FAKULTÄT FÜR !NFORMATIK

Data Modeling of Multi-agent Systems

A comparison of UML-based and Ontology-based approaches with special focus on didactic skills for ontology-based modeling

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Mag.rer.soc.oec

im Rahmen des Studiums

Informatikmanagement

eingereicht von

Min Liang

Matrikelnummer 0335324

an der

Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer/Betreuerin: Univ.-Prof. Dr. Stefan Biffl
Mitwirkung: Univ.-Ass. Mag. Thomas Moser

Wien, 06.05.2009 _____ _____

(Unterschrift Verfasserin) (Unterschrift Betreuer)

Abstract

Data modeling, the process of creating a data model by applying a data model theory to create a data model instance, always plays a crucial role in software engineering. Both UML (Unified Modeling Language) and ontology are important data modeling languages which correspond to data model theories in the field of software engineering and knowledge engineering. The target of audiences of this thesis are model engineers and software engineers who are interested in data modeling using either UML or ontologies, as well as at software engineers with knowledge in the traditional data modeling area who want to analyze the advantages and possible limitations of switching to a fairly new data modeling approach.

UML is defined by the OMG as a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to describe a system's blueprints, including conceptual elements such as business processes and system functions as well as concrete elements such as programming language statements, database schemes, and reusable software components.

An ontology is a formal representation of a set of concepts within a domain and the relationships between those concepts. It is used to reason about the properties of that domain, and may be used to define the domain. Three general ontology languages are defined in context of semantic web, namely OWL (Web ontology language), RDF (Resource description framework), RDF Schema. Ontologies are widely used in artificial intelligence and the semantic web, but a relative new research area regarding software engineering.

The goals of this thesis are a) to introduce the primary principles of UML and Ontology; b) to present overviews of standard tools like Visual Paradigm for UML and Protégé for ontologies; c) to show major differences between these two modeling approaches regarding a use case scenario from multi-agent based production automation simulation; d) The four major research issues of this thesis are to evaluate the difference between data modeling using either UML or ontologies, compare their dissimilar model consistency checking capabilities, research the possibility of mapping UML to OWL, and explore the revolution of UML and OWL; and e) to suggest didactics skills for ontology-based modeling teaching improvement. In order to answer these research issues, the overall data modeling process is performed using an example in the field of multi-agent systems for production automation simulation by means of an UML-based approach and an ontology-based approach. After the implementation of those two approaches, both approaches are evaluated regarding their visualization and expressions, consistency, performance and additional functions approximately. This evaluation characterizes and appraises the general features, advantages and limitations of UML and Ontology respectively, and additionally a detailed evaluation result is presented.

Zusammenfassung

Datenmodellierung ist der Prozess der Herstellung eines Datenmodells unter Verwendung von Datenmodellierungstheorien. UML und OWL sind wichtige Datenmodellierungssprachen bzw. Datenmodellierungstheorien im Bereich von Software Engineering und Knowledge Engineering. Die Zielgruppe für diese Arbeit sind Software und Modell Entwickler, die ihre Erfahrung mit traditionellen Datenmodellierungsmethoden vertiefen möchten, indem sie die Vorteile und möglichen Einschränkungen eines neuen, Ontologie-basierten Datenmodellierungsverfahrens analysieren.

UML wurde als eine graphische Sprache für die Visualisierung, Spezifikation, Gestaltung und Dokumentation der Artefakte von Software-intensiven System definiert. UML bietet ein standardisiertes Verfahren zur Beschreibung der Ausarbeitung eines Systems. Dies beinhaltet konzeptionelle Elemente, z.B. Geschäftsprozesse und Systemfunktionen, auch konkrete Elemente, z.B. Statements in Programmcode, Datenbank Schemas und wiederverwendbare Software Komponenten.

Eine Ontologie ist eine formale Darstellung einer Reihe von Konzepten innerhalb einer Domäne und der Beziehungen zwischen diesen Konzepten. Ontologien werden verwendet um die logische Folgerungen über die Eigenschaften einer Domäne abzuleiten und können verwendet werden um eine Domäne zu definieren. Ontologies sind weitverbreitet im Bereich künstliche Intelligenz und Semantic Web, aber auch in einem relativ neuen Forschungsbereich hinsichtlich Software Engineering verwendbar.

Die Ziele dieser Arbeit sind a) die Einführung in die Grundprinzipien von UML und Ontologien; b) ein Überblick über Standard Werkzeuge für die beiden Datenmodellierungssprachen, nämlich Visual Paradigm für UML und Protégé für Ontologien; c) das Herausarbeiten von grundsätzlichen Unterschieden zwischen beiden Modellierungsansätzen hinsichtlich eines Anwendungsszenarios aus dem Bereich der Multi-Agenten basierten Simulation von Produktionsautomatisierungssystemen; d) die Evaluation der Unterschiede zwischen Datenmodellierung mit UML oder Ontologien; e) der Vergleich ihrer jeweiligen unterschiedlichen Prüfungsfähigkeiten von Modellkonsistenz; f) zukünftige Verbesserungen bzw. Erweiterungen von UML und OWL zu untersuchen; und g) didaktische Methoden für die Verbesserung der Lehre im Bereich Ontologien vorzuschlagen.

