A Knowledge Based Reengineering Approach via Ontology and Description Logic

Ph.D Thesis

Hong Zhou

Software Technology Research Laboratory

De Montfort University

2011

To my wife, Yiqiong Wang,

my parents, Xiaomao Zhou and Shengyu Huang

for their love and support

Declaration

I declare that the work described in this thesis was originally carried out by me during the period of registration for the degree of Doctor of Philosophy at De Montfort University, U.K., from October 2006 to November 2010. It is submitted for the degree of Doctor of Philosophy at De Montfort University. Apart from the degree for which this thesis is currently applying, no other academic degree or award was applied for by me based on this work.

Acknowledgements

For many years I had dreamt about receiving a PhD and I would like to thank the many people who helped me achieve this dream in different ways when I undertook the work of this thesis.

I wish to express my most profound thanks to my supervisor Prof. Hongji Yang, for his invaluable advice, guidance and encouragement during my four-year study. He provided me with many useful comments and suggestions for the preparation of this thesis.

My thanks must go to Prof. David Budgen and Prof. Hussein Zedan, for examining my PhD thesis and providing many helpful suggestions. My research career will benefit tremendously from the research methodologies to which Prof. Budgen and Prof. Zedan introduced me.

A great many thanks go to the colleagues at De Montfort University, Dr. Feng Chen, Dr. Shaoyun Li, Mr. Zihou Zhou, Dr. Zhuopeng Zhang, Mr. Brian Graham, Mr. Peter Wells, Ms. Amanda Cook, Dr. Alan Brine and many others. I wish to thank them for their help and encouragement during the past years. Especially, I want to thank Peter and Amanda for agreeing to proof read my final thesis.

In addition, I would like to thank the Graduate School Office at De Montfort University for their outstanding management.

Finally, I wish to express thanks to my wife, Yiqiong Wang, my parents and my parents in law for their love, encouragements, patience and support over the past years. This thesis is dedicated to them.

Abstract

Traditional software reengineering often involves a great deal of manual effort by software maintainers. This is time consuming and error prone. Due to the knowledge intensive properties of software reengineering, a knowledge-based solution is proposed in this thesis to semi-automate some of this manual effort. This thesis aims to explore the principle research question: "How can software systems be described by knowledge representation techniques in order to semi-automate the manual effort in software reengineering?"

The underlying research procedure of this thesis is scientific method, which consists of: observation, proposition, test and conclusion. Ontology and description logic are employed to model and represent the knowledge in different software systems, which is integrated with domain knowledge. Model transformation is used to support ontology development. Description logic is used to implement ontology mapping algorithms, in which the problem of detecting semantic relationships is converted into the problem of deducing the satisfiability of logical formulae. Operating system ontology has been built with a top-down approach, and it was deployed to support platform specific software migration [132] and portable software development [18]. Data-dominant software ontology has been built via a bottom-up approach, and it was deployed to support program comprehension [131] and modularisation [130].

This thesis suggests that software systems can be represented by ontology and description logic. Consequently, it will help in semi-automating some of the manual tasks in software reengineering. However, there are also limitations: bottom-up ontology development may sacrifice some complexity of systems; top-down ontology development may become time consuming and complicated. In terms of future work, a greater number of diverse software system categories could be involved and different software system knowledge could be explored.

Table of Contents

List of Figures

List of Tables

List of Acronyms

API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
CIM	*Computation Independent Model*
DIG	*DL Implementation Group*
DL	*Description Logic*
EMF	*Eclipse Modelling Framework*
ER	*Entity-Relationship*
GUI	*Graphic User Interface*
ICE	*In-Circuit Emulator*
KR	*Knowledge Representation*
LOC	*Line Of Code*
MDA	*Model Driven Architecture*
MDE	*Model Driven Engineering*
MOF	*Meta-Object Facility*
OMG	*Object Management Group*
OPTIMA	*an Ontology-based PlaTform-specIfic software Migration Approach*
OS	*Operating System*

OWL	*Web Ontology Language*
PIM	*Platform Independent Model*
POSIX	*Portable Operating System Interface*
PSM	*Platform Specific Model*
QVT	*Query/View/Transformation*
RDF	*Resource Description Framework*
RDFS	*RDF Schema*
RTOS	*Real-time Operating System*
SPARQL	*Simple Protocol and RDF Query Language*
SWRL	*Semantic Web Rule Language*
SOA	*Service Oriented Architecture*
SQL	*Structured Query Language*
TS	*Technological Space*
UML	*Unified Modelling Language*
VOS	*Virtual Operating System*
VRTOS	*Virtual Real-Time Operating System*
XMI	*XML Meta-data Interchange*
XML	*eXtensible Markup Language*

Chapter 1 Introduction

Objectives

- To observe the need for the knowledge based software reengineering approach
- To explain the research objectives and select the research method
- To raise research questions and develop research propositions
- To highlight original contributions and define the measure of success
- To outline the organisation of the thesis

1.1 Problem Statement

The term "legacy system" is currently well-accepted and well-defined within both software research and the industry, which implies that people have already been convinced that new software becomes legacy software quickly and that this causes many problems in business and daily life. The growth in scale and functionality in any computing system that includes hardware and software systems will be inevitable. Evolution will be a way forward. Software reengineering, known as a combination of reverse engineering and forward engineering, is a practical solution for the problem of evolving existing computing systems [125]. Formal methods can be defined as mathematically based languages, techniques and tools for specifying and verifying systems [23]. It is one of the traditional software reengineering approaches, with which software engineers will be able to acquire a rigorous and precise description of the computing systems, and then to (semi-) automate the process of reengineering. However, many large systems may be too complicated to be described with formal methods. On the other hand, many reengineering activities such as top-down comprehension and bottom-up comprehension are based on cognitive theory [108], which mainly relies on domain knowledge and expertise rather than mathematically proved formulae. Cognitive theory based approaches are often manually performed by software maintainers [108], which are considered to be time consuming and error prone processes. Because of the complexity, the possibility of subtle errors and side effects is great. Moreover, some of these errors may cause catastrophic loss of money, time, and resources. Therefore, large systems are so complicated that it is impossible for a single individual to build and maintain all aspects of the system's design. Software programmers and maintainers of large systems are inundated by information overload.

"A Knowledge representation is a medium for efficient computation." [27] Knowledge-based approaches are often employed as solutions to (semi-) automate such tedious processes [5, 29, 38, 41, 59, 97, 103, 110, 123, 128, 129]. As an inherently knowledge intensive activity, software reengineering requires an understanding of many fields, from expertise to experience in the application domains. The integration of a knowledge-based approach and software reengineering will be one of the trends in the software reengineering research area.

Initially, Knowledge Representation (KR) was developed as a branch of Artificial Intelligence (AI) to enable computer systems to perform tasks that require human intelligence: information retrieval, resource allocation and logical reasoning etc. Recently, knowledge representation techniques have also been used in other fields, especially databases and object-oriented systems. Providing an effective high-level description of the world will be essential in a knowledge based approach, with which computer systems will be able to find implicit consequences of explicit knowledge.

The introduction of a knowledge-based approach into software reengineering can bridge the gap between software representation and mental model, and improve the efficiency and correctness of the software reengineering process. The focus of this approach will be on knowledge representation of software applications and problem domains, and traceability between a software system and its knowledge representation. The proposed research is targeted towards the development and usage of knowledge representation mechanisms to describe software applications and its problem domains, therefore semi-automating the manual effort in software reengineering. The main goal is to provide representation and inference techniques that allow properties of software to be described and inferred in a knowledge base. Ontology and description logic are selected as the underlying mechanism in this study. Ontology is a system of concepts in which all concepts are defined and interpreted in a declarative way [30]. Description logic provides the formal structure and rules of inference [7]. Both ontology and description logic are crucial in the proposed approach, since the terms and symbols will be ill-defined and confusing in description logic without ontology. Knowledge representation will be vague without description logic that deduces redundant or contradictory terms. Therefore, a knowledge based software reengineering approach is the integration of description logic and ontology to perform the task of constructing computable models and reasoning for some problem domains of reengineering.

1.2 Research Objectives and Research Methods

The research described in this thesis has the following objectives:

- to develop a knowledge based reengineering framework
- to create a guideline for representing software system with knowledge representation techniques
- to explore semi-automated mechanisms to generate and integrate knowledge representation of software systems
- to deploy and therefore to validate a knowledge based reengineering approach to different categories of software systems

The proposed research aims to build a practical knowledge based reengineering framework and to obtain a successful knowledge representation of software system. It is constructive, which implies that contributions will be made by introducing a new theory, algorithm, model, framework or methodology. However, it also involves complicated interaction between human being and software system. Therefore empirical research will be added in to explore such situations. Hence this thesis will reflect a combination of empirical and constructive research, which is both practical and academically rigorous. The following methods will be employed to fulfil the requirements of this constructive and empirical research:

- Formal method and cognitive theory: With the support of the mathematically proved formula, formal methods provide software reengineering with (semi-) automatic solutions, while cognitive theory mainly relies on domain knowledge/expertise and experience.

- Quantitative and qualitative methods: The proposed research reflects qualitative method by discussing wh-questions and the discussion of the more specific questions such as “how many” and “how often” implies quantitative method. Generally speaking, qualitative method provides the precondition of the usage of quantitative method.

- Modelling: The proposed research develops conceptual and knowledge based representations of computer systems to semi-automate tedious, time consuming and error prone processes.

- Classification: All software engineering research should be carried out in a systematic way, in which software taxonomy plays a very important role as the footstone. Software engineering researchers should be aware of the areas to which their studies belong and are related. Based on software system taxonomy discussed in Chapter 2, different categories of software systems will require different reengineering approaches due to the various functions and features that software systems may have. The following section describes the research method that is applied to this thesis, which links the constructive to the empirical.

1.3 Research Questions and Propositions

Research questions are the core part of the structure of the proposed research. The principal research question in this study is:

How can software systems be described by knowledge representation techniques in order to support (semi-) automating manual software reengineering tasks?

In order to answer this question, a set of research questions is defined that addresses the problem in detail.

RQ1: What knowledge of software system is going to be represented?

- What knowledge of software systems is needed in the context of software reengineering?
- What knowledge can be represented in relation to different categories of software systems?

RQ2: How can software system knowledge be represented?

- What knowledge representation techniques can be used to describe software system knowledge?
- How may a knowledge representation of software systems be created, i.e. manually or semi-automatically?
- How may software system knowledge be integrated?
- How may a software system be linked to its knowledge representation?
- What is the role of ontology and description logic in knowledge based software reengineering?

RQ3: How may software system knowledge be deployed in software reengineering?

- Which software reengineering activities require software system knowledge?
- How may software system knowledge be used in software reengineering

projects?

RQ4: How may tools support to validate the proposed approach be provided?

In order to explore these research questions, a series of research propositions are developed. The underlying proposition of this thesis is:

Ontology and description logic can be used to represent the knowledge of software systems in order to semi-automate some of the manual effort in reengineering and, as a result, improve the efficiency of reengineering projects.

This proposition is tested by developing, integrating and deploying software systems ontology in reengineering projects. A set of propositions is derived from the underlying one:

RP1: Ontology can be used to represent the knowledge of different software system. This proposition can be tested by developing ontology for different software system categories. Different types of system may require different methodologies for ontology development.

RP2: Domain ontology resources are available for ontology based domain-specific software system reengineering. This proposition can be tested by seeking the support of online ontology libraries.

RP3: Software system ontology can be used to semi-automate some manual tasks in software reengineering projects and hence improve their efficiency. This proposition can be tested by developing use cases for an ontology based software reengineering approach.

RP4: There are links between different perspectives of software knowledge within the same system. Integration of those different perspectives will enhance the understandability of existing software systems. This proposition can be tested by integrating different software system ontologies.

1.4 Original Contributions

A knowledge based software reengineering approach is proposed in the context of software reengineering and knowledge representation. It is an application of description logic and ontology to the task of constructing computable models for the software reengineering domain. The following are original contributions:

C1: A novel knowledge based software reengineering framework is developed, aiming to semi-automate the tedious, time consuming and error prone manual software reengineering tasks and thereby improving the efficiency of traditional software reengineering processes.

C2: Methodologies for generating software system ontology are investigated and classified in relation to the different categories of software systems.

C3: A series of practical design principles for building software system ontology is defined to guide and facilitate top-down software system ontology development.

C4: Creating operating system ontology is a novel idea proposed in this study. Operating system ontology is built under principles of software system ontology development. It has become a useful repository for software maintainers and researchers.

C5: Semi-automatic ontology generation methods are also investigated to create software system ontology in a bottom-up manner. Model transformation is the underlying technique supporting those methods.

C6: A description logic based ontology mapping algorithm is developed in this study, which transforms the problem of ontology mapping to the problem of checking satisfiability of logical formulae.

C7: A great deal of effort, including the definition of basic terms and relations in software systems, will be devoted to defining the ontology of software systems.

C8: A set of tools is developed to demonstrate and validate the proposed approach by deploying software system ontology to selected reengineering projects.

1.5 Measure of Success

The overall measure of success of a knowledge based software reengineering approach is how well it supports a successful software reengineering project. The following measures are given to judge the success of this thesis:

- The proposed approach should be able to deal with at least two different kinds of software systems.
- The generated knowledge representation of software systems should be machine readable in order to semi-automate some manual tasks.
- The extracted software system knowledge representation should be reliable enough to perform forward engineering.
- The proposed approach should be capable of realisation. i.e. is it possible to build a practical tool to demonstrate and validate the approach.
- The proposed approach should support the modern computing paradigms such as cloud computing.

1.6 Organisation of Thesis

The rest of the thesis is organised as follows:

Chapter 2 provides a general overview of software crisis, software engineering, software taxonomy and software reengineering, which is the background of this research. It also introduces the basic concepts related to the proposed approach such as model driven engineering, model transformation, knowledge representation, ontology and description logic, etc. Furthermore, a series of related studies, including, operating system modelling, platform-specific software migration and software portability, knowledge based software engineering methods, knowledge based software reengineering approaches, as well as knowledge based software tools is discussed.

Chapter 3 introduces the knowledge based software reengineering approach. An ontology based software reengineering framework is presented. An nntology based software reengineering process is also defined in five steps.

Chapter 4 describes the first two steps of an ontology based software reengineering process. Bottom-up and top-down methods are employed to generate software system ontology. The bottom-up approach is supported by model transformation techniques. Specific model transformation processes are discussed regarding semi-automated ontology generation. A series of operating system ontology development principles is proposed to support the top-down approach including some examples.

Chapter 5 works on software system ontology integration, which is defined as ontology mapping in this study. A description logic based ontology mapping algorithm is presented with examples.

Chapter 6 explores the deployment of software system ontology via different selected use cases. Ontology based software migration and ontology based program comprehension are discussed respectively with two different use cases for each one.

Chapter 7 describes toolset support for the proposed approach. An ontology based software migration toolset and an ontology based program understanding toolset are presented.

Chapter 8 summarises the thesis, draws conclusions and discusses the future work. The research questions are revisited and answered in order to evaluate the proposed approach.

Appendix A is the .owl file of the manually created prototype of RTOS ontology.

Appendix B lists all the related publications written by the author during the PhD study.

Chapter 2 Background and Related Work

Objectives

- To provide an overview of software engineering
- To provide an overview of software evolution and reengineering
- To provide an overview of Model Driven Engineering (MDE)
- To provide an overview of Knowledge Representation (KR) and Knowledge Engineering (KE)
- To review related projects, covering operating system modelling and development, software portability, platform specific software migration, knowledge based software engineering methods, knowledge based software reengineering approaches and knowledge based software tools support

2.1 Software Engineering

2.1.1 Software Crisis

The term 'software crisis' has been used for nearly 50 years to describe the recurring system development problems such as, going over time, going over budget, becoming unmanageable and of poor quality.

Firstly, Brooks [15] suggested that complexity is the cause of software crisis for the following reasons:

- Product flaws, cost overruns, and schedule delays are normally caused by communication difficulties amongst team members.
- Complete understanding of the entire system is almost impossible because of the difficulty of enumerating all the possible states of the program.
- Maintaining conceptual integrity becomes increasingly hard because of the difficulty of attaining an overview of the entire system.
- Potential security backdoors are always left over because of the difficulty of obtaining the structure of the program.
- Side effects are almost inevitable when introducing new features and functionalities.
- Complex functions are difficult to invoke in large systems.
- There is a steep learning curve for new personnel leading to inescapable project delays.

Furthermore, software change is another cause of software crisis. Successful software systems will need to respond to changes in the business, the customer requirements and also hardware and environment changes. Many software systems are constrained by the need to conform to ever-changing environments and systems. There are four main reasons for changing software [76]:

- **Perfection/Enhancement**. Changes are made to improve the software products, such as adding new functionalities, or enhancing system attributes such as performance and usability, etc.
- **Correction**. Changes are made to increase the accuracy or to rectify mistakes in software products.
- **Adaption**. Changes are made to ensure software products keep pace with ever-changing platforms/environments, e.g., operating systems, language

compilers, database management systems and other commercial components.

- **Prevention**. Changes are made to improve the further maintainability and reliability of software products.

2.1.2 Software Engineering

Along with software crisis, there is an exponential increase in the difficulty of designing, implementing and launching the software products. Software academia has been seeking methodologies which handle complexity and improve productivity as well as the quality of the software products.

As one of the most important elements of computer science, software engineering was originally introduced as a solution to "software crisis" [87]. It is defined by IEEE Computer Society's Software Engineering Body of Knowledge as "the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, and the study of these approaches; that is, the application of engineering to software [54]".

Software engineering has the following three components [125]:

- **Software engineering method** provides the methodologies and technologies for designing and building software products including data structures, program architecture, algorithms, coding, testing, and maintenance;

- **Software engineering toolset** is a set of tools that provide semi-automated support for software engineering methods;

- **Software engineering process** defines the process of software engineering method and holds software engineering method and toolset together.

Currently, there are many software engineering approaches, e.g., object-orientation programming (OOP), component-based development (CBD), service-oriented architecture (SOA) and cloud computing, etc. OOP, CBD and SOA are the most used ones in the software industry at present. As emerging and promising computing paradigms, grid computing and cloud computing have attracted increasing interest from software engineering researchers. Software academia has been working on developing relevant methodologies to implement grid computing and cloud computing, which also support distributed development and execution, software reuse, and robustness [52].

More advanced software development paradigms will be available in the near future and therefore evolving software products to adapt and fit those new paradigms will be indispensable.

2.1.2.1 Object Oriented Programming (OOP)

In the real world, people attempt to hide an object's complexities when solving problems. An object contains information and also provides mechanisms to manipulate information without distracting people with it's inner complexity. Similarly, object oriented programming borrows this idea and creates software that contains data and also provides methods to manipulate data without bothering user with the inner complexity of code. Object oriented programming has dominated the way programmers think about solving problems and it has therefore enjoyed enormous popularity since the 1990s. What follows is one of the formal definitions for object oriented programming; it also emphasises a few of the key elements of object oriented programming:

"An ***object-oriented*** *program consists of one or more objects that interact with one another to solve a problem. An* ***object*** *contains state information (data, represented by other objects) and operations (code). Objects interact by sending* ***messages*** *to each other. These messages are like procedure calls; the procedures are called* ***methods****. Every object is an instance of a class, which determines what data the object keeps as state information and what messages the object understands. The* ***protocol*** *of the class is the set of messages that its instances understand."* [35]

The rest of this section will provide a brief overview of object oriented programming by introducing main fundamental concepts and features.

Class is a template for an object, the fundamental structure of an object oriented program, containing data fields and methods to manipulate data. It is a blueprint that constructs software.

Instance is the actual object that is created based on the template of class at run-time and exists in the memory of the computer.

Inheritance is a mechanism that allows one class to share the properties of another by inheriting all state and behaviour of another class.

Encapsulation is a mechanism that allows or disallows access to data fields in an object. In other words, encapsulation conceals the functional details of a class and hides data from public view.

Abstraction is a mechanism that represents an object, showing only essential features and necessary details.

Overloading is a mechanism that provides methods with the ability to automatically adapt to fit different situations.

Polymorphism is a mechanism that allows an object to have different meanings and usages in different contexts. It is described as “many shapes” or “one interface, many implementations”.

2.1.2.2 Component-Based Development (CBD)

Traditional procedural programming views a software product as a linear process. However, this traditional approach is not able to deal with the pressure of building or rebuilding high-quality software in shorter time periods for the following two reasons: (1) code is almost non-reusable, therefore most lines of code will need to be rewritten. (2) functionalities are always distributed throughout the entire application, which makes it difficult to modify and maintain when changes are required. Hence there is an increasing need for a flexible and reusable programming approach for accelerating software development and enhancing the productivity and innovation of developers [109].

Component-based development borrows ideas from the manufacturing industry and emphasises the separation of goals by building software products with different components that take into account the wide range of functionalities that the software has to provide. Those components are normally developed as black boxes, which could be software packages, web services, or modules which implement a set of functionalities. Components are semantically related and can communicate with each other via predefined interfaces. To modify or maintain a component based software product is simply a matter of modifying or replacing relevant components without affecting the entire product. Component-based development comes with all the qualities that are desperately needed to replace traditional procedural programming paradigms, i.e. reuse, flexibility, scalability, better quality, cost reduction and faster time-to-market [47].

2.1.2.3 Service Oriented Architecture (SOA)

As defined by the World Wide Web Consortium (W3C), web service is “a set of components which can be invoked, and whose interface descriptions can be published and discovered [115].” SOA is software architecture developed for sharing functionalities in a widespread and flexible way. Web services are software components capable of performing a task to support machine-to-machine interaction over a network. Web Service Description Language (WSDL) is employed as a standard language to describe the functionalities and interfaces of web services. Users will need to connect to the Universal Description Discovery and Integration (UDDI) centre to search for their required web services. And Simple Object Access Protocol (SOAP) is used to transfer the requirement for information and to receive the real service. Hence, SOA is described as the “find, bind and execute” paradigm. There are six entities configured together to support SOA, namely, service consumer, service provider, service registry, service contact, service proxy and service lease [48]. Service consumer finds the service in the registry, binds to the service and then executes the functionalities of the service. Service provider is the service which accepts and executes the request from the consumer. Service registry is the directory on the network, containing all the available services. Service proxy is given by the provider to facilitate finding the contract and reference and then executing the service function. Service lease is like a contract in which the registry grants the consumer a valid time period. Implementing a service-oriented architecture can involve writing a web service, writing an application which uses web services, or both.

2.1.2.4 Cloud Computing

Cloud computing is one of the future trends of software engineering research, which implies a service-oriented architecture (SOA) aiming to reduce IT overheads by providing more a flexible and economic usebility for software end users. In essence, cloud computing provides a set of IT services from software applications to hardware devices, which are transparent to the end users. End users do not need to own any IT resources, but consume resources as services and pay for these as they use them. It could involve many related research areas such as distributed computing, grid computing, utility computing, web services, software as a service (SaaS), platform as a service (PaaS) etc. On one hand, researchers [127] are trying to build a layered classification, in which cloud computing research could be divided into five layers in a top-down manner, namely, cloud application layer (e.g. Software as a Service (SaaS)), cloud software environment layer (e.g. Platform as a Service (PaaS)), cloud software infrastructure layer (e.g., Virtualisation, Infrastructure as a Service (Iaas), Data-Storage as a Service (DaaS) and Communication as a Service (CaaS)), software kernel layer (e.g., Hardware as Service (HaaS)). And on the other, it is also important to compare cloud computing with other existing computing paradigms. Mei et al. [81] have done a qualitative comparison between cloud computing, service computing and pervasive computing from different aspects. They discovered three notable similarities among these computing paradigms, namely: I/O similarities between cloud computing and service computing; storage similarity between cloud computing and pervasive computing; and calculation similarities among all three paradigms.

2.1.3 Formal Methods

The term formal methods is used to refer to the techniques and tools based on sound mathematics and formal logic [122]. It can assure different forms and levels of rigor. On the one hand, most rigorous formal methods are equipped with fully formal specification languages with a precise semantics. On the other, English specifications with occasional mathematical notation embedded support least rigorous formal methods. Liu et al. [77] state that a formal method should consist of the following essential components, namely, a semantic model that defines the precise semantics of all terms and formulae with a sound mathematical/logical structure, a specification language that describes the intended functionalities and behaviours of the system, a verification system/refinement calculus that allows property verification and specification refinement, a development guideline that instructs how to use the formal method, and a tool that performs various tasks such as syntax checking and mathematical proving, etc.

In terms of applications of formal methods, software engineering research will benefit from the following aspects: (1) using formal methods to produce the specifications for software development; (2) using formal methods to produce the formal specifications for correctness check and system verification. Baumann [9] argues that reverse engineering methods must be based on a sound mathematical foundation in order to achieve the correctness and efficiency. In the area of reengineering, formal methods have also been put forward as means to

- formally specifying and verifying existing systems;
- introducing new functionalities;
- automatically generating program code; and
- improving systems design techniques [77].

Formal methods can normally be divided into four different categories in relation to purpose and usage, namely, formal specifications, formal proofs, model checking and abstraction [122].

- Formal specifications describe the external behaviour of the system based on two different approaches, i.e., property oriented approach and model oriented approach.
- Formal proofs are complete and convincing arguments for validity of some property of the system description.

- Model checking is to determine if the given finite state machine model satisfies requirements expressed by logical formulae.
- Abstraction is to simplify and ignore irrelevant details.

Regarding the methodologies, formal method can be classified into five different types, i.e, model-based formal method, logic-based formal method, algebraic formal method, process algebra formal method and net-based formal method [125].

- Model-based formal method models the system by explicitly defining states and operations that transform the states.
- Logic-based formal method is used to describe low level specifications, temporal and probabilistic behaviours of the system.
- Algebraic formal method defines system operations by relating the behaviour of different operations.
- Process algebra formal method represents system behaviour by constraints on all allowable observable communication between processes.
- Net-based formal method specifies systems by graphical notations.

2.1.4 Domain Engineering

Domain is defined as "an area of knowledge, which includes the knowledge of how to build software systems or parts of software systems in that area [26]." There are two different categories of domains, namely, horizontal domain and vertical domain. The horizontal domain category describes different parts of systems with regard to functionalities, e.g., database system, workflow system and GUI, etc. The vertical domain category contains different types of systems with regard to applications, e.g., payment system, human resource system, inventory management system and order processing system, etc.

Domain engineering, also known as product line engineering, is defined as "the entire process of reusing domain knowledge in the production of new software systems [53].” In other words, domain engineering is used to form and manipulate a repository of reusable assets of domain specific systems or system components by collecting, organising and storing past experience and knowledge [53]. Not only does domain engineering support new system development, it also supports the establishment, maintenance and evolution of existing systems. Capturing well-structured domain knowledge will contribute significantly to reverse engineering projects [26]. Domain knowledge in the form of reusable assets can facilitate program understanding by reducing the complexity of the program code.

Domain Analysis, Domain Design, and Domain Implementation are the three main processes of domain engineering [26]. Domain Analysis identifies and defines a set of reusable assets for the domain specific systems. Domain Design establishes a common architecture for the domain specific systems. Domain Implementation implements the reusable assets such as reusable components, domain-specific languages, generators, and a reuse infrastructure.

2.1.5 Software Taxonomy

In terms of software engineering, different types of software systems require different methodologies to design and develop due to the different functions and features that software systems may have. Ideally, any paper published containing a practical or empirical study should specify which type of software systems it applies to. In order to carry out empirical software research in a systematic way, software taxonomy will be needed. Software taxonomy could provide software engineering research with the following three advantages [36]:

- To provide contexts for empirical studies and to facilitate exploring the applicability of those studies.
- To make the methods more easily reusable by mapping to categories within software taxonomy.
- To assist software engineering education with a more systematic and structured

course design.

Unfortunately, however, there are only a few published software system taxonomies or systematisation available for software engineering research. The ACM computing taxonomy is one of the well-known taxonomies, but it may not be appropriate software system taxonomy as it describes the categories of computer science research. The ACM computing taxonomy does contain parts of the categories of software system application domains though. Another example of software system taxonomy is the classification system used by open-source community such as SourceForge and GoogleCode, etc. SourceForge mainly classifies their software systems based on their application domains, while Google Code approaches application domains slightly differently by relying mostly on non-hierarchical tagging of applications. However, both sites have provided excellent coverage on different software systems. In addition, the research on Problem Frames [57], i.e. the type of problem a software system solves, also suggests some important high-level categories of software systems.

After reviewing and comparing most of the published software taxonomy, this research will reference the one proposed by Forward and Lethbridge [36]. The top levels of their taxonomy include four categories, namely, data-dominant software, system software, control-dominant software and computation-dominant software. The data-dominant software category includes four subcategories which are consumer-oriented software, business-oriented software, design and engineering software and information display and transaction entry. The system software category includes eight subcategories which are operating systems, networking/communications, device/peripheral drivers, support utilities, middleware and system components, software backplanes, servers and malware. Control-dominant software includes four subcategories namely, hardware control, embedded software, real time control software and process control software. Computation-dominant software includes five subcategories namely, operations research, information management and manipulation, artistic creativity, scientific software and artificial intelligence. The more detailed taxonomy is listed in table 2-1.

Taking into account the software taxonomy described below and the features of different types of software system, this research will select data-dominant software and system software as main subjects.

- Data-dominant software
  - Consumer-oriented software
    - Communication and information (email, web browsers and FTP, etc.)
    - Productivity and creativity (word processors, spreadsheets and calculators, etc.)
    - Entertainment and education (photo/video management, media players, games and training/courseware, etc.)
    - Personal management (personal finance, tax preparation and health monitor, etc)
  - Business-oriented software
    - Strategic and operations analysis (risk analysis, financial analysis, workforce management and payroll management, etc.)
    - Corporate management (restaurant management, sales management and hospital management, etc.)
    - Information management and decision support systems (Data warehousing, Expert systems and help desk system, etc.)
    - Transaction processing (accounting, payroll, inventory management, bank transaction processing, etc.)
  - Design and engineering software
    - Development environment (implementation tools, version control and development environment plug-ins, etc)
    - Compilers
    - Automatic code generation
    - Database
    - CAD/CAE
    - Modelling/CASE
    - Testing
  - Information display and transaction entry
    - Information resources (maps and contact management etc.)
    - Standalone application for displaying information
    - Web applications/services (search engines, website content management and e-commerce etc.)

- System software
  - Operating systems
    - Accessibility
    - Administrator software
    - Emulation
    - Game console OS
    - Virtual machines
    - Kernels
  - Networking/Communications
  - Device/Peripheral drivers
  - Support utilities
    - Anti-virus
    - firewalls
    - compression
    - disk maintenance
    - screen capture
    - software installer
  - Middleware and system components

    - Database servers
    - Virtual machines
    - interoperability infrastructures
  - Software Backplanes
  - Servers
    - Email servers
    - Proxy servers
    - FTP servers
    - IM servers
  - Malware
    - Spyware
    - viruses

- Control-dominant software
  - Hardware control
    - Firmware
    - device control
  - Embedded software
  - Real time control software
  - Process control software

- Computation-dominant software
  - Operations research
    - Computer science hard problems
    - Simulation software
  - Information management and manipulation
    - Inventory control
    - Sales forecasting
    - search engine processing
  - Artistic creativity
    - photo manipulation
    - audio recording
    - music composition
  - Scientific software
    - Idle-time data analysis
    - simulation software
    - signal analysis software
    - computer vision

Table 2-1 Software Taxonomy [36]

2.2 Software Evolution and Reengineering

2.2.1 Software Change and Evolution

Every software system is subject to changes as long as it is still in use. The activities of software change can be classified into three categories: maintenance, modernisation, and replacement [125].

- **Software maintenance** is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment. Maintenance is an incremental and iterative process in which small changes are made to a system without major structural changes.

- **Software modernisation** involves more extensive changes than maintenance but conserves a significant portion of the existing system. These changes often include restructuring the system, enhancing functionality, or modifying software attributes. Software modernisation falls between the two extremes of system replacement and continued maintenance.

- **Software replacement** requires rebuilding the system from scratch and is resource intensive. Replacement is carried out when modernisation is not possible or cost-effective. Systems can be replaced incrementally where modernisation works as a preparatory step before beginning an incremental replacement effort.

Software systems need to continuously evolve in order to cope with changes. Software Evolution is defined as "the process of conducting continuous software reengineering," i.e. software reengineering is a single change cycle, while evolution will carry on indefinitely – software evolution is repeated software reengineering [125]. To a large extent, as a practical solution to the problem of evolving software systems, software reengineering has the potential to ease software crisis.

2.2.2 Laws of Software Evolution

The Laws of Software Evolution were introduced by M. Lehman when he was doing research to clarify classification schemes distinguishing three types of programs S, P and E [71, 72]. S-type program presents the program which can be formally specified. P-type program stands for the category which is an iterative process that cannot be specified. E-type program is embedded in the real world and is a computer program that solves a problem or implements a computer application in the real world domain [70].

Law	**Description**
I. Continuing Change	An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.
II. Increasing Complexity	As a program is evolved its complexity increases unless work is done to maintain or reduce it.
III. Self Regulation	The program evolution process is self regulating with close to normal distribution of measures of product and process attributes.
IV. Conservation of Organisational Stability	The average effective global activity rate on an evolving system is invariant over the product life time.
V. Conservation of Familiarity	During the active life of an evolving program, the content of successive releases is statistically invariant.
VI. Continuing Growth	Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.
VII. Declining Quality	E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.
VIII. Feedback System	E-type Programming Processes constitute Multi-loop, Multi-level Feedback systems and must be treated as such to be successfully modified or improved

Table 2-2 Lehman's Laws of Software Evolution [69]

Table 2-2 [69] lists all the eight laws. Law I – Continuing Change states that "An E-type program that is used must be continually adapted else it becomes progressively less satisfactory”. Accordingly, methodologies must be developed for evaluating, controlling and making changes.

2.2.3 Software Reengineering

Software reengineering is a form of modernisation that improves capabilities and/or maintainability of a legacy system by introducing modern technologies and practices. The purpose of software modernisation and reengineering is both to utilise existing software to take advantage of new technologies and to enable new development efforts to take advantage of reusing existing software, which has the potential to improve software productivity and quality across the entire life cycle. Software reengineering is significant in software evolution. System replacement is expensive, while reengineering is much cheaper. Moreover, the risk of losing any critical information which is embedded in legacy assets can be reduced by reengineering. The term “Software reengineering” has various valid definitions which represent different point of views and perspectives. Chikofsky and Cross [21] define it as “the examination and alteration of a subject system to reconstitute it in a new form and subsequent implementation of that form”. Arnold [3] defines software reengineering as “an activity that improves one’s understanding of software, or, prepares or improves the software itself for maintainability, reusability or evolvability”. Reengineering is also the general term for activities during corrective, adaptive, perfective or preventive software maintenance. Software Evolution is the process in which continuous software reengineering is conducted. In other words, software evolution is repeated software reengineering [125].

Bachman [8] introduced a software reengineering cycle chart to better understand the process of software reengineering. (Figure 2-1 [125]) Software reengineering is a combination of reverse engineering and forward engineering. The process of reverse engineering on an existing application starts with operation, i.e., defining the existing applications. Then the definition of the existing system will be raised to a higher level from operation to implementation, and then to specification and finally to requirements. The final result will then be validated and enhanced in order to be used in the forward engineering phase. The new application will be built on the result of reverse engineering in a reverse process of reverse engineering. The new application will also become an existing application at the moment it goes into production. Hence the reengineering cycle has been formed.

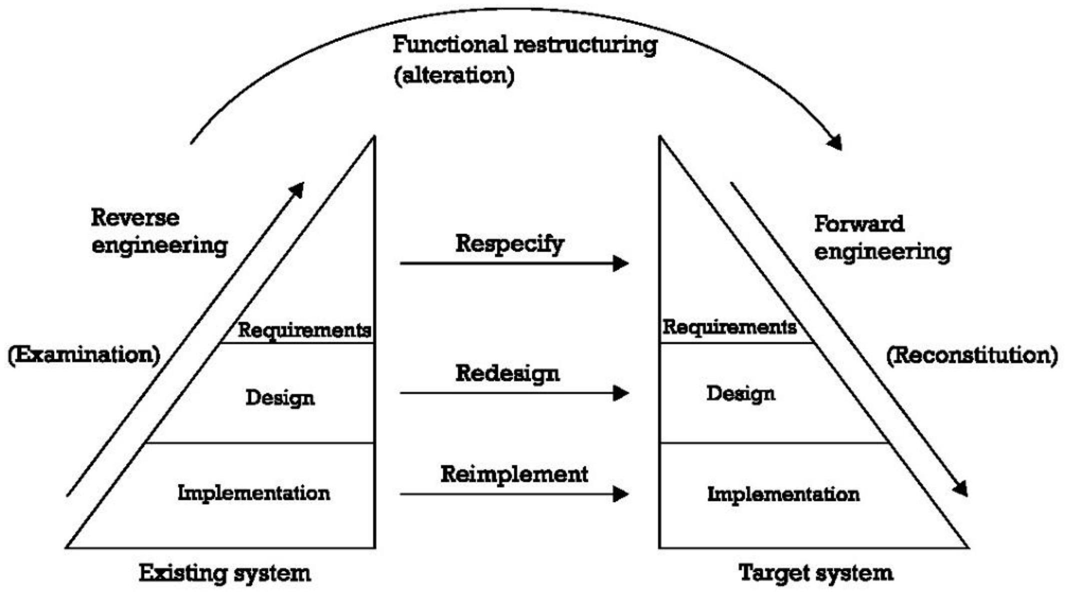


Figure 2-1 Software Reengineering Process [125]

2.2.4 Basic Concepts and Related Terms

The proposed research will be related to the following terms from the software reengineering glossary [125] in the domains of software reengineering.

Legacy system describes an old system which remains in operation within an organisation [120]. Organisations are in fear of keeping their legacy systems, since maintaining them is a significant drain on the organisation's resources. They are also afraid of replacing them. A major reason is that those legacy systems are enormously valuable assets. Having stood the test of time, they provide the most accurate statement of current business rules. However, all software systems will inevitably become legacy systems.

Reverse engineering is to analyse an existing system in order to a) identify the components and their interrelationships with the system, and b) represent the system in another or higher abstracted form.

Design recovery is to add domain knowledge and external information, etc. to the recreated design abstraction in order to obtain a meaningful higher-level abstraction of the subject system. Code, existing documentation, human experience and knowledge of the problem domain will be used together to achieve the goal.

Program understanding (program comprehension) is to understand the software system at source code level. The cognitive science of human mental processes plays a very important role in program understanding.

Restructuring is to transform the software system from one representation form to another one at the same abstraction level. The transformation should preserve the software system's external behaviour, e.g., functionality and semantics, etc.

Reverse specification is to abstract the specification from the source code or existing documentation. The abstracted specification will become the final product of reverse engineering and the input of forward engineering.

Program Transformation is the act of changing one program into another. The term program transformation is also used as a formal description of an algorithm that implements program transformation. The languages in which the program being transformed and the resulting program are written are called the source and target languages respectively.

Model Transformation is a mapping of a set of models onto another set of models or onto themselves, which can be broken into two broad categories: model translation and model rephrasing. In the former, a model is transformed into a model of a different language, and in the latter, a model is changed in the same modelling language.

2.3 Model Driven Architecture (MDA)

2.3.1 Model Driven Architecture

Model Driven Engineering (MDE) has been developed as a solution to handle the increased complexity of software systems [85]. As one of the MDE initiatives, the Model Driven Architecture (MDA) paradigm was introduced in 2001 and defined by Object Management Group (OMG) as an approach to addressing the increasing complexity in software development. The fundamental idea of MDA is to separate business and application logic from underlying platform technology. It shifts the focus of software development from coding to modelling. The MDA approach will penetrate the complete software development lifecycle, i.e., system analysis and design, programming, testing, component assembly, deployment and maintenance. The three main primary goals that MDA are trying to achieve are portability, interoperability and reusability.

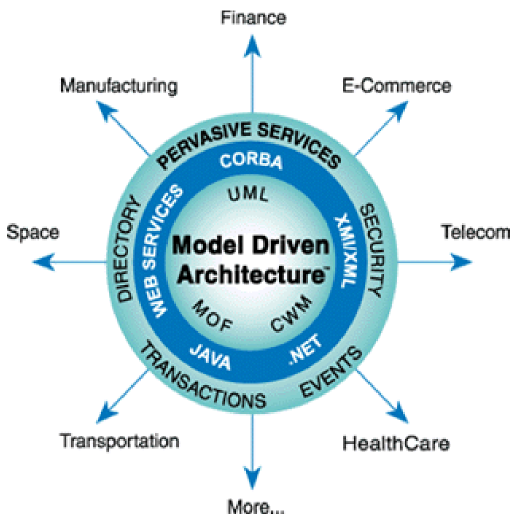


Figure 2-2 OMG Model Driven Architecture [92]

Figure 2-2 displays the MDA, which is open, vendor-neutral. The centre of the circle presents the core of the MDA, which is OMG's modelling standards: Unified Modelling Language (UML), Meta Object Facility (MOF) and Common Warehouse Metamodel (CWM). The core models are platform-independent and they are built to represent business functionality and behaviour using those modelling standards. Core models are then realised using any major open platforms such as CORBA, Java, .NET, Web Services and XMI/XML, etc. Therefore, the core of an application is insulated from technology, which enables interoperability both within and across platform boundaries.

Since business and technology are no longer tied to each other, they will be able to evolve respectively, i.e., business logic reflects new business needs, while technology keeps pace with new techniques.

The MDA specification emphasis is on different levels of models, including, Computational Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM). CIM is the most abstract model in MDA. It represents the business context business requirement without any computational complexities. PIM is refined CIM which describes the business functionalities and behaviour of the application but in a technology independent manner. PSM describes how this system can be implemented using a given technology and contains all required information in relation to a specific platform. The first step in implementing an MDA approach is to construct a PIM expressed via UML. A platform specialist will then transform PIM to PSM by adding a specific platform implementation. The PSM will represent both business and technical run-time semantics of the software products, in which the business logic should be consistent to the one expressed by PIM. The next step will be code generation. In a mature MDA environment, code generation should be more complete and automatic. Interface definition files, component definition files, source code and configuration files will be generated in this step. The automatic transformation process such as PIM to PSM and code generation can reduce development costs and improve software quality. The automation requires that MDA models be machine-readable and able to be automatically transformed by MDA tools into schemas, code skeletons, test harnesses, integration code, and deployment scripts for various platform [65]. The MDA models must conform to the following definitions in order to support automated MDA approaches [65].

Model *is a description of (part of) a system written in a well-defined language.*

Well-defined language *is a language with formal form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer.*

The MDA is defined and trademarked by the OMG. The OMG has defined a number of modelling languages that are suitable to write either PIM or PSM. UML is the most well-known and widely used one. The Object Constraint Language (OCL) is a query and expression language for UML. In the context of MDA, UML defines what models are considered to be valid. It describes the primitive model elements and how these elements can be combined together to construct a valid model. The primitive model elements form the representation of the different aspects and concepts of the problem domain. Formal modelling language has formal syntax and semantics. Formal syntax describes models in a precise and unambiguous way, while formal semantics assigns a semantic meaning to models.

MDA provides a framework which includes the following major elements [65]:

- A model which is a description of a system (PIM and PSM)
- A model which is written in a well-defined language
- A transformation definition which describes how a source model can be transformed into a target model
- A transformation tool which could perform model transformation (semi-) automatically

2.3.2 Meta Object Facility (MOF)

Most graphical modelling languages appear to be intuitional rather than formal. However, the rigor of the method is still very important in terms of interoperability. Defining a meta-model is one of the ways to improve the rigor. “Meta-model is a diagram, usually a class diagram, which defines the concepts of the language [37].”

Object Management Group (OMG) has defined the standardised metadata management framework Meta Object Facility (MOF) to enable the interoperability of model driven systems [91]. UML has been specified in this architecture. The notion of a “model” is the central concept in MOF. MOF defines a framework that supports building repositories of metadata (e.g., models) described by meta-models.

Figure 2-3 [45] shows a typical four-layered modelling architecture based on meta-model and MOF. MOF specification defines the most abstract layer M3 which provides an abstract language and a framework for specifying, constructing, and managing meta-models. Meta-model reside on layer M2 which provides meta-data to construct the model. Layer M1 is for the models which represent the software system and real life.

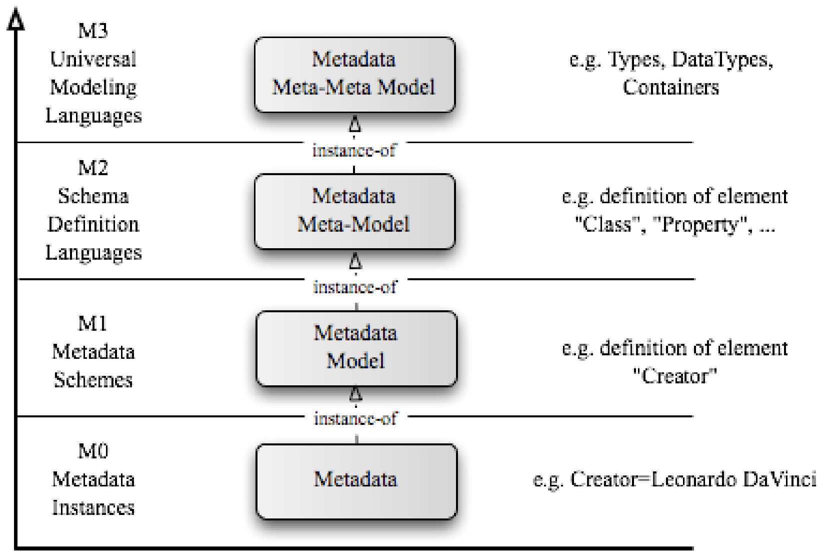

M3
Universal Modeling Languages
Metadata Meta-Meta Model
e.g. Types, DataTypes, Containers
instance-of
M2
Schema Definition Languages
Metadata Meta-Model
e.g. definition of element "Class", "Property", ...
instance-of
M1
Metadata Schemes
Metadata Model
e.g. definition of element "Creator"
instance-of
M0
Metadata Instances
Metadata
e.g. Creator=Leonardo DaVinci

Figure 2-3 MOF Meta-Levels Hierarchy [45]

XML Meta-Data Interchange (XMI) [89, 90] is used as a standard common model exchange format which enables developers to achieve the same understanding and interpretation when exchanging models via different technologies and tools. XMI is an XML standard for exchanging UML models. XML schema conversion rules indicate how the UML model can be converted to an XML document.

2.3.3 Modelling Maturity Levels

In order to make an assessment of the MDA approach, Modelling Maturity Levels (MML) [119] is proposed. MML is a classification system, which characterise the role of modelling in software development projects with five different levels.

- Level 0, the specification is kept in the software developers’ minds and there is no written down specification.

- Level 1, the specification is written down with natural language text in one or more documentations.
- Level 2, the specification is mainly composed of one or more natural language documentations and few models to explain the main structure of the system.
- Level 3, the specification is mainly composed of one or more models with natural language text as an additional supplement to explain more detailed information.
- Level 4, the specification is mainly composed of one or more precise models with natural language text as an additional supplement to explain more detailed information. This is the first level at which a model can be understood by a machine. The models at this level are precise enough to have a direct link with the actual code. The natural language texts play the same role as comments in source code.
- Level 5, the specification is composed of one or more precise models to explain more detailed information. Models are a complete, consistent, detailed, and precise description of the system, which are good enough to enable complete code generation.

2.3.4 Five Technical Space (TS)

A Technological Space (TS) *is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities.* [67]

In [67], five different technological spaces are discussed, namely, programming languages concrete and abstract syntax, ontology engineering, XML-based languages, data base management systems (DBMS) and MDA. Figure 2-4 presents an overview of five TSs. It illustrates that each TS is defined by two basic concepts, i.e., syntax TS is defined by program and grammar, XML TS is defined by document and schema, MDA TS is defined by model and metamodel, ontology engineering TS is defined by ontology and top-level ontology and DBMS TS is defined by data and schema. The figure also shows that all the TSs are related to each other, there is no isolated TS. The existence of various TSs means that given a system, one has to choose the TS that will be most appropriate for the expression of a model or a given usage.

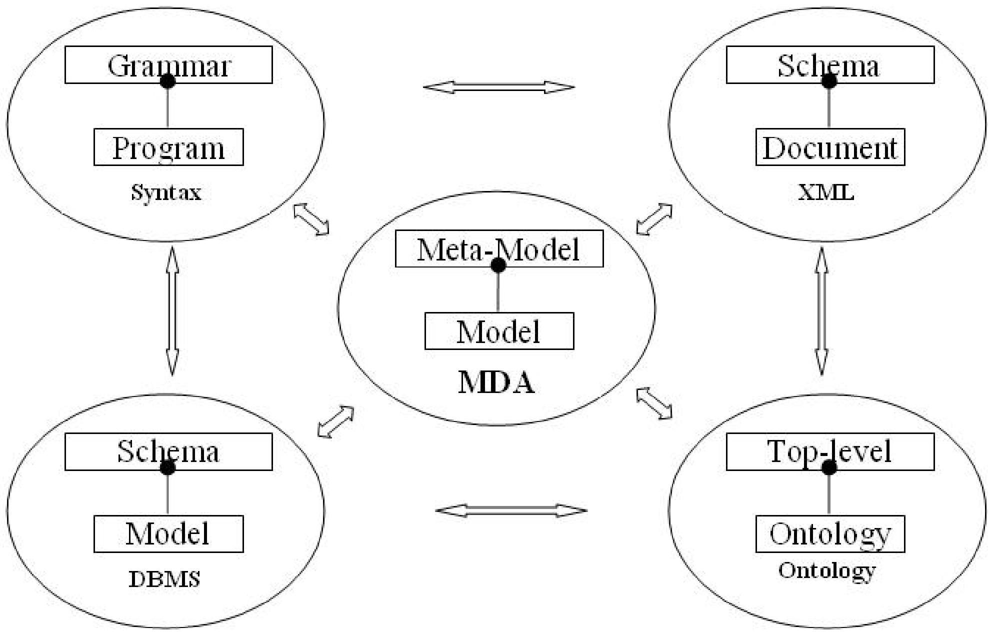


Figure 2-4 Five TSs and Their Links [67]

2.3.5 ATL Model Transformation

2.3.5.1 Model Transformation

Model transformation is defined by Kleppe et al. [65] by a set of terms.

"A transformation is the automatic generation of a target model from a source model, according to a transformation definition.

A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language.

A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language."

Model transformation [65] is one of the most important components of Model Driven Engineering (MDE). It can be used in a wide range of activities such as automatic code generation, model simulation, model synthesis, model evolution, model execution, model refactoring, model translation, model checking and model verification.

In general, model transformation could be divided into two different categories [82]: endogenous versus exogenous transformations and horizontal versus vertical transformations. The former category emphasise the language that is used to express the source and target models, i.e. endogenous deals with the transformations between models expressed in the same language (e.g. model optimisation and model refactoring), while exogenous transformations are between models expressed by different languages (e.g. code generation and model translation). The latter one is looking at the abstraction level of source and target models. The horizontal transformations indicate that both source and target models are at the same abstraction level (e.g. model refactoring and model translation), while vertical transformations suggest that source and target models are at different level of abstraction (e.g. refinement). Because of the importance of model transformation in MDE approaches, writing model transformation definitions is believed to be a common task in future software development.

There are a wide variety of existing model transformation languages, many of which originate from academy (e.g. ATL, Kermeta, Tefkat and SiTra, etc.) and others have emerged from industry (e.g. QVT (Query/View/Transformation) specification which is compatible with MOF and UML. The transformation languages can also be divided into different categories. The first category is based on whether the languages rely on a declarative or an imperative specification. Declarative languages are easier to write and understand by software engineers as they focus on defining a mapping between the source and target models. Imperative languages specify the steps to derive the target models from the source models. The declarative languages focus on what to transform while the imperative languages focus on how to transform. (e.g. the QVT specification has two different types: QVT Relational and QVT Operational.)

The second category of transformation languages is based on the form of the languages, namely, textual and visual. Textual transformation languages use textual description to specify the transformations while visual transformation languages specify transformations in a visual way.

A software engineering vocabulary is provided below in terms of model transformation activities.

Automatic code generation is one of the main underlying technologies supporting the MDA approach. In MDA based software development the platform independent model (PIM) will be transformed into the platform-specific model (PSM), and then source code will be automatically generated from PSM.

Model extraction is the reverse process of code generation, which extract models from source code.

Model translation is transforming a model into an equivalent model expressed by another modelling language.

Model simulation/execution is used to validate models by simulating/executing them.

Model checking/verification/validation is to check whether the models conform to the requirements.

2.3.5.2 Overview of ATL Model Transformation

The model transformation language used in this study is called ATL [62] (ATLAS Transformation Language) and it is developed as a part of the AMMA (ATLAS Model Management Architecture platform). It is a mix of declarative and imperative transformation languages.

Figure 2-5 is an overview of the ATL transformational approach. In this diagram, Ma is the source model and Mb is the target model. Model transformation is defined in mma2mmb.atl by the ATL language. MMa and MMb are metamodels. Ma conforms to MMa and Mb conforms to MMb. Model transformation definition mma2mmb.atl conforms to ATL. MMa, MMb and ATL conform to MOF.

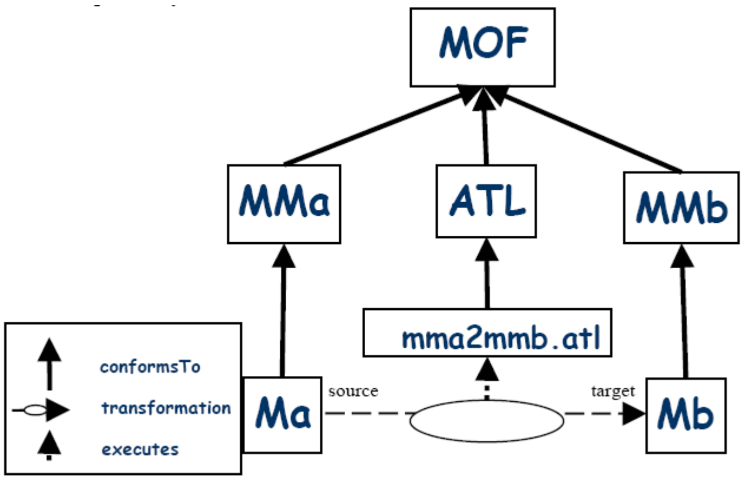


Figure 2-5 Overview of ATL Transformational Approach [62]

2.3.5.3 Transforming Models with ATL

Transformation definition in ATL includes a header section, import section, a number of helpers and transformation rules. The header section contains the name of the transformation and declares the source and target models. The term helper originates from OCL specification, in which operation and attribute helpers are defined. Operation helpers in ATL are used to navigate the source models. Attribute helpers are used to decorate source models before executing model transformation. The basic construct to express the transformation logic is the transformation rule, which can be either a declarative rule or an imperative rule. Declarative rules are also called matched rules in ATL. Matched rules have two parts, namely source pattern and target pattern. The source pattern is a set of matches in source models specifying a set of source types and a guard. Target pattern specifies the target type and a set of bindings indicating initial value. There are three different kinds of matched rules, which are standard rules, lazy rules and unique lazy rules. Standard rules are applied once for every match found in the source models. Lazy rules are triggered by other rules more than once, but unique lazy rules can only be triggered once. Matched rules can be inherited as a mechanism for specifying polymorphic rules. However, it may become difficult to specify pure declarative rules for complex source or target domains. Therefore, imperative rules are also introduced in ATL. Imperative rules allow native operation calls, and offer an imperative part in the transformation language.

2.4 Knowledge Representation (KR)

2.4.1 Knowledge Representation and Knowledge Engineering

Artificial Intelligence (AI) is the research which designs and develops intelligent computer systems to perform tasks that require human intelligence [104]. Knowledge Representation (KR) was originally developed as a branch of AI research. Currently, most advanced software products will be able to perform some AI task such as information retrieval, resource allocation, speech recognition, stock analysis, circuit design, virtual reality and language translation. Consequently AI related techniques like knowledge representation have been integrated into many other research fields, e.g., database system and object-oriented system.

