analysis [7, 12, 58] and component retrieval [22, 46]. There are some papers [22, 46, 76] which describe applications to detailed design. There are only a few papers describing applications to testing [5, 9]. No work has been done on the application of FCA to software integration, qualification testing, acceptance support or coding. Thus these areas present an opportunity to FCA researchers [77].

2.1.1 FCA Theory

In this section, we review the main principles of FCA by giving some definitions and using some examples. The detailed knowledge about FCA can be found in [28].

- **Formal Context:** The sets of formal objects and formal attributes together with their relation to each other form a formal context. The simplest format for writing down a formal context is a cross table. This is a rectangular table with one row for each object and one column for each attribute, having a cross in the intersection of row g with column m iff $(g, m) \in I$, where I is the incidence of the context [56].

  Definition 1: A formal context is defined [26, 16] as a triple (G, M, I) where G and M are sets and I is a binary relation $I \subseteq G \times M$. The elements of G and M are called *objects* and *attributes*, respectively. If $g \in G$ and $m \in M$ are in relation I, we write $(g, m) \in I$ or $g\ I\ M$ and say "object g has attribute M ".

  In FCA theory, there are no restrictions about the nature of objects and attributes. We may interchange the role of objects and attributes: if (G, M, I) is a formal context, then so is the dual context (M, G, I^{-1}) (with $(m, g) \in I^{-1} \Leftrightarrow (g, m) \in I$). It is also not necessary that G and M be disjoint or even different [26, 16]. The *Planets* formal context table [16] shown in Figure 2.1 illustrates these.

  When implementing a Conceptual Information System using methods of Formal Concept Analysis, the data is modeled mathematically by a *many-valued context* and is transformed via *conceptual scaling* [27]. This means that a formal context called conceptual scale is defined for each of the many-valued attributes which has the values of the attribute as objects. If a many-valued context and a conceptual scale are given, we can derive the realized scale, i.e., a formal

context which has the objects of the many-valued context as objects and the attributes of the scale as attributes. In the realized scale, an object has an attribute if the value assigned to the object in the many- valued context has the attribute in the conceptual scale [11].

	SizeSmall	SizeMedium	SizeLarge	DistanceNear	DistanceFar	MoonYes	MoonNo
Mercury	X			X			X
Venus	X			X			X
Earth	X			X		X	
Mars	X			X		X	
Jupiter			X		X	X	
Saturn			X		X	X	
Uranus		X			X	X	
Neptune		X			X	X	
Pluto	X				X	X	

Figure 2.1: *Planets* Formal Context Table

Definition 2: Many-valued context is a quadruple (G, M, W, I) consisting of three sets G, M, and W, and a ternary relation $I \subseteq G \times M \times W$ such that (g, m, w_1), $(g, m, w_2) \in I$ always implies $w_1 = w_2$. The elements of G, M, and W are respectively called objects, attributes, and attribute values. The tuple $(g, m, w) \in I$ is read as the object g that has the value w for the attribute m. M_m is the set of all m attributes that each $m \in M_m$ may be understood as a partial map from G into W with

$$m(g) = w :\Leftrightarrow (g, m, w) \in I.$$

To obtain formal concepts from a many-valued context (G, M, W, I), FCA offers the method of conceptual scaling which assigns a formal context (G_m, M_m, I_m) with $m(G) \subseteq G_m$, named a conceptual scale, to each (many-valued) attribute

$m \in M$. In most applications, a formal context $(G, \bigcup_{m \in M} \{m\} \times M_m, J)$ is derived from the many-valued context (G, M, W, I) by the conceptual scales (G_m, M_m, I_m) $(m \in M)$ where the relation J is defined by $g\ J\ (m, n){:} \Leftrightarrow m(g) = n$ [27].

Lattice Miner

File Edit Lattice Rules Context Window About

Context : Sample Binary Context

	a1	a2	b1	b2	c1	c2
1	X					
2		X				
3			X		X	
4			X			X
5				X	X	

Figure 2.2: Sample Binary Context Table

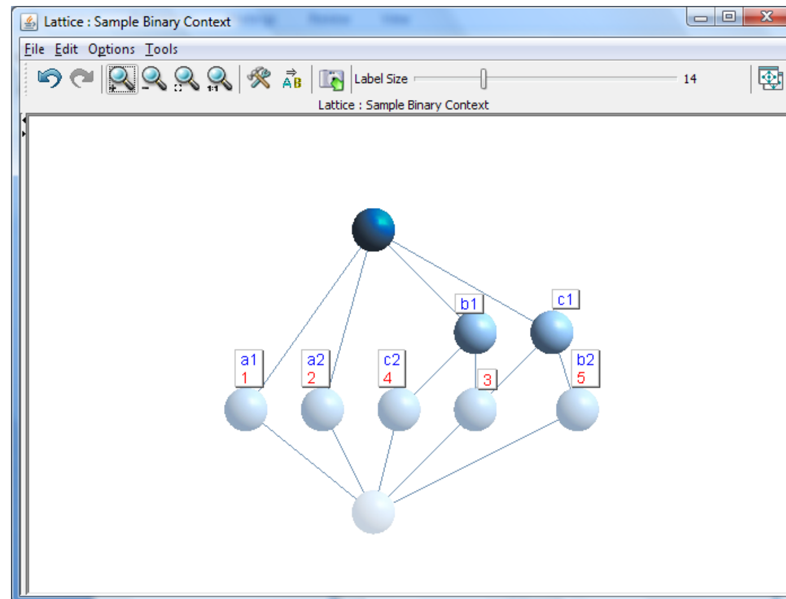

Figure 2.3: Reduced Labeling Concept Lattice

Some examples are extracted from [31] to illustrate the above definitions. Figure 2.2 and Figure 2.3 present the sample binary context table and its concept

lattice, while Figures 2.4, 2.5, and 2.6 illustrate the relevant many-valued context table and the concept lattice that are derived from conceptual scaling.

	a	b	c
1	a1		
2	a2		
3		b1	c1
4		b1	c2
5		b2	c1

Figure 2.4 (L): Many-valued context

	a	a1	a2
a	X		
a1	X	X	
a2	X		X

Figure 2.4 (R): Conceptual scale

	a	a1	a2	b	b1	b2	c	c1	c2
1	X	X							
2	X		X						
3				X	X		X	X	
4				X	X		X		X
5				X		X	X	X	

Figure 2.5: One-valued context derived from many-valued context

- **Derivation Operators:** Given a selection $A \subseteq G$ of objects from a formal context (G, M, I), we may ask which attributes from M are common to all these objects. This defines an operator that produces for every set $A \subseteq G$ of objects the set A' of their common attributes.

Definition 2: For $A \subseteq G$ of objects from a formal context (G, M, I), we let $A' := \{m \in M \mid g\ I\ m \text{ for all } g \in A\}$. Dually, we introduce for a set $B \subseteq M$ of attributes $B' :=$

$\{g \in G \mid g\ I\ m \text{ for all } m \in B\}$. These two operators are the *derivation operators* for (G, M, I) [26, 16].

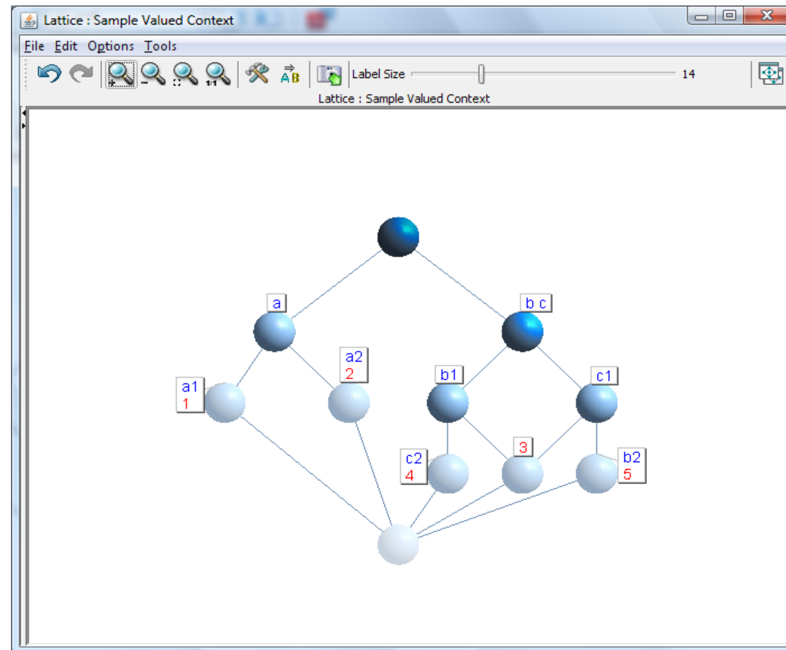

Figure 2.6: Concept Lattice of the derived one-valued context

- **Formal Concept:** A pair of a set of formal objects and a set of formal attributes that is "closed" (i.e., one can neither enlarge the attribute nor the object set) is called a formal concept. The set of formal objects of a formal concept is called its *extension*. The set of formal attributes is called its *intension*. For a given formal context, the formal concepts, their extensions and intensions are uniquely defined and fixed [56].

Definition 3: The pair (A, B) is a formal concept of formal context (G, M, I) iff $A \subseteq G$, $B \subseteq M$, $A' = B$, and $A = B'$. The set A is called the *extent* of the formal concept (A, B), and the set B is called its *intent* [26].

Lemma 1: The pair *(A, B) is a formal concept of (G, M, I) iff $A \subseteq G$, $B \subseteq M$, and A and B are each maximal (with respect to set inclusion) with the property $A \times B \subseteq I$* [26].

A formal context may have many formal concepts. The set of all formal concepts of (G, M, I) is denoted $\beta\,(G, M, I)$.

- **Conceptual Hierarchy:** Formal concepts can be (partially) ordered in a natural way. Again, the definition is inspired by the way we usually order concepts in a "subconcept-superconcept" hierarchy.

  Definition 4: Let (A_1, B_1) and (A_2, B_2) be formal concepts of (G, M, I). We say that (A_1, B_1) is a subconcept of (A_2, B_2) (and equivalently (A_2, B_2) is a superconcept of (A_1, B_1) iff $A_1 \subseteq A_2$. We use the $\leq$ symbol to express this relation and thus have:

$$(A_1, B_1) \leq (A_2, B_2) \Leftrightarrow A_1 \subseteq A_2 \Leftrightarrow B_2 \subseteq B_1.$$

  The set of all formal concepts of (G, M, I), ordered by this relation, is denoted $\beta\,(G, M, I)$ and is called the *concept lattice* of the formal context (G, M, I) [26, 16].

From a philosophical point of view a concept is a unit of thoughts consisting of two parts, the extension and the intension. The extension covers all objects belonging to this concept and the intension comprises all attributes valid for all those objects. Hence objects and attributes play an important role together with several relations. Example relations [86] are (1) the hierarchical "subconcept-superconcept" relation between concepts, (2) the implication relation between attributes, and (3) the incidence relation "an object has an attribute".

- **Supremum and Infimum:** The concept operations resemble more of the operations greatest common divisor and least common multiple.

  Definition 5: Let $(M, \leq)$ be a partially ordered set, and A be a subset of M. A lower bound of A is an element s of M with $s \leq a$, for all $a \in A$. An upper bound of A is defined dually. If there exists a largest element in the set of all lower bounds of A, then it is called the *infimum* (or *meet*) of A. It is denoted *inf* A or $\wedge A$. The *supremum* (or *join*) of A (*sup* A or $\vee A$) is defined dually. For $A = \{x, y\}$, we write also $x \wedge y$ for their infimum, and $x \vee y$ for their supremum. We use the large symbols $\vee$ and $\wedge$ for arbitrary *suprema* and *infima* [26].

  Lemma 2: *For any two formal concepts (A_1, B_1) and (A_2, B_2) of some formal context we obtain* [26, 16]

  1. *the infimum (greatest common subconcept) of (A_1, B_1) and (A_2, B_2) as*

  $$(A_1, B_1) \wedge (A_2, B_2) = (A_1 \cap A_2\ ,\ (B_1 \cup B2)''),$$

  2. *the supremum (least common superconcept) of (A_1, B_1) and (A_2, B_2) as*

  $$(A_1, B_1) \vee (A_2, B_2) = ((A_1 \cup A_2)''\ ,\ \ (B_1 \cap B_2)).$$

  Note that: The operation $(.)''$ is a *closure* operator [28]. Sets $A \subseteq G$, $B \subseteq M$ are called *closed* if $A = A''$ and $B = B''$. Obviously, extents and intents are closed sets [44].

- **Concept Lattice Diagram:** The concept lattice of (G, M, I) is the set of all formal concepts of (G, M, I), ordered by the subconcept-superconcept relation. Ordered sets of moderate size can conveniently be displayed as order diagrams. In a line diagram, each node represents a formal concept. A concept c_1 is a subconcept of a concept c_2 if and only if there is a path of descending edges from the node

representing c_2 to the node representing c_1. The name of an object g is always attached to the node representing the smallest concept with g in its extent; dually, the name of an attribute m is always attached to the node representing the largest concept with m in its intent. We can read the context relation from the diagram because an object g has an attribute m if and only if the concept labeled by g is a subconcept of the one labeled by m. The extent of a concept consists of all objects whose labels are attached to subconcepts. Dually, the intent consists of all attributes attached to superconcepts [26]. Figure 2.7 shows the *Planets* concept lattice diagram of the formal context depicted in Figure 2.1.

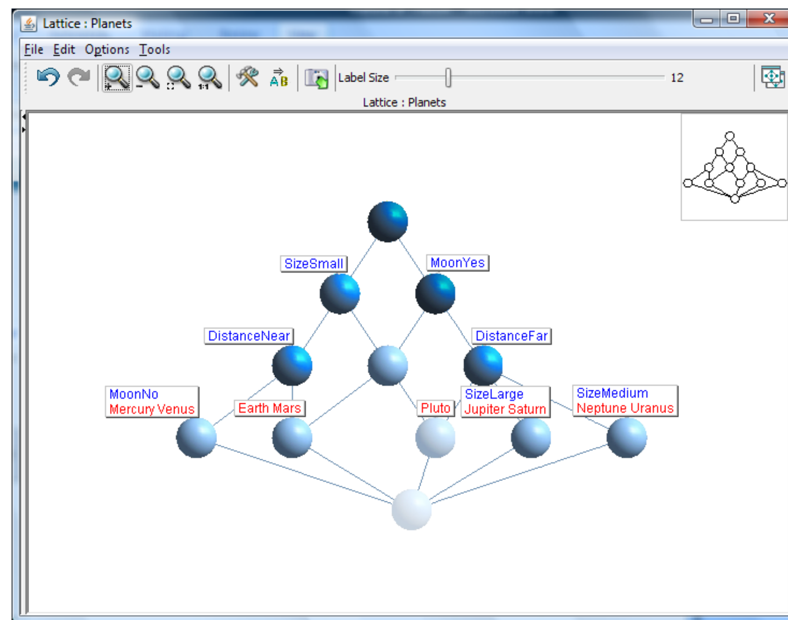

Figure 2.7: *Planets* Concept Lattice diagram

- **Complete Lattice:** A lattice is an algebraic structure with two operations, called meet (infimum) and join (supremum) that satisfy certain natural conditions. With the ordering relation $\leq$, the set of all formal concepts of a formal context forms a complete lattice called the concept lattice $\beta\ (G, M, I)$ [28].

Definition 6: A partially ordered set $\mathbb{V} := (V, \leq)$ is a lattice, if there exists, for every pair of elements $x, y \in V$, their infimum $x \wedge y$ and their supremum $x \vee y$ [26].

Concept lattices have an additional property that they are complete lattices. This means that the operations of infimum and supremum do not only work for an input consisting of two elements, but also works for arbitrarily many elements.

Definition 7: A partially ordered set $\mathbb{V} := (V, \leq)$ is a complete lattice, if for every set $A \subseteq V$ exist its infimum $\wedge V$ and its supremum $\vee A$ [26].

Definition 8: Let P be a non-empty ordered set.

(i) If $x \vee y$ and x ∧ y exist for all $x, y \in P$, then P is called a lattice.

(ii) If $\vee S$ and $\wedge S$ exist for all $S \subseteq P$, then P is called a complete lattice [16].

- **Fundamental Theorem of Formal Concept Analysis:** This theorem gives a precise formulation of the algebraic properties of concept lattices and is a basis for many other results. Its formulation contains some technical terms as follows:

  A set of elements of a complete lattice is called *supremum-dense* (*join-dense*), if every lattice element is a supremum of elements from this set. Dually, a set is called *infimum-dense* (*meet-dense*), if the infima that can be computed from this set exhaust all lattice elements [26, 16]. Figures 2.8 and Figure 2.9 depict join-dense and meet-dense of *Planets* concept lattice in dashed areas.

  Definition 9: Two complete lattices V and W are *isomorphic* ($V \cong W$), if there exists a bijective mapping $\varphi: V \to W$ *with* $x \leq y \Leftrightarrow \varphi(x) \leq \varphi(y)$. The mapping φ is then called lattice *isomorphism* between V and W.

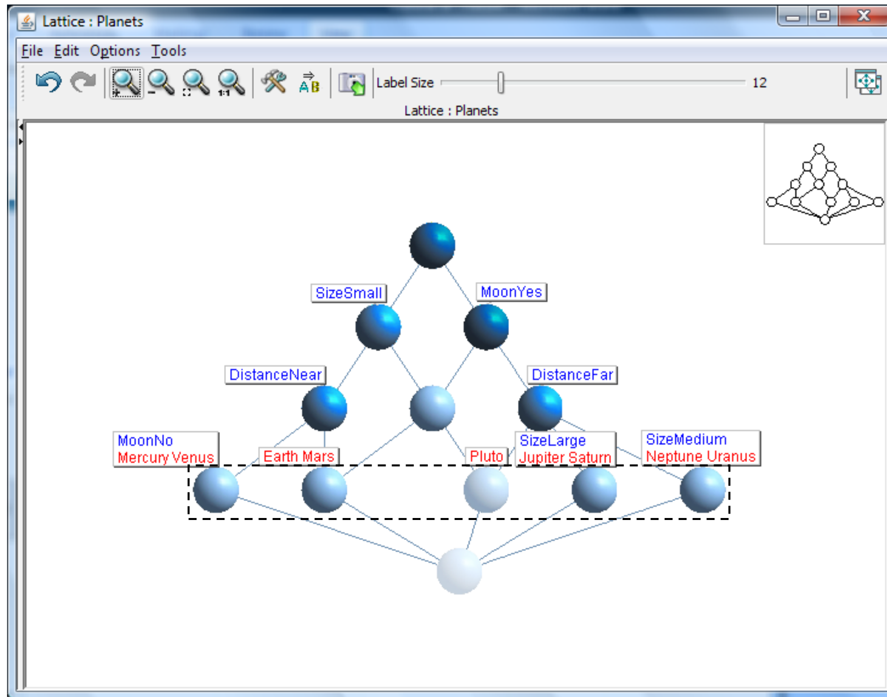

Figure 2.8: Join-dense of *Planets* Concept Lattice

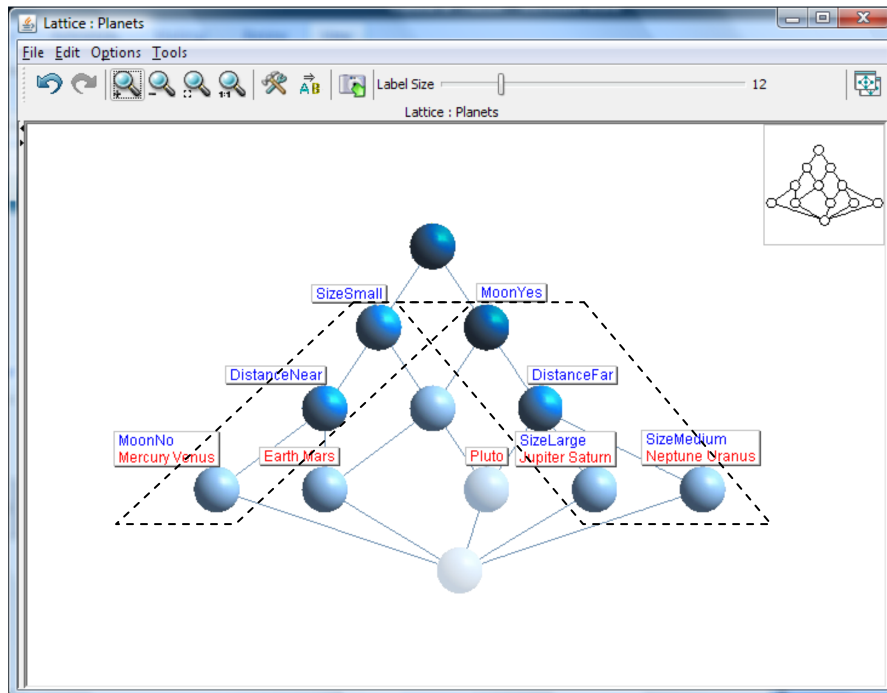

Figure 2.9: Meet-dense of *Planets* Concept Lattice

Theorem 1: *The concept lattice of any formal context (G, M, I) is a complete lattice. For an arbitrary set $\{(A_i, B_i) \mid i \in I\} \subseteq \beta\ (G, M, I)$ of formal concepts, the supremum is given by*

$$\bigvee_{i \in I}(A_i, B_i) = \left(\left(\bigcup_{i \in I} A_i\right)'', \bigcap_{i \in I} B_i\right)$$

and the infimum is given by

$$\bigwedge_{i \in I}(A_i, B_i) = \left(\bigcap_{i \in I} A_i, \left(\bigcup_{i \in I} B_i\right)''\right)$$

A complete lattice L is isomorphic to $\beta\ (G, M, I)$ iff there are mappings $\gamma: G \rightarrow L$ and

$\mu: M \rightarrow L$ such that $\gamma\ (G)$ is supremum-dense in L, and $\mu\ (M)$ is infimum-dense in L, and

$$g\ I\ m \iff \gamma\ (g) \leq \mu\ (m)$$

in particular, $L \cong \beta\ (L, L, \leq)$ [26, 16].

The first part of the theorem gives the precise formulation for infimum and supremum of arbitrary sets of formal concepts. The second part of the theorem gives, among other information, the fact that every complete lattice is isomorphic to a concept lattice. This means that for every complete lattice we must be able to find a set G of objects, a set M of attributes and a suitable relation I, such that the given lattice is isomorphic to $\beta\ (G, M, I)$.The theorem does not only say how this can be done, it describes in fact all possibilities to achieve this [26].

- **Implications:** *Implications* have been studied by Ganter & Wille [29] since 1986. They can be used for a step-wise computer-guided construction of conceptual knowledge called “attribute exploration” [28] that is developed into “concept exploration” [72] which can be used to explore sub-lattices of larger data sets. An attribute implication of a context is a pair of subsets of attributes, say X, Y, for

which $X' \subseteq Y'$, that is, each object having all attributes of X has also all attributes of Y. This notion corresponds to that of attribute inheritance in "Semantic nets" [23].

Definition 10: The implication relation $A \rightarrow B$ holds in a context (G, M, I) if every object intent respects $A \rightarrow B$. That is, if each object that has all the attributes in A also has all the attributes in B. We also say that $A \rightarrow B$ is an implication of (G, M, I). The set A is called the *premise,* and B is its *conclusion* [26].

Proposition 1: *An implication $A \rightarrow B$ holds in a context (G, M, I) if and only if $B \subseteq A''$, which is equivalent to $A' \subseteq B'$. It then automatically holds in the set of all concept intents as well* [26].

Note that: The derivation operator (.)′ has been defined in Definition 2, and the closure operator (.)″ is defined in Lemma 2.

An implication $A \rightarrow B$ holds in a context (G, M, I) if and only if each of the implications

$$A \rightarrow m, \quad m \in B,$$

holds ($A \rightarrow m$ is a short form for $A \rightarrow \{m\}$). We can read this off from a concept lattice diagram in the following manner: $A \rightarrow m$ holds if the infimum of the attribute concepts that correspond to the attributes in A is less than or equal to (partial order on the lattice) the attribute concept for m; formally, if

$$\wedge \{ \mu a \mid a \in A\} \leq \mu m.$$

$A \rightarrow B$ holds in a context (G, M, I) if

$$\wedge \{ \mu a \mid a \in A\} \leq \wedge \{ \mu b \mid b \in B\}.$$

Informally, implications between attributes can be found along upward paths in the lattice.

As an example, let us consider the *Planets* concept lattice (Figure 2.7). It is considered that μ (*DistanceFar*) $\leq \mu$ (*MoonYes*), which can be read as *DistanceFar* → *MoonYes*, or "A planet which is far away has a moon." As another example, we refer to *MoonNo* → {*DistanceNear*, *SizeSmall*} that means "A planet with no moon is near and its size is small". Also, the sample implication rule {*DistanceNear*, *DistanceFar*} → *SizeLarge* is always true, because its premise is contradictory.

Implications obey Armstrong rules [44]:

$$\frac{A \to B}{A \cup C \to B} \qquad \frac{A \to B,\ A \to C}{A \to B \cup C} \qquad \frac{A \to B,\ B \to C}{A \to C}$$

A minimal (in the number of implications) subset of implications, from which all other implications of a context can be deducted by means of Armstrong rules was characterized in [35]. This subset is called Duquenne-Guigues or stem base in the literature. Guigess and Duquenne [28] have proved that for every context with a finite set of attributes A, there is a sound, complete and non-redundant set of implications, called stem base or Duquenne–Guigues Basis. For this purpose, it defines a pseudo-intent as a set of attributes S which is not an intent ($S'' \neq S$), but contains the closure (P'') of every proper subset that is also pseudo intent [4]. The premises of implications of the stem base can be given by pseudo-intents.

For example, the *Planets* concept lattice (Figure 2.7) consists of 10 implications, including:

- *DistanceFar* → *MoonYes*
- *MoonNo* → {*DistanceNear*, *SizeSmall*}.

Non-base implications such as

- $\{DistanceFar, SizeSmall\} \rightarrow \{MoonYes, SizeSmall\}$
- $MoonNo \rightarrow DistanceNear$

can be derived by propositional logic [69].

2.1.2 Impact of FCA

FCA has been successfully used in the field of Software Engineering; almost all phases of the software life cycle like software architecture, modularization, program code, and configuration analysis have taken advantage of FCA and its beneficial effects. However, early phases of the software development process including requirements elicitation, domain and system modeling have not yet adequately used FCA as a formal framework [37]. This situation is remedied in this thesis.

"In principle, FCA can be used wherever concepts play a significant role in the software process." Referring to this aspect of Formal Concept Analysis, we can focus on requirements engineering (RE), use case analysis (UCA), object-oriented modeling, the analysis of class/object hierarchies and component retrieval. One of the interesting typical applications of FCA is to extract class candidates from the use case descriptions of a System Requirement Specification (SRS). Also FCA provides a "crossing of perspectives" between the functional view and the data view respectively represented by the use cases and "things". By this we mean that FCA fills the gap existing in almost all object oriented methods [37].

The class hierarchy as a principal component of object software development has been confronted with many difficulties in design and maintenance. Especially in the process of requirements evolution, when the size of the hierarchies grows and becomes

more elaborate as the result of modifications, this problem becomes more serious. Therefore, hierarchy construction and reconstruction that includes building the hierarchy from scratch, evolution of the class hierarchy to accommodate new requirements, reengineering of an existing class hierarchy, and merging existing hierarchies require vast work and effort in this field. Moreover, the existing algorithms in many recent approaches are not based on well-defined theoretical fundamentals. Formal Concept Analysis (FCA) would be an appropriate solution, since it proposes a natural theoretical framework for class hierarchy design and maintenance. In FCA, well-defined semantics which are independent from concrete algorithms are applied to the produced hierarchies. Also, the produced hierarchies comply with general quality criteria such as simplicity, comprehensibility, reusability, extensibility and maintainability. Besides, two other concrete quality criteria may be measured directly on the target software artifacts:

- **Minimizing redundancy:** Minimizing redundancy is a well-known software design principle that a class hierarchy should be built on. That is, each artifact in the code/specifications has to be defined in one single place. In addition, it increases the consistency of final result since redundancy makes the maintenance of the resulting software more complex by making inconsistencies between duplicate artifacts.
- **Subclasses as specializations:** Code reuse, especially in code libraries, is facilitated by inheritance hierarchies. Therefore, code sharing in the hierarchy for the reason of acquiring more comprehensibility and reusability may become the main purpose of creating the inheritance between the class hierarchies [31].

There are many software development scenarios within the class hierarchy life-cycle that take advantage of Formal Concept Analysis. Some inspiring examples are as design from scratch, refactoring, and reengineering. FCA provides a framework to deal

with various levels of specification details and offers different well-defined design structures [31].

2.1.3 FCA and Ontology

Recent researches have revealed the interactions among FCA and Conceptual Modeling, Artificial Intelligence (in particular Description Logics), Object-Oriented databases, and software engineering. FCA techniques help also the development of the Semantic Web and, in particular, ontology engineering. A conceptual hierarchy is extracted from the domain to be used for the manual or semi-automatic development of ontology. Moreover, since there are vast and domain-dedicated ontologies in the Web, FCA can be used for reusing and combining these independently developed ontologies. According to this FCA facility, similarity reasoning which is the possibility of determining similar concepts has become the principal part of Semantic Web development, especially to perform ontology mapping, integration, and alignment. Generally, these are difficult tasks that are time-consuming and error-prone because they require human interaction [23].

Domain ontology and Formal Concept Analysis (FCA) have common goal of modeling concepts but each of them has its own specifications and purposes. Domain ontology deals with modeling a “shared understanding of the domain of interest” and capturing conceptual knowledge accepted by domain experts. However, FCA supports the user in analyzing and structuring a domain of interest. Domain concepts in FCA consist of two sets including objects and attributes. Objects are the instances of the concept in that domain and attributes are the descriptors of the concept. It is important to mention that, FCA emphasizes on both extensional and intensional aspects, however

only the intensional part is considered by ontology. As a matter of fact, objects are not necessary in defining ontology, but they are one of the main components of concepts in FCA [23].

2.1.4 FCA Tools

Many tools for Formal Concept Analysis have been developed and are used for the construction, visualization and manipulation of concept lattices. There are open-source tools for most platforms and programming environments. In this thesis, we are using Lattice Miner [59, 13], the open-source Java program, for the following reasons: (1) it has general features that are necessary to develop concept lattices; (2) it provides the definition of binary, valued and nested context tables which are required for establishing and merging complex context tables; (3) the retrieved XML format Meta model is readable and easy to be processed. In this section the tools Concept Explorer (ConExp), ToscanaJ, and Lattice Miner are described.

2.1.4.1 Concept Explorer. Concept Explorer [15] (ConExp) is an interactive tool that allows users to properly explore the lattice by implementing basic functionality needed for study and research of Formal Concept Analysis (FCA). It can be used for analysis of simple attribute object tables, (called context in FCA) drawing the corresponding concept lattice and exploration of different dependencies, that exists between attributes. ConExp is released under BSD-style license. ConExp was first developed as a part of master's thesis at the National Technical University of Ukraine "KPI" in 2000. During the following years, it was extended and now is an open source project on Sourceforge [71]. Figure 2.10 depicts the same table of Figure 2.1, in Concept

Explorer tool. Figure 2.11 shows its concept lattice and Figure 2.12 its implication sets which are produced in Concept Explorer tool.

A	SizeSmall	SizeMedium	SizeLarge	DistanceNear	DistanceFar	MoonYes	MoonNo
Mercury	X			X			X
Venus	X			X			X
Earth	X			X		X	
Mars	X			X		X	
Jupiter			X		X	X	
Saturn			X		X	X	
Uranus		X			X	X	
Neptune		X			X	X	
Pluto	X				X	X	

Figure 2.10: *Planets* Formal Context Table in Concept Explorer Tool

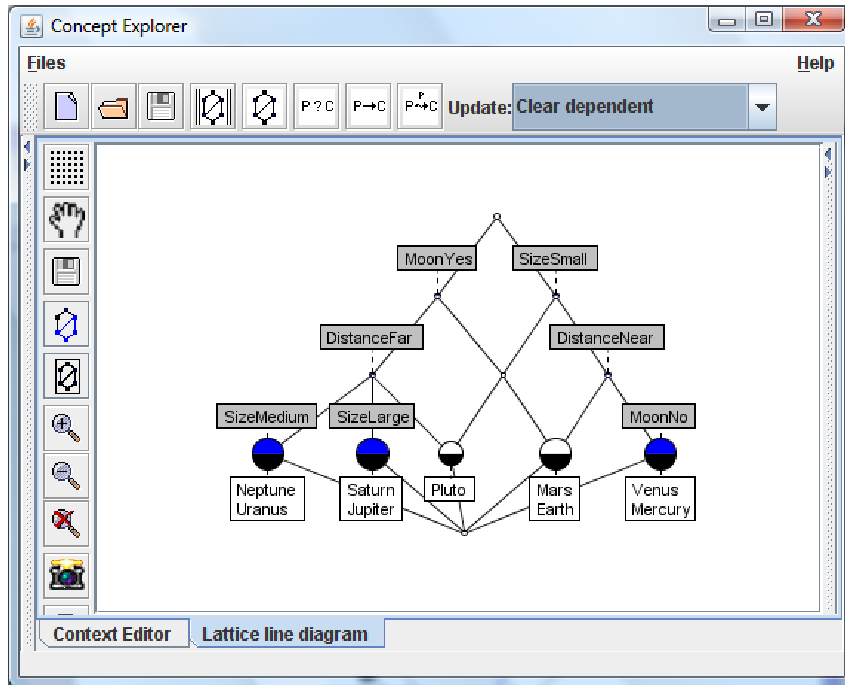

Figure 2.11: *Planets* Concept Lattice in Concept Explorer Tool

ConExp provides the functionality of “context processing”, consisting of “context editing”, “calculation of arrow relations”, and “reduction and purifying of context”. Also, the FCA operations such as “defining concepts count”, “calculating set of all concepts”, “construction of line diagrams”, “finding bases of implications and association rules holding in formal context”, “performing attribute exploration”, and “building concept lattices” are included.

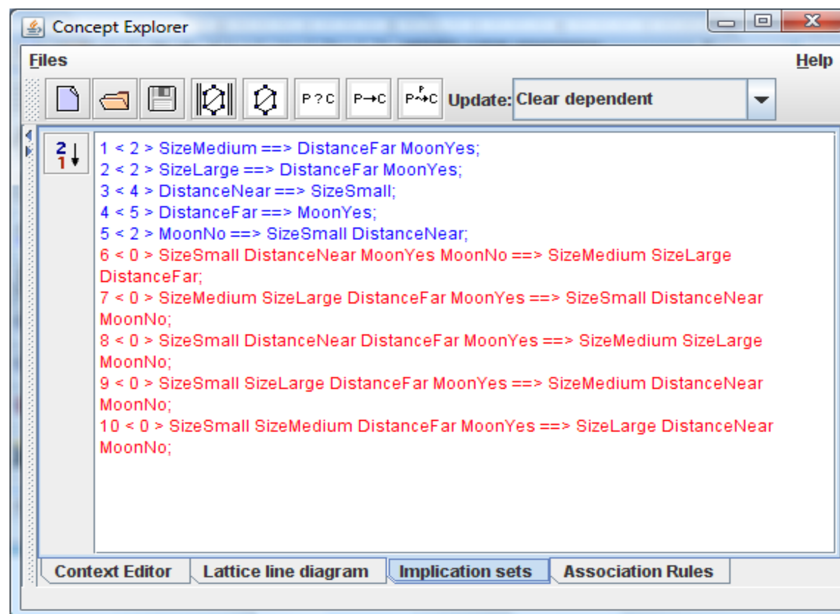

Figure 2.12: *Planets* implication sets in Concept Explorer Tool

ConExp allows working with several different data formats. However, it is recommended to use “cex” as the ConExp native format. This is XML-based format that stores information about context, lattice line diagram, and also, whether implications and/or associative rules were calculated. ConExp consists of two parts: GUI front-end and Library for performing experiments with algorithms. There are some useful features like compressed option on the context editor to give a better overview on large contexts. The algorithm developed for finding bases of implications which holds in context is based on FCA notion of pseudo-intents (Duquenne–Guigues basis) [28] and is based on

top-down approach like algorithms for calculating set of concepts and building line diagram.

2.1.4.2 ToscanaJ. ToscanaJ [11] was first implemented to realize the idea of Conceptual Information Systems which allow the analysis of data using concept-oriented methods. After ten years of development, the ToscanaJ suite provided programs for creating and using Conceptual Information Systems. Implemented as an open-source project and embedded into the larger Tockit [78] project, ToscanaJ is also a starting point for creating a common base for software development for Formal Concept Analysis. More than the older versions of Toscana it is open for extensions to support more advanced methods for conceptual analysis and retrieval of data. It is also developed as open source project on Sourceforge [71].

While ToscanaJ supports memory-mapped systems, its full potential can only be used in combination with a relational database system. If ToscanaJ runs connected to a relational database, the conceptual system engineer can customize label contents by giving SQL expressions in a specific XML syntax. To allow for easy deployment of smaller databases, ToscanaJ also comes with an embedded database engine. By this, a database engine is available in each ToscanaJ installation, which avoids the need for setting up a database engine or being bound to Windows and the Jet Engine (the database engine behind MS Access) with all its limitations. The engine embedded in ToscanaJ does not need any setup at all. ToscanaJ will just read an SQL script defining the database with “Create Table” and “Insert Into” statements and execute it on an internal database system.

An editor called *Elba* has been developed to assist the conceptual system engineers in creating ToscanaJ systems based on data stored in a relational database. A

screenshot of Elba editing a line diagram is shown in Figure 2.13 [11]. When creating a new system, the user is presented with a dialog to choose the type of database to connect to (Figure 2.14-left) [11]. After choosing one type, the user enters the necessary information for connecting to the specific database. Then Elba retrieves information about the available data tables and the names of their columns and presents it in the last step of the dialog (Figure 2.14-right) [11], in order to support the specification of a mapping from the data table into a many-valued context. When this step is completed, the user can start defining the conceptual scales. The available methods to create scales are:

- The method Attribute List is to be used as a first step towards the implementation of logical scaling [15]. Attributes are defined by SQL clauses and Elba creates the corresponding lattice by supporting all possible combinations.
- Context Table method is flexible that allows the user enter arbitrary strings for objects and attributes and select the incidence relation as required.
- Nominal Scale method enables the user to select single values from the set of all values as attributes of the scale. The user can also combine values using logical connectives.
- Ordinal Scale is used for ordered values represented by numbers. The resulting line diagram is a simple chain. The user simply enters the separating values, how they should be ordered and if an object with the exact value belongs to the upper or lower node. As a variation, interordinal scales can be created.
- Grid Scale allows the user to build the product of two ordinal scales in one diagram. If both scales refer to the same many-valued attribute, the resulting scale is the standard interordinal scale. If different attributes are chosen, the diagram visualizes the direct product of the two ordinal scales.

After the creation of the scale, a diagram with the same name is created.

ToscanaJ also creates nested diagrams, although only one level of nesting is produced. With both simple and nested diagrams, a highlighting function is available. Whenever the user clicks on a node, its filter and ideal are highlighted with stronger colors, while the rest of the diagram is slightly faded. Figure 2.15 shows the screenshot of two nested diagrams [27]. In the analysis process, clicking on one node results in a filter process, thus only the objects belonging to the selected node will be used for the following analysis.

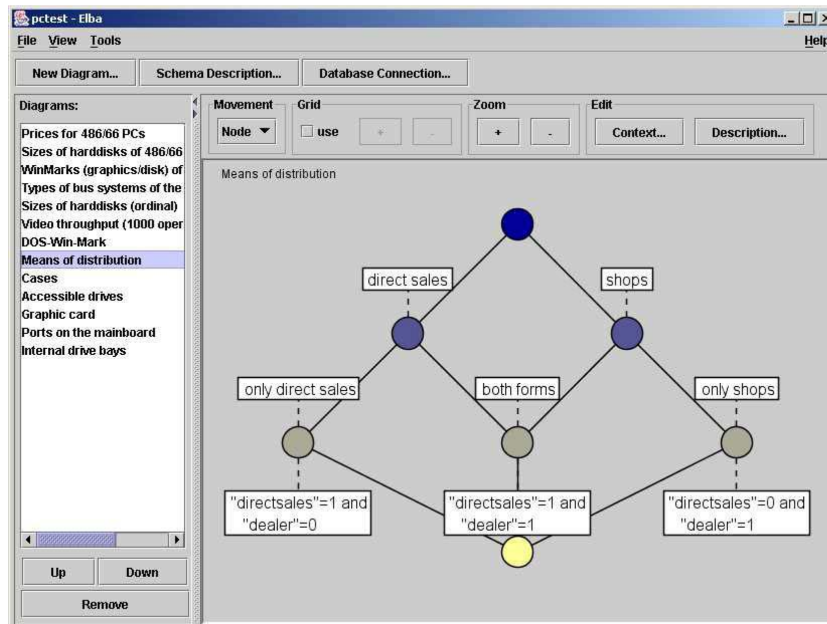

Figure 2.13: Elba's main window while a diagram is edited

ToscanaJ is the first Toscana running on multiple platforms and it can be run without a database. It aims at users who have only basic knowledge of FCA and not as conceptual system engineer. It is able to edit a many-valued context as spreadsheet view and it can import several formats like csc files, cxt-format, and XML output. Once the data is entered, either by import or by entering it manually, the many-valued

attributes can be scaled step by step to create lattices and thus diagrams. Finally, the user interface has become more intuitive while offering the same relevant features [27, 11].

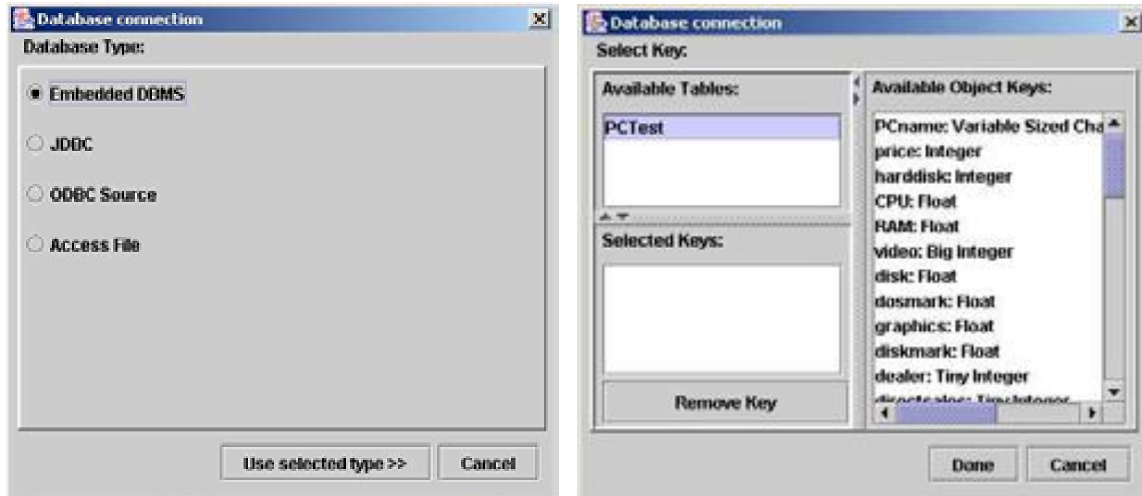

Figure 2.14: Dialogs for defining the database connection in Elba

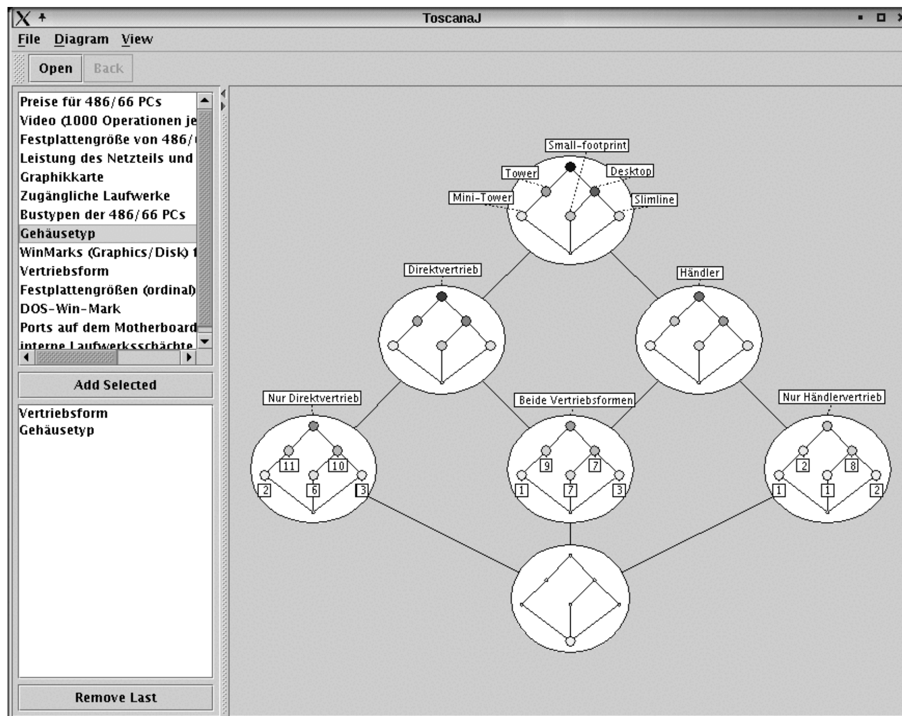

Figure 2.15: Screenshot of ToscanaJ with nested diagram and highlighting

2.1.4.3 Lattice Miner. The initial objective of the FCA tool called Lattice Miner [59] was to focus on visualization mechanisms for the representation of concept lattices, including nested line diagrams [28]. Later on, many other interesting features were integrated into the tool. Lattice Miner is a Java-based platform whose functions are articulated around a core. The Lattice Miner core provides all low-level operations and structures for the representation and manipulation of contexts, concept lattices and association rules. Mainly, the core of Lattice Miner consists of three modules: context, concept and association rule modules. The user interface offers a context editor and concept lattice manipulator to assist the user in a set of tasks. The architecture of Lattice Miner is open and modular enough to allow the integration of new features and facilities in each one of its components [13].

- **Context Module:** The context module offers all the basic operations and structures to manipulate binary and valued contexts as well as context decomposition to produce nested line diagrams. Basic context operations include apposition, subposition, generalization, clarification, reduction, and the complementary context computation. Apposition and subposition operations [83] are intended to ease the visualization of large concept lattices and do not have straightforward computational interpretation. Apposition is the horizontal concatenation of partial contexts sharing the same set of objects. Subposition, or vertical assembly of contexts upon a common attribute set, is dual to apposition [28]. Generalization of objects and attributes [13] is another way to get an abstract view of data since it allows in most cases to reduce the size of concept lattices. The module provides also the arrow relations for context reduction and decomposition, which are the methods proposed [28] to simplify lattice display. The tool provides the definition of binary context table (.lmb file format), valued

context table (.lmv file format), and nested context table (.lmn file format). Also, the tool recognizes the binary context produced by ConExp software tool (.cex file format), and Galicia SLF binary context (.slf file format). Figure 2.16 depicts the binary context editor where three levels of nesting are defined [13].

Attributes	Level 1		Level2			Level3		
	d	g	b	e	i	c	f	h
1		X	X					
2		X	X					X
3		X	X			X		X
4		X			X	X		X
5	X		X				X	
6	X		X			X	X	
7	X			X		X		

Figure 2.16: Binary Context Editor

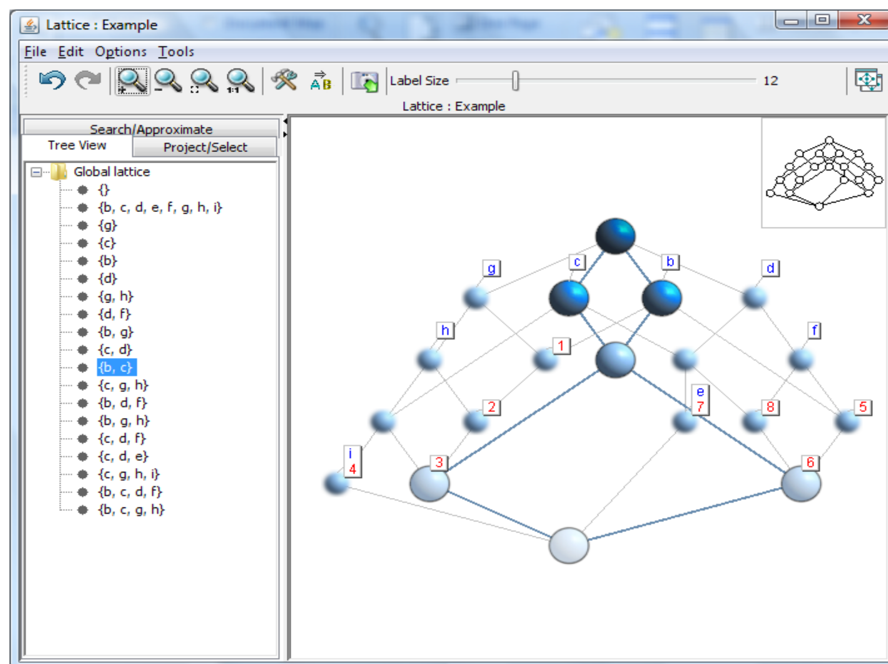

Figure 2.17: Concept Lattice, tree view structure

- **Concept Module:** The main function of the concept module is to generate the concepts of the current binary context and construct the corresponding lattice [13] (Figure 2.17). It provides the user with basic operators such as projection, selection, and exact search as well as advanced features like pair approximation. Some known algorithms such as Bordat's procedure, Godin's algorithm and NextClosure algorithm [25] are included in this module.
- **Association Rule Module:** This module [13] includes procedures for computing the Duquenne-Guigues base using the notion of pseudo-intent, generic base using the notion of generator, and informative bases of approximate rules using the notion of generator. Implications with negation [50] can be obtained using the apposition [83] of a context and its complementary. This module embeds also procedures for the computation of a non-redundant family C of implications and the closure of a set Y of attributes for the given implication set C.
- **User Interface:** The initial objective of Lattice Miner was to focus on lattice drawing and visualization either as a flat or nested structure by taking into account the cognitive process of human beings and known principles for lattice drawing (e.g., reducing the number of edge intersections, ensuring diagram symmetry). Some well-known visualization techniques such as focus & context and fisheye view have been implemented. The basic idea behind focus & context visualization paradigm is to allow a viewer to see important objects in full detail in the foreground (focus) while at the same time an overview of all the surrounding information (context) remains available in the background. The focus & context paradigm is translated into clear and blurred elements while the size of nodes and the intensity of their color were used to indicate their importance. Various forms of highlighting, labeling and animation are also provided.

Nested line diagram (NLD) [28] is a visualization means that allows the drawing of a concept lattice as a sub-structure of the product lattice of a set of lattices by combining their respective line diagrams into a nested structure. Nested line diagrams are offered to better handle the display of large lattices. Figure 2.18 shows the third level of the nested line diagram corresponding to the binary context of Figure 2.16 [13]. Each one of the inner nodes of this diagram represents a combination of attributes from the previous two (outer) levels. Real inner concepts (see the node on the left hand-side of the diagram) are identified by colored nodes while void elements are in grey color. Each node of levels 1 and 2 can be expanded to exhibit its internal line diagram. Both flat and nested diagrams can be saved as an image. Simple (flat) lattices can also be saved as an XML format file [13].

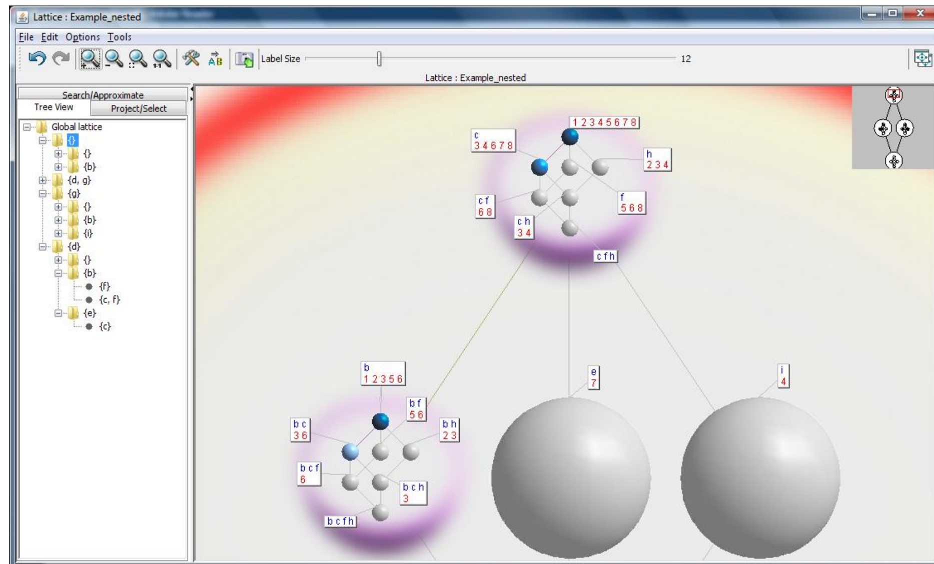

Figure 2.18: Nested line diagram

2.2 Ontology

A most commonly cited definition of ontology is the one offered by Gruber: "Ontology is a formal explicit specification of a shared conceptualization" [33]. A conceptualization is an abstract, simplified view of the world that we wish to represent for some purposes. Every knowledge base, knowledge-based system, or knowledge-level agent is committed to some conceptualization, explicitly or implicitly. A conceptualization, in this context, refers to an abstract model of how people think about things in the world, usually restricted to a particular subject area. An explicit specification means the concepts and relationships of the abstract model are given explicit terms and definitions [34].

Ontologies are used in artificial intelligence, the Semantic Web, systems engineering, software engineering, and information architecture as a form of knowledge representation about the world or some part of it. In computer science, ontology is a formal representation of the knowledge consisting of the concepts in a domain (classes), properties of each concept describing various features and attributes of the concept (roles or properties), and restrictions on properties (role restrictions). Ontology together with a set of individual instances of classes constitutes a knowledge base [52].

Building ontologies is difficult, time-consuming, and expensive, particularly if the goal is the design of an ontology that is formal enough to support automated inference. One reason is that, ontologies require consensus across a community whose members may have radically different visions of the domain under consideration. In practice, the quest for consensus is dealt with in a variety of ways. At one extreme, small lightweight ontologies are developed by large numbers of people and then merged. At the other extreme, rigorous formal ontologies are developed by consortia and standards

organizations. In the former case, there will be a greater need for ontology mapping and merging, while the latter case will require better support for collaborative design and analysis [34].

Although ontologies were originally motivated by the need for sharable and reusable knowledge bases, the reuse and sharing of ontologies themselves is still limited because the ontology users (and other designers) do not always share the same assumptions as the original designers. It is difficult for users to identify what the implicit assumptions were and to understand the key distinctions within the ontology [34]. Besides, the concepts included in ontology and the hierarchical ordering will be arbitrary to a certain extent, depending upon the purpose for which the ontology is created. This arises from the fact that objects are of varying importance for different purposes, and different properties of objects may be chosen as the criteria by which objects are classified. Subsequently, "domain-specific" ontologies are more applicable to industrial problems, but may be less reusable than generic ontologies.

The main reasons to develop ontology are [62]:

- To share common understanding of the structure of information among people or software agents
- To enable reuse of knowledge
- To make domain assumptions explicit
- To separate domain knowledge from the operational knowledge
- To analyze domain knowledge
- To increase interoperability among various domain of knowledge
- To enhance scalability of new knowledge into the existing domain
- To search/reason a specific knowledge in a domain knowledge

Ontology is not only a hierarchy of terms, but also a fully axiomatized theory about the domain. Generally, applications of ontology can be classified in different categories

that one of them is “Ontology as Specification”. Ontology of a given domain is created and it provides a vocabulary for specifying requirements for one or more target applications. In this case, ontology can be viewed as a domain model. The ontology is used as a basis for specification and development of domain applications, allowing knowledge reuse. Thus, ontology development is one approach that has contributed to the early stages of domain analysis [3]. The captured conceptualization and relations should be formally specified. OWL can be used to formally represent the results of domain analysis.

2.2.1 Ontology Web Language (OWL)

OWL is a standard development language from the World Wide Web Consortium (W3C) [23] that facilitate describing the concepts in a domain and also the relationships holding between concepts. It provides sets of operators like intersection, union and negation for concept classification and analysis. Since it is based on logical models, OWL can benefit from the use of the reasoner which checks the consistency of all concepts and definitions in the ontology and also recognizes which concepts fit under which definitions so that it can maintain the class hierarchy correctly. This is particularly useful when dealing with cases where classes can have more than one parent [39].

OWL ontologies may be categorized into three species or sub-languages: *OWL-Lite*, *OWL-DL* and *OWL-Full*. A defining feature of each sub-language is its expressiveness. OWL-Lite is the least expressive sub-language while, OWL-Full is the most expressive one. The expressiveness of OWL-DL falls between that of OWL-Lite and OWL-Full. OWL-Lite and OWL-DL are based on Description Logics with less expressiveness compared to OWL-Full. Description Logics are a decidable fragment of First Order Logic,

so can be used in automated reasoning. Therefore, it is possible to compute the classification hierarchy automatically and check for inconsistencies in an ontology that conforms to OWL-DL or OWL-Lite. The choice between OWL-Lite and OWL-DL may be based upon whether the simple constructs of OWL-Lite are sufficient or not. But still this checking depends strongly on how the ontology has been defined. OWL-Full is the most expressive OWL sub-language. It is intended to be used in situations where high expressiveness or powerful modeling facilities such as meta-classes is more important than being able to guarantee the decidability. It is therefore not possible to perform automated *reasoning* on OWL-Full ontologies [39]. Reasoning involves: (1) syntax checking, (2) consistency checking, ensuring that the ontology does not contain contradictory facts (3) subsumption, checking whether a class description is more general than another class description, and (4) query answering, retrieving knowledge from the knowledge base [67, 32].

2.2.2 Ontology Tools

Many different tools are available for building and maintaining ontologies. The most well known and widely used ontology tools available on the market are Protégé, TopBraid Composer, CMapTools Ontology Editor (COE), Altova SemanticWorks, and SMORE/SWOOP; among which most three important ones are described in this section. A set of necessary requirements have been made to evaluate the mentioned software tools. The resulting table can be found in Appendix A. Based on these requirements each tool has been evaluated and assigned a number on a 1 for poor – 10 for excellent scales in this Table [62]. In this survey, the tool TopBraid Composer is utilized for opening and representing the target OWL ontology.

2.2.2.1 Protégé. Protégé [57] is a free, open-source ontology editor/creator and knowledge-base framework and perhaps the most widely-used ontology creation tool on the market. Using protégé, ontologies can be edited and created using RDF/OWL script language (including OWL Full, DL and Light) or through its java-based plug-and-play environment. This environment provides a tabbed view of ontology, allowing the user to separate the ontological elements and look at all of the characteristics and relationships attributed to each object. Files can be exported to Clips, OWL, N-Triple and TURTLE formats.

Despite its ease of use compared to many other commercial and open-source ontology editors, Protégé does require a fundamental knowledge of ontology and its defined types of objects and relationships.

OwlViz is a mapping visualization plugin designed for Protégé. It allows the user to view an ontology as a concept map. However, OwlViz does not illustrate the relationships between each object, nor does it allow the user to create or edit the ontology within this view. Similar plugins include OntoViz and Techquila are used, although OwlViz is the better of the three.

2.2.2.2 TopBraid Composer. TopBraid Composer [80] is a professional development environment for W3C's Semantic Web standards RDF Schema, the OWL Web Ontology Language, the SPARQL Query Language and the Semantic Web Rule Language (SWRL). Composer provides a comprehensive set of features to cover the whole life cycle of semantic application development. In addition to being a complete ontology editor with refactoring support, composer also can be used as a run-time environment to execute rules, queries, reasoners and mash-ups.

Based on Eclipse, Composer can also be extended with custom Java plug-ins. This supports the rapid development of semantic applications in a single platform. Composer can be used to edit RDFS/OWL files in various formats, and also provides scalable database backends (Jena, AllegroGraph, Oracle 10g and Sesame) as well as multi-user support.

It has a number of features which make it a very competitive tool comparable to Protégé. Among these features are Integrated Development Environment for Semantic Web applications, UML-like Class, Diagrams, classification and Consistency Checking, Rules, SPARQL Queries, Data Source Mapping, Geography and Location Mapping, Calendar and Chart Mash-Ups, Semantic Web and Mash-Up development with RDFs and GRDDL, Visual RDF Graphs, Multi-User Support, Ontology-Driven Forms, Form Customization, Source Code Editing, Imports and Namespace Management, Import of Databases, UML, XML Schema and Spreadsheets, and HTML Documentation Generation.

TopBraid Composer is a very flexible platform that enables Java programmers to add customized extensions or to develop stand-alone Semantic Web applications. For example, it is fairly easy to add new kinds of windows, editors, menu entries or even new storage formats to Composer. One of the advantages of Composer is that it can serve as an application development framework: programmers can develop components as plug-ins and benefit from the rich features of Composer to run experiments and tests against real-world data. When the functionality has been sufficiently tested, the module can be deployed into a stand-alone application, especially on the TopBraid platform.

2.2.2.3 CMapTools Ontology Editor (COE). CMapTools [41] allows users to construct, navigate, share and criticize knowledge models represented as

concept maps. The COE application provides users with an outlet to create ontology in the form of concept maps. This was the version we evaluated extensively. CMap Server allows a group to collaborate online and provide feedback to one another.

CMapTools allow the user to import various types of XML and text documents and export ontologies in OWL, N-Triple (and its various formats) and TURTLE. It offers validation and concept suggestion tools. CMaps is a very appealing tool for our team's purpose as it is the only toolset which is primarily a mind/concept mapping tool with ontological features. The intended users of the tool require the ability to create maps that can be loaded by our ontology experts in ontology software and vice versa.

One of the major benefits of CMap Tools is that users need only a very fundamental understanding of ontology (mostly the types of relationships they must define). The ontology can then be created as a concept map using a simple drag and drop interface. A styles template also allows the user to quickly and easily customize their objects, lines and map in general. When loading ontology into CMap, it recognizes the types of relationships used and provides the repository of relationships to choose from when creating a relationship within the concept map, which is very helpful for anyone working on an ontology created by another author.

The zooming in and out is fairly limited and navigating large concept maps can be annoying when you need to work on multiple areas of ontology. So, CMap provides a Web Service where developers use the language of their choice and the Knowledge Exchange Architecture of CMapTools and the CMapServer known as KEA. The API for KEA consists of an XML specification for Web services interface. This allows programmers to create their own modules that provide some features that CMap currently do not provide, or possibly enhance an existing feature.

2.3 Model Transformation

The motivation behind this short survey is to emphasize model transformation as a formal method in software design process, which in some level guaranties consistency and correctness of existing components. The model transformation process consists of starting from an initial requirements elicitation step and to continue through feeding acquired information to a formal framework known as FCA. This framework provides a formal model based on concept lattices that remains in high-level abstract layers and needs to be transformed to more practical and convenient models. To automate the transformation process, Model Transformation approaches with an extend domain of methods and tools are available. In this section we will not go deep into different Model Transformation methods but on the other hand we will search for the best approach and method that is convenient for our model.

A model transformation in Model-Driven engineering takes as input a model conforming to a given Meta-model and produces as output another model conforming to a given Meta-model. To achieve this goal, there are many tools that support the automation of model transformation. These development tools not only offer the possibility of applying predefined model transformations on demand, but also offer a language that allows (advanced) users to define their own transformation rules and execute them on demand.

Performing a model transformation, taking one or more models as input and producing one or more models as output, requires a clear understanding of the abstract syntax and the semantics of both the source and target. A common technique for defining the abstract syntax of models and the inter-relationships between model elements is Meta-modeling. To define the required Meta-models, we need an

appropriate data schema to express input and output models. XML (Extensible Markup Language) is specially designed to be easy to use over the Web, to be human-readable and straightforward for applications to read and understand. XML is quickly becoming the universal syntax for information transfer; therefore a vast amount of XML transformation has XML as the destination as well as the source. The tools applied in this thesis will manipulate input and output models taking advantage of XML format. For example, the input model of transformation process is produced by an FCA tool known as Lattice Miner that creates the concept lattice in XML schema. Also, the final target model of this framework is considered to be an OWL ontology that certainly is in XML format.

Since our input and output models can be serialized as XML format using the XML Metadata, implementing model transformations using XSLT, which is a standard technology for transforming XML, seems very attractive. Extensible Stylesheet Language Transformation (XSLT) is an XML-tool to perform model transformation. It defines the mapping from some XML into another markup language like XML, HTML, or into plain text. XSLT Stylesheets are interpreted by XSLT processors, which generate a result from source XML. XSLT processors can be embedded in web browsers or be run from the command line to run Stylesheets. (Figure 2.19 [74])

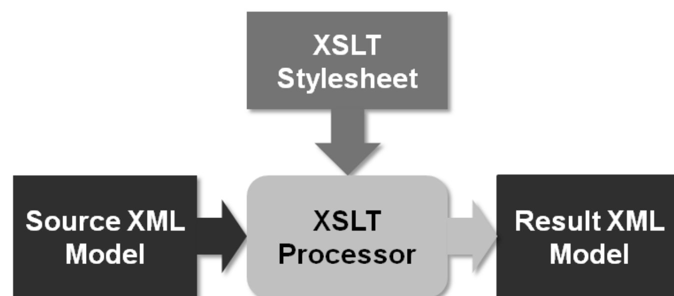

Figure 2.19: XSLT processor

XSLT uses XPath to select parts of XML to process and to perform calculations. XPath, the XML Path Language, is a query language for selecting nodes from an XML document. The most important role of XPath is to collect information from an XML document by navigating through the document. A secondary role of XPath is as a general expression language, to perform calculations. There are two types of implementing approaches in XSLT transformation: Push and Pull.

In the push approach, multiple templates are used, each matching different types of nodes to process a document-oriented XML. The contents of the input XML get pushed through the Stylesheet to be transformed. The final structure of the result is highly determined by the structure of the input.

In the pull approach, some special nodes are selected to change the order in which the input is processed or to only process certain portions of the input. Mostly, pull method is used for data-oriented XML when the structure of the input XML is fixed and it is obvious what exact result is going to be obtained from the input. The final structure of the result is mainly determined by the structure of the Stylesheet and how the templates fit together.

The best Stylesheet uses a hybrid of both approaches to process different parts of a particular XML document. In this survey, four Stylesheets have been created for implementing the transformation algorithms from FCA to ontology. All defined Stylesheets use both approaches to process different parts of input XML files. The templates are used to match the nodes that get pushed to the output XML files and the specified nodes are selected to change the structure of files.

Chapter 3

Case Study Statement

This Chapter explains the problem used as a case study in this thesis. The problem, called Common Component Modeling Example (CoCoME) [36] has been given as the benchmark case study by the component based software engineering community. This case study is the test bed used to compare the merits and drawbacks of different component based development techniques.

The author of [40] applied the trustworthy component-based methodology in [48] to *CoCoME* and showed that the approach in [48] is quite general to formally model such problems. However, the author of [40] identified the basic components only manually and created them using the Visual Modeling Tool [89]. In this thesis we use FCA to create ontology together with some semantic information and constraints on it which can be regarded as the domain model of the problem. The significance is that, to the best of our knowledge, FCA was never used by the software engineering research and development community as a means of formal domain analysis. OWL ontology is automatically constructed from the domain model and then it is automatically transformed into the trustworthy architecture language (TADL [49]) of the target component-based system.

First, the case study is briefly introduced. The detailed description of the case study can be found in [36]. Afterwards, the transformation tool in [40] is addressed as a tool to generate UPPAAL specification of CoCoME case study to verify the trustworthy properties. Next, the drawbacks of the solution stated in [89] and [40] are presented, and finally, we explain the work done in this thesis as a resolution to these problems.

3.1 Common Component Modeling Example (CoCoME)

CoCoME [36] is a common component modeling example that has been introduced by the component development community in order to evaluate and compare the practical application of existing component models using a common component-based system as a modeling example. CoCoME includes properties of real world systems and its size is limited to be modeled with reasonable effort. Besides, it comes from a domain which is easily understandable without heavy-weighted system requirements specifications.

CoCoME is a trading system which includes all transactions concerning the sales in a supermarket, starting from the customer interaction at cash desk, product scanning, payments, and inventory updates. Also, it includes the management considerations like ordering goods from wholesalers, generating various kinds of reports and even the product exchange process on low stock which is systematically manipulated.

3.2 System Overview

The first element of the trading system in CoCoME case study is the “Cash Desk”. Figure 3.1 extracted from [36], depicts an overview of the cash desk where the customer

purchases the products and pays for them. The cashier scans the products and handles the payment. Furthermore, the system provides an express checkout to speed up the sale process in which the customer can order only a few goods and the payment must be in cash.

The cash desk consists of the following devices:

- Cash Box: starts and finishes the sale transaction, and holds the received cash.
- Barcode Scanner: identifies the products being purchased.
- Card Reader: handles card payments (cash payments are handled by the Cash Box).
- Printer: prints the bill to be handed out to the customer at the end of the sale transaction.
- Light Display: signals the customer the current mode of the cash desk to identify if it is in normal mode or in express mode.
- Cash Desk PC: handles the sale transaction, communicates with the Bank, and integrates all devices at the cash desk.

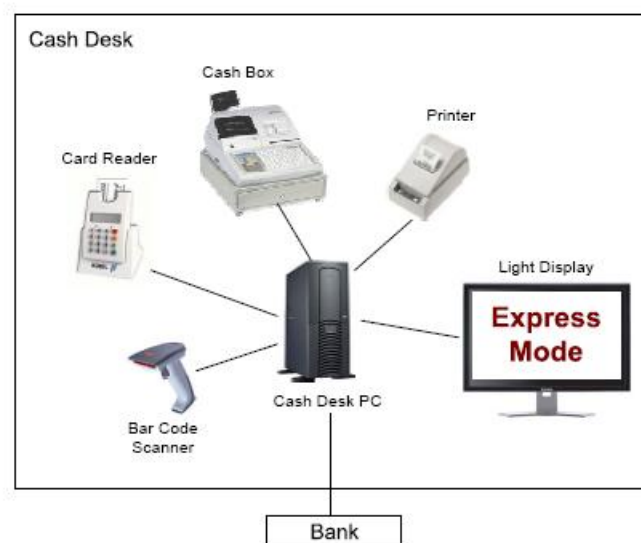

Figure 3.1: Hardware overview of Cash Desk

Each store consists of several cash desks connected to a Store Server and a Store Client in a network. The set of cash desks are called cash desk line. The manager is authorized to order products, view reports, change price, and administer the inventory by using the Store Client. Each store is connected to an Enterprise Server which in turn is connected to an Enterprise Client.

3.3 System Requirements Specification

The CoCoME case study introduces the trading system by defining the specifications of its use cases. The trading system use cases including all actors are presented in Figure 3.2 [36]. In this section, a brief description of the use cases including the functional and non-functional requirements of the trading system is presented. More details can be found in [36].

3.3.1 Process Sale

Purchasing goods by customers in store is provided in the Process Sale use case. The cashier is the only actor who interacts with the customer in this process when the customer presents the items to buy at the cash desk. The cashier begins the new sale process by pressing the *start new sale* button. Then the cashier enters the item identifier manually using the keyboard from the Cash Box or by using the Barcode Scanner. The system shows the product description, price and running total. Till now, one purchasing item is registered in the system. This process is repeated until the cashier ends entering items by pressing the *sale finished* button at the cash desk.

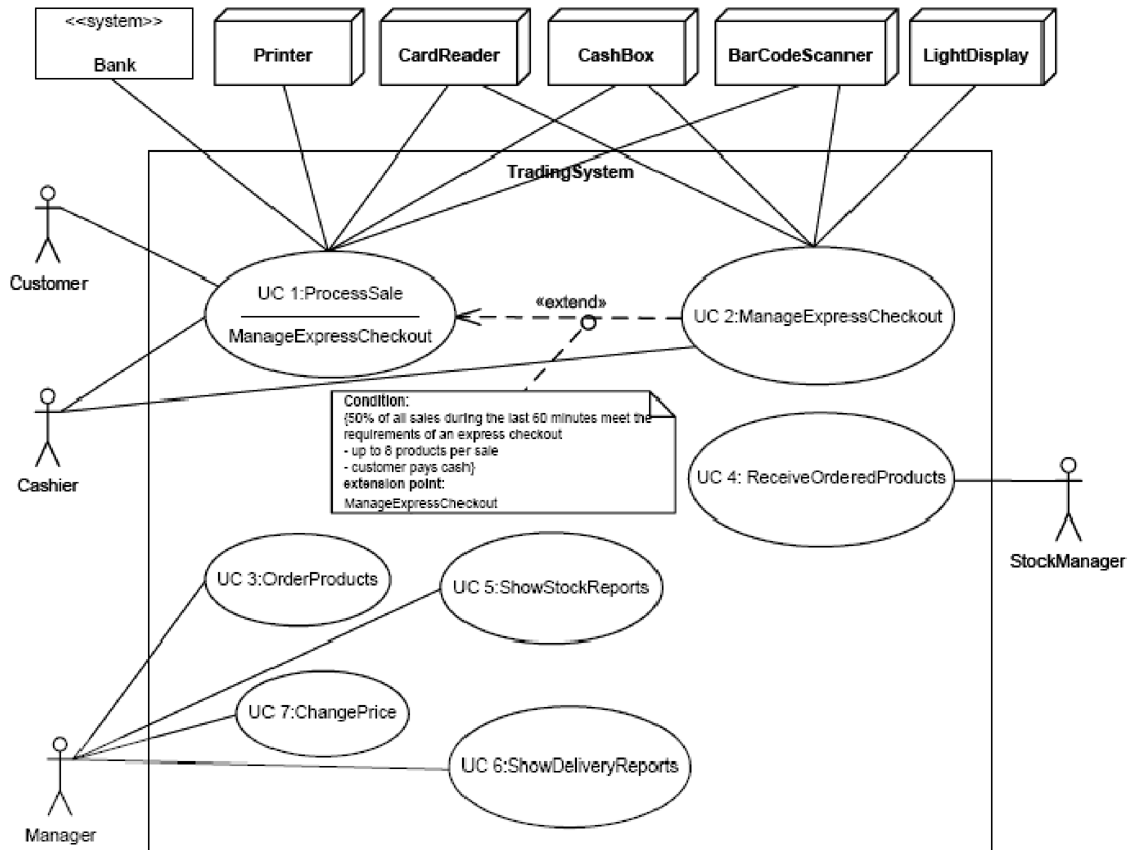

Figure 3.2: Trading system use cases

There are two options for the payment, Bar Payment and Card Payment. The cashier initiates bar payment by pressing *bar payment* button at the Cash Box. The cashier enters the cash received from the customer using the Cash Box and hands over the change and closes the Cash Box. Likewise, the cashier initiates card payment by pressing *card payment* button at the Cash Box. The customer's card is pulled though the Card Reader and after the Bank's approval the card is returned to the customer. In the case of card refusal, the card payment can be turned into a bar payment by pressing the *bar payment* button. Afterwards, the completed sale is registered in the inventory and the stock is updated. If the inventory is not available, the system saves the sale and loges it as soon as the inventory is available again. Finally, the transaction is finished by

printing a receipt which is handed on to the customer. The time requirements for processing the sale transaction are the following:

- Time for pressing the *start new sale* button = 1.0s;
- Time for printing the bill = 3.0 s;
- Time for pressing the *sale finished* button = 1.0 s;
- Time for updating the inventory = 2.0 s;
- Time for processing a bar payment = 120.0 s;
- Time for pressing the *bar payment* button = 1.0 s;
- Time for pressing the *card payment* button = 1.0 s;
- Time for waiting for card validation = 30.0 s;
- Time for scanning an item = 5.0 s;
- Time for showing product description, price, and running total = 1.0 s;

3.3.2 Manage Express Checkout

Changing Cash Desks from the normal mode to the express mode is provided in the Manage Express Checkout use case. However, the system enables the cashier to change it back to the normal mode by pressing the *disable express mode* button. This use case is triggered by the system when the condition for the express checkout is met, i.e., if 50% of all sales during the last 60 minutes has less than 8 items each. In this case, the Cash Desk which finishes the last sale will be changed to the express mode. So, the Light Display is switched to green and the maximum items per sale would be 8. Moreover, card payment is not allowed and the customer has to pay just in cash.

In the case of deactivating the express mode by the cashier, the Light Display is switched from green to black and the limitation on the number of items per sale is

removed. Besides, the customer is capable to pay either by card or in cash. The time requirements for managing an express checkout are as follows:

- Time for pressing the *disable express mode* button = 1.0 s;
- Time for switching to express mode = 1.0 s;
- Time for deactivating card payment = 1.0 s;
- Time for switching the green Light Display on = 1.0 s;

3.3.3 Order Products

Ordering new product items is provided in the Order Products use case. The manager is the only actor who has the authority to do this process at the Store Client in the case of supplying the store with some new products. At first, two lists of products are demonstrated by the system: one is the list with all products; the other is the list with products which are running out of stock. The manager chooses the products to order and enters the corresponding amount for each product item, and then presses the *order* button at the Store Client. The system sends the orders to the appropriate suppliers and generates an order identifier for each. Then the results are presented to the manager. The time requirements for ordering products are as follows:

- Time for pressing the *order* button = 1.0 s;
- Time for displaying the lists of products = 1.0 s;
- Time for entering the order and its amount = 10.0 s;

3.3.4 Receive Ordered Products

Accounting the ordered products which are newly arrived at the store is provided in the Receive Ordered Products use case. The stock manager is the actor who has the authority to do this process at the Store Client when the ordered products arrive at the store. The attached order identifier which has been assigned during ordering the products is verified by the stock manager. If the delivery is complete and correct, the stock manager enters the order identifier and presses the *roll in received order* button. Then, the inventory is updated by the system. In the case that the delivery is not complete or correct, the stock manager sends the products back to the supplier and waits for the new delivery but no changes are registered at the system. The time requirements for receiving ordered products are as follows:

- Time for pressing the *roll in received order* button = 1.0 s;
- Time for updating the inventory = 1.0 s;
- Time for displaying the ordered list = 1.0 s;

3.3.5 Show Stock Reports

Generating stock-related reports at the store is provided in the Show Stock Reports use case. The manager is the only actor who has the authority to see the statistics about the store in the reporting GUI. In this case, when the manager enters the store identifier at the Store Client and presses the *create report* button the system displays a report including all available stock items in the store. The time requirements for showing stock reports are the following:

- Time for entering store Id and pressing the *create report* button = 1.0 s;

- Time for generating the stock report = 1.0 s;

3.3.6 Show Delivery Reports

Generating delivery-related reports about the enterprise is provided in the Show Delivery Reports use case. The manager is the only actor who has the authority to see the statistics about the enterprise in the reporting GUI In this case, when the manager enters the supplier identifier at the Store Client and presses the *create report* button the system displays a report including the calculated mean time to delivery for each supplier of a specific enterprise. The time requirements for showing delivery reports are the following:

- Time for entering supplier Id and pressing *create report* button = 1.0 s;
- Time for generating the delivery report = 1.0 s;

3.3.7 Change Price

Changing the price of the products is provided in the Change Price use case. The manager is the only actor who has the authority to change the price of the products in the store at the Store Client. At first, a list of all available products in the store is demonstrated. The manager selects a product item and changes its price, and then presses ENTER to confirm it. The system changes the price and updates the inventory. From now on, the product will be sold with its new price. The time requirements for changing the price are as follows:

- Time for pressing ENTER = 1.0 s;
- Time for displaying the price list = 1.0 s;

- Time for entering the new price = 5.0 s;
- Time for updating the inventory = 5.0 s;

3.3.8 Product Exchange

Verifying the stock level for each product and sending a request to the Enterprise Server for such products are done in the Product Exchange use case. The Enterprise Server checks whether the required products are available at other stores and makes the necessary calculations to see if it is economical to ship the product to the requesting store. This Use Case is triggered by the system while only servers are involved as the actors. If the Store Server is connected to the Enterprise Server, the query of product shortage including its product identifier is sent. Otherwise, the query is queued and whenever the Enterprise Server is available it will be re-sent again. When the Enterprise Server receives the query, it determines all nearby stores which are less than 300 km away from the requesting store. Since the Enterprise Server does not have the current global data about the stores at any time, it asks them to flush their local data to the Enterprise Server in order to update its database. The time requirements for product exchange functionality are as follows:

- Time for query from the requesting Store Server to the Enterprise Server = 2.0 s;
- Time for query from the Enterprise Server to one nearby Store Server = 2.0 s;
- Time for flushing the cache of one Store Server and returning the result = 2.0 s;
- Time for making a decision by the Enterprise Server = 1.0 s;
- Time for marking the products as incoming by the Enterprise Server = 2.0 s;
- Time for sending the delivery request to the providing Store Server = 2.0 s;

3.4 Transformation Tool for Verification Process

Design time analysis is an important step in the process of developing software systems, with the goal of ensuring that the system design conforms to the design constraints that are stated as part of the functional and non-functional requirements. There are some well-known techniques for formally analyzing a design. These include model checking, axiom-based formal verification, and real-time scheduling analysis that take into account resource constraints. The transformation tool [40] has used model checking and real-time schedulability techniques to verify that the system under development is both safe and secure. To do so, the architecture of a trustworthy system, formally described in Trustworthy Architectural Description Language (TADL), is taken as the input for the analysis stage and is transformed into behavior protocols used by existing verification tools. A tool based on such techniques has been designed and implemented which automatically generates two types of models from a TADL description. One is the UPPAAL model, on which the security and safety properties of the system under design are formally verified. The second output is the TIMES model, on which real-time schedulability analysis is performed. The techniques and tools are applied to the CoCoME case study to illustrate the transformation process from a system defined using TADL model to UPPAAL model.

UPPAAL is one of the model checking tools which is used for the modeling, simulation and verification of real-time systems, and was jointly developed by Uppsala University and Aalborg University [6]. Model checking is the most successful approach in developing tools and techniques for checking the requirements and design of software systems. Figure 3.3 extracted from [53], shows the main idea behind model checking. The model checking tool takes as an input the requirements or design (called models)

and a property (called the specification) that the system should satisfy. The output of the tool is either yes, if it satisfies the specification and no, otherwise [53].

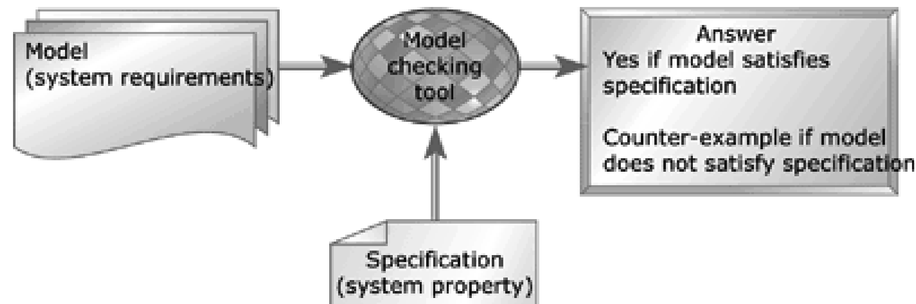

Figure 3.3: Model Checker overview

The main goal of UPPAAL tool is to model a system with timed automata using a graphical editor, simulate it to validate the behavior and then verify that the model is correct with respect to a certain set of properties. The UPPAAL verifier can be used to check the behavior of the system by defining different checking formulas. This can be done manually by the user after the system has been transformed into the UPPAAL model and opened using the UPPAAL tool. Figure 3.4 extracted from [40], shows the sample of the safety and security properties that were tested on the resulting UPPAAL model of the CoCoME case study. Also, the corresponding verification results are displayed in the status bar.

Now, a question arises and it is “what is the justification of the components?” and “how the components have been developed?” to be used in such a verification process. As proposed in [89] and [40], the application developer focuses on the modeling and analysis aspects to design a visual model of a component-based system using Visual Modeling Tool [89] without being burdened by the formalism. VMT tool provides a graphical user interface for developers to design components, connectors and system configurations along with their attributes and properties. Also, VMT tool can

automatically generate the relevant formal behavior model in TADL description language. Afterwards, the resulting TADL specification represents the input file to the Transformation Tool [40]. The tool can then perform the transformation to UPPAAL or TIMES, depending on the selected transformation type. If the UPPAAL transformation is selected, the produced XML file is provided as the input to the UPPAAL model checking tool to perform the required verification and simulation. If TIMES transformation is selected, the produced XML file is provided as the input to the TIMES tool, where schedulability analysis can be performed.