As a multidisciplinary subject, knowledge representation includes three fundamental elements, namely, ontology, logic and computation. Ontology is the study of existence; logic defines formal structure and studies inference with predefined rules; computation is the main part in software products that supports AI. These fundamental elements are strongly linked and interact with each other. Ontology makes sure that the terms and symbols are well and clearly defined. Logic supports knowledge representation for determining the redundant or the contradictory terms. Computation applies both ontology and logic to computer systems.

Knowledge Engineering (KE) is the study that constructs computable models in relation to ontology and logic to solve some practical problems in the different application domains [104]. The features and purpose of knowledge engineering has distinguished it from either mathematics or empirical sciences as a branch of engineering. On the one hand, pure mathematics may define incomputable and even infinite structure and mathematics may not need an application domain. Empirical sciences make computable predictions about the domain. But they may not need any purpose other than the pursuit of knowledge. However engineering uses science and mathematics for the purpose of solving practical problems in the different domain. Knowledge engineering is therefore defined as the branch of engineering that obtains knowledge about some subject and transforms it to computable form for some purpose [104].

2.4.1.1 Principles of Knowledge Representation

Three experts in knowledge representation, Randall Davis, Howard Schrobe, and peter Szolovits, wrote a critical review and analysis of the state of the art [27]. They summarised their conclusions in five basic principles about knowledge representations and their roles in artificial intelligence.

Principle	**Description**
Principle 1	A knowledge representation is a surrogate.
Principle 2	A knowledge representation is a set of ontological commitments.
Principle 3	A knowledge representation is a fragmentary theory of intelligent reasoning.
Principle 4	A Knowledge representation is a medium for efficient computation.
Principle 5	A knowledge representation is a medium of human expression.

Table 2-3 Principles of knowledge representation [104]

Table 2-3 lists five principles of knowledge representation [27, 104]. Principle 1 declares that a computational model is a surrogate for some real or hypothetical system. In Principle 2, the ontological commitments are determined by the types of variables in the knowledge representation. The procedural loop, the Description Logic formula, and the forward-chaining rule illustrate three different strategies for reasoning in Principle 3. Principle 4 shows that both the procedural and the declarative approaches can be transformed to a computable form. Principle 5 inters that since knowledge engineers must work with experts in other fields, they must be able to communicate with them in languages and notations that avoid the jargon of AI and computer science.

2.4.2 Ontology

2.4.2.1 An Overview of Ontology

Ontology is defined as "a formal, explicit specification of a shared conceptualisation [42]." The term conceptualisation indicates an abstract model of the real world with identified relevant properties. Explicit means that the model is explicitly defined. Formal implies that the model is machine-readable. Share reflects consensual knowledge that is accepted by a group [42]. Neches et al. [88] gave another definition, focused on the form of an ontology: "an ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary."

Different knowledge representation methods exist in the context of the formalisation of ontologies, each of which contains different components. However, they share the following main components:

- **Classes** represent a set of concepts which share similar features. Classes are usually organised in a structured hierarchy through which inheritance mechanisms can be applied.
- **Relations** represent associations among concepts, which are formally defined as any subset of a product of n sets, i.e., $R \subset C1 \times C2 \times ...Cn$. Ontologies usually define binary relations with two arguments, namely, the domain and the range. Binary relations are sometimes used to express concept properties.
- **Instances** represent individuals in ontology.

Similar to software engineering, studies of the ontology development process, ontology life cycle, design principles, methodologies for building ontologies, ontology languages and ontology tools have constructed a new research area – ontology engineering. Ontology development process and design principles define a systematic guideline for the new users to effectively develop ontology. The ontology life cycle manages the life cycle of ontology and emphasises the reuse and integration. Methodologies for building ontologies seek mechanisms which could generate ontology automatically. Ontology can formally be described by ontology language and is designed for use by applications which process the content of information instead of just presenting information to humans. To retrieve the information from the ontology, the ontology query languages will also be needed. The basic theory of such query languages can be divided into two mechanisms: RDF-based query and Logic/Rule-based query. RDF-based query is based on matching RDF triple notation with RDF graph, e.g., SPARQL [116], while Logic/Rule-based query is based on reasoning services provided by logic and rules, e.g., DIG interface [31] and SWRL [51]. There are many ontology related tools available to facilitate ontology engineering, mainly ontology editors. As one of the most widely used, Protege [105] has been developed by Stanford Medical Informatics (SMI) at Stanford University. It is an open source, standalone application with an extensible architecture. The core of its environment is the ontology editor and it also holds a library of plug-ins that will add more functionality to the environment.

2.4.2.2 Principles for the Design of Ontologies

It is believed that quality ontology will have the following features: clarity, extendibility, coherence, minimal encoding and minimal commitments. The following five principles may be concluded based on the features which guide a quality ontology design [104].

- **Principle 1:** Formal axioms and a complete definition (defined by necessary and sufficient conditions) are preferred over a partial definition (defined by only necessary or sufficient conditions).
- **Principle 2:** New terms should be defined by extending the existing vocabulary without revision of the existing definitions.
- **Principle 3:** Ontology should be coherent, that is, inferences should be consistent with the existing definitions.
- **Principle 4:** The conceptualisation should be specified at the knowledge level

without relying on any particular symbol-level encoding.

- **Principle 5:** Ontological commitment should be minimised by specifying the weakest theory and defining only essential terms.

2.4.3 Description Logic

2.4.3.1 An Overview of Description Logic

Description Logic (DL) describes domain through concepts (classes), roles (relationships) and individuals. It is a family of logic based knowledge representation formalisms. The evolution history suggests that description logic was originally known as terminological systems, in which representation language was used to establish the basic terminology of modelling domain. Subsequently, representation language was replaced by concept language which contains a set of concept-forming constructs. In more recent years, attention was further moved towards the properties of the underlying logical systems, description logics then formed [7].

A typical description logic knowledge base can normally be separated into two parts – a TBox (Terminology box) which is a set of axioms in the form of terminology describing the structure of domain (i.e. schema) and an ABox (Assertional knowledge box) which is a set of axioms describing a concrete situation (i.e. data). Terminology in TBox is also known as vocabulary which contains concepts that denote sets of individuals and roles that present binary relationships between concepts. Intensional knowledge is stored in TBox in the form of the declarations that describe general properties of concepts. TBox usually has a lattice-like structure because of the subsumption relationships among the concepts. On the other hand, extensional knowledge, also known as assertional knowledge, is stored in ABox. In a description logic knowledge base, TBox is usually not changing. On the contrary, ABox is usually contingent and therefore subject to occasional or constant change [7].

A description logic based knowledge representation system provides the ability to set up a knowledge base and to reason about its content as well as manipulate it. Elementary descriptions are atomic concepts and atomic roles. Complex descriptions can be inductively built on them by concept constructors. Description languages can be distinguished by the constructors they provide. The basic description language is AL (Attributive language). The other languages of this family are all extensions of AL. For instance, ALEN is the extension of AL by adding full existential quantification and number restrictions.

Research on Description Logic has covered both theoretical study like the complexity of reasoning and practical application such as implementation and development of knowledge representation systems in several problem domains. The key element has been the research methods which are based on a very close interaction between theory and practice. On the one hand, the formal and computational properties of logical reasoning such as decidability and complexity of various description logic formalisms have been studied. On the other, there are a few different description logic based knowledge representation systems with different expressive power and reasoning ability employed in different problem domains.

2.4.3.2 Description Logic Systems

Description logic systems can be divided into three different generations based on their historical evolution rather than their specific functionality, namely Pre-DL systems, DL systems and current DL systems. The transition from semantic networks to more well-founded terminological logic started with KL-ONE [13] which is thought to be the ancestor of DL systems. The earlier stage Pre-DL systems also derived from KL-ONE. A classification algorithm as well as data structures representing concepts were the main focus at this stage. Because of the trade-off between the expressive power of a DL language and the complexity of reasoning with it, DL systems could be further divided into three different categories regarding the implementation of reasoning. The first category can be defined as limited plus complete, which indicates that subsumption would be computed efficiently, possibly in polynomial time by restricting the set of constructs. The CLASSIC system [14] is thought to be the most successful example in this category. The second category can be defined as expressive plus incomplete. This category emphasises both strong expressive power and efficient reasoning ability. However reasoning algorithms turn out to be incomplete in this category. LOOM [78] and BACK [94] system are the noticeable examples in this category. The third category can be depicted as expressive plus complete. Compared with the second category, the reasoning algorithms are complete. KRIS [5] is a good example of this category. Current DL systems are focusing on the need for complete algorithms with strong expressive power support. With the extensions of tableau-based techniques and the introduction of several optimisation techniques, more advanced current DL systems have been developed. FaCT [50] is the most significant example.

2.4.4 Resource Description Framework (RDF)

Resource Description Framework (RDF) developed a basic ontology language for describing web resources which are normally identified by a Uniform Resource Identifier (URI). It is a data model represented in XML syntax with simple semantics, containing objects and their relations. RDF represents resources by RDF statements which indicate properties and their values. A property is a resource which has a name, while a property value can be another resource. Those RDF statements are written as a tri-tuple <Subject, Predicate, Object>. In an RDF Graph, resources are all related and linked together in a way that the subject of the tri-tuple could be the object of another tri-tuple.

In order to describe application-specific classes and their properties, RDF Schema (RDFS) is developed as an extension of RDF. RDFS allows for defining instances of classes, subclasses of classes and sub-properties of properties. However, RDF(S) has the following disadvantages in terms of describing resources [101]:

- RDF(S) cannot provide detailed description without localised range, domain, existence and cardinality constraints.
- RDF(S) does not provide transitive, inverse or symmetrical properties.

Therefore, OWL was developed as an extension of RDF(S) with following features:

- It is easy to understand and use.
- It is very expressive and can be formally specified.
- It can provide automated reasoning support.

2.4.5 Web Ontology Language (OWL)

More efficiency, greater knowledge sharing and ease of use are provided by the second generation of web, which is known as semantic web [101, 104]. The advantage of introducing semantic web is to enable automated collection and correlation of different information resources on web. It is designed to facilitate web users by speeding up the process of navigating and searching useful information across different webs. It is proposed by Tim Berners-Lee as the future web technique with which online information resources are expressed explicitly so that machines will be able to understand, process and integrate those resources automatically [10]. Semantic web can be considered as a common framework of data sharing and reuse across various applications and domains [101, 104]. It is a web of data on which both machines and people will be able to understand and process information resources. In order to support the automatic process and integration of information resources, machine-readable languages are the fundamental elements, i.e., the data must be expressed with machine readable semantics. To implement semantic web the existing rendering markup needs to be extended with semantic markup. Ontology is employed as a universal vocabulary for annotations, which is the key to the interoperability of semantic web. As introduced earlier, ontology is a study of the nature of existence which can formally describe a domain of discourse by capturing knowledge of the domain of interest, expressing the concepts of the domain and describing the relationships between concepts. Therefore ontology is composed of a finite set of concepts and relationships. There are four steps that have been defined for building semantic web, i.e., annotation, integration, inference and interoperation [80].

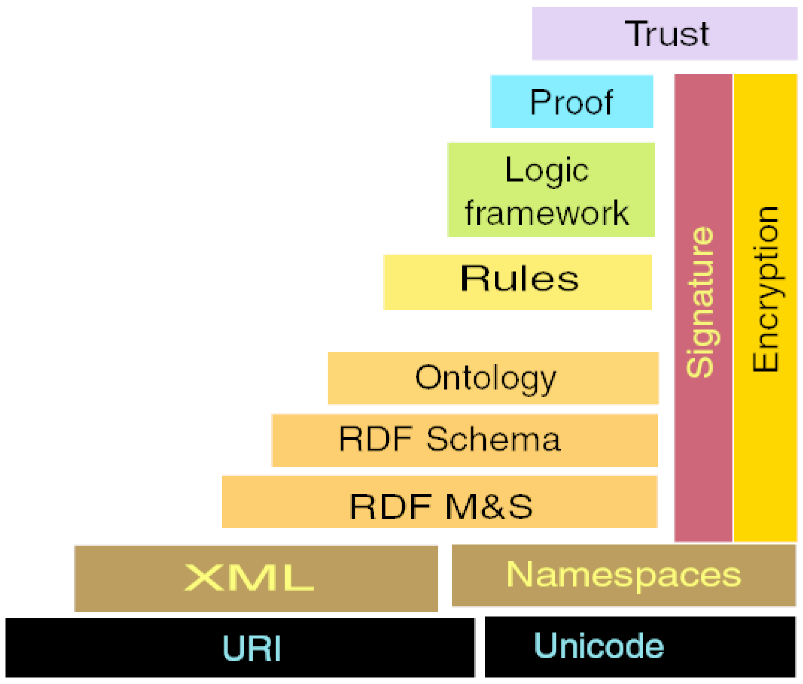


Figure 2-6 the Layers of Semantic Web Technologies [46]

Figure 2-6 presents the layers of semantic web technologies. There is a "language stack" defined to provide a basic mechanism that supports the usage of metadata, namely XML, Resource Description Framework (RDF) and Web Ontology Language (OWL). XML is the foundation of this language stack because of the ability to define customised tagging schemes. RDF is located in the middle of the stack as a flexible methodology for data representation. OWL is on the top of the stack and it provides a way to formally define the terminology used in web.

OWL is a language designed to define and instantiate web ontologies with which the meaning of terms and their relationships can be represented explicitly. It can be understood and processed by machines, which supports the goal of semantic web. Generally speaking the ontology written in OWL consists of the descriptions of three elements i.e. classes, properties and their instances.

OWL has three increasingly expressive sub-languages a the different balance between the expressive power and reasoning ability, namely OWL Lite, OWL DL and OWL Full. Those sub-languages are designed to facilitate different web users with specific requirements. OWL Lite is the least expressive one having strong reasoning ability because of simple class hierarchy and constraints. OWL DL is based on description logic. It is more expressive and provides computational completeness which indicates computable and decidable. OWL Full is the most expressive language which cannot guarantee computable and decidable. Hence OWL-DL is the most widely used language that is suitable for knowledge representation with reasoning support.

2.4.6 OWL Reasoning

OWL provides a reasoning service to help knowledge engineers and users build and use ontology by checking logical consistency of classes and computing implicit class hierarchy. The OWL reasoning service is especially important for designing and maintaining large global ontologies, i.e., integrating and sharing ontologies, checking consistency and implying implicit relationships. For most DLs the basic inference problems are decidable i.e. solving the problems in a finite number of steps.

OWL also provides the following reasoning services which can facilitate a knowledge based approach:

- **subsumption reasoning**: (a) to infer when one class A is a subclass of another class B, (b) to infer that B is a subclass of A if it is necessarily the case that all instances of B must be instances of A, (c) to build concept hierarchies representing the taxonomy – the classification of classes.
- **Satisfiability reasoning**: (a) to check when a concept is unsatisfiable, (b) to check whether the model is consistent.

In OWL reasoning classes can be described in terms of necessary and sufficient conditions while some frame-based languages can only have necessary conditions. The necessary conditions indicate that the condition must hold for checking the instance of the class, while the sufficient conditions indicate that the object must have the properties to be able to be recognised as a member of the class. In addition, the OWL reasoning service provides the way to perform automatic classification [101].

2.5 Related Work

2.5.1 Operating System Modelling and Development

In order to build an ontology for operating systems the problems of modelling and developing operating systems have been reviewed. Several works have focused on real-time operating system (RTOS) modelling and development.

Gerstlauer et al. [40] present a real time operating systems (RTOS) model built on top of existing system level design languages which by providing the key features typically available in any RTOS allows the designer to model the dynamic behaviour of multi-tasking systems at higher abstraction levels and incorporate it into existing design flows. Based on this model the refinement of system models to introduce dynamic scheduling is easy and can be done automatically. The adaptation of this model to another System Level Design Language (SLDL) like SystemC may be a hard and complex task because of the lack of support to model common services such as preemption and true multi-task execution. The RTOS model is considered to facilitate both the development and reengineering process of real time operating systems.

Madsen et al. [79] present a modelling framework consisting of basic RTOS service models including scheduling, synchronisation, resource allocation and a task model that is able to model periodic and aperiodic tasks as well as task properties.

Desmet et al. [28] propose a high-level model of a system-on-chip operating system (SoCOS). They provide a C++ library for system level design which offers services analogous to an operating system in software design. Real-time aspects can be gradually introduced without rewriting the code.

Hessel et al. [49] address scheduling decisions for real-time embedded software applications by introducing an abstract RTOS model as well as a novel approach to refine an unscheduled high-level model to high-level model with RTOS scheduling. Their model is similar to Madsen and Gerstlauer approaches but the SLDL is different.

Wang and Malik [117] propose an approach to tackle the issue of modelling device access behaviour with a formal model, by using extended event driven finite state machines to synthesise a correct-by-construction operating system based device driver.

Yi et al. [126] present the virtual synchronisation technique with OS modelling for the case where multiple software tasks are executed under the supervision of a real-time operating system.

Gauthier et al. [39] propose a methodology for automatic generation of application specific operating systems and automatic targeting of application code. Their method starts the automatic generation of an operating system from a very small and flexible kernel and includes only the operating system services specific to the application.

Khan et al. [64] propose an approach based on Model Driven Architecture to facilitating real-time systems development. A modelling methodology is applied in modelling system architecture and real-time behaviour via a subset of UML 2.0 diagram types with the associated concepts and notations. After that this model is transformed to C code following an associated mapping strategy. This approach can be used in both real-time system development and reengineering processes, however, the mapping of UML 2.0 models in C still lacks perfection because of the differences between object oriented concepts modelling and procedural programming.

2.5.2 Platform Specific Software Migration and Software Portability

A few literature reviews have been carried out in order to obtain the necessary information required by software migration. Many projects in industry have been carried out in relation to migration of software applications from one platform to another. Many world-famous software research institutes are doing research to facilitate migration between Windows and UNIX-like operating systems e.g. Microsoft Interoperability and Migration Centre [83], AT&T Labs Research [4], Cygwin [25], Interix [56].

Microsoft has released a UNIX Custom Application Migration Guide with toolset support [83] which provides best practices, tools and code samples on the planning, migrating and deploying of UNIX ANSI C/C++ and FORTRAN custom applications into the Microsoft Windows environment.

The UWIN package [4] is a software tool developed by AT&T Labs which allows UNIX applications to be built and run on the Windows environment with few, if any, changes necessary. Cygwin and Interix are alternatives of UWIN.

The portability of software application has been studied for decades of years. Software portability research has been proposed for many aspects such as program, data, user interface (UI) and documentation [63]. Many factors which hinder software portability have been indicated ranging from hardware platforms to operating system platforms.

Janka [58] presents a new development framework PeakWare for RACE (PW4R) which provides the ability to manage software and hardware libraries which supports software reuse and portability.

Vuletic et al. [114] propose a transparent, portable and hardware agnostic programming paradigm to achieve portability and uniform programming by reconfigurable computing. Mosbeck et al. [86] describe software portability in open architectures. They argue that standardised interfaces and a set of common services must be provided to facilitate application portability in open architectures known as abstraction and isolation methods. These three research areas are similar in that they all abstract a set of standard services and create a virtual layer to provide such standard services leading to the improvement of software portability. However none of these works indicates or utilises the knowledge intensive features to improve software portability.

2.5.3 Knowledge Based Software Engineering Methods

Knowledge based software engineering methods are those methods which utilise knowledge representation techniques to solve software engineering problems. Devedzic [30] proposes that ontologies are needed in all software systems. In his work, he suggests all software systems should always “know” about entities and their attributes and relationships in the relevant world i.e. all systems need knowledge.

The 2004 Guide to the Software Engineering Body of Knowledge (SWEBOK) [1] opens new perspectives on ontology engineering in the field of software engineering.

Wongthongtham et al. [121] proposes a software engineering ontology for software engineering knowledge management in multi-site software development environment. They argue that reaching a consensus of understanding is of benefit in a distributed multi-site software development environment. Software engineering ontology, which signifies project information such as development and changes in requirements or design, can be used to reach that consensus.

Guarino [43] demonstrates the significant role of ontology in information system development, leading to the perspective of ontology-driven information systems. Two orthogonal dimensions have been distinguished when discussing the impact that ontology can have on an information system: a temporal dimension which concerns whether ontology is used at development time or at run time and a structural dimension which concerns the effect of ontologies on information system components.

Zimmer and Rauschmayer [134] present a way of enhanced ontology-based software modelling. Their tool TUNA aims to combine XP and MDA by giving MDA rich means for integrating modelling concepts with the source code. Furtado et al. [38] propose a universal user interface design approach which is separated into three levels of abstraction. The creation of the domain ontology is the conceptual level, the elaboration of models is the logical level and the code transformation is the physical level.

2.5.4 Knowledge Based Software Reengineering Approaches

Knowledge-based software reengineering suggests working on both bottom-up and top-down manners. Bottom-up [17] strategy demonstrates a comprehension approach that starts with source code reading and then mentally chunks the low-level software artefacts into high-level mental abstractions. Top-down strategy illustrated by Brooks [16] describes a comprehension approach as being a hypothesis driven one in which an initially vague and general hypothesis is refined and elaborated based on information extracted from the program text and other documentation leading to a hierarchical comprehension structure. The introduction of a knowledge-based approach to software reengineering can bridge the gap between software representation and the mental model and improves the efficiency and correctness of software reengineering. The essential parts in this field are knowledge representation of application and problem domain and traceability between software source code and the domain knowledge base.

Many works have utilised ontology to facilitate program comprehension. Previous research in facilitating software engineering via Description Logic (DL) [7] has been carried out since the 1990s. The basic idea is to use a DL to implement a software information system (SIS) i.e. a system that would support software maintainers by helping them find out information about a large software system e.g. the first SIS, LaSSIE [29], was developed to assist the understanding of AT&T's Definity 75/85 software system.

The previous work which was carried out by Yang et al. [123] suggests that ontology has a great potential for legacy software understanding and re-engineering. RWSL [124] is used as an ontology language for knowledge representation. A concept-oriented belief revision approach to domain knowledge recovery from source code has been proposed [75].

Zhang et al. [128] propose an approach to identifying security flaws and security concerns. An ontology-based program representation is introduced to facilitate security experts and programmers specify their security concerns via the ontology. However a more comprehensive set of predefined queries to capture knowledge of security concerns will be difficult to achieve. They also present an approach to supporting website architectural evolution [129] which provides a consistent ontological representation for both source code and documentation. Through their ontology tool set they can detect implementation defects of architectural styles and inconsistencies between documents and source code in web-based applications and they can also discover some important properties of identified components. Nevertheless the linking between the source code ontology and documentation ontology will require further research.

Johnson and Soloway [61] present a knowledge-based program understanding approach which does on-line analysis and understanding of Pascal written by novice programmers. A knowledge base of programming plans and strategies, together with common bugs is used to construct the mapping between requirements and the code, which is in essence a reconstruction of the design and implementation steps.

Li et al. [74] introduce an innovative approach to recovering domain knowledge with enhanced reliability from source code. They divide domain knowledge into interconnected knowledge slices and match these knowledge slices against the source code. A simplified semantic network is proposed as the representation of domain knowledge that covers a rich set of necessary relationships among concepts. Each Concept in the semantic network tries to find its counterpart in source code during a knowledge recovery process.

2.5.5 Knowledge Based Software Tool Support

Knowledge based approaches are also used to improve software tools. Djuric et al. [32] propose that AI tools should be integrated with mainstream Software Engineering (SE) tools and thus become more widely known and used. In their work they developed the Air framework based on model-driven-architecture concepts. Based on such a framework they can easily extend mainstream SE tools with new functionalities. This is a trend in SE tools development.

In previous work carried out by Tsai et al. [110] tools that use the knowledge-based approach to maintain complex large-scale software systems are surveyed. ARIES [60] is viewed as applying the notion of software representation and incorporating a strong coupling to a transformation system. Requirements Apprentice (RA) [97] has been developed to fill the gap between informal and formal specifications. Requirements document will be updated with RA's understanding and the developers' interactive operations. REMAP [96] is focusing on process knowledge to reason about the consequences of changing conditions and requirements in system maintenance. SPECIFIER [84] is a specification derivation system which consists of three components: a preprocessor, a reasoner and a postprocessor. A knowledge base is included to support the reasoner in producing an informal specification. These tools cover the whole phases of the software life-cycle e.g. requirement, analysis and design.

CODA [79] is a knowledge-based automated designer's assistant and could assist designers in creating concurrent system designs by being embedded in computer-aided software engineering (CASE) systems. Previous research and experiments carried out on CODA has shown that advances in knowledge engineering hold potential for effective automation of software design methods.

Sidarkeviciute et al. [103] developed a knowledge-based toolkit for graphical presentation, or visualisation, of programs. It is proposed that the introduction of knowledge-based techniques increases the extensibility and modifiability of the code analysers. Moreover the knowledge-based toolkit provides an intelligent environment for storing knowledge about programs and performing reasoning on them.

A tool is specified in the work of Ambrosio et al. [2] in which ontologies are utilised to help the combination of application domain information and software reengineering knowledge, producing up-to-date documentation that evolves with time. Their tool includes two types of ontologies: structural ontologies and domain ontologies. However the linking technique between the two ontologies is not specified and is a major flaw in their work.

ONTODM [41] is an ontology-based tool supporting the specification of domain models in Multi-Agent Domain Engineering. GRAMO (Generic Requirement Analysis Method based on Ontologies) defines the activities to be accomplished in the construction of domain models and is proposed as the basic technique of ONTODM. Some of the advantages of using ontologies for representation of reusable products have been shown.

Ontology can also be used to manage and integrate reengineering tools in order to improve the reengineering process. Jin and Cordy [59] utilise an ontology-based approach to facilitate software analysis and reengineering tools integration via OASIS (Ontology Adaptive Service-Sharing Integration System), which encompasses a domain ontology and external tool adapters. The integration they propose is focused on service-sharing while most previous work is data-centric. Domain ontology is used as a knowledge base to facilitate service management. Such usage is similar in web service.

2.6 Summary

In this chapter the background and related work of knowledge based software reengineering are introduced:

- A brief overview of software crisis is presented – complexity and change are concluded as the two major causes for software crisis. Software engineering is then introduced with modern software engineering paradigms, namely, object oriented programming, component based development, service oriented architecture and cloud computing. Formal methods and domain engineering are also introduced as background knowledge to software engineering. In addition, software taxonomy is introduced to provide contexts for empirical studies and to facilitate exploring the applicability of those studies. Furthermore, it will make the methods more easily reused by mapping to categories within software taxonomy.

- Three different types of software change are introduced, namely, maintenance, modernisation and replacement. The law of software evolution is quoted to describe software evolution. A few different definitions on software reengineering are given and compared. Related basic terms are also explained.

- Model driven architecture is briefly introduced. The fundamental idea of MDA is the separation of business logic and technical support. Modelling Maturity Levels are also discussed as an assessment of the MDA approach. The concept of technical space is described with the comparison of different TSs. ATL model transformation is also introduced in the context of model driven engineering.

- Knowledge representation and knowledge engineering are reviewed. An overview of ontology is provided. Description Logic is also introduced with a definition of and applications to description logic systems. Moreover, resource description frameworks (RDF) and web ontology language (OWL) are also presented.

- The proposed research is to try to build a knowledge-based software reengineering framework in which software system knowledge represented by ontology and description logic are manipulated in order to facilitate software reengineering tasks such as software migration and program understanding. Hence there are a great range of related projects that this study has reviewed and referred to, covering operating system modelling and development, software portability, platform specific software migration, knowledge based software engineering methods, knowledge based software reengineering approaches and knowledge based software tools support.

Chapter 3 Developing Software System Ontology for Reengineering Use

Objectives

- To introduce ontology based software reengineering framework
- To discuss the scope of the proposed approach
- To describe the ontology based software reengineering process
- To describe the integration of software system ontology
- To describe the deployment of software system ontology

3.1 Overview

Software reengineering is the major technique for evolving software systems. The main step for reengineering is to perform program comprehension and then to implement a change in a safe manner, and to retrieve a form of software representation upon which this change can be performed more effectively. Although software reengineering has established itself as a crucial part of software lifecycle, the manual tasks in software reengineering are time consuming and error prone for the following reasons: (1) many existing software systems contain huge volumes of complicated source code, (2) many existing software systems have deteriorated supporting documentation, and (3) software maintainers are not domain experts and can not understand the existing systems from the perspective of the application domain. Hence, it is worth to investigate a semi-automated approach to assist software reengineering.

Generally speaking, traditional software reengineering projects will either start with a code-based program comprehension, or with a documentation-based one. On the one hand, since software source code is always complex and organised around specific functions, rather than domain concepts, code-based comprehension only returns code-related concepts, e.g., concepts up to a structural or algorithmic level of source code. On the other, driven by urgent submission deadlines, programmers hardly have time to write ‘profit-less’ documentation. This is again the case for the later stage of the software life circle, where programmers fail to keep the documentation up-to-date. Meanwhile, informal concepts defined in documentation, e.g., specification, are not directly related to the code, and are defined at a different level of abstraction than that of source code. All these facts show that in practice neither a code based approach nor a documentation-based one is sufficient. Achieving a more understandable form of software system will become necessary for performing software reengineering, which implies representing software systems from different perspectives, e.g., code structure, module functionality, application domain, data, etc. In addition, a unified platform is also required to more easily represent and integrate these different perspectives.

Knowledge representation can be used as a medium for efficient computation [27]. Due to the knowledge intensive feature of software reengineering projects, an ontology-based reengineering approach is proposed in this research. Ontology is used to represent the knowledge of software systems in the reengineering process. Once the software system ontology has been built, it could be used as a persistent aid to software reengineering projects. Furthermore, elicited knowledge stored in ontology will be accumulated in a reusable manner over a number of maintenance activities on a specific domain application. Last but not least, in a way, ontology also provides a unified platform which supports integration of different types of knowledge sources.

This research has been divided into three main parts: software ontology development, software ontology integration and software ontology deployment. Section 3.2 will introduce the knowledge based software reengineering framework, which is developed in this study.

3.2 Ontology Based Software Reengineering Framework

Figure 3-1 demonstrates an ontology-based software reengineering framework developed in this research, which includes three different steps in terms of ontology engineering. The first step is software system ontology generation, which focuses on capturing useful knowledge and representing them via ontology. The generation process is supported by three different methods, namely, top-down ontology generation, bottom-up ontology generation and middle-out ontology generation. Code ontology, data ontology and framework ontology could be generated by a bottom-up approach. Operating system ontology is built by a top-down approach. Application domain ontology may be obtained by a middle-out approach.

Software system ontology integration is the second step of the proposed approach, aiming at integrating the different ontologies generated in step one. A description logic based ontology mapping algorithm is employed in this study based on both structure-level and element-level. Code ontology, data ontology, framework ontology, domain ontology will be integrated in order to provide a comprehensive model for software reengineering projects.

The last step of this research is software system ontology deployment, which is the final goal of the proposed study. By deploying software system ontology in software reengineering projects, a knowledge-based approach can be employed to semi-automate the reengineering process such as program comprehension and software migration/porting, etc.

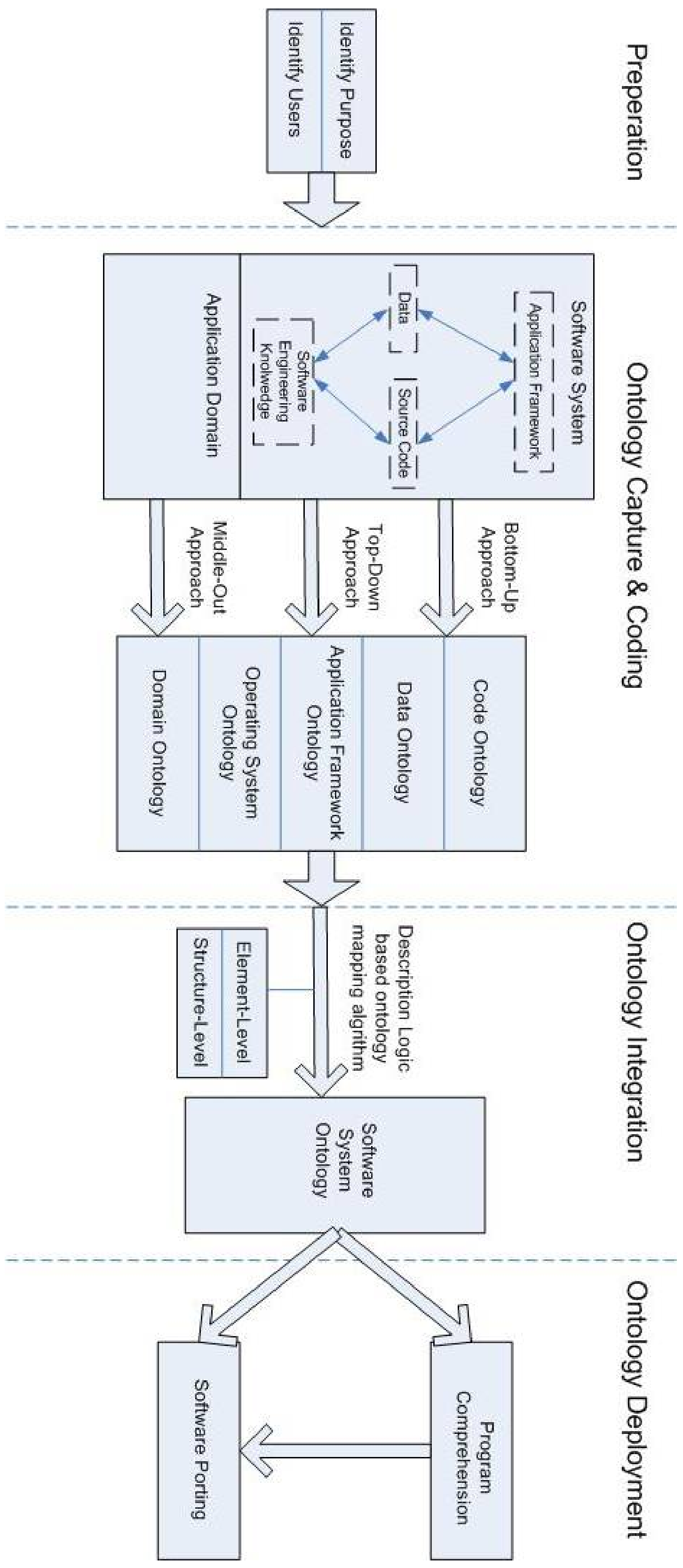


Figure 3-1 Ontology-Based Software Reengineering Framework

3.2.1 Selection of Software Systems and their Knowledge Representation Aspects

Software systems are far too complex for any human being to understand as a whole. To support understanding and reengineering such complex systems, three aspects are proposed as potential knowledge representation aspects. These are source code aspect, data aspect, and application framework aspect. Source code and data are two basic elements of a software system in a broad sense, while application framework always represents the relationship between source code and data.

Section 2.1.5 has emphasised the importance of having software taxonomy to improve software research by helping researchers to apply their research systematically to particular types of software systems. Taking into account the software taxonomy discussed in Section 2.1.5 and the features of different types of software system, this research will select data-dominant software and system software as main subjects. Specifically, business-oriented software is chosen from the data-dominant software category with the following constraints:

- The selected business-oriented software is implemented by an object-oriented programming language.
- The selected business-oriented software is implemented to interact with a relational database.
- The selected business-oriented software is implemented on Hibernate ORM framework.
- The selected business-oriented software needs to meet at least the first one of the above constraints.

On the other hand, an operating system is selected from the system software category. Although the operating system chosen for this study is not necessarily open source, access to a complete list of system call interfaces will be needed for the following reasons:

- System call interfaces can be relatively cheaply derived from operating system documentation.

- System call interfaces may be used to isomorphically represent both system models that are designed during the forward engineering phase, and legacy wrappers that are acquired during the reverse engineering phase, specifying the abstract services implemented by the legacy systems.

- Once mappings between ontology and system call interfaces are established, they may be straightforwardly transformed into parameterised components that rely for part of their execution on legacy wrappers.

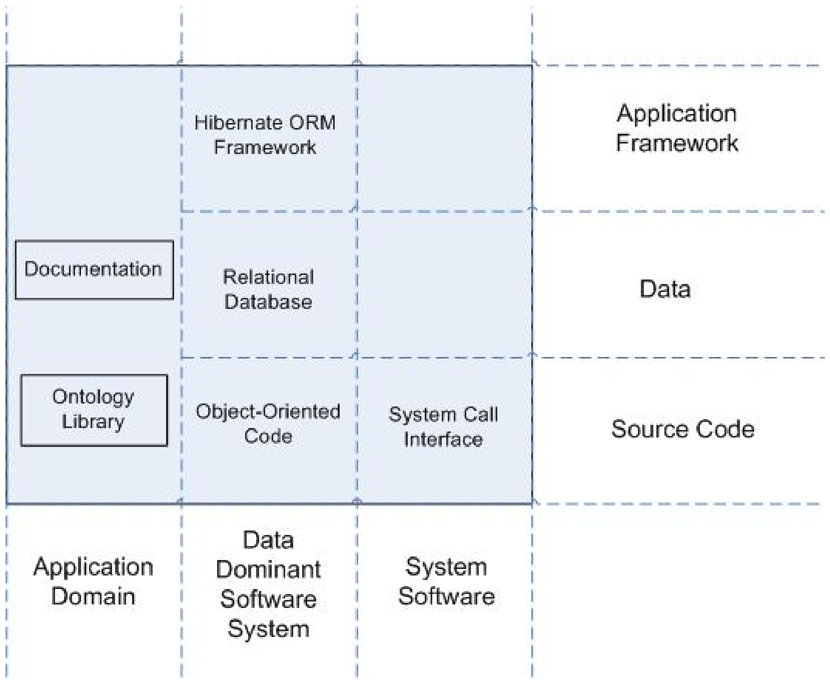


Figure 3-2 Knowledge Representation on Different Software Categories

Figure 3-2 illustrates the selection of software systems and their knowledge representation aspects. The proposed approach will be applied to different types of software systems by representing parts of or all of those three possible knowledge representation aspects with ontology. Horizontally, the knowledge representation aspects of software systems are drawn as three parallel levels. Vertically, two different types of software systems can be represented by ontology through those representation aspects. In addition, application domain knowledge is also drawn parallel to two different software domains as it will add an extra dimension to the software system knowledge representation. Consequently, software system ontology will be built to represent some of these knowledge representation aspects in order to semi-automate the software reengineering process. For operating systems, ontology is mainly built on the knowledge of system call interfaces, which is a source code aspect. While for business-oriented software, ontology can be built on all three representation perspectives. In this research, object-oriented source code, relational databases and Hibernate ORM frameworks are chosen from each level to a build software system ontology knowledge base.

3.2.2 Ontology Based Software Reengineering Process

The ontology based software reengineering process is one of the crucial parts in this research. As described in Section 2.4.2, there are a few studies in the ontology engineering field which focus on the process of building ontology. This research defines an ontology based software reengineering process by specifying and extending the generic ontology development process first proposed by Uschold and Gruninger [112, 113]. Preparation processes and deployment processes are added to the original ontology building process, while the original ontology building processes have been specified with the requirements of software system ontology in terms of reengineering. The extended process is:

1. **Preparation**. Aims to identify the purpose of building the ontology and its potential users. In this study, the purpose is to provide a software system knowledge base so that some of the software reengineering processes can be semi-automated through knowledge acquisition of the software system ontology. The potential users of software system ontology will be mainly software maintainers and developers. Therefore the majority of the knowledge stored in software system ontology should be that which software maintainers require rather than the common knowledge for the end users.

2. **Capture**. As an essential part of building software ontology, capture is the

process to which seeks generic and reusable representations of software system knowledge that can be reused across a variety of software reengineering activities. Key concepts and relations in software system ontology will be identified during this process. A brainstorm is performed at this stage, which researches all the potential knowledge representation aspects in order to find the components of the ontology. All the terms and key words from those different aspects will be written down on paper and then their relationships will be studied and analysed. As a result, all those terms and key words will be organised in a more structured way, in other words, classification.

3. **Coding**. As a formal description of concepts and relationships, ontology forms a shared terminology for the objects of interest in software reengineering. Through the coding process, software system ontology will be either manually or (semi-) automatically generated and will be written by an ontology representation language. There are two main ontology generation approaches used for coding, the so-called top-down approach and bottom-up approach. There is also a mixture of the two, known as middle-out approach. Different software system domains will require different ontology development methods. Coding and capture are simply merged into one step in this study, since they are always closely connected.

4. **Integration**. As a unified platform, ontology provides the methodologies to integrate knowledge sources from different aspects. Therefore, ontologies that represent different knowledge representation aspects of software system can be integrated into one, which is then used to provide knowledge acquisition and therefore assists software reengineering. The integration of different ontologies is sometimes also known as ontology alignment or ontology mapping. There are basically two different approaches, known as the schema-based approach and the instance-based one. Furthermore, there is also a distinction between element-level and structure-level mapping. In this study, ontology integration is implemented by a Description Logic based ontology mapping algorithm, which covers both element-level and structure-level mappings.

5. **Deployment**. As a knowledge representation technology, the final step will be to put it into use – deployment. In this study the deployment of software system ontology is aiming at semi-automating some of the software reengineering processes by providing knowledge acquisition to a software system ontology knowledge base. Program comprehension and software migration/porting are the two potential usage areas in which there are a large amount of manual activities performed by software maintainers, which prove to be time consuming and error prone. A knowledge-based approach is therefore introduced to replace these manual tasks to thereby improving traditional software reengineering.

3.2.3 Capture – Identification of Important Concepts and Relationships in Software Systems

Identification of important concepts and relationships in software systems is the goal of the ontology capture process. This section will briefly describe how software system ontology is being captured.

A brainstorm session is performed in order to seek out all potential concepts and relationships for building software system ontology. The participants in the brainstorm session are the first supervisor of the author, two academic staff members and two PhD students. The supervior and academic staffs are acting as domain experts as well. The brainstorm session is performed in a syndicate room, where people can discuss and write output on a whiteboard. The instructions set up to guide the brainstorm session are as follows:

- To identify individuals encountered in a software system domain; to further consider materialisation and values.
- To identify concepts that group these values.
- To distinguish independent concepts from relationship-roles
- To develop a taxonomy of concepts, to further consider disjointedness and allow for subconcepts

- To systematically search for part-whole relationships between objects, creating roles for them, and further consider making them subroles

- To determine local constraints regarding roles such as cardinality limits and value restrictions, and elaborate on any concepts introduced as value restrictions

- To determine more general constraints on relationships, such as same-as and is-a

It's aim is to decompose the software environment and to elicit the important terms that could be used in software reengineering. In general, components which constitute a software environment include operating systems, database systems, software applications, application framework techniques, programming languages, hardware systems, application domains and end users, etc. And knowledge contained in these components could be divided into three main types, namely, domain knowledge, software engineering knowledge and code knowledge.

Application Domain Knowledge illustrates knowledge in a specific problem domain, such as banking, shopping, management, etc. The term domain knowledge has the advantage of strongly emphasising a focus on domain concepts, not software entities. Obtaining domain knowledge helps in creating a taxonomy for the terminology or vocabulary of the problem domain, decomposing the problem space into understandable units (concepts). It can be used to enhance the communication between interested parties to clarify what the important concepts are, and how they are related. Domain knowledge describes things in the real world problem domain, not the software artefacts. In an ideal situation, domain knowledge could be obtained from the public ontology library, such as Protege Ontology Library [106]. If the domain ontology is not available through the ontology library, it will have to be built from scratch by the researchers and domain experts.

Software Engineering Knowledge represents knowledge in the software engineering field such as software design theory and programming languages, etc. Object-oriented (OO) programming and application programming interfaces (API) are considered to be the fundamental software engineering knowledge in this study. Object-oriented design has the following features which makes it possible to be transformed into the form of ontology:

- Object-oriented class denotes a set of objects with common features, while concept in ontology does the same thing.
- Object-oriented class has hierarchical structure, and hierarchical structure is the basic structure for taxonomy, which is also one of the features of ontology.
- Object-oriented class has properties, while ontology has two types of properties: object property and datatype property.
- Object-oriented class has relationships such as associations and dependencies. These relationships are represented as roles or properties in ontology.
- Class diagrams can be stored in an XML style and ontology language OWL-DL is also an XML style. It is therefore easy to transform between these two representations.

On the other hand, being created to interact with a set of predefined functions used by software components, APIs are implemented by different software products such as operating systems, libraries and applications. In essence, APIs are just the vocabulary and calling conventions that programmers need to use in order to access the services they provide. However, these vocabularies and calling conventions are somehow storing very useful knowledge about the software products and should be considered as candidates for composing software system ontology:

- APIs are relatively cheap to derive from the existing software products documentation.
- APIs describe different services provided by different software products, and therefore they could be used to model and distinguish software products in terms of functionality.
- APIs are always organised in a well-structured way, the classification of APIs can indicate the classification of software functionalities and components.

Code Knowledge is elicited from code straight away. Code is a relatively generalised concept, which could include all sorts of different software artefacts such as application frameworks, source code, databases, documentation, etc.

An application framework is a pre-defined program structure or a set of reusable common code hidden behind well-defined APIs, which provide generic functionalities. Using application frameworks can reduce development time by providing reusable generic functioning code allowing programmers to spend more time on system requirements instead of low-level system implementations. Meanwhile, application frameworks can sometimes also store the interrelations between different software components or between software components and data. For instance, the Hibernate ORM framework uses XML configuration files which store the mapping between an object-oriented domain model and a traditional relational database. Obtaining and maintaining a knowledge base for such implicit information hidden inside software systems could be very beneficial to any further software maintenance activities.

Source code is a set of statements or declarations written in a programming language which can be read and understood by human beings. Such statements or declarations could cover a great deal of information, such as design patterns, data structures, functionalities, business logics and their interrelationships, etc. As for most of the software reengineering projects, source code is the part to be targeted and modified in order to meet the new requirements. In order to explore the information of source code, a traditional software reengineering approach will rely on an Abstract Syntax Tree (AST), a data flow diagram, a control flow diagram or UML class diagram, etc., which are different software representation techniques that can represent many different aspects of software elements and their interrelationships. Obtaining source code knowledge is necessary for performing any software maintenance activities.

A database stores implicit knowledge about the system, which could be useful in software reengineering projects. The process of eliciting knowledge from a database is defined as data understanding in this study. At this stage, only a database schema is being studied. The actual data stored in a database does not concern software maintainers in terms of software change. It is the database schema stored in the database dictionary that concerns them. Things like the semantic meanings of the tables and columns, and the relationships among different tables and columns will affect a software maintainers' decisions when performing a modification on software, while things like an actual individual record stored in some specific table will not. With respect to the comprehension of data sources, database ontology can be used for the identification and association of semantically corresponding information concepts. Obtaining such database ontology could provide software reengineering with many benefits. Firstly, extracted database schema concepts could be used to link with the concepts recovered from source code, which could enhance the understandability of the concepts from code ontology. Secondly, extracted relationships from a database schema along with code ontology could be used to guide software reengineering therefore towards implementing a change in a safe manner. Thirdly, extracted database schema ontology could be used to improve data interoperability for other uses, e.g., data integration.

3.2.4 Coding – Generation of Software System Ontology

The final product of knowledge extraction in the previous steps is software system ontology, which will be formally represented in one of the popular ontology languages in order to support the acquisition and manipulation of software knowledge in the software reengineering process. There are two main approaches to developing software system ontology, namely, bottom-up ontology development and top-down ontology development.

A "bottom-up" method is a way of building ontology by starting from the other end of the spectrum, which indicates transforming other information forms into ontology species. A "top-down" method starts by thinking of and deciding about core principles, which are used to guide the development of foundational ontology. The "bottom-up" method requires that the developer knows how to transfer one model into another, while the "top-down" method requires that the developer has an almost complete understanding of the software system.

3.2.4.1 Bottom-Up Software System Ontology Generation

The "bottom-up" approach indicates starting from "legacy" and then transforming this into ontology species. Such "legacy" could be logic-based knowledge representation, well-structured information, databases, etc. Specifically, in this research, this "legacy" will be a UML class diagram, an XML configuration file of Hibernate ORM framework and a MySQL relational database, from which software system ontology will be generated. Model transformation techniques are crucial in this bottom-up approach and will be employed to transform the "legacy" into ontology.

Figure 3-3 illustrates the scenarios in which other models are transformed into OWL ontology by ATL model transformation language in the context of Model Driven Engineering (MDE). The KM3 model is selected as a mediator model in the core transformation, which will bridge the gap between different models and will provide extensibility for future research. The core transformation includes two distinct steps. The first step is dedicated to the mapping from software system "legacy" to the KM3 model, e.g., UML classes are mapped into KM3 classes, UML datatype to KM3 datatype, etc. The second step deals with the transformation between the KM3 model and the OWL knowledge model, e.g., KM3 classes are mapped into OWL classes, KM3 attributes into OWL datatype properties, KM3 references into OWL object properties, etc. Subsequently, an OWL knowledge model will be produced as a result of the core transformations. After performing the core transformations, one extra transformation is still needed, as the OWL knowledge model is not yet in a formal ontology format. In order to obtain a formal ontology that could be manipulated by an ontology editor, an OWL/XML extractor is required to transform the OWL knowledge model into the XML model with OWL/XML syntax elements. As a result, software system ontology is generated by a bottom-up approach, and a conversion between model engineering technical space and ontology technical space is conducted.

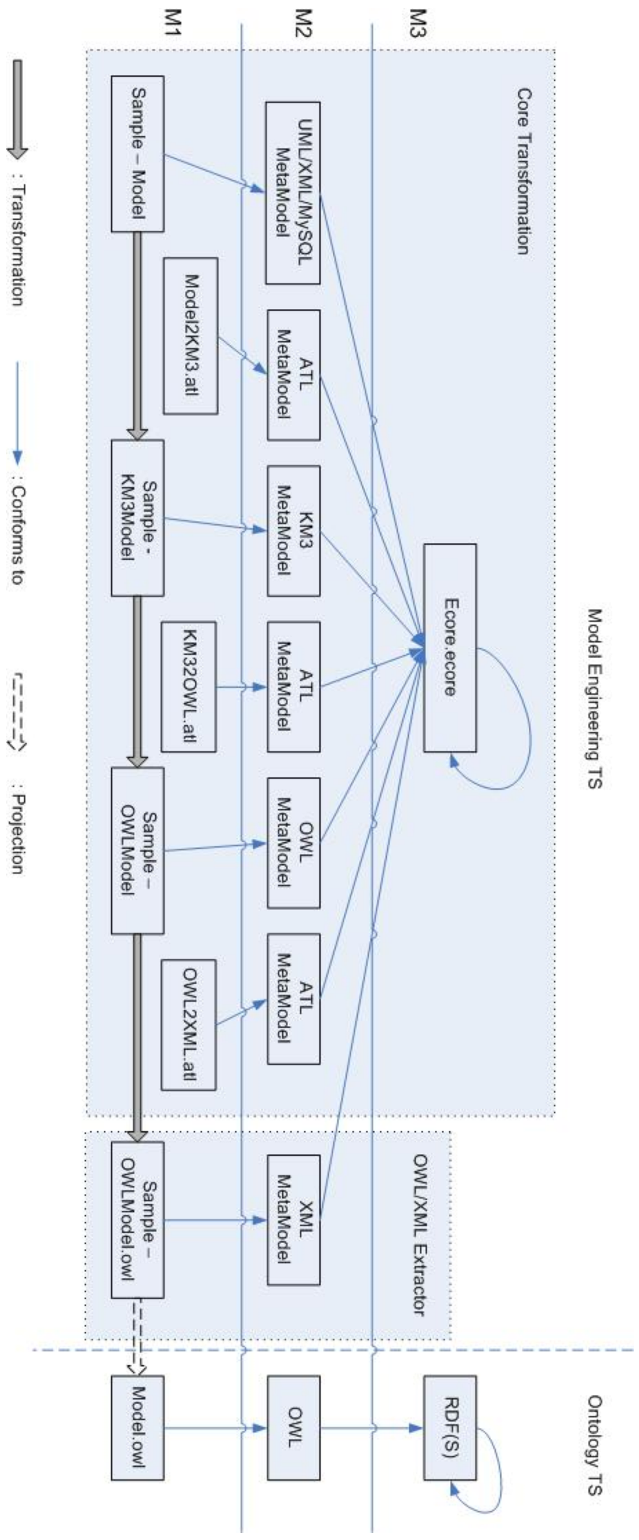
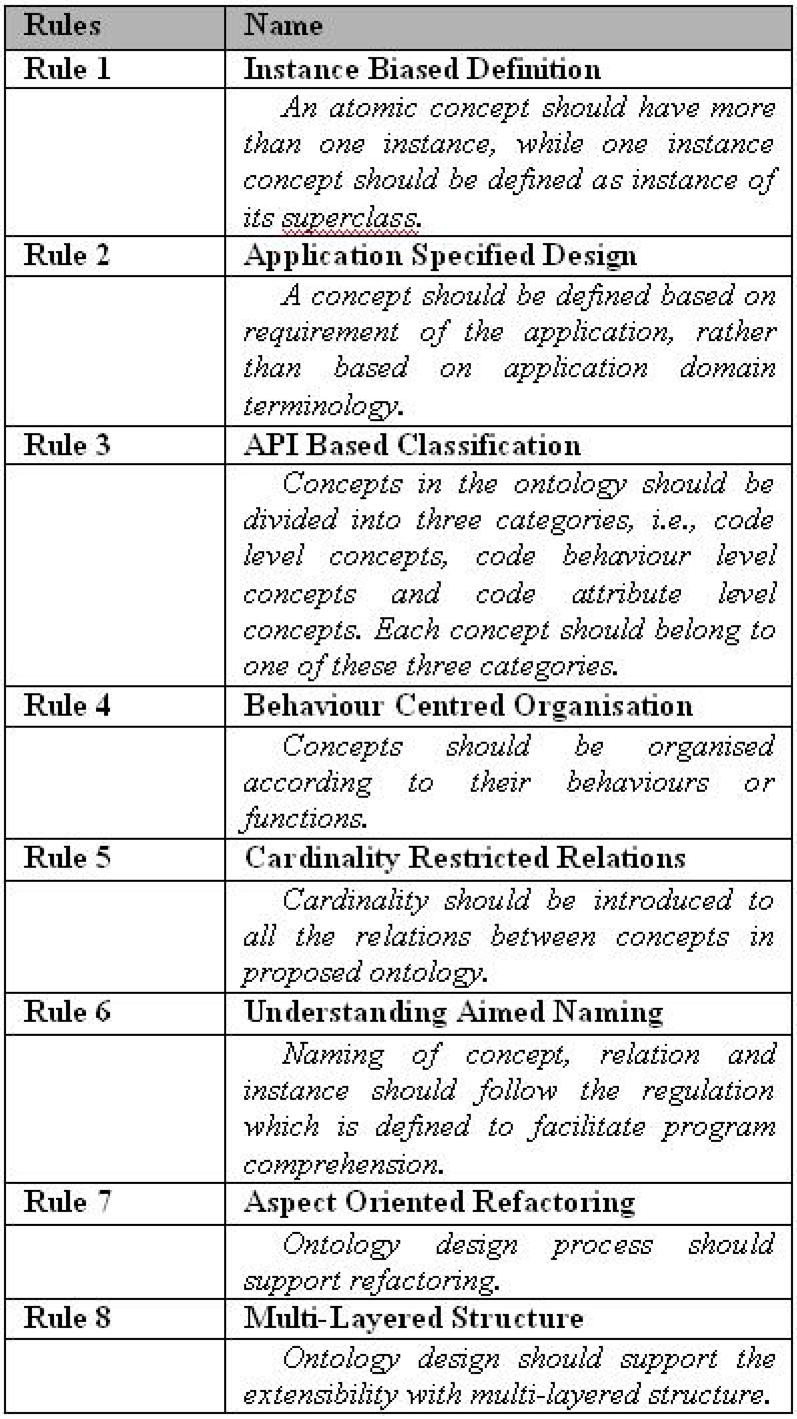


Figure 3-3 Bottom-up Ontology Generation Scenarios

3.2.4.2 Top-Down Software System Ontology Generation

The first step of the “top-down” approach is to build a blueprint by defining a set of core principles that will guide the software system ontology development process. For instance, when building operating system ontology, there are many aspects that could be used to represent the system, such as system components, system architecture and system services, etc. Which aspect should be represented in ontology will be the first question to answer. The top-down ontology development approach starts with identifying purposes and potential users of the ontology. In this study, software migration/porting has been set up as a relevant software reengineering scenario. Therefore, the operating system ontology should be developed to include the main concepts related to software migration/porting correspondingly. System call interfaces are selected as core concepts in this ontology. Table 3-1 describes a set of ontology development rules that has been defined in line with the purpose of software migration/porting [132]. Chapter 4 will detail each principle by giving examples of its usage in ontology development practice. Contrary to the bottom-up approach, the top-down approach requires more human effort, the ontology will be created manually if needed with respect to those guiding principles.



Rules	Name
Rule 1	**Instance Biased Definition**
	An atomic concept should have more than one instance, while one instance concept should be defined as instance of its superclass.
Rule 2	**Application Specified Design**
	A concept should be defined based on requirement of the application, rather than based on application domain terminology.
Rule 3	**API Based Classification**
	Concepts in the ontology should be divided into three categories, i.e., code level concepts, code behaviour level concepts and code attribute level concepts. Each concept should belong to one of these three categories.
Rule 4	**Behaviour Centred Organisation**
	Concepts should be organised according to their behaviours or functions.
Rule 5	**Cardinality Restricted Relations**
	Cardinality should be introduced to all the relations between concepts in proposed ontology.
Rule 6	**Understanding Aimed Naming**
	Naming of concept, relation and instance should follow the regulation which is defined to facilitate program comprehension.
Rule 7	**Aspect Oriented Refactoring**
	Ontology design process should support refactoring.
Rule 8	**Multi-Layered Structure**
	Ontology design should support the extensibility with multi-layered structure.

Table 3-1 Operating System Ontology Development Rules [132]

3.2.5 Integrating – Integrating Software System Ontology

Before implementing software system ontology on software reengineering tasks, ontologies generated from different information sources will need to be integrated in order to achieve consensus between divergent views of the software system. Software system ontology integration is another crucial part of this study. Ontology integration has many synonyms in the ontology engineering research arena. Generally speaking, software system ontology integration indicates ontology mapping, which is a process of finding semantic relationships between the notions (e.g., concepts, relations, etc.) of two different software system ontologies. Software system ontologies always have three different levels of knowledge, these are lexical knowledge, domain knowledge and structural knowledge. Lexical knowledge is about the semantic meanings of the terms that are used to describe the software system. For example WordNet is used as lexical knowledge in this algorithm. Domain knowledge is about the terms that are used to describe the specific domain in the real world. Structural knowledge is about the structures on which all the terms are organised in the software system, such as inheritance relationships and complicated binary relationships, etc. In other words, the hierarchical classification of software ontology contains structural knowledge. Description Logic is employed to represent all three levels of software system knowledge in logical formulae, and therefore to transform the problem of seeking semantic relationships between terms across different ontologies into deducing the satisfiabilty of logical formulae that are represented by Description Logic.

3.2.6 Software System Ontology Deployment

Software system ontology deployment is the implementation of software system ontology in the proposed software reengineering scenarios in which a knowledge based approach can be employed to semi-automate some of the reengineering processes. There are two main potential uses of software system ontology in reengineering scenarios, namely, program comprehension and software migration/porting. The following two sections will briefly discuss deploying software system ontology in these two reengineering activities respectively.

3.2.6.1 Ontology-Base Program Comprehension

Figure 3-4 depicts a program comprehension process for a small software system by deploying class diagram ontology and application domain ontology. The class diagram ontology represents all the knowledge in the code level, while the domain ontology represents a large number of key concepts in the problem domain. Hence, this ontology deployment for software reengineering will provide the following functionalities for program comprehension.

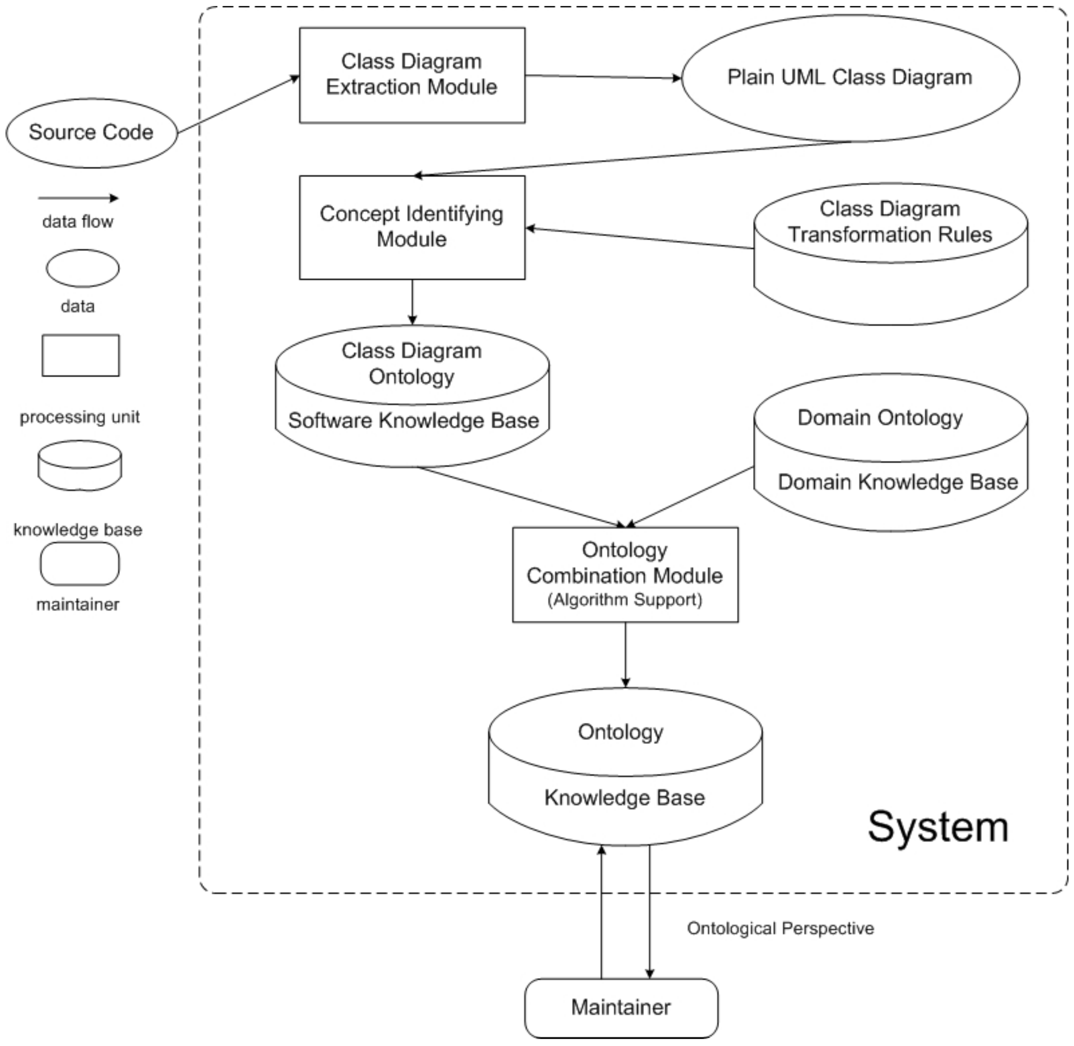


Figure 3-4 Program Comprehension by Deploying Code Ontology and Domain Ontology

Concept Recovery When one concept in domain ontology is matched to one concept in class diagram ontology, a neighbourhood (all the filler concepts of binary relations) analysis can be performed from both ontologies to identify and understand the neighbour of these two concepts. Thus, if a class in class diagram ontology is not given a meaningful name, it will still be able to be understood by matching to the concept in domain ontology.

Relation Specification The associations between two classes in class diagram ontology can be described as the matching of these two concepts to domain ontology and exploring the relationships between them in domain ontology. In addition, since the operations of object-oriented class will be transformed into properties of ontology in the next stage of this research, the semantics of some operations will be derived from domain ontology as well.

Design Defect Detection Domain ontology is on an abstract level, while class diagram ontology is on an implementation level. Having both of these two levels represented in ontologies will somehow bridge the gap between two different representation levels. Observing the differences between the abstract level and the implementation level will give the maintainer some useful information, and it will help to detect design defects of the system implementation as well.

Domain Understanding By mapping class diagram ontology and domain ontology, it will allow the software maintainer to understand the software system as a domain expert. Compared with source code and some forms of its representation such as class diagram and AST, domain ontology describes the things that the software could do. Hence, it will be much easier for the maintainer to understand the software.

On the other hand, Figure 3-5 represents a program understanding process in a relatively large scale software system, e.g., an enterprise software system, through deploying class diagram ontology, data ontology and application framework ontology.

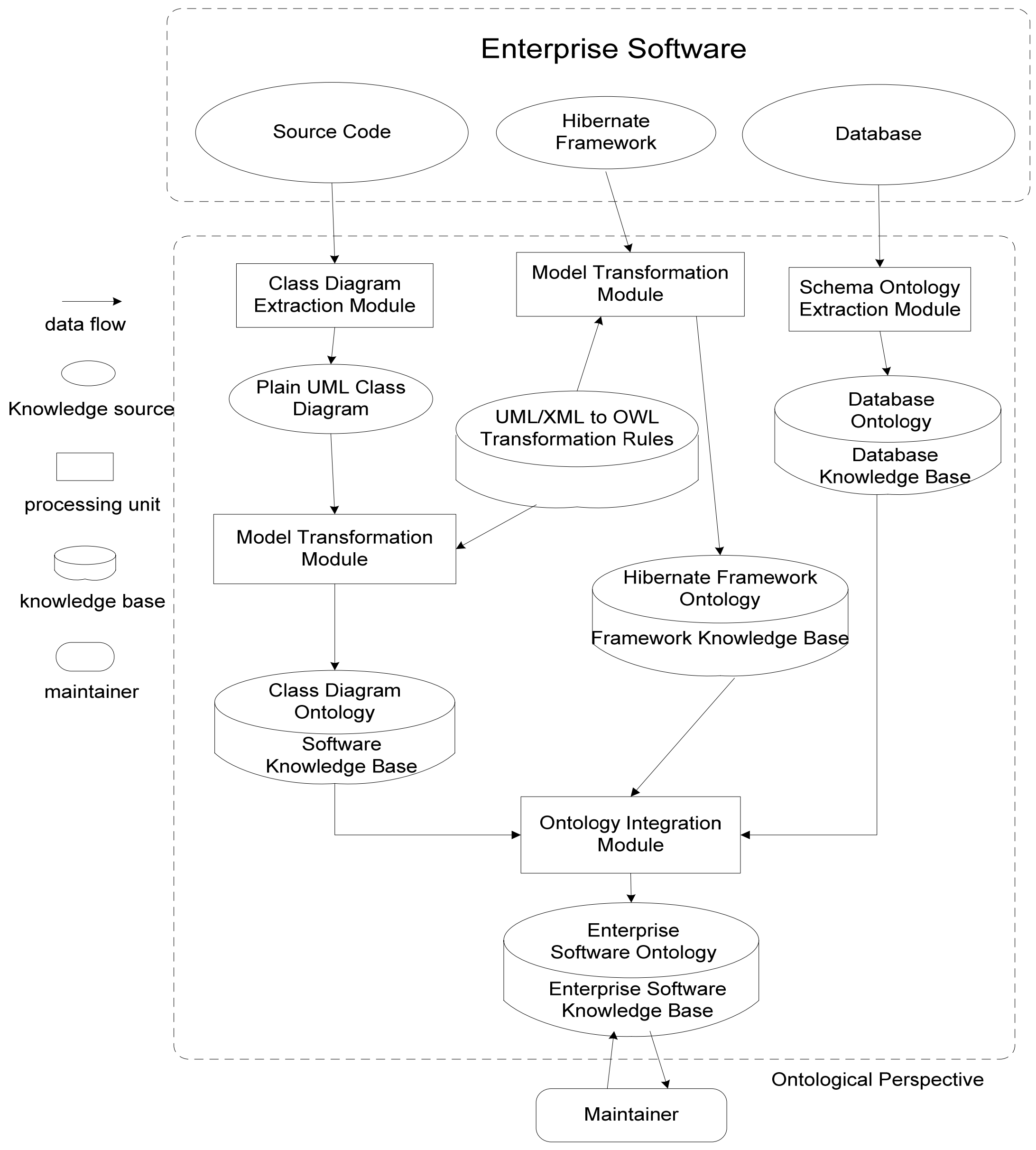


Figure 3-5 Program Comprehension by Deploying Code Ontology, Database Ontology and Hibernate ORM Framework Ontology

Service Identification The goal of deploying enterprise software ontology is to achieve a more comprehensive representational form of the software and then to use this new representational form to identify potential service candidates from legacy system for Service Oriented Architecture (SOA). The identification of SOA service candidates requires a decomposition of the software system with respect to the following principles, namely, loosely coupled components, and reusable functionalities. Therefore, the components and their interrelationships in the software need to be analysed, and the strongly related components need to stay together, while loosely coupled ones can be apart. In this reengineering scenario, the decomposition of enterprise software is accomplished by modularising enterprise software ontology, which is also known as an ontology partitioning. Currently, there are several studies on ontology partitioning [73, 102], this paper will adopt the structure-based partitioning algorithm proposed by Schlicht and Stuckenschmidt [102]. Their partitioning algorithm is based on the structural dependencies between concepts in ontology, which are represented through a weighted dependency graph. Then the strength of the dependencies between the concepts is calculated and the proportional strength network is obtained to detect sets of strongly related concepts. As a result, the concepts which are stronger related will be modularised and the original ontology will be divided into loosely coupled partitions. After applying a structure-based partitioning algorithm, the enterprise software ontology can be decomposed into a few modules. For each module, all the concepts are strongly related, and are organised around specific functions and domain concepts. Furthermore, all the modules are loose coupled. Hence, they can be considered to be potential service candidates in relation to the SOA environment.

3.2.6.2 Ontology-Base Software Migration/Porting and Portable Software Development

Figure 3-6 demonstrates an often-occurring reengineering scenario related to Real Time Operating System (RTOS) specific software migration/porting, the software migration/porting between two different platforms, e.g., from RTLinux to ThreadX. In this situation, when a software application is migrated from one RTOS platform to another, system APIs will play a crucial part in the migration process. Different RTOS platforms provide different APIs. Since the Portable Operating System Interface (POSIX) standard contains most of the standard UNIX compatible system call interfaces, many RTOS platforms support subsets of the POSIX standard. To transform a program automatically while keeping certain properties invariant, transformation rules need to be defined. Program transformation depends on matching detection. If inputs are matched with predefined patterns, the system will be rewritten according to the transformation rules. Based on the proposed operating system ontology and knowledge acquisition methodologies, knowledge based program transformation rules are defined for software migration between different platforms. If both source and target API belong to the operating system ontology repository, transformation will be performed based on the matched transformation rules. Otherwise, transformation of source API cannot be performed automatically. Although some situations may not be processed by the match algorithm, the ontology repository provides useful information of source and target APIs which can facilitate the maintainers to redefine the transformation rules. Hence, with human intervention, most APIs can be transformed.

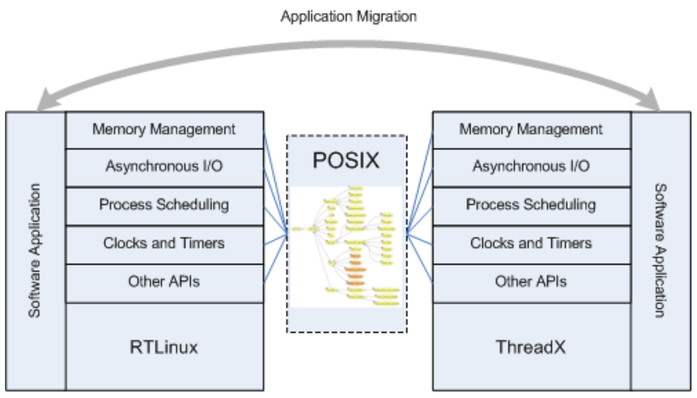


Figure 3-6 RTOS Specific Software Migration

Figure 3-7 still describes the issues in software reengineering related to developing a more portable software system. There are a wide variety of options available for the development of portable software applications. The essence of such development has always been related to standardisation, normally implemented by abstraction and isolation. Software middleware has proved to be an efficient and practical standardisation approach. As middleware, the Virtual Operating System (VOS) has successfully disentangled computing environments from their underlying operating system. Hence, the underlying operating system becomes totally transparent to the software applications, which therefore improves the portability of software applications. Through the OS ontology and knowledge acquisition, the functional equivalence of different operating systems can be established by defining and implementing a set of common system services. These system services can be separated into two types: platform independent services and platform specific services. However, when diverse operating systems share a collection of common user interfaces, a uniform environment is required, while the variants of the OS will destroy this uniformity. Even though the OS ontology provides good mechanisms to mange these variants, to reduce the proliferation of such variants, the VOS should be applied to a small specific application domain, rather than a large range of operating system environments.

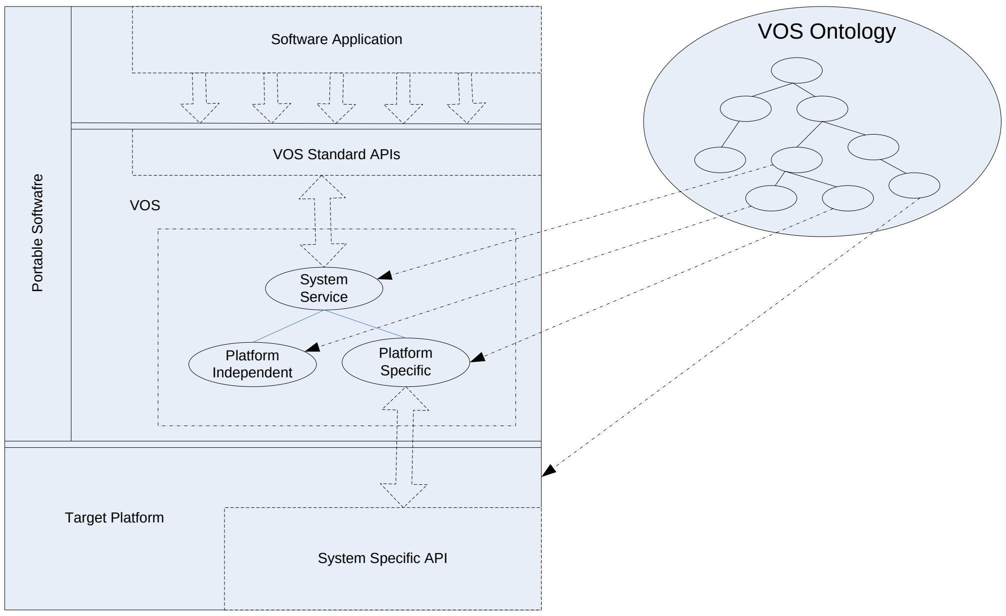


Figure 3-7 Ontology Based VRTOS Design

3.3 Summary

In this chapter a knowledge based software reengineering approach has been proposed, in which ontology and description logic are employed as a means to represent software systems in order to facilitate software reengineering projects:

- An ontology based software reengineering framework is developed in this study, which consists of three main parts in the context of ontology engineering, namely, software system ontology generation, software system ontology integration and software system ontology deployment.
- The scope of the proposed approach has been discussed based on the software taxonomy. Business-oriented software from the data-dominant software category

and operating systems from the system software category are selected as the main vertical subject domains. Application framework, data and source code are chosen as the main horizontal subject domains.

- The ontology based software reengineering process has been defined to support the proposed approach. Preparation, capture, coding, integration and deployment are defined as sequential processes to implement ontology based reengineering.

- The capture of software system ontology has been discussed regarding different types of knowledge in software systems, i.e., domain application knowledge, software engineering knowledge and code knowledge.

- The generation of software system ontology can be divided into two different approaches, i.e, bottom-up approach and top-down approach. The bottom-up approach is based on reverse engineering and model transformation techniques, which transforms software "legacy" to ontology. The top-down approach follows a set of predefined ontology design rules, which develops ontology from scratch.

- The integration of software system ontology has been discussed. Software system ontology integration indicates ontology mapping, which is a process of finding semantic relationships between notions (e.g., concepts, relations, etc.) of two different software system ontologies. This ontology mapping process is supported by a description logic based mapping algorithm.

- Software system ontology deployment has been discussed. This is the deployment of software system ontology in software reengineering projects to semi-automate some of the reengineering processes. This study will mainly focus on two reengineering activities, i.e., program comprehension and software migration, and will explore the knowledge based approach on four selected reengineering scenarios.

Chapter 4 Software System Ontology Capture and Coding

Objectives

- To discuss the bottom-up approach for generating software system ontology
- To discuss the top-down software system ontology generation approach

The entire software life cycle has always been known as a knowledge intensive process. For example, the waterfall model represents a classic software development process. In this model, domain knowledge will be needed in the requirements phase; a great deal of software engineering knowledge will be involved in the design, implementation and verification phases; and code knowledge will be a crucial part in the maintenance phase. Therefore, capturing and coding software ontology will take into account all these different aspects. This chapter explores the methodologies for capturing and coding ontology which represents the knowledge that covers different perspectives of software systems. As discussed in Chapter 3, there are two approaches that are employed in this study to generate software system ontology, namely, bottom-up and top-down approaches. The bottom-up approach is mainly focusing on the creation of software system ontology by transformation from other knowledge forms. While the top-down approach requires a blueprint for the entire software system before starting to build ontology. The remainder of this chapter will discuss each approach in detail in line with the ontology generation processes for different software systems.

4.1 Bottom-Up Software System Ontology Generation

4.1.1 Bottom-Up Ontology Generation – In a Nutshell

A bottom-up ontology development is relatively straightforward. In a nutshell, bottom-up ontology generation is a series of model transformation processes, in which ontology is generated by transforming it from other software models. Three transformation steps are involved in this ontology generation, namely, transformation between the software model and the KM3 (Kernel MetaMetaModel) model, transformation from the KM3 model to the OWL knowledge model and transformation from the OWL knowledge model to an .owl document.

Firstly, the software model covers three different aspects: the source code model, the software framework model and the software data model. The source code model could be any representation which expresses source code in a different view, e.g., UML class diagram, Abstract Syntax Tree and control flow diagram, etc. The software framework model could be obtained from framework configuration files which are widely used in framework based software, e.g., configuration files that store mappings in the Hibernate ORM framework, etc. The software data model is extracted from a relational database, e.g., the Entity-Relationship Model, etc.

Secondly, the KM3 model is introduced as a medium of the transformation between the software model and the OWL knowledge model. As a Domain Specific Language, KM3 is developed particularly for metamodel description, and is widely used in the ATL model transformation community. Over 300 metamodels have been expressed in KM3 and are available online in Atlantic Zoo, which provides possibilities for easily extending this research into other dimensions, i.e., replacing the OWL knowledge model with different models in different model driven engineering technical spaces.

However, the OWL knowledge model is not the final product of bottom-up ontology generation, as it is described by a modelling language and it is coded in XMI format. Hence, it will require the last transformation step to be transformed to an XML file with OWL/XML syntax elements. Consequently, the generated software ontology can be manipulated by ontology editors in the next stage of this study. The following sections will discuss each transformation in detail.

4.1.2 Source Code KM3 Model Capture and Generation

In general, source code is a collection of statements used by programmers to specify the actions performed by a computer. The knowledge model of source code could cover programming language knowledge, design knowledge, structure knowledge, function knowledge, feature knowledge and so on. There are several forms of source code representation which could be considered as candidates for source code knowledge model retrieval, e.g., an Abstract Syntax Tree (AST), a control flow graph, a data flow diagram, a UML class diagram, etc., and which of these are selected in practice depends on the particular reverse engineering task. A UML Class diagram is one of the frequently used software representations to aid reverse engineering projects. The proposed approach will generate a source code knowledge model in two steps, i.e., reverse engineering source code to obtain a UML class diagram and then transforming the UML class diagram to the KM3 model by a set of predefined model transformation rules.

4.1.2.1 UML Class Diagram Extraction

The first step of the source code KM3 model capture and generation is a UML class diagram extraction via reverse engineering techniques. For the external open source toolset unit that produces a UML class diagram, there are currently many choices, covering most object-oriented languages such as Java, C ++ and C #. The majority of these tools are developed as plug-in techniques for the popular integrated development environment (IDE), e.g., Eclipse and Microsoft Visual Studio. The EclipseUML 2007 [93] (normally known as Omondo), Jupe [11], MaintainJ [66], Green UML [111] and Topcased [34] are some of the Eclipse plug-ins which are widely used in software development projects. All of them have features which allow the software maintainer to extract visual representations of software systems in the form of class diagrams. The latest version of Microsoft Visual Studio provides a class diagram extraction function as well.

After experimenting with the above reverse engineering tools, similar problems have been spotted when extracting a class diagram from source code. Some open-source reverse engineering tools will crash during the extraction operation on large volumes of complex source code. After comparison with other tools, Topcased is the most stable open-source one for large scale source code and will be used to perform UML class diagram generation in this study.

4.1.2.2 UML Class Diagram to KM3 model Transformation

The transformation from a UML class diagram to a KM3 model can be performed automatically by an ATL model transformation. A set of UML class diagram to KM3 model transformation rules are written based on the comparison between the UML class diagram metamodel and the KM3 metamodel, both of which are expressed in KM3.

The crucial ATL rules for transforming a UML class diagram to the KM3 model is described bellow:

Rule 1: Transforming a UML Package to the KM3 Package

Nestedpackage and ownedmember of UML class diagram will be transformed to a set of contents of KM3 Package.

ATL Transformation:

```
rule UML2KM3Package {
    from
        s : UML!Package (not s.oclIsTypeOf(UML!Model))
    to
        t : KM3!Package (
            name <- s.name
            contents <- Set {s.nestedPackage, s.ownedMember}
        )
}
```

In this model transformation, the source pattern is the Package of a UML class diagram, and it is not a Model of a UML class diagram. The target pattern is the Package of KM3. The name of the UML Package will become the name of the KM3 Package. The nestedPackage and ownedMember of the UML Package will become the contents of the

KM3 Package.

Rule 2: Transforming UML Class to KM3 Class

Super class of UML class is transformed to super type of KM3 class; attribute of UML class is transformed to structural features of KM3 class; isAbstract of UML class is transformed to isAbstract of KM3 class.

ATL Transformation:

```
rule UML2KM3Class {
    from
        s : UML!Class
    to
        t : KM3!Class (
            name <- s.name,
            supertypes <- s.superClass,
            isAbstract <- s.isAbstract,
            structuralFeatures <- Set {s.attribute}
        )
}
```

In this model transformation, the source pattern is the Class of the UML class diagram. The target pattern is the Class of KM3. The name of the UML Class will become the name of the KM3 Class; the superClass of the UML Class will become the supertypes of the KM3 Class; the attribute isAbstract of UML Class will become the same attribute of the KM3 Class; and the attribute of the UML Class will become the structuralFeatures of the KM3 Class.

Rule 3: Transforming UML Datatype to KM3 Datatype

Datatype of UML class diagram is transformed to Datatype of KM3 directly.

ATL Transformation:

```
rule UML2KM3DataType {
    from
        s : UML!DataType
    to
        t : KM3!DataType(
        name <- s.name
        )
}
```

In this model transformation, the source pattern is the DataType of the UML class diagram. The target pattern is the DataType of KM3. The UML DataType will become the KM3 DataType directly by copying the name.

Rule 4: Transforming UML Enumeration to KM3 Enumeration

Enumeration of UML class diagram is transformed to Enumeration of KM3 by copying ownedLiteral of UML Enumeration to literals of KM3 Enumeration.

ATL Transformation:

```
rule UML2KM3Enumeration {
    from
        s : UML!Enumeration
    to
        t : KM3!Enumeration(
        name <- s.name,
        literals <- s.ownedLiteral
        )
}
```

In this model transformation, the source pattern is the Enumeration of the UML class diagram. The target pattern is the Enumeration of KM3. The name of the UML Enumeration will become the name of the KM3 Enumeration. The ownedLiteral of the UML Enumeration will become the literals of the KM3 Enumeration.

Rule 5: Transforming UML StructuralFeature to KM3 StructuralFeature

StructuralFeature of UML class diagram is transformed to StructuralFeature of KM3 directly by copying names and other attributes.

```
rule UML2KM3StructuralFeature {
    from
        s : UML!StructuralFeature
    to
        t : KM3!StructuralFeature(
            name <- s.name,
             lower <- s.lower,
            upper <- s.upper,
            isOrdered <- s.isOrdered,
            type <- thisModule.getType(s)
        )
}
```

In this model transformation, the source pattern is the StructuralFeature of the UML class diagram. The target pattern is the StructuralFeature of KM3. The name of the UML StructuralFeature will become the name of the KM3 StructuralFeature; the lower and upper of the UML StructuralFeature will become the lower and upper of the KM3 StructrualFeature; the attribute isOrdered of UML StructuralFeature will become the same attribute of the KM StructrualFeature.

Rule 6: Transforming UML Property to KM3 Attribute

Property of UML class diagram is transformed to attribute of KM3 directly.

```
rule UML2KM3Attribute {
    from
        s : UML!Property (s.association.oclIsUndefined())
    to
        t : KM3!Attribute(
        name <- s.name
        )
}
```

In this model transformation, the source pattern is the Property of the UML class diagram, and it is not an Association. The target pattern is the Attribute of KM3. The UML Property will become the KM3 Attribute directly by copying the name.

4.1.2.3 Source Code KM3 Model Capture and Generation – An Example

The following is a java class represented by a UML class diagram. This class diagram is extracted by Topcased from a sample of java code that is picked up from an open source enterprise application. The class is called Employee, which implements the function of employee management. For lack of space, some elements have been removed and replaced with "...".

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="2.1"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
xmlns:uml="http://www.eclipse.org/uml2/2.1.0/UML"
xmi:id="_MvvgwWZXEd-S3-0npnXdRQ">
  <packagedElement xmi:type="uml:Class" xmi:id="..." name="Employee">
    <generalization xmi:id="..." general="_MvvhYmZXEd-S3-0npnXdRQ"/>
       <ownedAttribute xmi:id="..." name="CLASS_ID" visibility="public"
        type="..."/>
       <ownedAttribute xmi:id="..." name="startDate"
```

```
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="endDate" visibility="private"
         type="..."/>
        <ownedAttribute xmi:id="..." name="tax" visibility="private"
         type="..."/>
        <ownedAttribute xmi:id="..." name="employeeCategory"
         visibility="private" type="..." association="..."/>
        <ownedAttribute xmi:id="..." name="employeeRank"
         visibility="private" type="..." association="..."/>
        <ownedAttribute xmi:id="..." name="payrollForm"
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="employeeAccount"
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="taxPrivilege"
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="salary" visibility="private"
         type="..."/>
        <ownedAttribute xmi:id="..." name="tariff" visibility="private"
         type="..."/>
        <ownedAttribute xmi:id="..." name="advance" visibility="private"
         type="..."/>
        <ownedAttribute xmi:id="..." name="premiumPercent"
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="sickPercent"
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="totalSeniorityYear"
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="totalSeniorityMonth"
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="totalSeniorityDay"
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="unbrokenSeniorityYear"
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="unbrokenSeniorityMonth"
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="unbrokenSeniorityDay"
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="contactableElement"
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="businessableElement"
         visibility="private" type="..."/>
        <ownedAttribute xmi:id="..." name="ledgerAccount"
         visibility="private" type="..."/>
    </packagedElement>
</uml:Model>
```

A KM3 model is produced by applying ATL model transformation Rule 1 – Rule 6 created in the above section.

```
package Employee {

    class Employee {
        attribute CLASS_ID: String;
        attribute startDate: Date;
        attribute endDate: Date;
        attribute tax: boolean;
        attribute payrollForm: PayrollForm
        attribute employeeAccount: String;
        attribute taxPrivilege: int;
        attribute salary: double;
        attribute tariff: double;
        attribute advance: double;
        attribute premiumPercent: float;
        attribute sickPercent: float;
        attribute totalSeniorityYear: int;
        attribute totalSeniorityMonth: int;
        attribute totalSeniorityDay: int;
        attribute contactableElement: ContactableElement;
        attribute businessableElement: BusinessableElement;
        attribute ledgerAccount: LedgerAccount;
        reference employeeCategory: EmployeeGategory;
        reference employeeRank: EmployeeRank;
    }

}
```

This KM3 model will be encoded in XMI format as following. For lack of space, some elements have been removed and replaced with "...".

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:km3="http://www.eclipse.org/gmt/2005/KM3">
  <km3:Metamodel>
    <contents name="Employee">
      <contents xsi:type="km3:Class" name="Employee" supertypes="...">
        <structuralFeatures xsi:type="km3:Attribute" name="CLASS_ID"
         lower="1" upper="1" type="..."/>
         <structuralFeatures xsi:type="km3:Attribute" name="startDate"
         lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute" name="endDate"
         lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute" name="tax" lower="1"
         upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute" name="payrollForm"
         lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute"
         name="employeeAccount" lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute" name="taxPrivilege"
         lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute" name="salary"
         lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute" name="tariff"
         lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute" name="advance"
         lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute"
         name="premiumPercent" lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute" name="sickPercent"
         lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute"
         name="totalSeniorityYear" lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute"
         name="totalSeniorityMonth" lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute"
         name="totalSeniorityDay" lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute"
         name="contactableElement" lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute"
         name="businessableElement" lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Attribute"
         name="ledgerAccount" lower="1" upper="1" type="..."/>
        <structuralFeatures xsi:type="km3:Reference"
         name="employeeCategory" lower="1" upper="-1" type="..."
         isContainer="true"/>
        <structuralFeatures xsi:type="km3:Reference"
         name="employeeRank" lower="1" upper="-1" type="..."
         isContainer="true"/>
       </contents>
   </km3:Metamodel>
   <km3:Reference name="Employee" lower="1" upper="1" type="..."
    opposite="/0/@contents.0/@contents.0/@structuralFeatures.9"/>
 </xmi:XMI>
```

The above KM3 model is stored in XMI format. It is the final product of a souce code knowledge model generation step, which will be transformed to the OWL model at the next stage.

4.1.3 Database KM3 Model Capture and Generation

Generally speaking, as part of a software application, a database always plays an important role. When reengineering software systems, analysing the related database becomes inevitable because of the complicated connection between source code and database. Such connection stores explicit knowledge about the system, which could be used to assist program comprehension. In addition, understanding the system database also provides further assistance for the maintenance tasks such as system integration. At this point, only the database schema is being used in the proposed approach. The MySQL database is selected as the main example. The retrieval of the database KM3 model could be divided into two steps. Firstly, a MySQL model will be extracted based on a text description of the database schema. Secondly, the MySQL model will be transformed to the KM3 model. The following section will discuss how to extract a KM3 model from a MySQL database schema.

4.1.3.1 A Model of a MySQL Database

Developing a model of a MySQL database is a relatively straight forward process. An XML file is selected as a text description for a MySQL database, which encodes the structure of a MySQL database. MySQL Structure Magic [95] is an open source PHP class that could export a MySQL database schema to XML format, which is used to create a source model of database knowledge model generation in this study.

A MySQL model is created by transformation from an XML database text description file. The following are the main ATL model transformation rules to support this process:

Rule 7: Creating a Database of MySQL model from XML database text description files

Database of MySQL model is created with the XML Element Root by copying name and all the instances that has name 'DATAONTO_TABLE' to tables.

```
rule XML2DBModelDataBase {
    from
        s : XML!Root
    to
        t : MySQL!DataBase (
            name <- s.getAttrVal('name'),
            tables <- XML!Element.allInstances()
                         ->select(e | e.name = 'DATAONTO_TABLE')
        )
}
```

In this model transformation, the source pattern is the Root of XML file. The target pattern is the DataBase of MySQL model. The name attribute value of Root will become the name of MySQL DataBase. The name values of the XML element DATAONTO_TABLE will become the names of the tables of MySQL DataBase.

Rule 8: Creating a Table of MySQL model from XML database text description files

Table of MySQL model is created with the XML Element that has name 'DATAONTO_TABLE' by copying Elements that have name'TableInfoTable' to columns.

```
rule XML2DBModelTable {
   from
      s : XML!Element ( s.name = 'DATAONTO_TABLE' )
   to
      t : MySQL!Table (
         name <- s.getAttrVal('name'),
         columns <-s.getElementsByName('TableInfoTable')
               ->asSequence()
               ->select(e | e.getFirstElementByName('Type')
                 .getTextValue().startsWith('tinyint')),
         database <- thisModule.rootElt
      )
}
```

In this model transformation, the source pattern is the Element of XML file whose name is DATAONTO_TABLE. The target pattern is the Table of MySQL. The name of the XML Element will become the name of MySQL Table. The XML Element Type of the Element TableInfoTable will become the columns of MySQL Table. The rootElt of DATAONTO_TABLE will become database of MySQL Table.

Rule 9: Creating a Column of MySQL model from XML database text description files

Column of MySQL model is created with XML element named 'TableInfoTable'.

```
rule XML2DBModelColumn {
   from
      s : XML!Element ( s.name = 'TableInfoTable' )
   to
      t : MySQL!Column (
         name <- s.getFirstElementByName('Field').getTextValue(),
         type <- s.getFirstElementByName('Type')
               .getTextValue().getTypeName(),
         isPrimaryKey <- s.getFirstElementByName('Key')
                     .getTextValue() = 'PRI',
         null <- s.getFirstElementByName('Null')
               .getTextValue() = 'YES',
         defaultValue <- s.getFirstElementByName('Default')
                     .getTextValue(),
         comment <- s.getFirstElementByName('Comment').getTextValue(),
         table <- s.parent
      )
}
```

In this model transformation, the source pattern is the Element of XML file whose name is TableInfoTable. The target pattern is the Column of MySQL. The name, type, isPrimaryKey, defaultValue and table of MySQL Column are obtained from the information stored in the XML Element whose name is TableInforTable. With the above three main transformations, a MySQL XML description will be transformed into a MySQL model stored in XMI format, which will be used as the input of next model transformation step, namely, MySQL model to KM3 model transformation.

4.1.3.2 MySQL Database Model to KM3 model Transformation

The last step of creating a MySQL database knowledge model is to transform the MySQL database model to KM3 model. The main ATL model transformation rules are given below.

Rule 10: Creating a Class of KM3 model from a Table of MySQL model.

```
rule DBModel2KM3Class {
   from
      s : MySQL!Table
   to
      t : KM3!Class (
         location <- '',
         name <- s.name,
         package <- thisModule.resolveTemp(thisModule.dataBaseElt,
                    'p'),
         isAbstract <- false,
         supertypes <- Set{},
         structuralFeatures <- s.columns,
         operations <- Sequence{}
      )
}
```

In this model transformation, the source pattern is the Table of MySQL. The target pattern is the Class of KM3. The name of MySQL Table will become the name of the KM3 Class; the dataBaseElt of MySQL Table will become the package of KM3 Class; the KM3 Class will not be Abstract; the columns of MySQL Table will become structuralFeatures of KM3 Class.

Rule 11: Creating an Attribute from a Column.

```
rule DBModel2KM3Attribute {
   from
      s : MySQL!Column
   to
      t : KM3!Attribute (
         location <- '',
         name <- s.name,
         package <- OclUndefined,
         lower <- 1,
         upper <- 1,
         isOrdered <- false,
         isUnique <- false,
         type <- d,
         owner <- s.table,
         subsetOf <- Set{},
         derivedFrom <- Set{}
      ),
      d : KM3!DataType (
         location <- '',
         name <- s.type.getKM3TypeName(),
         package <- thisModule.resolveTemp(thisModule.dataBaseElt,
                  'pt')
      )
}
```

In this model transformation, the the source pattern is the Column of MySQL. The target pattern is the Attribute of KM3. The name of the MySQL Column will become the name of KM3 Attribute; the location of KM3 Attribute will be ‘ ’, the package of KM3 Attribute will be undefined; the lower of KM3 Attribute will be 1, the upper of KM3 Attribute will be 1; the KM3 Attribute is not ordered; the KM3 Attribute is not unique; the type of KM3 Attribute will be KM3 DataType.

Rule 12: Creating a Reference of KM3 model from Column of MySQL model.

```
rule DBModel2KM3Reference {
    from
        s : MySQL!Column
    to
        t : KM3!Reference (
            location <- '',
            name <- s.name,
            package <- OclUndefined,
            lower <- 1,
            upper <- 1,
            isOrdered <- false,
            isUnique <- false,
            type <- s.getReferredTable,
            owner <- s.table,
            subsetOf <- Set{},
            derivedFrom <- Set{},
            isContainer <- false,
            opposite <- OclUndefined
        )
}
```

In this model transformation, the the source pattern is the Column of MySQL. The target pattern is the Reference of KM3. The name of the MySQL Column will become the name of KM3 Reference; the location of KM3 Reference will be ‘ ’, the package of KM3 Reference will be undefined; the lower of KM3 Reference will be 1, the upper of KM3 Reference will be 1; the KM3 Reference is not ordered; the KM3 Reference is not unique; the type of KM3 Reference will be the type of MySQL Table; the opposite will be undefined.

Rule 13: Creating an Enumeration of KM3 model from an EnumSet of MySQL model.

```
rule DBModel2KM3Enumeration {
    from
        s : MySQL!EnumSet
    to
        t : KM3!Enumeration (
            location <- '',
            name <-'Enum_'.concat(thisModule
                       .enumSet->indexOf(s).toString()),
            package <- thisModule.resolveTemp(thisModule.dataBaseElt,
                        'p'),
            literals <- s.enumItems
        )
}
```

In this model transformation, the source pattern is the EnumSet of MySQL. The target pattern is the Enumeration of KM3. The names of the KM3 Enumeration will be obtained from MySQL EnumSet by toString() method; the dataBaseElt of MySQL EnumSet will become the package of KM3 Enumeration; the enumItems of MySQL EnumSet will become the literals of KM3 Enumeration.

4.1.3.3 Database KM3 Model Capture and Generation – An Example

The following is an example MySQL database schema which has been extracted in XML format by the PHP class MySQL Structure Magic. For lack of space, some elements have been removed and replaced with “...”.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<DATAONTO_DATABASE name="Plazma">
<DATAONTO_TABLE name="employee">
  <TableInfoTable>
    <Field>ID</Field>
    <Type>tinyint(11) unsigned</Type>
    <Null></Null>
    <Key>PRI</Key>
    <Default></Default>
    <Extra>auto_increment</Extra>
    <Index_length>0</Index_length>
    <Data_free>0</Data_free>
    <Auto_increment>0</Auto_increment>
    <Create_time></Create_time>
    <Update_time></Update_time>
    <Check_time></Check_time>
    <Create_options></Create_options>
    <Comment></Comment>
  </TableInfoTable>

  … …

  <TableInfoTable>
    <Field>SALARY</Field>
    <Type>decimal(15,2)</Type>
    <Null></Null>
    <Key></Key>
    <Default>0.00</Default>
    <Extra>auto_increment</Extra>
    <Index_length>0</Index_length>
    <Data_free>0</Data_free>
    <Auto_increment>0</Auto_increment>
    <Create_time></Create_time>
    <Update_time></Update_time>
    <Check_time></Check_time>
    <Create_options></Create_options>
    <Comment></Comment>
  </TableInfoTable>
</DATAONTO_TABLE>
</DATAONTO_DATABASE>
```

Through ATL model transformation discussed in previous section, this XML database schema can be represented by KM3 model as following. For lack of space, some elements have been removed and replaced with "...".

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Metamodel xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="KM3" location="">
  <contents location="" name="Plazma">
    <contents xsi:type="Class" location="" name="employee">
      <structuralFeatures xsi:type="Attribute" location="" name="ID"
       lower="1" upper="1" type="..."/>
      <structuralFeatures xsi:type="Attribute" location=""
       name="ORGANIZATION_ID" lower="1" upper="1" type="..."/>
      <structuralFeatures xsi:type="Attribute" location=""
       name="PERSON_ID" lower="1" upper="1" type="..."/>

       … …

      <structuralFeatures xsi:type="Attribute" location=""
       name="START_DATE" lower="1" upper="1" type="..."/>
      <structuralFeatures xsi:type="Attribute" location="" name="SALARY"
       lower="1" upper="1" type="..."/>
    </contents>
    <contents xsi:type="Class" location="" name="employee_move">
      <structuralFeatures xsi:type="Attribute" location="" name="ID"
       lower="1" upper="1" type="..."/>
      <structuralFeatures xsi:type="Attribute" location=""
       name="OWNER_ID" lower="1" upper="1" type="..."/>

      … …

      <structuralFeatures xsi:type="Attribute" location="" name="AMOUNT"
       lower="1" upper="1" type="..."/>
      <structuralFeatures xsi:type="Attribute" location=""
       name="TAX_AMOUNT" lower="1" upper="1" type="..."/>
    </contents>
    <contents xsi:type="Class" location="" name="employee_payroll">
      <structuralFeatures xsi:type="Attribute" location="" name="ID"
       lower="1" upper="1" type="..."/>
      <structuralFeatures xsi:type="Attribute" location=""
       name="BRANCH_ID" lower="1" upper="1" type="..."/>

       … …

      <structuralFeatures xsi:type="Attribute" location=""
       name="PERCENT" lower="1" upper="1" type="..."/>
      <structuralFeatures xsi:type="Attribute" location=""
       name="OVERRIDE_MODE" lower="1" upper="1" type="..."/>
    </contents>
  </Metamodel>
```

4.1.4 Capture and Generation of the Software Framework KM3 Model

An application framework is a pre-defined program structure or a set of reusable common code hidden behind well-defined APIs, which provide generic functionalities. Using application framework can reduce development time by providing reusable generic function code thereby allowing programmers to spend more time on system requirements instead of low-level system implementations.

To fit the proposed approach and narrow down the research scope, this study will mainly focus on a database related software framework Hibernate ORM (Object Relational Mapping) framework. Hibernate framework uses XML documents to store the configuration information about persistent classes such as how the classes map to tables or columns in a database. With these XML mapping documents, Hibernate will be able to generate SQL at runtime and free programmers from writing SQL statements in the code. Hibernate framework normally provides programmers with the following advantages:

- Defining a set of APIs which can access and manipulate database
- Mapping code objects to database objects
- Providing SQL-like query languages which perform general database operations

Session, SessionFactory, Configuration, Transaction, Query and Criteria are the main concepts in Hibernate framework in terms of programming interfaces. These interface concepts are the first things to study in order to use Hibernate in layered architectures, however, the proposed approach is currently focusing on the usage of the XML mapping documents that Hibernate uses to associate code and database.

It is believed that obtaining the mapping knowledge from Hibernate framework can help building the connection between the source code knowledge model and the database knowledge model, which will provide the following advantages for the proposed approach:

- Simplifying the Ontology integration process developed in this study by providing existing links between two ontologies
- Assuring that the software modification is being carried out in a more secure way by providing a reasoning service to infer and avoid ripple effect.

Section 4.1.4.1 will discuss the ATL model transformation rules that are created for the retrieval of the software framework knowledge model in detail.

4.1.4.1 Transforming XML Hibernate Framework Mapping Files to the KM3 model

This section will discuss the process of transforming Hibernate ORM Framework XML mapping files to the KM3 model. The ATL model transformation rules are given below.

Rule 14: Transforming Element of XML mapping file to Source Code Class of KM3 model

```
rule HF2KM3SourceCodeClass {
   from
       s : XML!Element (
          s.oclIsTypeOf(XML!Element) and
          s.name = 'class'
       )
   to
       t : KM3!Class (
          name <- s.attribute.name
   )
}
```

In this model transformation, the source pattern is the Element of XML file, whose name is class. The target pattern is the Class of KM3. The XML Element will become the KM3 Class by copying the name of the attribute of XML Element.

Rule 15: Transforming Element of XML mapping file to Table Class of KM3 model

```
rule HF2KM3TableClass {
   from
      s : XML!Element (
          s.oclIsTypeOf(XML!Element) and
          s.name = 'class'
       )
   to
      t : KM3!Class (
       name <- s.attribute.table
   )
}
```

In this model transformation, the source pattern is the Element of XML file, whose name is class. The target pattern is the Class of KM3. The XML Element will become the KM3 Class by copying the table of the attribute of XML Element.

Rule 16 Transforming Element of XML mapping file to Source Code Attribute of KM3 model

```
rule HF2KM3SourceCodeAttribute {
   from
      s : XML!Element ( s.name = 'property' )
   to
      t : KM3!Attribute (
          location <- '',
          name <- s.name,
          package <- OclUndefined,
          lower <- 1,
          upper <- 1,
          isOrdered <- false,
          isUnique <- false,
          owner <- s.getClass(),
          subsetOf <- Set{},
          derivedFrom <- Set{}
      )
}
```

In this model transformation, the source pattern is the Element of XML file whose name is property. The target pattern is the Attribute of KM3. The location will be ‘ ’; the name of XML Element will become the name of KM3 Attribute; the lower will be 1; the upper will be 1.

Rule 17: Transforming Element of XML mapping file to Table Attribute of KM3 model

```
rule HF2KM3TableAttribute {
    from
        s : XML!Element ( s.name = 'property' )
    to
        t : KM3!Attribute (
           location <- '',
           name <- s.column,
           package <- OclUndefined,
           lower <- 1,
           upper <- 1,
           isOrdered <- false,
           isUnique <- false,
           type <- d,
           owner <- s.getClass().getValuebyName('table'),
           subsetOf <- Set{},
           derivedFrom <- Set{}
        )
}
```

In this model transformation, the source pattern is the Element of XML file whose name is property. The target pattern is the Attribute of KM3. The location will be ‘ ’; the name of XML Element will become the name of KM3 Attribute; the lower will be 1; the upper will be 1.

Rule 18: Transforming XML mappings to reference between source code class and table class of KM3 model.

```
rule HF2KM3STReference {
    from
         s : XML!Attribute (
           s.getElementName = 'class'
                   )
    to
        t1 : KM3!Reference (
           location <- '',
           name <- s.getValuebyName('name'),
           package <- OclUndefined,
           lower <- 0,
           upper <- 0-1,
           isOrdered <- false,
           isUnique <- false,
           isContainer <- false,
           opposite <- t2
        ),
        -- Reference owned by the referred Table
        t2 : KM3!Reference (
           location <- '',
           name <- s.getValuebyName('table'),
           package <- OclUndefined,
           lower <- 0,
           upper <- 0-1,
           isOrdered <- false,
           isUnique <- false,
           isContainer <- false,
           opposite <- t1
        )
}
```

In this model transformation, the source pattern is the Attribute of XML file. The target pattern is the Reference of KM3 between source code class and table class. The location will be ‘ ’; the lower will be 1; the upper will be 1; the opposite of t1 will be t2; the opposite of t2 will be t1.

Rule 19: Transforming XML mappings to reference between attribute of KM3 model.

```
rule HF2KM3AReference {
    from
        s : XML!Attribute (
            s.getElementName = 'property'
                    )
    to
        t1 : KM3!Reference (
            location <- '',
            name <- s.getValuebyName('name'),
            package <- OclUndefined,
            lower <- 0,
            upper <- 0-1,
            isOrdered <- false,
            isUnique <- false,
            isContainer <- false,
            opposite <- t2
        ),
        -- Reference owned by the referred Table
        t2 : KM3!Reference (
            location <- '',
            name <- s.getValuebyName('column'),
            package <- OclUndefined,
            lower <- 0,
            upper <- 0-1,
            isOrdered <- false,
            isUnique <- false,
            isContainer <- false,
            opposite <- t1
        )
}
```

In this model transformation, the source pattern is the Attribute of the Element of XML file whose element name is property. The target pattern is the Reference of KM3 between attributes. The location will be ' '; the lower will be 1; the upper will be 1; the opposite of t1 will be t2; the opposite of t2 will be t1.

4.1.4.2 Software Framework KM3 Model Capture and Generation – An Example

Here is an example for the Hibernate ORM Framework XML mapping file. For lack of space, some elements have been removed and replaced with "...".

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping default-lazy="false"
  package="org.plazmaforge.bsolution.employee.common.beans">
    <class name="Employee" table="EMPLOYEE">
        <id name="id" column="ID" type="java.lang.Integer">
            <generator class="sequence">
                <param name="sequence">businessable_sequence</param>
            </generator>
        </id>
        <property name="code" column="CODE" type="java.lang.String" />
        <property name="tax"
         type="org.hibernate.usertype.CustomBooleanType">
             <column name="IS_TAX"  sql-type="CHAR(1)"/>
        </property>
        <property name="startDate" column="START_DATE"
        type="java.util.Date" />
        <property name="endDate" column="END_DATE"
        type="java.util.Date" />
        ......
        <property name="salary" column="SALARY"
        type="java.lang.Double" />
        <property name="tariff" column="TARIFF"
        type="java.lang.Double" />
        ......
        <property name="unbrokenSeniorityYear"
        column="UNBROKEN_SENIORITY_YEAR" type="java.lang.Integer" />
        <property name="unbrokenSeniorityMonth"
        column="UNBROKEN_SENIORITY_MONTH" type="java.lang.Integer" />
        ......
        type="java.lang.Double" />
        ... ...
         <many-to-one name="employeeCategory"
         column="EMPLOYEE_CATEGORY_ID" class="EmployeeCategory" />
         <many-to-one name="employeeRank" column="EMPLOYEE_RANK_ID"
         class="EmployeeRank" />

        <!-- <one-to-one name="employeeBusinessableElement"
         class="EmployeeBusinessableElement" property-ref="employee"
         cascade="all"/> -->
        <!-- <one-to-one name="employeeContactableElement"
         class="EmployeeContactableElement" property-ref="employee"
         cascade="all"/> -->
    </class>
</hibernate-mapping>
```

Following KM3 model is obtained from the above XML Hibernate ORM Framework mapping file. For lack of space, some elements have been removed and replaced with "...".

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:km3="http://www.eclipse.org/gmt/2005/KM3">
  <km3:Metamodel>
      <contents xsi:type="km3:Class" name="Employee"
       supertypes="/0/@contents.0/@contents.12">
        <structuralFeatures xsi:type="km3:Reference" name="EMPLOYEE"
         lower="1" upper="-1" type="..." isContainer="true"/>
      </contents>
      <contents xsi:type="km3:Class" name="EmployeeCategory"
       supertypes="...">
        <structuralFeatures xsi:type="km3:Reference"
         name="EMPLOYEE_CATEGORY" lower="1" upper="-1" type="..."
         isContainer="true"/>
      </contents>
      <contents xsi:type="km3:Class" name="EmployeeDischarge"
       supertypes="...">
        <structuralFeatures xsi:type="km3:Reference"
         name="EMPLOYEE_DISCHARGE" lower="1" upper="-1" type="..."
         isContainer="true"/>
      </contents>
      <contents xsi:type="km3:Class" name="EmployeeHeader"
       supertypes="...">
        <structuralFeatures xsi:type="km3:Reference" name="EMPLOYEE"
         lower="1" upper="-1" type="..." isContainer="true"/>
      </contents>
      <contents xsi:type="km3:Class" name="EmployeeRank"
       supertypes="...">
        <structuralFeatures xsi:type="km3:Reference"
         name="EMPLOYEE_RANK" lower="1" upper="-1" type="..."
         isContainer="true"/>
      </contents>
      <contents xsi:type="km3:Class" name="EmployeeReception"
       supertypes="...">
        <structuralFeatures xsi:type="km3:Reference"
         name="EMPLOYEE_RECEPTION" lower="1" upper="-1" type="..."
         isContainer="true"/>
      </contents>
  </km3:Metamodel>
</xmi:XMI>
```

4.1.5 Software System OWL Knowledge Model Generation

4.1.5.1 Software System OWL Knowledge Model Generation – ATL Transformations

The KM3 model is acting as a medium in this bottom-up ontology generation approach. Once the KM3 models have been obtained from all the different aspects of the software system, they will be transformed to a more appropriate knowledge representation model, OWL knowledge models. In this section, selected key ATL model transformation rules that transform the KM3 model to the OWL knowledge model will be presented below.