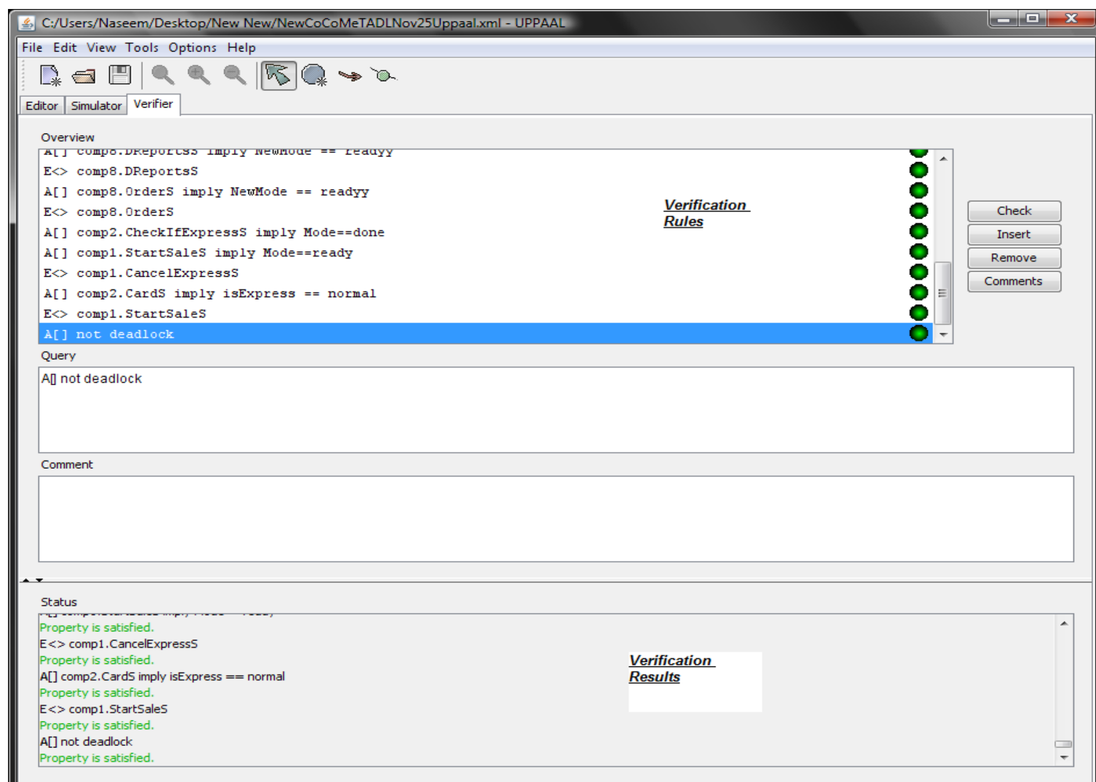

Figure 3.4: Verifying safety and security properties in UPPAAL

The proposed solution in [89] and [40] has the following drawbacks:

- The developer manually determines the system components as well as the functional and non-functional properties, intuitively and without following any

specific rules and methodologies and even without using the formal methods. As a consequence, some unpredictable deficiencies like redundancy, incompleteness, inconsistency, or contradiction may occur in system design. Even though VMT as a formal tool and TADL as a formal language are applied to fulfill the formalism, the verification process is done in the last steps of components design using model checking tools (UPPAAL tool), and/or schedulability analysis tools (TIMES tool). Actually, the analysis and reasoning about the behavior of the trustworthy components are burdened to the mentioned extended timed automata behavior models.

- The UPPAAL tool can be used to verify whether the model is correct with respect to a certain set of properties that are manually specified by the user. The properties that can be checked using this feature are Reachability, Safety and Liveness. Since the checking formulas are manually defined, some significant properties may be ignored to be verified, or may not correctly addressed by the specified formulas.
- When the verification rules of the UPPAAL model checker are performed, it may generate some results that are not satisfied by the UPPAAL verifier. These special conditions are not predicted in [40] and it does not provide any solution for them. Maybe the system developer should go back to the design phase and modify the corresponding elements or parameters in VMT tool, in order to achieve the satisfaction of the manually defined rules. However, there is not any specific guideline for the developer to reach to this goal that justifies the trustworthiness properties.

To solve the above problems, in this thesis, the Formal Concept Analysis as a mathematical theory is used to compose concept hierarchy and provide a formal basis for domain analysis that leads to design the component elements and other artifacts of

the system, including its functional and non-functional requirements. The application of FCA in the first stages of design has the advantage of constructing a consistent class hierarchy. Besides, by extracting the implication rules, the user would be able to make the logical deductions and discover the intra-concept relations between the system components and the design constraints. Afterwards, the derived system artifacts are automatically transformed to the OWL ontology elements by using model transformation process. The concept hierarchy developed in FCA is correspondingly transferred to the class hierarchy in resulting ontology and since the OWL ontology is based on logical models, the user can take advantage of using its reasoning engine to accomplish the syntax checking, consistency checking and subsumption. Therefore, if there would be any deficiency or contradiction in the developed ontology, the user can identify and fix it by modifying the relevant system elements in FCA. Finally, the verified ontology is automatically transformed to TADL architecture description language which is the formal specification of the dependable component-based system. The subsequent TADL file can be represented as the input file for the transformation tool in [40] to be transformed to UPPAAL model so that the safety and security properties of the system would be formally verified by UPPAAL model checking tool.

The methodology to be outlined in this thesis will be applied to CoCoME case study in Chapter 8, so that, the OWL ontology and the relevant components in TADL will be derived. Then, the results will be compared with the work done in [89] and [40] to justify the necessity of using Formal Concept Analysis in order to get more accurate component models.

Chapter 4

Methodology for Constructing Formal Context Tables

In this Chapter, the process for capturing the formal concepts and the rules for defining and integrating formal context tables using FCA mathematical theory are introduced. Then the concept lattice corresponding to the deduced concept hierarchy is developed. The basic idea is to construct simple formal context tables that contain partial relational information, and then combine them into a large table that is complete with respect to the relational information on the objects and attributes occurring in the use cases. Therefore, the approach for developing the entire context table of a software application consists of the following two main steps: (1) Partially defining context tables, and (2) Constructing a unified formal context table.

Identifying, defining and entering the concept definitions, especially in the case of large and complex application domains, are challenging tasks because it can be lengthy, costly, and controversial. User participation is particularly important in the early phases of software development which cannot be ignored. User has to be involved as much as possible to achieve a good and practicable analysis of the application field. Although the

beneficial techniques of text mining and some relevant tools are developed to reduce time and cost, some domain expert knowledge is required in this stage to capture the formal concepts and construct the context tables. Moreover, once the context tables are defined, they are joined and converted to a pruned concept lattice that is manually accomplished by the designer. The designer has to resolve possible conflicts and duplicates according to the rules provided in this work.

In Section 4.1, the definition of partial context tables is discussed. First, different types of context tables and how they are related to one another are described. Next, some rules are provided for composing the context tables. In Section 4.2, the integration of the partially context tables and the construction of a unified formal context table are explained. Section 4.3 presents the concept lattice containing the derived formal concept hierarchy. Finally, the advantages of the presented methodology in this Chapter are discussed in Section 4.4.

4.1 Partial Definition of Context Tables

FCA considers a binary relation *I* (incidence) over a pair of sets *O* (objects) and *A* (attributes). The relation is given by the matrix of its incidence relation (*oIa* means that object *o* has the attribute *a*) which is called a formal context [82]. Formal context Table is the most flexible and basic data structure of FCA. For a given formal context, the formal concepts, their extensions and intensions are uniquely defined and fixed. The set of all formal concepts of a formal context, made up of the closed subsets ordered by set-theoretical inclusion, forms a complete lattice, called the concept lattice. The user can derive a line diagram of the concept lattice from a given context, and conversely derive the context matching a line diagram.

The system to be developed may consist of several use cases, each containing some sets of various components. First, simple formal context tables that contain partial relational information are defined. Then the partially defined context tables are combined into a unified large table that is complete with respect to the relational information on objects and attributes occurring in the use cases. By this technique, the design phase is manipulated more precisely and efficiently.

The software designer realizes the system specifications by using domain knowledge and intuitively captures the extent objects, the intent attributes, and their binary relationships to define the formal concepts. The extent objects are the occurrences of varying conditions and different combination of attributes. The intent attributes are the features, specifications and peripherals of objects.

4.1.1 Different Types of Formal Context Tables

In FCA there are two types of context tables. These are binary context table and many-valued context table:

- **Binary context table:** Binary context table is a rectangular table with one row for each object and one column for each attribute, having a cross (x) in the intersection of row g with column m if and only if $(g, m) \in I$, where I is the incidence of the context [82]. As an example, we refer to Figure 2.2 and Figure 2.3 of Chapter 2, presenting the sample binary context table and its concept lattice.
- **Many-valued context table:** It is a context table in which the objects may have many-valued attributes. It consists of objects, attributes and the attribute values, where $(g, m, w) \in I$ is read as “The object g has the value w for the attribute m”.

To obtain formal concepts from a many-valued context, FCA offers the method of conceptual scaling [27] which transforms many-valued context tables to binary context tables. This means that a formal context called conceptual scale is defined for each of the many-valued attributes which has the values of the attribute as objects. If a many-valued context and a conceptual scale are given, we can derive the realized scale, i.e., a formal context which has the objects of the many-valued context as objects and the attributes of the scale as attributes. In the realized scale, an object has an attribute if the value assigned to the object in the many-valued context has the attribute in the conceptual scale [11]. Figures 2.4, 2.5, and 2.6 of Chapter 2 illustrate a sample many-valued context table and the concept lattice that are derived from conceptual scaling.

Lattice Miner [59, 13] is the FCA software tool employed in this thesis. It provides the definition of Binary Context Table (BCT), Valued Context Table (VCT), and Nested Context Table (NCT). The two latter tables may be automatically converted to binary context table. As an example, the *Planets* valued context table (Figure 4.1) is defined and converted to binary context table (Figure 4.2).

Lattice Miner

File Edit Lattice Rules Context Window About

Context : Planets

Context : Planets

	Planet	Size	Distance	Moon
Mercury		Small	Near	No
Venus		Small	Near	No
Earth		Small	Near	Yes
Mars		Small	Near	Yes
Jupiter		Large	Far	Yes
Saturn		Large	Far	Yes
Uranus		Medium	Far	Yes
Neptune		Medium	Far	Yes
Pluto		Small	Far	Yes

Figure 4.1: *Planets* Valued Context Table (VCT)

BCT corresponds to the FCA binary context table and VCT corresponds to the many-valued context table; while NCT is a combined table of two or more BCTs or VCTs. In NCT, the combined tables are concatenated with each other. When NCT is converted to BCT, the many-valued attributes are split into their diverse values. Lattice Miner tool differentiates the generated attributes by varying their names to make them unique.

	Planet_	Size_Large	Size_Medium	Size_Small	Distance_Far	Distance_Near	Moon_No	Moon_Yes
Mercury	X			X		X	X	
Venus	X			X		X	X	
Earth	X			X		X		X
Mars	X			X		X		X
Jupiter	X	X			X			X
Saturn	X	X			X			X
Uranus	X		X		X			X
Neptune	X		X		X			X
Pluto	X			X	X			X

Figure 4.2: *Planets* Binary Context Table (BCT)

	Att1	Att2	Att3
Obj1	a1	b1	c1
Obj2	a2	b2	c2
Obj3	a1	b3	c2

Figure 4.3: *Table1* Valued Context Table (VCT)

For example, the sample valued context tables *Table1* and *Table2* are defined and depicted in Figure 4.3 and Figure 4.4. Then, they are converted to BCTs (Figure 4.5 and

Figure 4.6). *Table3* is the nested context table of *Table1* and *Table2* shown in Figure 4.7. The NCT *Table3* is converted to binary context table (Figure 4.8).

Lattice Miner

File Edit Lattice Rules Context Window About

Context : Table2

Context : Table2

	Att4	Att5
Obj4	e1	f1
Obj5	e1	f2

Figure 4.4: *Table2* Valued Context Table (VCT)

Lattice Miner

File Edit Lattice Rules Context Window About

Context : Table1

Context : Table1

	Att1_a1	Att1_a2	Att2_b1	Att2_b2	Att2_b3	Att3_c1	Att3_c2
Obj1	X		X			X	
Obj2		X		X			X
Obj3	X				X		X

Figure 4.5: *Table1* Binary Context Table (BCT)

Lattice Miner

File Edit Lattice Rules Context Window About

Context : Table2

Context : Table2

	Att4_e1	Att5_f1	Att5_f2
Obj4	X	X	
Obj5	X		X

Figure 4.6: *Table2* Binary Context Table (BCT)

Attributes	Att1_a1	Att1_a2	Att2_b1	Att2_b2	Att2_b3	Att3_c1	Att3_c2	Att4_e1	Att5_f1	Att5_f2
	Level 1							Level2		
Obj1	X		X			X				
Obj2		X		X			X			
Obj3	X				X		X			
Obj4								X	X	
Obj5										

Figure 4.7: *Table3* Nested Context Table (NCT) of *Table1* and *Table2*

	Att1_a1	Att1_a2	Att2_b1	Att2_b2	Att2_b3	Att3_c1	Att3_c2	Att4_e1	Att5_f1	Att5_f2
Obj1	X		X			X				
Obj2		X		X			X			
Obj3	X				X		X			
Obj4								X	X	
Obj5								X		X

Figure 4.8: *Table3* Binary Context Table

Since the context tables defined for large software systems will contain a large number of intent attributes, some of which may have many values, the maintenance of such messy context tables is time consuming and error-prone. Therefore, in this thesis, first we construct a many-valued context table for each partially defined table, and then convert them into their corresponding binary context tables.

4.1.2 Definition of Attributes in Formal Context Tables

Formal context table in FCA consists of the abstract elements such as objects and their attributes that finally leads to obtain formal concepts. However, the goal of this thesis is to extract the components and their related artifacts which are relevant to the

component-based model for developing real-time reactive systems (TADL model). Therefore, a guideline specification is essential to help the designer in categorizing the concepts and to determine a convention for naming the attributes. Some primary specifications of TADL component model are described here to provide the necessary information.

4.1.2.1 Primary Specifications of TADL Component Model.

Some main elements of the TADL component model [49] are introduced as follows.

- **Component:** A component provides and requests services through public interfaces. Also, it defines attributes that define local value-type properties. Each component consists of many elements, one of which is the contract as shown in Figure 4.9 (L). The contract defines the behavior of the component.
- **Contract:** A contract defines the safety requirements that govern the interactions that occur at the interfaces of a component. Also, it defines time constraints that regulate the service requests and responses so that the reactions of a component respect any timeliness requirements. Predictability specification ensures that component reactions are precisely defined. Within the contract, a list of reactivity rules that define the request-response relationship between the services is given. The contents of the contract can be seen in Figure 4.9 (R).
- **Reactivity:** Reactivity has two services, a request service and a response service. As part of the contract, there are also time constraints and data constraints, which are used for safety purposes. Contract also includes the reactions to the request event, and the update element to define the post condition of the reactivity.

- **Service:** Service is a functionality provided or required by a component through public interfaces. A service can be provided by only one interface. A service can have multiple data parameters. A data parameter is a variable passed on to a component within a request for a service or passed on with a provided service. The type of the request service defined in an interface is *input* and the type of the response service is *output*. The type of the service defined in an internal interface is *internal*. A service may be of type input in one component and may be of type output in another component.

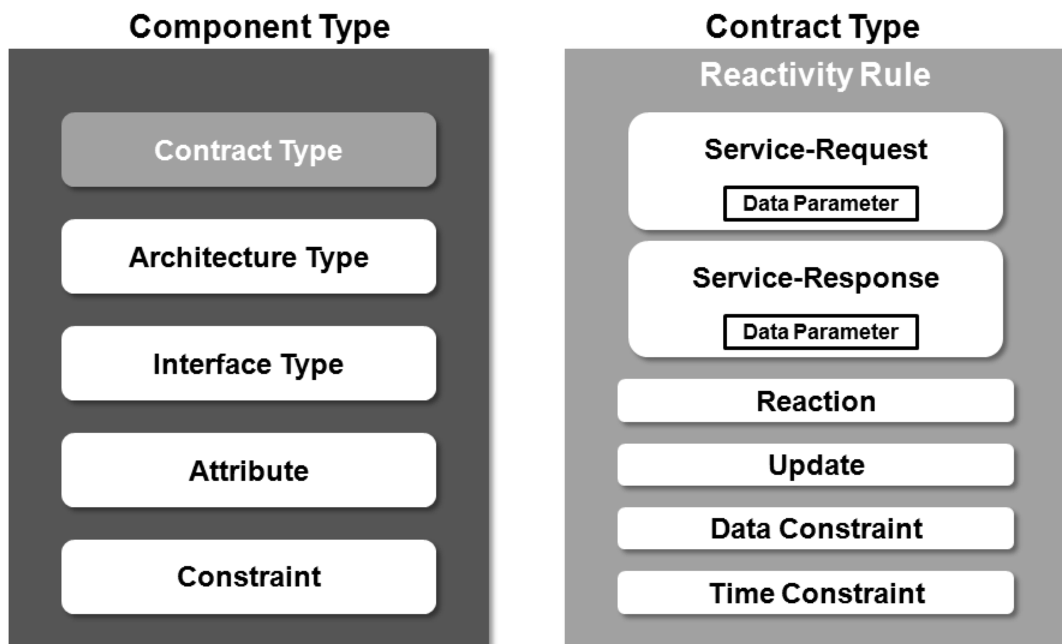

Figure 4.9 (L): Elements of ComponentType **Figure 4.9 (R):** Elements of ContractType

- **Safety contract:** A safety property is considered as part of the contract on a component type. It controls the way services are provided or requested. Each contract has a one-to-one relationship with a component type. A contract can have one or more safety properties, where a safety property defines an invariant specification over the component behavior. Each contract contains at least one

reactivity property, each of which expressing a relationship between a request and a response. A contract may have data and time constraints.

- **Security mechanism:** A security property deals with access control on the services and the data communicated with those services. The same security mechanism can be associated with several component types, and there is a one-to-many relationship between a security mechanism and component types. Role-Based Access Control (RBAC) is currently enforced as the only security mechanism, and has the following main elements: *user*, *group*, *role* and *privilege*. A user defines the identity on behalf of which the component will be executed. A group is a collection of users, and a user may belong to more than one group. A role defines the responsibilities that can be assigned to a user or a group in the system. A role aggregates a set of privileges, where each privilege defines a permission to perform a service or access to a data parameter. Security can be divided into two types: *Service security* and *Data security*. Service security considers the security of the services provided by the components, while Data security considers the security of the data transferred by the services. Service security ensures that:
  - every request received at a component interface is initialized by a user who has permission to request this service; if the user has no access permission, the request is ignored.
  - the user of a response sent from a component interface has permission to receive that response; if the user has no access permission, the response will not be sent.

There are the similar definitions for Data security. Figure 4.10, which is taken from [49], shows the trustworthy component model.

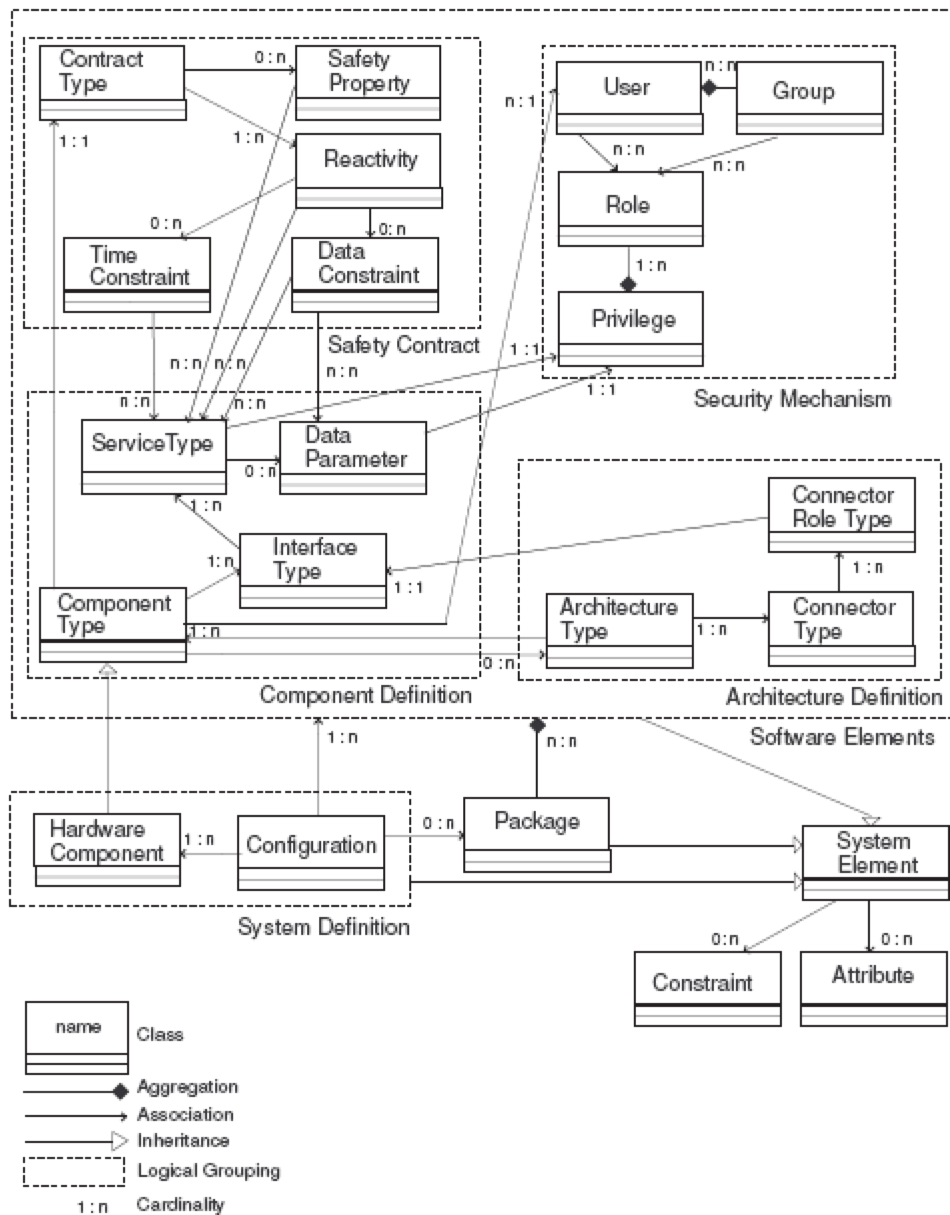

Figure 4.10: Trustworthy component model

4.1.2.2 Keywords to Specify Parameters. In a context table, rows denote objects and columns denote attributes. The incidence relation between an object and an attribute is shown by the corresponding crossing cell in the table. Some

keywords have been defined to address the specific parameters used in formal context table specification that are explained as follows:

- **Main Attribute:** All partially defined context tables should have only one attribute, called *main attribute*, which is defined as the common attribute of all objects. The main attribute will be the intent of the supremum node in the derived concept lattice.
- **Property of Attribute:** Every attribute in the context table may have one or more properties, defined as attributes. The properties of an attribute have the same incidence relation with the objects that the attribute has. In other words, whenever an object is in relation with an attribute, it is necessarily in relation with the attribute's properties. In the derived concept lattice, any attribute and its properties are gathered in the same node.
- **Time Constraint of Attribute:** Every attribute or its properties may have only one time constraint each. Time constraints of attributes or their properties are defined as attributes in the context table. Time constraint attribute has the same incidence relation with the objects that the attribute or its property has. In other words, whenever an object is in relation with an attribute, it is necessarily in relation with the attribute's time constraint. In derived concept lattice, any attribute and its time constraint are gathered in the same node.

4.1.2.3 Rules to Compose Partially Defined Context Tables.

Now, we introduce our methodology for categorizing the concepts in the formal context tables as follows:

- For each concept in the system, a partially defined context table is produced in FCA with the name of the concept.

- The name of the extent objects of the context table may be the name of the concept proceeded by a unique index.
- A new attribute is defined in the context table as the main attribute, with the name of the concept.
- For a pair of related functional requirements specified in the use-case analysis, an attribute is defined in the context table, called *functional requirements attribute*. The name of this attribute specifies the concept, the provided functional requirement and the requested functional requirement. Also, it is possible to define a functional requirement, so called internal, that is neither provided nor requested.
- Each functional requirements attribute may have one or more data constraints that must be satisfied to enable the functional requirements. For each data constraint, one attribute is defined in the context table, called *data constraint attribute*. Data constraints of any functional requirements attribute have the same incidence relation with the objects that the functional requirements attribute has. It is possible that one data constraint belongs to more than one functional requirements attribute.
- Each functional requirements attribute may have only one time constraint that defines the maximum allowed time between receiving a request and providing response. For the time constraint of the functional requirements attribute, the *time constraint attribute* is defined in the context table. Time constraint of any functional requirements attribute has the same incidence relation with the objects that the functional requirements attribute has.
- Each functional requirements attribute may trigger one or more actions. The actions are one or more functional requirements to be performed when they are

triggered by the functional requirements attribute. So, the *action attribute* is defined that contains all of the actions corresponding to the functional requirements attribute.

- Each functional requirements attribute may have one or more post conditions that lead to update the data parameters. So, for the updates related to the functional requirements attribute, the *update attributes* are defined in the context table.
- Each functional requirement may have one or more role privileges that are permitted to perform that functional requirement. So, for some provided or requested functional requirements, the *role privilege attributes* are defined in the context table. The name of the role privilege specifies the role name and the functional requirement name. Also, the role privilege attribute can specify the role name that is not permitted to perform the given functional requirement. Role privilege attribute of a given functional requirement has the same incidence relation with the objects that the corresponding functional requirements attributes have.
- If for any extent object there is no value for the many-valued attributes, a slash (/) is indicated in the corresponding crossing cell of the context table.

The different attribute types that might be defined in the context table and their corresponding name conventions are stated in Table 4.1. The + sign is used as a notation for string concatenation in the name convention. The '=' character is used as a separator inside the attribute name for implementation purposes. As an example, when defining a property of attribute that the attribute name is *CashDesk* and its property name is *CashDeskId*, the generated name according to this convention is *PropertyCashDesk = CashDeskId* (Table 4.1, row 1).

Table 4.1: Name Conventions for Attribute Types

	Attribute Type	Name Convention
1	Property of Attribute	‘Property’ + attribute name + ‘=’ + property name
2	Time Constraint of Attribute	‘TC’ + attribute name + ‘=’ + ‘T’ + time constraint value
3	Time Constraint of Property	‘TC’ + ‘Property’ + property name + ‘=’ + ‘T’ + time constraint value
4	Functional Requirements Attribute*	‘FR’ + concept name + ‘-‘ + provided functional requirement name + ‘_’ + ‘FR + requested functional requirement name
5	Data Constraint Attribute	‘DC’ + data constraint name
6	Time Constraint of Functional Requirements Attribute	‘TC’ + ‘FR + concept name + ‘-‘ + provided functional requirement name + ‘_’ + ‘FR + requested functional requirement name + ‘=’ + ‘T’ + time constraint value
7	Action Attribute**	‘Action’
8	Update Attribute	‘Update’ + ‘DC’ + data constraint name
9	Role Privilege	‘RolePrivilege’ + ‘-‘ + role name + ‘_’ + functional requirement name
10	Negative Role Privilege***	‘RolePrivilegeNot’ + ‘-‘ + role name + ‘_’ + functional requirement name

* In the name convention of Functional Requirements Attribute, if the provided or requested functional requirement is of type internal, the relevant keyword ‘FR is replaced by ‘IFR’.
** In the crossing cells corresponding to an Action Attribute, the action names separated by comas are registered.
*** Negative Role Privilege specifies the role name that is not permitted to perform the functional requirement.
TC: Time Constraint; DC: Data constraint; T: Time

Example 1

Suppose a system contains a concept named *Concept1* for which a valued context table is defined as depicted in Figure 4.11. Then, it is converted to *Concept1* binary context table (Figure 4.12). According to the rules to compose partially defined context tables presented in Section 4.1.2.3, some attributes are defined in *Concept1* context table as follows:

First, the main attribute *Concept1* is defined. Assume that *Concept1* provides the functional requirements *Req1* and *Req3*, and requests the functional requirements *Req2*, and has the internal functional requirement *Req4*. Moreover, we consider that whenever *Concep1* provides *Req1*, it requests *Req2*, and whenever *Concept1* provides *Req3*, it requests *Req4*. So, the functional requirements attributes *FRConcept1-Req1_FRReq2* and *FRConcept1-Req3_IFRReq4* are defined in the *Concept1* context table (see the name

conventions for attribute types Table 4.1, row 4). The data constraint attribute *DCMode* (Table 4.1, row 5) must be satisfied to enable the functional requirements attributes. *DCMode* has two values *DataC1* and *DataC2*. The time constraint attribute *TCFRConcept1-Req3_IFRReq4=T5.0S* (Table 4.1, row 6) defines the maximum allowed time to provide the functional requirement *Req3*. The Action attribute (Table 4.1, row7) is defined for the functional requirements attribute *FRConcept1-Req3_IFRReq4* that triggers the functional requirement *Req5*. The update attribute *UpdateDCMode* (Table 4.1, row8) is defined for the functional requirements attribute *FRConcept1-Req3_IFRReq4*, and updates the value of data constraint attribute *DCMode* from *DataC2* to *DataC3*.

	Concept1	DCMode	FRConcept1-Req1_FRReq2	FRConcept1-Req3_IFRReq4	TCFRConcept1-Req3_IFRReq4=T5.0S	UpdateDCMode	Action
Object1		DataC1		/	/	/	/
Object2		DataC2	/			DataC3	Req5

Figure 4.11: *Concept1* Valued Context Table (VCT)

	Concept1_	DCMode_DataC1	DCMode_DataC2	FRConcept1-Req1_FRReq2_	FRConcept1-Req3_IFRReq4_	TCFRConcept1-Req3_IFRReq4=T5.0S_	UpdateDCMode_DataC3	Action_Req5
Object1	X	X		X				
Object2	X		X		X	X	X	X

Figure 4.12: *Concept1* Binary Context Table (BCT)

4.2 Construction of Unified Formal Context Table

The concepts and the functional requirement properties defined in many-valued context tables are converted to binary context tables. Then, they must be manually combined and pruned in order to construct a unified formal context table. Some redundant attributes should be removed, some properties of attributes should be converted to attributes or some attributes should be unified and merged together. To achieve this goal, a number of rules have been defined to guide the designer to do this process accurately and precisely.

To obtain complete and assured results, it is recommended that the integration process is done gradually and in several steps accompanied by pruning operations. Developing an integration process that complies with this method would be much more reliable and efficient than defining a large combined context table which may contain deficiencies and/or redundancies. In the beginning, each many-valued context table is transformed into a binary context table, in other words all VCTs are converted to BCTs, and then the integration of the BCTs is manipulated one after another. The resulting context table obtained from combining two or more partially defined tables is merged subsequently by the other BCTs or merged context tables. This process continues progressively until the entire and unified formal context table is constructed.

4.2.1 Rules to Integrate Partially Defined Context Tables

The unified formal context table is derived now by combining the partially defined context tables. It is better to look for some priorities to identify the group of BCTs to be combined. It is recommended to start from the context tables containing the concepts

which are related and have common functional requirements. This facilitates the integration process by determining the common attributes to be merged. To integrate the partially defined context tables, the following steps should be fulfilled:

- A nested context table (NCT) is defined to combine the selected BCTs by assigning their table names for different levels of combination.
- The defined NCT is converted to the corresponding BCT.
- A main attribute with an arbitrary name is added to the combined BCT.

Table 4.2: Conditions and Integration Rules to merge partial context tables

	***Attribute1* Conditions**	***Attribute2* Conditions**	**Rules**
1	(type = attribute)	(is duplicate of *Attribute1*)	Rule 1
2	(type = attribute) & (has properties or time constraint)	(is duplicate of *Attribute1*) & (does not have the same properties or time constraint as *Attribute1*)	Rule 1, Rule 2
3	(type = attribute) & (has properties or time constraint)	(is duplicate of *Attribute1*) & (has at least one property or time constraint of *Attribute1*)	Rule 1, Rule 2, Rule 3
4	(type = attribute)	(is duplicate of *Attribute1*) & (has at least one more property or time constraint than *Attribute1*)	Rule 1, Rule 4
5	(*Attribute1* is to be eliminated)		Rule 5
6	(type = property of *Attribute3*)	(type = property of *Attribute4*) & (has the same name as *Attribute1*) & (*Attribute3* != *Attribute4*)	Rule 6

- The duplicate attributes are recognized to be merged or eliminated from the combined context table. Otherwise it will lead to redundancy and/or ambiguity. Table 4.2 specifies different conditions and their relevant integration rules to be applied in order to merge and prune the partially defined context tables. The integration rules and the corresponding actions are introduced in Table 4.3. The integration rules can be generalized for more than two duplicate attributes in the combined context tables.

Table 4.3: Integration Rules and Actions to merge partial context tables

	Rules	Actions
1	Rule 1	One of the duplicate attributes is eliminated. The extent objects in the relation with the removed attribute are denoted in the crossing cells of the remaining attribute.
2	Rule 2	The extent objects in the relation with the removed attribute are denoted in the crossing cells of the properties and the time constraint of the remaining attribute.
3	Rule 3	The properties and the time constraint of the removed attribute which are the same as the remaining attribute are also eliminated.
4	Rule 4	The properties and the time constraint of the removed attribute are maintained as the properties and time constraint of the remaining attribute. The extent objects in relation with the remaining attribute are denoted in the crossing cells of the maintained properties and time constraint.
5	Rule 5	The specified attribute to be eliminated along with its probable properties and time constraint are removed from the original many-valued context table. If any extent objects remain with no bond to any intent attribute, they are eliminated too.
6	Rule 6*	(1) The duplicate properties of different attributes cannot be merged and maintained as properties. One of the duplicate properties is eliminated based on the designer's decision. Or, (2) they are merged and one of them is maintained as a new attribute. The extent objects in relation with the removed property are denoted in the crossing cells of the new attribute. Or, (3) they are maintained as two different properties with different names. The names of the duplicate properties are modified in the original many-valued context tables. If the duplicate properties have time constraints, the names of their time constraints are also modified accordingly.

* Rule 6 has three extensions and based on the designer's decision, one of them is conformed.

Example 2

Suppose the mentioned system in Example 1 has another concept named *Concept2*. The valued and binary context tables of *Concept2* are shown in Figure 4.13 and Figure 4.14. According to the rules to compose the partially defined context tables presented in Section 4.1.2.3, some attributes are defined in *Concept2* context table as follows:

First, the main attribute *Concept2* is defined. Assume that *Concept2* provides the functional requirements *Req2* and *Req5*, and requests the functional requirements *Req3* and *Req6*. Also, it is considered that whenever *Concep2* provides *Req2*, it requests *Req3*, and whenever *Concept2* provides *Req5*, it requests *Req6*. So, the functional requirements attributes *FRConcept2-Req2_FRReq3* and *FRConcept2-Req5_FRReq6* are defined in the *Concept2* context table (see the name convention in Table 4.1, row 4). The data constraint attribute *DCMode* (see the name convention in Table 4.1, row 5) must be satisfied to enable the functional requirements attributes. *DCMode* has two values

DataC1 and *DataC3*. The update attribute *UpdateDCMode* (see the name convention in Table 4.1, row8) is defined for the functional requirements attributes, and updates the value of data constraint attribute *DCMode*.

Lattice Miner

File Edit Lattice Rules Context Window About

Context : Concept2

Context : Concept2

	Concept2	DCMode	FRConcept2-Req2_FRReq3	FRConcept2-Req5_FRReq6	UpdateDCMode
Object3		DataC1		/	DataC2
Object4		DataC3	/		DataC4

Figure 4.13: *Concept2* Valued Context Table (VCT)

Lattice Miner

File Edit Lattice Rules Context Window About

Context : Concept2

Context : Concept2

	Concept2_	DCMode_DataC1	DCMode_DataC3	FRConcept2-Req2_FRReq3_	FRConcept2-Req5_FRReq6_	UpdateDCMode_DataC2	UpdateDCMode_DataC4
Object3	X	X		X		X	
Object4	X		X		X		X

Figure 4.14: *Concept2* Binary Context Table (BCT)

According to the rules for integrating partially defined context tables presented in Section 4.2.1, two partially defined context tables *Concept1* and *Concept2* are merged and pruned as follows:

First, the nested context table *MergedConcepts* is defined to combine the partially defined context tables *Concept1* and *Concept2* (Figure 4.15). Then, the defined NCT is converted to the corresponding BCT which is shown in Figure 4.16.The main attribute *MergedConcepts* is added to the combined BCT. The duplicate attribute *DCMode_DataC1* in the merged context table (Figure 4.16) has to be merged according to Rule 1 of the

conditions and integration Rules in Table 4.2. Then, according to the integration rules and actions in Table 4.3, one of the duplicate attributes is eliminated. The extent objects in relation with the removed attribute are denoted in the crossing cells of the remaining attribute. Also, for the main attribute *MergedConcepts* a property attribute named *PropertyMergedConcepts=MergedConceptsId* is defined (Table 4.1, row 1). On the other hand, the role privilege attributes *RolePrivilege-Role1_Req1* and *RolePrivilegeNot-Role1_Req3* (Table 4.1, row 9 and row 10) are defined in the merged context table to specify that *Role1* is permitted to perform the functional requirement *Req1* and is not permitted to perform the functional requirement *Req3*. The resulting context table is depicted in Figure 4.17.

Figure 4.15: *MergedConcepts* Nested Context Table (NCT)

Figure 4.16: *MergedConcepts* Binary Context Table (BCT)

Lattice Miner

File Edit Lattice Rules Context Window About

Context : MergedConcepts

Context : MergedConcepts

	MergedConcepts	PropertyMergedConcepts=MergedConceptsId	DCMode_DataC1	DCMode_DataC2	DCMode_DataC3	Concept1_	FRConcept...	UpdateDC...	RolePrivilege-Role1_Req1
Object1	X	X	X			X	X		X
Object2	X	X		X		X		X	
Object3	X	X	X						
Object4	X	X			X				

Figure 4.17: Merged and pruned *MergedConcepts* context table

4.3 Concept Lattice Derivation

So far, a unified formal context table is constructed that contains the required formal concepts, captured from system specifications. In this step, the concept lattice corresponding to the derived formal concept hierarchy is obtained. This process may be systematically done by FCA software tools. They have the facility to draw the concept lattice diagram from the given formal context table.

Concept lattices are mathematical structures supported by a rich and well established formalism, namely, Formal Concept Analysis [28]. Wille [84] proposed to consider each element in the lattice as a concept and the corresponding graph (Hasse diagram) as the generalization/specialization relationship between concepts. From this perspective, the lattice represents a concept hierarchy. Each concept is a pair composed of an extension representing a subset of instances and an intension representing the common features for this set of instances [30]. The concepts of lattice are partially ordered in a “subconcept-superconcept” hierarchy. The main attribute of the unified formal context table is presented as the intent of the supremum node in the derived concept lattice. As an example, we refer to the concept lattice of merged and pruned *MergedConcepts* context table which is depicted in Figure 4.18.

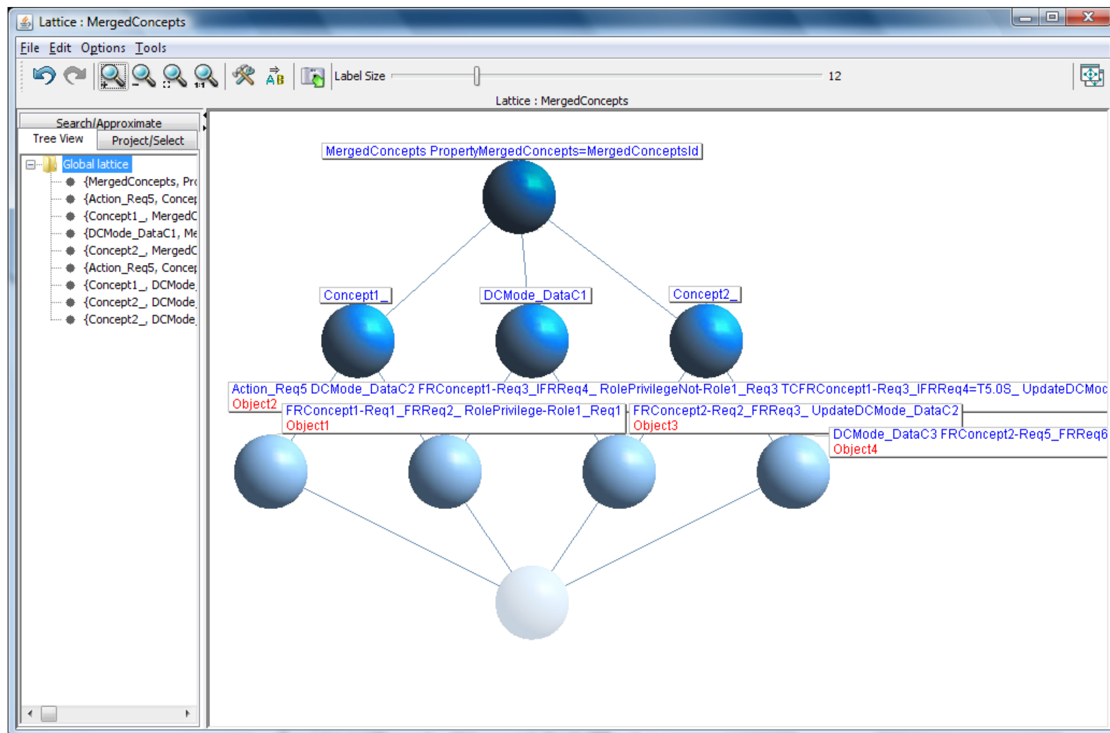

Figure 4.18: *MergedConcepts* Concept Lattice

Besides, FCA software tools, like Lattice Miner, provides the facility to extract the acquired concept lattice as an XML file with a special structural format that is editable by any XML editor. On the other hand, the implication rules are derived from the resulting concept lattice and are captured in another XML file. The XML file of concept lattice is merged with the XML file of the implication rules. The merged XML-format file can be saved to be transformed into the OWL-format ontology file. We discuss this at the next step of our methodology presented in Chapter 5.

4.4 Advantages of the presented method

When the partially defined formal context table is constructed and before starting the integration process, the OWL ontology corresponding to the given BCT can be obtained

by the model transformation process presented in this thesis (Chapter 5). Besides, in the intermediate steps of the integration process and before obtaining the final formal context table, the temporarily composed context table can be transformed into the corresponding OWL ontology. This is one of the privileges of the introduced methodology to construct formal context tables. Adhering to this technique provides the opportunity to verify the partially composed ontology at any stage of integration process. This affords the facility to find possible errors and inconsistencies before producing the final context table.

Another advantage of this approach is that it is possible to go back to the previous steps at any stage of the process and make modifications to retrieve the desired outputs. If the designer decides to modify the context tables, it is better to make changes in the original context tables (VCTs) and not in the merged tables. Otherwise, the traceability aspect, which is the ability to link and verify design artifacts belonging to every step in a process chain, will be violated. As an example, eliminating an attribute in the combined context table, while it is conserved in the original tables, will lead to inconsistency and difficulties in the future maintenance of the tables.