Rule 20: Transforming PrimitiveType of KM3 model to RDFSDataType of OWL knowledge model

```
rule KM3PrimitiveType2OWLRDFSDataType {
    from
        sd : KM3!DataType
    to
        td : OWL!RDFSDataType (
        name <- sd.name,
        uriRef <- tu
        ),
        tu : OWL!URIReference (
            tu <- turi
        ),
        turi : OWL!UniformResourceIdentifier (
               name <- thisModule.primitiveTypeMap.get(sd.name) )
}
```

In this model transformation, the source pattern is the DataType of KM3. The target pattern is the RDFSDataType of OWL. The name of KM3 Datatype will become the name of the OWL RDFSDataType; the uriRef of OWL RDFSDataType will be obtained from KM3 DataType via the UniformResourceIdentifier of OWL.

Rule 21: Transforming Class of KM3 model to Class of OWL knowledge model

```
rule KM3Class2OWLClass {
   from
      sc : KM3!Class
   to
      tc : OWL!OWLClass (
      name <- sc.name,
      uriRef <- tu,
      label <- tlabel,
      subClassOf <- sc.supertypes
      ),
      tlabel : OWL!PlainLiteral ( lexicalForm <- sc.name ),
      tu : OWL!URIReference ( tu <- turi ),
      turi : OWL!UniformResourceIdentifier ( name <- sc.name )
}
```

In this model transformation, the source pattern is the Class of KM3. The target pattern is the OWLClass of OWL. The name of KM3 Class will become the name of OWL Class; the uriRef of OWLClass will be obtained from KM3 Class via the UniformResourceIdentifier of OWL; the supertypes of KM3 Class will become the super class of OWLClass.

Rule 22: Transforming Attribute of KM3 model to DataTypeProperty of OWL knowledge model

```
rule KM3Att2OWLDataTypeProperty {
   from
      s : KM3!Attribute (
         f.type.oclIsTypeOf( KM3!DataType )
      )
   to
      t : OWL!OWLDatatypeProperty (
         name <- s.name,
         domain <- s.owner,
         range <- s.type,
         uriRef <- tu
      ),
      tu : OWL!URIReference ( fragmentIdentifier <- tl, tu <- turi ),
      tl : OWL!LocalName ( name <- s.owner.name + '.' + s.name ),
      turi : OWL!UniformResourceIdentifier (
            name <- s.owner.name + '.' + s.name )
}
```

In this model transformation, the source pattern is the Attribute of KM3. The target pattern is the OWLDatatypePropery of OWL. The name of the KM3 Attribute will become the name of OWLDatatypeProperty; the owner of the KM3 Attribute will become the domain of OWLDatatypeProperty; the type of the KM3 Attribute will become the range of OWLDatatypeProperty; the uriRef of OWLDatatypeProperty will be obtained via OWL UniformResourceIdentifier.

Rule 23: Transforming Reference of KM3 model to ObjectProperty of OWL model

```
rule KM3Ref2OWLObjectProperty {
    from
        s : KM3!Reference

    to
        t : OWL!OWLObjectProperty (
            name <- s.name,
            domain <- s.owner,
            range <- s.type,
            uriRef <- tu
            OWLInverseOf <- s.opposite,
            subPropertyOf <- s.subsetOf
        ),
        tu : OWL!URIReference ( fragmentIdentifier <- tl, tu <- turi ),
        tl : OWL!LocalName ( name <- s.owner.name + '.' + s.name ),
        turi : OWL!UniformResourceIdentifier (
               name <- s.owner.name + '.' + s.name )
}
```

In this model transformation, the source pattern is the Reference of KM3. The target pattern is the OWLObjectProperty of OWL. The name of KM3 Reference will become the name of OWLObjectProperty; the owner of KM3 Reference will become the domain of OWLObjectProperty; the type of KM3 Reference will become the range of OWLObjectProperty; the opposite of KM3 Reference will become OWLInverseOf; the subset of KM3 Reference will become the subProperty of OWLObjectProerty.

Rule 24: Transforming Emueration of KM3 model to EnumeratedClass of OWL knowledge model

```
rule KM3Enum2OWLEnumeratedClass {
    from
        se : KM3!Enumeration
    to
        te : OWL!EnumeratedClass (
            OWLOneOf <- se.literals,
            uriRef <- tu,
            label <- tlabel
        ),
        label : OWL!PlainLiteral ( lexicalForm <- se.name ),
        tu : OWL!URIReference (  fragmentIdentifier <- tl, tu <- turi ),
        tl : OWL!LocalName ( name <- se.name ),
        turi : OWL!UniformResourceIdentifier ( name <- se.name )
}
```

In this model transformation, the source pattern is the Enumeration of KM3. The target pattern is the EnumeratedClass. The uriRef of OWL EnumeratedClass will be obtained via OWL UniformResourceIdentifier.

4.1.5.2 Software System OWL Knowledge Model Generation – An Example

The ATL transformation discussed in the previous section has been applied to the KM3 model obtained in Section 4.1.2.3. As a result, following OWL knowledge model is generated. For lack of space, some elements have been removed and replaced with "...".

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns="OWL" xmlns:_1="RDFS">
  <OWLGraph statement="..." ontology="/4">
    <uriRef uri="/2" fragmentIdentifier="..." namespace="/3"/>
  </OWLGraph>
  <_1:Document xmlBase="/3">
  <_1:LocalName name="Employee" uriRef="..."/>
  <_1:UniformResourceIdentifier name="Employee" uriRef="..."/>
  <_1:RDFSDataType>
    <uriRef uri="/29"/>
  </_1:RDFSDataType>
  <_1:UniformResourceIdentifier name="..." uriRef=".."/>
  <_1:RDFSDataType>
    <uriRef uri="..."/>
  </_1:RDFSDataType>
  ... ...
  <_1:UniformResourceIdentifier name="..." uriRef="..."/>
  <OWLDatatypeProperty domain="/25" range="/34"
   propertyRestriction="/139">
    <uriRef uri="/38" fragmentIdentifier="/37"/>
  </OWLDatatypeProperty>
  <_1:LocalName name="1896" uriRef="/136/@uriRef.0"/>
  <_1:UniformResourceIdentifier name="1896" uriRef="/136/@uriRef.0"/>
  <CardinalityRestriction superClass="/25" OWLOnProperty="/36"
   OWLCardinality="/140"/>
  <_1:TypedLiteral lexicalForm="1" datatypeURI="/30/@uriRef.0"
   cardinalityRestriction="/159"/>
  <FunctionalProperty isDefinedBy="/66"/>
  ... ...
  <MinCardinalityRestriction superClass="/19" OWLOnProperty="/72"
   OWLMinCardinality="/168"/>
  <OWLStatement graph="/0" RDFpredicate="/72" RDFobject="/93"
   RDFsubject="/87"/>
</xmi:XMI>
```

4.1.6 Software System Ontology Generation – the Final owl File

4.1.6.1 Software System Ontology Generation – the Final owl File Transformation

The last step of this bottom-up ontology generation approach is to transform the OWL knowledge model to a formal ontology representation form, i.e., an XML file with OWL/XML syntax elements. The selected key transformation rules are given below.

Rule 25: Transforming Class of OWL model to Element of owl file

```
rule OWLClass2XMLCElement{
    from
        s : OWL!OWLClass (
            s.oclIsTypeOf(OWL!OWLClass)
        )
    to
        t : XML!Element (
            name <- 'owl:Class',
            children <- Sequence{tID,tlabel},
            parent <- OWL!OWLGraph.allInstances()
                      ->any( e | e.oclIsTypeOf(OWL!OWLGraph))
        ),
        tID : XML!Attribute ( name <- 'rdf:ID', value <- s.getURI() ),
        tlabel : XML!Element ( name <- 'rdfs:label',
                               children <- tlabeltext ),
        tlabeltext : XML!Text ( name <- '#text', value <- s.getLabel() )

        do {
             for (s1 in s.subClassOf ) {
                if (s1.oclIsTypeOf(OWL!OWLClass))
                 thisModule.makeSubClass(s,s1);
                if (s1.oclIsTypeOf(OWL!UnionClass))
                 thisModule.makeSubClass(s,s1);
                if (s1.oclIsTypeOf(OWL!CardinalityRestriction))
                 thisModule.makeCardinalityRestrictionSubClass(s,s1);
                if (s1.oclIsTypeOf(OWL!MaxCardinalityRestriction))
                 thisModule.makeMaxCardinalityRestrictionSubClass(s,s1);
                if (s1.oclIsTypeOf(OWL!MinCardinalityRestriction))
                 thisModule.makeMinCardinalityRestrictionSubClass(s,s1);
             }
        }
}
```

In this model transformation, the source pattern is OWLClass of OWL. The target pattern is the Element of XML. This transformation is storing OWL Class in owl file with XML syntax.

Rule 26: Transforming DataProperty of OWL model to Element of owl file

```
rule OWLDatatypeProperty2XMLDPElement {
    from
        sd : OWL!OWLDatatypeProperty
    to
        te : XML!Element (
            name <- 'owl:DatatypeProperty',
            children <- Sequence{tID,tdomain,trange},
            parent <- OWL!OWLGraph.allInstances()
                        ->any( e | e.oclIsTypeOf(OWL!OWLGraph))
        ),
        tID : XML!Attribute (
            name <- 'rdf:ID',
            value <- sd.getURI()
        ),
        tdomain : XML!Element (
            name <- 'rdfs:domain',
            children <- tdomainattr
        ),
        tdomainattr : XML!Attribute (
            name <- 'rdf:resource',
            value <- '#' + sd.domain
                     ->any( c | c.oclIsKindOf(OWL!OWLClass)).getURI()
        ),
        trange : XML!Element (
            name <- 'rdfs:range',
            children <- trangeattr
        ),
        trangeattr : XML!Attribute (
            name <- 'rdf:resource',
            value <-  sd.range->any(c |c.oclIsKindOf(
                       OWL!RDFSDataType)).getURI()
        )
}
```

In this model transformation, the source pattern is OWLDatatypeProperty of OWL. The target pattern is the Element of XML. This transformation is storing OWL DatatypeProperty in owl file with XML syntax.

Rule 27: Transforming ObjectProperty of OWL model to Element of owl file

```
rule OWLObjectProperty2XMLOPElement {
    from
        so : OWL!OWLObjectProperty (
            to.oclIsTypeOf(OWL!OWLObjectProperty)
        )
    to
        te : XML!Element (
            name <- 'owl:ObjectProperty',
            children <- Sequence{tID, tdomain, trange},
            parent <- OWL!OWLGraph.allInstances()
                       ->any( e | e.oclIsTypeOf(OWL!OWLGraph))
        ),
        tID : XML!Attribute (
           name <- 'rdf:ID',
           value <- so.getURI()
        ),
        tdomain : XML!Element (
           name <- 'rdfs:domain',
           children <- tdomainattr
        ),
        tdomainattr : XML!Attribute (
           name <-  'rdf:resource',
           value <- '#' + so.domain->any( c | c.oclIsKindOf(
                                           OWL!OWLClass)).getURI()
        ),
        trange : XML!Element (
           name <- 'rdfs:range',
           children <- trangeattr
        ),
        trangeattr : XML!Attribute (
           name <- 'rdf:resource',
           value <- '#' + so.range->any( c | c.oclIsKindOf(
                                           OWL!OWLClass)).getURI()
        )

        do {
           if (not so.OWLInverseOf.oclIsUndefined())
              thisModule.addInverse(so);

           for (s1 in so.subPropertyOf ) {
              if (s1.oclIsKindOf(OWL!OWLObjectProperty))
                 thisModule.makeSubProperty(so,s1);
           }
        }
}
```

In this model transformation, the source pattern is the OWLObjectProperty of OWL, the target pattern is the Element of XML. This transformation is storing OWL ObjectProperty in owl file with XML syntax.

Rule 28: Transforming FunctionalProperty of OWL model to Element of owl file

```
rule OWLFunctionalProperty2XMLFPElement {
   from
      sf : OWL!FunctionalProperty (
         sf.oclIsTypeOf( OWL!FunctionalProperty )
      )
   to
      te : XML!Element (
         name <- 'owl:FunctionalProperty',
         children <- ta,
         parent <- OWL!OWLGraph.allInstances()
                   ->any( e | e.oclIsTypeOf(OWL!OWLGraph))
      ),
        ta : XML!Attribute (
            name <- 'rdf:about',
            value <- '#' + sf.isDefinedBy->asSequence()
                   ->any( e | e.oclIsKindOf(OWL!Property ) ).getURI()
        )
}
```

In this model transformation, the source pattern is FunctionalProperty of OWL. The target pattern is the Element of XML. This transformation is storing OWL FunctionalProperty in owl file with XML syntax.

Rule 29: Transforming EnumeratedClass of OWL model to Element of owl file

```
rule OWLEnumeratedClass2XMLEElement {
    from
        sec : OWL!EnumeratedClass (
            sec.oclIsTypeOf( OWL!EnumeratedClass )
        )
    to
        te : XML!Element (
            name <- 'owl:Class',
            children <- Sequence{tID,tlabel,toneOf},
            parent <- OWL!OWLGraph.allInstances()
                      ->any( e | e.oclIsTypeOf(OWL!OWLGraph))
        ),
        tID : XML!Attribute (
            name <- 'rdf:ID',
            value <- sec.getURI()
        ),
        tlabel : XML!Element (
            name <- 'rdfs:label',
            children <- tlabeltext
        ),
        tlabeltext : XML!Text (
            name <- '#text',
            value <- sec.getLabel()
        ),
        toneOf : XML!Element (
            name <- 'owl:oneOf',
            children <- Sequence{ toneOfAtt, ec.OWLOneOf
                ->collect( e | thisModule.IndividualLiteral2Element( e ) )
            }
        ),
        toneOfAtt : XML!Attribute (
            name <- 'rdf:parseType',
            value <- 'Collection'
        )

}
```

In this model transformation, the source pattern is the EnumeratedClass of OWL. The target pattern is the Element of XML. This transformation is storing OWL

EnumeratedClass in an .owl file with XML syntax.

With the above ATL model transformations, the OWL models which are encoded in XMI format will be converted to .owl files with OWL/XML syntax elements. As a result, the bottom-up ontology generation approach has produced the final product – software ontology, which can be manipulated in the next stage of this study – softare ontology deployment.

4.1.6.2 Software System Ontology Generation – An Example

The final owl file retrieved from the OWL knowledge model described in Section 4.1.5.2 is given as following. The selected key transformation rules are given below.

```
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<rdf:RDF
    xmlns="http://www.owl-ontologies.com/Ontology1274641707.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    ... ...
    xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://www.owl-ontologies.com/Ontology1274641707.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="BusinessableElement"/>
  <owl:Class rdf:ID="ContactableElement"/>
  <owl:Class rdf:ID="Employee"/>
  <owl:Class rdf:ID="PayrollForm"/>
  <owl:ObjectProperty rdf:ID="contactableElement">
    <rdfs:domain>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#ContactableElement"/>
          <owl:Class rdf:about="#Employee"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:domain>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="payrollForm">
    <rdfs:domain>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#PayrollForm"/>
          <owl:Class rdf:about="#Employee"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:domain>
  </owl:ObjectProperty>
  ... ...
<rdf:RDF
    xmlns="http://www.owl-ontologies.com/Ontology1274641707.owl#"
    ... ...
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="BusinessableElement"/>
  <owl:Class rdf:ID="ContactableElement"/>
  <owl:Class rdf:ID="Employee"/>
  <owl:Class rdf:ID="PayrollForm"/>
  <owl:ObjectProperty rdf:ID="contactableElement">
    <rdfs:domain>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#ContactableElement"/>
          <owl:Class rdf:about="#Employee"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:domain>
  </owl:ObjectProperty>
  ... ...
  <owl:DatatypeProperty rdf:ID="employeeAccount">
    <rdfs:domain rdf:resource="#Employee"/>
    <rdfs:range rdf:resource="..."/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="tax">
    <rdfs:range rdf:resource="..."/>
    <rdfs:domain rdf:resource="#Employee"/>
  </owl:DatatypeProperty>
  ... ...
  <owl:DatatypeProperty rdf:ID="CLASS_ID">
    <rdfs:range rdf:resource="..."/>
    <rdfs:domain rdf:resource="#Employee"/>
  </owl:DatatypeProperty>
</rdf:RDF>
```

4.2 Top-Down Software System Ontology Development

4.2.1 Top-Down Operating System Ontology Development – An Example

When a bottom-up approach cannot help to build ontology, a top-down approach will be considered. Compared with the bottom-up approach, a top-down ontology development is not straightforward but more complicated. The starting point for a top-down approach is thinking of and deciding about the core principles of ontology development. For example, which features should be represented when building an operating system ontology is the very first question to be answered in the process of operating system ontology generation. An operating system is always very complicated and it could contain a lot of knowledge aspects. Hence, producing a set of core design principles for ontology development will be essential in a top-down approach and it also requires a comprehensive analysis of the problem domain, i.e. analysing the related knowledge in the problem domain and the potential usage of ontology. For instance, before building an operating system ontology, many different knowledge aspects need to be considered, such as operating system components, operating system architecture, operating system functions and features, and operating system principles, etc. In addition, there are many potential uses for operating system ontology including software education, software engineering and software reengineering, etc. On the one hand, operating system ontology could be easily used for education purposes in order to demonstrate the structures, basic concepts and their relationships in the operating system. On the other, operating system ontology can also be used as a knowledge base which will provide the means to semi-automate some of the processes in both reverse and forward engineering projects. As a result, such comprehensive and heuristic ontology capture processes should include a set of principles and criteria that could be a guideline for the process.

In this research, operating system ontology is mainly developed for software reengineering purposes. A brainstorm has been performed in the first instance to decide which aspects of the operating system should be represented by ontology in relation to software reengineering activities. The following aspects have been chosen to represent the operating system at the brainstorm stage.

Operating System Functions/Services includes process management, memory management, file system management, I/O system management, network management, security management and graphical user interface management, etc. The related operating system concepts could be organised based on this classification. For instance, process creation, process deletion, process suspension, process resumption, process synchronisation and process communication are all related concepts in the process management category.

Operating System Architecture/Components contains concepts such as kernel, system call interface, pipes, filters, utilities, device drivers, executable programs and configurable environment, etc. The related operating system concepts in this category are mainly about the system structure and components.

Operating System Principles/Theories cover the basic operating systems concepts, with an emphasis on internals, design and performance issues. e.g. sequential processes, concurrent processes, processor management, store management, scheduling algorithms and resource protection and are the main elements which compose the operating system theory category.

Many overlaps may be seen between each aspect when comparing these three. To narrow down the research problem in this study, software porting and platform specific software migration has been chosen as the potential reengineering scenario in which operating system ontology will be used. In order to meet the requirements of this reengineering scenario, system call interfaces are selected as the main subjects that the operating system ontology will represent. The next section will describe ontology development principles for the operating system in detail.

4.2.2 Operating System Ontology Development Rules

The key to top-down operating system ontology development is to provide a systematic guideline. A potential use for the proposed operating system ontology is to allow software maintainers to get the required information quickly and precisely. For instance, in a software migration and porting scenario, if developers are looking for information about existing POSIX APIs that are defined in both operating systems providing a thread creating service, they can query operating system ontology by retrieving the related system call interface concepts which are defined in both OSs. If the query result suggests that both systems have POSIX API implemented to create a thread, then that part of the application can be migrated by replacing API's names directly. If the query result shows that neither system has such POSIX APIs, then the application code will need to be re-implemented during the migration. Likewise, if one platform's features are not supported by another, corresponding development is required. To fulfil the requirement of those scenarios, eight rules have been defined focusing on different development aspects for operating system ontology.

Rule 1: Instance Biased Definition.

An atomic concept should have more than one instance, while one instance concept should be defined as an instance of its superclass.

For example, when concept API_standard is introduced, it can be divided into POSIX_standard and non-POSIX_standard. However, POSIX_standard should not be considered as a concept since it only has one instance, i.e., POSIX. Hence, other non-POSIX standards such as WIN32 will be used as instances of API_standard, as well as POSIX. Moreover, all the particular OSs should be in the same instance level of OS ontology.

Rule 2: Application Specified Design.

A concept should be defined based on specific requirements of the application domain, rather than based on particular application domain terminology.

Which means in this case, the granularity of the concept depends on the potential usage of the ontology in the problem domain, but not the lexical structure. For instance, concept API is not divided into any other smaller concepts, since maintainers consider API as a whole concept when performing program understanding. Hence, concept API does not need to be divided into small concepts.

Rule 3: API Based Classification.

Concepts in the operating system ontology should be divided into three categories, i.e., code level concepts, code behaviour level concepts and code attribute level concepts. Each concept should belong to one of these three categories.

This means, API should play a central role when classifying the concepts. For example, concept API is a code level concept, which is used to describe code; concept System_service is a code behaviour level concept, which describes the behaviour of the particular code; and concept Data_type belongs to code attribute level concept, which is attribute of particular code.

Rule 4: Behaviour Centred Organisation.

Concepts should be organised according to their behaviours or functions.

That is to say, a defined concept is a code level concept while a primitive concept is a code behaviour level or code attribute level concept. Primitive concepts compose defined concept, and they will also facilitate the definition of defined concept. For instance, the concept API is a code level concept; the concept System_service is a code behaviour level concept; the concept Data_type is a code attribute level concept; subclass of the concept API will be a defined concept and defined by the concepts System_service, Data_type. All the concepts are organised according to the behaviour level concept.

Rule 5: Cardinality Restricted Relations.

Cardinality should be introduced to all the relations between concepts in proposed operating system ontology.

This means, having cardinality for relations will restrict the instances which are related to these relations. For example, each API should have one and only one return type; each API should provide at least one system service, etc. Cardinality will facilitate the management of concepts instances and their relations, ensuring the consistency of the ontology.

Rule 6: Understanding Aimed Naming.

Naming of concept and relation should follow the regulation which is defined to facilitate program comprehension.

That is to say, a naming regulation that is aiming to facilitate program understanding is defined. A regulation for concept naming should be concrete. For instance, it is helpful if the name of the subclass consists of a part of the name of the superclass; the name of instance should contain the acronym of particular OS where the instance exists, e.g., concepts Thread_service, Thread_api and instances Thread_service_create, rtl_pthread_create.

Rule 7: Aspect Oriented Restructuring.

Ontology design process should support restructuring.

Aspect Oriented Programming [20] is widely used to restructure the application and its ideas can also be used in ontology design. A concept can be split into sub-concepts based on different concerns (aspects). For example, Concept API and System_service used to be designed as one concept to describe particular API, e.g., concept Thread_create_api and Mutex_init_api. However, during the implementation process, this design is found to be improper when more API concepts are added. As a consequence, former design should be restructured into two parts: API and System_service, which can be combined together to describe a particular API by defined concept.

Rule 8: Multi-Layered Structure.

Ontology design should support the extensibility with multi-layered structure.

This means, the whole design should be extended easily, with the condition that introducing new concept can only impact relations. For example, all the subclasses of OSThing should be disjointed, which provides the possibility of extending the knowledge base.

4.2.3 Operating System Ontology Development – An Example

According to the operating system ontology development principles proposed in the previous section, operating system ontologies have been built. Figure 4-1 illustrates an overview of structure of operating system ontology. In the operating system ontology, OSThing is the subclass of owl:Thing and is the superclass of all the other concepts in the operating system ontology. The subclass concepts of OSThing is organised in six categories, namely, Operating System, System Call Interface, System Service, System Architecture, Data Type, Driver. Operating System, System Call Interface, System Service and Data Type are the main categories that are involved in this study, which are developed in order to meet the requirements of software porting and migration. System

Architecture and Driver categories are not implemented at the moment and are left for the future extension for different software reengineering scenarios. The System Call Interface category includes API, Parameter and API standard. API can be divided into six main categories based on the functionalities, e.g., thread_api, mutex_api, semaphore_api, message_queue_api, etc. API standard has no subclass concepts, only contains three instances, POSIX, NONPOSIX and WIN32. System Service is strongly related to System Call Interface category, as System Service represents the function of System Call Interface. In the operating system ontology, System_service is used to define different API by declaring what service API will provide. Parameter and Data Type are two auxiliary concepts which will help to represent and query API more accurately.

Figure 4-2 gives an example of concrete concepts and their instances in operating system ontology. Different instances and their relations of concepts are shown in this graph. The rectangular box with many small boxes inside indicates instance. The singular box represents concept. The arrow line with different labels exhibits binary relation, also known as object property in ontology. The arrow line with 'io' label depicts 'instance of' relation. The arrow line with 'isa' label shows 'is a' relation. From this specific view, the powerful representation ability of operating system ontology is highlighted. Following the eight operating system ontology design principles, not only concepts, instances and simple relations in operating system domain are illustrated by such ontology, but also the complex knowledge representation can be performed as well, e.g., instance rtl_pthread_create suggests that system call interface pthread_create is provided by RTLinux operating system, and it is the POSIX API which provides thread creating service with the integer return type and needs the pointer of thread as core parameter. Meanwhile, its counterpart in ThreadX operating system can be spotted, which indicates tx_thread_create is NONPOSIX API which also provides thread creating service with unsigned integer return type and pointer of thread as core parameter.

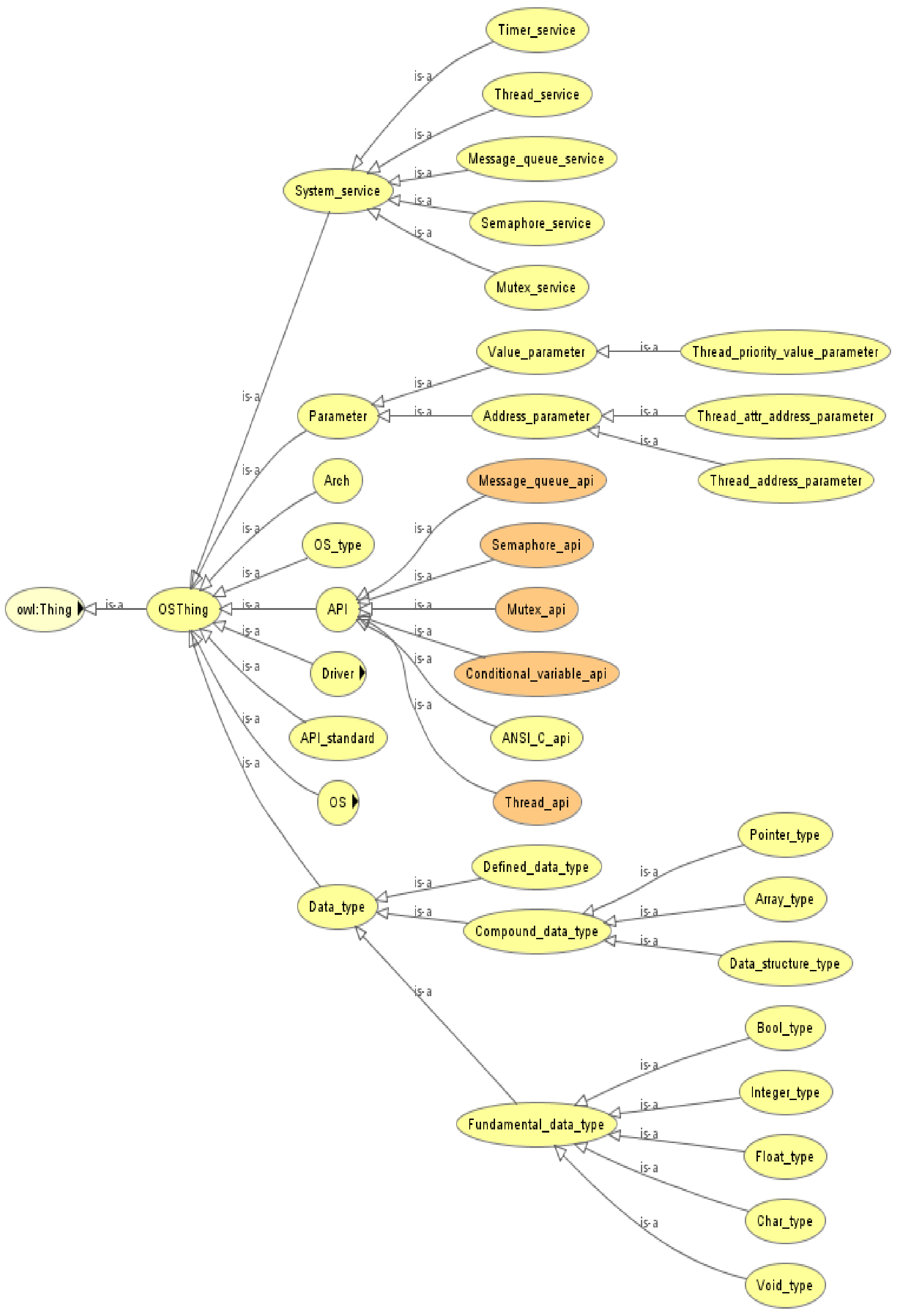


Figure 4-1 Structure of Operating System Ontology

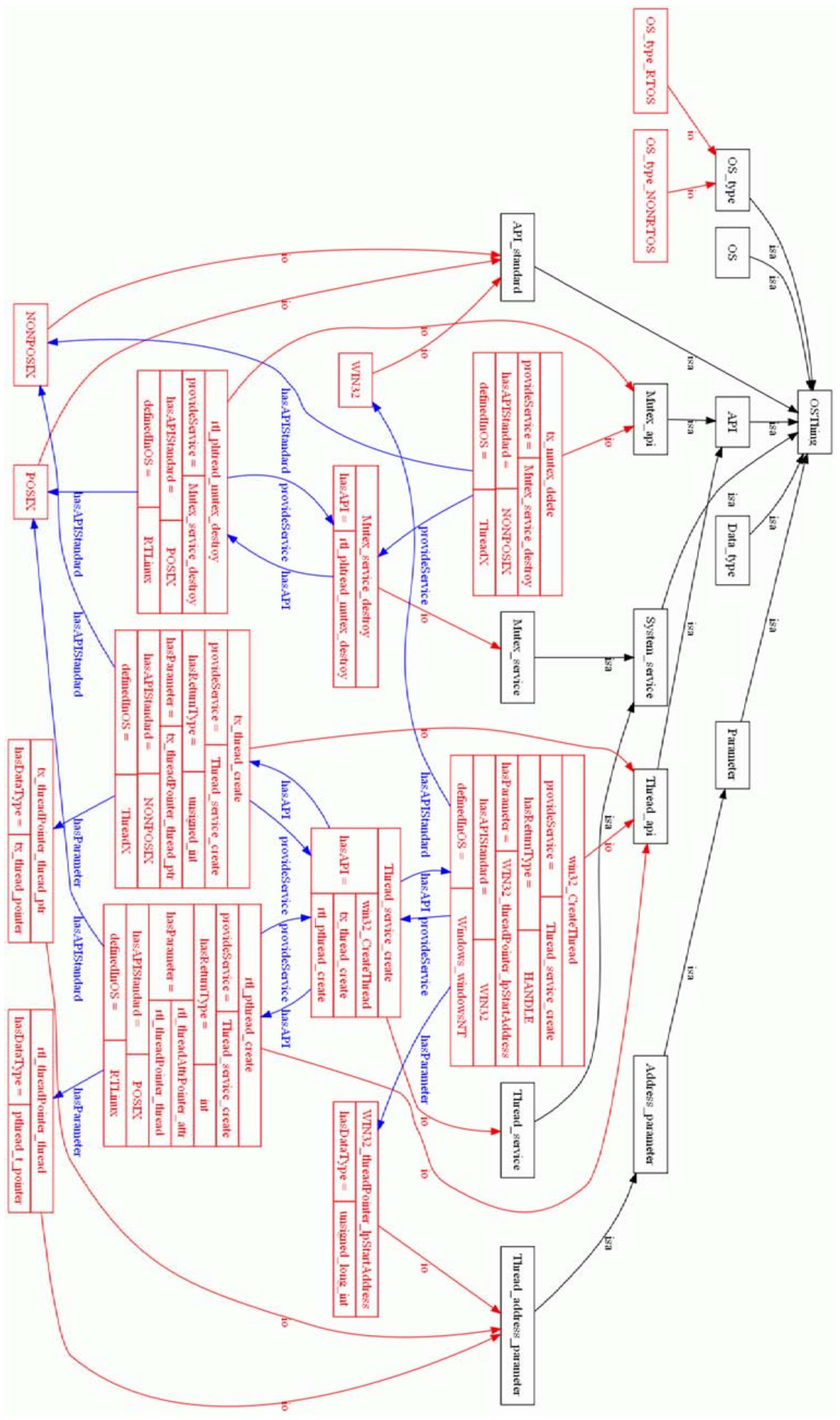


Figure 4-2 an Example of Operating System Ontology

The information depicted in the graph can also be represented formally by Description Logic.

$$
\begin{aligned}
\text{rtl\_pthread\_create} \equiv\ & \forall\ \text{definedInOS.RTLinux} \\
& \cap \forall\ \text{hasAPIStandard.POSIX} \\
& \cap \forall\ \text{hasReturnType.int} \\
& \cap \forall\ \text{provideService.thread\_service\_create} \\
& \cap \forall\ \text{hasParameter. rtl\_threadAttrPointer\_attr} \\
& \cap \forall\ \text{hasParameter. rtl\_threadPointer\_thread}
\end{aligned}
$$

$$
\begin{aligned}
\text{tx\_thread\_delete} \equiv\ & \forall\ \text{definedInOS.ThreadX} \\
& \cap \forall\ \text{hasAPIStandard.NONPOSIX} \\
& \cap \forall\ \text{hasReturnType.int} \\
& \cap \forall\ \text{provideService.thread\_service\_delete} \\
& \cap \forall\ \text{hasParameter. tx\_threadPointer\_thread}
\end{aligned}
$$

$$
\begin{aligned}
\text{win32\_TerminateThread} \equiv\ & \forall\ \text{definedInOS.Windows} \\
& \cap \forall\ \text{hasAPIStandard.WIN32} \\
& \cap \forall\ \text{hasReturnType.int} \\
& \cap \forall\ \text{provideService.thread\_service\_create} \\
& \cap \forall\ \text{hasParameter.win\_threadPointer\_thread}
\end{aligned}
$$

$$
\begin{aligned}
\text{rtl\_pthread\_mutex\_init} \equiv\ & \forall\ \text{definedInOS.RTLinux} \\
& \cap \forall\ \text{hasAPIStandard.POSIX} \\
& \cap \forall\ \text{hasReturnType.int} \\
& \cap \forall\ \text{provideService.ThreadServiceCreate} \\
& \cap \forall\ \text{hasParameter. rtl\_threadAttrPointer\_attr} \\
& \cap \forall\ \text{hasParameter. rtl\_threadPointer\_thread}
\end{aligned}
$$

$$
\begin{aligned}
\text{tx\_mutex\_delete} \equiv\ & \forall\ \text{definedInOS.ThreadX} \\
& \cap \forall\ \text{hasAPIStandard.NONPOSIX} \\
& \cap \forall\ \text{hasReturnType.int} \\
& \cap \forall\ \text{provideService.thread\_service\_delete} \\
& \cap \forall\ \text{hasParameter. tx\_threadPointer\_thread}
\end{aligned}
$$

Furthermore, knowledge acquisition can be conducted based on such ontology. For example, the developers can easily get the information of the APIs which provide a similar system service. The following example demonstrates some queries to operating system ontology, which reveals the fact that the Windows NT system provides system service to create a thread, the API for this service is CreatThread(), it is a NON POSIX, the return type is HANDLE, the parameter for this API is a pointer for the new thread.

While ThreadX system provide the creating thread service as well, it is invoked by thread_create(), it is a NON POSIX, the return type is unsigned int, the parameter for it is also a pointer for the new thread.

$$\mathit{Thread\_api}(?x) \wedge \mathit{Thread\_service}(\mathit{Thread\_service\_create}) \wedge \mathit{Windows}(\mathit{Windows\_windowsNT}) \wedge \mathit{definedInOS}(?x, \mathit{Windows\_windowsNT}) \wedge \mathit{provideService}(?x, \mathit{Thread\_service\_create}) \wedge \mathit{API\_standard}(?y) \wedge \mathit{Data\_type}(?z) \rightarrow \mathit{hasAPI}(\mathit{Windows\_windowsNT}, ?x) \wedge \mathit{hasAPIStandard}(?x, ?y) \wedge \mathit{hasReturnType}(?x, ?z)$$

$$\mathit{Thread\_api}(?x) \wedge \mathit{Thread\_service}(\mathit{Thread\_service\_create}) \wedge \mathit{Windows}(\mathit{ThreadX}) \wedge \mathit{definedInOS}(?x, \mathit{ThreadX}) \wedge \mathit{provideService}(?x, \mathit{Thread\_service\_create}) \wedge \mathit{API\_standard}(?y) \wedge \mathit{Data\_type}(?z) \rightarrow \mathit{hasAPI}(\mathit{ThreadX}, ?x) \wedge \mathit{hasAPIStandard}(?x, ?y) \wedge \mathit{hasReturnType}(?x, ?z)$$

4.3 Summary

This chapter explores the methodologies for software system ontology generation. A bottom-up approach is relatively straightforward. In a nutshell, bottom-up ontology generation is a series of model transformation processes, in which ontology is generated by transformation from other software models. When a bottom-up approach cannot help to build ontology, a top-down approach will be considered. Compared with the bottom-up approach, a top-down ontology development is more complicated and indirect. It suggests building ontology from scratch based on predefined guidelines. Hence, it requires a blueprint for the entire software system before starting to build ontology. The starting point for a top-down approach is thinking of and deciding about the core principles of ontology development.

- Three transformation steps are defined in the bottom-up ontology generation approach, namely, transformation between the software model and the KM3 model, transformation from the KM3 model to the OWL knowledge model and transformation from the OWL knowledge model to an .owl document.

- Three components are proposed to construct a software model, namely, source code model, software framework model and software data model. The KM3 model is introduced as a medium for the transformation between the software model and the OWL knowledge model.

- Six model transformation scenarios have been defined to compose a bottom-up ontology generation approach, i.e., class diagram to the KM3 model, XML database description to the database model, database model to the KM3 model, XML Hibernate configuration to the KM3 model , the KM3 model to the OWL model and the OWL model to .owl documents.

- Twenty-nine ATL transformation rules are defined in this chapter. Rule 1 – Rule 6 support class diagram to the KM3 model transformation. Rule 7 – Rule 9 implement XML description to the database model transformation. Rule 10 – Rule 13 are applied to the database model to the KM3 model transformation. Rule 14 – Rule 19 are defined to transform XML Hibernate configuration to the KM3 model. Rule 20 – Rule 24 implement the KM3 model to the OWL model transformation. Rule 25 – Rule 29 will transform the OWL model to .owl documentation.

- Three potential operating system knowledge representation perspectives have been proposed, namely, operating system functions/services, operating system architectures/components and operating system principles/theories.

- Eight operating system ontology development rules have been defined focusing on different development aspects in order to fulfil the requirements of the proposed software reengineering scenarios, i.e., platform specific software migration and portable software development.

Chapter 5 Software System Ontology Integration via Inference in Description Logic

Objectives

- To define software system ontology integration
- To present a Description Logic based ontology mapping algorithm
- To demonstrate how to represent software ontology by Description Logic

This chapter aims at providing methodologies for integrating different software system ontologies. Ontology integration has many synonyms in the ontology engineering research area. Generally speaking, software system ontology integration reflects ontology mapping, which is a process of finding semantic relationships between entities (e.g., concepts, relations, etc.) across two different software system ontologies. However, in practice most ontology mapping processes are performed manually by domain experts at the moment. Therefore it will be a time consuming, tedious and error-prone process [22]. A few researchers have addressed the ontology mapping problem from different disciplines such as data analysis, machine learning, language engineering and knowledge engineering, etc. In order to achieve an accurate (semi-) automatic large-scale ontology mapping, one single method may be unlikely to succeed. Hence, combining different approaches would be a more effective way. The proposed DL-based ontology mapping approach is based on CTXMATCH [12], which is an algorithm for detecting semantic mappings between hierarchical classifications (HCs) via propositional logical deduction. However, a CTXMATCH algorithm can only deal with unary predicates, and it cannot handle the binary predicates such as properties or roles [12]. Hence, the proposed software system ontology mapping approach extends the CTXMATCH algorithm by exploring the expressive power and efficient reasoning of description logic.

Software system ontologies always have three different levels of knowledge. These are lexical knowledge, domain knowledge and structural knowledge. Lexical knowledge is about the semantic meanings of the terms that are used to describe a software system. To understand the lexical meaning, WordNet is employed as a lexical knowledge base in this research. Domain knowledge is about the terms that are used to describe the specific problem domain in the real world. Structural knowledge is about the structures on which all the terms are organised in a software system, such as inheritance relations and complicated binary relations, etc. In other words, the hierarchical classification of software ontology contains structural knowledge. Description Logic is employed to represent all three levels of software system knowledge in logical formulae, and therefore to transform the problem of seeking semantic relationships between terms across different ontologies into deducing the satisfiability of logical formulae that are represented by Description Logic.

5.1 Software System Ontology Mapping Algorithm

5.1.1 Definition

A software system ontology mapping detects a semantic relationship between a term (concepts, relations, etc.) of software system ontology (source ontology) and a term of a different software system ontology (target ontology). A formal definition is given below.

Definition 5.1 An ontology mapping m from a ontology O1 = <C1, R1>, called source ontology to a ontology O2 = <C2, R2>, called target ontology, is a set of 3-tuple <ci, Rel, cj> where:

- C1 and C2 are set of concepts

- R1 and R2 are sets of relations
- Rel is semantic relationship, Rel $\in \{ \subseteq, \supseteq, \equiv, \perp \}$;
- ci is arbitary concept, ci $\in$ C1;
- cj is arbitary concept, cj $\in$ C2.

c1 $\subseteq$ c2 means that c1 is less general than c2; c1 $\supseteq$ c2 means that c1 is more general than c2; c1 $\equiv$ c2 means that c1 $\subseteq$ c2 and c1 $\supseteq$ c2, i.e., c1 is equivalent to c2; c1 $\perp$ c2 means that c1 is disjoint from c2, i.e., there is no semantic relationship between c1 and c2.

5.1.2 Overview

In this section, a software system ontology mapping algorithm is presented. The basic idea of the proposed mapping algorithm is to represent terms of both source ontology and target ontology by Description Logic formulae with relevant lexical, domain and structural knowledge, and then to transform the problem of detecting semantic relationships into the problem of deducing satisfiability of Description Logic formulae. i.e., the term Cs in source ontology and the term Ct in target ontology will be encoded as Description Logic formulae by combing knowledge from all three knowledge levels. Therefore, detecting whether one term Cs in source ontology is subsumed by the other one Ct in target ontology will become a problem of testing whether the Cs $\cap \neg$Ct is unsatisfiable; detecting whether one term Cs in source ontology and the term Ct in target ontology are equivalent will be become a problem of testing both (Cs $\cap \neg$Ct) and ($\neg$Cs $\cap$ Ct) are unsatisfiable; detecting whether one term Cs in source ontology and the term Ct in target ontology are disjoint will become a problem of testing whether Cs $\cap$ Ct is unsatisifiable.

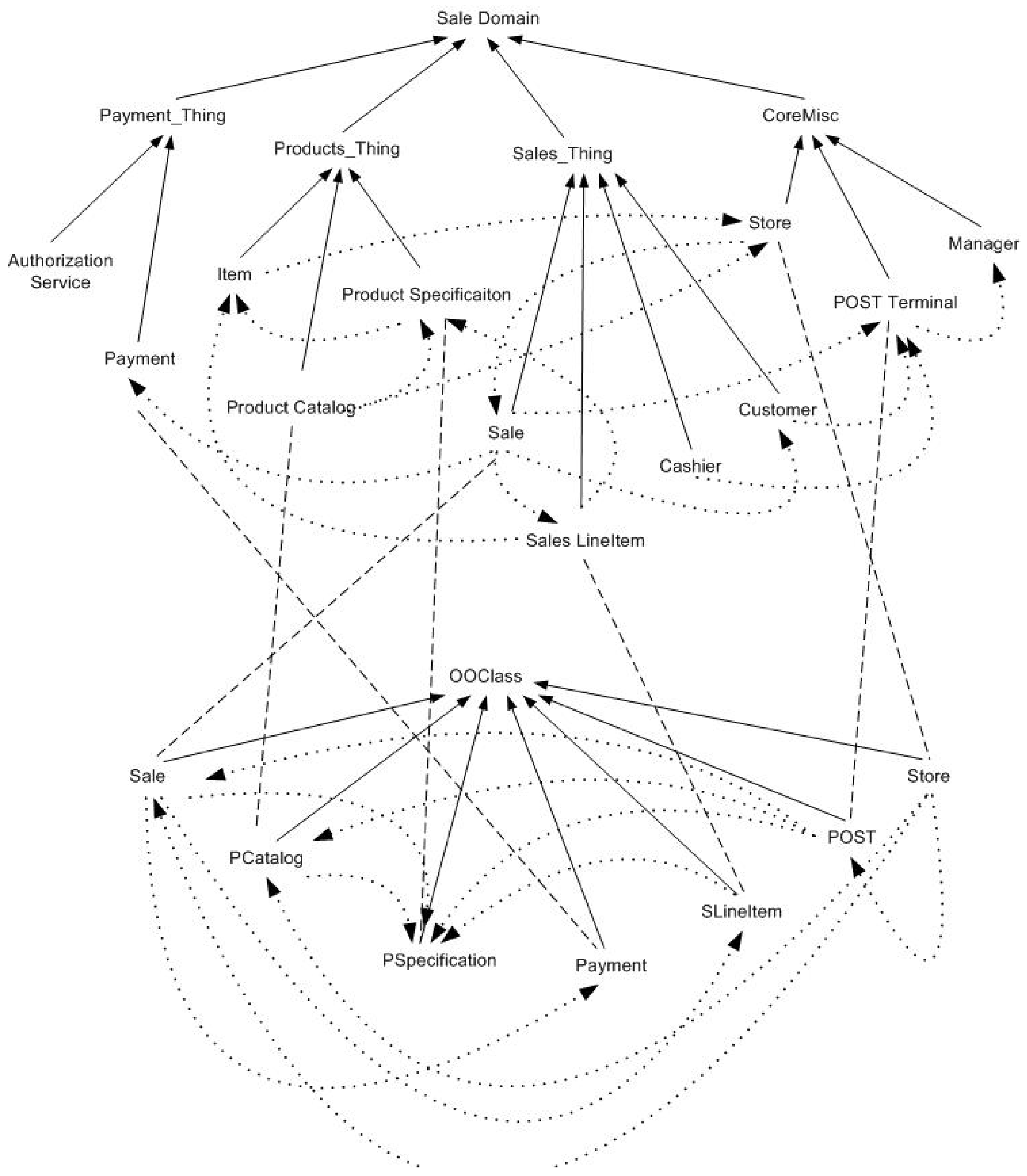


Figure 5-1 POST System Ontology and Domain Ontology

To describe the algorithm more easily, Figure 5-1 presents an example of point-of-sale terminal (POST) system ontology mapping. Firstly, POST system ontology Ops and POST domain ontology Opd are obtained via the proposed ontology generation approach. Secondly, in order to build the mapping between POST system source code and a POST system domain, so that program comprehension may be supported, the semantic relationships between any arbitrary concept in Ops and all the concepts in Opd will be checked. Initially, the algorithm will focus on discovering semantic mapping between one particular concept of source ontology and one particular concept of target ontology. In this POST terminal system example, the semantic relationship between concept POST terminal in POST domain ontology and concept POST in POST system ontology is checked. Once the semantic mapping between two particular concepts is achieved, the algorithm will then iterate to detect more semantic mappings between arbitrary concepts across different ontologies. The basic idea of implementing the proposed algorithm could be divided into four steps:

Step 1: to express the concepts C1 of O1 and C2 of O2 by DL formulae that contain other related concepts in O1 and O2 respectively and the relationships between them.

Step 2: to access a lexical knowledge base, which is WordNet in this study, in order to unify the terms used in DL formulae produced by step 1.

Step 3: to access WordNet to determine the semantic relationships among the unified terms generated in step 2, which will be one of the following two relationships: hypernyms and hyponyms.

Step 4: to detect the semantic relationships between the unified DL formulae that express C1 of O1 and C2 of O2 by DL reasoning. Along with the subsumption relationships obtained in step 3 as the premises for reasoning, tableau algorithm [6, 7] of DL is employed to reason the semantic relationships between complex concepts.

The algorithm is trying to sort out the problem that given two software system ontologies O1 and O2, for the arbitrary concepts C1 of O1 and C2 of O2, which one of the four semantic relationships (defined as Rel in Definition 6.1) is held between them.

The general algorithm will take two inputs:

- SK = <Cs, Os> is a 2-tuples which includes an arbitrary concept Cs and source ontology Os, and Cs $\in$ Os. For instance, in the above ontology mapping scenario, source code ontology Osc is Os, Cs is an arbitrary concept in Osc.

- TK = <Ct, Ot, Oaux> is a 3-tuples which includes an individual concept Ct, target ontology Ot and auxiliary ontology Oaux. For example, in the above ontology mapping scenario, relational database ontology Odb is Ot, and WORDNET ontology Oword is Oaux. The main goal of the algorithm is to find the semantic relationships between Cs in Os and all the concepts belonging to Ot. For the sake of simplicity, the algorithm is only focusing on checking the semantic relationship between Cs in Os and one individual concept Ct in Ot.

The output of the algorithm will simply be the semantic relationship existing between the concept Cs in Os and the concept Ct in Ot. According to definition 6.1, such semantic relationships Rel $\in \{ \subseteq, \supseteq, \equiv, \perp \}$. Correspondingly, the algorithm can be iterated in order to get the semantic relationships existing between any individual concept Cs in Os and all the concepts in Ot. As a result, any arbitrary concept in Os may be semantically related to at least one concept in Ot. Os and Ot are therefore integrated. The following sections will present the mapping algorithm in detail.

5.1.3 Software Ontology Mapping Algorithm – SWONTOMAP

Algorithm 5.1 SWONTOMAP (SK, TK)

```
Input:
    SK = <Cs, Os> is 2-tuples, where Cs is any arbitrary concept
                in the source ontology
                Os is the source ontology being mapped
                Cs ∈ Os
    TK = <Ct, Ot, Oaux> is 3-tuples, where Ct is one individual
                concept in the target ontology
                Ot is the target ontology being mapped
                Oaux is an auxiliary ontology to support mapping
                Ct ∈ Ot

Variable:
    α, β are Description Logic formulae
    relation Rel is returned binary semantic relation

Main Body:
1   α ← CONSTRUCT-DL-FORMULA (Cs, Os, Oaux);
2   β ← CONSTRUCT-DL-FORMULA (Ct, Ot, Oaux);
```

```
3  Rel ← SEMANTIC-DETECTION (α, β, Oaux);
4  Return Rel;
```

The mapping algorithm SWONTOMAP only has 4 lines in the main body. Line 1 builds a Description Logic formula α which expresses an individual concept Cs by the conjunction of all its superclasses and associated axioms that contain relationships between Cs and other concepts in the source ontology Os. Line 2 similarly builds the Description Logic formula β to express the individual concept Ct by the conjunction of all its superclasses and associated axioms that contain relationships between Ct and other concepts in the target ontology Ot. Finally, line 3 detects the semantic relationship between the two Description Logic formulae. Line 4 returns the semantic relationships detected between Cs and Ct. The following two sections will describe the implementation of those two top-level sub-algorithms CONSTRUCT-DL-FORMULA and SEMANTIC-DETECTION in more detail.

5.1.4 Sub-algorithm – CONSTRUCT-DL-FORMULA

Algorithm 5.2 CONSTRUCT-DL-FORMULA (Ci, O, Oaux)

```
Input:
   Ci is an arbitrary concept
   O is an ontology
   Oaux is an auxiliary ontology to support the mapping
   Ci ∈ O

Variable:
   array SynAxiomSet[][] stores a set of Description Logic axioms which
   express the synonyms of a given concept.
   sub-ontology Ors is a reduced sub-ontology of O
   Cj represents an arbitrary concept Cj ∈ Ors    1<j<|Ors|
   |.| is a function which calculates the number of concepts in ontology
   χ is a returned Description Logic formula

Main Body:
1  Ors ← REDUCED-SUB-ONTOLOGY (Ci, O);
```

```
2   for (j=1; j<| Ors |; j++)
3      {SynAxiomSet[Cj][] ← GENERATE-SYNAXIOMSET (Cj, Ors)};
4   for (j=1; j<| Ors |; j++)
5 { SynAxiomSet[Cj][] ← SYNAXIOMSET-FILTER (Cj, Ors, Oaux,
6                                               SynAxiomSet[][])};
7   χ ← CONSTRUCT-DLAXIOM-CONCEPT (Ci, SynAxiomSet[][], Ors, Oaux);
8   Return χ;
```

This sub-algorithm constructs a comparable structure via an arbitrary concept Ci in an ontology O with the help of an auxiliary ontology Oaux. Line 1 produces a reduced sub-ontology Ors related to the concept Ci. The implementation of REDUCED-SUB-ONTOLOGY is quite simple. It is just to rebuild the ontology by including the given concept Ci and all its superclasses, subclasses and the concepts that are related to Ci. Line 2 and Line 3 compose a for-each loop, in which each concept Cj in ontology Ors is assigned a set of synonyms which are expressed in Description Logic axioms. Line 4 to Line 6 also compose a for-each loop, in which the set of axioms of each concept Cj in ontology Ors is filtered by SYNAXIOMSET-FILTER, unreasonable synonym axioms associated to Cj will be removed. Lastly, Line 7 builds the Description Logic formula χ by sub-algorithms CONSTRUCT-DLAXIOM-CONCEPT in relation to the filtered synonym set of Cj, concept Ci and ontology Ors.

5.1.5 Sub-algorithm – SEMANTIC-DETECTION

Algorithm 5.3 SEMANTIC-DETECTION (α, β, Oaux)

Input:

```
Description Logic formula α
Description Logic formula β
ontology Oaux
```

Variable:

```
array μ[] is a set of Description Logic axioms
array κ[] is to store deductional pairs
```

Main Body:

```
1   μ[] ← CONSTRUCT-GLOBAL-AXIOMS (α, β, Oaux);
```

```
2  $\kappa$[] $\leftarrow$ GENERATE-DEDUCTIONAL-FORMULAE ($\alpha$, $\beta$, $\mu$[]);
3  for (i=1; i<|$\kappa$[]|; i++)
4  {if SATISFIES($\neg\kappa$[i].formula) then
5   Return $\kappa$[i].relation;
6   else Return Null; }
```

Line 1 constructs a global axiom which indicates the semantic relationships existing between individual concepts belonging to two different Description Logic formulae α and β in line with the auxiliary ontology. Line 2 builds an array to store deductional pairs. The deductional pairs are encoded as <formula, relation>. The formula is a Description Logic formula and the relationship is the semantic one obtained between two axioms when the formula holds. E.g. α and β are two Description Logic formulae; $\{\subseteq, \supseteq, \equiv, \perp\}$ are the possible semantic relationships that exist between two Description Logic formulae; the deductional pairs regarding their semantic relationships will be $<\alpha\cap\neg\beta\rightarrow\perp, \subseteq>$, $<\neg\alpha\cap\beta\rightarrow\perp, \supseteq>$, $<(\alpha\cap\neg\beta)\leftrightarrow(\neg\alpha\cap\beta), \equiv>$ and $<\alpha\cap\beta\rightarrow\perp, \perp>$. Line 3 to Line 6 is to seek the semantic relationship existing between the two Description Logic formulae α and β, and it is implemented by testing the satisfiability of the DL formula in each deductional pair with tableau algorithm. When the formula is found to be satisfiable, the associated relationship is then returned.

5.1.6 Sub-algorithm – SYNAXIOMS-FILTER

Algorithm 5.4 SYNAXIOMSET-FILTER (Cj, Ors, Oaux, SynAxiomSet[][])