Chapter 5

Transformation from Context Tables to Concepts Formation

The methodology for defining formal context tables has been presented in Chapter 4. In this Chapter, we analyze the resulting concept lattice obtained from formal context tables in FCA to realize the concepts and the concept hierarchy. Capturing this knowledge is necessary to find a model of shared understanding of the domain, determine the ontological classes, and build the one to one corresponding elements in ontology. One of the contributions of this thesis is to introduce a general solution to achieve this goal. A set of rules has been developed for transforming context tables to ontology. The proposed rules are implemented automatically by a model transformation approach that is presented in Chapter 6.

This Chapter is structured as follows. In Section 5.1, the class hierarchy in concept lattice is discussed and the various relations between nodes are explained. The same analysis is done on ontological class hierarchy in Section 5.2, and the various relations and the property restrictions in ontologies are discussed. Then, the transformation rules

from lattice hierarchy into the ontological class hierarchy are provided in Section 5.3. Finally, in Section 5.4, the advantages of the proposed methodology are described.

5.1 Class Hierarchy in Concept Lattice

Formal Concept Analysis (FCA) provides a natural theoretical framework for class hierarchy design and maintenance. The hierarchies produced within this framework have a well-defined semantics that remains independent from the concrete algorithms used. In addition, the produced hierarchies tend to conform to general quality criteria such as simplicity, reusability, comprehensibility, extensibility and maintainability [31].

The concept lattice that is computed from the formal context table is depicted as a line diagram consisting of the nodes representing the formal concepts. Each node contains the objects and the attributes belonging to the formal concept corresponding to that node. The nodes are ordered in a '*subconcept-superconcept*' hierarchy [26]. The formal concept of node A is *subconcept* of node B, if the extent of node A is the subset of the extent of node B. Equivalently stated, the formal concept of node A is *subconcept* of node B if the intent of node B is the subset of the intent of node A. Thus, while traversing the concept lattice from bottom to up, the number of objects in the extent of the upper nodes increases while the number of the attributes in the intent decreases.

The two important operations in concept lattice are *meet* (*infimum*) and *join* (*supremum*) [26]. The infimum operation is applied on two or more concepts and the result is another concept in the same concept lattice. Its extension is obtained by intersecting the extensions of the given concepts, and its intension is obtained by taking the union of the intensions of the given concepts. The supremum operation is applied on two or more concepts and the result is another concept in the same concept lattice. Its

extension is obtained by taking the union of the extensions of the given concepts, and its intension is obtained by intersecting the intensions of the given concepts. As an example, we can refer to the *Planets* concept lattice depicted in Figure 5.1, which is derived from the *Planets* binary context table presented in Chapter 4, Figure 4.2. It is considered that, the extension of the supremum concept is the union of the extensions of its subconcepts. Similarly, the intension of the infimum concept is the union of the intensions of its superconcepts.

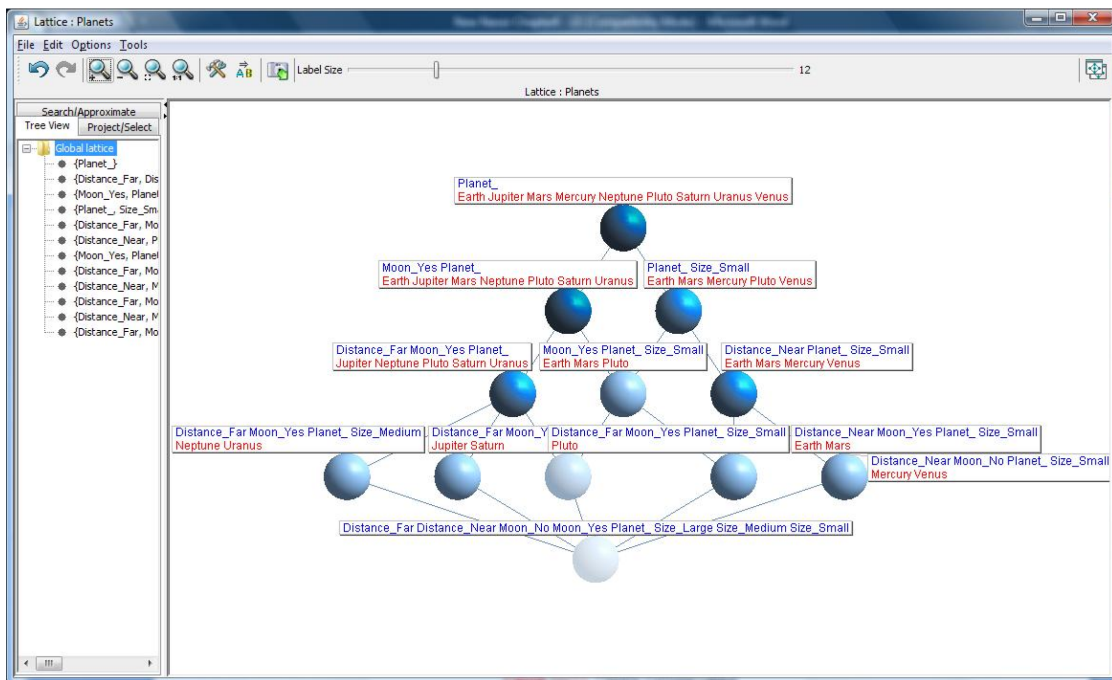

Figure 5.1: *Planets* Complete Labeling Concept Lattice

If every node in the concept lattice diagram is marked by the corresponding extents and intents, this would lead to a great cluttering of the picture. In order to overcome this problem the *reduced labeling* technique [88, 54] is used. In this method, each object and each attribute is entered only once in the diagram. A concept is labeled with attribute a, if it is the largest concept having a in its intent. This means that all the concepts below the

concept labeled with the attribute a contains a in their intent set of attributes. Similarly, a concept is labeled with an object o if it is the smallest concept having o in its extent. The reduced labeling does not lead to a loss of information, because the intent and extent of any concept can be read off from the diagram. Thus, the extent of any node in the line diagram contains the collection of all objects belonging to the nodes below and its intent contains the collection of all attributes belonging to the nodes above. As an example, the reduced labeling concept lattice of the *Planets* context table is shown in Figure 5.2. We notice that the concept labeled with attribute *Size_Small* is the largest concept having the given attribute in its intent. Also, the concept labeled with the object *Pluto* is the smallest concept having the given object in its extent.

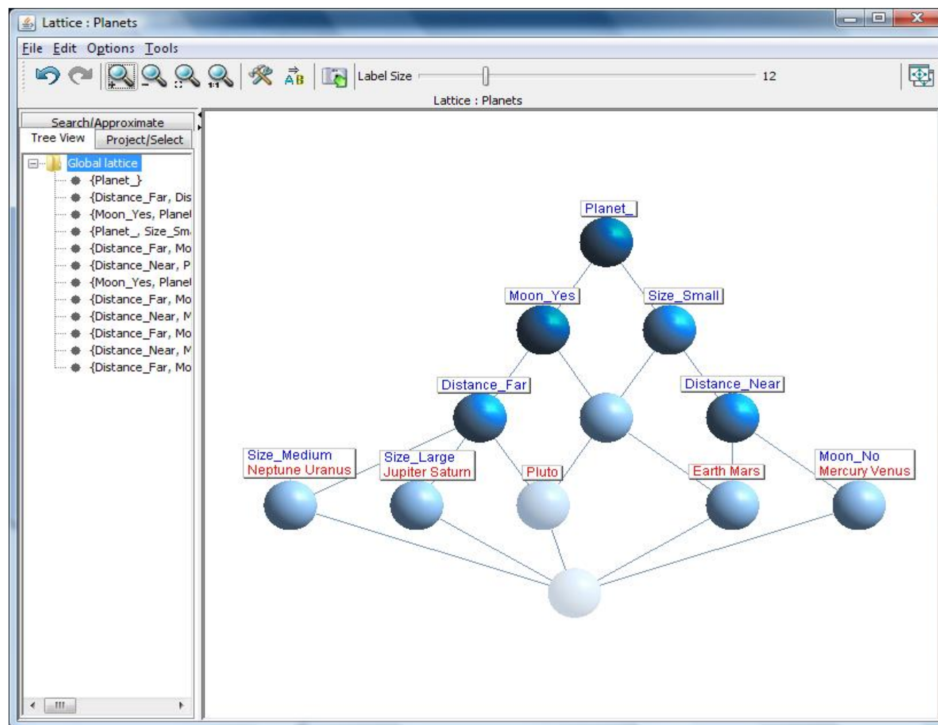

Figure 5.2: *Planets* Reduced Labeling Concept Lattice

Implication [26, 23] is another relation among the attributes of a context that can be used in conceptual knowledge and conceptual learning. An attribute implication of a

context is a pair of subsets of attributes, say X, Y, where each object having all attributes of X has also all attributes of Y, and this is depicted as $X \rightarrow Y$. The set X is called the *premise,* and Y is its *conclusion*. Informally, implications between attributes can be found along upward paths in the lattice. Actually, we are not interested in all valid implications of the context, but only in a certain minimal basis of implications from which all other valid implications can be deducted [69].

5.2 Class Hierarchy in Ontology

Like Formal Concept Analysis, domain ontology has the goal of modeling concepts; however it has its own specifications and purposes. Ontology deals with modeling shared understanding of the domain and capturing conceptual knowledge accepted by domain experts. Moreover, objects are not necessary in defining ontology and only the intensional aspect is considered by ontologies [23]. The concepts in ontology have the hierarchical order and the properties of objects are chosen as the criteria to classify the objects. Ontology may be visualized as an abstract graph with nodes representing the objects and labeled arcs representing the relations [62].

Ontology Web Language (OWL) [67, 32] facilitates describing the concepts in a domain and also the relationships holding between concepts. It provides a set of operators like intersection, union and negation for concept classification and analysis. Since it is based on logical models, OWL can benefit from the use of the reasoner which checks the consistency of all concepts and definitions in the ontology and also recognizes which concepts fit under which definitions so that it can maintain the class hierarchy correctly.

OWL ontology consists of *individuals*, *classes*, and *properties*. Individuals represent the objects of the domain and are the instances of the classes. OWL classes are interpreted as sets that contain individuals. Classes are built up of descriptions that specify the conditions that must be satisfied by an individual to be a member of the class. Classes are a concrete representation of the concepts. Classes may be organized into a subclass-superclass hierarchy, which is also known as taxonomy. Subclasses specialize ('are subsumed by') their superclasses. The empty ontology contains one class called *Thing*. The class *Thing* is the class that represents the set containing all individuals. So, all classes are subclasses of *Thing* [39].

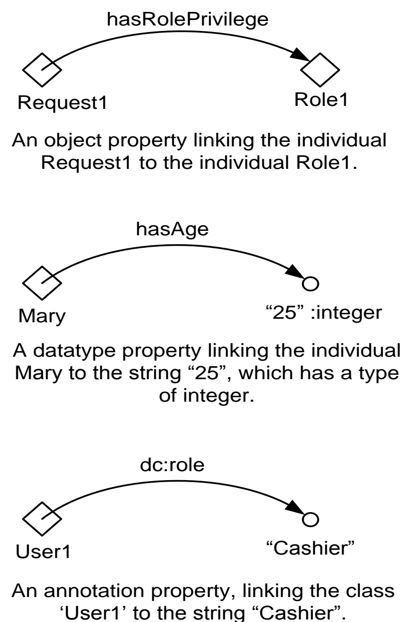

Figure 5.3: Various types of OWL properties

OWL properties represent relationships. The two main types of properties in OWL are *object properties* and *datatype properties*. Object properties are binary relations between two individuals. Datatype properties describe relationships between individuals and data values. OWL also has a third type of property, called *annotation property*.

Annotation property can be used to add information (metadata) to classes, individuals and object/datatype properties. Note that it is also possible to create subproperties of object/datatype properties. However, it is not possible to mix and match the object properties and the datatype properties with regards to subproperties. For example, it is not possible to create an object property that is the subproperty of a datatype property and vice-versa. The various types of OWL properties are depicted in Figure 5.3.

Object properties may have various characteristics as follows [39]:

- **Functional Property:** For the given individual, there can be at most one related individual via the property.
- **Inverse Property:** Property can have an inverse. For example, the inverse of property hasOwner is the property IsOwnedBy.
- **Inverse Functional Property:** For the given property, there can be at most one related individual via the inverse property. That means the inverse property is also functional.
- **Transitive Property:** If property P is transitive, and P relates individual a to individual b, and also individual b to individual c, then it can be inferred that individual a is related to individual c via property P. If a property is transitive, then its inverse property should also be transitive. The transitive property cannot be functional.
- **Symmetric Property:** If property P is symmetric, and P relates individual a to individual b, then it can be inferred that individual b is related to individual a via property P.
- **Antisymmetric Property:** If property P is antisymmetric, and P relates individual a to individual b, then individual b cannot be related to individual a via property P.
- **Reflexive Property:** If property P is reflexive, it relates individual a to itself.

- **Irreflexive Property:** If property P is reflexive, it relates individual a to individual b where individual a and individual b are not the same.

Most of the mentioned characteristics of the object properties cannot be applied for the datatype properties. For example, datatype properties are not allowed to be transitive, symmetric or have inverse properties.

Properties may have a *domain* and a *range* specified. Properties link individuals from the domain to individuals from the range. In OWL, domains and ranges are not constraints to be checked. However, they are considered as axioms in reasoning.

In OWL, we can define *restrictions*. The restriction property describes an anonymous class of an individual, based on the relationships that members of the class participate in. OWL restrictions fall into three main categories [39]:

- **Quantifier Restriction:** Quantifier restriction can be further categorized into *existential* restriction and *universal* restriction. Existential restriction, also known as 'someValuesFrom' restriction, describes the class of individuals that participate in at least one relationship between a specified property and individuals that are members of a specified class. Existential restriction may be denoted by the existential quantifier $\exists$. Universal restriction, also known as 'allValuesFrom' restriction, describes the class of individuals that for a given property only have relationships between this property and individuals that are members of a specified class. Universal restriction may be denoted by the universal quantifier $\forall$.
- **Cardinality Restriction:** Cardinality restriction describes the class of individuals that have at least, at most or exactly a specified number of relationships with other individuals. For a given property P, a Minimum Cardinality Restriction specifies the minimum number of P relationships that an individual must

participate in. A Maximum Cardinality Restriction specifies the maximum number of *P* relationships that an individual can participate in. A Cardinality Restriction specifies the exact number of *P* relationships that an individual must participate in.

- **hasValue Restriction:** The restriction 'hasValue' describes the class of individuals that have at least one relationship between a specified property and a specific individual. The restriction 'hasValue' is denoted by the symbol ∋.

We consider the sample *MergedConcepts* concept lattice presented in Chapter 4. The OWL ontology obtained after the model transformation process contains the above described property restrictions. As an example of the existential qualifier restriction, we can refer to the ontological class *MergedConcepts* that participates in at least one relationship between the property *hasMergedConceptsProperty* and individuals that are members of the class *MergedConceptsId*. The relevant OWL source code is illustrated as follows:

```
 <owl:Class rdf:ID="MergedConcepts">
    <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >MergedConcepts
    </rdfs:label>
    <rdfs:subClassOf rdf:resource="#MClass-MergedConcepts-PropertyMergedConceptsId"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:someValuesFrom rdf:resource="#MergedConceptsId"/>
        <owl:onProperty>
          <rdf:Property rdf:ID="hasMergedConceptsProperty"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

As an example of the universal qualifier restriction, we can refer to the ontological class *Concept1* that for the property *hasProvidedFR* has only relationships with the individuals that are members of either the class *Req1* or the class *Req3*. The relevant OWL source code is illustrated as follows:

```
<owl:Class rdf:ID="Concept1">
    <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >Concept1
    </rdfs:label>
    <rdfs:subClassOf rdf:resource="#MClass-MergedConcepts-PropertyMergedConceptsId"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty>
                <rdf:Property rdf:ID="hasProvidedFR"/>
            </owl:onProperty>
            <owl:allValuesFrom>
                <owl:Class>
                    <owl:unionOf rdf:parseType="Collection">
                        <owl:Class rdf:about="#Req1"/>
                      <owl:Class rdf:about="#Req3"/>
                    </owl:unionOf>
                </owl:Class>
            </owl:allValuesFrom>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

As the example of the restriction ‘hasValue’, we can refer to the ontological class *FRConcept1-Req1_FRReq2* that has at least one relationship between the property *hasMode* and the specific individual *DataC1*. The relevant OWL source code is illustrated as follows:

```
<owl:Class rdf:ID="FRConcept1-Req1_FRReq2">
    <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >FRConcept1-Req1_FRReq2
    </rdfs:label>
    <rdfs:subClassOf rdf:resource="#Concept1"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:hasValue rdf:resource="#DataC1"/>
            <owl:onProperty>
                <rdf:Property rdf:ID="hasMode"/>
            </owl:onProperty>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

Finally, we can conclude that the object relationships in OWL ontology are either the subclass-superclass relation that is defined by ‘subClassOf’ keyword, or the arbitrary object properties that are defined to link the objects.

5.3 Transformation Rules to Build Ontology from FCA

This section introduces the transformation rules for the automatic generation of OWL ontology based on the analysis of the concept hierarchy derived from FCA. Concept lattice as a mathematical framework is the input model of this transformation process, so that its elements and their perceived relations are mapped into their relevant ontological elements in a one to one relationship. Basically, the formal concepts in FCA are going to be transformed into the concepts in ontology. Also, the relations between the formal concepts in FCA are going to be transformed into the relations among the concepts in ontology. Note that, the proposed transformation rules are defined based on the concept hierarchy that is retrieved from *reduced labeling* technique on the concept lattice. The principal schema of one to one corresponding relations among the elements of FCA and OWL ontology are shown in Figure 5.4.

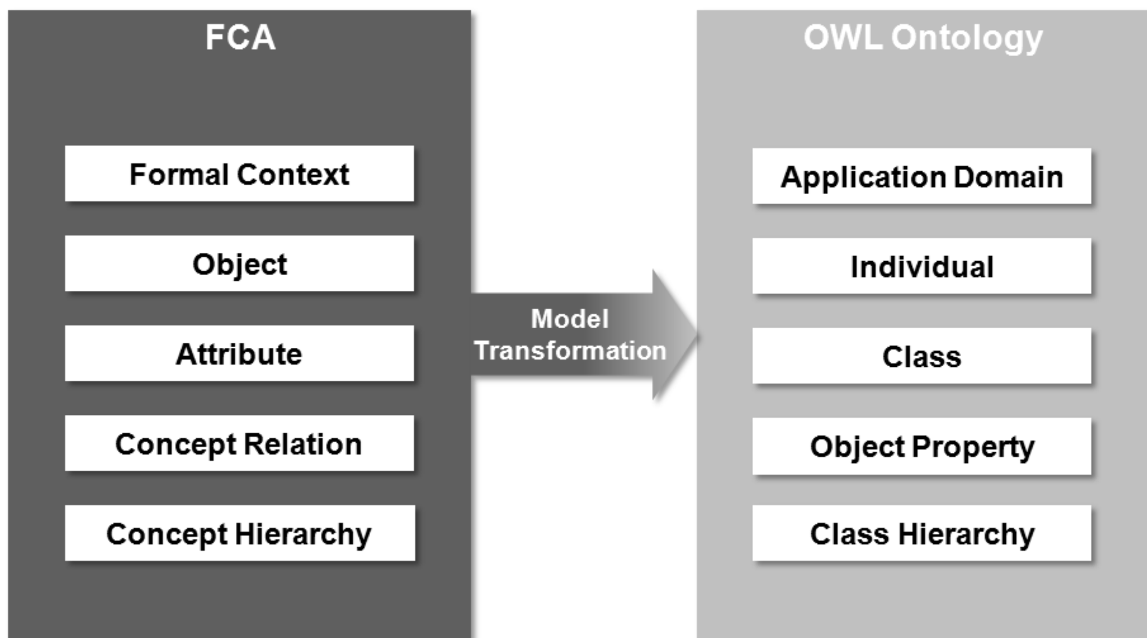

Figure 5.4: From FCA to OWL Ontology

The provided transformation rules are classified into six principal rules. These are, Ontology Overview Definition Rule, Class Definition Rules, Class Hierarchy Rules, Individual Definition Rules, Object Property Definition Rules, and Implication Rules. By adhering to the consecutive proposed rules, the ontological class hierarchy is composed to build the final OWL ontology.

5.3.1 Ontology Overview Definition Rule

Ontology overview consists of the base URI location, default namespace, and ontology language specified as the header of the ontology file. They are identified in the form of prefixes to abbreviate the URIs of the namespaces used in ontology.

According to the ontology overview definition rule, the name of the ontology and the ontology overview parameters are defined based on the name of the derived concept lattice in FCA. A sample ontology overview is presented as follows:

Base URI (Location): xml:base="http://example.org/CoCoME">
Default Namespace: xmlns="http://example.org/CoCoME#"
Owl: xmlns:owl="http://www.w3.org/2002/07/owl#"
rdf: xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
Rdfs:xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
Xsd: xmlns:xsd="http://www.w3.org/2001/XMLSchema#"

5.3.2 Class Definition Rules

Generally, formal concepts in FCA are transformed into the classes in ontology. However, more details are explained to clarify the particular significances and declare various class types as follows:

- **Root Class:** Supremum node in the lattice diagram is the top element of the concept hierarchy. The ontological class corresponding to the supremum node is called *root* class which is the parent of all other classes in the hierarchy. The name of the root class is the name of the attribute of the supremum node in the lattice. The root class is considered as the subclass of the class *Thing* in ontology.
- **Anonymous Node:** The nodes in the lattice diagram without any intent or extent are called *anonymous*. In the transformation process, no class is defined in ontology corresponding to anonymous nodes.
- **Class/Multi-attribute Class:** For each node with intent, a class is defined in ontology with the name of its attribute. If the intent of node contains more than one attribute, the name of the defined class is composed of the attribute names of the intent. Such a class is called *Multi-attribute class*, or briefly *MClass*.
- **Object Class:** For each node without intent that has extent, a class is defined in ontology with the name of its objects. Such a class is called *object class*, or briefly *ObjClass*.
- **Object Classes Class:** *ObjectClasses* is defined as a subclass of the root class. All object classes in the ontology are categorized as the subclasses of this class.
- **FRequirements Class:** *FRequirements* is defined as a subclass of the root class. All functional requirements captured from the functional requirements attributes in concept lattice are categorized as the subclasses of this class.
- **Trustworthy Classes:** Since the implementation of the trustworthiness is one of the contributions of this thesis, trustworthy classes are defined in OWL ontology in order to conform to the safety contracts and the security mechanisms specified in TADL component models.

- **Time Constraint Class:** To accomplish the safety requirement, the class *TimeConstraint* is defined as a subclass of the root class. Then, an equivalent class is defined for the class *TimeConstraint* that contains all time constraint values specified in the time constraint attributes of the concept lattice.
- **Data Constraints Class:** To accomplish the safety requirements, the class *DataConstraints* is defined as the subclass of the root class. For each data constraint attribute of the concept lattice, a class is defined as the subclass of the class *DataConstraints*. Then, an equivalent class is defined for each data constraint class that contains all possible values it can hold.
- **Roles Class:** To accomplish the security purposes, the class *Roles* is defined as the subclass of the root class. Then, for each role specified in the concept lattice, a role class is defined as the subclass of the class *Roles*.

5.3.3 Class Hierarchy Definition Rules

Concept hierarchy in FCA is transformed into the class hierarchy in ontology. Since for each non-anonymous node in concept lattice a corresponding class is defined in the ontology, the immediate superconcept of each node in the concept hierarchy is considered as the superclass of the ontological class corresponding to that node. So the relation ‘subClassOf’ is defined in ontology between the node class and its superclass. The particular cases are explained accordingly:

- Supremum node of the concept lattice is the top element of the concept hierarchy, and does not have any superconcept. Therefore, the root class which is corresponding to the supremum node is defined as the subclass of the class *Thing* in ontology.

- If the immediate superconcept of a given node is anonymous, the root class is defined as the superclass of the ontological class corresponding to the given node. The reason is that, for the anonymous nodes there is no corresponding class in the ontology, so it cannot be considered as the superclass.
- Each object class is defined as the subclass of the class *ObjectClasses*.
- The functional requirements captured from the functional requirements attributes in concept lattice may have three different types: *provided*, *requested*, and *internal*. Accordingly, three classes called *ProvidedFRs*, *RequestedFRs*, and *InternalFRs* are defined as the subclasses of the class *FRequirements*. So the functional requirements in the ontology are categorized to be the subclass of one the above subclasses, according to their class type.
- If a given node is not corresponding to an object class and has more than one superconcept in the hierarchy, then the corresponding class will have more than one superclass in the ontology. First, the root class is defined as one of the superclasses. Second, the intersection of the classes corresponding to the superconcepts of the given node is defined as another superclass. Subsequently, if any of the superconcepts of the given node is anonymous, the superconcepts of the anonymous node are captured from the hierarchy and added to the set of superclasses.
- If a given node has some property attribute in its intent, the corresponding class is a *MClass* that is already defined in the ontology. For the main attribute of that node, a main-attribute class with the name of the main attribute is defined as the subclass of that *MClass*. Afterwards, for each property attribute in the intent of the given node, a new property class with the name of the property attribute is defined as the subclass of the *MClass*.

5.3.4 Individual Definition Rules

Basically, the objects of formal concepts are transformed into the individuals in ontology. The particular details are described accordingly:

- If the objects belong to the extent of a given node, which has both intent and extent, the objects are defined as the individuals or instances of the ontological class corresponding to that node.
- If the objects belong to the extent of a given node that is corresponding to an object class, the objects are defined as the individuals or instances of that object class.

5.3.5 Object Property Definition Rules

Generally, various types of relations among the formal concepts are transformed into the relevant object properties in ontology. More details are specified as follows:

- **has Constraint:** One of the object properties defined in the ontology is the property *hasConstraint*. The property *hasConstraint* is defined to manipulate the trustworthy requirements in ontology. The object classes do not have constraints because they do not have any attribute to be restricted by any constraint. The subproperties of the property *hasConstraint* are explained as follows:
    - **has Time Constraint:** The property *hasTimeConstraint* is defined as the subproperty of the property *hasConstraint*, with the range class *TimeConstraint*. It relates an attribute class, a property class, or a functional requirement with its time constraint and specifies the relevant time value.

- **has Data Constraint:** The property *hasDataConstraint* is defined as the subproperty of the property *hasConstraint*, with the range class *DataConstraints*. For each data constraint class which is defined as the subclass of the class *DataConstraints*, a corresponding data constraint property is defined as the subproperty of the property *hasDataConstraint*. The range is the data constraint class. The data constraint properties relate the functional requirements with their data constraints and specify their relevant data values.
- **has Security Constraint:** The property *hasSecurityConstraint* is defined as the subproperty of the property *hasConstraint*. The property *hasSecurityConstraint* specifies the triple security requirement (role, functional requirement, privilege) and defines whether or not a user has the privilege of accessing the functional requirement. It consists of the following subproperties:
    - **has Role Privilege:** The property *hasRolePrivilege* is defined as the subproperty of the property *hasSecurityConstraint*, with the domain class *FRequirements* and the range class *Roles*. This property relates a given functional requirement with the role that has the privilege of access to that functional requirement.
    - **hasnot Role Privilege:** The property *hasnotRolePrivilege* is defined as the subproperty of the property *hasSecurityConstraint*, with the domain class *FRequirements* and the range class *Roles*. This property relates a given functional requirement with the role that does not have the privilege of access to that functional requirement.

- **has FR Privilege:** The property *hasFRPrivilege* is defined as the subproperty of the property *hasSecurityConstraint*, with the domain class *Roles* and the range class *FRequirements*. This property relates a role with the functional requirements and the role has the privilege of access to those functional requirements.
- **hasnot FR Privilege:** The property *hasnotFRPrivilege* is defined as the subproperty of the property *hasSecurityConstraint*, with the domain class *Roles* and the range class *FRequirements*. This property relates a role with given functional requirements that the role does not have any privilege of access to those functional requirements.

Other than the property *hasConstraint* which is applied to specify the trustworthy attributes, there are some other general properties which are defined to relate the individuals as follows:

- **has Functional Requirement:** The property *hasFRequirement* is defined to relate any concept with the functional requirements that the concept provides or requests. This property consists of three subproperties as follows:
  - **has Provided FR:** The property *hasProvidedFR* is defined as the subproperty of the property *hasFRequirement*, with the range class *ProvidedFRs*. This property relates a concept with its provided functional requirements.
  - **has Requested FR:** The property *hasRequestedFR* is defined as the subproperty of the property *hasFRequirement*, with the range class *RequestedFRs*. This property relates a concept with its requested functional requirements.
  - **has Internal FR:** The property *hasInternalFR* is defined as the subproperty of the property *hasFRequirement*, with the range class *InternalFRs*. This property relates a concept with its internal functional requirements.

- **has Property:** The property *hasProperty* is defined to relate a main attribute class with its property attributes. For each multi-attribute class defined in the ontology, a new property is defined as the subproperty of the property *hasProperty* that contains the name of the main attribute of the *MClass*. The range is the *MClass* and the domain is the main attribute class. Each *hasProperty* is the inverse property of its corresponding *isPropertyOf*.
- **is Property Of:** The property *isPropertyOf* is defined to relate the property attributes with their main attribute class. For each *hasProperty* defined in ontology, an *isPropertyOf* is defined in ontology that contains the name of the main attribute of the *MClass*. The property *isPropertyOf* is the inverse property of its corresponding *hasProperty*. The domain is the *MClass* and the range is the main attribute class.

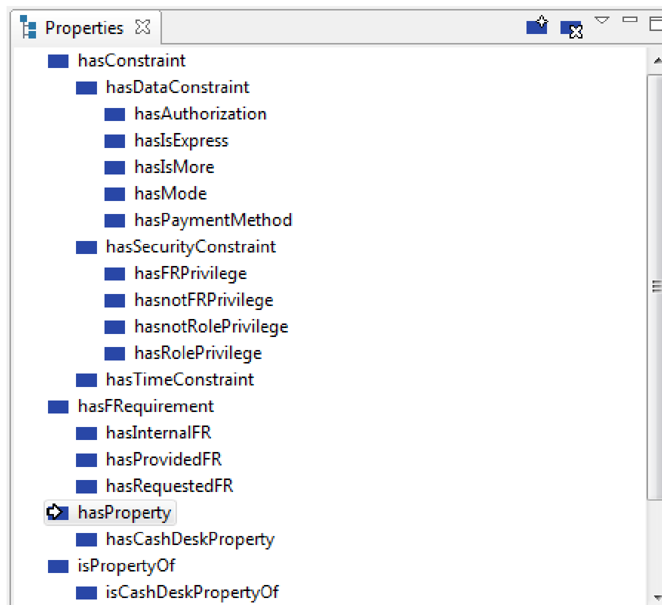

Figure 5.5: Sample hierarchy of object properties in OWL ontology

- **has Object Class Property:** The property *hasObjClassProperty* is defined to relate all object classes with their superclasses in the hierarchy. All object

classes defined in ontology are related to their superclasses by the same property, with the domain *ObjClasses*. The superclasses of a given object class are determined by traversing through the lattice hierarchy to find all superconcepts of the concept corresponding to the given object class. This process continues by extracting the superconcepts of the superconcepts until the supremum node of the lattice is reached. If any extracted superconcept is anonymous or is corresponding to another object class, it is ignored. If any obtained superconcept is *MClass*, its main attribute class is considered as the superclass of the given object class.

Figure 5.5 illustrates the hierarchy of the object properties in the obtained OWL ontology.

5.3.6 Implication Rules

In FCA, the implication rules help us to better realize the relations among the concepts. The implication rules can be derived from the propositional logic to provide the pairs of subsets of attributes, so that if an object has all attributes of a premise, then it has also all attributes of the conclusion. Besides, FCA software tools can derive the implication rules from the concept lattice of a context. The XML-format file of the concept lattice is merged with the XML-format file of the implication rules and the unified XML file is transformed into the OWL ontology.

Among the implication rules derived from the concept lattice, there is the relationship between the functional requirements attribute of a concept as the premise and its data constraints as the conclusion. Therefore, the functional requirements attributes of a concept and their related data constraints and their values are extracted from the

implication rules. As an example, we consider the concept *CashBox* and its functional requirements attribute *IFRCashBox-CheckIfExpress__FRCheckLastHour* as the premise of an implication rule and the data constraint *DCMode_Done* as the conclusion. After the transformation process and according to the object property definition rules, the ontological class corresponding to the mentioned functional requirements attribute will have the property *hasMode* with the value *Done* in the ontology.

The same implication rules exist for the actions and the data parameter updates. Since the actions and updates are not transformed into the ontological elements, they are not involved in the transformation rules to build the ontology, but will be considered in the TADL component model.

```
<rule>
  <premise>
    {IFRCashBox-CheckIfExpress_FRCheckLastHour_}
  </premise>
  <consequence>
    {CashBox_, CashDesk, CoCoME, DCMode_Done, PropertyCashDesk=CashDeskId,
    PropertyCashDesk=CashDeskPC, PropertyCashDesk=InStore, PropertyCashDesk=Sale,
    UpdateDCMode_Waiting}
  </consequence>
  <support>0.04</support>
  <confidence>1.0</confidence>
</rule>
```

5.4 Name Convention of Ontology Elements

The different elements that are formed in ontology by the automatic transformation process and their corresponding name conventions are stated in Table 5.1. The + sign is used as a notation for string concatenation in the name convention. The sign * is used as a notation for repeating the same structure in the name convention. As an example,

when defining a role class that the role name is *Cashier*, the name convention is *Role-Cashier* (see Table 5.1, row 2).

Table 5.1: Name Conventions for Ontology Elements

	Ontology Element	**Name Convention**
1	Multi-attribute Class	'MClass' + '-' + main attribute name + ('-' + 'Property' + property attribute name)*
2	Role Class	'Role' + '-' + role name
3	Object Class	'ObjClass' + '-' + object class name
4	*has property* Property	'has' + main attribute of the MClass + 'Property'
5	*is property of* Property	'is' + main attribute of the MClass + 'PropertyOf'
6	*has data constraint* Property	'has' + data constraint name

Example

As an example, we can refer to the sample context table *MergedConcepts* presented in Chapter 4, Figure 4.18 and its corresponding concept lattice presented in Chapter 4, Figure 4.19. According to the class definition rules provided in this Chapter, the ontological classes *Concept1* and *Concept2* are derived from the concepts *Concept1 and Concept2* belonging to the concept lattice *MergedConcepts*. Also, the functional requirements attribute *FRConcept1-Req3_IFRReq4* has the data constraint *hasMode* with the value *DataC2* and a time constraint with the value 5.0 seconds. According to the individual definition rules, the object *Object2* is the only individual of the class *FRConcept1-Req3_IFRReq4*. The ontological class *FRConcept1-Req3_IFRReq4* and its object are illustrated in Figure 5.6.

```
<owl:Class rdf:ID="FRConcept1-Req3_IFRReq4">
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >FRConcept1-Req3_IFRReq4
  </rdfs:label>
  <rdfs:subClassOf rdf:resource="#Concept1"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:hasValue rdf:resource="#T5.0S"/>
      <owl:onProperty>
        <rdf:Property rdf:ID="hasTimeConstraint"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:hasValue rdf:resource="#DataC2"/>
      <owl:onProperty>
        <rdf:Property rdf:ID="hasMode"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<FRConcept1-Req3_IFRReq4 rdf:ID="Object2">
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Object2
  </rdfs:label>
</FRConcept1-Req3_IFRReq4>
```

Figure 5.6: Class *FRConcept1-Req3_IFRReq4* and its object

```
<owl:Class rdf:ID="Role-Role1">
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Role-Role1</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Roles"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <rdf:Property rdf:ID="hasFRPrivilege"/></owl:onProperty>
      <owl:allValuesFrom>
        <owl:Class>
          <owl:intersectionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Req1"/>
          </owl:intersectionOf>
        </owl:Class>
      </owl:allValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Role-Role1">
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Role-Role1</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Roles"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <rdf:Property rdf:ID="hasnotFRPrivilege"/></owl:onProperty>
      <owl:allValuesFrom>
        <owl:Class>
          <owl:intersectionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Req3"/>
          </owl:intersectionOf>
        </owl:Class>
      </owl:allValuesFrom>
    </owl:Restriction>
```

Figure 5.7: Properties *hasFRPrivilege* and *hasnotFRPrivilege*

According to the object property definition rules, two 'has Security Constraints' are depicted in Figure 5.7 and Figure 5.8. As an example for 'has FR Privilege' (Figure 5.7), the class *Role1* is defined with the property *hasFRPrivilege* to have access to the functional requirement *Req1*. The property *hasnotFRPrivilege* indicates that *Role1* has no access permission to the functional requirement *Req3*. As an example for 'has Role Privilege' (Figure 5.8), the class *Req1* is defined with the property *hasRolePrivilege* to be accessed by *Role1*. The property *hasnotRolePrivilege* indicates that the functional requirement *Req3* cannot be accessed by *Role1*.

MergedConcepts.owl* - Microsoft Visual Studio

File Edit View Debug XML Tools Test Window Help

MergedConcepts.owl*

```
<owl:Class rdf:ID="Req1">
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >Req1
  </rdfs:label>
  <rdfs:subClassOf rdf:resource="#ProvidedFRs"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:hasValue rdf:resource="#Role-Role1"/>
      <owl:onProperty>
        <rdf:Property rdf:ID="hasRolePrivilege"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Req3">
  <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >Req3
  </rdfs:label>
  <rdfs:subClassOf rdf:resource="#ProvidedFRs"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:hasValue rdf:resource="#Role-Role1"/>
      <owl:onProperty>
        <rdf:Property rdf:ID="hasnotRolePrivilege"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Ready Ln 191 Col 40 Ch 40 INS

Figure 5.8: Properties *hasRolePrivilege* and *hasnotRolePrivilege*

5.5 Advantage of Proposed Technique

The proposed technique to build the OWL ontology from the FCA concept lattice has some advantages. First, the generated concept hierarchy is consistent. Next, the implication rules detect concept relationships accurately. Besides, trustworthy criteria are specified at domain level, which is the principle need for developing dependable software systems. Moreover, the defined rules and conventions improve the quality of software design. Furthermore, the proposed technique to build the OWL ontology from the concept lattice has the flexibility to be accomplished at any stage of the formal context construction. It means, the transformation rules are applied either to construct the partially formed OWL ontologies from the partially defined context tables, or to compose the final ontology corresponding to the entire unified context table. This facility provides a specific opportunity for the designer to examine the partially formed ontologies for inconsistencies, redundancies, or contradictions before the design stage terminates. This can be achieved by using the reasoning engine of the OWL ontology. The steps to be followed are (1) syntax checking, (2) consistency checking to ensure that the ontology does not contain contradictions, (3) subsumption checking, which is to ensure that a class description is more general than another class description, and (4) query answering, which involves retrieving knowledge from the knowledge base [67, 32]. Therefore, the designer can go back to the previous step, if necessary, of defining formal context tables and modify the deficiencies until all checking are successfully completed. Thus, our approach not only is applying FCA as the mathematical framework to take advantage of the formalism, but also is getting benefit from the reasoning engine of OWL ontology to ensure the dependability of the developed product.

Chapter 6

Implementation of Transformation from Concepts to Ontology

The transformation rules introduced in Chapter 5 are implemented in this Chapter. An automated model transformation technique for generating the OWL ontology from the concept lattice is proposed. The input meta-model is an XML format file that is obtained by merging the concept lattice and its derived implication rules created by the FCA software tool. The output model in OWL format consists of the definitions of ontology overview, classes, individuals, and object properties as well as the class hierarchy which includes the subclass-superclass relationships.

In Section 6.1, the structure of the input meta-model is described and the XML format of the input file that conforms to the concept lattice specifications is presented. Then, in Section 6.2, the structural elements in ontology are explained to represent the OWL output model. Section 6.3 discusses the model transformation consisting of the four transformation procedures, which are implemented by using XSLT [74, 47] model transformation framework and XPath [74] language. The transformation procedures are defined independently to accomplish the transformation process in several steps. In

each transformation step, the input model is processed to produce the output model, which in turn will be the input model of the next transformation step. Finally, the transformation rules from concept lattice to OWL ontology are fulfilled by executing all transformation steps sequentially, and the target OWL ontology is derived as the output model of the last transformation step.

6.1 Input Model

The input model of the transformation process, serialized in XML format, contains the concept lattice and its derived implication rules, which are captured from the unified formal context table in FCA. Lattice Miner [59, 13] is the FCA software tool which is employed in this thesis to generate the concept lattice and its implication rules. The generated input XML file consists of two main parts that are concatenated to compose the input model: (1) the structural elements of the concept lattice, (2) the implication rules derived from the concept lattice. It has to be mentioned that, obtained concept lattice is not a reduced labeling lattice, but a complete lattice. To clarify the input model, its structural elements are declared as follows:

6.1.1 Concept Lattice Structural Elements

- The root element <LAT> with two attributes *Desc* and *type* represents the specified concept lattice. The attribute *Desc* contains the name of the defined formal context and its corresponding concept lattice. The attribute *type* contains the lattice type, i.e., “Concept Lattice”. The root element <LAT> contains the other child elements <OBJS>, <ATTS>, <NODS>, and <rules_base>. The child

element <rules_base> is the root element of the implication rules that is discussed in Subsection 6.1.2.

- The element <OBJS> consisting of <OBJ> child elements represents the defined objects of the lattice. Each <OBJ> element has the attribute *id*. The contents of the tag <OBJ> are the names of the objects in the extent of the nodes of the concept lattice.
- The element <ATTS> consisting of the child element <ATT> represents the defined attributes of the lattice. Each <ATT> element has the attribute *id*. The contents of the tag <ATT> are the names of the attributes in the intent of the nodes of the concept lattice.
- The element <NODS> consisting of the child elements <NOD> represents the concept lattice nodes. Each <NOD> element consists of the attribute *id*, and the child elements <EXT>, <INT> and <SUP_NOD>.
- The child element <EXT> of <NOD> consists of <OBJ> child elements representing the objects in the extent of the lattice node denoted by the element <NOD>. Each <OBJ> child element has an attribute *id*.
- The child element <INT> of <NOD> consists of <ATT> child elements representing the attributes in the intent of the lattice node denoted by <NOD>. Each <ATT> child element has an attribute *id*.
- The child element <SUP_NOD> of <NOD> consists of <PARENT> child elements representing the immediate superconcepts of the lattice node denoted by <NOD>. Each <PARENT> child element has an attribute *id*.

Figure 6.1 illustrates the XML specifications of concept lattice structural elements. The tag <LAT> is the root element.

```xml
<LAT Desc=" lattice-name" type="ConceptLattice">
    <MINSUPP>0.0</MINSUPP>
    <OBJS>
        <OBJ id="object-id">object-name</OBJ>
    </OBJS>
    <ATTS>
        <ATT id="attribute-id">attribute-name</ATT>
    </ATTS>
    <NODS>
        <NOD id="node-id">
            <EXT>
                <OBJ id="object-id" />
            </EXT>
            <INT>
                <ATT id="attribute-id" />
            </INT>
            <SUP_NOD >
                <PARENT id="parent-id" />
            </SUP_NOD>
        </NOD>
    </NODS>
    <rules_base/>
</LAT>
```

Figure 6.1: Lattice specifications in XML format

MergedConcepts.lat.xml - Microsoft Visual Studio

File Edit View Debug XML Tools Test Window Help

MergedConcepts.lat.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<LAT Desc="MergedConcepts - Merged.slf - #OfNodes = 9" type="ConceptLattice">
  <MINSUPP>0.0</MINSUPP>
  <OBJS>
    <OBJ id="0">Object1</OBJ>
    <OBJ id="1">Object2</OBJ>
    <OBJ id="2">Object3</OBJ>
    <OBJ id="3">Object4</OBJ>
  </OBJS>
  <ATTS>
    <ATT id="0">Action_Req5</ATT>
    <ATT id="1">Concept1_</ATT>
    <ATT id="2">Concept2_</ATT>
    <ATT id="3">DCMode_DataC1</ATT>
    <ATT id="4">DCMode_DataC2</ATT>
    <ATT id="5">DCMode_DataC3</ATT>
    <ATT id="6">FRConcept1-Req1_FRReq2_</ATT>
    <ATT id="7">FRConcept1-Req3_IFRReq4_</ATT>
    <ATT id="8">FRConcept2-Req2_FRReq3_</ATT>
    <ATT id="9">FRConcept2-Req5_FRReq6_</ATT>
    <ATT id="10">MergedConcepts</ATT>
    <ATT id="11">PropertyMergedConcepts=MergedConceptsId</ATT>
    <ATT id="12">RolePrivilege-Role1_Req1</ATT>
    <ATT id="13">RolePrivilegeNot-Role1_Req3</ATT>
    <ATT id="14">TCFRConcept1-Req3_IFRReq4=T5.0S_</ATT>
    <ATT id="15">UpdateDCMode_DataC2</ATT>
    <ATT id="16">UpdateDCMode_DataC3</ATT>
    <ATT id="17">UpdateDCMode_DataC4</ATT>
  </ATTS>
  <NODS>
    <NOD id="0">
      <EXT>
        <OBJ id="0" />
```

Ready Ln 3 Col 16 Ch 16 INS

Figure 6.2: The Concept Lattice *MergedConcepts* XML file

As an example, we can refer to the concept lattice *MergedConcepts* presented in Chapter 4, Figure 4.19. The structural elements of the XML file derived from the concept lattice *MergedConcepts* are illustrated in Figure 6.2, which is captured by Lattice Miner.

6.1.2 Implication Rules of Concept Lattice

- The root element <rules_base> represents the implication rules of the concept lattice, that contains the child elements <specs> and <rules>.
- The element <specs> defines the specifications of the rules and consists of the child elements <context_name>, <minimal_support>, and <minimal_confidence>.
- The child element <context_name> specifies the name of the defined formal context and its corresponding concept lattice.
- The child element <minimal_support> specifies the minimum threshold on support, to use a constraint on selecting the significant and interesting rules from the set of all possible rules. The support *supp(X)* of a set of attributes X is defined as the proportion of objects in the extent of the node in the concept lattice which contain the set of attributes. At the time of capturing the implication rules from the concept lattice, the software tool asks for specifying the minimal support. Therefore, the proportion support of the selected implication rules is not less than the minimal support ($\geq\delta$).
- The child element <minimal_confidence> specifies the minimum threshold on confidence, to be used as a constraint for selecting the significant and interesting rules from the set of all possible rules. The confidence of a rule is defined as conf $(X\rightarrow Y)$ = supp$(X \cup Y)$/supp(X). It means that for the specified confidence percentage, the rule $(X \rightarrow Y)$ is correct for the objects having the set of attributes

belonging to X. At the time of capturing the implication rules from the concept lattice, the software tool asks for specifying the minimal confidence. Therefore, the proportion confidence of the selected implication rules is not less than the minimal confidence ($\geq\gamma$).

- The element <rules> consists of the child elements <rules_number> and <rules>. The element <rules_number> contains the number of the selected implication rules according to the minimal support and minimal confidence constraints. The element <rules> consists of <rule> child elements. Each <rule> element represents an implication rule of the concept lattice and consists of the child elements <premise>, <consequence>, <support>, and <confidence>.
- The child element <premise> contains a set of attributes X that are at the left-hand-side of the implication rule $X \rightarrow Y$. It means each object having all attributes of X has also all attributes of Y.
- The child element <consequence> contains a set of attributes Y that are at the right-hand-side of the implication rule $X \rightarrow Y$. It means each object having all attributes of X has also all attributes of Y.
- The child element <support> holds the proportion support of the current <rule> element that is not less than the minimal support.
- The child element <confidence> holds the proportion confidence of the current <rule> element that is not less than the minimal confidence.

Figure 6.3 depicts the XML specifications of the implication rules of concept lattice. The tag <rules-base> is the root element.

```
< rules_base>
    <specs>
        <context_name>context-name</context_name>
        <minimal_support>minimal_support</minimal_support>
        <minimal_confidence>minimal_confidence</minimal_confidence>
    </specs>
    <rules>
        <rules_number>rules_number</rules_number>
        <rule>
            <premise>{premise-name}</premise>
            <consequence>{consequence-name}</consequence>
            <support>support-number</support>
            <confidence>confidence-number</confidence>
        </rule>
    </rules>
</ rules_base >
```

Figure 6.3: Rules specifications in XML format

The implication rules depicted in Figure 6.4 are derived from the sample concept lattice *MergedConcepts* presented in Chapter 4, Figure 4.19. The implication rules are captured by Lattice Miner.

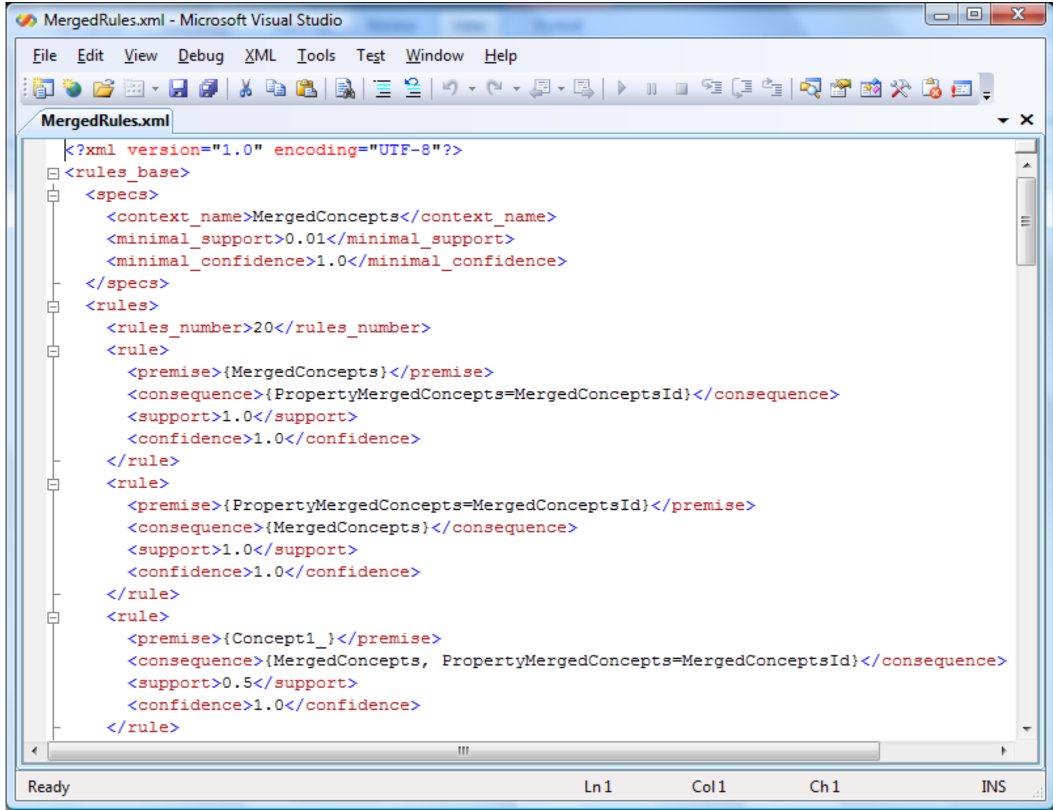

Figure 6.4: Implication rules of *MergedConcepts* Concept Lattice

6.2 Output Model

The output model of the transformation process is an OWL format file of Ontology. TopBraid Composer [80] is the ontology software tool which is employed in this thesis to open the output OWL file and demonstrate its correctness using the reasoning engine. The OWL format output model consists of the definitions of ontology overview, classes, individuals, properties, object restrictions, equivalent classes, as well as the class hierarchy which indicates the subclass-superclass relationships. To clarify the output model, its structural elements are declared as follows:

- Ontology overview consists of the base URI location, default namespace, and ontology language specified as the header of the ontology file. They are identified in the form of prefixes to abbreviate the URIs of the namespaces used in ontology. Also, an annotation element adds the information about the ontology which is provided by the tag <versionInfo>. The name of the ontology and the ontology overview parameters are defined based on the name of the concept lattice in FCA. Figure 6.5 illustrates the OWL specifications of the ontology overview.

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns="http://example.org/lattice-name#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xml:base="http://example.org/lattice-name">
    <owl:Ontology rdf:about="">
        <owl:versionInfo rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
            version-information
        </owl:versionInfo>
    </owl:Ontology>
</ rdf:RDF >
```

Figure 6.5: Ontology overview specifications in OWL format

- Ontological classes and the class hierarchy are defined based on the formal concepts and the concept hierarchy in FCA. Various class types are defined in ontology that all of them follow the same structural elements. The class specifications consist of the class name, its label name and its superclass name. Figure 6.6 illustrates the OWL specifications of the ontological classes.
- Individuals in ontology are defined based on the objects of the extents in the concept lattice. The individual specifications consist of the individual name and its label name. Figure 6.7 shows the OWL specifications of the individuals.

```
<owl:Class rdf:ID="class-name">
    <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >label-name
    </rdfs:label>
    <rdfs:subClassOf>
        <owl:Class rdf:about="#superclass-name"/>
    </rdfs:subClassOf>
</owl:Class>
```

Figure 6.6: Ontological class specifications in OWL format

```
< class-name rdf:ID=”individual-name”>
    <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >individual-name
    </rdfs:label>
</class-name>
```

Figure 6.7: Individual specifications in OWL format

- Object properties in ontology are defined based on the various relationships that hold among the formal concepts. Various object property types are defined in ontology that all of them follow the same structural elements. The object property specifications consist of the property name, its inverse property name if applicable, its domain and range class names, and its super-property name. Figure 6.8 illustrates the OWL specifications of the object properties.

```
<rdf:Property rdf:about="#property-name">
  <owl:inverseOf>
     <rdf:Property rdf:about="#inverse-property-name"/>
  </owl:inverseOf>
  <rdfs:domain rdf:resource="#class-name"/>
  <rdfs:range rdf:resource="# class-name "/>
  <rdfs:subPropertyOf>
     <rdf:Property rdf:about="#super-property-name"/>
  </rdfs:subPropertyOf>
</rdf:Property>
```

Figure 6.8: Object property specifications in OWL format

- Property restrictions are defined for the classes that hold the object property relations. In addition to the class specifications, the property restriction consists of another subclass element that contains the restriction child element that is imposed on a property. It can be a *quantifier* restriction, *cardinality* restriction, or *hasValue* restriction. Figure 6.9 and Figure 6.10 illustrate the OWL specifications of the two sample property restrictions.

```
<owl:Class rdf:ID="class-name">
   <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >label-name
   </rdfs:label>
   <rdfs:subClassOf>
      <owl:Class rdf:about="#superclass-name"/>
   </rdfs:subClassOf>
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty>
            <rdf:Property rdf:ID="property-name"/>
         </owl:onProperty>
         <owl:allValuesFrom>
            <owl:Class>
               <owl:intersectionOf rdf:parseType="Collection">
                  <owl:Class rdf:about="#class-name"/>
               </owl:intersectionOf>
            </owl:Class>
         </owl:allValuesFrom>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

Figure 6.9: AllValuesFrom Property restriction specifications in OWL format

```
<owl:Class rdf:ID="class-name">
    <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >label-name
    </rdfs:label>
    <rdfs:subClassOf>
        <owl:Class rdf:about="#superclass-name"/>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty>
                <rdf:Property rdf:ID="property-name"/>
            </owl:onProperty>
            <owl:hasValue rdf:resource="#property-value"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

Figure 6.10: HasValue Property restriction specifications in OWL format

```
<owl:Class rdf:ID="class-name">
    <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >label-name
    </rdfs:label>
    <rdfs:subClassOf>
        <owl:Class rdf:about="#superclass-name"/>
    </rdfs:subClassOf>
    <owl:equivalentClass>
        <owl:Class>
            <owl:oneOf rdf:parseType="Collection">
                <class-name rdf:ID="individual-value"/>
            </owl:oneOf>
        </owl:Class>
    </owl:equivalentClass>
</owl:Class>
```

Figure 6.11: Equivalent class specifications in OWL format

- Equivalent classes are other class axioms that are used to define another relation type between the classes in ontology. So, a class can be defined as the equivalent class of another class or some more restrictions can be imposed to simulate the specialization between the classes. Moreover, a class can be defined equivalent to the set of individuals that are described in the enumeration. In this case, these individuals have been asserted to be all different from each other and the class can only be one of them and nothing else. Figure 6.11 illustrates the OWL specifications of the equivalent class to the set of individuals.

As part of the class specifications, the equivalent class consists of another <owl:Class> element that contains the 'oneOf' restriction that is imposed on the set of individuals.

6.3 Model Transformation

In this Section, our proposed transformation techniques is introduced to transform the concept lattice as the input model into the OWL ontology as the output model. This model transformation consists of four transformation procedures: (1) Lattice Reducer, (2) Class Definer, (3) Pre-phase Definition, and (4) Ontology Builder. Each transformation procedure is defined independently. They are executed successively to accomplish the transformation approach through several steps.

The transformation procedures are implemented by using XSLT model transformation framework and XPath language. As discussed before, the procedures use a hybrid of Push and Pull methods to do the process. Templates are used to match the nodes that get pushed to the output XML files and the specified nodes are selected to change the structure of files. Each transformation procedure is separately defined by a XSL Stylesheet.

In the beginning, the transformation procedures specified by XSL Stylesheets are implemented by XSLT 2.0 using the trial version of Oxygen XML Editor 11.2. Later on, the XSLT transformation is performed by java programming using the XSLT jar files. The second platform has two advantages. First, the resulting XSLT transformation programs are independent of different versions of tools. Second, the defined transformation procedures are executed as an application package, all at once, which is much more efficient.

In each transformation step, the input model is processed to produce the output model, which in turn will be the input model of the next transformation step. By executing the transformation procedures successively, the transformation process is completed and the OWL ontology is obtained. However, the entire transformation process is automatically done in the background. Figure 6.12 depicts the model transformation from concepts to ontology.

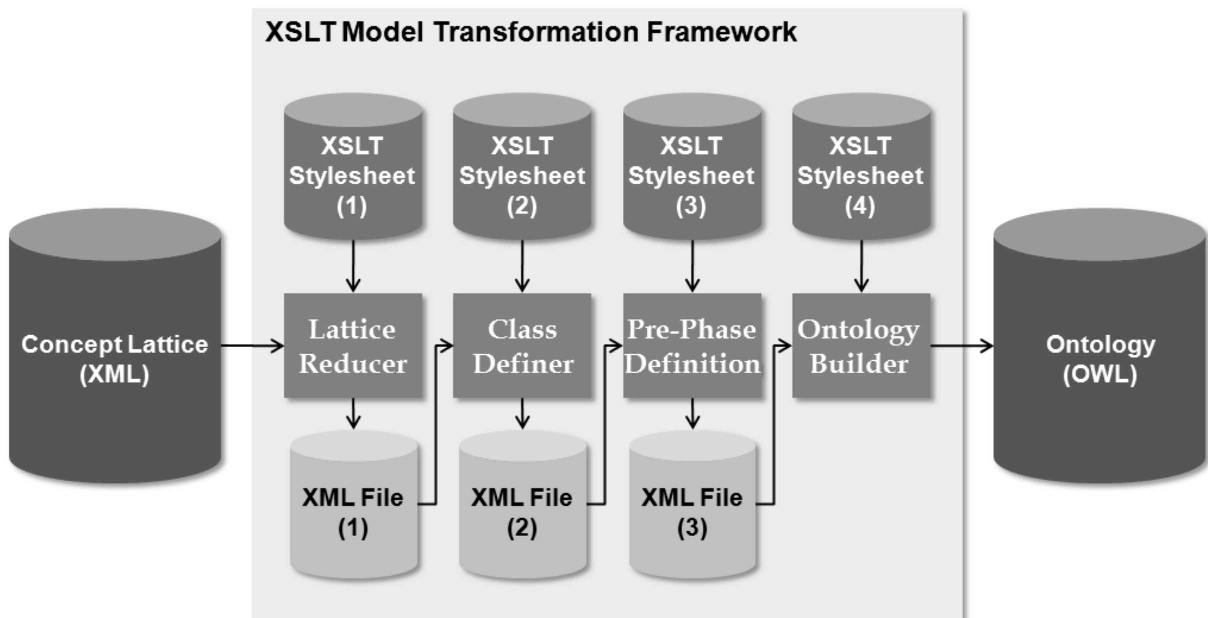

Figure 6.12: Model Transformation from Concepts to Ontology

6.3.1 Lattice Reducer Procedure

Lattice Reducer is the first transformation procedure that transforms the concept lattice to the reduced labeling lattice. Also, it prunes the input XML file by removing the extra and inappropriate attributes that are generated through the conversion process from VCT to BCT in lattice Miner software tool. The input model described in Section 6.1

is the input XML file of the Lattice Reducer procedure that is transformed into another intermediate XML file. Lattice Reducer procedure consists of the following steps which are executed successively:

- **Step 1:** The attributes *type* and *Desc* of the element <LAT>, the element <OBJS>/<OBJ>, the element <NOD>/<SUP_NOD>/<PARENT>, and the element <LAT>/<rules>, as well as its child elements and attributes are unchanged by the transformation.
- **Step 2:** The elements <ATTS>/<ATT>, which are the attributes of the concept lattice are unchanged by the transformation, except for the elements ending with slash character. For such <ATTS>/<ATT> elements, the attribute *comment* with the value "Ignored" is added. These attributes correspond to the objects that no value is specified for them and are automatically generated by Lattice Miner through the conversion process from VCT to BCT.
- **Step 3:** To transform the concept lattice to the reduced labeling lattice, if any of their extent objects is found in the succeeding <NOD> elements, it must be removed. So, for such <NOD>/<EXT>/<OBJ> element, a *comment* attribute with the value "Ignored" is added.
- **Step 4:** To transform the concept lattice to the reduced labeling lattice, if any of their intent attributes is found in the preceding <NOD> elements, it must be removed. So, for such <NOD>/<INT>/<ATT> element, a *comment* attribute with the value "Ignored" is added.
- **Step 5:** For the elements <NOD>/<INT>/<ATT> that are time constraint, data constraint, update, action, or role privilege, a *comment* attribute with the value "Ignored" is added.

- **Step 6:** The element <NOD>/<INT>/<ATT> which ends with slash character is tagged as “Ignored” to be eliminated from the output model.

As an example, the Lattice Reducer procedure is applied to the sample concept lattice *MergedConcepts*, which is introduced in Chapter 4. The XML-format of the concept lattice *MergedConcepts* is illustrated in Figure 6.2 and is considered as the input of the Lattice Reducer procedure. The yielded output XML file is depicted in Figure 6.13.

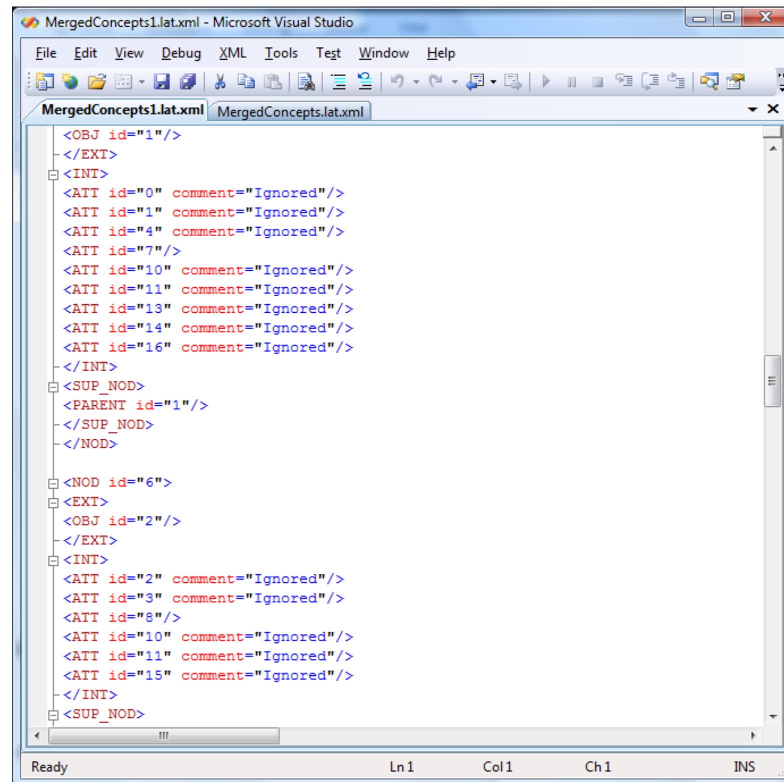

Figure 6.13: Output XML file of Lattice Reducer procedure

6.3.2 Class Definer Procedure

Class Definer is the second transformation procedure that determines the ontological classes corresponding to the formal concepts in the concept lattice. The subclass-

superclass relations in addition to the property relations are figured out. Also, the safety properties and the security constraints are identified. Finally, the functional requirements as a part of the functional requirements attributes are specified with their data constraints, updates, and reactions. The output XML file acquired from Lattice Reducer procedure is the input of the Class Definer procedure. The input is transformed into another intermediate XML file. Class Definer procedure consists of the following steps, which are executed successively:

- **Step 1:** The attributes *type* and *Desc* of the element <LAT>, the element <OBJS>/<OBJ>, the element <NOD>/<SUP_NOD>/<PARENT>, and the element <LAT>/<rules>, as well as its child elements and attributes are unchanged by the transformation.
- **Step 2:** The elements <ATTS>/<ATT>, which are the attributes of the concept lattice, are unchanged by the transformation, except for the intent attributes having the *comment* attribute with the value "Ignored".
- **Step 3:** The child elements <EXT>/<OBJ> and <INT>/<ATT> of the element <NODS>/<NOD> are unchanged by the transformation, except for the extent objects and intent attributes having the *comment* attribute with the value "Ignored".
- **Step 4:** The new element <NODNAMES>/<NOD>, having the attributes *id*, *name*, *parent*, and *mainName*, is added to the output XML file. Each <NOD> element corresponds to a node in the concept lattice with the same *id*, which contains the detected ontological class name and its superclass. The element <NOD> without any attribute *name* and *parent,* called anonymous node, is not the candidate for any ontological classes. The detected ontological classes are from various class types such as root class, attribute/multi-attribute class, and object class (described in Chapter 5). The attribute *mainName* is assigned to the name of the

main attribute of the multi-attribute class. For the object class, all superconcepts are extracted from the lattice hierarchy and added as the element <NODNAMES>/<NOD>/<SUP>. The technique for determining the superconcepts of an object class is described in Chapter 5.

- **Step 5:** To specify the time constraint properties, the new element <TIMECONSTRAINTS>/<TC>, having the attributes *id*, *name*, and *value*, is added to the output XML file. Each <TC> element corresponds to the time constraint of either an attribute class or a property attribute class. The attribute *id* is assigned to the *id* attribute of the corresponding time constraint property of the element <ATTS>/<ATT>. The attribute *name* is assigned to the attribute class name that time constraint is defined for. The attribute *value* is assigned to the time value. Only one time constraint may be defined for any attribute class or property attribute class.
- **Step 6:** To specify the data constraint properties, the new element <DATACONSTRAINTS>/<DC>, having the attributes *id* and *name*, is added to the output XML file. Each <DC> element corresponds to a data constraint property of an attribute class. The attribute *id* is assigned to the *id* attribute of the corresponding data constraint property of the element <ATTS>/<ATT>. The attribute *name* contains both the data constraint name and value.
- **Step 7:** To specify the updates of data constraint properties, the new element <UPDATEDATACONSTRAINTS>/<UDC>, having the attributes *id* and *name*, is added to the output XML file. Each <UDC> element corresponds to an updated data constraint. The attribute *id* is assigned to the *id* attribute of the corresponding update of the element <ATTS>/<ATT>. The attribute *name* contains both the updated data constraint name and value.

- **Step 8:** To specify the actions, the new element <ACTIONS>/<ACT>, having the attributes *id*, *name*, and *component*, is added to the output XML file. Each <ACT> element corresponds to a functional requirements attribute that may contain a set of actions. The attribute *id* is assigned to the *id* attribute of the corresponding action of the element <ATTS>/<ATT>. The attribute *name* contains the action name. The attribute *concept* is assigned to the concept name that the action belongs to. The concept name is derived from the implication rules.
- **Step 9:** To specify the property *hasProperty*, the new element <HASPROPERTIES>/<HP>, having the attributes *id*, *domain*, and *name*, is added to the output XML file. The element <HP> corresponds to the property *hasProperty*, which generates the relation between a main attribute class and its property attribute classes. The attribute *id* is assigned to the *id* attribute of the corresponding property attribute of the element <ATTS>/<ATT>. The attribute *domain* is assigned to the main attribute class name. The attribute *name* is assigned to the property attribute class name.
- **Step 10:** To specify the role privileges, the new element <ROLEPRIVILEGES>/<RP>, having the attributes *id* and *name*, is added to the output XML file. Each <RP> element corresponds to a role privilege attribute. The attribute *id* is assigned to the *id* attribute of the corresponding role privilege of the element <ATTS>/<ATT>. The attribute *name* contains both the role name and the functional requirement name that the role has the privilege of accessing. Also, the attribute *name* may contain the role name that does not have any privilege of accessing to the given functional requirement.
- **Step 11:** The new element <FREQUIREMENTS>/<FR>, having the attributes *name*, *from*, and *to*, is added to the output XML file. The element <FR>

corresponds to a functional requirements attribute containing the provided and requested functional requirements. The attribute *name* is assigned to the functional requirement attribute name. The attribute *from* is assigned to the concept name having the provided functional requirement. The attribute *to* is assigned to the concept name having the requested functional requirement. For the internal functional requirements, the attributes *from* and *to* will indicate the internal functional requirements.

- **Step 12:** The element <FR>/<DCS> specifies the data constraint of the functional requirements attribute. The element <DCS>/<DC>, having the attribute *name*, corresponds to a data constraint of the functional requirements attribute. The attribute *name* is assigned to the data constraint name.
- **Step 13:** The element <FR>/<UPDATES> specifies the updated data constraint. The element <UPDATES>/<UP>, having the attribute *name*, corresponds to an updated data constraint. The attribute *name* is assigned to the name of the updated data constraint and its new value.
- **Step 14:** The element <FR>/<FRACTIONS> specifies the actions of the functional requirements attribute. The element <FRACTIONS>/<AC>, having the attribute *name*, corresponds to a functional requirements attribute that may contain a set of actions. The attribute *name* contains the functional requirement attribute *name*.

As an example, the Class Definer procedure is applied to the sample concept lattice *MergedConcepts*. The XML file illustrated in Figure 6.13 is considered as the input of the Class Definer procedure. The yielded output XML file is depicted in Figure 6.14.

```xml
</NODS>
<NODNAMES>
<NOD id="0" name="MClass-MergedConcepts-PropertyMergedConceptsId" mainName="MergedConcepts" parent="Thing"/>
<NOD id="1" name="Concept1" parent="MergedConcepts"/>
<NOD id="2"/>
<NOD id="3" name="Concept2" parent="MergedConcepts"/>
<NOD id="4" name="FRConcept1-Req1_FRReq2" parent="Concept1"/>
<NOD id="5" name="FRConcept1-Req3_IFRReq4" parent="Concept1"/>
<NOD id="6" name="FRConcept2-Req2_FRReq3" parent="Concept2"/>
<NOD id="7" name="FRConcept2-Req5_FRReq6" parent="Concept2"/>
<NOD id="8"/>
</NODNAMES>
<TIMECONSTRAINTS>
<TC id="14" name="FRConcept1-Req3_IFRReq4" value="T5.0S"/>
</TIMECONSTRAINTS>
<DATACONSTRAINTS>
<DC id="3" name="Mode_DataC1"/>
<DC id="4" name="Mode_DataC2"/>
<DC id="5" name="Mode_DataC3"/>
</DATACONSTRAINTS>
<UPDATEDATACONSTRAINTS>
<UDC id="15" name="UpdateDCMode_DataC2"/>
<UDC id="16" name="UpdateDCMode_DataC3"/>
<UDC id="17" name="UpdateDCMode_DataC4"/>
</UPDATEDATACONSTRAINTS>
<ACTIONS>
<ACT id="0" name="Action_Req5" concept="Concept1"/>
</ACTIONS>
<HASPROPERTIES>
<HP id="11" domain="MergedConcepts" name="MergedConceptsId"/>
```


Figure 6.14: Output XML file of Class Definer procedure

6.3.3 Pre-phase Definition Procedure

Pre-phase Definition is the third transformation procedure that determines the parent nodes, concept relation types and functional requirement types, which are the necessary data for the next phase where the OWL ontology will be composed. The output XML file acquired from Class Definer procedure is the input model of the Pre-phase Definition procedure that is transformed into another intermediate XML file. Pre-phase Definition procedure consists of the following steps.

- **Step 1:** The attributes *type* and *Desc* of <LAT>, its child elements <OBJS>, <ATTS>, <NODS>, and <NODNAMES>, besides all their child elements and attributes are unchanged by the transformation. Also, the child elements <TIMECONSTRAINTS>, <DATACONSTRAINTS>, <HASPROPERTIES>,

<ROLEPRIVILEGES>, <FREQUIREMENTS>, <UPDATEDATACONSTRAINTS> , <ACTIONS>, and <rules> of <LAT>, as well as their child elements and attributes are unchanged by the transformation.

- **Step 2:** To specify the subclass-superclass hierarchy among the concepts, the new element <PARENTNODES>/<PN>, having the attributes *id* and *name,* are added to the output XML file. Each <PN> element corresponds to a class representing a concept. The first <PN> element contains the root class representing the main concept. The attribute *id* is assigned to the *id* attribute of the corresponding class node of the element <NODNAMES>/<NOD>. The attribute *name* is assigned to the class name.
- **Step 3:** The element <PN>/<CHILDREN>/<CH>, having the attribute *name*, corresponds to only the subclasses of the element <PN> which represent the concepts. The attribute *name* contains the subclass name.
- **Step 4:** To specify the relations among the functional requirements of the concepts, the new element <ConceptRelationTypes>/<CONCEPT>, having the attribute *name*, is added to the output XML file. Each <CONCEPT> element corresponds to a class representing a concept, which has relation with other concepts. The attribute *name* is assigned to the *name* attribute of the corresponding class node of the element <NODNAMES>/<NOD>. For the multi-attribute classes, the attribute *mainName* is specified.
- **Step 5:** The element <CONCEPT>/<ConceptRelations>/<CR>, having the attribute *name*, corresponds only to the classes, which represent the concepts and have relation with the element <CONCEPT>.
- **Step 6:** To specify various functional requirement types of the concepts, the new element <FRTYPES>/<CONC>, having the attribute *name,* is added to the output

XML file. Each <CONC> element corresponds to a class representing a concept, which has relation with other concepts. The attribute *name* is assigned to the *name* attribute of the corresponding class node of the element <NODNAMES>/<NOD>. For the multi-attribute classes, the attribute *mainName* is specified.

- **Step 7:** The element <FRTYPES>/<CONC> consists of the child elements <InternalFRs>, <ProvidedFRs>, and <RequestedFRs>, which contain the child elements <INTFR>, <PFR>, and <RFR> respectively. The attribute *name* of these elements contains the functional requirement names of their relevant functional requirement types.

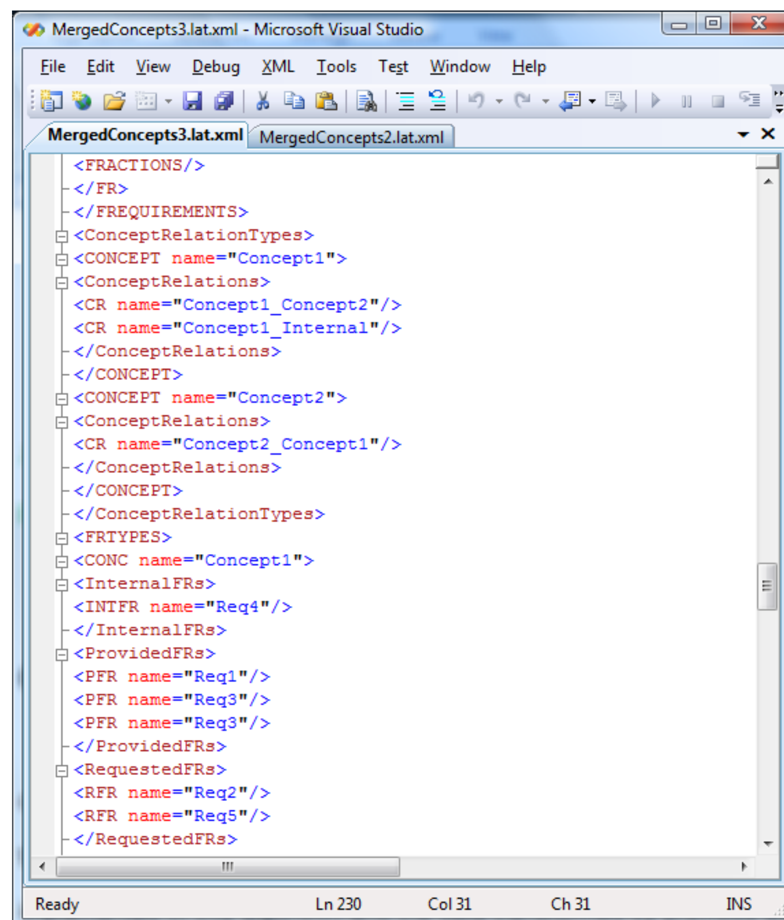

Figure 6.15: Output XML file of Pre-phase Definition procedure

As an example, the Pre-phase Definition procedure is applied to the sample concept lattice *MergedConcepts*. The XML file illustrated in Figure 6.14 is considered as the input of the Pre-phase Definition procedure. The yielded output XML file is depicted in Figure 6.15.

6.3.4 Ontology Builder Procedure

Ontology Builder is the fourth and last transformation procedure that builds the target OWL ontology. The output XML file acquired from Pre-phase Definition procedure is the input model of the Ontology Builder procedure. Actually, the transformation rules described in Chapter 5 are implemented in this procedure and the data transformed by previous procedures are used to do this job properly. One of the benefits of developing several procedures to do the transformation process is to exempt the designer from defining all details in the context tables of FCA. That means, only the required data are manually specified by user and the rest of the job, which are expanding and extracting the necessary information from the data, are accomplished automatically by the model transformation process. The OWL format output model consists of the definitions of ontology overview, classes, individuals, object properties, object restrictions, equivalent classes, as well as the class hierarchy which includes the subclass-superclass relationships. Ontology Builder procedure consists of the following steps which are executed successively:

- **Step 1:** Ontology Overview is defined based on the attribute *Desc* of the element <LAT>. Also, an annotation about the ontology is added as the tag <owl:versionInfo> to the output OWL file.

- **Step 2:** The property *hasConstraint*, its subproperty *hasSecurityConstraint*, and also the subproperties of the property *hasSecurityConstraint* are defined.
- **Step 3:** The class *Roles* is defined as the subclass of the root class. Then, the role names and privileges of accessing to the functional requirement names are extracted from the element <ROLEPRIVILEGES>/<RP>, and defined as the subclasses of the class *Roles*. For each defined role class, the restriction *allValuesFrom* is defined on the properties *hasFRPrivilege* and *hasnotFRPrivilege*.
- **Step 4:** The class *FRequirements* is defined as the subclass of the root class. Then, the classes *ProvidedFRs*, *RequestedFRs*, and *InternalFRs* are defined as the subclasses of the class *FRequirements*.
- **Step 5:** The property *hasFRequirement* and its subclasses *hasProvidedFR*, *hasRequestedFR*, and *hasInternalFR* are defined.
- **Step 6:** All provided, requested, and internal functional requirement classes are extracted from the element <FRTYPES>/<CONC> and are defined as the subclasses of their relevant functional requirement classes. The actions are also defined as the requested functional requirement classes. Besides, the role privileges of the functional requirements are extracted from the element <ROLEPRIVILEGES>/<RP>, and for each defined functional requirement class, the restriction *hasValue* are defined on the properties *hasRolePrivilege* and *hasnotRolePrivilege*.
- **Step 7:** The property *hasDataConstraint* is defined as the subproperty of the property *hasConstraint*. Then, all data constraints are extracted from the element <DATACONSTRAINTS>/<DC>, and are defined as the subproperties of the property *hasDataConstraint*.

- **Step 8:** The class *DataConstraints* is defined as the subclass of the root class. Then, all data constraints are extracted from the element <DATACONSTRAINTS>/<DC>, and are defined as the subclasses of the class *DataConstraints*. Besides, their values are specified by the equivalent class definition.
- **Step 9:** The property *hasTimeConstraint* is defined as the subproperty of the property *hasConstraint*.
- **Step 10:** The class *TimeConstraint* is defined as the subclass of the root class. Then, all time constraint values are extracted from the element <TIMECONSTRAINTS>/<TC>, and are specified by the equivalent class definition.
- **Step 11:** The property *hasProperty* is defined. Then, for each main attribute class extracted from the element <HASPROPERTIES>/<HP>, a property is defined as the subproperty of the property *hasProperty*. The defined property is determined as the inverse property of its corresponding property *isPropertyOf*.
- **Step 12:** The property *isPropertyOf* is defined. Then, for each main attribute class extracted from the element <HASPROPERTIES>/<HP>, a property is defined as the subproperty of the property *isPropertyOf*. The defined property is determined as the inverse property of its corresponding property *hasProperty*.
- **Step 13:** For the object classes, the class *ObjClasses* is defined as the subclass of the root class. Also, the property *hasObjClassProperty* is defined.
- **Step 14:** For all nodes specified in the element <NODNAMES>/<NOD>, the corresponding classes and their superclasses are defined. For multi-attribute classes, the main attribute and the property attributes are defined as the subclasses of the *MClass*.

- **Step 15:** The object classes are defined as the subclasses of the class *ObjClasses*. Also, the restriction *someValuesFrom* is defined on the property *hasObjClassProperty* for the superclasses of the object class that are extracted from the element <NODNAMES>/<NOD >/<SUP>.
- **Step 16:** For the classes that have any corresponding time constraint property in the element <TIMECONSTRAINTS>/<TC>, the restriction *hasValue* is defined on the property *hasTimeConstraint*.
- **Step 17:** For the multi-attribute classes, the restriction *someValuesFrom* is defined on the property *hasProperty* of the main attribute class. Also, the restriction *someValuesFrom* is defined on the property *isPropertyOf* of the property attribute classes.
- **Step 18:** For the classes that have any corresponding functional requirements in the element <FRTYPES>/<CONC>, the restriction *allValuesFrom* is defined on the relevant subproperties of the property *hasFRequirement*.
- **Step 19:** For the classes that have any corresponding data constraints on the functional requirements in the element <FREQUIREMENTS>/<FR>/<DCS>/<DC>, the restriction *hasValue* is defined on the relevant subproperties of the property *hasDataConstraint*.
- **Step 20:** For the classes that have any extent objects in the element <EXT>/<OBJ> of the corresponding nodes, the object names are extracted from the element <OBJS>/<OBJ> and defined as the individuals of the classes.

As an example, the Ontology Builder procedure is applied to the sample concept lattice *MergedConcepts*. The XML file illustrated in Figure 6.15 is considered as the input of the Ontology Builder procedure. The yielded output OWL file is depicted in Figure 6.16.

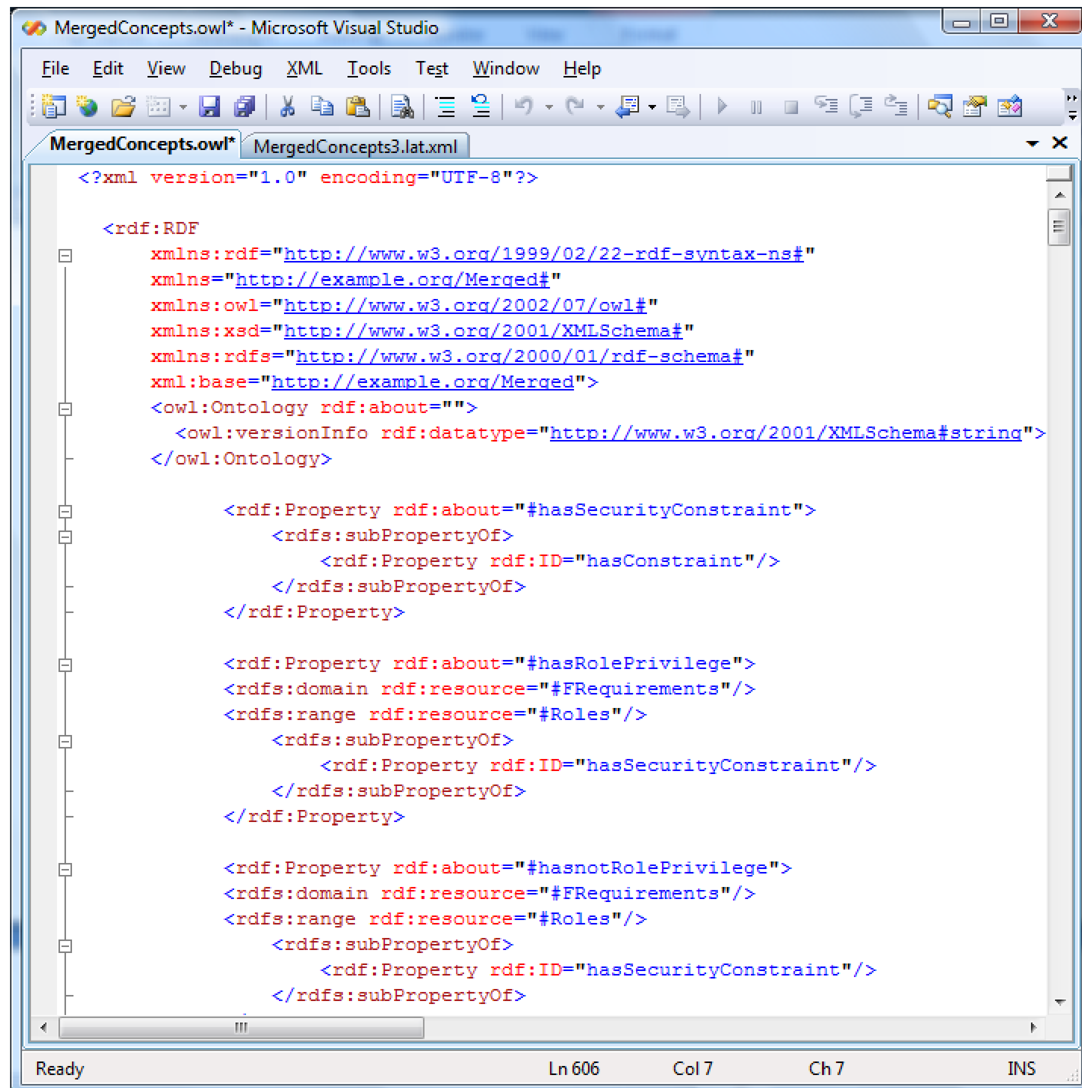

Figure 6.16: Output OWL file of Ontology Builder procedure

The obtained OWL file is opened in the TopBraid Composer software tool. The derived *MergedConcepts* OWL ontology is illustrated in Figure 6.17.