```
Input:
    Ontology Ors is reduced sub-ontology
    Ontology Oaux is auxiliary ontology
    Concept Cj
    array SynAxiomSet[][]

Variable:
    relation Rel = Null
    Csup is superclass of Cj
    Csub is subclass of Cj
    Csib is sibling class of Cj
    Description Logic axiom {synaxiom, synaxiomsuper, synaxiomsub,
    Synaxiomsib} ∈ SynAxiomSet[][]
```

Main Body:

```
1   for each synaxiom in SynAxiomSet[Cj][]
2       for each superclass Csup of Cj in Ors do
3           for each synaxiomsuper in SynAxiomSet[Csup][]
4               Rel ← EXTRACT-RELATION (synaxiomsuper, synaxiom, Oaux);
5           if (Rel = Null) then
6               remove synaxiom off SynAxiomSet[Cj][];
7           Rel ← Null;
8   for each synaxiom in SynAxiomSet[Cj][]
9       for each subclass Csub of Cj in Ors do
10          for each synaxiomsuper in SynAxiomSet[Csup][]
11              Rel ← EXTRACT-RELATION (synaxiomsub, synaxiom, Oaux);
12          if (Rel = Null) then
13              remove synaxiom off SynAxiomSet[Cj][];
14          Rel ← Null;
15  for each synaxiom in SynAxiomSet[Cj][] do
16      for each sibling class Csib of Cj in Orb do
17          for each synaxiomsib in SynAxiomSet[Csib][] do
18              Rel ← EXTRACT-RELATION (synaxiomsib, synaxiom, Oaux);
19          if (Rel3 = Null) then
20              remove synaxiom off SynAxiomSet[Cj][];
21  Return SynAxiomSet[Cj][];
```

The function of this sub-algorithm is to eliminate those synonym axioms associated to the given concept Cj which are obviously contradictory to the context of Cj. Firstly, Line 1 to Line 7 is to check the contradictory axiom with the SynAxiomSet[Csup][] of the superclasses of Cj. And the axiom will be removed if the axiom is not related to any axiom associated to Csup. Secondly, Line 8 to Line 14 is to check the contradictory axiom with the SynAxiomSet[Csub][] of the subclasses of Cj. And the axiom will be removed if the axiom is not related to any axiom associated to Csub. Thirdly, Line 15 to Line 20 is to check the contradictory axiom with the SynAxiomSet[Csib][] of the sibling classes of Cj. And the axiom will be removed if the axiom is not related to any axiom associated to Csib.

5.1.7 Sub-algorithm – CONSTRUCT-DLAXIOM-CONCEPT

Algorithm 5.5 CONSTRUCT-DLAXIOM-CONCEPT (Cj, SynAxiomSet[][], Ors, Oaux)

```
Input:
    concept Cj
    array SynAxiomSet[][]
    Ontology Ors
    ontology Oaux

Variable:
    formula $\nu$ = Null
    Relation Rel1, Rel2 = Null
    concept Csib is sibling class of concept Cj
    Set subclassSet[Cj] is a set of all the subclasses of Cj
    Set superclassSet[Cj] is a set of all the superclasses of Cj
    Csub $\in$ subclassSet[Cj]
    Csuper $\in$ superclassSet[Cj]
    int e1 is the number of subclasses in subclassSet[Cj]
    int e2 is the number of superclasses in superclassSet[Cj]

Main Body:
1   for each SynAxiomSet[Cj][i] in SynAxiomSet[Cj] do
2       for each Csib in Ors do
3           for each SynAxiomSet[Csib][n] in SynAxiomSet[Csib] do
4               Rel1 $\leftarrow$ EXTRACT-RELATION (SynAxiomSet[Cj][i],
5                       SynAxiomSet[Csib][n], Oaux);
6               if Rel = hypernym then
7                   Rel2 $\leftarrow$ hypernym;
8           if (Rel2 $\neq$ Null) then
9   SynAxiomSet[Cj][i] $\leftarrow$ SynAxiomSet[Cj][i]$\cap$ $\neg$SynAxiomSet[Csib][n];
10 e1 $\leftarrow$ |subclassSet[Cj]|;
11  $\nu$ $\leftarrow$ $\cap$e1 ($\cup$i SynAxiomSet[Csub][i]);
12 e2 $\leftarrow$ |superclassSet[Cj]|;
13  $\nu$ $\leftarrow$ $\cap$e2 SynAxiomSet[Csuper][i];
14 Return $\nu$;
```

This sub-algorithm is to construct a formula which expresses the semantic of concept Cj.

Line 1 to Line 9 is to seek semantic relationships between the axioms of the given concept Cj and its siblings. If a semantic relationship is detected, the SynAxiomSet[Cj][] will be refined by excluding the axiom of that sibling. Line 10 and Line 11 construct the formula ν by the conjunction of the axioms associated to all its subclasses and the axioms calculated by the disjunction of all the axioms associated to the concept Cj. As a result, the formula ν will approximate the meaning of concept Cj.

5.2 Using Description Logic

The underlying theory of the proposed ontology mapping algorithm is to transfer the detection of semantic relationships to the deduction of satisfiability of logical formulae. As a member of a knowledge representation family, Description Logic is selected to represent the concepts and relationships of the ontology in a structured and formally defined way in order to support the mapping algorithm.

5.2.1 Representing Software Systems Concepts in Description Logic

Firstly, for operating system ontology, some obvious classes of individuals including thread, timer, semaphore, mutex and message, etc. are normally modelled using atomic/primitive concepts in Description Logic. For object oriented data dominant system ontology, atomic/primitive concepts are used to model each primitive data type such as String, Date, Integer Float and Boolean, etc.

Secondly, other classes such as pthread_create_api, pthread_mutex_destroy_api are more complicated and are normally modelled as defined concepts in Description Logic. For object oriented data dominant system ontology, each object oriented class is considered to be a complex concept and therefore to be modelled as a defined class with properties. There are essential properties and incidental properties to be defined to distinguish primitive from defined concepts. Necessary and sufficient conditions and necessary conditions are used to define those different properties.

Following are some examples of the defined concepts of different software system ontologies represented in Description Logic.

$$
\begin{aligned}
\text{tx\_pthread\_create} \equiv\ & \text{API} \\
& \cap \forall\ \text{definedInOS.ThreadX} \\
& \cap \forall\ \text{hasAPIStandard.NONPOSIX} \\
& \cap \forall\ \text{hasReturnType.unsigned\_int} \\
& \cap \forall\ \text{provideService.thread\_service\_create} \\
& \cap \forall\ \text{hasParameter.tx\_threadPointer\_thread\_ptr}
\end{aligned}
$$

The above DL formula describes an operating system ontology concept tx_pthread_create, which is an API that defined in a ThreadX operating system, does not implement POSIX standard, returns an unsigned_int type value, takes a pointer as argument, and implements the function of thread creating.

$$
\begin{aligned}
\text{POST\_Terminal} \equiv\ & \text{CoreMisc} \\
& \cap \forall\ \text{Startedby.Manager} \\
& \cap \forall\ \text{Operatedby.Cashier} \\
& \cap \forall\ \text{Captures.Sale} \\
& \cap \forall\ \text{Locatedin.Store} \\
& \cap \forall\ \text{Looks-in.ProductCatalog} \\
& \cap \forall\ \text{Queries.ProductSpecification}
\end{aligned}
$$

The above DL formula describes an application domain ontology concept POST Terminal, which is a core miscellaneous concept that is located in store, is started by the manager, is operated by the cashier, captures the sale, looks in the product catalogue, and queries the product specification.

$$
\begin{aligned}
\text{UML\_POST} \equiv\ & \text{UML\_OOClass} \\
& \cap \forall\ \text{associateto.UML\_Store} \\
& \cap \forall\ \text{associateto.UML\_ProductCatalog} \\
& \cap \forall\ \text{associateto.UML\_Sale} \\
& \cap \forall\ \text{dependon.UML\_ProductSpecification}
\end{aligned}
$$

The above DL formula describes a UML class diagram ontology concept UML_POST, which is an object oriented class that associates to UML_Store, associates to UML_ProductCatalog, associates to UML_Sale, and depends on UML_ProductSpecification.

In many cases there are specialised subconcepts representing subsets of individuals that are also of interest. There are a few special aspects of the subconcepts that should be modelled in order to capture the knowledge of a software system properly. Normally, subclasses should be disjoint from each other. For example, in operating system ontology, thread_api and mutex_api are disjoint subclasses of API. Description Logic supports negation which is modelled by adding the complement of one concept to the necessary properties of the other concept. Normally, entire collections of subclasses are disjoint.

$$\text{thread\_api} \subseteq \neg\, (\text{mutex\_api} \cup \text{semaphore\_api} \cup \text{message\_queue\_api})$$

5.2.2 Representing Software Systems Relationships in Description Logic

In Description Logic, binary relationships are modelled as roles and properties. The following are frequently used constraints to express relationships in this research:

- Cardinality constraints – indicate the range of the number of classes that can be linked to the main class via a role;
- Domain constraints – indicate the kind of classes that can be linked to the main class via a role;
- Inverse constraints – indicate the inverse relationships between the roles.

For instance, an operating system API has exactly one return type, which is a Data_type, and exactly one API standard, which is either POSIX or NONPOSIX; an operating system API may have none or more parameters; the role definedInOS is the inverse role of hasAPI.

Sometimes, reified relationships also need to be considered in order to model the problem domain more accurately, which indicates that properties can also be defined by other properties. When defining a reified relationship, it is necessary to distinguish those properties determining the reified relationship from the ones qualifying it. Like the concepts, properties also have hierarchical structure – properties can inherit other properties.

5.2.3 Supporting Ontology Mapping Algorithms with Description Logic

In this section, a description logic based ontology mapping algorithm is demonstrated in detail with the example given in Section 5.1.2. POST system ontology Ops is the source ontology, POST system domain ontology Opd is the target ontology. UML_POST is a concept in POST system ontology, and POST_Terminal is a concept in POST domain ontology. The mapping algorithm will be applied to detect the semantic relationship between those two concepts.

As described in Section 5.1.2, the problem of detecting semantic relationships will be eventually converted into the problem of deducing the satisfiability of logical formulae. The first step is to express the concepts UML_POST and POST_Terminal by DL formulae that contain other related concepts in POST system ontology and POST domain ontology respectively along with the relations among them.

$$\begin{aligned}\text{POST\_Terminal} \equiv\ & \text{CoreMisc}\\ & \cap \forall\ \text{Startedby.Manager}\\ & \cap \forall\ \text{Operatedby.Cashier}\\ & \cap \forall\ \text{Captures.Sale}\\ & \cap \forall\ \text{Locatedin.Store}\\ & \cap \forall\ \text{Looks-in.ProductCatalog}\\ & \cap \forall\ \text{Queries.ProductSpecification}\end{aligned}$$

$$\begin{aligned}\text{UML\_POST} \equiv\ & \text{UML\_OOClass}\\ & \cap \forall\ \text{associateto.UML\_Store}\\ & \cap \forall\ \text{associateto.UML\_ProductCatalog}\\ & \cap \forall\ \text{associateto.UML\_Sale}\\ & \cap \forall\ \text{dependon.UML\_ProductSpecification}\end{aligned}$$

In the next stage, the terms in description logic formulae have to be unified via WordNet ontology. For example, the concept of store has two synonym concepts. Store#1 means shop, store#2 means storage. After applying algorithm 5.4 SYNAXIOMS-FILTER, the second synonym can be removed. Since reverse engineered UML class diagram can only represent two simple relationships, i.e., association and dependency, which are hypernyms to any other binary relationships, all the relationships will be replaced by associateto. Therefore, the two concepts can be represented as the following:

$$\begin{aligned}
\text{POST\#1} \equiv\ & \text{CoreMisc\#1} \\
& \cap \forall\ \text{associateto.Manager\#1} \\
& \cap \forall\ \text{associateto.Cashier\#1} \\
& \cap \forall\ \text{associateto.Sale\#1} \\
& \cap \forall\ \text{associateto.Store\#1} \\
& \cap \forall\ \text{associateto.ProductCatalog\#1} \\
& \cap \forall\ \text{associateto.ProductSpecification\#1}
\end{aligned}$$

$$\begin{aligned}
\text{POST\#2} \equiv\ & \text{UML\_OOClass\#1} \\
& \cap \forall\ \text{associateto.UML\_Store\#1} \\
& \cap \forall\ \text{associateto.UML\_ProductCatalog\#1} \\
& \cap \forall\ \text{associateto.UML\_Sale\#1} \\
& \cap \forall\ \text{associateto.UML\_ProductSpecification\#1}
\end{aligned}$$

After unifying the two DL formulae, the subsumption relationships are searched among the concepts in the formulae. In this example, UML_OOClass#1 and CoreMisc#1 are assigned the subsumption relationship as object oriented class can implement any core miscellaneous concept. Therefore

$$\text{UML\_OOClass\#1} \subseteq \text{CoreMisc\#1}$$

In the final stage, the mapping algorithm will deduce satisfiability of the following four DL formulae.

- $\text{POST\#1} \subseteq \text{POST\#2} \Leftrightarrow \text{POST\#1} \cap \neg\ \text{POST\#2}$ is unsatisfiable.
- $\text{POST\#2} \subseteq \text{POST\#1} \Leftrightarrow \text{POST\#2} \cap \neg\ \text{POST\#1}$ is unsatisfiable.
- $\text{POST\#1} \equiv \text{POST\#2} \Leftrightarrow (\text{POST\#1} \cap \neg\ \text{POST\#2})$ is unsatisfiable. And $(\text{POST\#2} \cap \neg\ \text{POST\#1})$ is unsatisfiable.
- $\text{POST\#1} \perp \text{POST\#2} \Leftrightarrow \text{POST\#1} \cap \text{POST\#2}$ is unsatisfiable.

POST#1 and POST#2 will then be extended with the complex DL formulae. To simplify the description, let:

C_0 = POST#1 (1)
C'_0 = POST#2 (2)
C_1 = Manager#1 (3)
C_2 = Cashier#1 (4)
C_3 = Sale#1 (5)
C_4 = Store#1 (6)
C_5 = ProductCatalog#1 (7)
C_6 = ProductSpecification#1 (8)
C_7 = CoreMisc (9)

C_8 = UML_OOClass#1 (10)

associateto = R (11)

$C_0 = C_7 \cap \forall R.C_1 \cap \forall R.C_2 \cap \forall R.C_3 \cap \forall R.C_4 \cap \forall R.C_5 \cap \forall R.C_6$ (12)

$C'_0 = C_8 \cap \forall R.C_4 \cap \forall R.C_5 \cap \forall R.C_3 \cap \forall R.C_6$ (13)

$C_8 \subseteq C_7$ (14)

$C_0 \subseteq C'_0 \Leftrightarrow C_0 \cap \neg C'_0$ (15)

$C'_0 \subseteq C_0 \Leftrightarrow C'_0 \cap \neg C_0$ (16)

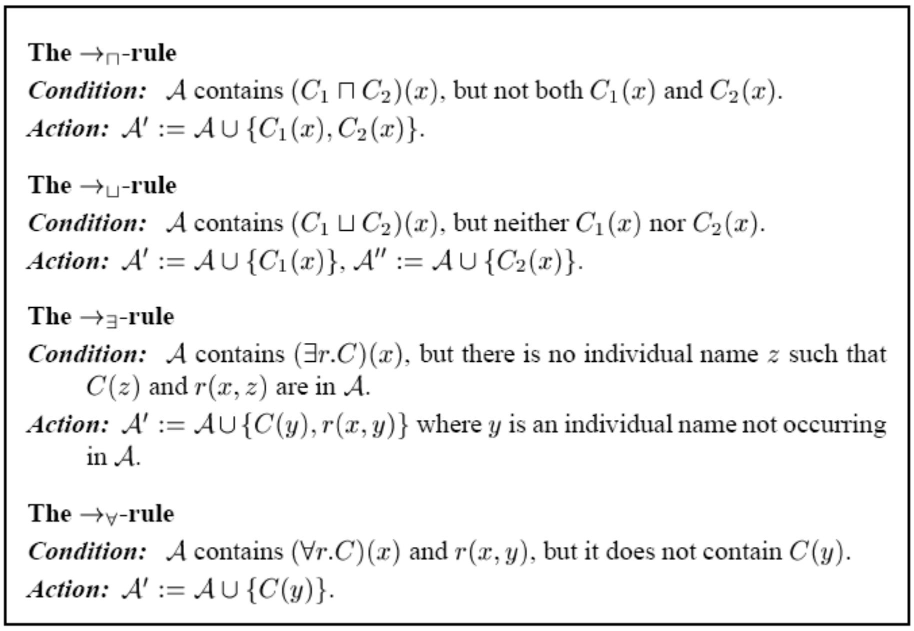
$(C_7 \cap \forall R.C_1 \cap \forall R.C_2 \cap \forall R.C_3 \cap \forall R.C_4 \cap \forall R.C_5 \cap \forall R.C_6) \cap$
$\neg(C_8 \cap \forall R.C_4 \cap \forall R.C_5 \cap \forall R.C_3 \cap \forall R.C_6)$ (17)

$(C_8 \cap \forall R.C_4 \cap \forall R.C_5 \cap \forall R.C_3 \cap \forall R.C_6) \cap$
$\neg (C_7 \cap \forall R.C_1 \cap \forall R.C_2 \cap \forall R.C_3 \cap \forall R.C_4 \cap \forall R.C_5 \cap \forall R.C_6)$ (18)

$C_7 \cap \forall R.C_1 \cap \forall R.C_2 \cap \forall R.C_3 \cap \forall R.C_4 \cap \forall R.C_5 \cap \forall R.C_6 \cap$
$(\neg C_8 \cup \exists R.\neg C_4 \cup \exists R.\neg C_5 \cup \exists R.\neg C_3 \cup \exists R.\neg C_6)$ (19)

$C_8 \cap \forall R.C_4 \cap \forall R.C_5 \cap \forall R.C_3 \cap \forall R.C_6 \cap$
$(\neg C_7 \cup \exists R.\neg C_1 \cup \exists R.\neg C_2 \cup \exists R.\neg C_3 \cup \exists R.\neg C_4 \cup \exists R.\neg C_5 \cup \exists R.\neg C_6)$ (20)

Formula (15) and (16) are two examples of checking the semantic relationship POST#1 $\subseteq$ POST#2 and POST#2 $\subseteq$ POST#1. Formula (17) and (18) extend (15) and (16) by replacing (1) and (2) with (12) and (13). (19) and (20) are generated by applying De Morgan's laws to (17) and (18).



The $\rightarrow_{\sqcap}$-rule
Condition: $\mathcal{A}$ contains $(C_1 \sqcap C_2)(x)$, but not both $C_1(x)$ and $C_2(x)$.
Action: $\mathcal{A}' := \mathcal{A} \cup \{C_1(x), C_2(x)\}$.

The $\rightarrow_{\sqcup}$-rule
Condition: $\mathcal{A}$ contains $(C_1 \sqcup C_2)(x)$, but neither $C_1(x)$ nor $C_2(x)$.
Action: $\mathcal{A}' := \mathcal{A} \cup \{C_1(x)\}$, $\mathcal{A}'' := \mathcal{A} \cup \{C_2(x)\}$.

The $\rightarrow_{\exists}$-rule
Condition: $\mathcal{A}$ contains $(\exists r.C)(x)$, but there is no individual name z such that $C(z)$ and $r(x,z)$ are in $\mathcal{A}$.
Action: $\mathcal{A}' := \mathcal{A} \cup \{C(y), r(x,y)\}$ where y is an individual name not occurring in $\mathcal{A}$.

The $\rightarrow_{\forall}$-rule
Condition: $\mathcal{A}$ contains $(\forall r.C)(x)$ and $r(x,y)$, but it does not contain $C(y)$.
Action: $\mathcal{A}' := \mathcal{A} \cup \{C(y)\}$.

Figure 5-2 Transformation Rules of Tableau Algorithm [6, 7]

Assume that there is an individual b which satisfies formula (19). After applying the transformation rules of tableau algorithm [6, 7] (Figure 5-2), b must satisfy the following five constraints:

$b \in C_7$, $b \in \forall R.C_1$, $b \in \forall R.C_2$, $b \in \forall R.C_3$, $b \in \forall R.C_4$, $b \in \forall R.C_5$,
$b \in \forall R.C_6$ and $b \in \neg C_8$, (a)
Or
$b \in C7$, $b \in \forall R.C1$, $b \in \forall R.C2$, $b \in \forall R.C3$, $b \in \forall R.C4$, $b \in \forall R.C5$, $b \in \forall R.C6$ and
$b \in \exists R.\neg C4$, (b)
Or
$b \in C7$, $b \in \forall R.C1$, $b \in \forall R.C2$, $b \in \forall R.C3$, $b \in \forall R.C4$, $b \in \forall R.C5$, $b \in \forall R.C6$ and
$b \in \exists R.\neg C5$, (c)
Or
$b \in C7$, $b \in \forall R.C1$, $b \in \forall R.C2$, $b \in \forall R.C3$, $b \in \forall R.C4$, $b \in \forall R.C5$, $b \in \forall R.C6$ and
$b \in \exists R.\neg C3$, (d)
Or
$b \in C7$, $b \in \forall R.C1$, $b \in \forall R.C2$, $b \in \forall R.C3$, $b \in \forall R.C4$, $b \in \forall R.C5$, $b \in \forall R.C6$ and
$b \in \exists R.\neg C6$, (e)

Since formula (14) $C_8 \subseteq C_7$, therefore, $C_8 \cap \neg C_7 \subseteq \perp$, so there is a clash in (a) between $b \in C_7$ and $b \in \neg C_8$. Furthermore, clashes will be detected between $b \in \forall R.C_4$ and $b \in \exists R.\neg C_4$ in (b), $b \in \forall R.C_5$ and $b \in \exists R.\neg C_5$ in (c), $b \in \forall R.C_3$ and $b \in \exists R.\neg C_3$ in (d), $b \in \forall R.C_6$ and $b \in \exists R.\neg C_6$ in (e). Hence, there is no such individual b which will satisfy the formula (19). Formula (19) is unsatisifiable. Therefore, $C_0 \subseteq C'_0$, i.e., POST#1 $\subseteq$ POST#2. Analogously, formula (20) is satisifiable, which indicates that C'_0 is not subsumed by C_0, i.e., POST#2 is not subsumed by POST#1.

This example shows that with the DL based ontology mapping algorithm, a subsumption relationship between two concepts across POST domain ontology and POST system ontology can be found. This subsumption relationship indicates that class POST in UML diagram implements the function of POST terminal. Since the code is more abstract than the concrete concept in domain, the domain terms will be subsumed by the code concepts.

5.3 Summary

In this chapter, methodologies for integrating different software system ontologies are presented. Ontology integration has many synonyms in the ontology engineering research area. In this study, software system ontology integration indicates ontology mapping, which is a process of finding semantic relationships between entities (e.g., concepts, relationships, etc.) across two different software system ontologies. A DL-based ontology mapping approach is proposed by extending the CTXMATCH algorithm by exploring the expressive power and efficient reasoning of description logic. The basic idea of the proposed mapping algorithm is to represent terms of both source ontology and target ontology by DL formulae with relevant lexical, domain and structural knowledge, and then to transform the problem of detecting semantic relationships into the problem of deducing satisfiability of DL formulae..

- Software system ontology integration is defined as an activity to detect a semantic relationship between a term (concepts, relations, etc.) of software system ontology (source ontology) and a term of a different software system ontology (target ontology).

- Four steps are defined to implement the proposed mapping algorithm. Firstly, the concepts are represented with DL formulae. Secondly, the terms in DL formulae are unified by accessing WordNet. Thirdly, subsumption relationships between the terms across the DL formulae are detected by accessing WordNet. Fourthly, a tableau algorithm is applied to deduce the satisfiability of the generated DL formulae. As a result, the semantic relationships between the concepts are returned.

- Representing software system ontology with description logic is discussed, including representing primitive concepts, representing defined concepts and representing binary relationships (i.e. role/object property).

- Examples are given to show how to represent software system ontology in DL. Furthermore, an example is given to demonstrate the DL-based ontology mapping algorithm in detail.

Chapter 6 Software System Ontology Deployment and Use Cases

Objectives

- To demonstrate deploying software system ontology in different software reengineering scenarios
- To discuss the deployment of operating system ontology in platform-specific software migration/porting
- To discuss the deployment of operating system ontology in portable embedded software development
- To discuss the deployment of data-dominant software system ontology in program comprehension
- To discuss the deployment of data-dominant software system ontology in software modularisation
- To demonstrate the ontology deployment with relevant use cases

This chapter discusses the deployment of software system ontology. Different software reengineering scenarios have been selected to demonstrate the usage of software system ontology.

Firstly, the deployment of operating system ontology is explored in the field of platform-specific software migration/porting and portable software development, which may look similar but actually focus on different aims. Portable software implies that the software was initially designed to fit several different platforms while software migration/porting indicates making an existing software application run successfully on a different platform by replacing one set of system dependencies with another. Thus, providing a mapping between different platform dependencies is a crucial requirement in software migration/porting and operating system ontology can meet this requirement by building the mappings based on knowledge acquisition. To ensure software portability, one of the common solutions is to provide a standard set of application programming interfaces (APIs) which implement system call services that are available on all the target platforms. For example, the POSIX interface is designed to be portable and the POSIX standard contains most of the standard UNIX compatible system call interfaces. The hierarchical classification that operating system ontology provides will help to create standardisation.

Secondly, the deployment of data-dominant software ontology is explored in the field of program comprehension and software modularisation. Traditional approaches for software comprehension typically take either a code-based program comprehension, or a documentation-based one. However, in practice, neither code-based nor documentation-based program understanding is sufficient. It is necessary to develop a new software representation technique that embodies software systems on both program level and it's corresponding domain concept level, that is to say, to formulate a representation which contains both source code knowledge and domain knowledge. Ontology-based program comprehension is developed based on this idea. Furthermore, software modularisation could be processed based on the ontology-based comprehension. The components and their interrelationships in the software system will be analysed by examining concepts and their relationships in the software system ontology. The strongly related components need to stay together while loosely coupled ones can be separate.

The following sections will discuss in depth the deploying of software system ontology in the above software reengineering scenarios.

6.1 Deploying Operating System Ontology to Facilitate Platform-Specific Software Migration/Porting

6.1.1 Platform-Specific Software Migration/Porting

A Real-Time Operating System (RTOS) is becoming crucial in embedded systems. Because of the ever increasing complexity of embedded systems, RTOS is employed to fulfil the requirements of managing precise timing and limited resources for different real-time applications. RTOS is responsible for allocating the processors and computing resources to the cooperating tasks to enable them to execute according to their timing constraints. Platform specific software migration/porting is one of the significant problems in RTOS-based software reengineering domain.

6.1.2 Ontology-based PlaTform specIfic software Migration Approach (OPTIMA)

As an inherently knowledge intensive activity, platform specific software migration requires a great deal of knowledge in areas ranging from expertise to experience in the different platform domains. The proposed method is an Ontology-based PlaTform specIfic software Migration Approach, called OPTIMA [132]. To port a program (semi-) automatically while keeping certain properties invariant, migration rules need to be defined. Software porting depends on matching detection. If inputs are matched with predefined patterns, the system will be rewritten according to the migration rules. For example, if maintainers are willing to get the information about existing POSIX APIs that are defined in both systems providing a thread creating service, they can query the knowledge base by querying APIs which are defined in RTLinux and ThreadX, implementing a POSIX standard and providing a thread creating service. This particular query will be performed by an ontology query language or a DL reasoning service, which is provided in the OPTIMA toolkit. If the query result suggests that both systems have a POSIX API implemented to create a thread, then that part of the application can be migrated by directly changing the API's names. If the query result shows that both systems do not have such a POSIX API, then the application will need to be rewritten during the migration. Based on the RTOS ontology repository and knowledge acquisition, program migration rules are defined for software porting between different platforms. Every migration rule starts with querying the ontology repository to get information on both source and target APIs. Each query to the operating system ontology includes:

- hasAPIStandard(API_Standard),
- hasParameter(Parameter),
- provideService(System_service), and
- hasReturnType(Data_type).

If both source and target APIs match the concepts in the operating system ontology, platform specific software migration/porting will be performed based on the predefined migration rules. Otherwise, automatic migration of source API cannot be performed. Given the fact that some parts of the program cannot be processed by automatic software migration/porting, the ontology repository can still provide useful information on source and target platforms which can enables maintainers to rewrite the program manually. Hence, with human intervention, OPTIMA can provide a practical semi-automated solution to platform-specific software migration/porting.

6.1.2.1 Rules for RTOS Specific Software Migration

To make use of the RTOS ontology repository, experiments are undertaken with an OPTIMA toolkit to perform ontology-based program transformation and knowledge acquisition. A series of transformation rules have been defined.

Rule Set 1: ANSI_C2ANSI_C API Transformation

Preconditions:

1. assert(SourceAPI.isInstance(true)), and
2. assert(SourceAPI.hasAPIStandard.equals(POSIX)), and

3. assert(SourceAPI.getRDFType().equals(ANSI_C_api));

Transformation rules:

TargetAPI.all()=SourceAPI.all();

Rule Set 2: POSIX2POSIX API Transformation

Preconditions:

1. assert(SourceAPI.isInstance(true)), and
2. assert(TargetAPI.isInstance(true)), and
3. assert(SourceAPI.hasAPIStandard.equals(POSIX)), and
4. assert(TargetAPI.hasAPIStandard.equals(POSIX)), and
5. assert(!SourceAPI.getRDFType().equals(ANSI_C_api));

Transformation rules:

TargetAPI.all()=SourceAPI.all();

Rule Set 3, 4 and 5: POSIX2NONPOSIX, NONPOSIX2POSIX, NONPOSIX2NONPOSIX API Transformation

Preconditions:

1. assert(SourceAPI.isInstance(true)), and
2. assert(TargetAPI.isInstance(true)), and
3. (assert(SourceAPI.hasAPIStandard.equals(POSIX)), and

   assert(TargetAPI.hasAPIStandard.equals(NONPOSIX))),

Or (assert(SourceAPI.hasAPIStandard.equals(NONOSIX)), and

   assert(TargetAPI.hasAPIStandard.equals(POSIX))), Or (assert(SourceAPI.hasAPIStandard.equals(NONPOSIX)), and

assert(TargetAPI.hasAPIStandard.equals(NONPOSIX))), and

4. assert(!SourceAPI.getRDFType().equals(ANSI_C_api));

Transformation rules:

Replace target API with source API.

6.1.3 Use case of OPTIMA

The use case has been selected from [125], which is a simplified pump control system for a mining environment. The system is used to pump mine water, which collects in a sump at the bottom of the shaft, to the surface. The main safety requirement is that the pump should not be operated when the level of methane gas in the mine reaches a high value due to the risk of explosion. Such a system was first implemented in an RTLinux environment previously, which needs to be run on a ThreadX platform now, i.e. the software migration from RTLinux to ThreadX. This use case is a good example of an RTOS specific software migration, which helps to demonstrate the properties of an OPTIMA approach.

6.1.3.1 RTOS Specific Program Transformation

Based on an OPTIMA approach, knowledge acquisition based transformation rules have been applied, and a program transformation will be performed. A section of code from selected use case is presented to illustrate the OPTIMA approach. The following is an RTLinux version of a sample of the source code.

```
mqid_t   MsgQueID;
void* ThreadProc(void* para)
{
#define MSGLEN   32
     char msgbuf[MSGLEN];
     int msglen;
     int prio;
     msglen = mq_receive(MsgQueID, msgbuf, MSGLEN, &prio);
     pthread_cancel(pthread_self());
     return 0;
}
void* ThreadProc2(void* para)
{
	int ret;
	ret = mq_send(MsgQueID,"Start",6,0);
#if 0
	pthread_sleep(pthread_self(),400);
#endif
	pthread_cancel(pthread_self());
	return 0;
}
void main()
{
	pthread_attr_t	attr;
	pthread_t		ThreadId1;
	pthread_t		ThreadId2;
	StartVos();
	MsgQueID = mq_open("Thread1MessageQueue",O_CREAT | O_APPEND, 0, NULL);
	pthread_attr_init(&attr);
	pthread_attr_setschedpolicy(&attr,SCHED_FIFO);
	pthread_create(&ThreadId1,&attr,ThreadProc,(void *)1);
	pthread_setschedprio(ThreadId1,9);
	pthread_create(&ThreadId2,&attr,ThreadProc2,(void *)2);
	pthread_setschedprio(ThreadId2,9);
	pthread_attr_destroy(&attr);
	Idle();
	mq_close(MsgQueID);
}
```

APIs in source code are extracted with the parser and matched to the APIs provided by target platforms via knowledge acquisitions. Based on different sets of transformation rules, most of the source code can be transformed to act on target platforms automatically, whilst part of the source code has to be transformed by intervention from software maintainers. For instance, an RTLinux POSIX/ANSI C API can be transformed into a ThreadX POSIX/ANSI C API directly, such as printf(). An RTLinux POSIX API can be transformed into a ThreadX NONPOSIX API using specific transformation rules, e.g., from pthread_create() to tx_thread_create(). The ThreadX version of the previous code sample is given below, which is migrated by the OPTIMA approach.

```
mqid_t	MsgQueID;
void* ThreadProc(void* para)
{
#define MSGLEN	32
	char msgbuf[MSGLEN];
	int msglen;
	int prio;
```

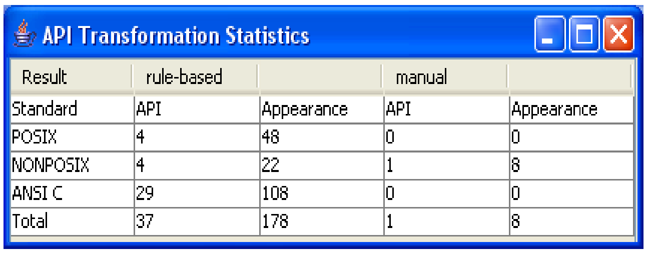
```
    msglen = tx_queue_receive(MsgQueID, msgbuf, MSGLEN);
    tx_thread_terminate(this);
    return 0;
}
void* ThreadProc2(void* para)
{
    int ret;
    ret = tx_queue_send(MsgQueID,"Start");
#if 0
    tx_thread_sleep(400);
#endif
   tx_thread_terminate(this);
    return 0;
}
void main()
{
    pthread_attr_t    attr;
    pthread_t          ThreadId1;
    pthread_t          ThreadId2;
    StartVos();
    tx_queue_open(&MsgQueID, "Thread1MessageQueue", 0, NULL);
    tx_thread_create (&ThreadId1, "thread 1", ThreadProc, 0, pointer,
    DEMO_STACK_SIZE, 1, 1, TX_NO_TIME_SLICE, TX_AUTO_START);
    tx_thread_priority_change (&ThreadId1, 9, NULL);
     tx_thread_create (&ThreadId1, "thread 1", ThreadProc2, 0, pointer,
    DEMO_STACK_SIZE, 1, 1, TX_NO_TIME_SLICE, TX_AUTO_START);
    tx_thread_priority_change (&ThreadId2, 9, NULL);
    tx_queue_delete(MsgQueID);

}
```

6.1.3.2 Metric for Software Migration

A software metric is defined to evaluate an OPTIMA transformation tool. The software source code of this use case contains 30 ".c" and ".h" files, in which 14 files call the POSIX system APIs, while 4 files call the NONPOSIX system APIs. In this case, 33 POSIX APIs are called 156 times while 5 NONPOSIX APIs are called 30 times. As well as 29 POSIX/ANSI C APIs, there are 4 POSIX RTOS APIs and 5 NONPOSIX APIs, are found in the proposed ontology. With the instructions of ontology-based transformation rules, the source application can be transformed to the target application.

As shown in Figure 6-1, 4 RTLinux POSIX APIs that appear 48 times in the application are successfully transformed based on transformation rules, also 4 RTLinux NONPOSIX APIs which appear 22 times in the application. 29 POSIX/ANSI C APIs can also be transformed under the rules, appearancing a total of 108 times. 1 RTLinux

NONPOSIX API with 8 appearances cannot be transformed, since there is no proper transformation rule. But this API can be transformed manually, with the support of knowledge acquired from the RTOS ontology repository. That is to say, 83% of the APIs can be transformed directly and automatically. 12% of the APIs need some human intervention and can be transformed semi-automatically. 5% of the APIs cannot be transformed semi-automatically, and have to be converted manually by maintainers. The experimental results are encouraging, show that software migration performed by OPTIMA is more efficient.



Result	rule-based		manual	
Standard	API	Appearance	API	Appearance
POSIX	4	48	0	0
NONPOSIX	4	22	1	8
ANSI C	29	108	0	0
Total	37	178	1	8

Figure 6-1 Metric for Software Migration

6.1.3.3 Discussion

Software migration is inherently knowledge intensive. It requires much domain knowledge including system knowledge as well as expertise and experience from specialists. Adding a knowledge dimension to the software migration approach would be a good way to facilitate the software migration process by making it more efficient and accurate.

An OPTIMA approach is proposed which will provide understandability, specification, reusability, knowledge acquisition and reliability for a software migration. Although about twenty percent of APIs still need to be transformed manually, the use case shows that the proposed approach can greatly facilitate software migration. The results of this use case show that the transformed source code can run correctly on a ThreadX platform. However, an OPTIMA approach will face following challenges:

- Although an OPTIMA approach is based on MDA concepts, the integration of MDA and ontology is still in the preliminary stage.

- More domain ontologies need to be designed to strengthen the proposed approach and to widen its range of use.
- The program transformation rules also need be completed.

6.2 Deploying Operating System Ontology to Support Portable Embedded Software Development

RTOSs are introduced to support embedded software. However, the general development environment of the embedded software is In-Circuit Emulator (ICE), which has following disadvantages:

- It is hard to parallel software and hardware development.
- It is hard to separate hardware and software errors during the development.
- ICE is very expensive and does not support multi-users.
- ICE is a proprietary system without full-featured testing and debugging tool support.

Software (re-)engineering researchers have therefore been looking for a solution to overcome these disadvantages. General speaking, supplying a general domain-specific pattern or architecture will provide software development with adaptability, reusability and line-product [19, 98]. As a result, it will reduce the costs and delays of development and maintenance. One of the common solutions is to develop the software on a general platform, such as Windows, and then port it to the specific RTOS with few changes. Software portability is thus one of the most important issues during embedded system development.

Virtual Real Time Operating System (VRTOS) can be classified as a middleware technique, which refers to the layer of interfaces and services that resides between the operating system and the application, aiming to facilitate the development, deployment and management of embedded software applications. From the developer’s point of view, VRTOS provides a unified RTOS development environment on one common platform (which might not be RTOS). i.e. VRTOS utilises the services offered by the underlying operating system to emulate RTOS services to embedded software. Different implementations of the VRTOS provide the same service interfaces so that platforms become transparent and embedded software becomes independent from the operating system. Thus, software applications that are developed on VRTOS are portable to different operating systems environments.

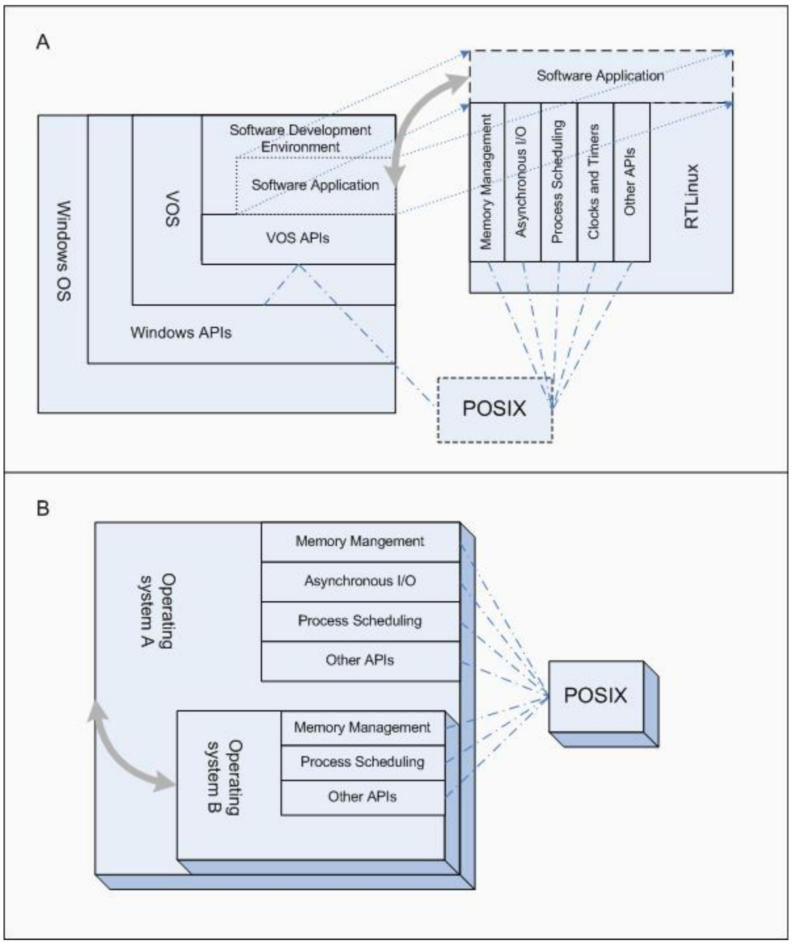


Figure 6-2 VRTOS Development and OS “Crop”

Figure 6-2A demonstrates basic scenarios of middleware technique that support portability. VRTOS plays a role as middleware which runs on Windows. VRTOS can support different RTOS platforms and embedded software developed on VRTOS can be ported to a target RTOS platform directly without too much change. Figure 6-2B will be discussed in Section 6.2.3.

6.2.1 Ontology-Based Portable Embedded System Development

The advantages of portable software have been recognised for many years. Portable software development is an inherently knowledge intensive activity, which requires a great deal of knowledge regarding the specifications of different platform environments. To manage such a large amount of knowledge, an ontology-based approach is therefore introduced. As a result, it will lessen the burden of collecting, classifying and processing information across different platforms for the developers. Section 6.2 discusses an ontology-based portable software development approach [18], focusing on the development of a VRTOS.

In this ontology-based approach, ontology will play a role as an RTOS domain knowledge base. Ontology can provide a vocabulary of terms and relationships to model specific domains, and it can facilitate the construction of the domain-specific solutions. The RTOS ontology plays a core role in the development of portable software applications for the following reasons:

- RTOS ontology provides semantic meaning for the RTOS functions and properties.
- RTOS ontology enables knowledge sharing and further knowledge analysis of different platforms.
- RTOS ontology defines a set of common system services which will be used as a standard for portable software development.
- RTOS ontology provides guidance for software porting via knowledge acquisition.

The following sections will discuss how to use RTOS ontology to guide the development of a VRTOS on the Windows platform. This process involves two stages, i.e. a VRTOS design stage and an implementation stage.

6.2.1.1 VRTOS Design Stage

When the VRTOS is being designed, system analysis will be performed based on knowledge acquisition. By accessing the RTOS ontology repository, the concepts, design policies and mechanisms of the RTOSs can be developed. Such information can be used to define programming interfaces for the VRTOS. Using the taxonomy of RTOS ontology, a set of common system services can be extracted as standard system services, which are implemented as platform independent entities in the VRTOS. On the other hand, the platform specific part could also be derived from the RTOS ontology to meet the requirements of the different domain. Through the knowledge retrieval service provided by the RTOS ontology repository, this process will become much easier and quicker.

Thread_api(?x) ∧ Thread_service(Thread_service_create) ∧ Windows(Windows_windowsNT) ∧ definedInOS(?x, Windows_windowsNT) ∧ provideService(?x, Thread_service_create) ∧ API_standard(?y) ∧ Data_type(?z) → hasAPI(Windows_windowsNT, ?x) ∧ hasAPIStandard(?x, ?y) ∧ hasReturnType(?x, ?z)

Thread_api(?x) ∧ Thread_service(Thread_service_create) ∧ Windows(ThreadX) ∧ definedInOS(?x, ThreadX) ∧ provideService(?x, Thread_service_create) ∧ API_standard(?y) ∧ Data_type(?z) → hasAPI(ThreadX, ?x) ∧ hasAPIStandard(?x, ?y) ∧ hasReturnType(?x, ?z)

Figure 6-3 Enquiries for Retrieval of System Service

Figure 6-3 shows that the RTOS ontology repository could be interrogated by system services using an ontology query language. For instance, the Windows NT system provides a system service to create a thread, the API for this service is CreatThread(), it is a NON POSIX, the return type is HANDLE, the parameter for this API is a pointer for the new thread. Furthermore, the ThreadX system provides the creating thread service as well and it is invoked by thread_create(). It is a NON POSIX, the return type is unsigned int, the parameter for it is also a pointer for the new thread.

$$\boxed{\begin{gathered} \mathit{Thread\_api(?x)} \wedge \mathit{Thread\_api(?y)} \wedge \mathit{Thread\_service(?z)} \\ \wedge \mathit{Windows(?a)} \wedge \mathit{Embedded\text{-}misc(?b)} \wedge \\ \mathit{definedInOS(?x, ?a)} \wedge \mathit{definedInOS(?y, ?b)} \wedge \\ \mathit{provideService(?x, ?z)} \wedge \mathit{provideService(?y, ?z)} \wedge \\ \mathit{Virtual\_OS(?d)} \wedge \mathit{Thread\_api(?c)} \wedge \mathit{definedInOS(?c, ?d)} \\ \rightarrow \mathit{Thread\_service(?z)} \wedge \mathit{provideService(?c, ?z)} \end{gathered}}$$

Figure 6-4 Enquiries for Retrieval of Similar Features

Figure 6-4 demonstrates an example of formulating similar threading functions for the two target operating systems.

$$\boxed{\begin{gathered} \mathit{Thread\_api(?x)} \wedge \mathit{Windows(Windows\_windowsNT)} \wedge \\ \mathit{definedInOS(?x, Windows\_windowsNT)} \wedge \mathit{hasAPI(?x, POSIX)} \\ \rightarrow \mathit{hasAPI(Windows\_windowsNT, ?x)} \\ \\ \mathit{Thread\_api(?x)} \wedge \mathit{OS(?y)} \wedge \mathit{definedInOS(?x, ?y)} \wedge \\ \mathit{hasAPI(?x, POSIX)} \rightarrow \mathit{hasAPI(?y, ?x)} \end{gathered}}$$

Figure 6-5 Enquiries for Windows POSIX

Figure 6-5 presents an enquiry concerning the thread APIs in a Windows NT system which supports the POSIX standard.

By defining SWRL based ontology enquiries, system analysis can be performed. It is assumed that the VRTOS will provide the following standard virtual system service which features:

- The VRTOS provides the application layer with a set of uniform system services to perform the tasks of threading and scheduling, making the different platforms transparent for the developers. In the thread programming part of the VRTOS, Windows thread APIs can be used to emulate POSIX thread programming. VRTOSs also provide different scheduling policies and priorities.
- Memory management is one of the crucial features in an application layer. Currently, memory allocation and free are available and memory usage tracking is provided as well.
- Message queue, mutex and semaphore services are developed as system independent components, which are not related to the system API on the target platform. In addition, a timer service is also provided for the application layer.

6.2.1.2 VRTOS Implementation Stage

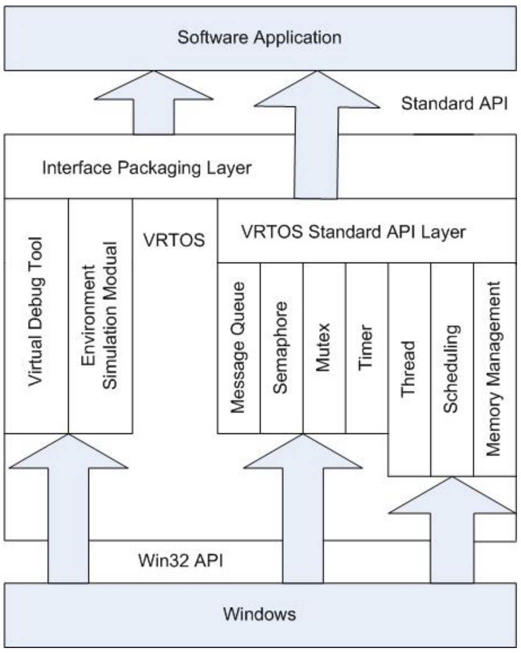


Figure 6-6 Architecture of a VRTOS on Windows Platform [118]

The fully implemented VRTOS is quite complex. At this time only the main processes of the development are presented to show that VRTOS development based on RTOS ontology is effective and time-saving. The architecture of the VRTOS on a Windows platform is shown in Figure 6-6. The VRTOS is implemented on Windows 2000 to provide a standard virtual system service to manage system resources such as memory, thread, mutex, semaphore, message queue, timer etc., and to provide debug and exception handling tools as well. The VRTOS provides a Kernel API Layer, which supports the real-time POSIX Standard [55]. An Interface Layer is designed that can be extended for different RTOSs, e.g. ThreadX [33] or RTLinux [100]. The VRTOS supports the pre-emptive schedule policy of the First-Come-First-Served (FCFS) style and simulates many kinds of system resources. A visual debug tool that enables external environment simulation greatly facilitates the debugging of embedded software [118].

6.2.2 Test Cases

A set of test cases has been used to verify the VRTOS development environment. In VRTOS there are 3 system simulation functions, 30 thread functions (10 of them are mutex related functions), 2 memory management functions, 7 message queue functions, 6 semaphore related functions and 9 timer related functions. Due to the classification of the system APIs, five sets of 30 testing cases are designed for testing the VRTOS, (i.e. message queue, memory management, mutex, semaphore and timer). Each case includes the use of threads and scheduling. The purpose of these test cases is to validate the correctness of the VRTOS API and to provide instructions for end users. The following is a sample of test cases that are used to check the developing environment of the VRTOS.

```
#include "vos.h"
pthread_mutex_t         MutexId;

void* ThreadProc(void* para)
{
      printf("Thread %u run:\n",(ULONG)para);
      printf("parameter is %u\n",para);

      printf("thread %u End Mutex unLock\n",(ULONG)para);
      pthread_mutex_unlock(&MutexId);
```

```
    printf("Thread %u finish.\n",(ULONG)para);

    pthread_cancel(pthread_self());
    return 0;
}


void* ThreadProc2(void* para)
{
    printf("Thread %u run:\n",(ULONG)para);
    printf("parameter is %u\n",para);
    printf("thread %d Begin Mutex Lock\n",(ULONG)para);
    if ( pthread_mutex_lock(&MutexId) == 0 )
    {
        printf("thread %d Mutex Lock success\n",(ULONG)para);
    }
    else
    {
        printf("thread %d Mutex Lock failure\n",(ULONG)para);
    }

#if 0
    printf("thread %u is in sleep\n",(ULONG)para);
    pthread_sleep(pthread_self(),100);
#endif
    pthread_mutex_unlock(&MutexId);
    printf("thread %u End Mutex unLock\n",(ULONG)para);
    printf("Thread %u finish.\n",(ULONG)para);
    pthread_cancel(pthread_self());
    return 0;
}

void main()
{
    pthread_attr_t    attr;
    pthread_t         ThreadId1;
    pthread_t         ThreadId2;
    pthread_mutexattr_t mattr;
    StartVos();
    pthread_mutexattr_init(&mattr);
    pthread_mutexattr_settype(&mattr,PTHREAD_MUTEX_ERRORCHECK);
    pthread_mutex_init(&MutexId,&mattr);
    pthread_mutexattr_destroy(&mattr);
    pthread_attr_init(&attr);
    pthread_attr_setschedpolicy(&attr,SCHED_FIFO);
    pthread_create(&ThreadId1,&attr,ThreadProc,(void *)1);
    pthread_setschedprio(ThreadId1,5);
    pthread_mutex_lock(&MutexId);
    pthread_create(&ThreadId2,&attr,ThreadProc2,(void *)2);
    pthread_setschedprio(ThreadId2,9);
    pthread_attr_destroy(&attr);
    Idle();
    pthread_mutex_destroy(&MutexId);
    StopVos();
}
```

This test case is trying to investigate the priority of threads. Two threads are created, A and B. Thread A's priority is lower than thread B. Thread B uses a mutex to stop itself. The mutex must be unlocked before thread A can stop or terminate. The expected result is that thread A (with lower priority) will only be activated when thread B (with higher priority) stops or terminates. The test results show that the case performed on the VRTOS is running properly and that the VRTOS design and development are successful.

6.2.3 Discussions

Section 6.2 proposes an ontology-based middleware approach to developing a VRTOS to enhance the portability of software applications in the context of embedded software development. The essence of a middleware based approach is standardisation, which is normally implemented by abstraction and isolation. Being middleware, the VRTOS has successfully isolated developing environments from their underlying operating system. Hence, the underlying operating system becomes totally transparent to the software applications. Through the RTOS ontology and knowledge representation techniques, the functional equivalence of different operating systems has been established by defining and implementing a set of common system services. These system services can be divided into two types: platform independent services and platform specific services. However, there is a balance to be struck between this standardisation and diversity. The VRTOS should only be applied to a small specific application domain, rather than a large range of operating system environments. Furthermore, the two differing costs incurred in this approach should be discussed, i.e., the cost of building an RTOS ontology and the cost of implementing a VRTOS. Building an RTOS ontology repository is undoubtedly a time-consuming endeavour, because a large amount of domain knowledge will be analysed and represented by ontology. But this ontology development cost is similar to the one spent for any other systems analysis in the program migration process. Different operating systems need to be studied and understood before porting applications to different platforms. In addition, RTOS ontology is reusable and expandable. The cost of implementing a VRTOS is incurred once for each different target operating system. The efforts of developing a VRTOS is great when compared to that required for porting a single program. However, it is small when compared to the cost of migrating an organisation's software. Personnel retraining costs should also be taken into account.

It is understood that building ontology for operating system is a huge project. Currently, the RTOS ontology is still at the prototype stage and mainly focuses on the programming interface. Another potential use is demonstrated by the Figure 6-2B (presented at the beginning of Section 6.2). It is a software reengineering scenario in which the operating system is "cropped" in order to fit a new hardware environment such as a limited resource device (e.g. a PDA). Because not all of the system call interfaces are needed in the limited resource device, only a subset of the APIs are needed and implemented. In this situation, the RTOS ontology repository manages the "crop" of the operating system in terms of API dependencies.

6.3 Deploying Data-Dominant Software System Ontology to Facilitate Program Comprehension

The proposed research has suggested that software systems should be represented on both program level (source code concepts) and its corresponding semantics level (domain concepts) to facilitate software reengineering. There exist several forms of software representation which enable programs to be understood, e.g., Abstract Syntax Tree (AST), control flow graph, data flow diagram, class diagram, etc. Which of these is selected in practice depends on the particular reverse engineering task. For example, if side effect analysis is required, then both data flow diagram and control flow graphs should be used. If an object-oriented design is required it will be necessary to employ a UML class diagram to describe the system components and their interrelationships. However, it is still hard to understand a program using only code level representations, e.g. class diagram, AST, etc., because the code is always organised around specific functions, rather than specific domain concepts. A UML class diagram is one of the most used software representations to aid reverse engineering projects. Deriving a class diagram from an arbitrary object-oriented program has the following disadvantages which will hinder or even defeat program understanding:

- Large software programs contain millions of lines of source code, which will generate hundreds of class diagrams. The size of the workforce dedicated to understanding these class diagrams is often huge. Software programmers and maintainers are plagued with information overload.
- Class diagrams can be automatically generated, but they may not be properly understood because of the different naming conventions or programming styles. As a result, it may be difficult to understand the functions or features of an object-oriented class in a class diagram.
- Extracting a class diagram only transforms source code from one representation form to another at the same abstraction level. Hence there will be a lack of context and explicit domain knowledge.
- A class diagram is not computable, that is to say, it will not support any inference or knowledge acquisition in the context of knowledge representation. Thus, it is not possible to glean implicit information by manipulating class diagrams.

To address these problems, knowledge features other than code will need to be integrated into the software representations. For instance, domain experts could hold an alternative view which deserves to be integrated into the code based comprehension. This section proposes a novel approach to constructing an ontological perspective for a software system [131, 133]. Ontology will help software maintainers improve understanding of the application and it's role in the problem domain.

6.3.1 Developing Application Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology

Most program understanding processes are cognitive and manually performed by software maintainers, who typically solve the problems by realising "plans" in the source code. Research was carried out by AT&T to analyse the time software maintainers spent on the different categories of reengineering tasks [99]. In their research, it was discovered that the maintainers had dedicated up to 60% of their time performing searches, i.e. looking for the relevant concepts based on the complicated interrelationships in the source code. Another study was performed by MCC, which argues that software maintainers need to understand the domain before performing any software reengineering tasks [24]. Section 6.3 proposes an ontology-based program comprehension approach, which derives an ontological perspective for software systems. This ontology is a combination of two others: domain ontology and code ontology. This framework has been discussed in Section 3.2.6. Code ontology is semi-automatically generated by the bottom-up approach discussed in Chapter 4. Domain ontology is obtained from an online ontology repository with some modifications based on software documentation and the expertise of domain professionals. The proposed approach will be demonstrated in the next section, along with a selected use case.

6.3.2 Use case – Point of Sale Terminal (POST)

The example software program used in this section was taken from an object-oriented design text book [68]. A slight modification was made to make the code more suitable for validating the proposed approach. It is a point-of-sale terminal (POST) system. A point-of-sale terminal is a computer system used to record sales and handle payments, and it is typically used in retail stores. It includes hardware components such as a computer and a bar code scanner, and software to run the system. The Java source code for a POST system is given and the proposed ontology-based software comprehension approach is employed to understand this code.

6.3.2.1 POST Domain Ontology

Domain ontology represents the knowledge of specific application domains, e.g., banking, retailing, and human resource management, etc. It focuses on domain concepts, rather than software elements. The fundamental components are concepts, object properties of concepts (represented as binary relationships), datatype properties of concepts (represented as unary relationships) and instances of concepts. It will be used to facilitate the communication between interested parties by clarifying the important concepts and how they are related. Domain ontology describes the things in the real world. Hence, the following elements are not suitable for domain ontology: software artefacts (detailed software design) and responsibilities or methods (in term of Class Responsibility Collaborator Card). Identifying concepts is crucial for creating meaningful domain ontology. Finding concepts for domain ontology can be carried out by the following two methods: concept category list and noun phrase identification. Furthermore, domain ontology can be modelled with description logic, and the reasoning and explanation facilities provided by DL supports the validating of the domain ontology.

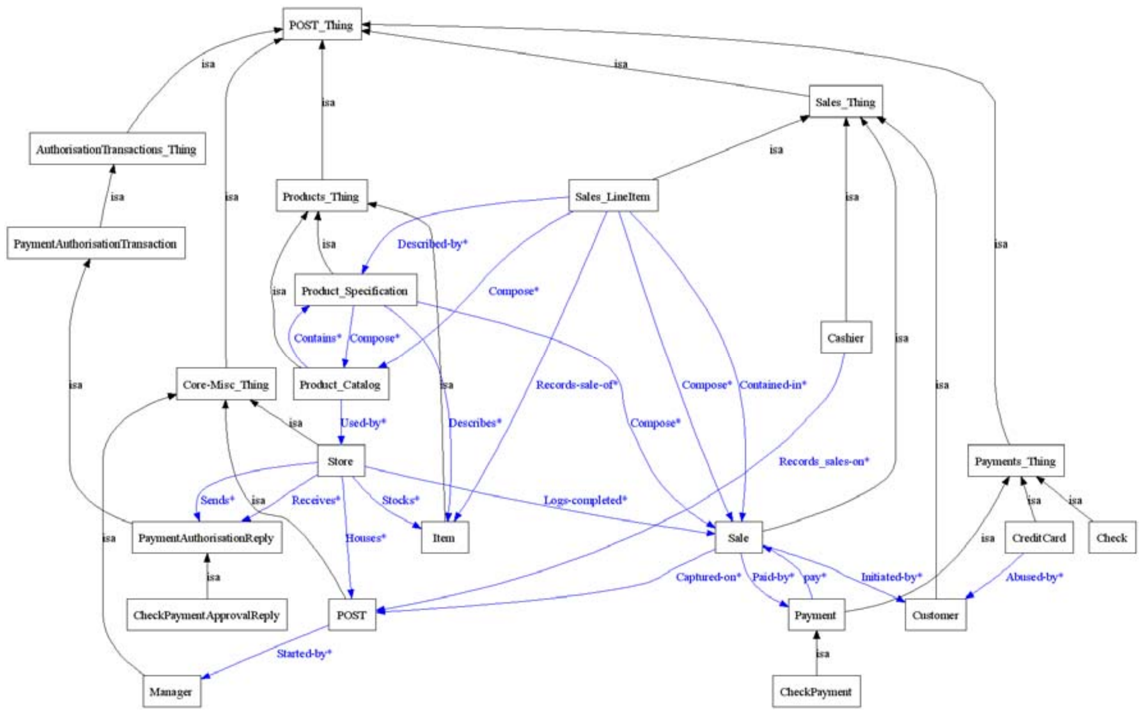


Figure 6-7 Domain Ontology for POST System

In this use case, the domain ontology should represent meaningful (to the domain expert and software maintainer) concepts in the POST system domain. Figure 6-7 depicts a small part of the domain ontology for the POST system. Through domain ontology, domain knowledge can be represented in DL, such as:

$$\text{Sales\_LineItem} \equiv \text{Order}$$
$$\cap \text{ Describedby.Product\_Specification}$$

∩ Records-sale-of.Item ∩ Contained-in.Sale
∩ Compose.Sale

This POST system domain ontology and its representation in DL allowing a software maintainer to know what a sales line item is in the context of the retail POST system.

6.3.2.2 Class Diagram Extraction

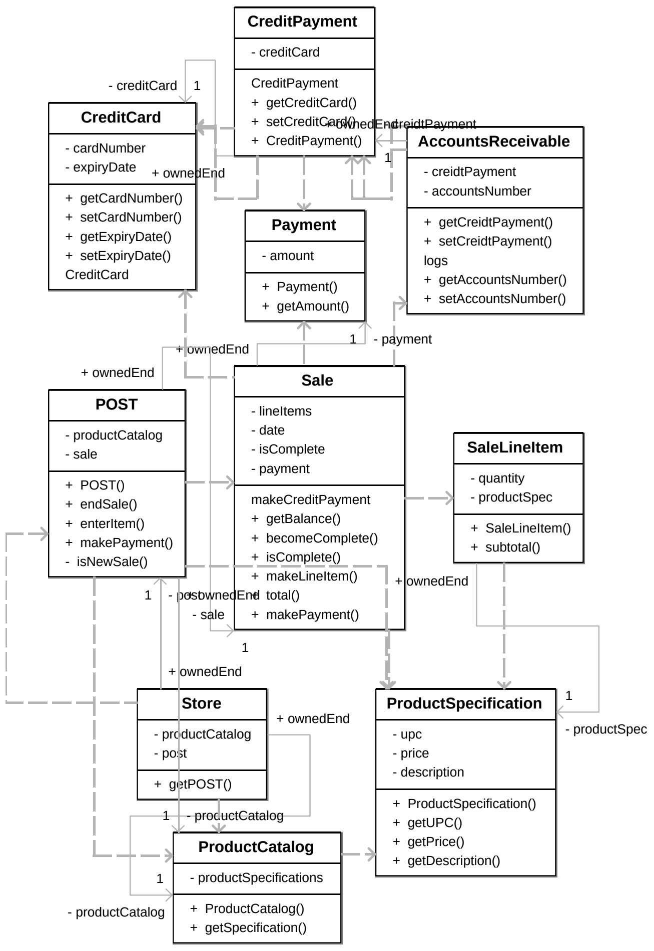


Figure 6-8 Extracted Class Diagram for POST

Figure 6-8 demonstrates part of the extracted class diagram from the POST system implementation code. To fit the scenario of the proposed approach, some modifications have been made to the source code. Through this class diagram, object-oriented class can be represented with its attributes and operations in a very competent way. However, if the class name is not clear, it will be very hard to understand the function of that class, and it will hinder the program comprehension process. Moreover, the associations and dependencies can be detected semi-automatically by analysing the associated attributes, variables and their life cycles. They cannot, however, be associated to any particular type of relationship. Thus, reforming the code representation is not enough for program comprehension without introducing a new knowledge scope.

6.3.2.3 Populating Class Diagram Ontology

Based on the bottom-up ontology generation approach discussed in Chapter 4, the POST system ontology can be semi-automatically created. Figure 6-9 presents a part of the POST system ontology.

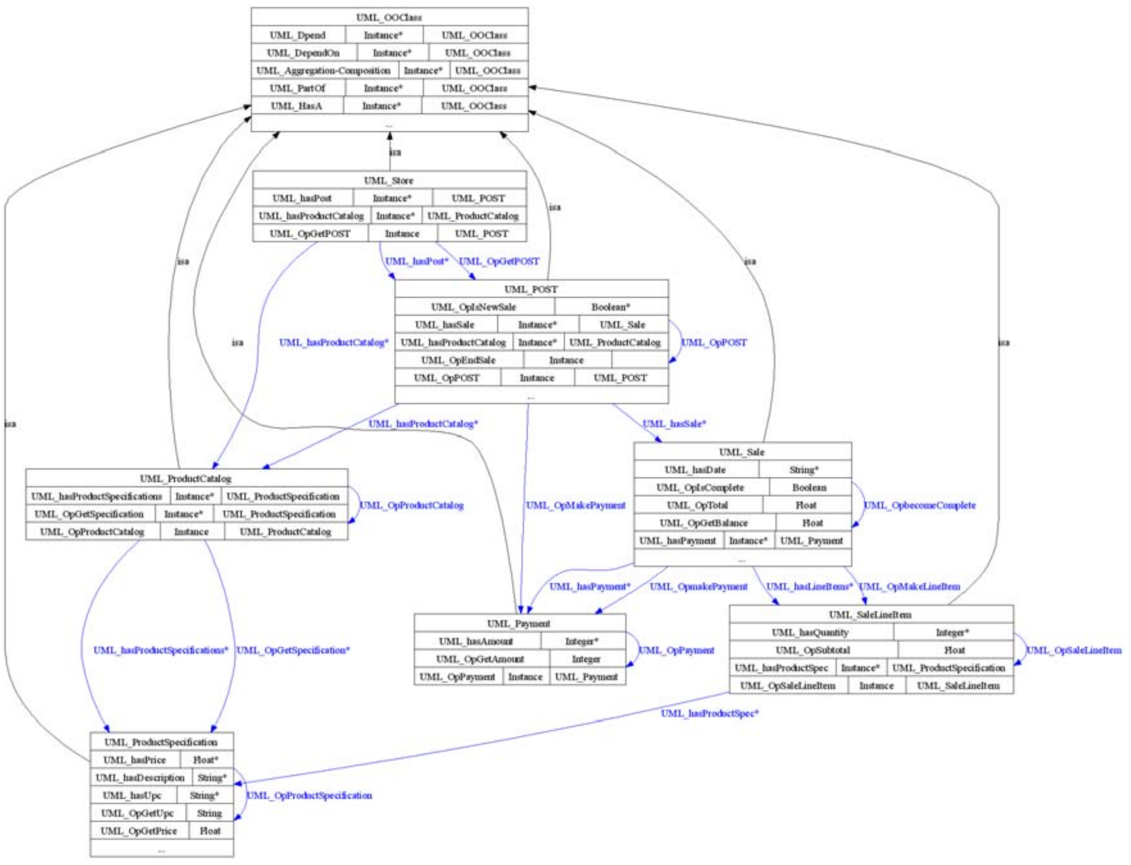

UML_OOClass
UML_Dpend | Instance* | UML_OOClass
UML_DependOn | Instance* | UML_OOClass
UML_Aggregation-Composition | Instance* | UML_OOClass
UML_PartOf | Instance* | UML_OOClass
UML_HasA | Instance* | UML_OOClass
isa
UML_Store
UML_hasPost | Instance* | UML_POST
UML_hasProductCatalog | Instance* | UML_ProductCatalog
UML_OpGetPOST | Instance | UML_POST
UML_hasPost*
UML_OpGetPOST
UML_POST
UML_OpIsNewSale | Boolean*
UML_hasSale | Instance* | UML_Sale
UML_hasProductCatalog | Instance* | UML_ProductCatalog
UML_OpEndSale | Instance
UML_OpPOST | Instance | UML_POST
UML_OpPOST
UML_hasProductCatalog*
UML_hasSale*
UML_OpMakePayment
UML_ProductCatalog
UML_hasProductSpecifications | Instance* | UML_ProductSpecification
UML_OpGetSpecification | Instance* | UML_ProductSpecification
UML_OpProductCatalog | Instance | UML_ProductCatalog
UML_OpProductCatalog
UML_Sale
UML_hasDate | String*
UML_OpIsComplete | Boolean
UML_OpTotal | Float
UML_OpGetBalance | Float
UML_hasPayment | Instance* | UML_Payment
UML_OpbecomeComplete
UML_hasPayment*
UML_OpmakePayment
UML_hasLineItems*
UML_OpMakeLineItem
UML_Payment
UML_hasAmount | Integer*
UML_OpGetAmount | Integer
UML_OpPayment | Instance | UML_Payment
UML_OpPayment
UML_SaleLineItem
UML_hasQuantity | Integer*
UML_OpSubtotal | Float
UML_hasProductSpec | Instance* | UML_ProductSpecification
UML_OpSaleLineItem | Instance | UML_SaleLineItem
UML_OpSaleLineItem
UML_hasProductSpecifications*
UML_OpGetSpecification*
UML_hasProductSpec*
UML_ProductSpecification
UML_hasPrice | Float*
UML_hasDescription | String*
UML_hasUpc | String*
UML_OpGetUpc | String
UML_OpGetPrice | Float
UML_OpProductSpecification

Figure 6-9 Populated Class Diagram Ontology

With this POST system ontology, the concepts in a UML class diagram could be understood and represented by DL. For instance,

$$
\begin{aligned}
\text{UML\_POST} \equiv\ & \text{UML\_OOClass} \\
& \cap \forall\ \text{associateto.UML\_Store}
\end{aligned}
$$

$\cap \forall$ associateto.UML_ProductCatalog
$\cap \forall$ associateto.UML_Sale
$\cap \forall$ dependon.UML_ProductSpecification

Along with the POST domain ontology, the POST system ontology will be able to provide software maintainers with a more understandable view of the POST system. Moreover, the knowledge acquisition techniques will improve the efficiency.

6.3.2.4 Understanding a POST System by Application Specific Ontology

A. Comprehension by Integrating System Ontology and Domain Ontology

After applying the DL based ontology mapping algorithm discussed in Chapter 5, semantic relationships have been detected between the concepts across the POST system ontology and the POST domain ontology. For instance, a subsumption relationship has been detected between the concept UML_POST in the system ontology and the concept POST_Terminal in the domain ontology, i.e. POST_Terminal $\subseteq$ UML_POST. This can be interpreted by saying that the Java class POST is a more abstract code level concept simulating and implementing the functionalities of the real domain object POST in the context of retail activity. In addition, it is easy for the software maintainers to understand what a POST terminal is and what it will do by referring to domain ontology. Therefore, it will assist software maintainers to understand the java class by providing a corresponding application context, which is missing in a traditional cognition based program understanding.

B. Comprehension by Reifying Association/Dependency

Once the mapping has been built between the concepts across the system ontology and the domain ontology, the meaningful concrete relationships stored in the domain ontology will be matched to the simple relationships (mainly, association and dependency) stored in the software system ontology. In this example from a class diagram ontology, the association between the concept UML_POST and UML_Sale can be reified as UML_Sale is captured on UML_POST after being matched to the domain ontology. This information will help maintainers to understand the POST system more easily.

C. Detection of Design Defects

It is noticed that the concepts in domain ontology may not have a direct relationship with each other; in class diagram they do have direct relationships. This can be considered as the differences between the abstract design level and the implementation level. Some of these differences are necessary for program implementation, whilst some are not. By analysing the differences between the two ontolgies, some of the design defects can be discovered. In this example, the concepts POST and Product_Catalog are not directly related in domain ontology, but the concepts UML_POST and UML_ProductCatalog have a binary relationship in the class diagram ontology. From the object-oriented point of viewpoint, loose coupling is advocated. As a result, class UML_Store should be introduced here as an association class to connect the concepts UML_POST and UML_ProductCatalog. Consequently, the direct relation between the concepts UML_POST and UML_ProductCatalog can be removed, and these two components will be implemented in a loose coupling manner after being reverse engineered.

6.3.2.5 Use case Analysis

A software metric has been used to validate this proposed approach. Table 6-1 illustrates the software metrics of the PSOT system comprehension process. The sample code is written in Java, with 205 lines of code. According to the transformation rules of class diagram to ontology 16 concepts and 18 properties have been derived from the class diagram which is generated automatically by Eclipse UML plug-ins. In POST system domain ontology, there are 53 concepts and 31 properties. Following the DL based ontology mapping algorithm 12 concepts and 15 properties can be matched in these two ontologies. It is noticed that several concepts in the class diagram still cannot be matched to the concepts in the domain ontology due to the difference between the problem domain and its implementation. However, the result is still encouraging. Combining domain ontology and code ontology to develop application specific ontology will improve the efficiency of the program comprehension process.

Metric Element	
Line of Code	205
Concepts in Domain Ontology	53
Concepts in Code Ontology	16
Properties in Domain Ontology	31
Properties in Code Ontology	18
Concepts being matched	12
Relations being matched	15

Table 6-1 Metric for Software System Ontology

6.3.3 Discussions

Section 6.3 proposes a program comprehension approach by developing application specific ontology. Most traditional program understanding approaches use an AST, a data flow diagram, a control flow graph, or a UML class diagram to assist the software maintainer understand and analyse software system. However, these kinds of software representation forms are just transforming source code into another form within the same abstraction level. With respect to the knowledge intensive features of both the software system and the program understanding process, the proposed approach uses ontology to represent both the software system and the problem domain. This introduces domain knowledge into the program comprehension process and bridges the gap between the two different levels. It will enhance the understandability of the software system by integrating the two ontologies. It is apparent that the system ontology is very simple, containing perhaps less than fifty concepts. The domain ontology is more complicated, containing perhaps more than two hundred concepts. This can be explained by the fact that the software system is an abstract model built to simulate and resolve real world problems.

6.4 Deployment of Data-Dominant Software System Ontology in Software Modularisation

Cloud computing is one of the future trends of software engineering research. This implies a service-oriented architecture (SOA) providing more flexible and economic usage for software end users. The research question in cloud computing for software reengineering will be how to decompose the legacy system into potential service candidates that will fit into a cloud computing environment. In particular, how to detect and understand the loosely coupled reusable components of a legacy system and then to make these components work as services in the cloud. Section 6.4 is going to explore the first part of this question, i.e., understanding legacy software and decomposing it into potential service candidates that could meet the requirements of cloud computing. More specifically, this section proposes an ontology-based approach to reengineering an enterprise software system for the cloud computing environment [130].

Enterprise software is a suite of programs that is intended to solve an enterprise problem and is always complex and large-scale. It performs a set of functions or processes to meet the general requirements of many different organisations and industries. Generally speaking, most enterprise software will have the following features: a.) it uses object-oriented design patterns, b.) it uses relational database management systems for data storage, c.) using reusable libraries/application frameworks, d.) it hides implementations in the back of well-defined interfaces, and e.) it provides commonly needed functions and services, e.g., ERP, CRM and so on.

As described in Figure 3-5, the proposed approach is to utilise ontology and it's related techniques to explore the concepts and relationships in enterprise software and thereby to enhance enterprise software comprehension. Enterprise software ontology is developed by integrating three others: code ontology, database ontology and Hibernate ORM framework ontology. By analysing and modularising strongly related concepts in enterprise software ontology, decomposing the legacy software into loosely coupled modules that are considered to be potential service candidates will be enabled in a cloud computing environment.

6.4.1 Partitioning Ontology to Identify Potential Service Candidates for Cloud Computing

The goal of developing enterprise software ontology is to achieve a more comprehensive representation form of software system and then to deploy this new form in software reengineering projects. This will identify the potential service candidates of a legacy system for the cloud computing environment. The identification of cloud computing service candidates requires a decomposition of the software system with respect to the following two principles; loosely coupled components and reusable functionalities. Therefore the components and their interrelationships in the software system need to be analysed, and the strongly related components need to stay together while loosely coupled ones do not. In this research, the decomposition of enterprise software is accomplished by modularising enterprise software ontology, also known as ontology partitioning.

Currently, there are a couple of studies on ontology partitioning [73, 102]. This paper will adopt the structure-based partitioning algorithm proposed by Schlicht and Stuckenschmidt [102]. Their partitioning algorithm is based on the structural dependencies between concepts in ontology, and is represented through a weighted dependency graph. The strength of the dependencies between the concepts is then calculated and the proportional strength network is obtained to detect sets of strongly related concepts. As a result, the concepts which are more strongly related will be modularised and the original ontology will be divided into loosely coupled partitions.

After applying a structure-based partitioning algorithm, the enterprise software ontology can be decomposed into a few modules. For each module, all the concepts are strongly related, and also organised around specific functions and domain concepts. Moreover, all the modules are loosely coupled as well. Thus, they can be considered to be potential service candidates in relation to a cloud computing environment.

6.4.2 Use case – PLAZMA BUSINESS SOLUTION SYSTEM

6.4.2.1 Plazma Business Solution System

The Plazma [44] business solution system is an open-source ERP+CRM application for middle business. It contains seven enterprise functionalities including accounts management, contacts management, sales management, tasks management, campaigns management, products management and analytical reports. The database server supports Oracle, PostgreSQL, MySQL, Firebird and HSQL. It is also compatible with Windows, Linux and MacOS. There is no doubt that it is a very powerful and robust business solution system, but the end user will need more IT resources to support it, indicating that it could become a burden for the end user to some extent. However, all the above features also show that this enterprise software could be a potential cloud computing application in terms of scalability and flexibility. The proposed approach is applied to reengineer the Plazma business software for cloud computing.

6.4.2.2 Ontology Generation

Topcased [34] is used to produce a class diagram. It supports reversing Java from both a project and a JAR. A UML file containing all the classes and associations has been generated by this reverse engineering tool. With the bottom-up ontology generation approach discussed in Chapter 4, this UML file has been transformed to an OWL file containing concepts and relationships. 575 classes have been derived from the source code and transformed into ontology concepts. 684 associations have been extracted and restored as relationships in ontology. The following is a sample of the Plazma system ontology. For lack of space, some elements have been removed and replaced with "...".

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns="http://www.owl-ontologies.com/Ontology1274641707.owl#"
    ... ...>
  <owl:Ontology rdf:about=""/>
  ... ...
  <owl:Class rdf:ID="BusinessableElement"/>
  <owl:Class rdf:ID="ContactableElement"/>
  <owl:Class rdf:ID="Employee"/>
  <owl:Class rdf:ID="PayrollForm"/>
  <owl:ObjectProperty rdf:ID="contactableElement">
```

```
    <rdfs:domain>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#ContactableElement"/>
          <owl:Class rdf:about="#Employee"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:domain>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="payrollForm">
    <rdfs:domain>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#PayrollForm"/>
          <owl:Class rdf:about="#Employee"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:domain>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="businessableElement">
    <rdfs:domain>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#BusinessableElement"/>
          <owl:Class rdf:about="#Employee"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:domain>
  </owl:ObjectProperty>
  <owl:DatatypeProperty rdf:ID="employeeAccount">
    <rdfs:domain rdf:resource="#Employee"/>
    <rdfs:range rdf:resource="..."/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="tax">
    <rdfs:range rdf:resource="..."/>
    <rdfs:domain rdf:resource="#Employee"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="startDate">
    <rdfs:domain rdf:resource="#Employee"/>
    <rdfs:range rdf:resource="..."/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="CLASS_ID">
    <rdfs:range rdf:resource="..."/>
    <rdfs:domain rdf:resource="#Employee"/>
  </owl:DatatypeProperty>
</rdf:RDF>
```

MySQL Structure Magic [95] is used to generate an XML database schema description. After applying the bottom-up ontology generation approach discussed in Chapter 4, this XML file has been transformed into an OWL file storing database ontology. 644 concepts have been extracted from the Plazma database and saved into an OWL file. The following is a sample of the Plazma database ontology. For lack of space, some elements have been removed and replaced with "...".

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    ...>
  <owl:Ontology rdf:about=""/>
<owl:Class rdf:about=".../plazma#employee">
    <db:hasForeignKeys>
      <db:ForeignKey rdf:about="...">
        <rdfs:label rdf:datatype="...">...
         employee_category.ID</rdfs:label>
        <db:hasRefFieldProperty>
          <owl:FunctionalProperty
           rdf:about=".../plazma#employee_category.ID"/>
        </db:hasRefFieldProperty>
        <db:hasLocFieldProperty>
          <owl:FunctionalProperty
           rdf:about=".../plazma#employee.EMPLOYEE_CATEGORY_ID"/>
        </db:hasLocFieldProperty>
        <db:hasLocalField rdf:datatype="..."
        >EMPLOYEE_CATEGORY_ID</db:hasLocalField>
        <db:hasLocTableClass rdf:resource=".../plazma#employee"/>
        <db:hasReferenceTable rdf:datatype="..."
        >employee_category</db:hasReferenceTable>
        <db:hasRefTableClass
         rdf:resource=".../plazma#employee_category"/>
        <db:hasFKName rdf:datatype="..."
        >fk_EMPLOYEE_CATEGORY_ID_employee_category_ID</db:hasFKName>
        <db:hasReferenceField rdf:datatype="..."
        >ID</db:hasReferenceField>
      </db:ForeignKey>
    </db:hasForeignKeys>
    <db:hasForeignKeys>
      <db:ForeignKey rdf:about="...">
        <db:hasLocFieldProperty>
          <owl:FunctionalProperty
           rdf:about=".../plazma#employee.PERSON_ID"/>
        </db:hasLocFieldProperty>
        <db:hasRefFieldProperty>
          <owl:FunctionalProperty rdf:about=".../plazma#person.ID"/>
        </db:hasRefFieldProperty>
        <rdfs:label rdf:datatype="..."
        >FK: employee.PERSON_ID ---&gt; person.ID</rdfs:label>
        <db:hasLocTableClass rdf:resource=".../plazma#employee"/>
        <db:hasRefTableClass>
          <owl:Class rdf:about=".../plazma#person"/>
        </db:hasRefTableClass>
        <db:hasFKName rdf:datatype="..."
        >fk_PERSON_ID_person_ID</db:hasFKName>
        <db:hasReferenceField rdf:datatype="..."
        >ID</db:hasReferenceField>
        <db:hasLocalField rdf:datatype="..."
        >PERSON_ID</db:hasLocalField>
        <db:hasReferenceTable rdf:datatype="..."
        >person</db:hasReferenceTable>
      </db:ForeignKey>
    </db:hasForeignKeys>
    <db:isBridgeTable rdf:datatype="...">false</db:isBridgeTable>
    ... ...
  </owl:Class>
... ...
</rdf:RDF>
```

Hibernate ORM Framework ontology is created based on the Hibernate mapping files. The bottom-up ontology generation approach discussed in Chapter 4 has been applied to this XML configuration file. 452 concepts have been extracted from model transformations, in which 226 concepts are mapping with code ontology concepts and 226 concepts are mapping with database ontology concepts. The following is a sample of the Hibernate configuration file ontology. For lack of space, some elements have been removed and replaced with "...".

```
<?xml version="1.0"?>
<rdf:RDF
    ...>
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="UML_PersonHeader">
    <owl:disjointWith>
      <owl:Class rdf:ID="Column_PERSON_ID"/>
    </owl:disjointWith>
  </owl:Class>
  ... ...
  <owl:Class rdf:ID="UML_Department"/>
  <owl:Class rdf:about="#UML_EmployeeRank">
    <owl:disjointWith rdf:resource="#Column_EMPLOYEE_RANK_ID"/>
  </owl:Class>
  <owl:Class rdf:ID="UML_Employee">
    <owl:disjointWith>
      <owl:Class rdf:ID="Table_EMPLOYEE"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:ID="Column_EMPLOYEE_CATEGORY_ID">
    <owl:disjointWith>
      <owl:Class rdf:ID="UML_EmployeeCategory"/>
    </owl:disjointWith>
  </owl:Class>
  ... ...
  <owl:ObjectProperty rdf:ID="map_to">
    <rdfs:domain rdf:resource="#UML_Employee"/>
    <rdfs:range rdf:resource="#Table_EMPLOYEE"/>
    <rdf:type rdf:resource="..."/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="many_to_one">
    <rdf:type rdf:resource="..."/>
    <rdfs:range rdf:resource="#Column_EMPLOYEE_CATEGORY_ID"/>
    <rdfs:domain rdf:resource="#Column_EMPLOYEE_CATEGORY_ID"/>
  </owl:ObjectProperty>
</rdf:RDF>
```

6.4.2.3 Plazma Enterprise Ontology Integration

Building on the DL based ontology mapping algorithm discussed in Chapter 5, the final comprehensive enterprise software ontology is generated by integrating code ontology, database ontology and Hibernate ORM Framework ontology. 870 concepts and 1215 relationships are observed in this final ontology for Plazma business solution systems. This will provide more knowledge perspectives for program comprehension by integrating knowledge regarding code, data and application framework.

6.4.2.4 Identification of Potential Service Candidates for Cloud Computing

After applying a structure-based partitioning algorithm to the Plazma enterprise software ontology, 6 partitions have been initially obtained, namely, finance partition, sale and purchase partition, product and inventory partition, project management partition, human resources and payroll partition, contacts management partition. The finance partition has 165 concepts including accounting, banking, cash, payment, tax, currency, etc. Sale and purchase partition has 145 concepts including sale order, sale plan, sale invoice, purchase order, purchase invoice, etc. Product and inventory partition has 133 concepts including inventory move, inventory writeoff, inventory outcome, product info, product price, product stock, manufacture, goods, etc. Project management partition has 98 concepts including task, task status, task priority, task type, etc. Human resources and payroll partition has 182 concepts including organisation, personality, employee, store, warehouse, person job, education type, employee rank, etc. Contacts management partition has 147 concepts including email, phone, address, partner, partner group, industry, etc. Consequently, 6 potential service candidates for cloud computing environment are obtained. These are financial and accounting service, sale and purchase management service, product and inventory management service, project management service, human resources and payroll service and contacts management service. Hence, the Plazma business solution can be reengineered for cloud computing by providing these 6 services, which will still meet the requirements of the end user, but require less IT resources to support it. That is the goal of cloud computing – flexibility, scalability and economy.

6.4.3 Discussions

This section proposes an ontology-based approach for reengineering enterprise software for cloud computing, i.e. to identify potential service candidates from the legacy system. The following is a five-step ontology development process. The enterprise software ontology is created by generating and integrating code ontology, database ontology and Hibernate framework ontology. The deployment is performed by analysing and modularising strongly related concepts in the enterprise software ontology. It will facilitate the understanding and decomposing of the legacy software into loosely coupled modules that are considered to be potential service candidates in a cloud computing environment. The use case is conducted on an open-source ERP+CRM system, and shows that the proposed approach in terms of semi-automation for large-scale software systems is an efficient reengineering methodology. 6 potential service candidates are successfully identified from the legacy software, which will then be reused in cloud computing. However, this approach is not fully automatic, human intervention is still needed.

6.5 Summary

This chapter explores the final parts of a knowledge based software reengineering framework and the deployment (the potential usage) of software system ontology in a few software reengineering tasks. Most of these reengineering tasks employ cognition based approaches, which require a great deal of manual effort of software maintainers. This is time consuming and error prone. Previous chapters have discussed the benefits and methodologies of representing a software system with ontology. Consequently, ontology based approaches are proposed to simulate the cognition processes of software maintainers, thus evading some of the manual tasks. Hopefully, traditional software reengineering can be improved by introducing knowledge based approaches and knowledge dimensions.

- Software migration is inherently knowledge intensive, and requires a large amount of domain knowledge, system knowledge and expertise as well as experience from specialists. Adding knowledge dimensions to a software migration approach will facilitate the software migration process by making it more efficient and accurate. An OPTIMA approach is proposed to provide understandability, specification, reusability, knowledge acquisition and reliability for software migration.

- An ontology-based middleware approach has been proposed to develop a VRTOS to enhance the portability of software applications in the context of embedded software development. As middleware, the VRTOS has successfully isolated developing environments from their underlying operating system. Thus, the system becomes totally transparent to the software applications. Through the RTOS ontology repository and knowledge representation techniques, the functional equivalence of different operating systems has been established by defining and implementing a set of common system services provided by the VRTOS.

- With respect to knowledge intensive features of both the software system and the cognitive theory based program understanding process, an ontology based program comprehension approach is proposed by generating and integrating two different ontologies, i.e., software system ontology and domain ontology. This approach introduces domain knowledge into the program comprehension process and bridges the gap between the two different levels.

- Cloud computing is one of the future trends of software engineering research. An ontology-based approach has been explored to reengineer enterprise software for cloud computing, i.e. to identify potential service candidates from the legacy system. The basic idea of this approach is to decompose legacy systems by analysing and modularising strongly related concepts in enterprise software ontology. As a result, it will permit the understanding of legacy software and decompose it into loosely coupled modules that can be considered to be potential service candidates in a cloud computing environment.

Chapter 7 Tools Support

Objectives

- To describe the architecture of the prototype tools
- To illustrate each tool for the proposed research

Having introduced both the theoretical and technical aspects of the proposed knowledge based software reengineering approaches in the previous chapters, this describes how those approaches are implemented and verified by the software reengineering tools. Section 7.1 will present the prototype of an OPTIMA migration tool which is implemented to validate the ontology based platform specific software migration approach (Section 6.1). Section 7.2 will present a prototype tools suite which supports bottom-up software system ontology generation and DL based ontology integration processes. It is implemented mainly to validate the ontology based software reengineering approaches discussed in Section 6.3 and Section 6.4. Automation is the ultimate goal of the proposed research and toolset. However, it is understood that human intervention is almost inevitable in those cognition based tools since AI will never completely overtake human intelligence. Hence, these prototype software reengineering tools will only be able to semi-automate the reengineering process.

7.1 OPTIMA Miagration Tool

To support the OPTIMA approach discussed in Section 6.1, a knowledge-based platform specific software migration tool is designed and a prototype of this tool has been implemented to validate the OPTIMA approach. The following sections introduce the details of this prototype tool, including the architecture, design, implementation, functionalities and graphical user interface (GUI).

7.1.1 Architecture of OPTIMA Toolkit

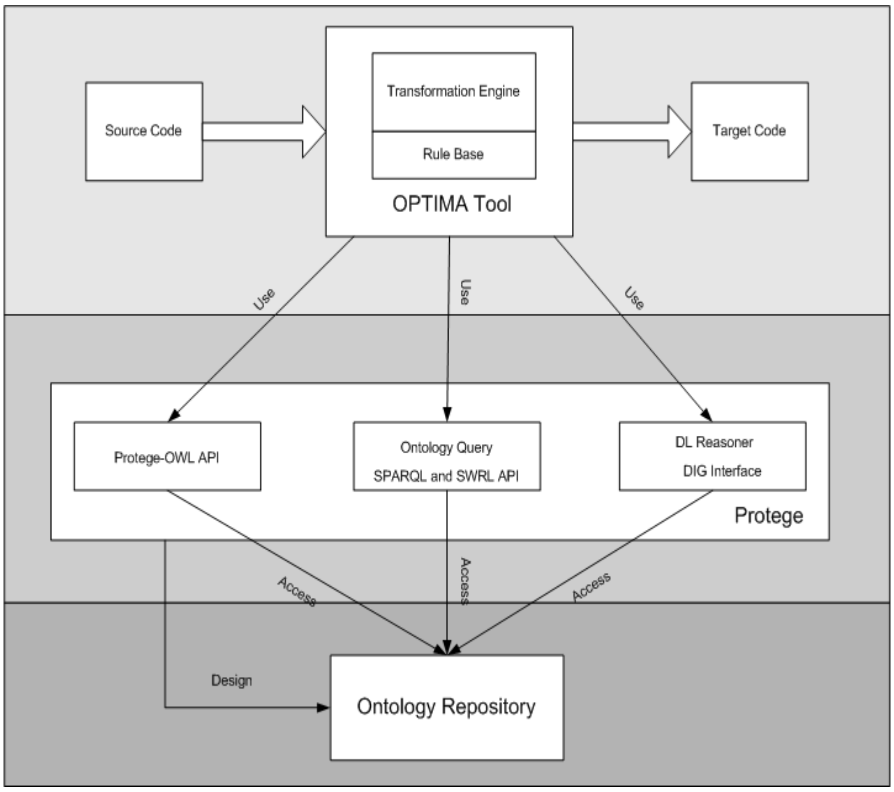
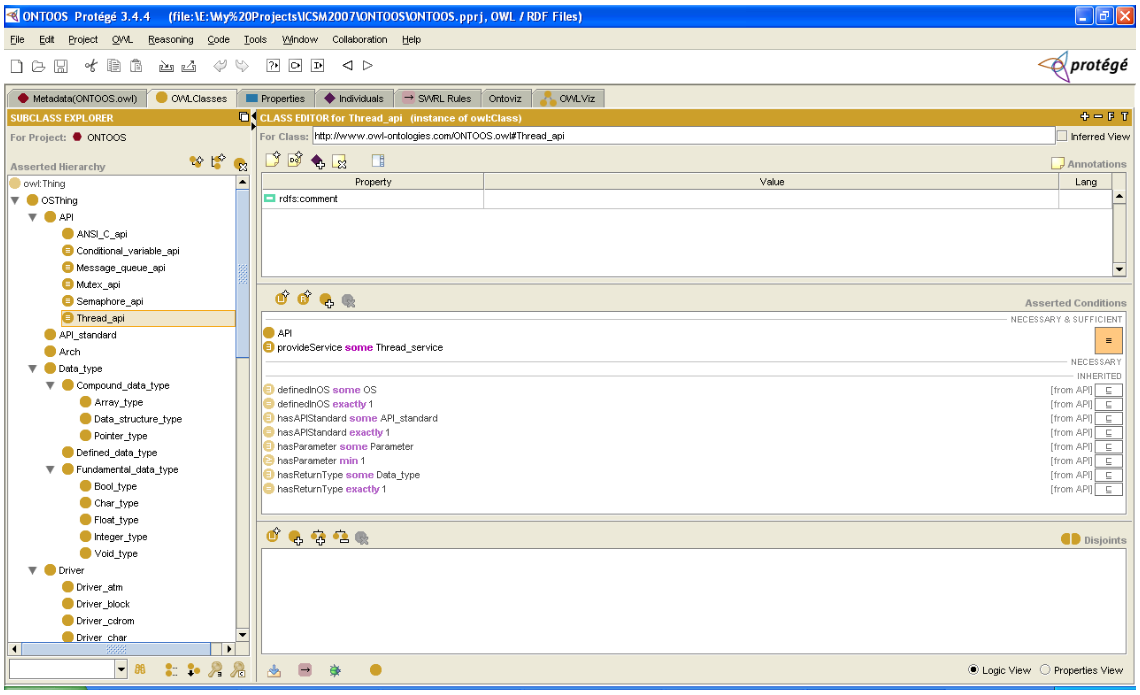


Figure 7-1 Architecture of OPTIMA Toolkit

Figure 7-1 demonstrates the architecture of the OPTIMA toolkit, which can be divided into three layers: software migration layer, ontology accessing and processing layer and ontology repository layer.

7.1.2 Ontology Repository Layer

The ontology repository layer stores the operating system ontology that is developed based on the top-down ontology generation approach discussed in Section 4.2. Following the software system ontology development process defined in Section 3.2.2 and the eight operating system ontology development rules presented in Section 4.2.2, the operating system ontology repository is built up step by step. This ontology is developed by the Protege ontology editor [105] and stored as an ontological repository, which plays a core role in the OPTIMA migration tool. OWL-DL [101] is used as the ontology language in which to store the ontology. Figure 7-2 is a screenshot of the Protege ontology editor, in which an operating system ontology repository is developed.


ONTOOS Protégé 3.4.4 (file:\E:\My%20Projects\ICSM2007\ONTOOS\ONTOOS.pprj, OWL / RDF Files)
File Edit Project OWL Reasoning Code Tools Window Collaboration Help
protégé
Metadata(ONTOOS.owl) OWLClasses Properties Individuals SWRL Rules Ontoviz OWLViz
SUBCLASS EXPLORER
For Project: ONTOOS
Asserted Hierarchy
CLASS EDITOR for Thread_api (instance of owl:Class)
Inferred View
Annotations
Asserted Conditions
Disjoints
Logic View Properties View

For Class: http://www.owl-ontologies.com/ONTOOS.owl#Thread_api

```
owl:Thing
▼ OSThing
  ▼ API
      ANSI_C_api
      Conditional_variable_api
      Message_queue_api
      Mutex_api
      Semaphore_api
      Thread_api
    API_standard
    Arch
  ▼ Data_type
    ▼ Compound_data_type
        Array_type
        Data_structure_type
        Pointer_type
      Defined_data_type
    ▼ Fundamental_data_type
        Bool_type
        Char_type
        Float_type
        Integer_type
        Void_type
  ▼ Driver
      Driver_atm
      Driver_block
      Driver_cdrom
      Driver_char
```

Property	Value	Lang
rdfs:comment		

```
NECESSARY & SUFFICIENT
API
provideService some Thread_service
NECESSARY
INHERITED
definedInOS some OS                       [from API]
definedInOS exactly 1                     [from API]
hasAPIStandard some API_standard          [from API]
hasAPIStandard exactly 1                  [from API]
hasParameter some Parameter               [from API]
hasParameter min 1                        [from API]
hasReturnType some Data_type              [from API]
hasReturnType exactly 1                   [from API]
```

Figure 7-2 Protege Ontology Editor Screenshot

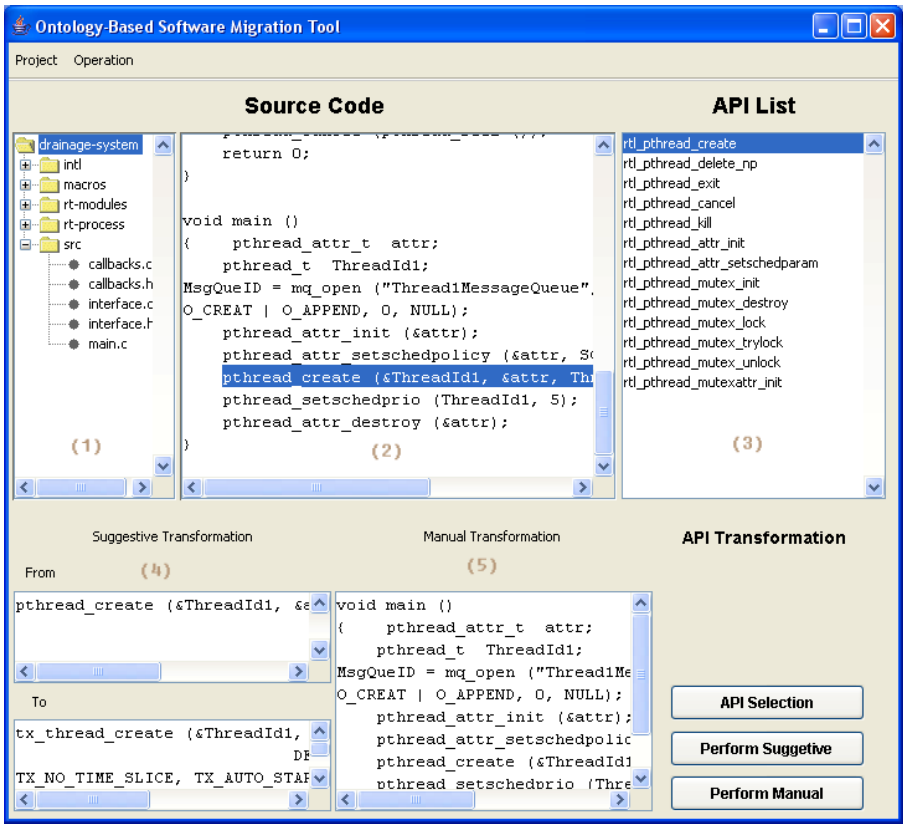
7.1.3 Ontology Accessing and Processing Layer

Protege-OWL API [107] is used in the ontology accessing and processing layer, which is an open-source Java library for the Web Ontology Language (OWL) and RDF (S). It provides APIs [107] which will allow programmers to develop both Protege plug-in and stand-alone applications that can access and utilise the ontology. Protege APIs in this layer encompass everything from ordinary ontology operations such as creating a new class and storing ontology in an .owl file, to knowledge acquisitions such as ontology query and DL reasoning.

7.1.4 Software Migration Layer

The OPTIMA transformation tool is in the software migration layer, which is implemented as a stand-alone application and is based on the Protege-OWL API. It enables software maintainers to perform program transformations either semi-automatically or manually. It consists of a graphical user interface, a program transformation engine and a transformation rule base. The source code will be analysed by a parser first. Then the analysed source code will be sent to the OPTIMA transformation tool. After that, the ontology-based program transformation approach will be conducted by accessing the ontology repository and the transformation rule base. Target code will be the displayed at the end.

Figure 7-3 shows the main form of the OPTIMA transformation tool, which consists of five sections: in section 1, all the folders and source files can be displayed; in section 2, the source code of a particular file can be presented; in section 3, the source API from a particular source code section can be shown; in section 4, the suggested transformation of a particular source code section can be shown; in section 5, the tool enables maintainers to intervene in the transformation process by performing some part of the program transformation manually.



```
    return 0;
}

void main ()
{     pthread_attr_t  attr;
    pthread_t  ThreadId1;
MsgQueID = mq_open ("Thread1MessageQueue",
O_CREAT | O_APPEND, 0, NULL);
    pthread_attr_init (&attr);
    pthread_attr_setschedpolicy (&attr, S
    pthread_create (&ThreadId1, &attr, Th
    pthread_setschedprio (ThreadId1, 5);
    pthread_attr_destroy (&attr);
}
```

```
rtl_pthread_create
rtl_pthread_delete_np
rtl_pthread_exit
rtl_pthread_cancel
rtl_pthread_kill
rtl_pthread_attr_init
rtl_pthread_attr_setschedparam
rtl_pthread_mutex_init
rtl_pthread_mutex_destroy
rtl_pthread_mutex_lock
rtl_pthread_mutex_trylock
rtl_pthread_mutex_unlock
rtl_pthread_mutexattr_init
```

```
pthread_create (&ThreadId1, &a
```

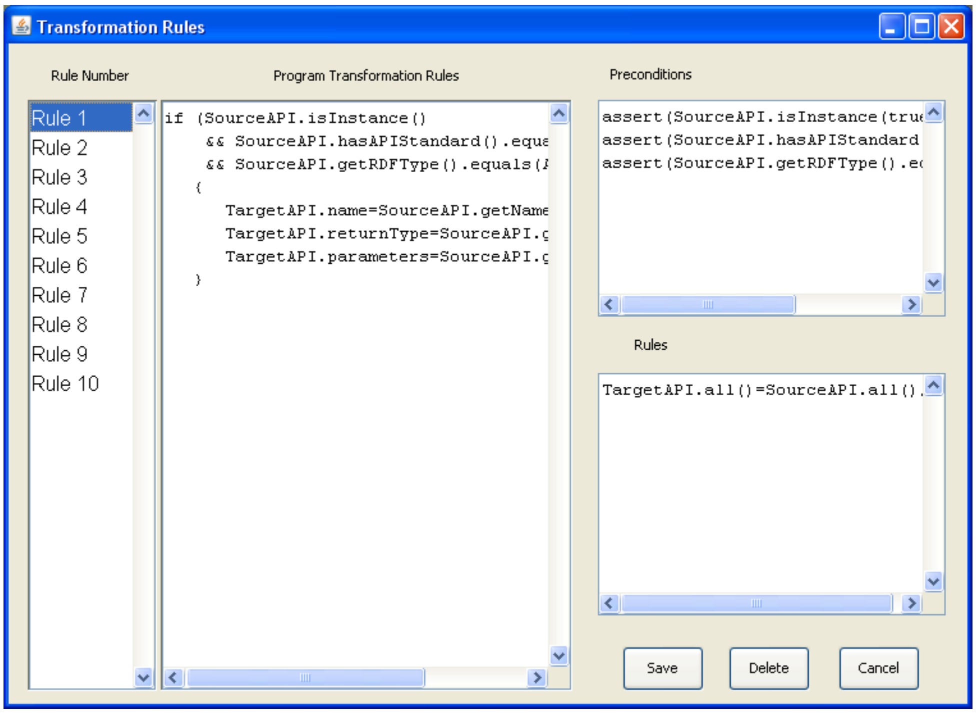
```
tx_thread_create (&ThreadId1,
                            DE
TX_NO_TIME_SLICE, TX_AUTO_STA
```

```
void main ()
{     pthread_attr_t  attr;
    pthread_t  ThreadId1;
MsgQueID = mq_open ("Thread1Me
O_CREAT | O_APPEND, 0, NULL);
    pthread_attr_init (&attr);
    pthread_attr_setschedpolic
    pthread_create (&ThreadId1
    pthread_setschedprio (Thre
```

Figure 7-3 OPTIMA Transformation Tool

7.1.4.1 Transformation Function

The suggested transformation will be given based on the transformation rules and knowledge acquisition from the ontology repository.



```
if (SourceAPI.isInstance()
    && SourceAPI.hasAPIStandard().equa
    && SourceAPI.getRDFType().equals(A
   {
      TargetAPI.name=SourceAPI.getName
      TargetAPI.returnType=SourceAPI.g
      TargetAPI.parameters=SourceAPI.g
   }
```

```
assert(SourceAPI.isInstance(true
assert(SourceAPI.hasAPIStandard
assert(SourceAPI.getRDFType().ec
```

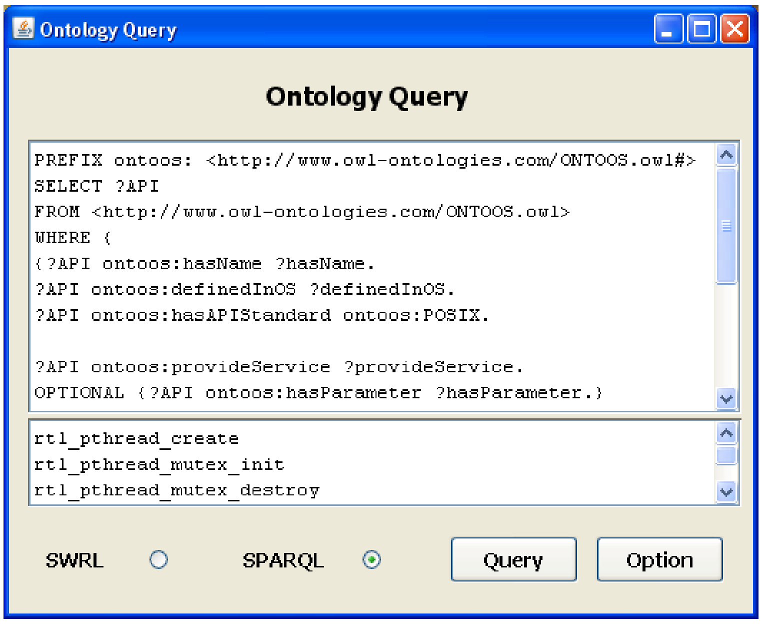
```
TargetAPI.all()=SourceAPI.all().
```

Figure 7-4 Transformation Rule Definition Interface

Figure 7-4 shows the transformation rule definition function of the OPTIMA transformation tool. Maintainers can define a set of transformation rules for each particular situation. Alternatively, a manual transformation can also be performed if needed, e.g., when the OPTIMA transformation tool cannot find the matching transformation rules for source code, more information from the ontology repository will be provided to facilitate the manual migration process.

7.1.4.2 Acquisition Function

The OPTIMA transformation tool is capable of API searching and matching based on the maintainers' assertions and requirements. They can perform API searching by selecting the features of a particular API. The selection of particular features will then be translated into a specific ontology query language, e.g., SPARQL or SWRL.



```
PREFIX ontoos: <http://www.owl-ontologies.com/ONTOOS.owl#>
SELECT ?API
FROM <http://www.owl-ontologies.com/ONTOOS.owl>
WHERE {
{?API ontoos:hasName ?hasName.
?API ontoos:definedInOS ?definedInOS.
?API ontoos:hasAPIStandard ontoos:POSIX.