Figures 6.18 (a) and Figure 6.18 (b) illustrate the proposed approach and the contribution of this thesis. Domain analysis is fulfilled by the application of Formal Concept Analysis. The provided framework consists of two model transformation process: from formal concepts to OWL ontology, and from OWL ontology to TADL description language. The transformation process is done by using the XSLT model

transformation framework. The FCA software tool, Lattice Miner, and the ontology software tool, TopBraid Composer are applied to generate formal TADL artifacts. Finally, the obtained TADL description language may be used in component-based software development.

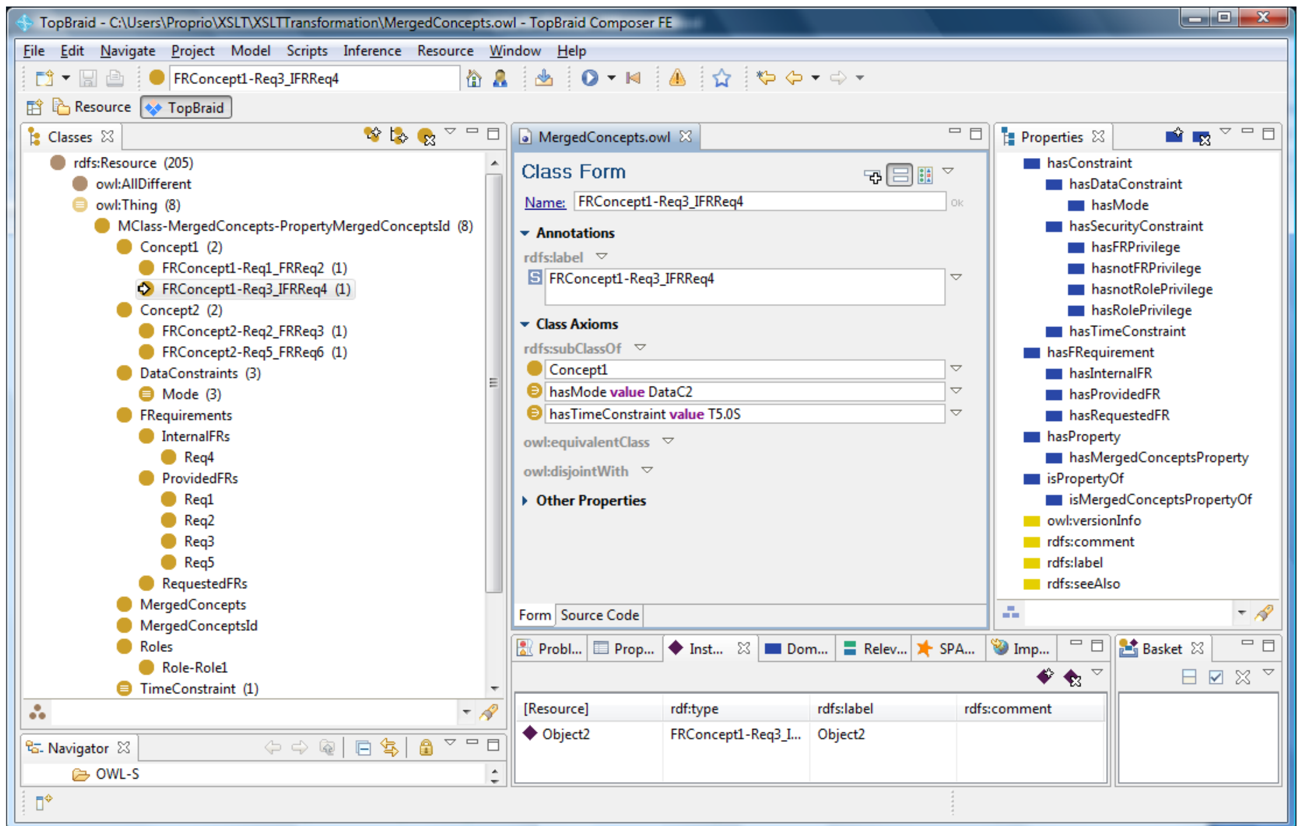

Figure 6.17: *MergedConcepts* OWL Ontology

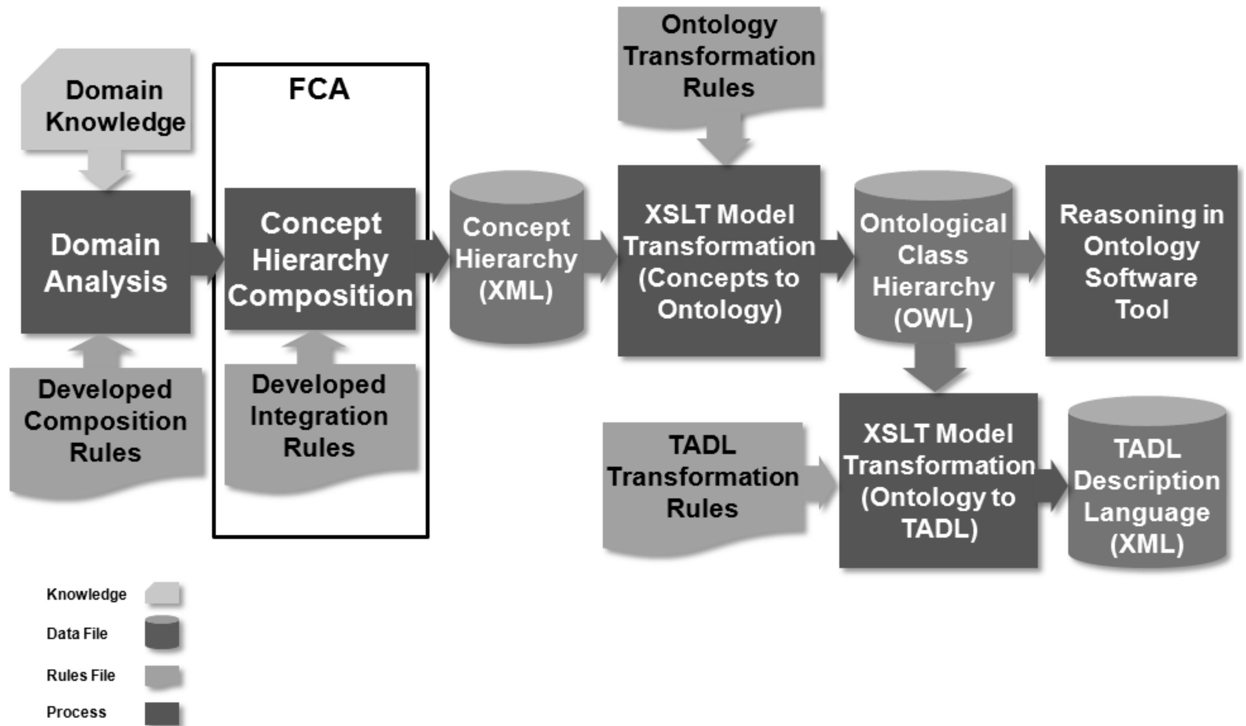

Figure 6.18 (a): The schema of the proposed approach

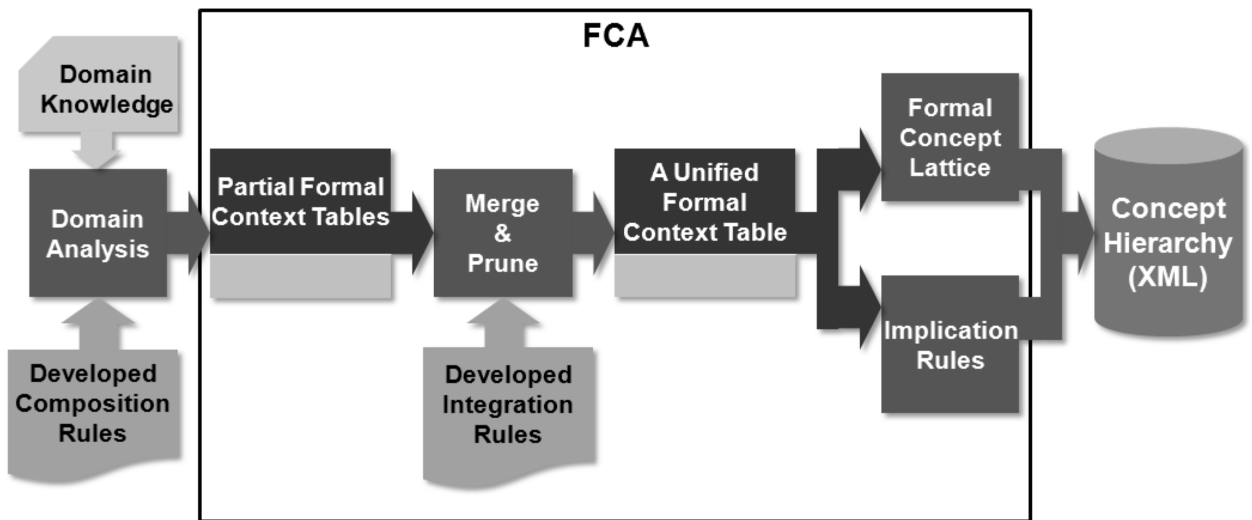

Figure 6.18 (b): FCA part of the proposed approach

Chapter 7

Model Transformation from Ontology to Components

In this Chapter, the OWL ontology is automatically transformed into the architecture description language TADL [49], which in turn will be used to develop the trustworthy component-based systems. To do so, we have implemented a model transformation framework to produce automatically the specifications of reusable components and component-based architecture of the relevant trustworthy system. The input meta-model is an OWL format file that consists of the definitions of ontology overview, classes, individuals, properties, safety and security requirement properties, as well as the class hierarchy, including the subclass-superclass relationships. The output model in XML format is TADL architecture description language. The transformation rules are captured from [48], which are implemented by applying XSLT [74, 47] model transformation framework and XPath [74] language.

In Section 7.1, the structure of the output meta-model is introduced by presenting the XML schemas of TADL. Section 7.2 presents the transformation rules. Section 7.3

discusses the transformation model from OWL ontology to TADL description language, which specifies the reusable components and the component-based architecture.

7.1 TADL XML Schemas

TADL [49], the trustworthy architecture description language, is defined as a high level specification for dependable component models. TADL has an XML-based representation containing the system definition which satisfies the trustworthy component model XML-based schemas. The XML schemas that correspond to the TADL description are explained as follows.

7.1.1 InterfaceType Schema

The interface type schema contains an ordered sequence of the following sub elements:

- **name:** a simple element to specify the name of the interface.
- **protocol:** a simple element to specify the protocol of an interface.
- **Attribute:** a complex element to specify the attributes of an interface. It is an ordered sequence of the simple elements *name*, *datatype*, *value*, and *description*.
- **ServiceType:** a complex element to specify the service types of an interface. It is an ordered sequence of the following sub-elements:
    - ❖ **name:** a simple element to specify the name of a service
    - ❖ **id:** a simple element to specify a unique identifier for a service.
    - ❖ **type:** a simple element to specify a service type.

- **Attribute:** a complex element to specify the attributes of a service. It is an ordered sequence of the simple elements *name*, *datatype*, *value*, and *description*.
- **constraint:** a simple element to specify the constraints in a service.
- **ParameterType:** a complex element to specify the parameters in a service. It is an ordered sequence of the simple elements *name*, *datatype*, *value*, and *description*.
- **Property:** a complex element to specify the properties of a service. It is an ordered sequence of the simple elements *name and value.*
- **description:** a simple text element to store the annotation of the service.

- **description:** a simple text element to store the annotation of the interface.

The terms minOccurs and maxOccurs in the XML Schema respectively specify the minimum and maximum occurrences of an element.

7.1.2 ComponentType Schema

The component type schema contains an ordered sequence of the following sub elements:

- **name:** a simple element to specify the name of a component.
- **Property:** a complex element to specify the properties of a component. It is an ordered sequence of the simple elements *name and value.*
- **Attribute:** a complex element to specify the attributes of a component. It is an ordered sequence of the simple elements *name*, *datatype*, *value*, and *description.*
- **constraint:** a simple element to specify the constraints of a component.

- **User:** a complex element to define the users of a component. The schema of the user is discussed in section 7.1.6.
- **InterfaceType:** a complex element to specify the interfaces of a component. The schema of the interface type is discussed in section 7.1.1.
- **ArchitectureType:** a complex element to specify the architectural structure of a component. It is composed of an ordered sequence of the following elements:
  - ❖ **name:** a simple element to specify the name of an architecture.
  - ❖ **ComponentType:** a complex element to specify the components in the architecture. The schema of the component type is discussed in section 7.1.2.
  - ❖ **ConnectorType:** a complex element to specify the connectors in the architecture. The schema of the connector type is discussed in section 7.1.3.
  - ❖ **Attribute:** a complex element to specify the attributes of the architecture. It is an ordered sequence of the simple elements *name*, *datatype*, *value*, and *description*.
  - ❖ **constraint:** a simple element to specify the constraints in an architecture.
  - ❖ **Attachment:** a complex element to define the connections of a connector role and the interface of a component. It is composed of an ordered sequence of the following elements:
    - **name:** a simple element to specify the name of an attachment.
    - **ConnectorType:** a complex element to specify the connector in an attachment. The schema of the connector type is discussed in section 7.1.3.
    - **ConnectorRoleType:** a complex element to specify the connector role in an attachment. The schema of the connector role type is discussed in section 7.1.3.

    - **InterfaceType:** a complex element to specify the interface in an attachment. The schema of the interface type is discussed in section 7.1.1.
    - **ComponentType:** a complex element to specify the component in an attachment. The schema of the component type is discussed in section 7.1.2.
    - **InterfaceType:** a complex element to specify the other interface in an attachment. The schema of the interface type is discussed in section 7.1.1.
    - **description:** a simple text element to store the annotation of the attachment.
  - ❖ **description:** a simple text element to store the annotation of the architecture.
- **ContractType:** a complex element to specify the safety contract of a component. The schema of the contract type is discussed in section 7.1.4.
- **description:** a simple text element to store the annotation of the component.

7.1.3 ConnectorType Schema

The connector type schema contains an ordered sequence of the following sub elements:

- **name:** a simple element to specify the name of a connector.
- **ConnectorRoleType:** a complex element to specify the roles of a connector. It is composed of an ordered sequence of the following elements:
  - ❖ **name:** a simple element to specify the name of the connector role.

- **Attribute:** a complex element to specify the attributes of a connector role. It is an ordered sequence of the simple elements *name*, *datatype*, *value*, and *description*.
- **constraint:** a simple element to specify the constraints of a connector role.
- **InterfaceType:** a complex element to specify the interface attached to the connector role. The schema of the interface type is discussed in section 7.1.1.
- **description:** a simple text element to store the annotation of the connector role.

- **Attribute:** a complex element to specify the attributes of a connector. It is an ordered sequence of the simple elements *name*, *datatype*, *value*, and *description*.
- **constraint:** a simple element to specify the constraints of a connector.
- **description:** a simple text element to store the annotation of the connector.

7.1.4 ContractType Schema

The contract type schema contains an ordered sequence of the following sub elements:

- **name:** a simple element to specify the name of a contract.
- **DataConstraint:** a complex element to specify the data constraints in a reactivity of a contract. It is composed of an ordered sequence of the following elements:
    - **name:** a simple element to specify the name of a data constraint.
    - **Request ServiceType:** a complex element to specify the request service of a data constraint. The schema of the service type is discussed in section 7.1.1.

- **Response ServiceType:** a complex element to specify the response service of a data constraint. The schema of the service type is discussed in section 7.1.1.
- **constraint:** a simple element to specify the constraints in a data constraint.
- **description:** a simple text element to store the annotation of the data constraint.

- **TimeConstraint:** a complex element to specify the time constraints in a reactivity of a contract. It is composed of an ordered sequence of the following elements:
  - **name:** a simple element to specify the name of the time constraint.
  - **Attribute:** a complex element to specify the attributes of a time constraint. It is an ordered sequence of the simple elements *name*, *datatype*, *value*, and *description.*
  - **constraint:** a simple element to specify the constraints in a time constraint.
  - **Request ServiceType:** a complex element to specify the request service of a time constraint. The schema of the service type is discussed in section 7.1.1.
  - **Response ServiceType:** a complex element to specify the response service of a time constraint. The schema of the service type is discussed in section 7.1.1.
  - **maxSafeTime:** a simple element to specify the maximum allowed time between receiving a request and providing a response.
  - **description:** a simple text element to store the annotation of the time constraint.
- **Reactivity:** a complex element to specify the reactivity property of a contract.
  - **name:** a simple element to specify the name of a reactivity.
  - **id:** a simple element to specify a unique identifier for each reactivity.

- **Request ServiceType:** a complex element to specify the request service of reactivity. The schema of the service type is discussed in section 7.1.1.
- **Response ServiceType:** a complex element to specify the response service of reactivity. The schema of the service type is discussed in section 7.1.1.
- **DataConstraint:** a complex element to include data constraints in reactivity. The schema of the data constraint has been discussed above.
- **TimeConstraint:** a complex element to include time constraints in reactivity. The schema of the time constraint has been discussed above.
- **Update:** a complex element to specify the updates of the data parameters. It is an ordered sequence of the simple elements *toBeUpdated* and *value.*
- **description:** a simple text element to store the annotation of the reactivity.

- **SafetyProperty:** a complex element to specify the safety property of a contract. It is composed of an ordered sequence of the following elements:
  - **name:** a simple element to specify the name of a safety property.
  - **ServiceType:** a complex element to specify the services that are restricted by the safety property. The schema of the service type is discussed in section 7.1.1.
  - **constraint:** a simple element to specify the constraints in a safety property.
  - **description:** a simple text element to store the annotation of the safety property.
- **description:** a simple text element to store the annotation of the contract.

7.1.5 PackageType Schema

The package type schema contains an ordered sequence of the following sub elements:

- **name:** a simple element to specify the name of a package.
- **Version:** a simple element to specify the version of a package.
- **InterfaceType:** a complex element to specify the interfaces in a package. The schema of the interface type is discussed in section 7.1.1.
- **ContractType:** a complex element to specify the contracts in a package. The schema of the contract type is discussed in section 7.1.4.
- **ConnectorType:** a complex element to specify the connectors in a package. The schema of the connector type is discussed in section 7.1.3.
- **ComponentType:** a complex element to specify the components in a package. The schema of the component type is discussed in section 7.1.2.
- **description:** a simple text element to store the annotation of the package.
- **PackageType:** a complex element to specify the sub packages in a package.

7.1.6 RBAC Schema

The Role-Based Access Control (RBAC) schema contains an ordered sequence of the following sub elements:

- **name:** a simple element to specify the name of a RBAC.
- **User/Group/Role/Privilege:** a complex element in a RBAC. The schema of Users/ Groups/Roles/Privileges is composed of an ordered sequence of the following:

- **name:** a simple element to specify the name of the users/groups/roles/privileges.
- **Attribute:** a complex element to specify the attributes of a user/group/role/privilege. It is an ordered sequence of the simple elements *name*, *datatype*, *value*, and *description*.
- **constraint:** a simple element to specify the constraints of a user/group/role/privilege.
- **description:** a simple element to store the annotation of a user/group/role/privilege.

- **UserGroupAssignments:** a complex element to specify the user-group assignments. It is composed of an ordered sequence of the complex elements *User and Group.*
- **UserRolesAssignments:** a complex element to specify the user-role assignments. It is composed of an ordered sequence of the complex elements *User and Role.*
- **GroupRolesAssignments:** a complex element to specify the group-role assignments. It is composed of an ordered sequence of the complex elements *Group and Role.*
- **ServiceType:** a complex element to include the services that is restricted by the RBAC. The schema of the service type is discussed in section 7.1.1.
- **ParameterType:** a complex element to include the parameters that is restricted by the RBAC. The schema of the parameter type is discussed in section 7.1.1.
- **PrivilegesForServices:** a complex element to assign service privileges to specific roles. It is composed of an ordered sequence of the composite elements *ServiceType, Privilege, and Role.*

- **PrivilegesForDataParameters:** a complex element to assign data parameter privileges to specific roles. It is composed of an ordered sequence of the composite elements *ParameterType, Privilege, and Role.*
- **description:** a simple text element to store the annotation of the RBAC.

7.1.7 System Schema

The system configuration schema contains an ordered sequence of the following sub elements:

- **name:** a simple element to specify the name of a system.
- **Attribute:** a complex element to specify the attributes of a system. It is an ordered sequence of the simple elements *name*, *datatype*, *value*, and *description.*
- **ComponentType:** a complex element to specify the components in a system. The schema of the component type is discussed in section 7.1.2.
- **Deploy:** a complex element to state the hardware component in which each software component is deployed. It is composed of an ordered sequence of the complex elements *HardwareComponentType and ComponentType*. The schema of ComponentType is discussed in section 7.1.2, and the schema of a hardware component type is composed of a sequence as follows:
  - **name:** a simple element to specify the name of the hardware component.
  - **Attribute:** a complex element to specify the attributes of a hardware component. It is an ordered sequence of the simple elements *name*, *datatype*, *value*, and *description.*

    - **constraints:** a simple element to specify the constraints of a hardware component.
    - **InterfaceType:** a complex element to specify the interfaces of a hardware component. The schema of an interface type is discussed in section 7.1.1.
    - **description:** a simple text element to store the annotation of the hardware component.
- **description:** a simple text element to store the annotation of the system.
- **RBAC:** a complex element to include a security mechanism in a system. The schema of the RBAC is discussed in section 7.1.6.

7.2 Transformation Rules

Component-based development is a particular software production method, tailored for developing reusable components and integrating existing components to create software systems. The primary reason for using component-based methodology in software development is that, it increases reuse potential. Components, their specifications, and other system artifacts can be used because they exist as independent architectural elements. Components are composed only on demand. On the other hand, ontologies has provided powerful improvements for creating and storing reusable knowledge building blocks in a well defined machine-readable format. Since reuse requires domain knowledge, and it is embedded in ontology, it seems reasonable to use ontology for deriving the reusable concepts and transforming them to reusable components. Motivated by this rationale, the OWL ontology obtained in Section 6 is now transformed into components, as described by TADL. We use XML to represent both OWL and TADL and explain the model transformation process from the OWL to TADL.

Ontology is a domain model that results in detailed specifications of reusable knowledge. When it is applied to component-based development, detailed specification of reusable components and component-based architectures are produced. In order to achieve an efficient component-based development, appropriate domain ontology must be built. The captured conceptualization and relations in ontology should be formally specified, so we have used OWL to formally represent the results of domain analysis. Besides, OWL provides the facility of sharing and reusing ontologies, as well as using the ontology reasoning to accomplish syntax consistency and subsumption checking. Subsequently, this enables mapping the OWL ontology formalization to formal TADL description.

The model transformation process from OWL ontology to TADL components implements the transformation rules in [48] by using XSLT [74, 47] model transformation framework and XPath language [74]. According to the transformation rules [48], mapping occurs between OWL language constructs and their relevant TADL constructs as follows:

- Entities are mapped to components [48]. The part-of relation between entities is mapped to composite components where a component consists of multiple constituent components. Note that the sub-class-of relation is not supported in the current version of TADL.
- Data are mapped to attributes [48]. An attribute is a data element that can be associated with any construct in TADL.
- Functional requirements are mapped to services [48]. For every functional requirement, a service is created in TADL. Also, two events are created for each service: a request for service and a response of the service. The has-property and request-property relations help identify which component is providing the service and which components are consuming it. A service is provided by the

component which is related to the functional requirement by the has-property relation. An interface is created for each component. The request and response events are associated with this interface. The services provided and consumed by the component are provided and requested at this interface. A connector is created for every request-property relation to provide a means to communicate requested and provided services. If two components are related by multiple service requests then it is sufficient to create one connector for the communication between the two components.

- Non-functional requirements are used to define the contract of each component [48]. The contract contains services, safety, security, reliability, availability, and any other non-functional requirements. A one to one mapping occurs between elements of these types of non-functional requirements. For example, a safety property is created in the component contract for every safety requirement in the ontology.
- Constraints are mapped into their corresponding synonym in TADL [48]. A constraint is an invariant on services.

The principal schema of one to one corresponding relations among the elements of OWL ontology and TADL constructs are shown in Figure 7.1.

Figure 7.2 illustrates [48] the steps of capturing domain components by application of domain engineering. Domain analysis, as the first step of domain engineering, aims to understand each system, its interactions with other systems, the constituent components, their functional and non-functional requirements, and the data and events stored and communicated between them. Domain analysis yields an ontology representing the knowledge base of the domain. Building ontologies is a major approach for capturing and representing reusable knowledge. The domain architecture can be deduced from the ontology. It includes the applications, their relations, and

trustworthiness. The domain architecture when applied to a special application is called the application architecture. The application architecture including the constituent domain concepts and their detail specifications are transformed into the TADL components, which is generated from the ontology. A component definition contains the details about data and trustworthiness aspects, as well as the functional, non-functional, and structural requirements. This knowledge and the resulting TADL specifications are stored in a repository and reused in system development processes.

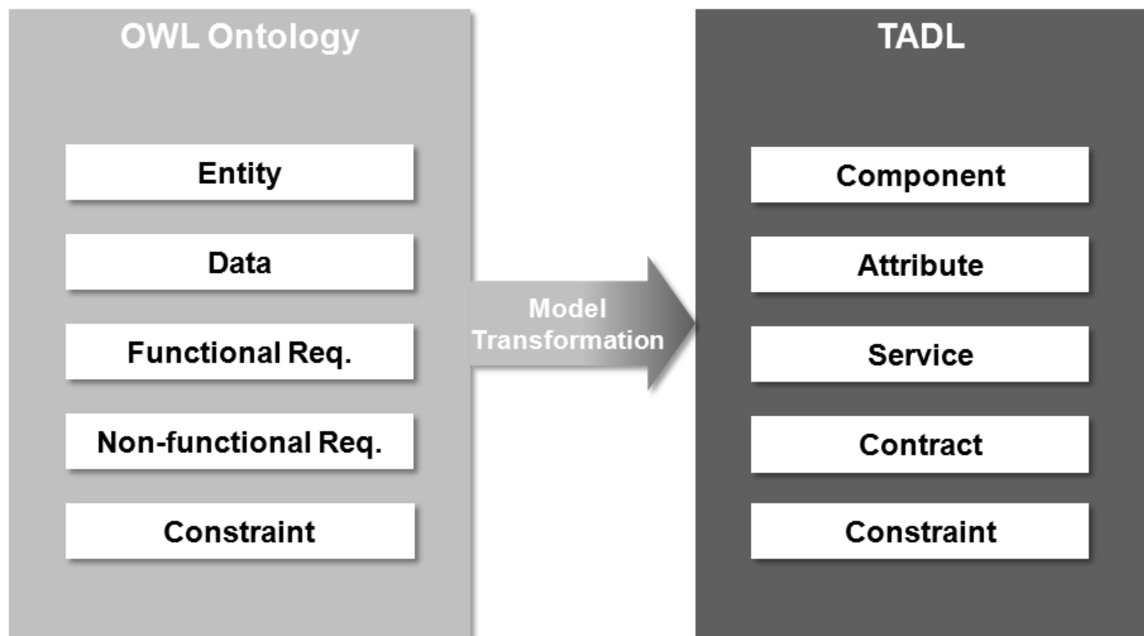

Figure 7.1: From OWL Ontology to TADL

During the component development, the component requirements are defined in TADL for new components or reused from the repository for existing domain components .To validate the formal component definitions an iterative process of validation is conducted to ensure that the system design is syntactically and semantically correct with respect to TADL correction rules. Then, the specification is analyzed and the component behavior is generated automatically as extended time-automata using the

transformation tool [40]. The output is an extended time automata which is compatible with the UPPAAL modeling language. Afterwards, verification is conducted using UPPAAL model checking techniques to verify the correctness of the design. An iterative process of verification occurs until the design successfully passes the validation checks on functional requirements, and safety, security, and timeliness properties. In case of errors or violation of any requirement, the component is redesigned using TADL specifications and the process starts over again. If the system design is correct, the selected components, which are retrieved from the repository, are integrated to develop the component-based system.

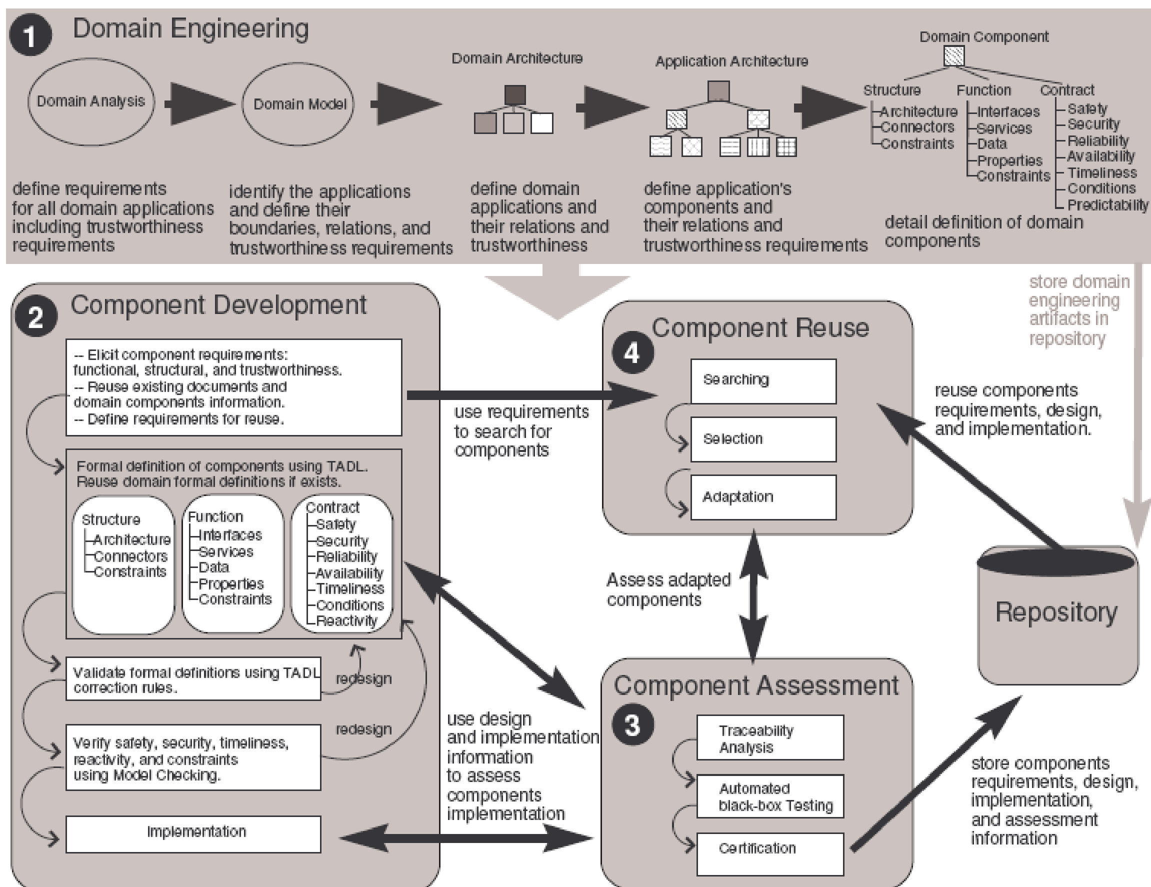

Figure 7.2: Domain Engineering and Component Development

7.3 Model Transformation from OWL Ontology to TADL

The model transformation process from OWL ontology to TADL is implemented by using XSLT model transformation framework and XPath language. The XSLT transformation process is performed by java programming, using the XSLT jar files. The input model of this transformation process is the OWL file derived from the transformation model in Chapter 6. The output model is the TADL XML schemas, which are described in Section 7.1. The transformation rules are described in Section 7.2.

The model transformation process consists of several steps, which are executed successively. The description of TADL XML schema composed in each step is as follows:

- **Step 1:** The new element <Configuration>, having the attribute *name* and the child elements<componentType> and <RBAC>, is added to the output XML file. The attribute *name* is set to the ontology name, which is taken from the name of the root class.
- **Step 2:** The new element <components>, having the attribute *name* and the child elements <attribute>, <interfaceTypes>, <architectureType>, and <contract>, is added to the output XML file. The attribute *name* is set to the ontology name, which is taken from the name of the root class. The child element <attribute> consists of the attributes *name* and *value* which are assigned to the ontological element <DATACONSTRAINTS>/<DC>.
- **Step 3:** The new element <interfaceTypes>, having the attribute *name* and the child element <serviceType>, is added to the output XML file. The attribute *name* is assigned to the ontological element <ConceptRelationTypes>/<CONCEPT>. The element <serviceType>, having the attributes *name* and *type*, are assigned to

the ontological element <FRTYPES>/<CONC>. The attribute *type* is defined according to the child elements <InternalFRs>, <ProvidedFRs>, or <RequestedFRs> of the element <FRTYPES>/<CONC>.

- **Step 4:** The element <architectureType>, having the attribute *name* and the child elements <componentType> and <connectorType>, is added to the output XML file. Composite components are implemented by the element <architectureType> of the ComponentType schema. The attribute *name* is set to the composite component name.
- **Step 5:** The element <architectureType>/<componentType>, is added to the output XML file. The element <componentType> are assigned to the ontological element <PARENTNODES>/<PN>.
- **Step 6:** The element <<connectorType>, having the attribute *name* and the child element <connectorRoleType>, is added to the output XML file. All constituent components of a composite component are connected by the connector types in which their interface types are defined. The attribute *name* is assigned to the ontological element <ConceptRelationTypes>/<CONCEPT>/<ConceptRelations>/<CR>.
- **Step 7:** The element <connectorType>/<connectorRoleType>, having the attribute *name* and the child element <interfaceTypes>, is added to the output XML file. The attribute *name* is assigned to the ontological element <ConceptRelations>/<CR>. Each <connectorType> consists of two <connectorRoleType> child elements. The second connector role type is defined as the inverse of the first one, it means, the attribute *type* of the <serviceType> elements that belongs to the interface types are exchanged.

- **Step 8:** The element <interfaceTypes> is added and its constituent child elements are assigned to the ontological elements as explained in Step 3, except that the attribute *type* of the element <serviceType> may be of type *input* or *output* (but not of type *internal*).
- **Step 9:** The element <contract>, having the attribute *name* and the child elements <dataConstraint>, <timeConstraint>, and <reactivity>, is added to the output XML file. The attribute *name* is assigned to the concept name.
- **Step 10:** The element <contract>/<dataConstraint>, having the attribute *name* and the child elements <service-request>, <service-response>, and <constraint>, is added. The attribute *name* is assigned to the concept name. The attributes *name* and *type* of the request and response services are assigned to the ontological element <FREQUIREMENTS>/<FR>. The element <constraint> is set to the data constraints and their values, which are assigned to the ontological element <DCS>/<DC>.
- **Step 11:** The element <contract>/<timeConstraint>, having the attribute *name* and the child elements <service-request>, <service-response>, and <maxSafeTime>, is added. The attribute *name* is assigned to the concept name. The attributes *name* and *type* of the request and response services are assigned to the ontological element <TIMECONSTRAINTS>/<TC>. The element <maxSafeTime> is assigned to the attribute *value* of the tag <TC>.
- **Step 12:** The element <contract>/<reactivity>, having the attribute *name* and the child elements <service-request>, <service-response>, <dataConstraint>, <timeConstraint>, <update>, and <action>, is added. The attribute *name* is assigned to the concept name. The attributes *name* and *type* of the request and

response services are assigned to the ontological element <FREQUIREMENTS>/<FR>.

- **Step 13:** The element <reactivity>/<dataConstraint> is added to the output XML file, if the reactivity has any data constraint. The element <dataConstraint> and its child elements are assigned to the ontological elements, as explained in Step 10.
- **Step 14:** The element <reactivity>/<timeConstraint> is added to the output XML file, if the reactivity has any time constraint. The element <timeConstraint> and its child elements are assigned to the ontological elements, as explained in Step 11.
- **Step 15:** The element <reactivity>/<update> is added to the output XML file, if the reactivity has any update property. The attributes *toBeUpdated* and *value* are assigned to the ontological element <UPDATES>/<UP>.
- **Step 16:** The element <reactivity>/<action> is added to the output XML file, if the reactivity has any reaction property. The attribute *name* is assigned to the ontological element <FRACTIONS>/<AC> and the attributes *from* and *to* are assigned to the element <FREQUIREMENTS>/<FR>.
- **Step 17:** The element <RBAC>, having the attribute *name* and the child elements <users>, <userRolesAssignments>, <privilegesForServices>, <serviceType> and <roles> is added to the output XML file. The attribute *name* is assigned to the name of the root class.
- **Step 18:** The child elements <users>, <roles> and <serviceType> of <RBAC>, having the attribute *name*, are added to the output XML file. The attribute *name* is assigned to the ontological element <ROLEPRIVILEGES>/<RP>.

- **Step 19:** The element <RBAC>/<userRolesAssignments>, having the child elements <users> and <roles>, is added to the output XML file.
- **Step 20:** The element <RBAC>/<privilegesForServices>, having the child elements <service>, <privilege>, and <role>, is added. The elements <service> and <role> are assigned to the ontological elements, as explained in Step 18. The attribute *name* of the element <privilege> is set to *true* or *false*, according to the content of the ontological element <ROLEPRIVILEGES>/<RP>.

The obtained TADL file complies with the XML schemas that are explained in Section 7.1. After the transformation process, the *MergedConcepts* TADL file is opened in the Microsoft Visual Studio software tool, which is depicted in Figure 7.3.

Figure 7.4 and Figure 7.5 show two parts of the *MergedConcepts* TADL XML file, which are automatically generated through the model transformation process. In Figure 7.4, one of the reactivity properties of the component *Concept1* is depicted. The connector type *Concept1_Concept2* and the two relevant connector role types are shown in Figure 7.5.

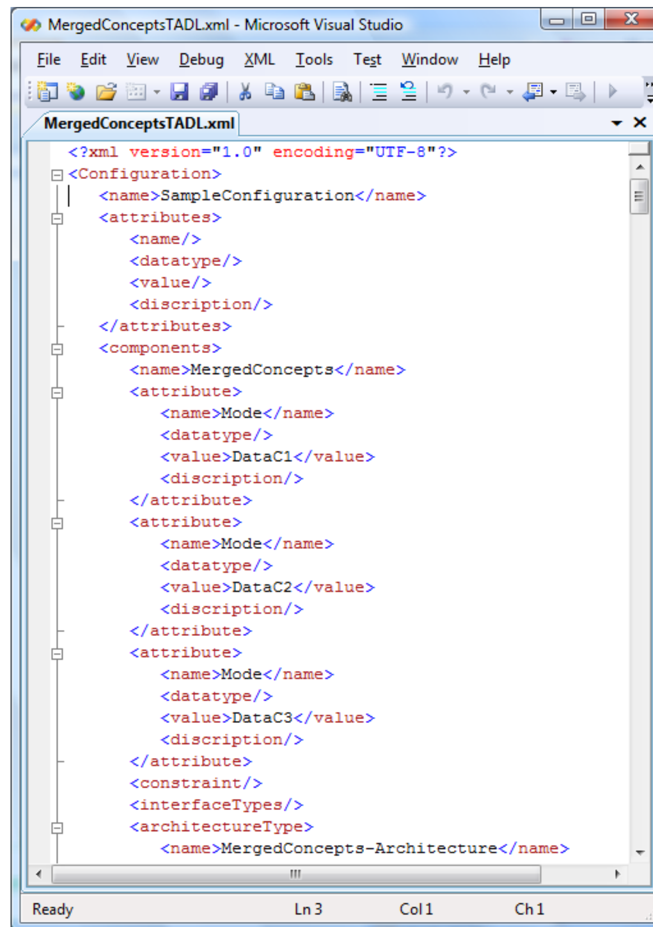

Figure 7.3: *MergedConcepts* TADL file in Visual Studio

```
<reactivity>
    <name>Concept1-2</name>
    <id/>
    <service-request>
        <name>Req3</name>
        <id/><type>input</type>
        <constraint/>
    </service-request>
    <service-response>
        <name>Req4</name>
        <id/><type>output</type>
        <constraint/>
    </service-response>
    <dataConstraint>
        <name>Concept1DataConstraint2</name>
        <service-request>
            <name>Req3</name>
            <id/><type>input</type>
            <constraint/>
        </service-request>
        <service-response>
            <name>Req4</name>
            <id/><type>output</type>
            <constraint/>
        </service-response>
        <constraint>Mode==DataC2</constraint>
        <descreption/>
    </dataConstraint>
    <timeConstraint>
        <name>Concept1TimeConstraint1</name>
        <constraint/>
        <service-request>
            <name>Req3</name>
            <id/><type>input</type>
            <constraint/>
        </service-request>
        <service-response>
            <name>Req4</name>
            <id/><type>output</type>
            <constraint/>
        </service-response>
        <maxSafeTime>5</maxSafeTime>
    </timeConstraint>
    <update>
        <toBeUpdated>Mode</toBeUpdated>
        <value>DataC3</value>
    </update>
    <action>
        <name>Req5</name>
        <id/><type/><description/>
        <from>Req4</from>
        <FromId/><to>idel</to>
    </action>
</reactivity>
```

Figure 7.4: *MergedConcepts* TADL XML file (part1)

```xml
<connectorType>
    <name>ConnectorTypeConcept1_Concept2</name>
    <connectorRoleType>
        <name>ConnectorRoleType1Concept1_Concept2</name>
        <constraint/>
        <interfaceTypes>
            <name>Concept1_Concept2</name>
            <protocol/>
            <serviceType>
                <name>Req3</name>
                <id/><type>input</type>
                <constraint/>
            </serviceType>
            <serviceType>
                <name>Req2</name>
                <id/><type>output</type>
                <constraint/>
            </serviceType>
            <serviceType>
                <name>Req5</name>
                <id/><type>output</type>
                <constraint/>
            </serviceType>
        </interfaceTypes>
    </connectorRoleType>
    <connectorRoleType>
        <name>ConnectorRoleType2Concept1_Concept2</name>
        <constraint/>
        <interfaceTypes>
            <name>Concept2_Concept1</name>
            <protocol/>
            <serviceType>
                <name>Req3</name>
                <id/><type>output</type>
                <constraint/>
            </serviceType>
            <serviceType>
                <name>Req2</name>
                <id/><type>input</type>
                <constraint/>
            </serviceType>
            <serviceType>
                <name>Req5</name>
                <id/><type>input</type>
                <constraint/>
            </serviceType>
        </interfaceTypes>
    </connectorRoleType>
    <constraint/>
    <descreption/>
</connectorType>
```

Figure 7.5: *MergedConcepts* TADL XML file (part2)

Chapter 8

Case Study and Evaluation

The Common Component Modeling Example (CoCoME) case study explained in Chapter 3 is used to present the methodology introduced in this thesis. One of the contributions of our approach is the application of Formal Concept Analysis (FCA) in domain analysis to develop OWL ontology as a domain model, and then transform automatically the derived ontology into the TADL description of the target component-based system. Therefore, the introduced techniques and tools are applied to the CoCoME case study, to demonstrate that the methodology described in this thesis can be generalized to common component-based systems. Afterwards, we are going to evaluate our approach by discussing the obtained results and comparing them with what has been done in previous works such as the VMT tool [89] and the Transformation tool [40].