?API ontoos:provideService ?provideService.
OPTIONAL {?API ontoos:hasParameter ?hasParameter.}
```

```
rtl_pthread_create
rtl_pthread_mutex_init
rtl_pthread_mutex_destroy
```

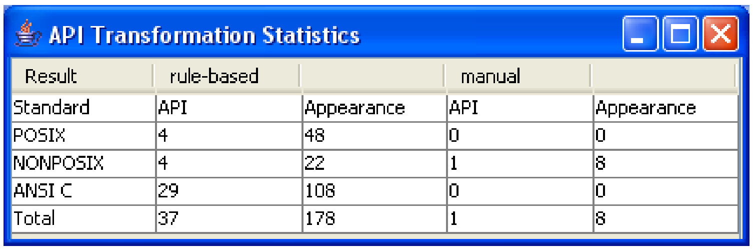
SWRL ○ [✓ SPARQL]

Figure 7-5 Ontology Query Interface

The interface shown in Figure 7-5 enables software maintainers to query the API by writing in specific ontology query languages.

7.1.4.3 Software Metrics Function

The OPTIMA transformation tool provides software metrics functions to count the number of APIs that are transformed during the migration. The software maintainers can get the useful information displayed as software metrics, which will help validating the OPTIMA approach. Figure 7-6 illustrates software metrics of a software migration by the OPTIMA transformation tool.



Result	rule-based		manual	
Standard	API	Appearance	API	Appearance
POSIX	4	48	0	0
NONPOSIX	4	22	1	8
ANSI C	29	108	0	0
Total	37	178	1	8

Figure 7-6 Software Metrics Function Interface

7.2 OntoComp

7.2.1 Architecture of OntoComp

To support the bottom-up ontology generation approach and the DL based ontology mapping algorithm, a prototype toolkit OntoComp (Ontology for Comprehension) is designed as a stand-alone application based on Protege-OWL API [107]. Figure 7-7 illustrates the architecture for this toolset. It includes two main parts: an ontology generation module and an ontology integration module.

The original inputs for OntoComp are three different files: Java source code, an XML Hibernate mapping file and a MySQL database. Two external open source tools are employed to process the input files, namely, Topcased Modelling Tools for deriving a UML class diagram from Java source code and MySQL Structure Magic for extracting an XML database schema from MySQL database. The final inputs for OntoComp are a UML class diagram, an XML Hibernate mapping file and an XML database schema. The ontology generation module transforms these into corresponding ontologies. The implementation of the ontology generation module is implemented based on the ATL model transformation (discussed in Chapter 4). The outputs of the ontology generation module then become the inputs of the ontology integration module, which integrates different ontologies by ontology mapping. The ontology integration module is implemented based on the DL based ontology mapping algorithm (presented in Chapter 5). Jena API is used to support parsing of the input ontology and DL reasoning, whilst Protege-OWL API is used to manipulate the ontology. The output of OntoComp is an integrated software system ontology which contains different knowledge perspectives.

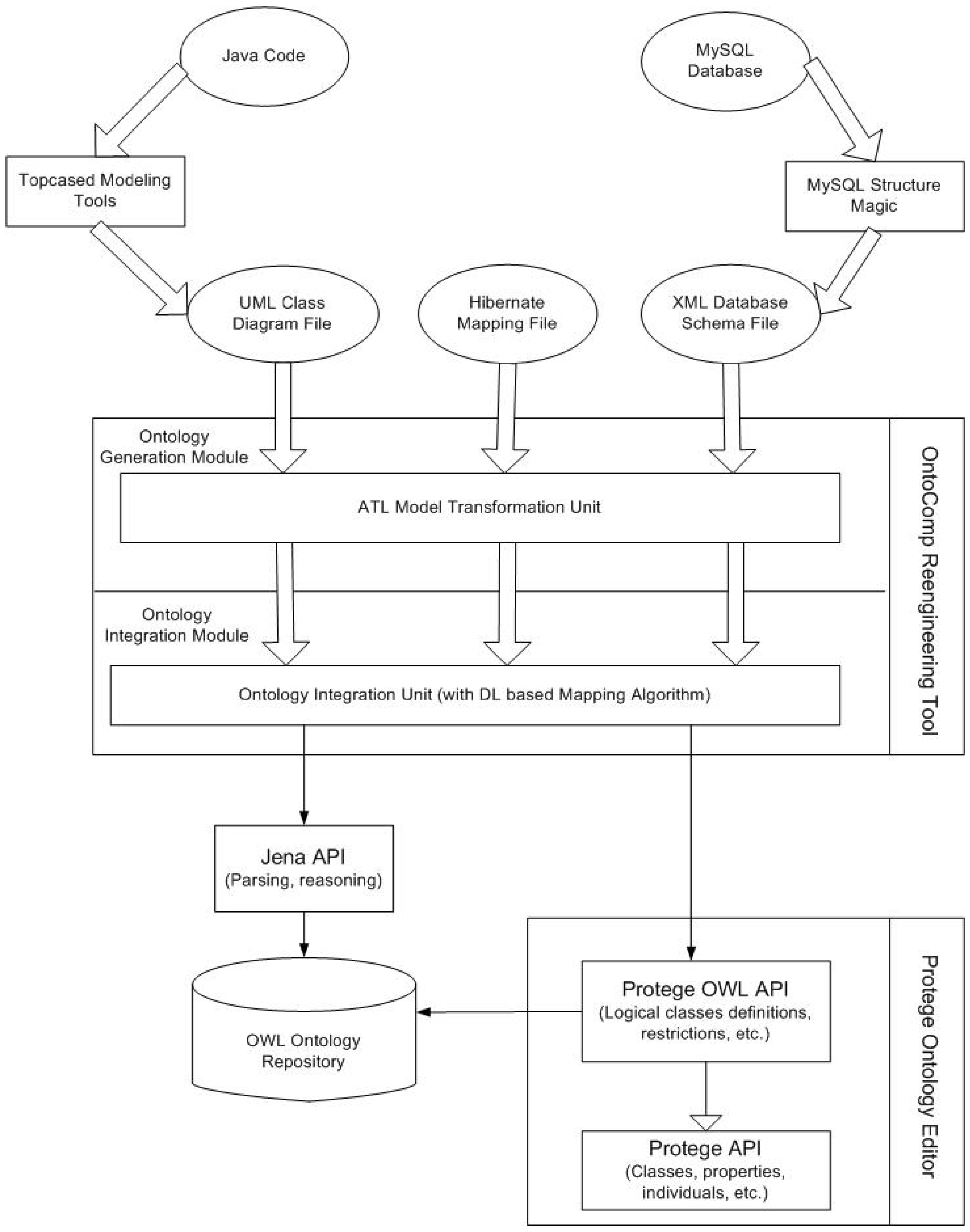

Java Code
MySQL Database
Topcased Modeling Tools
MySQL Structure Magic
UML Class Diagram File
Hibernate Mapping File
XML Database Schema File
Ontology Generation Module
ATL Model Transformation Unit
Ontology Integration Module
Ontology Integration Unit (with DL based Mapping Algorithm)
OntoComp Reengineering Tool
Jena API (Parsing, reasoning)
OWL Ontology Repository
Protege OWL API (Logical classes definitions, restrictions, etc.)
Protege API (Classes, properties, individuals, etc.)
Protege Ontology Editor

Figure 7-7 OntoComp Architecture

7.2.2 OntoComp Reengineering Tool

Based on the architecture presented in Figure 7-7, a prototype of the OntoComp reengineering tool has been developed. The main function of OntoComp include semi-automatic generation of different software system ontologies, semi-automatic integration of different software system ontologies, metrics and ontology query. The semi-automatic ontology generation is implemented by the bottom-up ontology generation approach and model transformation discussed in Chapter 4. The semi-automatic ontology integration is based on the DL based ontology mapping algorithm proposed in Chapter 5. The statistics function will support the further analysis of the proposed approaches. The ontology query function is implemented to supply knowledge acquisition to the integrated software system ontology. The following section will introduce the main GUI interface of the OntoComp reengineering tool and its functionalities.

7.2.2.1 OntoComp Main Interface

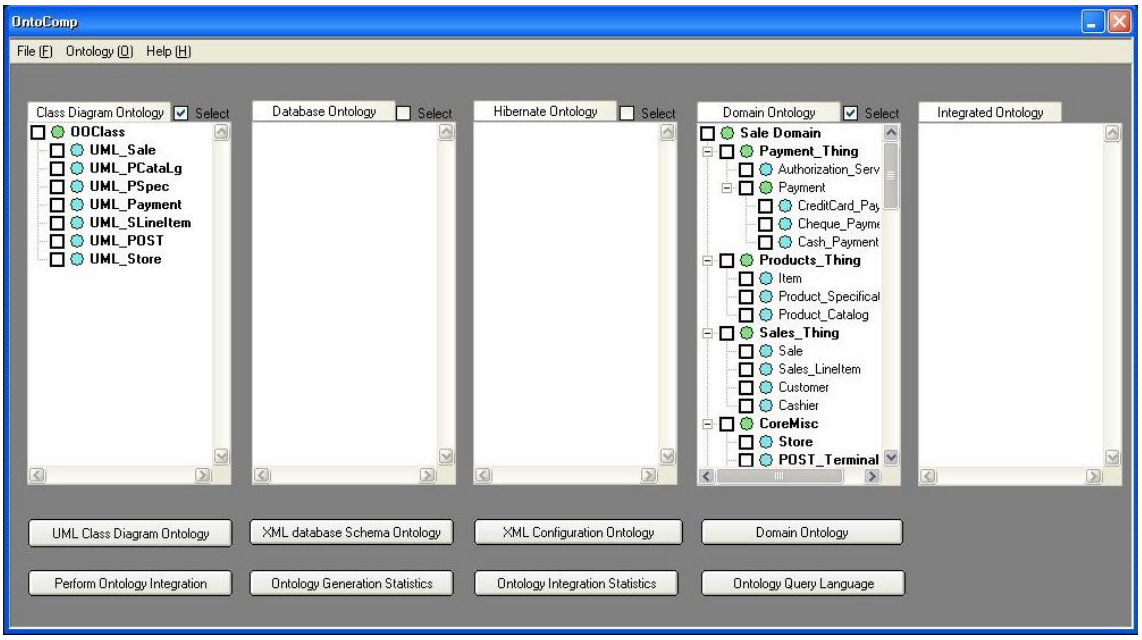


Figure 7-8 OntoComp Main Interface

Figure 7-8 presents the main interface of OntoComp. Currently, it has five ontology windows and eight buttons. The ontology windows are displaying: class diagram ontology, database schema ontology, Hibernate ontology, domain ontology and integrated ontology respectively. The buttons on the top row implement the functions of generating or loading different software system ontologies. At the moment, the class diagram ontology, the database schema ontology and Hibernate mapping ontology can be semi-automatically generated. The "perform ontology integration" button will integrate the selected ontologies by semantic mapping. The metrics buttons will create software metrics for analysis. The "ontology query language" button will allow maintainers to perform knowledge acquisitions on the ontology repository using ontology query languages.

7.2.2.2 OntoComp Ontology Generation and Integration

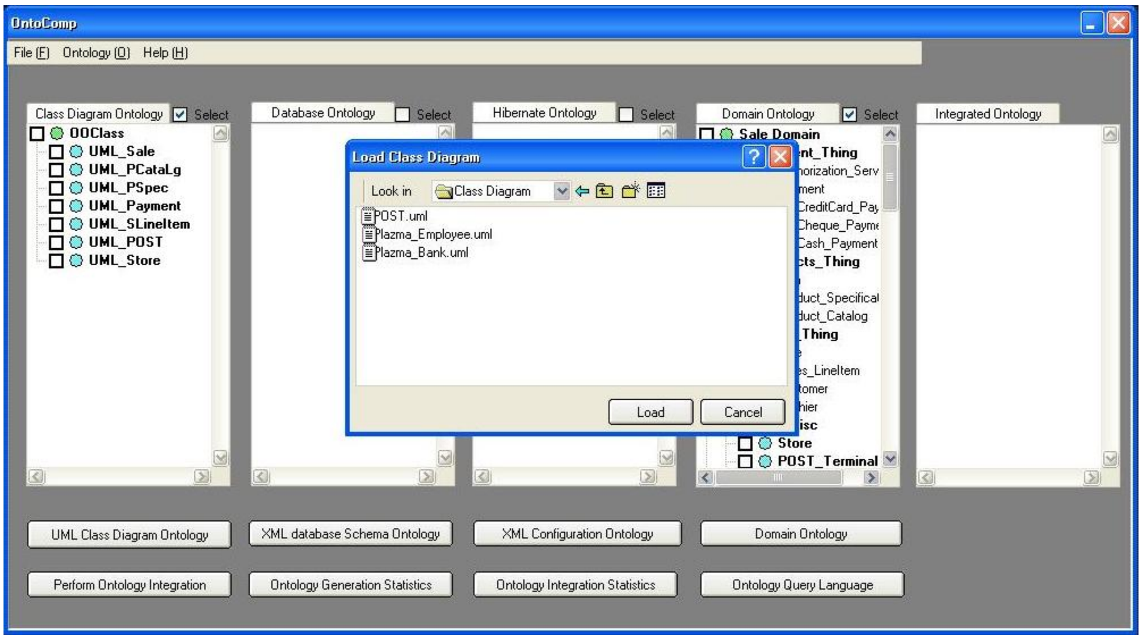
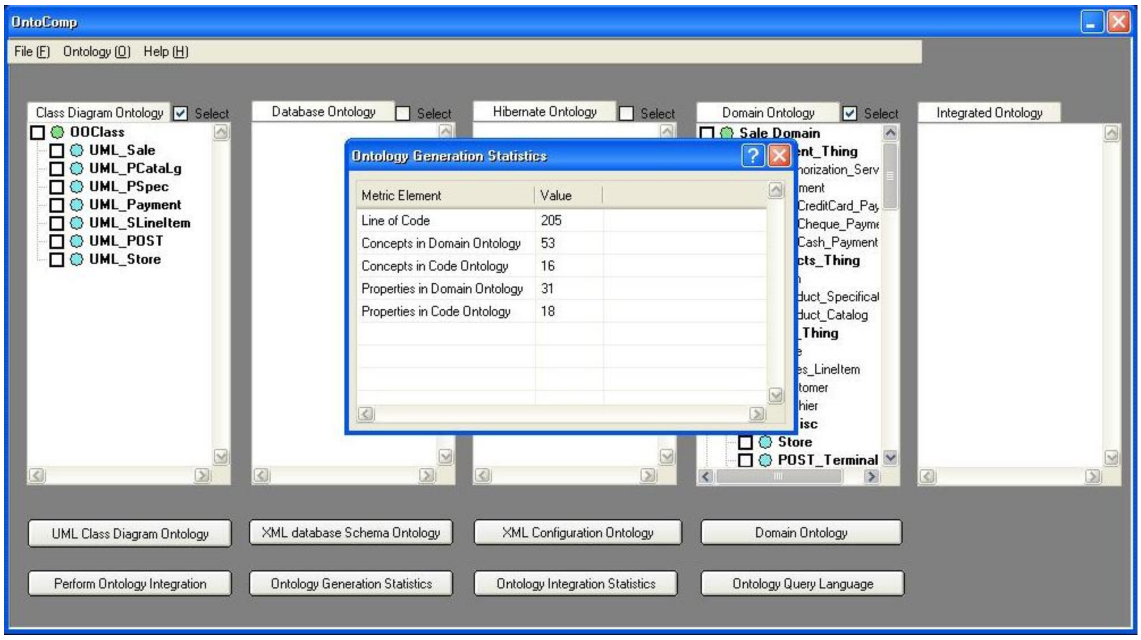


Figure 7-9 OntoComp Ontology Generation and Integration

Figure 7-9 shows the interface of the class diagram generation function. The class diagram ontology will be automatically generated after loading the uml class diagram file. Similarly, Database schema ontology and Hibernate configuration ontology will be generated by OntoComp. Domain ontology cannot be automatically generated at the moment: clicking the "domain ontology" button will only load existing domain ontology and display it in the ontology window. Once the different software system ontologies have been generated, clicking the "perform ontology integration" button will integrate selected software system ontologies and display the result in the ontology window.

7.2.2.3 OntoComp Software Metrics Function



Metric Element	Value
Line of Code	205
Concepts in Domain Ontology	53
Concepts in Code Ontology	16
Properties in Domain Ontology	31
Properties in Code Ontology	18

Figure 7-10 OntoComp Metrics Function

Figure 7-10 demonstrates the interface of the software metrics function of OntoComp. In order to evaluate and validate the proposed ontology based software comprehension approach, software metrics will be needed to perform analysis and comparison. Hence, OntoComp is implemented to be able to calculate metrics that are predefined by software maintainers, such as line of code, the number of concepts in the code ontology, the number of properties in the domain ontology, the number of semantic relationships that are detected during ontology integration, etc. By comparing different data from different reengineering projects, software maintainers will be able to find out what type of software system is most suitable for comprehension by the proposed approach.

7.2.2.4 OntoComp Ontology Query

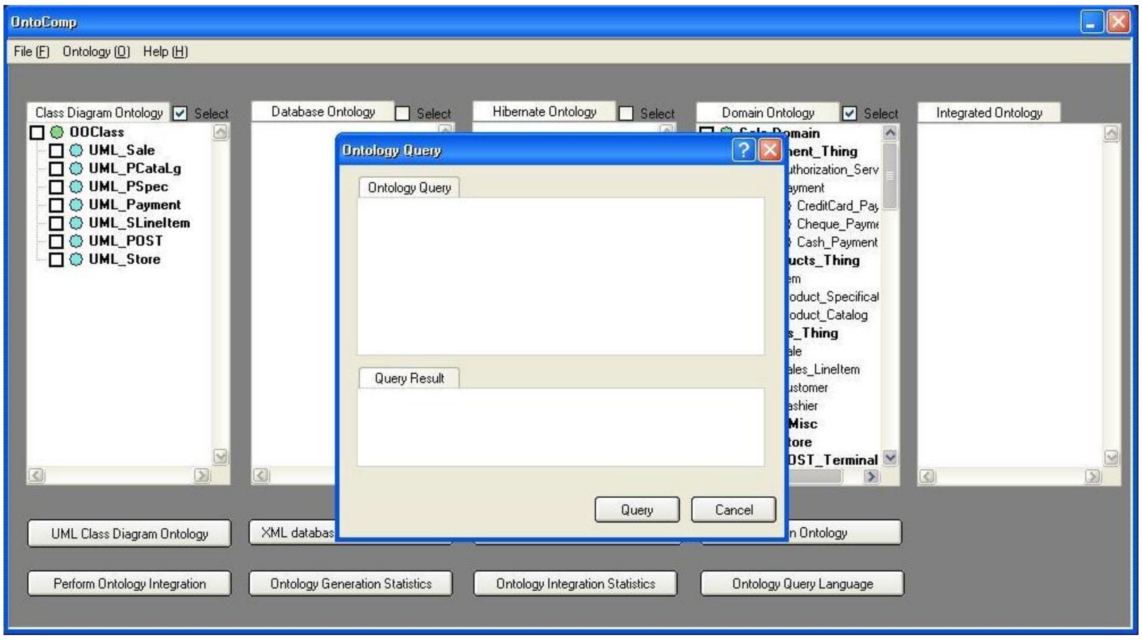


Figure 7-11 OntoComp Ontology Query

Figure 7-11 illustrates the ontology query function of OntoComp. It will allow software maintainers to perform knowledge acquisitions on the software system ontology repository using an ontology query language.

7.3 Summary

In this chapter, two prototype tools are introduced to support and validate the proposed ontology based software reengineering approaches. Those prototype tools are designed to simulate the cognition process based on the mental models represented by knowledge representation techniques. As a result, the traditional reengineering approaches should improve, reducing the number of error prone and time consuming manual tasks. However, human intervention in these tools is still almost inevitable since AI will never completely overtake human intelligence. Hence, these prototype software reengineering tools will only provide a semi-automated support to software maintainers.

➢ The prototype of the OPTIMA migration tool is designed to implement, support and validate the ontology based platform specific software migration approach. It has a three-layered architecture, which includes an ontology repository layer, an ontology accessing and processing layer and a rule-based software migration layer. The top-down ontology generation approach proposed in Chapter 4 supports the creation of the ontology repository layer. Jena API and Protege-OWL API are used to implement the ontology accessing and processing layer. The software migration layer contains a source code parser and a transformation rule base, which has a core role in performing a knowledge based software migration.

- The prototype of OntoComp (Ontology for Comprehension) is designed and implemented to support and validate the ontology based program understanding approaches discussed in Section 6.3 and Section 6.4. It takes three input files, namely, a UML class diagram file, an XML Hibernate framework mapping file and an XML MySQL database schema file. UML class diagram file is generated by Topcased Modelling tools, and the XML database schema file is created by MySQL Structure Magic. OntoComp has two main components: an ontology generation module and an ontology integration module. The ontology generation module is implemented using the bottom-up ontology generation approach proposed in Chapter 4; ATL model transformations are the main part of this module. The ontology integration module is realised by the DL based ontology mapping algorithm. Protege-OWL API is used to implement the manipulation of ontology, and a Jena API is used to implement the ontology parser and reasoning service. The output of OntoComp is an integrated software system ontology, in which semantic relationships between terms across the different ontologies have been detected and marked.

Chapter 8 Conclusions

Objectives

- To summarise the thesis and draw conclusions
- To revisit original contributions
- To evaluate the research by answering the research questions, reviewing the research hypotheses and revisiting the success criteria
- To illustrate the limitations of the work
- To propose future work

8.1 Summary of Thesis

This thesis aims to improve the traditional software reengineering methods by proposing a knowledge based software reengineering approach via ontology and description logic. The basic idea is to employ knowledge representation techniques to describe software systems and problem domains, and as a result, create a semi-automated software program to simulate and replace the software maintainers' mental processes and physical tasks during the reengineering procedure.

The research described in this thesis is postulated in the context of knowledge engineering and software reengineering. Ontology and description logic are selected to support a knowledge representation of a software system and its problem domain. The proposed research can be divided into three main stages, namely, software system ontology generation, software system ontology integration and software system ontology deployment. The ontology generation stage is supported by both bottom-up and top-down approaches. The ontology integration stage is implemented by a DL based ontology mapping algorithm. The ontology deployment stage explores the potential uses of a software system ontology in reengineering projects. In order to guarantee that the proposed research is systematic and well-structured, reference has been made to software taxonomy. The two main research subjects in this study are system software and data-dominant software. Use cases employing platform specific software migration, portable embedded software development, program comprehension and modularisation are performed to validate the proposed approach. The prototype of the supporting tools are designed and implemented to support and facilitate use cases.

8.2 Revisiting Original Contributions

This thesis proposes knowledge based solutions to some of the shortcomings in the traditional approaches to software reengineering, as observed in Chapter 1. A knowledge based software reengineering framework is proposed in Chapter 3. This section will revisit and extend the eight expected original contributions presented in Chapter 1 as follows:

- C1: In Chapter 3, a systematic ontology based software reengineering process has been proposed. Five steps have been defined: preparation, capture, coding, integration and development.

- C2: In Chapter 3, software system knowledge has been classified into three different categories: application domain knowledge, software engineering knowledge and code knowledge.

- C3: In Chapter 3, software system ontology generation has been divided into two approaches: the bottom-up and top-down.

- C4: In Chapter 4, the bottom-up software system ontology generation approach has been proposed based on model transformation techniques.

- C5: In Chapter 4, the six model transformation scenarios have been defined to implement bottom-up ontology generation.

- C6: In Chapter 4, twenty-nine model transformation rules are written in ATL model transformation language to support bottom-up ontology generation.
- C7: In Chapter 4, the top-down software system ontology generation approach has been proposed.
- C8: In Chapter 4, operating system ontology has been classified into three different representation categories: operating system functions/services, operating system architectures/components and operating system principles/theories.
- C9: In Chapter 4, eight operating system ontology development rules have been defined focusing on different development aspects in order to fulfil the requirements of software reengineering.
- C10: In Chapter 4, an operating system ontology has been developed based on the top-down approach.
- C11: In Chapter 6, software system ontology integration is formally defined as a process for detecting semantic relationships between concepts across different software system ontologies.
- C12: In Chapter 5, a description logic based ontology mapping algorithm is designed, which transforms the problem of detecting semantic relationships between concepts across different ontologies into the problem of deducing the satisfiability of DL formulae.
- C13: In Chapter 6, the ontology based platform specific software migration approach OPTIMA has been proposed and validated with a use case.
- C14: In Chapter 6, the ontology based portable embedded software development approach has been proposed and validated with a use case.

- C15: In Chapter 6, the ontology based program comprehension approach has been proposed and validated with use cases.
- C16: In Chapter 6, the ontology based program modularisation approach to identifying potential services for cloud computing has been proposed and validated with use cases.
- C17: In Chapter 7, a prototype of the OPTIMA software migration tool has been designed and implemented to support and validate the proposed approach.
- C18: In Chapter 7, a prototype of the OntoComp program comprehension tool has been designed and realised to support and validate the proposed approaches.

8.3 Evaluation

8.3.1 Answering Research Questions

The evaluation of this study starts by answering the proposed research questions. The global research question presented in Chapter 1 was:

How can software systems be described by knowledge representation techniques in order to support (semi-) automating manual software reengineering tasks?

This question has been answered by proposing a knowledge-based software reengineering framework and ontology based software reengineering process. The bottom-up and top-down approaches provide a means to represent software system knowledge in ontology. Description logic is employed to integrate different software system ontologies. Moreover, software system ontology has been deployed in several different software reengineering scenarios to replace the manual tasks, such as software migration and program comprehension.

A set of research questions was defined subsequently to refine this global question in detail.

RQ1: What knowledge of software systems is going to be represented?

Three different type of knowledge can be represented: application domain knowledge, software engineering knowledge and code knowledge. (Section 3.2.3)

- *What knowledge of software systems is needed in the context of software reengineering?*

Application domain knowledge, software engineering knowledge and code knowledge are all required in software reengineering. (Section 3.2.3)

- *What knowledge can be represented in relation to different categories of software systems?*

Code knowledge is required by system software related reengineering; application domain knowledge, software engineering knowledge and code knowledge are required by data-dominant software system reengineering. (Section 3.2.1)

RQ2: How may software system knowledge be represented?

Bottom-up and top-down approaches have been proposed to represent software system knowledge in ontology. (Section 3.2.4)

- *What knowledge representation techniques can be used to describe software system knowledge?*

Ontology and description logic are employed to represent software system knowledge. (Section 3.2)

- *How may a knowledge representation of software systems be created, i.e. manually or (semi-) automatically?*

Bottom-up and top-down approaches are proposed to create an ontology representation of software system. The bottom-up approach is a semi-automatic approach, whilst the top-down can be manual approach. (Chapter 4)

- *How may software system knowledge be integrated?*

Ontology integration is defined as a mapping detection of the semantic relationships between concepts across different ontologies. A description logic based ontology mapping algorithm is implemented to transform the problem of ontology mapping into one of logical formulae deduction. (Chapter 5)

- *How may a software system be linked with its knowledge representation?*

The linkage between a software system and it's ontology representation has been built during the ontology generation process, especially for bottom-up ontology generation. (Chapter 4)

- *What is the role of ontology and description logic in knowledge based software reengineering?*

Ontology is employed to represent software system knowledge and to facilitate knowledge based software reengineering approaches. Description logic is employed to support ontology mapping with logical deduction. (Chapter 5 and Chapter 6)

RQ3: How may software system knowledge be deployed in software reengineering?

Software system ontology deployment is the last step of the five steps in ontology based reengineering process. Knowledge acquisition is programed to retrieve the required knowledge from software system ontology. As a result, some of the manual efforts of reengineering can be semi-automated. Following two questions will explore it in detail. (Section 3.2.6 and Chapter 6)

- *Which software reengineering activities require software system knowledge?*

Ontology based software reengineering approaches have been discussed with the selected use cases, namely, ontology based software migration, ontology based portable software development, ontology based program comprehension and ontology based software modularisation. (Section 3.2.6 and Chapter 6)

- *How may software system knowledge be used in software reengineering* *projects?*

Firstly, operating system ontology has been deployed in platform specific software migration, in which the manual migration process is replaced by semi-automatic migration based on knowledge acquisition to OS ontology. Secondly, operating system ontology has been deployed in virtual operating system development, in which the manual system analysis is replaced by semi-automatic analysis of system API via knowledge acquisition to OS ontology. Thirdly, data-dominant software system ontology has been deployed in program comprehension process, which facilitates program understanding by integrating code ontology and domain ontology to generate domain-specific software system ontology. Fourthly, data-dominant software system ontology has been deployed in software modularisation process, which aims to generate service candidates for cloud computing environment by integrating code ontology, database ontology and framework ontology and then partitioning them to different modules. (Section 3.2.6 and Chapter 6)

RQ4: How may tools support to validate the proposed approach be provided?

The prototype of OPTIMA migration tool has been developed to validate OPTIMA approach. The prototype of OntoComp tool has been developed to validate ontology based program comprehension and modularisation. (Chapter 7)

8.3.2 Revisiting Research Propositions

The underlying proposition of this study is that *"Ontology and description logic can be used to represent the knowledge of software systems in order to semi-automate some of the manual efforts in reengineering and, as a result, improve the efficiency of the reengineering projects."* Knowledge based software reengineering framework and ontology based software reengineering process have been proposed in this thesis, which shows that this proposition is sound.

RP1: Ontology can be used to represent the knowledge of different software systems.

The knowledge in system software and data-dominant software has been represented in ontology in this thesis, which shows that this proposition is sound.

RP2: Domain ontology resources are available for ontology based domain-specific software system reengineering.

The existence of protege ontology library and etc. shows that this proposition is sound.

RP3: Software system ontology can be used to semi-automate some manual tasks in software reengineering projects and hence improve their efficiency.

The four ontology based software reengineering scnarios and the selected use cases show that this proposition is sound.

RP4: There are links between different perspectives of software knowledge within the same system. Integration of those different perspectives will enhance the understandability of existing software systems.

The mapping and integration of code ontology, database ontology, application framework ontology and domain ontology show that this proposition is sound.

8.3.3 Revisiting the Measure of Success

In Chapter 1, a set of measures are defined to validate the success of the proposed research described in this thesis. This section will revisit the predefined measure of success.

The proposed approach should be able to deal with at least two different kinds of software systems.

The proposed knowledge based software reengineering approach is able to deal with the selected use cases in system software and data-dominant software.

The extracted knowledge representation of software system should be machine readable in order to semi-automate some manual efforts.

OWL is employed to represent software system knowledge, which is machine readable.

The extracted software system knowledge representation should be reliable to perform forward engineering.

An ontology based portable software development approach has been proposed in Section 6.2, most of which is forward engineering.

The proposed approach should be feasible for realisation. i.e. It is possible to build a practical tool to demonstrate and validate the approach.

The prototype of OPTIMA migration tool, VRTOS and the prototype of OntoComp program comprehension tool have been designed and implemented to support and validate the proposed approaches in this study.

The proposed approach should support the modern computing paradigms such as cloud computing.

An ontology based program understanding and modularisation approach has been proposed to identify the potential service candidates in context of cloud computing in Section 6.4.

8.4 Limitations

Having discussed the original contributions and success criteria, the proposed research described in this thesis also has following limitations:

The bottom-up ontology generation approach may sacrifice some complex features of software system.

The fundamental mechanism of bottom-up ontology generation approach is model transformation, which is implemented based on the direct mapping and matching transformation rules. However, some of the complex situations may not be able to be represented by the transformation rules. At the current stage, those complicated situations are just simply ignored since they do not directly affect the result.

The top-down ontology generation approach may become complicated and time *consuming.*

The top-down ontology generation may become a time consuming process. Because of the special features of top-down approach, it can not be implemented (semi-) automatically. Manually create software system ontology based on design principles may become too time consuming and may even become a burden of the reengineering project. There is a right balance between manually creating software system ontology and reusing existing software system ontology. However, top-down approach may still become too complicated and time consuming.

8.5 Future Work

Based on the discussions regarding research questions, research propositions, original contributions, success criteria and limitations in the previous sections, the conclusions can be drawn. The knowledge based software reengineering approach via ontology and description logic, described in this thesis, is a novel, systematic and practical methodology for software reengineering. The use cases and supporting tools have supported and verified the success of the approach. The fact that human interventions are still required indicates that it is only semi-automatic. However, given the fact the pure manual efforts are time-consuming and error prone, this semi-automatic approach will improve the traditional software reengineering process greatly.

The research presented in this thesis is not the terminus. The following future work can be suggested to be pursued based on the present work.

- Based on the software taxonomy discussed in Section 2.1.5, the application domain of the proposed approach can be vertically extended by adding two more different software categories, i.e., control dominant software and computation dominant software.

- Based on the Figure 4.2 in Section 4.2.1, the proposed approach can be horizontally extended by adding or refining knowledge representation aspects. Apart from application domain knowledge, software engineering knowledge and code knowledge, more knowledge could be added in this approach. On the other hand, this knowledge could be refined into more detailed subcategories.

- Instead of using generic model transformation rules, bottom-up ontology generation approach could be improved by including more complex model transformation rules for the different situations.
- Top-down ontology generation approach could be improved by integrating bottom-up approach for some aspects. Therefore, a more efficient and straightforward middle-out ontology generation approach will be created.

References

[1] A. Abran and J. W.Moore, *Guide to the Software Engineering Body of Knowledge (SWEBOK) 2004 Version,* IEEE Computer Society, 2004.

[2] A. Ambrosio, D. Santos, F. Lucena and J. Silva, "Software Engineering Documentation: An Ontology-Based Approach", *the WebMedia & LA-Web 2004 Joint Conference 10th Brazilian Symposium on Multimedia and the Web 2nd Latin American Web Congress*, Oct. 2004, pp. 38-40.

[3] R. S. Arnold, *A Road Map Guide to Software Re-engineering,* IEEE Computer Society Press, 1992.

[4] AT&T, "AT&T Labs Research: UWIN", http://www.research.att.com/sw/tools/uwin/.

[5] F. Baader and B. Hollunder, "KRIS: Knowledge Representation and Inference System", *ACM SIGART Bulletin - Special Issue on Implemented Knowledge Representation and Reasoning Systems,* vol. 2(3), 1991, pp. 8-14.

[6] F. Baader and U. Sattler, "An Overview of Tableau Algorithms for Description Logics", *Studia Logica,* vol. 69(1), 2001, pp. 5-40.

[7] F. Baader, D. Calvanese, D. McGuinness, D. NardiPeter and Patel-Schneider, *The Description Logic Handbook,* Cambridge University Press, Jan. 2003.

[8] C. Bachman, "A CASE for Reverse Engineering", *Datamation,* vol. 34(13), 1988, pp. 49-56.

[9] P. Baumann, J. Fassler, M. Kiser, Z. Ozturk and L. Richter, "Semantics-based Reverse Engineering", Technical Report 94.08, Department of Computer Science, University of Zurich, Switzerland 1994.

[10] T. Berners-Lee, J. Hendler and O. Lassila, "The Semantic Web", *Scientific American,* vol. 284(5), May. 2001, pp. 34-43.

[11] Binaervarianz, "Jupe project", http://jupe.binaervarianz.de/.

[12] P. Bouquet, L. Serafini and S. Zanobini, "Semantic Coordination: A New Approach and An Application", *2nd International Semantic Web Conference (ISWC2003)*, USA, 2003, pp. 130-145.

[13] R. Brachman and J. Schmolze, "An Overview of the KL-ONE Knowledge Representation System", *Cognitive Science,* vol. 9(2), 1985, pp. 171-216.

[14] R. Brachman, A. Borgida, D. McGuinness, P. Patel-Schneider and L. A. Resnick, "The CLASSIC Knowledge Representation System or, KL-ONE: The Next Generation", MorganKaufman, 1989.

[15] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer,* vol. 20(4), Apr. 1987, pp. 10-19.

[16] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs", *International Journal of Man-Machine Studies,* vol. 18(6), June 1983, pp. 543-554.

[17] M. Buscher, M. Christensen, K. M. Hansen, P. Mogensen and D. Shapiro, *Configuring User-Designer Relations,* Springer, 2009.

[18] F. Chen, H. Zhou, J. Li, R. Liu, H. Yang, H. Li, H. Guo and Y. Wang, "An Ontology-based Approach to Portable Embedded System Development", *21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, Boston, USA, Jul. 2009, pp. 569-574.

[19] F. Chen, H. Guo, L. Dai and H. Yang, "An Application Framework for Ontology-based Data Mining", *Journal of Dalian University of Technology,* vol. Suppl., 43(S1), Oct. 2003.

[20] F. Chen, H. Yang, H. Guo and T. Liu, "Aspect-Oriented Programming based Software Evolution with Microsoft .NET." *21st IEEE International Conference on Software Maintenance (4 pages poster paper)*, Budapest, Hungary, Sep. 2005.

[21] E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery: a Taxonomy", *IEEE Software,* vol. 7(1), Jan. 1990, pp. 13-17.

[22] N. Choi, I.-Y. Song and H. Han, "A Survey on Ontology Mapping", *ACM*

SIGMOD Record, vol. 35(3), 2006, pp. 34-41.

[23] E. M. Clarke and J. M. Wing, "Formal Methods: State of the Art and Future Directions", *ACM Computing Surveys,* vol. 28(4), Sep. 1996, pp. 626-643.

[24] B. Curtis, H. Krasner and N. Iscoe, "A Field Study of the Software Design Process for Large Systems", *Communications of the ACM,* vol. 31(11), 1988, pp. 1268-1287.

[25] CygnusSolutions, "Cygwin", http://cygwin.com/.

[26] K. Czarnecki and U. W. Eisenecker, *Generative Programming,* Addison Wesley, 2000.

[27] R. Davis, H. Schrobe and P. Szolovits, "What is a knowledge representation?" *AI Magazine,* vol. 14(1), 1993, pp. 17-33.

[28] D. Desmet, D. Verkest and H. D. Man, "Operating System based Software Generation for Systems-on-Chip", *37th Annual ACM IEEE Design Automation Conference(DAC'00)*, Los Angeles, CA, Jun. 2000, pp. 396-401.

[29] P. Devanbu, R. J. Brachman, P. G. Selfridge and B. W. Ballard, "LaSSIE: a Knowledge-Based Software Information System", *Communications of the ACM,* vol. 34(5), May 1991, pp. 34-49.

[30] V. Devedzic, "Understanding Ontological Engineering", *Communications of the ACM,* vol. 45(4), Apr. 2002, pp. 136-144.

[31] DIG, "The new DIG interface standard (DIG 2.0)", http://dl.kr.org/dig/interface.html.

[32] D. Djuric, V. Devedzic and D. Gasevic, "Adopting Software Engineering Trends in AI", *IEEE Intelligent Systems,* vol. 22(1), Jan./Feb. 2007, pp. 59-66.

[33] ExpressLogic, "ThreadX User Guide", Express Logic, Inc., http://www.expresslogic.com.

[34] P. Farail, "Topcased", http://www.topcased.org/.

[35] R. Finkel, *Advanced Programming Language Design,* Addison-Wesley Publishing Company, 1996.

[36] A. Forward and T. C. Lethbridge, "A Taxonomy of Software Types to Facilitate Search and Evidence-Based Software Engineering", *2008 Conference of the Center for Advanced Studies on Collaborative Research: meeting of minds (CASCON '08)*, 2008.

[37] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition,* Addison Wesley, Sep. 2003.

[38] E. Furtado, J. J. V. Furtado, W. B. Silva, D. W. T. Rodrigues, L. d. S. Taddeo, Q. Limbourg and J. Vanderdonckt, "An Ontology Based Method for Universal Design of User Interfaces", *Workshop on Multiple User Interfaces over the Internet: Engineering and Applications Trends*, Lille, France, Sep. 2001.

[39] L. Gauthier, S. Yoo and A. A. Jerraya, "Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 20(11), Nov. 2001, pp. 1293-1301.

[40] A. Gerstlauer, H. Yu and D. D. Gajski, "RTOS Modeling for System Level Design", *Design, Automation and Test in Europe(DATE'03)*, Messe Munich, Germany, Mar. 2003, pp. 130-135.

[41] R. Girardi, C. G. d. Faria and L. Balby, "Ontology-based Domain Modeling of Multi-Agent Systems", *3rd International Workshop on Agent-Oriented Methodologies at International Conference on Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'04)*, Vancouver, Canada, Oct. 2004, pp. 51-62.

[42] T. R. Gruber, "A Translation Approach to Portable Ontology Specifications", *Knowledge Acquisition,* vol. 5(2), 1993, pp. 199-220.

[43] N. Guarino, "Formal Ontology and Information Systems", *1st International Conference on Formal Ontologies in Information Systems(FOIS'98)*, Trento, Italy, Jun. 1998, pp. 3-15.

[44] O. Hapon, "Plazma Business Solutions", http://plazma.sourceforge.net.

[45] B. Haslhofer, "MOF", http://metadaten-twr.org/2008/09/22/mof/.

[46] D. Hazaël-Massieux, "The Semantic Web and its applications at W3C", http://www.w3.org/2003/Talks/simo-semwebapp/all.htm.

[47] G. T. Heineman and W. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together,* Addison-Wesley Professional, 2001.

[48] M. Herr, U. Bath and A. Koschel, "Implementation of a Service Oriented Architecture at Deutsche Post MAIL", *Lecture Notes in Computer Science 3250, Springer, ECOWS 2004*, Erfurt, Germany, Sep. 2004, pp. 227-238.

[49] F. Hessel, V. M. D. Rosa, C. E. Reif, C. Marcon and T. G. S. D. Santos, "Scheduling Refinement in Abstract RTOS Models", *ACM Transactions on Embedded Computing Systems (TECS),* vol. 5(2), May 2006, pp. 342-354.

[50] I. Horrocks, "The FaCT System", 2nd International Conference on Analytic Tableaux and Related Methods (TABLEAUX'98), 1998, pp. 307--312.

[51] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof and M. Dean, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML", http://www.w3.org/Submission/SWRL/, May 2004.

[52] M. N. Huhns, "Software Development with Objects, Agents, and Services", *3rd International Workshop on Agent-Oriented Methodologies (Keynotes)*, Vancouver, Canada, Oct. 2004.

[53] IEEE, "Reuse - Software Engineering Online", http://www.computer.org/portal/web/seonline/reuse.

[54] IEEE, "IEEE Standard Collection: Software Engineering", IEEE Inc., New York, 1997.

[55] IEEE, "The Open Group Base Specifications Issue 6. IEEE Std 1003.1-2001", The IEEE and The Open Group, 2001.

[56] InteropSystems, "Interix", http://www.interix.com/.

[57] M. Jackson, "Problem Frames and Software Engineering", *Journal of Information and Software Technology,* vol. 47(14), 2005, pp. 903-912.

[58] R. Janka, "A New Development Framework Based On Efficient Middleware for Real-Time Embedded Heterogeneous Multicomputers", *IEEE Conference and*

Workshop on Engineering of Computer-Based Systems (ECBS '99), Nashville, USA, Mar. 1999, pp. 261-268.

[59] D. Jin and J. R. Cordy, "Ontology-Based Software Analysis and Reengineering Tool Integration: The OASIS Service-Sharing Methodology", *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Budapest, Hungary, Sep. 2005, pp. 613-616.

[60] W. Johnson, M. S. Feather and D. R. Harris, "Representation and Presentation of Requirements Knowledge", *IEEE Transaction on Software Engineering,* vol. 18, Oct. 1992, pp. 853-869.

[61] W. L. Johnson and E. Soloway, "PROUST: Knowledge-Based Program Understanding", *IEEE Transaction on Software Engineering,* vol. SE-11(3), Mar. 1985, pp. 267-275.

[62] F. Jouault, F. Allilaire, J. Bézivin and I. Kurtev, "ATL: A model transformation tool", *Science of Computer Programming,* vol. 72(1-2), 2008, pp. 31-39.

[63] H. Kaindl, "Portability of Software", *ACM SIGPLAN Notices,* vol. 23(6), 1988, pp. 59 - 68.

[64] M. U. Khan, K. Geihs, F. Gutbrodt, P. Göhner and R. Trauter, "Model-Driven Development of Real-Time Systems with UML 2.0 and C", *4th Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD/MOMPES'06)*, Potsdam, Germany, Mar. 2006, pp. 33-42.

[65] A. Kleppe, J. Warmer and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise,* Addison Wesley, 2003.

[66] C. Kothapalli, "MaintainJ", http://www.maintainj.com/.

[67] I. Kurtev, J. Bezivin and M. Aksit, "Technical Spaces: an Initial Appraisal", *Confederated International Conferences (CoopIS, DOA'02), Industrial Track*, Irvine, 2002.

[68] C. Larman, *Applying UML and patterns: an introduction to object-oriented analysis and design,* Prentice Hall PTR, 1997.

[69] M. M. Lehman, "Laws of Software Evolution Revisited", *LNCS 1149*, 1997, pp. 108-124.

[70] M. M. Lehman and J. F. Ramil, "Towards a Theory of Software Evolution and Its Practical Impact", *International Symposium on the Principles of Software Evolution, Invited Talk*, Washington, USA, 2000, pp. 2-11.

[71] M. M. Lehman and J. F. Ramil, "Software Evolution and Software Evolution Processes", *Annals of Software Engineering,* vol. 14(1-4), 2002, pp. 275-309.

[72] M. M. Lehman and J. F. Ramil, "Software Evolution in the Age of Component-Based Software Engineering", *IEE Proceedings Software*, Dec. 2000, pp. 249-255.

[73] Z. Lei, S. Xia, Z. Xia and Z. Yong, "Study on Ontology Partition Based on Ant Colony Algorithm", *9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD' 08)*, Thailand, Aug. 2008, pp. 73-78.

[74] Y. Li, H. Yang and W. C. Chu, "A Concept-Oriented Belief Revision Approach to Domain Knowledge Recovery from Source Code", *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 13(1), 2001, pp. 31-52.

[75] Y. Li, H. Yang and W. Chu, "A Concept-Oriented Belief Revision Approach to Domain Knowledge Recovery from Source Code", *Journal of Software Maintenance: Research and Practice,* vol. 13(1), Jan. 2001, pp. 31-52.

[76] B. P. Lientz and E. B. Swanson, *Software Maintenance Management,* Addison-Wesley Longman Publishing Co., Inc., 1980.

[77] X. Liu, Z. Chen, H. Yang, H. Zedan and W. Chu, "A Design Framework for System Re-engineering", *4th Asia-Pacific Software Engineering and International Computer Science Conference*, Hong Kong, 1997, pp. 342-352.

[78] R. MacGregor and R. Bates, "The Loom Knowledge Representation Language", 1987.

[79] J. Madsen, K. Virk and M. Gonzales, "Abstract RTOS Modelling for Multiprocessor System-on-Chip", *International Symposium on*

System-on-chip(SoC'03), Tampere, Finland, Nov. 2003, pp. 147-150.

[80] B. McBride, "Four Steps Towards the Widespread Adoption of a Semantic Web", *1st International Semantic Web Conference on The Semantic Web*, Italy, 2002, pp. 419-422.

[81] L. Mei, W. K. Chan and T. H. Tse, "A Tale of Clouds: Paradigm Comparisons and Some Thoughts on Research Issues", *IEEE Asia-Pacific Services Computing Conference (APSCC'08)*, Yilan, Taiwan, Dec. 2008, pp. 464-469.

[82] T. Mens and P. V. Gorp, "A Taxonomy of Model Transformation", *International Workshop on Graph and Model Transformation (GraMoT 2005)*, 2006, pp. 125-142.

[83] Microsoft, "Microsoft Interoperability and Migration Center", http://ms.helifan.net/technet/interopmigration/default.mspx.

[84] K. Miriyala and M. T. Harandi, "Automatic Derivation of Formal Software Specifications from Informal Descriptions", *IEEE Transaction on Software Engineering,* vol. 17, Oct. 1991, pp. 1,126-1,142.

[85] P. Mohagheghi and V. Dehlen, "Where Is the Proof? - A Review of Experiences from Applying MDE in Industry", *4th European Conference on Model Driven Architecture Fondations and Applications (EDMDA-FA'08)*, Berlin, Germany, Jun. 2008, pp. 432-443.

[86] R. A. Mosbeck, L. C. Reeve and J. R. Thedens, " Software Portability in Open Architectures", *IEEE/AIAA 20th Digital Avionics Systems Conference (DASC'01)*, Daytona Beach, USA, Oct. 2001, pp. 9E1/1-9E1/8.

[87] P. Naur and B. Randell, "Software Engineering: Report on a Conference Sponsored by NATO Science Committee", Garmisch, Germany, Oct. 1968.

[88] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator and W. Swartout, "Enabling Technology For Knowledge Sharing", *AI Magazine,* vol. 12(3), 1991, pp. 36-56.

[89] OMG, "XML Metadata Interchange (XMI), v2.0 specification", omg/formal/03-05-01, 2003.

[90] OMG, "MOF 2.0/XMI Mapping Specification, v2.1", omg/formal/03-05-01, 2005.

[91] OMG, "Meta Object Facility (MOF) Specification v1.4", Apr. 2002.

[92] OMG, "MDA Guide Version 1.0.1 ", omg/2003-06-01, Jun. 2003.

[93] OMONDO, "EclipseUML 2007 Europa ", http://www.eclipsedownload.com/.

[94] C. Peltason, "The BACK System - An Overview", *ACM SIGART Bulletin - Special Issue on Implemented Knowledge Representation and Reasoning Systems,* vol. 2(3), 1991, pp. 114-119.

[95] A. Pokos, "MySQL Structure Magic ", http://www.phpclasses.org/package/4740-PHP-Synchronize-MySQL-database-schemata.html.

[96] B. Ramesh and V. Dhar, "Supporting Systems Development by Capturing Deliberations During Requirements Engineering", *IEEE Transaction on Software Engineering,* vol. 18, Jun. 1992, pp. 498-510.

[97] C. Rich and R. C. Waters, "Knowledge Intensive Software Engineering", *IEEE Transaction on Knowledge and Data Engineering,* vol. 4, Oct. 1992, pp. 424-430.

[98] D. C. Rine and R. M. Sonnemann, "Investments in Reusable Software: A Study of Software Reuse Investment Success Factors", *Journal of Systems and Software,* vol. 41, 1997, pp. 17-32.

[99] J. B. Ronald, P. Devanbu, P. G. Selfridge, D. Belanger and Y. Chen, "Toward a Software Information System", *AT&T Technical Journal,* vol. 69(2), 1990, pp. 22-41.

[100] RTLinux, "RTLinux V3.0 Source Code", ftp://ftp.rtlinux.com/pub/rtlinux/v3/.

[101] G. K. Saha, "Web Ontology Language (OWL) and Semantic Web", *Ubiquity* vol. 8(35), 2007, pp. 1-24.

[102] A. Schlicht and H. Stuckenschmidt, "A Flexible Partitioning Tool for Large Ontologies", *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT '08)*, Sydney, Australia, Dec. 2008, pp.

482-488.

[103] D. Sidarkeviciute, E. Tyugu and A. Kuusik, "A Knowledge-Based Toolkit for Software Visualisation", *11th Knowledge-Based Software Engineering Conference*, Syracuse, USA, Sep. 1996, pp. 125-133.

[104] J. F. Sowa, *Knowledge Representation,* Brooks/Cole, an imprint of Thomson Learning, 2000.

[105] Stanford, "Protege Ontology Editor and Knowledge Acquisition System", http://protege.stanford.edu/.

[106] Stanford, "Protege Ontology Library", http://protegewiki.stanford.edu/wiki/Protege_Ontology_Library.

[107] Stanford, "Protégé Programming Development Kit (PDK)", http://protege.stanford.edu/doc/dev.html.

[108] M.-A. Storey, "Theories, Methods and Tools in Program Comprehension: Past, Present and Future", *13th International Workshop on Program Comprehension (IWPC'05)*, Missouri, USA, May. 2005, pp. 181-191.

[109] C. Szyperski, *Component Software: Beyond Object-Oriented Programming,* Addison-Wesley Professional, 2002.

[110] J. J. P. Tsai, A. Liu, E. Juan and A. Sahay, "Knowledge-Based Software Architectures: Acquisition, Specification, and Verification", *IEEE Transaction on Knowledge and Data Engineering,* vol. 11(1), Jan./Feb. 1999, pp. 187-201.

[111] UB, "Green Project", http://green.sourceforge.net/.

[112] M. Uschold and M. King, "Towards a Methodology for Building Ontologies", *Workshop on Basic Ontological Issues in Knowledge Sharing*, 1995.

[113] M. Uschold and M. Gruninger, "Ontologies: Principles, Methods and Applications", *Knowledge Engineering Review,* vol. 11, 1996, pp. 93--136.

[114] M. Vuletic, L. Pozzi and P. Ienne, "Programming Transparency and Portable Hardware Interfacing: Towards General-Purpose Reconfigurable Computing", *15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'04)*, Galveston, Texas, Sep. 2004, pp.

339-351.

[115] W3C, "Web Services Glossary", http://www.w3.org/TR/ws-gloss/, 2002.

[116] W3C, "SPARQL Query Language for RDF", http://www.w3.org/TR/rdf-sparql-query/, 2008.

[117] S. Wang and S. Malik, "Synthesizing Operating System Based Device Drivers in Embedded Systems", *1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis(CODES+ISSS'03)*, Newport Beach, CA, Oct. 2003, pp. 37-44.

[118] Y. Wang, F. Chen, L. Xu, H. Guo and J. Wan, "An Implementation of a Multi-Interface Virtual Real-Time Operating System on Windows Platform", *Journal of Dalian University of Technology,* vol. Suppl., 43(S1), Oct. 2003, pp. 100-102 (in Chinese).

[119] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA,* Addison-Wesley, 2003.

[120] I. Warren, *The Renaissance of Legacy Systems: Method Support for Software-System Evolution,* Springer-Verlag, 1999.

[121] P. Wongthongtham, E. Chang and T. S. Dillon, "Methodology for Multi-site Software Engineering Using Ontology", *International Conference on Software Engineering Research and Practice (SERP '04)*, USA, 2004, pp. 477-482.

[122] J. Woodcock, P. G. Larsen, J. Bicarregui and J. Fitzgerald, "Formal Methods: Practice and Experience", *ACM Computing Surveys (CSUR),* vol. 41(4), Oct. 2009, pp. 1-36.

[123] H. Yang, Z. Cui and P. O'Brien, "Extracting Ontologies from Legacy Systems for Understanding and Re-Engineering", *23rd IEEE Annual International Computer Software and Applications Conference (COMPSAC'99)*, Washington, USA, 1999, pp. 21-26.

[124] H. Yang, X. Liu and H. Zedan, "Abstraction: A Key Notion for Reverse Engineering in a System Reengineering Approach", *Journal of Software Maintenance: Research and Practice,* vol. 12, 2000, pp. 197-228.

[125] H. Yang and M. Ward, *Successful Evolution of Software Systems,* Artech House, Inc., 2003.

[126] Y. Yi, D. Kim and S. Ha, "Virtual Synchronization Technique with OS Modeling for Fast and Time-accurate Cosimulation", *1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis(CODES+ISSS'03)*, Newport Beach, CA, Oct. 2003, pp. 1-6.

[127] L. Youseff, M. Butrico and D. D. Silva, "Toward a Unified Ontology of Cloud Computing", *Grid Computing Environments Workshop (GCE'08)*, Nov. 2008, pp. 1-10.

[128] Y. Zhang, J. Rilling and V. Haarslev, "An Ontology-based Approach to Software Comprehension - Reasoning about Security Concerns", *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Chicago, USA, Sep. 2006, pp. 333-342.

[129] Y. Zhang, R. Witte, J. Rilling and V. Haarslev, "Ontology-based Program Comprehension Tool Supporting Website Architectural Evolution", *8th IEEE International Symposium on Web Site Evolution (WSE'06)*, Philadelphia, PA, Sep. 2006, pp. 41-49.

[130] H. Zhou, H. Yang and A. Hugill, "An Ontology-Based Approach to Reengineering Enterprise Software for Cloud Computing", *34rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'10)*, Korea, 2010, pp. 383-388.

[131] H. Zhou, F. Chen and H. Yang, "Developing Application Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology", *8th International Conference on Quality Software (QSIC'08)*, Oxford, UK, Aug. 2008.

[132] H. Zhou, J. Kang, F. Chen and H. Yang, "OPTIMA: an Ontology-based PlaTform-specIfic software Migration Approach", *7th International Conference on Quality Software (QSIC'07)*, Portland, Oregon, USA, Oct. 2007, pp. 143-152.

[133] H. Zhou, "COSS: Comprehension by Ontologising Software System", *24th IEEE International Conference on Software Maintenance (ICSM'08)*, Beijing,

China, Sep. 2008.

[134] C. Zimmer and A. Rauschmayer, "Tuna: Ontology-Based Source Code Navigation and Annotation", *Workshop on Ontologies as Software Engineering Artifacts (OOPSLA)*, Vancouver, Canada, 2004.

Appendix A Prototype of RTOS Ontology

This section presents an owl file of the manually created RTOS ontology. It is only a prototype, which does not cover all the concepts and instances in RTOS ontology.