8.1 Case Study Implementation

Common Component Modeling Example [36], which is a benchmark case study for testing the modeling ability of component-based systems, is provided to explain our approach. We have tested our methodologies on CoCoME case study to illustrate and verify the process.

One of the composite components of the *CoCoME* case study, explained in Chapter 3, is the *Store System*. The *Store System* contains some other components such as *CashBox*, *Cashier*, and *Inventory* which are implemented to illustrate the methodology presented in this thesis. The implementation process consists of 3 steps. These are (1) formal context table definition in FCA and concept lattice derivation, (2) transformation model from concept lattice to OWL ontology, and (3) transformation model from OWL ontology to TADL description.

8.1.1 Context Table Definition and Concept Lattice Derivation

Before going into the details of each concept and its corresponding context table, it is important to introduce the global-level variables, which are defined as the data parameters to make constraints on the functional requirement properties. These variables with their possible values are described in Table 8.1.

At first, formal context tables that contain the concepts including objects and their attributes are defined. Then, the partially defined context tables are combined into a unified table that respects entirely the relational information on objects and attributes occurring in the use cases. The many-valued context tables are defined and converted into the binary context tables by using the conceptual scaling method. The lattice miner

software tool is used to define the Valued Context Tables (VCTs), which are converted into the Binary Context Tables (BCTs).

Table 8.1: Global variables in *CoCoME* case study

	Variable	**Values**	**Description**
1	Mode	{Disable, Done, InSale, Ready, Waiting}	Used to control the mode of the Cash Box; Disable: CashBox is in disabling process of the express mode, Done: Sale is done, InSale: Sale is in process, Ready: CashBox is ready, Waiting: CashBox is waiting.
2	IsExpress	{Express, Normal}	Used to represent the operation of the Cash Box, which can either be in express or in normal mode.
3	IsMore	{1, 2}	Used to represesnt if more items remain to perchase; 1: No more item, 2: More item.
4	Authorization	{1, 2}	Used to represent the authorization results of the card payment; 1:Approved, 2:Declined.
5	PaymentMethod	{1, 2}	Used to represent the method of payment; 1: Cash, 2: Card.

CashBox

First, a partial context table is defined for the concept *CashBox*, which performs the sale process initiated by the cashier, and holds the received cash. The concept *CashBox* has the following functional requirements:

- The functional requirements provided by the concept *CashBox* are *PassItem, BarCode, Info, SaleFinished, Cash, Card, Approved, Declined, YesExpress, NotExpress, DisableExpress.*
- The functional requirements requested by the concept *CashBox* are *Scan, GetInfo, Pay, ReadCard, Print, CheckLastHour, TurnLightOn, AddToInventory, IsMoreItem, TurnLightOff.*
- The functional requirements internally accomplished within the concept *CashBox* are *AddTotal, ReturnChange, CheckIfExpress, Ignore, ChangeModeToNormal.*

The functional requirement properties of the concept *CashBox* are shown in Table 8.2. The functional requirement properties consist of the following main elements:

Provided functional requirement, Requested functional requirement, Data constraint (represents the pre-conditions), Time constraint, Update (represents the post-conditions), and Action (represents the triggered functional requirements) (See Table 8.2).

Table 8.2: *CashBox* functional requirement properties

	Provided Functional Requirement	Requested Functional Requirement	Data Constraint	Time Constraint	Updates	Actions
1	PassItem	Scan	Mode==InSale IsMore==2			
2	Barcode	GetInfo	Mode==InSale			AddToInventory Print
3	Info	AddTotal	Mode==InSale	1.0S		IsMoreItem
4	SaleFinished	Pay	Mode==InSale IsMore==1	1.0S		
5	Cash	ReturnChange	Mode==InSale IsMore==1 PaymentMethod==1	120.0S	Mode:=Done	AddToInventory Print
6	Card	ReadCard	Mode==InSale PaymentMethod==2 IsExpress==Normal	1.0S		
7	Approved	Print	Mode==InSale IsMore==1 PaymentMethod==2 IsExpress==Normal Authorization==1		Mode:=Done	AddToInventory
8	Declined	Pay	Mode==InSale IsMore==1 PaymentMethod==2 IsExpress==Normal Authorization==2			
9	CheckIfExpress	CheckLastHour	Mode==Done		Mode:=Waiting	
10	YesExpress	TurnLightOn	Mode==Waiting	1.0S	Mode:=Ready IsExpress:=Express	
11	NotExpress	Ignore	Mode==Waiting		Mode:=Ready IsExpress:=Normal	TurnLightOff
12	DisableExpress	ChangeModeTo Normal	Mode==Disable IsExpress==Express	1.0S	IsExpress:=Normal	TurnLightOff

The functional requirement properties of the concept *CashBox* are defined as the attribute variables in the *CashBox* context table, depicted in Table 8.3.

Table 8.3: Attribute variables of *CashBox* context table

Attribute Variable	Type	Values
CashBox	Main Attribute	
DCMode	DC Attribute	{Done, Disable, InSale, Waiting}
DCIsExpress	DC Attribute	{Express, Normal}
DCPaymentMethod	DC Attribute	{1, 2}
DCIsMore	DC Attribute	{1, 2}
DCAuthorization	DC Attribute	{1, 2}
IFRCashBox-CheckIfExpress_FRCheckLastHour	Functional Requirements Attribute	
FRCashBox-DisableExpress_IFRChangeModeToNormal	Functional Requirements Attribute	
FRCashBox-YesExpress_FRTurnLightOn	Functional Requirements Attribute	
FRCashBox-NotExpress_IFRIgnore	Functional Requirements Attribute	
FRCashBox-Info_IFRAddTotal	Functional Requirements Attribute	
FRCashBox-Cash_IFRReturnChange	Functional Requirements Attribute	
FRCashBox-SaleFinished_FRPay	Functional Requirements Attribute	
FRCashBox-Card_FRReadCard	Functional Requirements Attribute	
FRCashBox-PassItem_FRScan	Functional Requirements Attribute	
FRCashBox-Barcode_FRGetInfo	Functional Requirements Attribute	
FRCashBox-Approved_FRPrint	Functional Requirements Attribute	
FRCashBox-Declined_FRPay	Functional Requirements Attribute	
UpdateDCMode	Update Attribute	{Done, Ready, Waiting}
UpdateDCIsExpress	Update Attribute	{Express, Normal}
Action	Action Attribute	{AddToInventory, Print, TurnLightOff, IsMoreItem}
TCFRCashBox-DisableExpress_IFRChangeModeToNormal	TC of Functional Requirements Attribute	T1.0S
TCFRCashBox-YesExpress_FRTurnLightOn	TC of Functional Requirements Attribute	T1.0S
TCFRCashBox-Info_IFRAddTotal	TC of Functional Requirements Attribute	T1.0S
TCFRCashBox-Cash_IFRReturnChange	TC of Functional Requirements Attribute	T120.0S
TCFRCashBox-SaleFinished_FRPay	TC of Functional Requirements Attribute	T1.0S
TCFRCashBox-Card_FRReadCard	TC of Functional Requirements Attribute	T1.0S

The attribute variables of the concept *CashBox* are determined in the valued context table shown in Figure 8.1. Then, it is converted into the corresponding BCT, which is shown in Figure 8.2.

Lattice Miner

File Edit Lattice Rules Context Window About

Context : CashBox

	CashBox	DCMode	DCIsExpress	DCIsMore	FRCashBox-DisableExpress_IFRChangeModeToNormal	UpdateDCMode	UpdateDCIsExpress	Action
CashBox1		Done	/	/	/	Waiting	/	/
CashBox2		Disable	Express	/		/	Normal	TurnLightOff
CashBox3		Waiting	/	/	/	Ready	Express	/
CashBox4		Waiting	/	/	/	Ready	Normal	TurnLightOff
CashBox5		InSale	/	/	/	/	/	IsMoreItem
CashBox6		InSale	/	1	/	Done	/	AddToInventory-Print
CashBox7		InSale	/	1	/	/	/	/
CashBox8		InSale	Normal	/	/	/	/	/
CashBox9		InSale	/	2	/	/	/	/
CashBox10		InSale	/	/	/	/	/	AddToInventory-Print
CashBox11		InSale	Normal	1	/	Done	/	AddToInventory
CashBox12		InSale	Normal	1	/	/	/	/

Figure 8.1: *CashBox* Valued Context Table (VCT)

Lattice Miner

File Edit Lattice Rules Context Window About

Context : CashBox

	CashBox_	DCMode_Disable	DCMode_Done	DCMode_InSale	DCMode_Waiting	DCIsExpress_Express	DCIsExpress_Normal	FRCashBox-DisableExpress_IFRChangeModeToNormal_
CashBox1	X		X					
CashBox2	X	X				X		X
CashBox3	X				X			
CashBox4	X				X			
CashBox5	X			X				
CashBox6	X			X				
CashBox7	X			X				
CashBox8	X			X			X	
CashBox9	X			X				
CashBox10	X			X				
CashBox11	X			X			X	
CashBox12	X			X			X	

Figure 8.2: *CashBox* Binary Context Table (BCT)

Cashier

The other partially defined context table is for concept *Cashier*, who operates the cash box, begins/ends the sale process, scans the products, handles the payment, and

manages the cash box mode. The concept *Cashier* has the following functional requirements:

- The functional requirements provided by the concept *Cashier* are *IsMoreItem, Pay*.
- The functional requirements requested by the concept *Cashier* are *DisableExpress, PassItem, SaleFinished, Cash, Card*.
- The functional requirements internally accomplished within the concept *Cashier* are *CancelExpress, StartSale*.

Some functional requirement properties of concept *Cashier* are shown in Table 8.4. The functional requirement properties consist of the following main elements: Provided functional requirement, Requested functional requirement, Data constraint (represents the pre-conditions), Time constraint, Update (represents the post-conditions), and Action (represents the triggered functional requirements) (See Table 8.4).

Table 8.4: *Cashier* functional requirement properties

	Provided Functional Requirement	Requested Functional Requirement	Data Constraint	Time Constraint	Updates	Actions
1	CancelExpress	DisableExpress	Mode==Ready IsExpress==Express		Mode:=Disable	
2	StartSale	PassItem	Mode==Ready	1.0S	Mode:=InSale	
3	IsMoreItem	PassItem	Mode==InSale IsMore==2			
4	IsMoreItem	SaleFinished	Mode==InSale IsMore==1			
5	Pay	Cash	Mode==InSale IsMore==1 PaymentMethod==1			
6	Pay	Card	Mode==InSale IsMore==1 PaymentMethod==2 IsExpress==Normal			

The functional requirement properties of the concept *Cashier* are defined as the attribute variables in the *Cashier* context table, which are depicted in Table 8.5.

Table 8.5: Attribute variables of *Cashier* context table

Attribute Variable	Type	Values
Cashier	Main Attribute	
DCMode	DC Attribute	{Ready, InSale}
DCIsExpress	DC Attribute	{Express, Normal}
DCPaymentMethod	DC Attribute	{1, 2}
DCIsMore	DC Attribute	{1, 2}
IFRCashier-CancelExpress_FRDisableExpress	Functional Requirements Attribute	
FRCashier-Pay_FRCard	Functional Requirements Attribute	
FRCashier-Pay_FRCash	Functional Requirements Attribute	
FRCashier-IsMoreItem_FRSaleFinished	Functional Requirements Attribute	
FRCashier-IsMoreItem_FRPassItem	Functional Requirements Attribute	
IFRCashier-StartSale_FRPassItem	Functional Requirements Attribute	
UpdateDCMode	Update Attribute	{Disable, InSale}
TCIFRCashier-StartSale_FRPassItem	TC of Functional Requirements Attribute	T1.0S

The attribute variables of the concept *Cashier* are determined in the valued context table shown in Figure 8.3. Then, it is converted into the corresponding BCT, which is shown in Figure 8.4.

Lattice Miner

File Edit Lattice Rules Context Window About

Context : Cashier

Context : CashBox | Context : Cashier

	Cashier	DCMode	DCIsExpress	DCPaymentMethod	DCIsMore	IFRCashier-StartSale_FRPassItem	UpdateDCMode	TCIFRCashier-StartSale_FRPassItem=T1.0S
Cashier1		Ready	Express	/	/	/	Disable	/
Cashier2		InSale	Normal	2	1	/	/	/
Cashier3		InSale	/	1	1	/	/	/
Cashier4		InSale	/	/	1	/	/	/
Cashier5		InSale	/	/	2	/	/	/
Cashier6		Ready	/	/	/		InSale	

Figure 8.3: *Cashier* Valued Context Table (VCT)

CashDesk

Now, it is the time to merge the two defined context tables *CashBox* and *Cashier,* and build the *CashDesk* context table. The context table *CashDesk* is constructed according to

the integration rules defined in Chapter 4. First, a nested context table (NCT) is defined to combine the *CashBox* and *Cashier*, which is depicted in Figure 8.5. Then the defined NCT is converted to the corresponding BCT and pruned by merging the duplicate attributes, as shown in Figure 8.6.

A main attribute with the name *CashDesk* is added to the combined BCT.

Lattice Miner

File Edit Lattice Rules Context Window About

Context : Cashier

Context : Cashier

	Cashier_	DCMode_InSale	DCMode_Ready	DCIsExpress_Express	DCIsExpress_Normal	DCPaymentMethod_1	DCPaymentMethod_2	IFRCashier-StartSale_FRPassItem_
Cashier1	X		X	X				
Cashier2	X	X			X		X	
Cashier3	X	X				X		
Cashier4	X	X						
Cashier5	X	X						
Cashier6	X		X					X

Figure 8.4: *Cashier* Binary Context Table (BCT)

Lattice Miner

File Edit Lattice Rules Context Window About

Context : CashDesk

Context : CashDesk

							Level2						
Attributes	TCFRCash...	FRCashBo...	FRCashBo...	FRCashBo...	FRCashBo...	CashBox_	DCMode_I...	DCMode_R...	DCIsExpre...	DCIsExpre...	DCPaymen...	DCPaymen...	DCIsMore_1
CashBox1						X							
CashBox2						X							
CashBox3						X							
CashBox4						X							
CashBox5						X							
CashBox6						X							
CashBox7						X							
CashBox8	X					X							
CashBox9		X				X							
CashBox10			X			X							
CashBox11				X		X							
CashBox12					X	X							
Cashier1								X	X				
Cashier2							X			X		X	X
Cashier3							X				X		X
Cashier4							X						X
Cashier5							X						
Cashier6													

Figure 8.5: *CashDesk* Nested Context Table (NCT)

	CashDesk	DCIsMore_1	DCIsMore_2	DCMode_InSale	DCMode_Waiting	FRCashBox-Card_FRReadCard_	TCFRCashBox-Card_FRReadCard=T1.0S_	PropertyCashDesk=CashDeskId
CashBox1	X							X
CashBox2	X							X
CashBox3	X				X			X
CashBox4	X				X			X
CashBox5	X			X				X
CashBox6	X	X		X				X
CashBox7	X	X		X				X
CashBox8	X			X		X	X	X
CashBox9	X		X	X				X
CashBox10	X			X				X
CashBox11	X	X		X				X
CashBox12	X	X		X				X
Cashier1	X							X
Cashier2	X	X		X				X
Cashier3	X	X		X				X
Cashier4	X	X		X				X
Cashier5	X		X	X				X
Cashier6	X							X

Figure 8.6: Merged and pruned *CashDesk* Binary Context Table (BCT)

The duplicate attributes of the context tables *CashBox* and *Cashier* are as following: *DCMode_InSale, DCIsExpress_Express, DCIsExpress_Normal, DCPaymentMethod_1, DCPaymentMethod_2, DCIsMore_1, DCIsMore_2.* According to the conditions and integration rules discussed in Chapter 4 the partial context tables are merged and pruned. Rule 1 is applied for the above duplicate attributes. So, one of the duplicate attributes is eliminated, and the extent objects in relation with the removed attribute are denoted in the crossing cells of the remaining attribute.

Table 8.6: Newly added attribute variables of *CashDesk* context table

Attribute Variable	Type	Values
CashDesk	Main Attribute	
PropertyCashDesk=CashDeskId	Property of Attribute	
PropertyCashDesk=Sale	Property of Attribute	
PropertyCashDesk=CashDeskPC	Property of Attribute	
PropertyCashDesk=InStore	Property of Attribute	

Besides, the attribute variables in *CashBox* and *Cashier* are merged and transferred into *CashDesk* context table. Some attribute variables identifying the property attributes of the main attribute *CashDesk* are added to the *CashDesk* context table. The newly added attribute variables of *CashDesk* context table are shown in Table 8.6.

Inventory

The other partially defined context table is the concept *Inventory*, which represents the store server inventory. All information about the store such as the completed sale process is registered in the inventory. The concept *Inventory* has the following functional requirements:

- The functional requirements provided by the concept *Inventory* are *GetInfo, CheckLastHour, AddToInventory*.
- The functional requirements requested by the concept *Inventory* are *Info, YesExpress, NotExpress.*
- The functional requirement internally accomplished within the concept *Inventory* is *InfoAdded.*

Table 8.7: *Inventory* functional requirement properties

	Provided Functional Requirement	Requested Functional Requirement	Data Constraint	Time Constraint	Updates	Actions
1	GetInfo	Info	Mode==InSale			
2	CheckLastHour	YesExpress	Mode==Waiting			
3	CheckLastHour	NotExpress	Mode== Waiting			
4	AddToInventory	InfoAdded	Mode==InSale	2.0S		

Some functional requirement properties of the concept *Inventory* are shown in Table 8.7. The functional requirement properties consist of the following main elements:

Provided functional requirement, Requested functional requirement, Data constraint (represents the pre-conditions), Time constraint, Update (represents the post-conditions), and Action (represents the triggered functional requirements) (See Table 8.7).

The functional requirement properties of the concept *Inventory* are defined as the attribute variables in the *Inventory* context table, which are depicted in Table 8.8.

Table 8.8: Attribute variables of *Inventory* context table

Attribute Variable	Type	Values
Inventory	Main Attribute	
DCMode	DC Attribute	{Waiting, InSale}
FRInventory-CheckLastHour_FRYesExpress	Functional Requirements Attribute	
FRInventory-GetInfo_FRInfo	Functional Requirements Attribute	
FRInventory-CheckLastHour_FRNotExpress	Functional Requirements Attribute	
FRInventory-AddToInventory_IFRInfoAdded	Functional Requirements Attribute	
TCFRInventory-AddToInventory_IFRInfoAdded	TC of Functional Requirements Attribute	T2.0S

The attribute variables of the concept *Inventory* are determined in the valued context table shown in Figure 8.7. Then, it is converted into the corresponding BCT, which is shown in Figure 8.8.

Figure 8.7: *Inventory* Valued Context Table (VCT)

	Inventory_	DCMode_InSale	DCMode_Waiting	FRInventory-AddToInventory_IFRInfoAdded_	TCFRInventory-AddToInventory_IFRInfoAdded=T2.0S_
Inventory1	X		X		
Inventory2	X	X			
Inventory3	X		X		
Inventory4	X	X		X	X

Figure 8.8: *Inventory* Binary Context Table (BCT)

CoCoME

When all the context tables are constructed, they can be combined according to the integration rules explained in Chapter 4. *CoCoME*, the unified context table, is derived from the combination of *CashDesk* and *Inventory* context tables. First, a nested context table (NCT) is defined to combine the mentioned context tables, and then the defined NCT is converted to the corresponding BCT and pruned by merging the duplicate attributes. A main attribute with the name *CoCoME* is added to the combined BCT.

The duplicate attributes of the context tables *CashDesk* and *Inventory* are *DCMode_InSale*, and *DCMode_Waiting*.

According to the conditions and integration rules discussed in Chapter 4, the partially defined context tables are merged and pruned. Rule 1 is applied for the above duplicate attributes. So, one of the duplicate attributes is eliminated, and the extent objects in relation with the removed attribute are denoted in the crossing cells of the remaining attribute. The obtained unified context table is depicted in Figure 8.9.

Besides the attribute variables in *CashDesk* and *Inventory* that are merged and transferred into *CoCoME* context table, some attribute variables identifying the security properties are added to the *CoCoME* context table. The security property is one of the essential credentials of trustworthiness during the design stage. The security properties

are role privilege attributes that specify which role has or does not have the privilege of providing which functional requirement. The newly added attribute variables of *CoCoME* context table are shown in Table 8.9.

Lattice Miner

File Edit Lattice Rules Context Window About

Context : CoCoME

	CoCoME	Inventory_	CashBox_	Cashier_	CashDesk	PropertyCashDesk=CashDeskId	RolePrivilege-Cashier_Pay	RolePrivilegeNot-Manager_IsMoreItem	FRCashBox-Info_IFRAddTotal_
CashBox1	X		X		X	X			
CashBox2	X		X		X	X			
CashBox3	X		X		X	X			
CashBox4	X		X		X	X			
CashBox5	X		X		X	X			X
CashBox6	X		X		X	X			
CashBox7	X		X		X	X	X		
CashBox8	X		X		X	X			
CashBox9	X		X		X	X			
CashBox10	X		X		X	X			
CashBox11	X		X		X	X			
CashBox12	X		X		X	X	X		
Cashier1	X			X	X	X			
Cashier2	X			X	X	X	X		
Cashier3	X			X	X	X	X		
Cashier4	X			X	X	X		X	
Cashier5	X			X	X	X		X	
Cashier6	X			X	X	X			
Inventory1	X	X							
Inventory2	X	X							
Inventory3	X	X							
Inventory4	X	X							

Figure 8.9: *CoCoME* Binary Context Table (BCT)

Table 8.9: Newly added attribute variables of *CoCoME* context table

Attribute Variable	Type	Values
CoCoME	Main Attribute	
RolePrivilege-Cashier_CancelExpress	Role Privilege	
RolePrivilege-Cashier_IsMoreItem	Role Privilege	
RolePrivilege-Cashier_Pay	Role Privilege	
RolePrivilege-Cashier_StartSale	Role Privilege	
RolePrivilegeNot-StockManager_CancelExpress	Negative Role Privilege	
RolePrivilegeNot-StockManager_IsMoreItem	Negative Role Privilege	
RolePrivilegeNot-Manager_IsMoreItem	Negative Role Privilege	

So far, the unified context table *CoCoME* is constructed which contains the formal concepts, captured from the system requirements specifications. In this step, the

concept lattice corresponding to the derived formal concept hierarchy is generated by the Lattice Miner [59, 13] software tool.

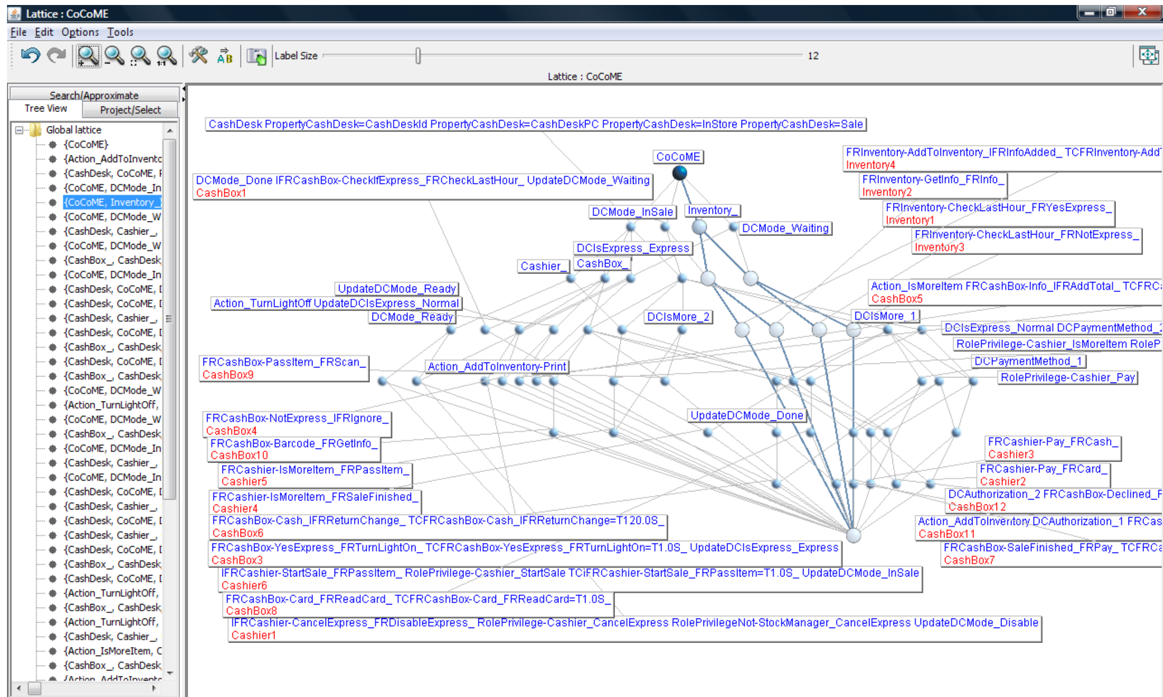

Figure 8.10: *CoCoME* Concept Lattice

There is a facility of drawing the concept lattice diagram from the given formal context table that is provided by almost FCA software tools. The lattice represents the concept hierarchy. The main attribute *CoCoME* of the unified formal context table is presented as the intent of the supremum node in the derived concept lattice. The intents and extents are represented in the boxes beside the nodes. The reduced labeling concept lattice of the context table *CoCoME* is shown in Figure 8.10.

Afterwards, the concept lattice *CoCoME* is saved as an XML file by Lattice Miner. Although Lattice Miner can show the reduced labeling concept lattice, it cannot be saved as a reduced labeling XML file. The reducing process is done as the first step of the model transformation process described in Chapter 5. The acquired XML-format file of

the concept lattice *CoCoME* is opened in Microsoft Visual Studio and depicted in Figure 8.11.

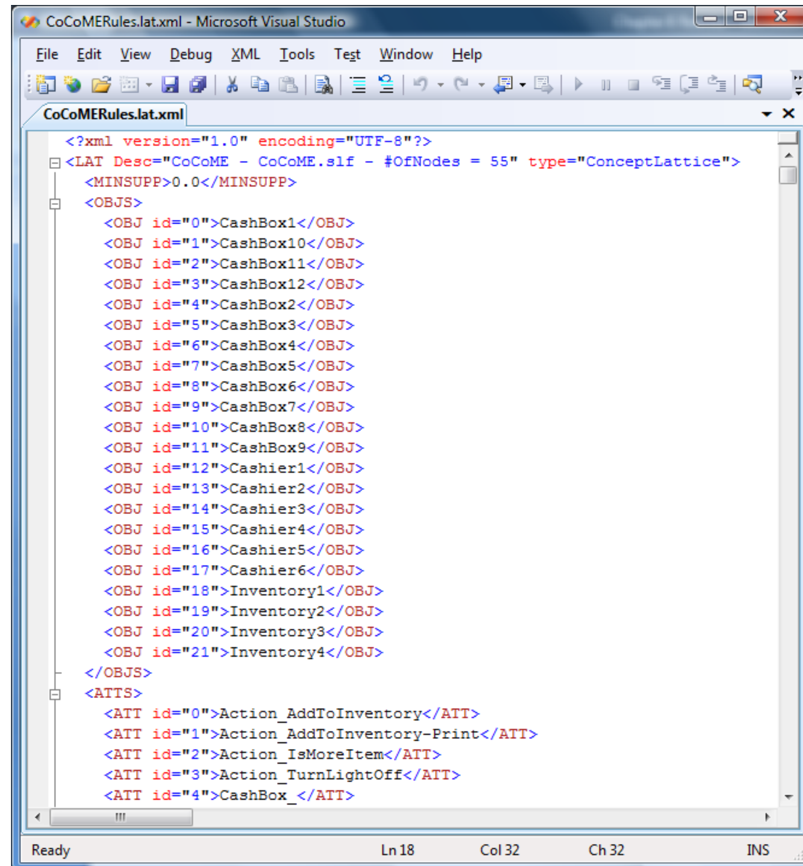

Figure 8.11: XML-format *CoCoME* Concept Lattice

Also, the implication rules are derived from the concept lattice *CoCoME* and exported in another XML file, which is depicted in Figure 8.12. The value of the minimum support has been assigned to one percent and the minimum confidence is assigned to hundred percent. Finally, the XML file of concept lattice and the XML file of the implication rules are merged. The merged XML-format file is conserved to be transformed into the OWL-format ontology at the next step of our methodology, which is described in the following section.

RulesCoCoME.xml - Microsoft Visual Studio

File Edit View Debug XML Tools Test Window Help

RulesCoCoME.xml


```xml
<?xml version="1.0" encoding="UTF-8"?>
<rules_base>
  <specs>
    <context_name>CoCoME</context_name>
    <minimal_support>0.01</minimal_support>
    <minimal_confidence>1.0</minimal_confidence>
  </specs>
  <rules>
    <rules_number>126</rules_number>
    <rule>
      <premise>{CashDesk}</premise>
      <consequence>{CoCoME, PropertyCashDesk=CashDeskId, PropertyCashDesk=CashDe
      <support>0.81</support>
      <confidence>1.0</confidence>
    </rule>
    <rule>
      <premise>{PropertyCashDesk=CashDeskId}</premise>
      <consequence>{CashDesk, CoCoME, PropertyCashDesk=CashDeskPC, PropertyCashDe
      <support>0.81</support>
      <confidence>1.0</confidence>
    </rule>
    <rule>
      <premise>{PropertyCashDesk=CashDeskPC}</premise>
      <consequence>{CashDesk, CoCoME, PropertyCashDesk=CashDeskId, PropertyCashDe
      <support>0.81</support>
      <confidence>1.0</confidence>
    </rule>
    <rule>
      <premise>{PropertyCashDesk=InStore}</premise>
      <consequence>{CashDesk, CoCoME, PropertyCashDesk=CashDeskId, PropertyCashDe
      <support>0.81</support>
      <confidence>1.0</confidence>
    </rule>
```


Ready Ln 1 Col 1 Ch 1 INS

Figure 8.12: XML-format implication rules of *CoCoME* Concept Lattice

8.1.2 Transformation from Concept Lattice to OWL Ontology

In this step, the concept lattice *CoCoME* is automatically transformed into the OWL-format ontology by executing the transformation rules defined in Chapter 5. Basically, the formal concepts in FCA are going to be transformed into the classes in ontology, and the relations between the formal concepts in FCA are going to be transformed into the relations among the ontological classes. After the transformation process, the constructed OWL ontology may be opened in TopBraid Composer [80] software tool, which is shown in Figure 8.13.

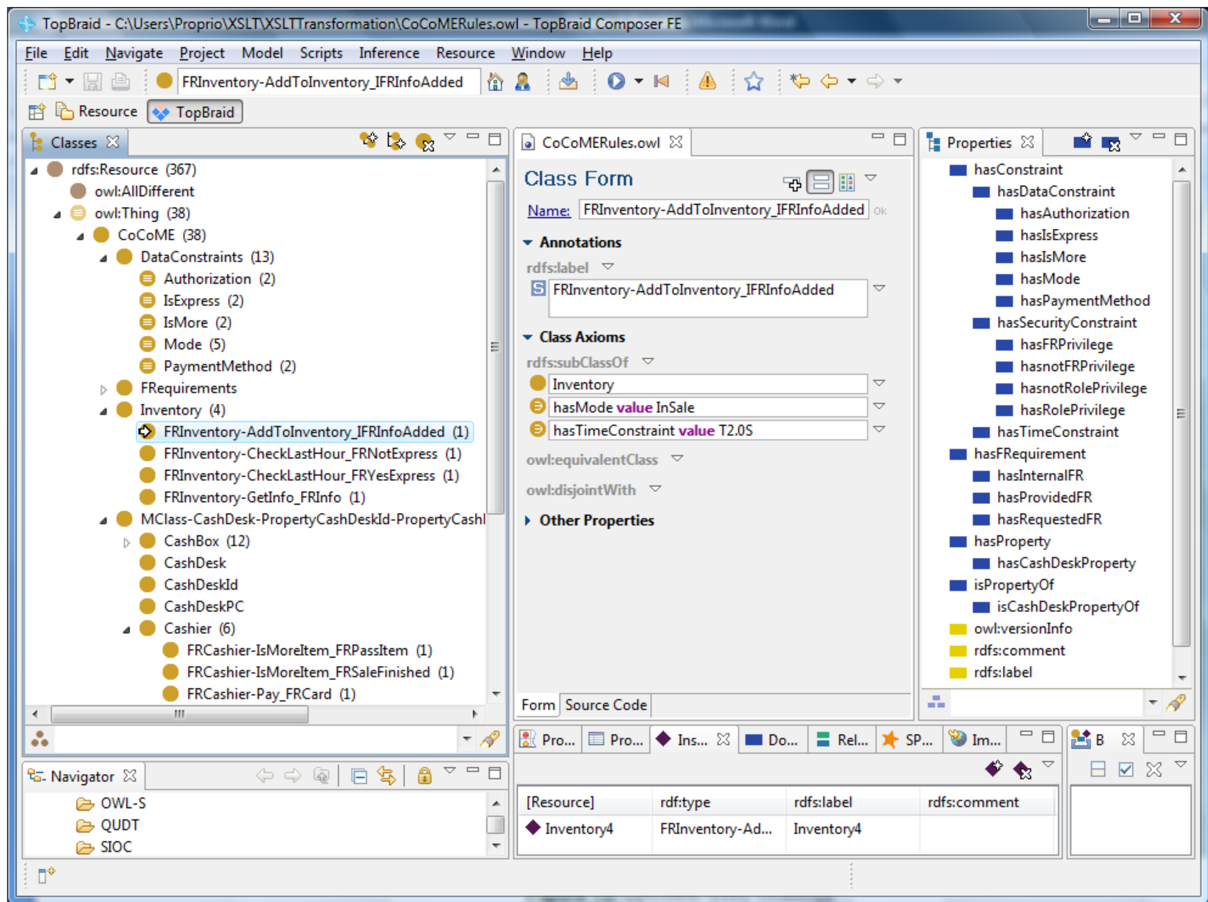

Figure 8.13: *CoCoME* OWL Ontology

The OWL ontology *CoCoME* consists of the class definitions, individuals, properties, subclass-superclass hierarchy, as well as the safety and security requirements. Moreover, the elements of this ontology are distributed to separate tables, (see Tables 8.10, 8.11, 8.12, and 8.13) in order to categorize the elements, specify their types, super-elements, properties and individuals or values accordingly. The attribute/Multi-attribute classes, and also the property classes of the ontology *CoCoME* along with their super-classes, properties and possible individuals are illustrated in Table 8.10.

The trustworthy classes, consisting of data constraints, time constraints, and role classes, along with their super-classes, properties and values are determined in Table 8.11.

Table 8.10: Ontological Attribute/Multi-attribute/Property classes

	Ontological Element	Ontological Element Type	Super-element	Property	Individual
1	CoCoME	Root Class	Thing		
2	Inventory	Attribute Class	CoCoME	hasProvidedFR, hasInternalFR, hasRequestedFR	{Inventory1, Inventory2, Inventory3, Inventory4}
3	MClass-CashDesk	Multi-attribute Class	MClass-CashDesk	hasCashDeskProperty	
4	CashBox	Attribute Class	MClass-CashDesk	hasProvidedFR, hasInternalFR, hasRequestedFR	{CashBox1, CashBox2, CashBox3, CashBox4, CashBox5,…, CashBox12}
5	Cashier	Attribute Class	MClass-CashDesk	hasProvidedFR, hasInternalFR, hasRequestedFR	{Cashier1,Cashier2, Cashier3, Cashier4, Cashier5, Cashier6}
6	CashDeskId	Property Class	MClass-CashDesk	isCashDeskPropertyOf	
7	CashDeskPC	Property Class	MClass-CashDesk	isCashDeskPropertyOf	
8	Sale	Property Class	MClass-CashDesk	isCashDeskPropertyOf	
9	InStore	Property Class	MClass-CashDesk	isCashDeskPropertyOf	

Table 8.11: Ontological trustworthy classes

	Ontological Element	Ontological Element Type	Super-element	Property	Value
1	DataConstraints	Trustworthy Class	CoCoME		
2	Authorization	DataConstraint Class	DataConstraints		{1, 2}
3	IsExpress	DataConstraint Class	DataConstraints		{Express, Normal}
4	IsMore	DataConstraint Class	DataConstraints		{1, 2}
5	Mode	DataConstraint Class	DataConstraints		{Disable, Done InSale, Ready, Waiting}
6	PaymentMethod	DataConstraint Class	DataConstraints		{1, 2}
7	TimeConstraint	Trustworthy Class	CoCoME		{T1.0S, T2.0S, T120.0S}
8	Roles	Trustworthy Class	CoCoME		
9	Role-Cashier	Role Class	Roles	hasFRPrivilege	
10	Role-Manager	Role Class	Roles	hasFRPrivilege	
11	Role-StockManager	Role Class	Roles	hasFRPrivilege	

The provided, requested, and internal functional requirements of the concept *Cashier*, and their super-classes and properties are illustrated in Table 8.12.

Table 8.12: *Cashier* Ontological functional requirement classes

	Ontological Element	Ontological Element Type	Super-element	Property
1	FRequirements	Attribute Class	CoCoME	
2	ProvidedFRs	Attribute Class	FRequirements	
3	InternalFRs	Attribute Class	FRequirements	
4	RequestedFRs	Attribute Class	FRequirements	
5	IsMoreItem	Functional Requirement Class	ProvidedFRs	hasRolePrivilege hasnotRolePrivilege
6	Pay	Functional Requirement Class	ProvidedFRs	hasRolePrivilege
7	CancelExpress	Functional Requirement Class	InternalFRs	hasRolePrivilege hasnotRolePrivilege
8	StartSale	Functional Requirement Class	InternalFRs	hasRolePrivilege
9	PassItem	Functional Requirement Class	RequestedFRs	
10	SaleFinished	Functional Requirement Class	RequestedFRs	
11	DisableExpress	Functional Requirement Class	RequestedFRs	
12	Card	Functional Requirement Class	RequestedFRs	
13	Cash	Functional Requirement Class	RequestedFRs	
14	IFRCashier-CancelExpress__FRDisableExpress	Attribute Class	Cashier	hasMode hasIsExpress
15	IFRCashier-StartSale__FRPassItem	Attribute Class	Cashier	hasMode hasTimeConstraint
16	FRCashier-IsMoreItem__FRPassItem	Attribute Class	Cashier	hasMode hasIsMore
17	FRCashier-IsMoreItem__FRSaleFinished	Attribute Class	Cashier	hasMode hasIsMore
18	FRCashier-Pay__FRCard	Attribute Class	Cashier	hasMode hasIsExpress hasIsMore hasPaymentMethod
19	FRCashier-Pay__FRCash	Attribute Class	Cashier	hasMode hasIsMore hasPaymentMethod

Finally, the defined properties of the ontology *CoCoME* and their super-properties, domains, and ranges are stated in Table 8.13.

The OWL ontology *CoCoME* was verified in TopBraid Composer software tool by running the inferences including superclass and consistency checking, and no contradiction or redundancy was found.

Table 8.13: Ontological properties

	Ontological Property	Super-property	Domain	Range	Inverse of
1	hasConstraint				
2	hasDataConstraint	hasConstraint		DataConstraints	
3	hasAuthorization	hasDataConstraint		Authorization	
4	hasIsExpress	hasDataConstraint		IsExpress	
5	hasIsMore	hasDataConstraint		IsMore	
6	hasMode	hasDataConstraint		Mode	
7	hasPaymentMethod	hasDataConstraint		PaymentMethod	
8	hasSecurityConstraint	hasConstraint			
9	hasRolePrivilege	hasSecurityConstraint	FRequirements	Roles	
10	hasnotRolePrivilege	hasSecurityConstraint	FRequirements	Roles	
11	hasFRPrivilege	hasSecurityConstraint	Roles	FRequirements	
12	hasnotFRPrivilege	hasSecurityConstraint	Roles	FRequirements	
13	hasTimeConstraint	hasConstraint		TimeConstraint	
14	hasProperty				isPropertyOf
15	hasCashDeskProperty	hasProperty	CashDesk	MClass-CashDesk	isCashDeskPropertyOf
16	isPropertyOf				hasProperty
17	isCashDeskPropertyOf	isPropertyOf	MClass-CashDesk	CashDesk	hasCashDeskProperty
18	hasFRequirement				
19	hasProvidedFR	hasFRequirement		ProvidedFRs	
20	hasInternalFR	hasFRequirement		InternalFRs	
21	hasRequestedFR	hasFRequirement		RequestedFRs	

8.1.3 Transformation from OWL Ontology to TADL Description

In this step, the OWL ontology *CoCoME* obtained from model transformation process is automatically transformed to TADL description language by mapping the ontological elements to their corresponding TADL constructs. This mapping is accomplished by the model transformation processing explained in Chapter 7. The automatically generated output meta-model is the *CoCoME* TADL XML-format file that contains the detailed specifications of reusable components and component-based architecture of the relevant trustworthy system. The obtained TADL file complies with the XML schemas

that are explained in Chapter 7. After the transformation process, the *CoCoME* TADL XML-format file is opened in Microsoft Visual Studio and depicted in Figure 8.14. Figure 8.15 and Figure 8.16 show two parts of the *CoCoME* TADL XML file, which are automatically generated through the model transformation process. In Figure 8.15, one of the reactivity properties of the component *CashBox* is depicted. The connector type *CashBox_Inventory* and the two relevant connector role types are shown in Figure 8.16.

```
OutputTADL.xml - Microsoft Visual Studio
File  Edit  View  Debug  XML  Tools  Test  Window  Help
OutputTADL.xml

<componentType>
   <name>CashDesk</name>
   <constraint/>
   <interfaceTypes/>
   <architectureType>
      <name>CashDesk</name>
      <componentType>
         <name>CashBox</name>
         <attribute/>
         <constraint/>
         <interfaceTypes>
            <name>CashBox_Inventory</name>
            <protocol/>
            <serviceType>
               <name>Info</name>
               <id/>
               <type>input</type>
               <constraint/>
            </serviceType>
            <serviceType>
               <name>YesExpress</name>
               <id/>
               <type>input</type>
               <constraint/>
            </serviceType>
            <serviceType>
               <name>NotExpress</name>
               <id/>
               <type>input</type>

Ready          Ln 110     Col 19     Ch 19          INS
```

Figure 8.14: *CoCoME* TADL file

```
<reactivity>
    <name>CashBox2</name>
    <id/>
    <service-request>
        <name>DisableExpress</name>
        <id/><type>input</type>
        <constraint/>
    </service-request>
    <service-response>
        <name>ChangeModeToNormal</name>
        <id/><type>output</type>
        <constraint/>
    </service-response>
    <dataConstraint>
        <name>CashBoxDataConstraint2</name>
        <service-request>
            <name>DisableExpress</name>
            <id/><type>input</type>
            <constraint/>
        </service-request>
        <service-response>
            <name>ChangeModeToNormal</name>
            <id/><type>output</type>
            <constraint/>
        </service-response>
        <constraint>IsExpress==Express and Mode==Disable</constraint>
        <descreption/>
    </dataConstraint>
    <timeConstraint>
        <name>CashBoxTimeConstraint3</name>
        <constraint/>
        <service-request>
            <name>DisableExpress</name>
            <id/><type>input</type>
            <constraint/>
        </service-request>
        <service-response>
            <name>ChangeModeToNormal</name>
            <id/><type>output</type>
            <constraint/>
        </service-response>
        <maxSafeTime>1</maxSafeTime>
    </timeConstraint>
    <update>
        <toBeUpdated>IsExpress</toBeUpdated>
        <value>Normal</value>
    </update>
    <action>
        <name>TurnLightOff</name>
        <id/><type/><description/>
        <from>ChangeModeToNormal</from>
        <FromId/><to>idel</to>
    </action>
</reactivity>
```

Figure 8.15: The tag Reactivity of *CoCoME* TADL file

```xml
<connectorType>
    <name>ConnectorTypeCashBox_Inventory</name>
    <connectorRoleType>
        <name>ConnectorRoleType1CashBox_Inventory</name>
        <constraint/>
        <interfaceType>
            <name>CashBox_Inventory</name>
            <protocol/>
            <serviceType>
                <name>Info</name><id/><type>input</type>
                <constraint/>
            </serviceType>
            <serviceType>
                <name>YesExpress</name><id/><type>input</type><constraint/>
            </serviceType>
            <serviceType>
                <name>NotExpress</name><id/><type>input</type><constraint/>
            </serviceType>
            <serviceType>
                <name>CheckLastHour</name><id/><type>output</type><constraint/>
            </serviceType>
            <serviceType>
                <name>GetInfo</name><id/><type>output</type><constraint/>
            </serviceType>
            <serviceType>
                <name>AddToInventory</name><id/><type>output</type><constraint/>
            </serviceType>
        </interfaceType>
    </connectorRoleType>
    <connectorRoleType>
        <name>ConnectorRoleType2CashBox_Inventory</name>
        <constraint/>
        <interfaceType>
            <name>Inventory_CashBox</name>
            <protocol/>
            <serviceType>
                <name>Info</name><id/><type>output</type><constraint/>
            </serviceType>
            <serviceType>
                <name>YesExpress</name><id/><type>output</type><constraint/>
            </serviceType>
            <serviceType>
                <name>NotExpress</name><id/><type>output</type><constraint/>
            </serviceType>
            <serviceType>
                <name>CheckLastHour</name><id/><type>input</type><constraint/>
            </serviceType>
            <serviceType>
                <name>GetInfo</name><id/><type>input</type><constraint/>
            </serviceType>
            <serviceType>
                <name>AddToInventory</name><id/><type>input</type><constraint/>
            </serviceType>
        </interfaceType>
    </connectorRoleType>
    <constraint/>
    <descreption/>
</connectorType>
```

Figure 8.16: The tag ConnectorType of *CoCoME* TADL file

8.2 Evaluation

In this Section, we evaluate the results of our approach by reviewing the proposed methodologies and comparing them with the previous works done by VMT tool [89] and Transformation tool [40].

The VMT tool [89] generates the TADL description language of component-based systems by providing a graphical user interface for developers to manually design components, connectors and system configuration. The derived TADL file from the VMT tool represents the formal behavior model, which contains all XML schemas described in Chapter 7. Presently, the VMT tool does not have the facility of opening the TADL files created by tools, other than VMT.

The Transformation tool [40] takes the obtained TADL description as the input file and transforms it to XML-format files for UPPALL and TIMES tools. The Transformation tool does not support the *architecture type* of TADL. That means the composite components and their interior sub-components cannot be transformed and verified by the UPPAAL and TIMES tools. By adding this feature in future, all component types of TADL description language can be verified by the model checking tools such as UPPAAL and TIMES.

The drawbacks of the solution stated in [89] and [40] are mentioned in Chapter 3. Below, the significance of the solutions obtained by using the Formal Concept Analysis formalism is discussed.

In this thesis, Formal Concept Analysis (FCA), a mathematical theory based on the formalization of concept hierarchy and lattice theory, has been applied for domain analysis. This effort itself is both new and novel. It was never attempted before. The application of FCA in the first stages of design has the advantage of constructing a

consistent class hierarchy. As a concrete example we refer to the implemented *CoCoME* case study, in which the functional requirement properties of the concept *CashBox* (see Table 8.2) consists of provided and requested functional requirements, data constraints, time constraints, updates and actions. First, the attribute variables of the concept *CashBox* are defined separately from the attribute variables of other concepts (see Table 8.3). Second, the names of the functional requirement properties and their time constraints are distinctively defined by including the name of the concept *CashBox*, e.g., *FRCashBox-SaleFinished_FRPay* is one of the functional requirement properties of *CashBox* that provides the functional requirement *SaleFinished* and requests the functional requirement *Pay*. Third, the attribute variables like data constraints, updates, and actions which may be shared between the context tables are integrated and pruned in the merged context tables, e.g., *DCMode_InSale* is a shared attribute variable between *CashBox* and *Cashier* binary context tables (see Figure 8.2 and Figure 8.4) that has been merged in the *CashDesk* binary context table (see Figure 8.6).

Besides, by extracting the implication rules, the user would be able to make the logical deductions and discover the intra-concept relations between the concepts, their functional requirements and the constraints of the functional requirements. As another concrete example we refer to the generated implication rules of the concept lattice *CoCoME* (see Figure 8.12). Each implication rule has the following structure:

```
<rule>
   <premise>
      {FRCashBox-NotExpress__IFRIgnore}
   </premise>
   <consequence>
      {Action_TurnLightOff, CashBox_, CashDesk, CoCoME, DCMode_Waiting,
      UpdateDCIsExpress_Normal, UpdateDCMode_Ready}
   </consequence>
   <support>0.04</support>
   <confidence>1.0</confidence>
</rule>
```

The above implication rule implies that each object having the functional requirement attribute *FRCashBox-NotExpress_IFRIgnore*, with the support of four percent and the confidence of hundred percent, also has all the attribute variables within the consequence element. In other words, whenever the functional requirement *FRCashBox-NotExpress_IFRIgnore* occurs, the data constraint *Mode* has the value *Waiting*, which is updated to *Ready*. Also, the data constraint *IsExpress* is updated to *Normal*, and the action *TurnLightOff* is triggered by the functional requirement *NotExpress*.

Some guidelines are specified to merge the defined context tables (see the rules to compose and integrate partially defined context tables in Chapter 4). As an example, the time constraint definition of a functional requirement property has the following notation: *TCFRCashBox-SaleFinished_FRPay=T1.0S* that indicates the maximum allowed time to fulfill the functional requirement *SaleFinished.*

It is important to mention that, the trustworthy credentials such as the safety and security requirements are identified in the context tables of FCA. As an example of security property we refer to the role privileges that are defined in *CoCoME* binary context table (see Figure 8.9). As an example, the role privilege *RolePrivilege-Cashier_Pay* implies that any user having the role *Cashier* has the privilege of providing the functional requirement *Pay*. Also, the mentioned role privilege attribute is in incidence relation with the objects *CashBox7*, *CashBox12*, *Cashier2*, and *Cashier3* having the functional requirement attributes *FRCashBox-SaleFinished_FRPay*, *FRCashBox-Declined_FRPay*, *FRCashier-Pay_FRCard*, and *FRCashier-Pay_FRCash,* respectively. On the other hand, the negative role privilege attribute, e.g., *RolePrivilegeNot-Manager_IsMoreItem* declares that any user having the role *Manager* does not have the privilege of access to provide the functional requirement *IsMoreItem*.

An OWL ontology, as a formal representation of domain knowledge is automatically produced by the model transformation process. So, the concept hierarchy developed in

the concept lattice of FCA is correspondingly transferred to the class hierarchy in the resulting ontology. Finally, the OWL ontology *CoCoME* is opened in TopBraid Composer software tool. As a concrete example, we refer to the functional requirement properties of the concept *Inventory* in the concept lattice *CoCoME* (see Figure 8.10). After the model transformation process from concepts to ontology, the subconcepts of the concept *Inventory* have the corresponding class hierarchy generated in ontology, which are depicted as the subclasses of the class *Inventory* (see Figure 8.13).

It has to be mentioned that, the implication rules derived from the concept lattice *CoCoME* are applied to the model transformation process from concepts to ontology, in order to capture the concepts holding the premise-consequence relationship. By detecting such intra-concept relations in concept lattice, we figured out the corresponding ontological classes and their relationships more accurately. For instance, the following implication rule implies that when the functional requirement *Approved* is provided and the functional requirement *Print* is requested, with the confidence of hundred percent the data constraint *Mode* has the value *InSale*, and is updated to *Done*. Also, the functional requirement *AddToInventory* is triggered by providing the functional requirement *Approved*.

```
<rule>
  <premise>{FRCashBox-Approved__FRPrint}
  </premise>
  <consequence>{Action_AddToInventory, CashBox_, CashDesk, CoCoME, DCAuthorization_1,
                   DCIsExpress_Normal, DCIsMore_1, DCMode_InSale, DCPaymentMethod_2,
                   UpdateDCMode_Done}
  </consequence>
  <support>0.04</support>
  <confidence>1.0</confidence>
</rule>
```

The target ontology can be utilized as a shared knowledge containing reusable concepts, and the queries and assertions are exchanged with ontology among domain experts. Since the OWL ontology is based on logical models, the user can take advantage of using its reasoning engine to accomplish the syntax checking, consistency checking and subsumption. Therefore, if there would be any deficiency or contradiction in the developed ontology, the user can identify and fix it by modifying the relevant context tables of FCA. The iterative process of validation is conducted to ensure that the system design is syntactically and semantically correct with respect to ontology reasoning. The OWL ontology *CoCoME* was verified in TopBraid Composer software tool by running the inferences including superclass inference, and consistency checking inference and no contradiction or redundancy was found.

One of the advantages of our methodology is that, the ontology verification process can be done for any partially defined context table and not necessarily for the final integrated context table. For example, when the concept *CashBox* along with its functional and non-functional requirements is defined in *CashBox* context table, it may be separately converted to build the OWL ontology *CashBox*, which is opened in TopBraid Composer and verified by running inferences.

Afterwards, the verified ontology is automatically transformed to TADL [49] architecture description language which is the formal specification of the dependable component-based system. The implemented model transformation process supports the *architecture type* of TADL. That means, the composite components and their interior sub-components are generated properly. As an example, we refer to the *CoCoME* TADL file that consists of the component types *CashBox*, *Cashier*, and *Inventory*. The *CashBox* and *Cashier* components are defined inside an architecture type named *CashDesk*. The composite component *CashDesk* and the component *Inventory* are defined inside the architecture type *StoreSystem*.

Finally, for doing the formal analysis of design, the architecture of the trustworthy system, formally described in TADL, is taken as the input for the analysis stage and is transformed to the behavior protocols which are used by existing concept verification tools. The transformation tool [40] has used model checking and real-time schedulability techniques to verify that the system under development is both safe and secure. This tool automatically generates two model types from a TADL description. One is the UPPAAL model on which the security and safety properties of the system under design are formally verified. The second type is the TIMES model, on which real-time schedulability analysis is performed.

The *CoCoME* TADL file, which is generated through the provided model transformation process in this thesis, was examined by the Transformation tool [40] to generate the UPPAAL and TIMES model types. However, since the Transformation tool [40] does not support the architecture type in TADL, the inner level component types were not created in the output models. Therefore, we temporarily changed the TADL file so that all defined components within the architecture types were transferred to the first level of the hierarchy. In other words, the hierarchy of the components was changed to a linear architecture. After doing this adjustment, the *CoCoME* TADL file was converted by the Transformation tool [40] and the output model was successfully evaluated by the UPPAAL verification tool.

Chapter 9

Conclusion

This thesis has introduced a formal approach that aims to perform domain analysis by an application of Formal Concept Analysis (FCA) theory. Although the work presented in this thesis was primarily motivated by the methodology proposed in [48] for the development of dependable software systems, the framework that has been developed in this thesis will be useful for any component-based software development methodology. The current Component-Based Software Engineering (CBSE) practices have not adequately dealt with the practical aspects of domain analysis, yet all CBSE approaches agree on its importance. When faced with the construction of trustworthy software it is necessary to develop trustworthiness criteria at the domain level [42].

Many of the available solutions in CBSE either cannot or do not provide a mechanism for constructing and specifying trustworthiness criteria at the domain level. It is also the case that existing solutions do not making use of formalisms and easy-to-use formal analysis tools in the current analysis methods. The results of this thesis address these drawbacks and offers effective solutions.

A trustworthy domain model is constructed in the thesis, by introducing and implementing automatic model transformation approaches. The OWL ontology obtained

from the initial FCA models may be verified by running the inference checking of ontology software tools, and/or comparing to other existing ontologies in the same domain. The trustworthy requirements are specified at the first stage of design activities and the obtained OWL ontology is transformed to the target TADL component model. The trustworthiness properties are thus stated based on domain properties. We demonstrated this methodology by means of the Common Component Modeling Example (CoCoME) [36] case study.

9.1 Summary of Results

In this section, we discuss and evaluate the results achieved in this thesis with respect to the contributions stated in Chapter 1.

1. *Defining "Formal Concepts" and "Trustworthy Properties" using FCA through domain analysis*. In Chapter 4, we provided a domain analysis methodology for capturing the formal concepts and constructing formal context tables using FCA mathematical theory. The research problems, proposed solutions, and limitation are stated below:
   - Problem 1: The currently available analysis methods do not use formalisms, or have difficult to use formal analysis tools.
   - Solution 1: The application of FCA in the first stages of design had the advantage of constructing a concrete formal concept hierarchy. Moreover, the rules extracted by the formal methods enabled us to make the logical deductions to identify the relations among the concepts and the design constraints.

- Problem 2: The maintenance of the messy context tables defined for the software systems containing large number of intent attributes, is time consuming and error-prone.
- Solution 2: We took advantage of many-valued context tables and converted them to their corresponding binary context tables.
- Limitation: In the provided approach, capturing formal concepts from the system requirements specifications, also defining and merging the formal context tables are manually done. A future contribution can be the application of text mining techniques and using the relevant tools to reduce time and cost. However, the role of the designer may not be ignored, since some domain expert knowledge is required in this field.

2. *Defining rules and conventions to specify "Component-based Artifacts and Trustworthy Properties" in formal context tables*. In Chapter 4, we provided the rules to categorize the concepts and determine the name conventions for the various attribute names of the formal context tables. Also, the safety, security and timeliness properties were defined in the formal context tables. The research problems, proposed solutions, and limitation are stated below.
   - Problem 1: The component models are localized in the selection of the appropriate fault tolerant mechanisms at the final development stages, and not at the primary steps of software development process.
   - Solution 1: The trustworthy requirements were defined as the attributes of the formal context tables, at the first steps of the software design.
   - Limitation: The proposed methodology provides the facility of composing safety, security, and timeliness properties. Further research is required to

extend the rules for defining other trustworthy requirements such as reliability and availability.

3. *Defining rules and conventions to integrate "Partially Defined Context Tables" and construct "Unified Formal Concept Lattice".* In Chapter 4, the integration rules were introduced to merge and prune the partially defined context tables in order to construct a unified formal context table. Also, the defined priorities identified the group of BCTs to be combined. Some redundant attributes were removed, some properties of attributes were converted to attributes and some attributes were unified and merged together.
   - Limitation: The integration process of partially defined context tables is manually done. Further research is required to investigate automated methods.
4. *Defining and implementing a "Model Transformation Approach" to generate "OWL Ontology" containing the "Trustworthy Requirements" from formal concept lattice.* In Chapters 5 and 6, a model transformation approach was provided to transform concept lattice to OWL ontology. The obtained OWL domain model contains the trustworthy criteria. The research problems, proposed solutions, and limitation are stated below:
   - Problem 1: The concept lattice that is transformed to OWL ontology is not a reduced labeling lattice, but a complete lattice, in which every node is marked by all corresponding extents and intents. This will lead to a great cluttering of the picture and the redundancy of data.
   - Solution 1: The reduced labeling technique was used to overcome this problem. In Chapter 6, the transformation algorithm "Lattice Reducer" was developed as the first step of the proposed model transformation technique and transformed the concept lattice to the reduced labeling lattice.

- Problem 2: Besides the '*subconcept-superconcept'* order among the nodes of a concept lattice, there is an essential need to realize other relations among the attributes of a context.
- Solution 2: In Chapters 5 and 6, the implication rules derived from the concept lattice were applied to the model transformation process in order to capture the concepts holding the premise-consequence relationship. By detecting such intra-concept relations in concept lattice, we figured out the corresponding ontological classes and their relationships, more accurately.
- Limitation: Whenever the designer makes any change in the target ontology, the modifications are not reflected in the defined formal context tables in FCA. Further research is required to implement reverse transformation process from ontology to FCA context tables. This new opportunity will provide the facility of backward traceability to reduce the effort required to determine the impacts of modifications.

5. *Generating "TADL Specification" of the component-based system by implementation of a "Model Transformation Technique" from OWL ontology*. In Chapter 7, we implemented a model transformation framework to automatically produce the TADL specification of reusable component-based architecture of the relevant trustworthy system.

9.2 Assessment

The formal approach presented in this thesis is a contribution to CBSE via taking advantage of formalism, provided by FCA mathematical theory and OWL domain model ontology. When the obtained OWL ontology passes the verification of the reasoning

check, it is transformed automatically to the TADL architecture description language which is the formal specification of the dependable component-based system. In this section, we evaluate our formal approach with respect to completeness, reusability, testability, and usability criteria.

Completeness: *Are the formal artifacts generated in our methodology complete? Is the completeness criteria retained through the model transformation processes?*

The completeness of the constituent elements of our methodology is illustrated as follows:

- *Ontology*: Through model transformation process, all derived formal concepts are transformed to their corresponding artifacts in OWL ontology. It can be confirmed from the provided rules in Chapter 5 that for each defined formal concept belonging to the context tables, there is a corresponding class, property or individual in the target OWL ontology. This OWL ontology can be compared to the corresponding existing ontologies in the same domain to verify its completeness. Any incompleteness detected in this step can be adjusted by returning back to the context table definition step.
- *Component Model*: The obtained OWL ontology was transformed to the formal component model which includes all the elements specified in the TADL XML Schemas presented in Chapter 7. Since the created TADL description is a one to one mapping from a complete OWL ontology, then we can claim that it is also complete.
- *Trustworthiness*: All trustworthy properties regarding to the TADL specification such as safety, security, and timeliness are sufficiently defined as non-functional requirements from the first stages of design in the FCA formal context tables. The same trustworthy specifications are entirely transformed to the generated OWL ontology and also to the target TADL component model. So, the

transformation is complete with respect to trustworthiness property stated at the domain level, however only domain experts can ensure an acceptable completeness level for the property.

- *Case Study*: We have tested our approach on Common Component Modeling Example, which is a benchmark case study for testing the modeling ability of component models. The results are provided in Chapter 8. It shows that our methodology is capable of modeling such case study, by doing domain analysis, building OWL ontology, and generating TADL specification of such component model.

Reusability: *Does the formal artifacts generated in our methodology support reuse?*

The reusability of the constituent elements of our methodology is illustrated as follows:

- *Formal Concepts*: The objects and attributes as the abstract elements of the formal context tables are reusable and may be defined in several formal context tables. Also, every formal context table corresponding to a concept in the use case is defined separately. So, they may be merged with other formal context tables or may be reused for other systems.
- *Ontology*: Building ontologies is a major approach for capturing and representing reusable knowledge. The ontological elements including classes, properties and individuals may be used for creating and storing reusable building blocks in a well defined machine-readable format. Also, the ontologies can be merged together. In this thesis, the reusable concepts are derived from ontology and transformed to reusable components.
- *Component Model*: Since every TADL element is described separately, it is possible to reuse these definitions for different systems. The defined repository

tool hosts the component-based system specifications so that the elements can be reused.

Testability: *Is it possible to validate whether or not the specifications of the formal artifacts generated in our methodology are right?*

The testability of the constituent elements of our methodology is illustrated as follows:

- *Formal Concepts*: There are well-formed rules and name conventions provided in Chapter 4 for defining and verifying the objects and various attribute types of the formal context tables. Also, each partially defined context table can be automatically transformed to the partially formed OWL ontology and verified by the reasoning engine of the ontology software tools. On the other hand, the implication rules derived from the concept lattice can be considered as a confident source of data to verify the correctness of the relations holding among the concepts.
- *Ontology*: OWL ontology can be verified by the reasoning engine of the ontology. The verification includes syntax and consistency checking to ensure that the ontology does not contain contradictions. It also includes subsumption to ensure that subclasses are defined correctly.
- *Component Model*: There are some well-formed rules provided for the elements of the TADL formal model. Also, the Transformation tool [40] transforms the TADL description to XML files for UPPAAL and TIMES model checkers. As an example, UPPAAL model checker performs the required verification, simulation, and schedulability analysis.

Usability: *Are the process steps easy for the designer to follow?*

The usability of the constituent elements of our methodology is illustrated as follows:

- *Formal Concepts*: Although FCA is a mathematical theory, following the provided rules and name conventions, and defining the abstract elements of the formal context tables in a FCA software tool is not a complex task. Also, the integration rules are provided to merge and prune the partially defined context tables. The rest of the process consists of drawing concept lattice diagram, extracting lattice in an XML file, generating the implication rules and exporting them to an XML file, which are automatically accomplished by Lattice Miner software tool.
- *Ontology*: OWL ontology is a formal domain model that is based on description logics. The formalism is working in the background, while the OWL ontology is automatically generated by the model transformation techniques. The results are represented in the user interface of the TopBraid Composer software tool, and the user can easily run the automated reasoning to verify the obtained ontology.
- *Component Model*: The formal component model, which is automatically generated by implementing the model transformation techniques, is based on the TADL XML Schemas. Then, the Transformation tool [40] automatically produces two extended timed automata to be verified by the model checking tools UPPAAL and TIMES. Thus, the whole process is supported by tools.

9.3 Case Study

In this section, we briefly state the work done to apply our design methodology on a benchmark case study in the field of component-based development. The case study is the Common Component Modeling Example (CoCoME) [36]. The goals of applying our methodology on this case study are:

- Test the domain analysis and construction of the formal context tables and concept lattice in Lattice Miner software tool; the application of FCA in the first stages of design had the advantage of constructing a consistent class hierarchy.
- Test the rules to compose partially defined context tables, based on the TADL component elements. The trustworthy credentials such as safety, security, and timeliness were included.
- Test the rules to integrate partially defined context tables;
- Test the implication rules extracted from the derived concept lattice. By using the implication rules, the user can make the logical deductions to verify the intra-concept relations between the concepts and design constraints.
- Test the automatic model transformation process, which transforms concept lattice to OWL ontology. By using the implication rules derived from concept lattice, ontological classes and their relationships were figured out.
- Verification of the obtained OWL ontology in TopBraid Composer software tool by running the inferences including superclass inference and consistency checking inference. No contradiction or redundancy was found.
- Test the automatic model transformation process, which transforms the verified OWL ontology to TADL specification. The implemented model transformation supports the *architecture type* of TADL. It means the composite components and their interior sub-components were generated properly.
- Test the automatic model transformation process, which transforms TADL description to extended timed automata.

- Test the verification of the properties of trustworthiness in the obtained extended timed automata by the UPPAAL model checker. The safety and security properties were successfully evaluated by the UPPAAL verification tool.

References

[1] V. Alagar and M. Mohammad. *A component model for trustworthy real-time reactive systems development*. In International Workshop on Formal Aspects of Component Software (FACS07), Sophia-Antipolis, France, September 2007.

[2] V. Alagar, M. Mohammad, and K. Wan. *The Role of Concept, Context, and Component for Dependable Software Development*. In Proceedings of ICFCA'2010. vol. 5986, pp.34-50, Springer, 2010.

[3] R. De Almeida Falbo, G. Guizzardi, and K. Duarte. *An Ontological Approach to Domain Engineering*. In: International Conference on Software Engineering and Knowledge Engineering (SEKE), pp. 351--358, Ischia, Italy, 2002.

[4] J. A. Alonso, J. Borrego, M. J. Hidalgo, F. J. Martín, and J. L. Ruiz. *Verification of the formal concept analysis*. RACSAM (Revista de la Real Academia de Ciencias), Serie A: Matematicas, 98:3–16, 2004.

[5] G. Ammons, D.Mandelin, R. Bodik, and J.R. Larus. *Debugging temporal specifications with concept analysis*. In Proceedings of the Conference on Programming Language Design and Implementation PLDI'03. ACM, June 2003.

[6] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Möller, P. Pettersson, C. Weise, and W. Yi. *UPPAAL - Now, Next, and Future*. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, Modeling and Verification of Parallel Processes, number 2067 in Lecture Notes in Computer Science Tutorial, pages 100 125. Springer–Verlag, 2001.

[7] U. Andelfinger. *Diskursive Anforderungsanalyse*. Ein Beitrag zum Reduktionsproblem bei Systementwicklungen in der Informatik. Peter Lang, Frankfurt, 1997.

[8] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. *Basic concepts and taxonomy of dependable and secure computing*. IEEE Transactions on Dependable and Secure Computing,(1):11–33, 2004.

[9] T. Ball. *The concept of dynamic analysis*. In Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 216–234, September 1999.

[10] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[11] P. Becker and J. H. Correia. *The TosCanaJ suite for implementing conceptual information systems*. LNCS (LNAI), vol. 3626, pp. 324–348. Springer, Heidelberg (2005).

[12] K. Böttger, R. Schwitter, D. Richards, O. Aguilera, and D. Mollá. *Reconciling use cases via controlled language and graphical models*. In INAP'2001 - Proceedings of the 14th International Conference on Applications of Prolog, pages 20–22, Japan, October 2001. University of Tokyo.

[13] L. Boumedjout and L. Kwuida. *Lattice Miner: A Tool for Concept Lattice Construction and Exploration*. In Supplementary Proceeding of International Conference on Formal concept analysis (ICFCA'10), 2010.

[14] B. Chandrasekaran, J. R. Josephson, and V. R. Benjamins. *What are ontologies, and why do we need them?* IEEE Intelligent Systems, 14(1):20–26, 1999.

[15] *Concept explorer*: http://conexp.sourceforge.net/license.html ,last accessed on March 15, 2011.

[16] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge, U.K.: Cambridge Univ. Press, 1990.

[17] H. Dicky, C. Dony, M. Huchard, and T. Libourel. *ARES, adding a class and restructuring inheritance hierarchy*. In BDA : Onzièmes Journées Bases de Données Avancées, pages 25 42, 1995.

[18] S. Düwel and W. Hesse. *Bridging the gap between Use Case Analysis and Class Structure Design by Formal Concept Analysis*. In: J. Ebert, U. Frank (Hrsg.): Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik. Proc. "Modellierung 2000", pp. 27-40, Fölbach-Verlag, Koblenz 2000.

[19] S. Düwel and W. Hesse. *Identifying Candidate Objects During System Analysis*, Proc. CAiSE'98/IFIP 8.1 3rdInt. Workshop on Evaluation of ModelingMethods in System Analysis and Design (EMMSAD'98), Pisa 1998.

[20] T. Eisenbarth, R. Koschke, and D. Simon. *Aiding program comprehension by static and dynamic feature analysis*. In Proceedings of ICSM2001 - The International Conference on Software Maintenance, pages 602–611. IEEE Computer Society Press, 2001.

[21] T. Eisenbarth, R. Koschke, and D. Simon. *Locating features in source code. IEEE Transactions on Software Engineering*, 29(3):195–209, March 2003.

[22] B. Fischer. *Specification-based browsing of software component libraries*. In Automated Software Engineering, pages 74–83, 1998.

[23] A. Formica. *Ontology-based concept similarity in Formal Concept Analysis. Information Sciences*, 176(18):2624–2641, 2006.

[24] M. Fowler. *Refactoring, Improving the Design of Existing Code*. Addison Wesley, 1999.

[25] B. Ganter. *Two basic algorithms in concept analysis*. Preprint 831, Technische Hochschule Darmstadt, June 1984.

[26] B. Ganter and G. Stumme. *Formal Concept Analysis: Methods and Applications in Computer Science*. TU Dresden. 2003.

[27] B. Ganter and R. Wille. *Conceptual scaling*. In F. Roberts (Ed.), Applications of combinatorics and graph theory to the biological and social sciences. Berlin: Springer, 139-167, 1989.

[28] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc., 1999.

[29] B. Ganter and R. Wille. *Implikationen und Abhangigkeiten zwischen Merkmalen*. In P. O. Degens, H.-J. Hermes, and O. Opitz (Eds.), Die Klassifikation und ihr Umfeld. Frankfurt: Indeks, 171-185, 1986.

[30] R. Godin, R. Missaoui, and H. Alaoui. *Incremental concept formation algorithms based on Galois (concept) lattices*. Computational Intelligence, 11(2):246-267 (1995).

[31] R. Godin and P. Valtchev. *Formal concept analysis-based class hierarchy design in object-oriented software development*. In: B. Ganter, G. Stumme, and R. Wille. (eds.) Formal Concept Analysis. LNCS (LNAI), vol. 3626, pp. 192–207. Springer, Heidelberg (2005).

[32] B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, and U. Sattler. *Owl 2: The next step for owl. Web Semantics: Science, Services and Agents on the World Wide Web*, 6(4):309–322, 2008.

[33] T. R. Gruber. *A translation approach to portable ontology specifications*. Knowledge Acquisition, 5: 199-220, 1993.

[34] M. Gruninger and J. Lee. *Ontology: Applications and Design*. Commun. ACM, 45(2), 2002.

[35] J.-L. Guigues and V. Duquenne. *Familles minimales d'implications informatives résultant d'un tableau de données binaries.* Math. Sci. Hum. 24, 95, 5–18, 1986.

[36] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller. *The Common Component Modeling Example*. volume 5153 of LNCS, chapter CoCoME - The Common Component Modeling Example, pages 16–53. Springer, Heidelberg, 2008.

[37] W. Hesse and T. Tilley. *Formal Concept Analysis Used for Software Analysis and Modeling*. In B. Ganter, G. Stumme and R. Wille, editors, Formal Concept Analysis: Foundations and Applications, pages 288-303. Springer-Verlag, Germany, 2005.

[38] P. Hitzler and H. Schärfe. *Conceptual Structures in Practice*. Chapman & Hall/CRC Press, Boca Raton, FL, 2009.

[39] M. Horridge, S. Jupp, G. Moulton, A. Rector, R. Stevens, and C. Wroe. *A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools.* Edition 1.1, The University Of Manchester, October 16, 2007.

[40] N. Ibrahim. *Transforming architectural descriptions of component based systems for formal analysis*. Master thesis, Concordia University, 2008.

[41] *IHMC Cmap Tools*: http://cmap.ihmc.us/ ,last accessed on March 15, 2011.

[42] D. Jackson. "*A Direct Path to Dependable Software: Who could fault an approach that offers greater credibility at reduced cost?*", Communications of the ACM, Vol. 52 No. 4, 2009, Pages 78-88, A version of this article with a fuller list of references is available at http://sdg.csail.mit.edu/publications.html ,last accessed on March 15, 2011.

[43] H. Kaiya and M. Saeki. *Using domain ontology as domain knowledge for requirements elicitation*. In Proc. Of the IEEE Int. Req. Eng. Conf. (RE), pages 186–195, 2006.

[44] S. Kuznetsov. *On the intractability of computing the Duquenne-Guigues base*. J. Univers. Comput. Sci. 10(8), 927–933, 2004.

[45] Y. Q. Lee and W. Y. Zhao. *Domain requirements elicitation and analysis-An ontology-based approach*. In: Proceedings of Workshop on Computational Science in Software Engineering (CSSE), pp. 805-813, 2006.

[46] C. Lindig. *Concept-based component retrieval*. In J. Köhler, F., Giunchiglia, C. Green, and C. Walther, editors, Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs, pages 21–25, August 1995.

[47] S. Mangano. *XSLT Cookbook*. O'Reilly, 2nd Edition, December 2005.

[48] M. Mohammad. *A Formal Component-Based Software Engineering Approach for Developing Trustworthy Systems*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada (2009).

[49] M. Mohammad and V. Alagar. *TADL - An Architecture Description Language for Trustworthy Component-Based Systems*. In Proceedings of the 2nd European Conference of Software Architecture(ECSA 2008), Lecture Notes in Computer Science (LNCS 5292), 290-297, October 2008.

[50] R. Missaoui, L. Nourine, and Y. Renaud. *Generating positive and negative exact rules using formal concept analysis*: Problems and solutions. In ICFCA, pages 169–181, 2008.

[51] C. Mundie, P. de Vries, P. Haynes, and M. Corwine. *Trustworthy computing*. Microsoft White Paper, October 2002.

[52] N. F. Noy and D. L. McGuinness. *Ontology development 101: A guide to creating your first ontology*. Technical Report KSL-01-05, Stanford Knowledge Systems Laboratory, 2001.

[53] G. K. Palshikar. *An introduction to model checking*. http://www.embedded.com/columns/technicalinsights/17603352?requestid=179878 , December 2004, last accessed on March 15, 2011.

[54] Y. Park. *Software retrieval by samples using concept analysis*. Journal of Systems and Software, 2000. 54(3): p. 179-83.

[55] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005.

[56] U. Priss. *Formal Concept Analysis in information science.* Annual Review of Information Science and Technology (ARIST) 40 (2006), in press.

[57] *Protégé*: http://protege.stanford.edu/ ,last accessed on March 15, 2011.

[58] D. Richards, K. Boettger, and A. Fure. *Using RECOCASE to compare use cases from multiple viewpoints*. In Proceedings of the 13th Australasian Conference on Information Systems ACIS 2002, Melbourne, December 2002.

[59] G. Roberge. *Visualisation des résultats de la fouille des données dans les treillis des concepts*. Master's thesis, Université du Québec en Outaouais, 2007.

[60] H.A. Sahraoui, W. Melo, H. Lounis, and F. Dumont. *Applying concept formation methods to object identification in procedural code*. In Proceedings of International Conference on Automated Software Engineering (ASE '97), pages 210–218. IEEE, November 1997.

[61] T. Saridakis. *Robust Development of Dependable Software Systems, in Rapport de recherche INRIA*, juin 1999.

[62] B. K. Sarker, P. Wallace, and W. Gill. *Some Observations on Mind Map and Ontology Building Tools for Knowledge Management*. Ubiquity - Association for Computing Machinery, 2008.

[63] I. Schmitt and S. Conrad. *Restructuring object-oriented database schemata by concept analysis*. In T. Polle, T. Ripke, and K.-D. Schewe, editors, Fundamentals of Information Systems (Post-Proceedings 7th International Workshop on Foundations of Models and Languages for Data and Objects FoMLaDO'98), pages 177–185, Boston, 1999. Kluwer Academic Publishers.

[64] I. Schmitt and G. Saake. *Merging inheritance hierarchies for database integration*. In Proceedings of the 3rd International Conference on Cooperative Information Systems (CoopIS'98), New York, August 1998.

[65] F. B. Schneider, S. M. Bellovin, and A. S. Inouye. *Building trustworthy systems: Lessons from the ptn and internet*. IEEE Internet Computing, 3(6):64–72, 1999.

[66] M. Siff and T. Reps. *Identifying modules via concept analysis*. In Proceedings of the International Conference on Software Maintenance, pages 170–179. IEEE Computer Society Press, 1997.

[67] M. K. Smith, C. Welty, and D. L. McGuinness. *Owl web ontology language guide*. W3C Recommendation, February 2004. http://www.w3.org/TR/2004/REC-owl-guide-20040210/ last accessed on March 15, 2011.

[68] G. Snelting. *Software reengineering based on concept lattices*. In Proceedings 4th European Conference on Software Maintenance and Reengineeering, pages 3–12. IEEE, 2000.

[69] G. Snelting and F. Tip. *Understanding Class Hierarchies Using Concept Analysis*, ACM Transactions on Programming Languages and Systems, pp. 540-582, May 2000.

[70] I. Sommerville. *Software Engineering*. Addison Wesley, 8th edition, 2007.

[71] *Sourceforge*: http://www.sourceforge.net/ ,last accessed on March 15, 2011.

[72] G. Stumme. *Concept Exploration - A Tool for Creating and Exploring Conceptual Hierarchies*. In D. Lukose, H. Delugach, M. Keeler, L. Searle, and J. F. Sowa (Eds.), Conceptual Structures: Fulfilling Peirce's Dream. Proc. ICCS'97. LNAI 1257. Berlin: Springer, 318-331,1997.

[73] L. Szathmary and A. Napoli. *Coron: A framework for levelwise itemset mining Algorithms.* In Supplementary Proceedings of the Third International Conf. on Formal Concept Analysis (ICFCA'05), Lens, pages 110–113, 2005.

[74] J. Tennison. *Beginning XSLT 2.0: From Novice to Professional*. Apress, 2nd edition, 2005.

[75] T. Tilley. *Tool support for FCA*. In ICFCA, pages 104–111, 2004.

[76] T. Tilley. *Towards an FCA based tool for visualising formal specifications*. In B. Ganter and A. de Moor, editors, Using Conceptual Structures: Contributions to ICCS 2003, pages 227–240. Shaker Verlag, 2003.

[77] T. Tilley, R. Cole, P. Becker, and P. Eklund. *A Survey of Formal Concept Analysis Support for Software Engineering Activities*. In Proc. 1st International Conference on Formal Concept Analysis, 2003.

[78] *Tockit*: http://www.tockit.org/ ,last accessed on March 15, 2011.

[79] P. Tonella. *Concept analysis for module restructuring*. IEEE Transactions on Software Engineering, 27(4):351–363, April 2001.

[80] *TopBraid Composer*: http://www.topbraidcomposer.com/ ,last accessed on March 15, 2011.

[81] P. Valtchev, D. Grosser, C. Roume, and M. R. Hacene. *Galicia: An open platform for lattices, In Using Conceptual Structures*: Contributions to the 11th Intl. Conference on Conceptual Structures (ICCS'03, pages 241–254. Shaker Verlag, 2003.

[82] P. Valtchev, R. Missaoui, and R. Godin. *Formal concept analysis for knowledge discovery and data mining: the new challenges*. In: P. Eklund (Ed.), Concept Lattices: Proceedings of the Second International Conference on Formal Concept Analysis (FCA'04), Lecture Notes in Computer Science, vol. 2961, Springer, Berlin, 2004, pp. 352–371.

[83] P. Valtchev, R. Missaoui, and P. Lebrun. *A partition-based approach towards building Galois (concept) lattices*. Discrete Mathematics, 256(3):801-829, 2002.

[84] R. Wille. *Restructuring lattice theory: an approach based on hierarchies of concepts*. In Ordered sets. Edited by I. Rival. Reidel, Boston, pp. 445-470, 1982.

[85] M. Wirsing and , R. Ronchaud, editors. *Report on the EU/NSF Strategic Workshop on Engineering Software-Intensive Systems*, Edinburgh, UK, May 2004.

[86] K. E. Wolff. *A first course in Formal Concept Analysis - How to understand line diagrams*. In: Faulbaum, F. (ed.): SoftStat´93, Advances in Statistical Software 4, Gustav Fischer Verlag, Stuttgart 1994, 429-438.

[87] K. E. Wolff. *Einführung in die Formale Begriffsanalyse*. Actes 19e Séminaire Lotharingien de Combinatoire. 85-96. Strasbourg, Publication de l' Institut de Recherche Mathématique Avancée, 1988.

[88] S. Yevtushenko. *Computing and Visualizing Concept Lattices*. PhD thesis, TU Darmstadt, Germany, 2004.

[89] Z. Yun. *A visual modeling tool for the development of trustworthy component-based systems*. Master thesis, Concordia University, 2009.

Appendix A.

Table of Requirements for Evaluating Ontology Tools

Features	Protégé (w/OwlViz)	Altova SemanticWorks	SMORE /SWOOP	CMAP Tools(COE)	TopBraid
Create Presentations	2	2	2	6	2
Present Presentations (Step through)	2	2	2	8	2
Record Meeting Minutes - Template	2	2	2	6	2
Linking between maps	4	4	2	7	5
Word Processor import/export	2	4	5	7	2
Save Map as Template	4	5	4	7	4
Spellchecking	3	6	6	9	5
File Attachments	1	2	4	7	7
Usability					
Personal productivity (Easy Mind Map creation)	3	4	2	8	5
Ease of use	6	3	5	7	8
Technical and User Support					
Customizable elements for maps	3	4	2	8	5
Exposed SDK	10	2	4	6	10
Support Materials	9	7	5	8	10
Speed	8	6	8	7	8
System requirements (PC specs)	9	7	9	9	8
User community support	10	8	6	9	9
Help support	9	9	6	9	9
Cost	10	7	10	10	4
Company support	10	8	4	8	9
Company stability	8	10	5	7	9
Mac Support	10	1	1	10	10
IBM-PC Support	10	10	10	10	10
Ontology Related					
Create Ontology	10	10	6	6	9
Export OWL-RDF file	10	10	9	8	10
Create named links	8	9	7	9	10
Total Score	70.05	62	54	79	73.90