```
<?xml version="1.0"?>
<rdf:RDF
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xmlns:owl="http://www.w3.org/2002/07/owl#"
      xmlns="http://www.owl-ontologies.com/ONTOOS.owl#"
      xmlns:p1="http://www.owl-ontologies.com/assert.owl#"
   xml:base="http://www.owl-ontologies.com/ONTOOS.owl">
   <owl:Ontology rdf:about=""/>
   <owl:Class rdf:ID="Void_type">
      <rdfs:subClassOf>
         <owl:Class rdf:ID="Fundamental_data_type"/>
      </rdfs:subClassOf>
      <owl:disjointWith>
         <owl:Class rdf:ID="Integer_type"/>
      </owl:disjointWith>
      <owl:disjointWith>
         <owl:Class rdf:ID="Float_type"/>
      </owl:disjointWith>
      <owl:disjointWith>
         <owl:Class rdf:ID="Char_type"/>
      </owl:disjointWith>
      <owl:disjointWith>
         <owl:Class rdf:ID="Bool_type"/>
      </owl:disjointWith>
   </owl:Class>
   <owl:Class rdf:ID="Timer_api">
      <owl:equivalentClass>
         <owl:Class>
            <owl:intersectionOf rdf:parseType="Collection">
               <owl:Class rdf:ID="API"/>
               <owl:Restriction>
                  <owl:onProperty>
                     <owl:ObjectProperty rdf:ID="provideService"/>
                  </owl:onProperty>
                  <owl:someValuesFrom>
                     <owl:Class rdf:ID="Timer_service"/>
                  </owl:someValuesFrom>
               </owl:Restriction>
            </owl:intersectionOf>
         </owl:Class>
      </owl:equivalentClass>
   </owl:Class>
   <owl:Class rdf:ID="Data_structure_type">
```

```
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Compound_data_type"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:ID="Pointer_type"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Array_type"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:ID="Thread_service">
    <owl:disjointWith>
      <owl:Class rdf:ID="Message_queue_service"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Mutex_service"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Semaphore_service"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Timer_service"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:ID="System_service"/>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:someValuesFrom>
          <owl:Class rdf:ID="Thread_api"/>
        </owl:someValuesFrom>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasAPI"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="OS_type">
    <owl:disjointWith>
      <owl:Class rdf:about="#System_service"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Parameter"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#API"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="OS"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Data_type"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="API_standard"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
```

```
        <owl:Class rdf:ID="OSThing"/>
      </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="#Char_type">
      <rdfs:subClassOf>
        <owl:Class rdf:about="#Fundamental_data_type"/>
      </rdfs:subClassOf>
      <owl:disjointWith rdf:resource="#Void_type"/>
      <owl:disjointWith>
        <owl:Class rdf:about="#Integer_type"/>
      </owl:disjointWith>
      <owl:disjointWith>
        <owl:Class rdf:about="#Float_type"/>
      </owl:disjointWith>
      <owl:disjointWith>
        <owl:Class rdf:about="#Bool_type"/>
      </owl:disjointWith>
    </owl:Class>
    <owl:Class rdf:ID="Thread_address_parameter">
      <rdfs:subClassOf>
        <owl:Class rdf:ID="Address_parameter"/>
      </rdfs:subClassOf>
      <owl:disjointWith>
        <owl:Class rdf:ID="Thread_attr_address_parameter"/>
      </owl:disjointWith>
    </owl:Class>
    <owl:Class rdf:ID="Linux">
      <rdfs:subClassOf>
        <owl:Class rdf:about="#OS"/>
      </rdfs:subClassOf>
      <owl:disjointWith>
        <owl:Class rdf:ID="Windows"/>
      </owl:disjointWith>
      <owl:disjointWith>
        <owl:Class rdf:ID="Embedded-misc"/>
      </owl:disjointWith>
    </owl:Class>
    <owl:Class rdf:about="#Bool_type">
      <owl:disjointWith rdf:resource="#Void_type"/>
      <owl:disjointWith>
        <owl:Class rdf:about="#Integer_type"/>
      </owl:disjointWith>
      <owl:disjointWith>
        <owl:Class rdf:about="#Float_type"/>
      </owl:disjointWith>
      <owl:disjointWith rdf:resource="#Char_type"/>
      <rdfs:subClassOf>
        <owl:Class rdf:about="#Fundamental_data_type"/>
      </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="#Thread_api">
      <owl:equivalentClass>
        <owl:Class>
          <owl:intersectionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#API"/>
            <owl:Restriction>
              <owl:someValuesFrom rdf:resource="#Thread_service"/>
```

```
                <owl:onProperty>
                  <owl:ObjectProperty rdf:about="#provideService"/>
                </owl:onProperty>
              </owl:Restriction>
            </owl:intersectionOf>
          </owl:Class>
        </owl:equivalentClass>
      </owl:Class>
      <owl:Class rdf:ID="Message_queue_api">
        <owl:equivalentClass>
          <owl:Class>
            <owl:intersectionOf rdf:parseType="Collection">
              <owl:Class rdf:about="#API"/>
              <owl:Restriction>
                <owl:someValuesFrom>
                  <owl:Class rdf:about="#Message_queue_service"/>
                </owl:someValuesFrom>
                <owl:onProperty>
                  <owl:ObjectProperty rdf:about="#provideService"/>
                </owl:onProperty>
              </owl:Restriction>
            </owl:intersectionOf>
          </owl:Class>
        </owl:equivalentClass>
      </owl:Class>
      <owl:Class rdf:about="#Thread_attr_address_parameter">
        <rdfs:subClassOf>
          <owl:Class rdf:about="#Address_parameter"/>
        </rdfs:subClassOf>
        <owl:disjointWith rdf:resource="#Thread_address_parameter"/>
      </owl:Class>
      <owl:Class rdf:about="#Compound_data_type">
        <rdfs:subClassOf>
          <owl:Class rdf:about="#Data_type"/>
        </rdfs:subClassOf>
        <owl:disjointWith>
          <owl:Class rdf:about="#Fundamental_data_type"/>
        </owl:disjointWith>
        <owl:disjointWith>
          <owl:Class rdf:ID="Defined_data_type"/>
        </owl:disjointWith>
      </owl:Class>
      <owl:Class rdf:about="#Semaphore_service">
        <owl:disjointWith>
          <owl:Class rdf:about="#Message_queue_service"/>
        </owl:disjointWith>
        <owl:disjointWith>
          <owl:Class rdf:about="#Mutex_service"/>
        </owl:disjointWith>
        <owl:disjointWith rdf:resource="#Thread_service"/>
        <owl:disjointWith>
          <owl:Class rdf:about="#Timer_service"/>
        </owl:disjointWith>
        <rdfs:subClassOf>
          <owl:Class rdf:about="#System_service"/>
        </rdfs:subClassOf>
      </owl:Class>
```

```
<owl:Class rdf:about="#Integer_type">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Fundamental_data_type"/>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#Void_type"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Float_type"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Char_type"/>
  <owl:disjointWith rdf:resource="#Bool_type"/>
</owl:Class>
<owl:Class rdf:about="#Parameter">
  <owl:disjointWith>
    <owl:Class rdf:about="#API"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Data_type"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:someValuesFrom>
        <owl:Class rdf:about="#Data_type"/>
      </owl:someValuesFrom>
      <owl:onProperty>
        <owl:FunctionalProperty rdf:ID="hasDataType"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:FunctionalProperty rdf:about="#hasDataType"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#OSThing"/>
  <owl:disjointWith rdf:resource="#OS_type"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#OS"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#API_standard"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#System_service"/>
  </owl:disjointWith>
</owl:Class>
<owl:Class rdf:about="#API_standard">
  <owl:disjointWith>
    <owl:Class rdf:about="#API"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#OS_type"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#OS"/>
  </owl:disjointWith>
```

```
    <owl:disjointWith>
      <owl:Class rdf:about="#System_service"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Parameter"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Data_type"/>
    </owl:disjointWith>
    <rdfs:subClassOf rdf:resource="#OSThing"/>
  </owl:Class>
  <owl:Class rdf:about="#Embedded-misc">
    <owl:disjointWith>
      <owl:Class rdf:about="#Windows"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Linux"/>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#OS"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="ANSI_C_service">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#System_service"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="ANSI_C_api">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#API"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Semaphore_api">
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#API"/>
          <owl:Restriction>
            <owl:someValuesFrom rdf:resource="#Semaphore_service"/>
            <owl:onProperty>
              <owl:ObjectProperty rdf:about="#provideService"/>
            </owl:onProperty>
          </owl:Restriction>
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
  </owl:Class>
  <owl:Class rdf:about="#Array_type">
    <owl:disjointWith>
      <owl:Class rdf:about="#Pointer_type"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Data_structure_type"/>
    <rdfs:subClassOf rdf:resource="#Compound_data_type"/>
  </owl:Class>
  <owl:Class rdf:ID="Thread_priority_value_parameter">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Value_parameter"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#Fundamental_data_type">
    <owl:disjointWith>
```

```
      <owl:Class rdf:about="#Defined_data_type"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Compound_data_type"/>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Data_type"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#Windows">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#OS"/>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Linux"/>
    <owl:disjointWith rdf:resource="#Embedded-misc"/>
  </owl:Class>
  <owl:Class rdf:ID="Mutex_api">
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#API"/>
          <owl:Restriction>
            <owl:onProperty>
              <owl:ObjectProperty rdf:about="#provideService"/>
            </owl:onProperty>
            <owl:someValuesFrom>
              <owl:Class rdf:about="#Mutex_service"/>
            </owl:someValuesFrom>
          </owl:Restriction>
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
  </owl:Class>
  <owl:Class rdf:about="#Timer_service">
    <owl:disjointWith>
      <owl:Class rdf:about="#Message_queue_service"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Mutex_service"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Semaphore_service"/>
    <owl:disjointWith rdf:resource="#Thread_service"/>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#System_service"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#Mutex_service">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#System_service"/>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:someValuesFrom rdf:resource="#Mutex_api"/>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasAPI"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith>
```

```
        <owl:Class rdf:about="#Message_queue_service"/>
      </owl:disjointWith>
      <owl:disjointWith rdf:resource="#Semaphore_service"/>
      <owl:disjointWith rdf:resource="#Thread_service"/>
      <owl:disjointWith rdf:resource="#Timer_service"/>
    </owl:Class>
    <owl:Class rdf:about="#Message_queue_service">
      <owl:disjointWith rdf:resource="#Mutex_service"/>
      <owl:disjointWith rdf:resource="#Semaphore_service"/>
      <owl:disjointWith rdf:resource="#Thread_service"/>
      <owl:disjointWith rdf:resource="#Timer_service"/>
      <rdfs:subClassOf>
        <owl:Class rdf:about="#System_service"/>
      </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="#System_service">
      <owl:disjointWith rdf:resource="#Parameter"/>
      <rdfs:subClassOf rdf:resource="#OSThing"/>
      <owl:disjointWith>
        <owl:Class rdf:about="#OS"/>
      </owl:disjointWith>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:someValuesFrom>
            <owl:Class rdf:about="#API"/>
          </owl:someValuesFrom>
          <owl:onProperty>
            <owl:ObjectProperty rdf:about="#hasAPI"/>
          </owl:onProperty>
        </owl:Restriction>
      </rdfs:subClassOf>
      <owl:disjointWith>
        <owl:Class rdf:about="#Data_type"/>
      </owl:disjointWith>
      <owl:disjointWith>
        <owl:Class rdf:about="#API"/>
      </owl:disjointWith>
      <owl:disjointWith rdf:resource="#API_standard"/>
      <owl:disjointWith rdf:resource="#OS_type"/>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >1</owl:minCardinality>
          <owl:onProperty>
            <owl:ObjectProperty rdf:about="#hasAPI"/>
          </owl:onProperty>
        </owl:Restriction>
      </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="#Defined_data_type">
      <rdfs:subClassOf>
        <owl:Class rdf:about="#Data_type"/>
      </rdfs:subClassOf>
      <owl:disjointWith rdf:resource="#Fundamental_data_type"/>
      <owl:disjointWith rdf:resource="#Compound_data_type"/>
    </owl:Class>
    <owl:Class rdf:about="#Address_parameter">
      <owl:disjointWith>
        <owl:Class rdf:about="#Value_parameter"/>
      </owl:disjointWith>
      <rdfs:subClassOf rdf:resource="#Parameter"/>
    </owl:Class>
    <owl:Class rdf:about="#OS">
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty>
            <owl:FunctionalProperty rdf:ID="hasOSType"/>
          </owl:onProperty>
          <owl:someValuesFrom rdf:resource="#OS_type"/>
        </owl:Restriction>
      </rdfs:subClassOf>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty>
            <owl:ObjectProperty rdf:about="#hasAPI"/>
          </owl:onProperty>
          <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >1</owl:minCardinality>
        </owl:Restriction>
      </rdfs:subClassOf>
      <owl:disjointWith rdf:resource="#System_service"/>
      <owl:disjointWith>
        <owl:Class rdf:about="#API"/>
      </owl:disjointWith>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty>
            <owl:FunctionalProperty rdf:about="#hasOSType"/>
          </owl:onProperty>
          <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >1</owl:cardinality>
        </owl:Restriction>
      </rdfs:subClassOf>
      <owl:disjointWith rdf:resource="#Parameter"/>
      <rdfs:subClassOf rdf:resource="#OSThing"/>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >1</owl:cardinality>
          <owl:onProperty>
            <owl:DatatypeProperty rdf:ID="hasName"/>
          </owl:onProperty>
        </owl:Restriction>
      </rdfs:subClassOf>
      <owl:disjointWith rdf:resource="#API_standard"/>
      <owl:disjointWith>
        <owl:Class rdf:about="#Data_type"/>
      </owl:disjointWith>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty>
            <owl:ObjectProperty rdf:about="#hasAPI"/>
          </owl:onProperty>
          <owl:someValuesFrom>
```

```
            <owl:Class rdf:about="#API"/>
         </owl:someValuesFrom>
      </owl:Restriction>
   </rdfs:subClassOf>
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty>
            <owl:DatatypeProperty rdf:about="#hasName"/>
         </owl:onProperty>
         <owl:allValuesFrom rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
      </owl:Restriction>
   </rdfs:subClassOf>
   <owl:disjointWith rdf:resource="#OS_type"/>
</owl:Class>
<owl:Class rdf:ID="Conditional_variable_api">
   <owl:equivalentClass>
      <owl:Class>
         <owl:intersectionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#API"/>
            <owl:Restriction>
               <owl:onProperty>
                  <owl:ObjectProperty rdf:about="#provideService"/>
               </owl:onProperty>
               <owl:someValuesFrom rdf:resource="#Mutex_service"/>
            </owl:Restriction>
         </owl:intersectionOf>
      </owl:Class>
   </owl:equivalentClass>
</owl:Class>
<owl:Class rdf:about="#API">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty>
            <owl:DatatypeProperty rdf:about="#hasName"/>
         </owl:onProperty>
         <owl:allValuesFrom rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
      </owl:Restriction>
   </rdfs:subClassOf>
   <owl:disjointWith rdf:resource="#System_service"/>
   <owl:disjointWith rdf:resource="#OS"/>
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty>
            <owl:FunctionalProperty rdf:ID="hasAPIStandard"/>
         </owl:onProperty>
         <owl:someValuesFrom rdf:resource="#API_standard"/>
      </owl:Restriction>
   </rdfs:subClassOf>
   <owl:disjointWith rdf:resource="#OS_type"/>
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:someValuesFrom rdf:resource="#System_service"/>
         <owl:onProperty>
            <owl:ObjectProperty rdf:about="#provideService"/>
         </owl:onProperty>
      </owl:Restriction>
   </rdfs:subClassOf>
```

```
<owl:disjointWith rdf:resource="#API_standard"/>
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty>
      <owl:FunctionalProperty rdf:ID="hasReturnType"/>
    </owl:onProperty>
    <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
    >1</owl:cardinality>
  </owl:Restriction>
</rdfs:subClassOf>
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty>
      <owl:ObjectProperty rdf:ID="hasParameter"/>
    </owl:onProperty>
    <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
    >1</owl:minCardinality>
  </owl:Restriction>
</rdfs:subClassOf>
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty>
      <owl:ObjectProperty rdf:about="#hasParameter"/>
    </owl:onProperty>
    <owl:someValuesFrom rdf:resource="#Parameter"/>
  </owl:Restriction>
</rdfs:subClassOf>
<rdfs:subClassOf rdf:resource="#OSThing"/>
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty>
      <owl:ObjectProperty rdf:ID="definedInOS"/>
    </owl:onProperty>
    <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
    >1</owl:cardinality>
  </owl:Restriction>
</rdfs:subClassOf>
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:someValuesFrom>
      <owl:Class rdf:about="#Data_type"/>
    </owl:someValuesFrom>
    <owl:onProperty>
      <owl:FunctionalProperty rdf:about="#hasReturnType"/>
    </owl:onProperty>
  </owl:Restriction>
</rdfs:subClassOf>
<owl:disjointWith>
  <owl:Class rdf:about="#Data_type"/>
</owl:disjointWith>
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty>
      <owl:ObjectProperty rdf:about="#definedInOS"/>
    </owl:onProperty>
    <owl:someValuesFrom rdf:resource="#OS"/>
  </owl:Restriction>
```

```
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:about="#hasName"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:FunctionalProperty rdf:about="#hasAPIStandard"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Parameter"/>
  </owl:Class>
  <owl:Class rdf:ID="Conditional_variable_service">
    <rdfs:subClassOf rdf:resource="#System_service"/>
  </owl:Class>
  <owl:Class rdf:about="#Pointer_type">
    <owl:disjointWith rdf:resource="#Data_structure_type"/>
    <owl:disjointWith rdf:resource="#Array_type"/>
    <rdfs:subClassOf rdf:resource="#Compound_data_type"/>
  </owl:Class>
  <owl:Class rdf:about="#Float_type">
    <owl:disjointWith rdf:resource="#Void_type"/>
    <owl:disjointWith rdf:resource="#Integer_type"/>
    <owl:disjointWith rdf:resource="#Char_type"/>
    <owl:disjointWith rdf:resource="#Bool_type"/>
    <rdfs:subClassOf rdf:resource="#Fundamental_data_type"/>
  </owl:Class>
  <owl:Class rdf:about="#Data_type">
    <rdfs:subClassOf rdf:resource="#OSThing"/>
    <owl:disjointWith rdf:resource="#API_standard"/>
    <owl:disjointWith rdf:resource="#Parameter"/>
    <owl:disjointWith rdf:resource="#API"/>
    <owl:disjointWith rdf:resource="#System_service"/>
    <owl:disjointWith rdf:resource="#OS_type"/>
    <owl:disjointWith rdf:resource="#OS"/>
  </owl:Class>
  <owl:Class rdf:about="#Value_parameter">
    <owl:disjointWith rdf:resource="#Address_parameter"/>
    <rdfs:subClassOf rdf:resource="#Parameter"/>
  </owl:Class>
  <owl:ObjectProperty rdf:about="#hasParameter">
    <rdfs:range rdf:resource="#Parameter"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:about="#definedInOS">
    <rdfs:range rdf:resource="#OS"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:about="#provideService">
    <rdfs:range rdf:resource="#System_service"/>
```

```
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasAPI">
  <rdfs:range rdf:resource="#API"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:about="#hasName">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
<owl:FunctionalProperty rdf:about="#hasDataType">
  <rdfs:range rdf:resource="#Data_type"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:about="#hasOSType">
  <rdfs:range rdf:resource="#OS_type"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:about="#hasAPIStandard">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  <rdfs:range rdf:resource="#API_standard"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:about="#hasReturnType">
  <rdfs:range rdf:resource="#Data_type"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
</owl:FunctionalProperty>
<Message_queue_api rdf:ID="tx_queue_flush">
  <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >tx_queue_flush</hasName>
  <hasReturnType>
    <Integer_type rdf:ID="unsigned_int"/>
  </hasReturnType>
  <definedInOS>
    <Embedded-misc rdf:ID="ThreadX">
      <hasAPI>
        <Timer_api rdf:ID="tx_time_get">
          <hasAPIStandard>
            <API_standard rdf:ID="NONPOSIX"/>
          </hasAPIStandard>
          <hasReturnType>
            <Integer_type rdf:ID="unsigned_long_int"/>
          </hasReturnType>
          <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >tx_time_get</hasName>
          <definedInOS rdf:resource="#ThreadX"/>
          <provideService>
            <Timer_service rdf:ID="Timer_service_time_get"/>
          </provideService>
        </Timer_api>
      </hasAPI>
      <hasAPI>
        <Message_queue_api rdf:ID="tx_queue_front_send">
          <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >tx_queue_front_send</hasName>
          <hasReturnType rdf:resource="#unsigned_int"/>
          <hasAPIStandard rdf:resource="#NONPOSIX"/>
          <definedInOS rdf:resource="#ThreadX"/>
        </Message_queue_api>
      </hasAPI>
      <hasAPI>
        <Timer_api rdf:ID="tx_time_set">
          <definedInOS rdf:resource="#ThreadX"/>
          <hasReturnType>
            <Void_type rdf:ID="void"/>
          </hasReturnType>
          <provideService>
            <Timer_service rdf:ID="Timer_service_time_set"/>
          </provideService>
          <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >tx_time_set</hasName>
          <hasAPIStandard rdf:resource="#NONPOSIX"/>
        </Timer_api>
      </hasAPI>
      <hasAPI>
        <Timer_api rdf:ID="tx_timer_deactivate">
          <definedInOS rdf:resource="#ThreadX"/>
          <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >tx_timer_deactivate</hasName>
          <hasAPIStandard rdf:resource="#NONPOSIX"/>
          <hasReturnType rdf:resource="#unsigned_int"/>
        </Timer_api>
      </hasAPI>
      <hasAPI>
        <Thread_api rdf:ID="tx_thread_relinquish">
          <definedInOS rdf:resource="#ThreadX"/>
          <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >tx_thread_relinquish</hasName>
          <hasReturnType rdf:resource="#void"/>
          <hasAPIStandard rdf:resource="#NONPOSIX"/>
        </Thread_api>
      </hasAPI>
      <hasAPI>
        <Timer_api rdf:ID="tx_timer_change">
          <definedInOS rdf:resource="#ThreadX"/>
          <hasAPIStandard rdf:resource="#NONPOSIX"/>
          <hasReturnType rdf:resource="#unsigned_int"/>
          <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >tx_timer_change</hasName>
        </Timer_api>
      </hasAPI>
      <hasAPI>
        <Message_queue_api rdf:ID="tx_queue_create">
          <provideService>
            <Message_queue_service rdf:ID="Message_queue_service_queue_create"/>
          </provideService>
          <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >tx_queue_create</hasName>
          <hasReturnType rdf:resource="#unsigned_int"/>
          <definedInOS rdf:resource="#ThreadX"/>
          <hasAPIStandard rdf:resource="#NONPOSIX"/>
        </Message_queue_api>
      </hasAPI>
      <hasAPI>
        <Thread_api rdf:ID="tx_thread_priority_change">
          <hasReturnType rdf:resource="#unsigned_int"/>
          <hasAPIStandard rdf:resource="#NONPOSIX"/>
          <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
```

```
            >tx_thread_priority_change</hasName>
            <definedInOS rdf:resource="#ThreadX"/>
            <hasParameter>
               <Thread_address_parameter rdf:ID="tx_threadPointer_thread_ptr">
                  <hasDataType>
                     <Pointer_type rdf:ID="tx_thread_pointer"/>
                  </hasDataType>
               </Thread_address_parameter>
            </hasParameter>
         </Thread_api>
      </hasAPI>
      <hasAPI>
         <Message_queue_api rdf:ID="tx_queue_prioritize">
            <hasReturnType rdf:resource="#unsigned_int"/>
            <hasAPIStandard rdf:resource="#NONPOSIX"/>
            <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >tx_queue_prioritize</hasName>
            <definedInOS rdf:resource="#ThreadX"/>
         </Message_queue_api>
      </hasAPI>
      <hasAPI>
         <Thread_api rdf:ID="tx_thread_info_get">
            <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >tx_thread_info_get</hasName>
            <hasParameter rdf:resource="#tx_threadPointer_thread_ptr"/>
            <hasReturnType rdf:resource="#unsigned_int"/>
            <hasAPIStandard rdf:resource="#NONPOSIX"/>
            <definedInOS rdf:resource="#ThreadX"/>
         </Thread_api>
      </hasAPI>
      <hasAPI>
         <Mutex_api rdf:ID="tx_mutex_get">
            <hasAPIStandard rdf:resource="#NONPOSIX"/>
            <hasReturnType rdf:resource="#unsigned_int"/>
            <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >tx_mutex_get</hasName>
            <definedInOS rdf:resource="#ThreadX"/>
         </Mutex_api>
      </hasAPI>
      <hasAPI>
         <Timer_api rdf:ID="tx_timer_create">
            <hasReturnType rdf:resource="#unsigned_int"/>
            <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >tx_timer_create</hasName>
            <definedInOS rdf:resource="#ThreadX"/>
            <hasAPIStandard rdf:resource="#NONPOSIX"/>
         </Timer_api>
      </hasAPI>
      <hasAPI>
         <Thread_api rdf:ID="tx_thread_resume">
            <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >tx_thread_resume</hasName>
            <definedInOS rdf:resource="#ThreadX"/>
            <hasAPIStandard rdf:resource="#NONPOSIX"/>
            <hasReturnType rdf:resource="#unsigned_int"/>
            <hasParameter rdf:resource="#tx_threadPointer_thread_ptr"/>
         </Thread_api>
```

```
</hasAPI>
<hasAPI>
  <Thread_api rdf:ID="tx_thread_delete">
    <hasReturnType rdf:resource="#unsigned_int"/>
    <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >tx_thread_delete</hasName>
    <hasAPIStandard rdf:resource="#NONPOSIX"/>
    <hasParameter rdf:resource="#tx_threadPointer_thread_ptr"/>
    <provideService>
      <Thread_service rdf:ID="Thread_service_delete">
        <hasAPI>
          <Thread_api rdf:ID="rtl_pthread_delete_np">
            <provideService rdf:resource="#Thread_service_delete"/>
            <hasParameter>
              <Thread_address_parameter rdf:ID="rtl_threadPointer_thread">
                <hasDataType>
                  <Pointer_type rdf:ID="pthread_t_pointer"/>
                </hasDataType>
              </Thread_address_parameter>
            </hasParameter>
            <hasAPIStandard rdf:resource="#NONPOSIX"/>
            <hasReturnType>
              <Integer_type rdf:ID="int"/>
            </hasReturnType>
            <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_pthread_delete_np</hasName>
            <definedInOS>
              <Linux rdf:ID="RTLinux">
                <hasAPI>
                  <Thread_api rdf:ID="rtl_pthread_suspend_np">
                    <definedInOS rdf:resource="#RTLinux"/>
                    <hasReturnType rdf:resource="#int"/>
                    <hasName
                      rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                    >rtl_pthread_suspend_np</hasName>
                    <hasParameter rdf:resource="#rtl_threadPointer_thread"/>
                    <hasAPIStandard rdf:resource="#NONPOSIX"/>
                  </Thread_api>
                </hasAPI>
                <hasAPI>
                  <Conditional_variable_api rdf:ID="rtl_pthread_cond_init">
                    <hasName
                      rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                    >rtl_pthread_cond_init</hasName>
                    <provideService>
                      <Conditional_variable_service
                        rdf:ID="Conditional_variable_service_cond_create"/>
                    </provideService>
                    <hasAPIStandard>
                      <API_standard rdf:ID="POSIX"/>
                    </hasAPIStandard>
                    <definedInOS rdf:resource="#RTLinux"/>
                    <hasReturnType rdf:resource="#int"/>
                  </Conditional_variable_api>
                </hasAPI>
                <hasAPI>
                  <Thread_api rdf:ID="rtl_pthread_selfe">
            <hasAPIStandard rdf:resource="#POSIX"/>
            <hasReturnType rdf:resource="#pthread_t_pointer"/>
            <hasName
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_pthread_selfe</hasName>
            <definedInOS rdf:resource="#RTLinux"/>
            <hasParameter rdf:resource="#void"/>
        </Thread_api>
    </hasAPI>
    <hasOSType>
        <OS_type rdf:ID="OS_type_RTOS"/>
    </hasOSType>
    <hasAPI>
        <Timer_api rdf:ID="rtl_usleep">
            <hasAPIStandard rdf:resource="#POSIX"/>
            <definedInOS rdf:resource="#RTLinux"/>
            <hasName
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_usleep</hasName>
        </Timer_api>
    </hasAPI>
    <hasAPI>
        <Thread_api rdf:ID="rtl_pthread_exit">
            <hasAPIStandard rdf:resource="#POSIX"/>
            <hasName
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_pthread_exit</hasName>
            <hasReturnType rdf:resource="#void"/>
            <definedInOS rdf:resource="#RTLinux"/>
        </Thread_api>
    </hasAPI>
    <hasAPI>
        <Semaphore_api rdf:ID="rtl_sem_destroy">
            <hasReturnType rdf:resource="#int"/>
            <definedInOS rdf:resource="#RTLinux"/>
            <hasAPIStandard rdf:resource="#POSIX"/>
            <provideService>
                <Semaphore_service
                  rdf:ID="Semaphore_service_semaphore_delete"/>
            </provideService>
            <hasName
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_sem_destroy</hasName>
        </Semaphore_api>
    </hasAPI>
    <hasAPI>
        <Mutex_api rdf:ID="rtl_pthread_mutex_trylock">
            <hasReturnType rdf:resource="#int"/>
            <hasAPIStandard rdf:resource="#POSIX"/>
            <hasName
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_pthread_mutex_trylock</hasName>
            <definedInOS rdf:resource="#RTLinux"/>
        </Mutex_api>
    </hasAPI>
    <hasAPI>
        <Mutex_api rdf:ID="rtl_pthread_mutex_unlock">
```

```
            <hasName
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_pthread_mutex_unlock</hasName>
            <hasAPIStandard rdf:resource="#POSIX"/>
            <definedInOS rdf:resource="#RTLinux"/>
            <hasReturnType rdf:resource="#int"/>
          </Mutex_api>
        </hasAPI>
        <hasAPI>
          <Thread_api rdf:ID="rtl_pthread_join">
            <hasName
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_pthread_join</hasName>
            <hasParameter rdf:resource="#rtl_threadPointer_thread"/>
            <definedInOS rdf:resource="#RTLinux"/>
            <hasAPIStandard rdf:resource="#POSIX"/>
            <hasReturnType rdf:resource="#int"/>
          </Thread_api>
        </hasAPI>
        <hasAPI>
          <Mutex_api rdf:ID="rtl_phtread_mutex_lock">
            <hasName
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_phtread_mutex_lock</hasName>
            <hasReturnType rdf:resource="#int"/>
            <definedInOS rdf:resource="#RTLinux"/>
            <hasAPIStandard rdf:resource="#POSIX"/>
          </Mutex_api>
        </hasAPI>
        <hasAPI>
          <Conditional_variable_api rdf:ID="rtl_pthread_cond_wait">
            <definedInOS rdf:resource="#RTLinux"/>
            <hasReturnType rdf:resource="#int"/>
            <hasAPIStandard rdf:resource="#POSIX"/>
            <hasName
               rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_pthread_cond_wait</hasName>
          </Conditional_variable_api>
        </hasAPI>
        <hasAPI>
          <Thread_api rdf:ID="rtl_pthread_kill">
            <hasParameter rdf:resource="#rtl_threadPointer_thread"/>
            <hasName
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_pthread_kill</hasName>
            <definedInOS rdf:resource="#RTLinux"/>
            <hasReturnType rdf:resource="#int"/>
            <hasAPIStandard rdf:resource="#POSIX"/>
          </Thread_api>
        </hasAPI>
        <hasAPI>
          <Thread_api rdf:ID="rtl_pthread_create">
            <hasName
               rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_pthread_create</hasName>
            <hasParameter rdf:resource="#rtl_threadPointer_thread"/>
            <hasParameter>
```

```
  <Thread_attr_address_parameter
  rdf:ID="rtl_threadAttrPointer_attr"/>
</hasParameter>
<definedInOS rdf:resource="#RTLinux"/>
<hasReturnType rdf:resource="#int"/>
<hasAPIStandard rdf:resource="#POSIX"/>
<provideService>
  <Thread_service rdf:ID="Thread_service_create">
    <hasAPI rdf:resource="#rtl_pthread_create"/>
    <hasAPI>
      <Thread_api rdf:ID="tx_thread_create">
        <hasAPIStandard rdf:resource="#NONPOSIX"/>
        <provideService rdf:resource="#Thread_service_create"/>
        <hasReturnType rdf:resource="#unsigned_int"/>
        <hasParameter
          rdf:resource="#tx_threadPointer_thread_ptr"/>
        <hasName
       rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >tx_thread_create</hasName>
        <definedInOS rdf:resource="#ThreadX"/>
      </Thread_api>
    </hasAPI>
    <hasAPI>
      <Thread_api rdf:ID="win32_CreateThread">
        <definedInOS>
          <Windows rdf:ID="Windows_windowsNT">
            <hasAPI>
              <Thread_api rdf:ID="win32_TerminateThread">
                <definedInOS
                 rdf:resource="#Windows_windowsNT"/>
                <hasName rdf:datatype=
                "http://www.w3.org/2001/XMLSchema#string"
                >win32_TermintateThread</hasName>
                <provideService>
                  <Thread_service
                    rdf:ID="Thread_service_terminate">
                    <hasAPI>
                      <Thread_api
                        rdf:ID="tx_thread_terminate">
                        <hasAPIStandard
                          rdf:resource="#NONPOSIX"/>
                        <hasName rdf:datatype=
                 "http://www.w3.org/2001/XMLSchema#string"
                        >tx_thread_terminate</hasName>
                        <hasReturnType
                         rdf:resource="#unsigned_int"/>
                        <hasParameter
                 rdf:resource="#tx_threadPointer_thread_ptr"/>
                        <provideService
                   rdf:resource="#Thread_service_terminate"/>
                        <definedInOS
                        rdf:resource="#ThreadX"/>
                      </Thread_api>
                    </hasAPI>
                  </Thread_service>
                </provideService>
                <hasAPIStandard>
```

```
                                                    <API_standard rdf:ID="WIN32"/>
                                                  </hasAPIStandard>
                                                </Thread_api>
                                              </hasAPI>
                                              <hasAPI rdf:resource="#win32_CreateThread"/>
                                              <hasOSType>
                                                <OS_type rdf:ID="OS_type_NONRTOS"/>
                                              </hasOSType>
                                              <hasName rdf:datatype=
                                              "http://www.w3.org/2001/XMLSchema#string"
                                              >WindowNT</hasName>
                                            </Windows>
                                          </definedInOS>
                                          <hasParameter>
                                            <Thread_address_parameter
                                                rdf:ID="WIN32_threadPointer_lpStartAddress">
                                              <hasDataType rdf:resource="#unsigned_long_int"/>
                                            </Thread_address_parameter>
                                          </hasParameter>
                                          <hasName
                                         rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                                          >win32_CreateThread</hasName>
                                          <hasReturnType>
                                            <Integer_type rdf:ID="HANDLE"/>
                                          </hasReturnType>
                                          <provideService rdf:resource="#Thread_service_create"/>
                                          <hasAPIStandard rdf:resource="#WIN32"/>
                                        </Thread_api>
                                      </hasAPI>
                                    </Thread_service>
                                  </provideService>
                                </Thread_api>
                              </hasAPI>
                              <hasAPI>
                                <Conditional_variable_api rdf:ID="rtl_pthread_cond_timedwait">
                                  <hasName
                                      rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                                  >rtl_pthread_cond_timedwait</hasName>
                                  <hasReturnType rdf:resource="#int"/>
                                  <hasAPIStandard rdf:resource="#POSIX"/>
                                  <definedInOS rdf:resource="#RTLinux"/>
                                </Conditional_variable_api>
                              </hasAPI>
                              <hasAPI>
                                <Timer_api rdf:ID="rtl_nanosleep">
                                  <definedInOS rdf:resource="#RTLinux"/>
                                  <hasReturnType rdf:resource="#int"/>
                                  <hasName
                                    rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                                  >rtl_nanosleep</hasName>
                                  <hasAPIStandard rdf:resource="#POSIX"/>
                                </Timer_api>
                              </hasAPI>
                              <hasAPI>
                                <Timer_api rdf:ID="rtl_clock_settime">
                                  <provideService rdf:resource="#Timer_service_time_set"/>
                                  <definedInOS rdf:resource="#RTLinux"/>
```

```
            <hasAPIStandard rdf:resource="#POSIX"/>
            <hasReturnType rdf:resource="#int"/>
            <hasName
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_clock_settime</hasName>
        </Timer_api>
    </hasAPI>
    <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >RTLinux</hasName>
    <hasAPI>
        <Thread_api rdf:ID="rtl_pthread_wakeup_np">
            <hasParameter rdf:resource="#rtl_threadPointer_thread"/>
            <hasAPIStandard rdf:resource="#NONPOSIX"/>
            <hasName
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_pthread_wakeup_np</hasName>
            <definedInOS rdf:resource="#RTLinux"/>
            <hasReturnType rdf:resource="#int"/>
        </Thread_api>
    </hasAPI>
    <hasAPI>
        <Conditional_variable_api rdf:ID="rtl_pthread_cond_signal">
            <hasReturnType rdf:resource="#int"/>
            <hasName
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_pthread_cond_signal</hasName>
            <hasAPIStandard rdf:resource="#POSIX"/>
            <definedInOS rdf:resource="#RTLinux"/>
        </Conditional_variable_api>
    </hasAPI>
    <hasAPI>
        <Conditional_variable_api rdf:ID="rtl_pthread_cond_destroy">
            <definedInOS rdf:resource="#RTLinux"/>
            <hasReturnType rdf:resource="#int"/>
            <hasAPIStandard rdf:resource="#POSIX"/>
            <hasName
              rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_pthread_cond_destroy</hasName>
        </Conditional_variable_api>
    </hasAPI>
    <hasAPI rdf:resource="#rtl_pthread_delete_np"/>
    <hasAPI>
        <Mutex_api rdf:ID="rtl_phtread_mutex_destroy">
            <hasName
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >rtl_phtread_mutex_destroy</hasName>
            <definedInOS rdf:resource="#RTLinux"/>
            <hasAPIStandard rdf:resource="#POSIX"/>
            <hasReturnType rdf:resource="#int"/>
            <provideService>
                <Mutex_service rdf:ID="Mutex_service_destroy">
                    <hasAPI rdf:resource="#rtl_phtread_mutex_destroy"/>
                </Mutex_service>
            </provideService>
        </Mutex_api>
    </hasAPI>
    <hasAPI>
```

```
        <Thread_api rdf:ID="rtl_pthread_yield">
          <definedInOS rdf:resource="#RTLinux"/>
          <hasAPIStandard rdf:resource="#POSIX"/>
          <hasName
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >rtl_pthread_yield</hasName>
          <hasReturnType rdf:resource="#int"/>
        </Thread_api>
      </hasAPI>
      <hasAPI>
        <Conditional_variable_api rdf:ID="rtl_pthread_cond_broadcast">
          <definedInOS rdf:resource="#RTLinux"/>
          <hasReturnType rdf:resource="#int"/>
          <hasAPIStandard rdf:resource="#POSIX"/>
          <hasName
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >rtl_pthread_cond_broadcast</hasName>
        </Conditional_variable_api>
      </hasAPI>
      <hasAPI>
        <Semaphore_api rdf:ID="rtl_sem_init">
          <provideService>
            <Semaphore_service
            rdf:ID="Semaphore_service_semaphore_create"/>
          </provideService>
          <hasReturnType rdf:resource="#int"/>
          <hasName
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >rtl_sem_init</hasName>
          <hasAPIStandard rdf:resource="#POSIX"/>
          <definedInOS rdf:resource="#RTLinux"/>
        </Semaphore_api>
      </hasAPI>
      <hasAPI>
        <Thread_api rdf:ID="rtl_pthread_setfp_np">
          <hasReturnType rdf:resource="#int"/>
          <definedInOS rdf:resource="#RTLinux"/>
          <hasName
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >rtl_pthread_setfp_np</hasName>
          <hasAPIStandard rdf:resource="#NONPOSIX"/>
          <hasParameter rdf:resource="#rtl_threadPointer_thread"/>
        </Thread_api>
      </hasAPI>
      <hasAPI>
        <Mutex_api rdf:ID="rtl_pthread_mutex_init">
          <provideService>
            <Mutex_service rdf:ID="Mutex_service_init">
              <hasAPI>
                <Mutex_api rdf:ID="tx_mutex_create">
                  <definedInOS rdf:resource="#ThreadX"/>
                  <hasAPIStandard rdf:resource="#NONPOSIX"/>
                  <hasName
                 rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                  >tx_mutex_create</hasName>
                  <provideService rdf:resource="#Mutex_service_init"/>
                </Mutex_api>
              </hasAPI>
              <hasAPI rdf:resource="#rtl_pthread_mutex_init"/>
            </Mutex_service>
          </provideService>
          <definedInOS rdf:resource="#RTLinux"/>
          <hasName
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >rtl_pthread_mutex_init</hasName>
          <hasAPIStandard rdf:resource="#POSIX"/>
          <hasReturnType rdf:resource="#int"/>
        </Mutex_api>
      </hasAPI>
      <hasAPI>
        <Thread_api rdf:ID="rtl_pthread_make_priodic_np">
          <hasName
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >rtl_pthread_make_priodic_np</hasName>
          <hasParameter rdf:resource="#rtl_threadPointer_thread"/>
          <definedInOS rdf:resource="#RTLinux"/>
          <hasAPIStandard rdf:resource="#NONPOSIX"/>
          <hasReturnType rdf:resource="#int"/>
        </Thread_api>
      </hasAPI>
      <hasAPI>
        <Thread_api rdf:ID="rtl_pthread_wait_np">
          <hasAPIStandard rdf:resource="#NONPOSIX"/>
          <hasReturnType rdf:resource="#int"/>
          <definedInOS rdf:resource="#RTLinux"/>
          <hasName
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >rtl_pthread_wait_np</hasName>
          <hasParameter rdf:resource="#void"/>
        </Thread_api>
      </hasAPI>
      <hasAPI>
        <Thread_api rdf:ID="rtl_pthread_cancel">
          <hasParameter rdf:resource="#rtl_threadPointer_thread"/>
          <hasAPIStandard rdf:resource="#POSIX"/>
          <hasReturnType rdf:resource="#int"/>
          <hasName
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >rtl_pthread_cancel</hasName>
          <definedInOS rdf:resource="#RTLinux"/>
        </Thread_api>
      </hasAPI>
      <hasAPI>
        <Mutex_api rdf:ID="rtl_pthread_timedlock">
          <hasReturnType rdf:resource="#int"/>
          <hasAPIStandard rdf:resource="#POSIX"/>
          <hasName
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >rtl_pthread_timedlock</hasName>
          <definedInOS rdf:resource="#RTLinux"/>
        </Mutex_api>
      </hasAPI>
      <hasAPI>
        <Timer_api rdf:ID="rtl_time">
```

```
                              <hasName
                                rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                              >rtl_time</hasName>
                              <hasAPIStandard rdf:resource="#POSIX"/>
                              <definedInOS rdf:resource="#RTLinux"/>
                            </Timer_api>
                          </hasAPI>
                        </Linux>
                      </definedInOS>
                    </Thread_api>
                  </hasAPI>
                  <hasAPI rdf:resource="#tx_thread_delete"/>
                </Thread_service>
              </provideService>
              <definedInOS rdf:resource="#ThreadX"/>
            </Thread_api>
          </hasAPI>
          <hasAPI>
            <Semaphore_api rdf:ID="tx_semaphore_prioritize">
              <hasAPIStandard rdf:resource="#NONPOSIX"/>
              <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
              >tx_semaphore_prioritize</hasName>
              <definedInOS rdf:resource="#ThreadX"/>
              <hasReturnType rdf:resource="#unsigned_int"/>
            </Semaphore_api>
          </hasAPI>
          <hasAPI rdf:resource="#tx_thread_terminate"/>
          <hasAPI>
            <Mutex_api rdf:ID="tx_mutex_prioritize">
              <definedInOS rdf:resource="#ThreadX"/>
              <hasAPIStandard rdf:resource="#NONPOSIX"/>
              <hasReturnType rdf:resource="#unsigned_int"/>
              <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
              >tx_mutex_prioritize</hasName>
            </Mutex_api>
          </hasAPI>
          <hasAPI>
            <Semaphore_api rdf:ID="tx_semaphore_put">
              <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
              >tx_semaphore_put</hasName>
              <hasReturnType rdf:resource="#unsigned_int"/>
              <hasAPIStandard rdf:resource="#NONPOSIX"/>
              <definedInOS rdf:resource="#ThreadX"/>
            </Semaphore_api>
          </hasAPI>
          <hasAPI>
            <Message_queue_api rdf:ID="tx_queue_send">
              <hasReturnType rdf:resource="#unsigned_int"/>
              <hasAPIStandard rdf:resource="#NONPOSIX"/>
              <definedInOS rdf:resource="#ThreadX"/>
              <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
              >tx_queue_send</hasName>
            </Message_queue_api>
          </hasAPI>
          <hasAPI>
            <Message_queue_api rdf:ID="tx_queue_delete">
              <definedInOS rdf:resource="#ThreadX"/>
```

```
        <hasAPIStandard rdf:resource="#NONPOSIX"/>
        <hasReturnType rdf:resource="#unsigned_int"/>
        <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >tx_queue_delete</hasName>
        <provideService>
          <Message_queue_service rdf:ID="Message_queue_service_queue_delete"/>
        </provideService>
      </Message_queue_api>
    </hasAPI>
    <hasAPI>
      <Timer_api rdf:ID="tx_timer_delete">
        <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >tx_timer_delete</hasName>
        <hasReturnType rdf:resource="#unsigned_int"/>
        <hasAPIStandard rdf:resource="#NONPOSIX"/>
        <definedInOS rdf:resource="#ThreadX"/>
      </Timer_api>
    </hasAPI>
    <hasAPI>
      <Mutex_api rdf:ID="tx_mutex_put">
        <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >tx_mutex_put</hasName>
        <definedInOS rdf:resource="#ThreadX"/>
        <hasReturnType rdf:resource="#unsigned_int"/>
        <hasAPIStandard rdf:resource="#NONPOSIX"/>
      </Mutex_api>
    </hasAPI>
    <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >ThreadX</hasName>
    <hasAPI>
      <Mutex_api rdf:ID="tx_mutex_info_get">
        <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >tx_mutex_info_get</hasName>
        <hasAPIStandard rdf:resource="#NONPOSIX"/>
        <definedInOS rdf:resource="#ThreadX"/>
        <hasReturnType rdf:resource="#unsigned_int"/>
      </Mutex_api>
    </hasAPI>
    <hasAPI>
      <Message_queue_api rdf:ID="tx_queue_receive">
        <definedInOS rdf:resource="#ThreadX"/>
        <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >tx_queue_receive</hasName>
        <hasReturnType rdf:resource="#unsigned_int"/>
        <hasAPIStandard rdf:resource="#NONPOSIX"/>
      </Message_queue_api>
    </hasAPI>
    <hasAPI>
      <Timer_api rdf:ID="tx_timer_activate">
        <hasReturnType rdf:resource="#unsigned_int"/>
        <hasAPIStandard rdf:resource="#NONPOSIX"/>
        <definedInOS rdf:resource="#ThreadX"/>
        <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >tx_timer_activate</hasName>
      </Timer_api>
    </hasAPI>
    <hasAPI rdf:resource="#tx_thread_create"/>
```

```
<hasAPI>
  <Message_queue_api rdf:ID="tx_queue_info_get">
    <hasReturnType rdf:resource="#unsigned_int"/>
    <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >tx_queue_info_get</hasName>
    <hasAPIStandard rdf:resource="#NONPOSIX"/>
    <definedInOS rdf:resource="#ThreadX"/>
  </Message_queue_api>
</hasAPI>
<hasAPI rdf:resource="#tx_queue_flush"/>
<hasAPI>
  <Thread_api rdf:ID="tx_thread_preemption_change">
    <hasAPIStandard rdf:resource="#NONPOSIX"/>
    <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >tx_thread_preemption_change</hasName>
    <hasReturnType rdf:resource="#unsigned_int"/>
    <hasParameter rdf:resource="#tx_threadPointer_thread_ptr"/>
    <definedInOS rdf:resource="#ThreadX"/>
  </Thread_api>
</hasAPI>
<hasAPI>
  <Thread_api rdf:ID="tx_thread_wait_abort">
    <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >tx_thread_wait_abort</hasName>
    <hasParameter rdf:resource="#tx_threadPointer_thread_ptr"/>
    <hasAPIStandard rdf:resource="#NONPOSIX"/>
    <hasReturnType rdf:resource="#unsigned_int"/>
    <definedInOS rdf:resource="#ThreadX"/>
  </Thread_api>
</hasAPI>
<hasAPI>
  <Semaphore_api rdf:ID="tx_semaphore_delete">
    <provideService rdf:resource="#Semaphore_service_semaphore_delete"/>
    <hasAPIStandard rdf:resource="#NONPOSIX"/>
    <hasReturnType rdf:resource="#unsigned_int"/>
    <definedInOS rdf:resource="#ThreadX"/>
    <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >tx_semaphore_delete</hasName>
  </Semaphore_api>
</hasAPI>
<hasAPI>
  <Semaphore_api rdf:ID="tx_semaphore_get">
    <definedInOS rdf:resource="#ThreadX"/>
    <hasAPIStandard rdf:resource="#NONPOSIX"/>
    <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >tx_semaphore_get</hasName>
    <hasReturnType rdf:resource="#unsigned_int"/>
  </Semaphore_api>
</hasAPI>
<hasAPI>
  <Thread_api rdf:ID="tx_thread_suspend">
    <hasReturnType rdf:resource="#unsigned_int"/>
    <hasAPIStandard rdf:resource="#NONPOSIX"/>
    <hasParameter rdf:resource="#tx_threadPointer_thread_ptr"/>
    <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >tx_thread_suspend</hasName>
    <definedInOS rdf:resource="#ThreadX"/>
```

```
        </Thread_api>
    </hasAPI>
    <hasAPI>
        <Semaphore_api rdf:ID="tx_semaphore_create">
            <definedInOS rdf:resource="#ThreadX"/>
            <provideService rdf:resource="#Semaphore_service_semaphore_create"/>
            <hasAPIStandard rdf:resource="#NONPOSIX"/>
            <hasReturnType rdf:resource="#unsigned_int"/>
            <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >tx_semaphore_create</hasName>
        </Semaphore_api>
    </hasAPI>
    <hasAPI>
        <Timer_api rdf:ID="tx_timer_info_get">
            <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >tx_timer_info_get</hasName>
            <definedInOS rdf:resource="#ThreadX"/>
            <hasAPIStandard rdf:resource="#NONPOSIX"/>
            <hasReturnType rdf:resource="#unsigned_int"/>
        </Timer_api>
    </hasAPI>
    <hasAPI>
        <Thread_api rdf:ID="tx_thread_sleep">
            <hasReturnType rdf:resource="#unsigned_int"/>
            <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >tx_thread_sleep</hasName>
            <definedInOS rdf:resource="#ThreadX"/>
            <hasAPIStandard rdf:resource="#NONPOSIX"/>
        </Thread_api>
    </hasAPI>
    <hasAPI>
        <Mutex_api rdf:ID="tx_mutex_delete">
            <definedInOS rdf:resource="#ThreadX"/>
            <hasAPIStandard rdf:resource="#NONPOSIX"/>
            <provideService rdf:resource="#Mutex_service_destroy"/>
            <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >tx_mutex_delete</hasName>
        </Mutex_api>
    </hasAPI>
    <hasOSType rdf:resource="#OS_type_RTOS"/>
    <hasAPI rdf:resource="#tx_mutex_create"/>
    <hasAPI>
        <Thread_api rdf:ID="tx_thread_time_slice_change">
            <hasReturnType rdf:resource="#unsigned_int"/>
            <hasParameter rdf:resource="#tx_threadPointer_thread_ptr"/>
            <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >tx_thread_time_slice_change</hasName>
            <definedInOS rdf:resource="#ThreadX"/>
            <hasAPIStandard rdf:resource="#NONPOSIX"/>
        </Thread_api>
    </hasAPI>
    <hasAPI>
        <Semaphore_api rdf:ID="tx_semaphore_info_get">
            <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >tx_semaphore_info_get</hasName>
            <hasReturnType rdf:resource="#unsigned_int"/>
            <hasAPIStandard rdf:resource="#NONPOSIX"/>
```

```
              <definedInOS rdf:resource="#ThreadX"/>
            </Semaphore_api>
          </hasAPI>
        </Embedded-misc>
      </definedInOS>
      <hasAPIStandard rdf:resource="#NONPOSIX"/>
    </Message_queue_api>
    <Array_type rdf:ID="array"/>
    <Integer_type rdf:ID="unsigned"/>
    <Linux rdf:ID="Ubuntu">
      <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >Ubuntu</hasName>
      <hasOSType rdf:resource="#OS_type_NONRTOS"/>
    </Linux>
    <Linux rdf:ID="RedHat9">
      <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >RedHat9</hasName>
      <hasOSType rdf:resource="#OS_type_NONRTOS"/>
    </Linux>
    <ANSI_C_service rdf:ID="ANSI_C_service_print"/>
    <Timer_api rdf:ID="rtl_clock_getres">
      <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >rtl_clock_getres</hasName>
      <definedInOS rdf:resource="#RTLinux"/>
      <hasAPIStandard rdf:resource="#POSIX"/>
      <hasReturnType rdf:resource="#int"/>
    </Timer_api>
    <Char_type rdf:ID="char"/>
    <Data_structure_type rdf:ID="tx_thread"/>
    <Pointer_type rdf:ID="pthread_attr_t_pointer"/>
    <Integer_type rdf:ID="short_int"/>
    <Float_type rdf:ID="float"/>
    <Windows rdf:ID="Windows_windowsXP">
      <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >WindowsXP</hasName>
      <hasOSType rdf:resource="#OS_type_NONRTOS"/>
    </Windows>
    <Thread_attr_address_parameter rdf:ID="WIN32_threadAttrPointer_lpThreadAttributes"/>
    <ANSI_C_api rdf:ID="printf">
      <provideService rdf:resource="#ANSI_C_service_print"/>
      <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >printf</hasName>
    </ANSI_C_api>
    <Bool_type rdf:ID="bool"/>
    <Timer_api rdf:ID="rtl_clock_gettime">
      <definedInOS rdf:resource="#RTLinux"/>
      <hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >rtl_clock_gettime</hasName>
      <provideService rdf:resource="#Timer_service_time_get"/>
      <hasAPIStandard rdf:resource="#POSIX"/>
      <hasReturnType rdf:resource="#int"/>
    </Timer_api>
    <Float_type rdf:ID="double"/>
    <Float_type rdf:ID="long_double"/>
    <Integer_type rdf:ID="long_int"/>
</rdf:RDF>
<!-- Created with Protege (with OWL Plugin 3.3, Build 399)    http://protege.stanford.edu -->
```

Appendix B List of Publications

[1] Hong Zhou, Hongji Yang and Andrew Hugill, "An Ontology-Based Approach to Reengineering Enterprise Software for Cloud Computing", 34rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'10), Seoul, Korea, Jul. 2010, pp. 383-388.

[2] Feng Chen, Hong Zhou, Riumin Liu, Hongji Yang, He Guo and Yuxin Wang, "An Ontology-based Approach to Portable Embedded System Development", *21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, Boston, USA, Jul. 2009, pp. 569-574.

[3] Zihou Zhou, Hong Zhou and Hongji Yang, "Evaluating Websites Using A Practical Quality Model", *14th International Conference on Automation & Computing Society in the UK (ICAC'08)* , London, England, Sept. 2008, pp. 114-119.

[4] Feng Chen, Hongji Yang, Hong Zhou and Bing Qiao, "Web-based System Evolution in Model Driven Architecture", *10th IEEE International Symposium on Web Site Evolution (WSE 2008)*, Beijing, China, Oct. 2008, pp. 69-72.

[5] Hong Zhou, "COSS: Comprehension by Ontologising Software System", *24th IEEE International Conference on Software Maintenance (ICSM'08)*, Beijing, China, Sep. 2008, pp. 432-435.

[6] Hong Zhou, Feng Chen and Hongji Yang, "Developing Application Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology", *8th International Conference on Quality Software (QSIC'08)*, Oxford, UK, Aug. 2008, pp. 225-234.

[7] Hong Zhou, Jian Kang, Feng Chen and Hongji Yang, "OPTIMA: an Ontology-based PlaTform-specIfic software Migration Approach", *7th International Conference on Quality Software (QSIC'07)*, Portland, Oregon, USA, Oct. 2007, pp. 143-152.

[8] Jian Kang, Hong Zhou and Hongji Yang, "Task Decomposition for Communication Computation Overlap to Reengineer a Web-Based System", *11th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'07)*, Arizona, USA., Mar. 2007, pp. 205-212.