Zur Beantwortung der Forschungsfragen wird ein umfassender Datenmodellierungsprozess im Bereich der Multi-Agenten basierten Simulation von Produktionsautomatisierungssystemen mit UML und Ontologien durchgeführt. Beide Ansätze werden hinsichtlich ihrer Visualisierungsmöglichkeiten und Ausdrücke, Konsistenz, Leistung und zusätzlicher Funktionen evaluiert. Diese Evaluation charakterisiert und bewertet die allgemeine Eigenschaften, bzw. Vorteile und Einschränkungen von UML und Ontologien in einem detaillierten Evaluationsergebnis.

Eidesstattliche Erklärung

Ich versichere, dass ich die beiliegende Masterarbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlich oder sinngemäß übernommenen Stellen in der Arbeit gekennzeichnet habe.

Unterschrift

Table of Contents

List of Figures

List of Tables

1. Introduction

The first section introduces a brief motivation of the whole project and states the importance and primary principles of data modeling based on UML and Ontology. Additionally, it represents the outline of the thesis structure.

Motivation

High quality software is not only built to meet customer's requirements; but also an "artifact" with high reliability, stability and extendibility. Moreover, on-time and on-budget delivery is always a key point for every software developer and project manager to consider.

Software quality has to be controlled not only at the beginning of whole software lifecycle, but also be checked in multiple phases during the whole software lifecycle. As emphasized in many software engineering books, a more clear, accurate and detailed customer requirements analysis will more likely lead to a final success. In order to achieve this, it is very important to have a special tool for every developer to build the bridge between real world and digital systems before the real development starts.

One of the most popular ways to achieve this goal is the usage of data modeling. Data modeling is the process of creating a data model by applying a data model theory to create a data model instance, it often used to define and analyze data requirements for business processes of a system [5]. It acts as a framework and offers a better platform to create opportunities for simplification, reduction of monitoring and better risk management [11] [41].

A data model theory represents the formal description of the way to structure and store the data, it usually contains three components, the structural part represents the main data structure, such as entities and objects which are required to model databases; the integrity part represents the constraint rules which integrate the main data structure structurally; the manipulation part represents the operations which can be used to update and query the data of the structural part in the database [22].

The data model instance is always applying the data model theory in order to create a concrete data model instance for some certain applications [2]. Each data model instance has three types normally, conceptual data model, logical data model and physical data model. The conceptual data model represents the domain concepts of business requirements, contains mainly entities, attributes and relationships between entities within a domain. The logical data model represents the technology of data manipulation, such as data tables, XML files etc. The physical data model represents the physical structure of database that stores the data, such as CPU etc [2].

This thesis aims at model engineers and software engineers who are

interested in data modeling using either UML or ontologies, as well as for software engineers with knowledge in the traditional data modeling area who want to analyse the advantages and possible limitations of switching to a fairly new data modeling approach. This thesis provides both model engineers and software engineers to learn the theories and practical experiments of UML and ontology-based data modeling, engineers can benefit from getting the research evaluation results of UML and ontologies in various aspects in order to integrate and employ two data modeling approaches more wisely and accurately.

There exist several data modeling languages and support tools based on diverse data model theories. Both UML (Unified Modeling Language) and ontology are important data modeling languages correspond to data model theories in the field of software engineering and knowledge engineering. The UML (Unified Modeling Language) is a standardized general-purpose modeling language defined by the OMG in the field of software engineering [2]. It offers an object-oriented way to write a system's blueprints, including conceptual elements such as business processes and system functions as well as concrete elements such as programming language statements, database schemas, and reusable software components [3] [11]. UML is very easy and good at modeling and documenting the system, well understandable for software engineers to read and denote, but it lacks of formality which makes it hard for machine to process and ensure models consistent automatically.

Another popular approach for data modeling is ontology. The concept of ontologies in computer science is defined as a formal representation of a set of concepts within a domain and the relationships between those concepts. Ontology languages are formal languages used to construct ontologies. They allow the encoding of knowledge about specific domains and often include reasoning rules that support the processing of that knowledge [27]. Three general ontology languages are created in context of semantic web, like OWL (Web ontology language), RDF (Resource description framework)/RDF Schema, DAML+OIL which are the current and prevenient releases of ontology languages. Ontology has a large logical expressiveness and is a well formal specification language for building domain knowledge, enables automated validation and consistency checking in the field of knowledge engineering, nevertheless, its complex and formal representations are often difficult for engineers to learn and understand [12].

In this work, the scenario is to establish two data models of automation production processes for different kinds of distributed agents in a manufacturing plant by means of UML and ontologies respectively. Suppose that a corresponding simulation system needs to be built for its monitoring and controlling, this production line could be used separately by each kind of various agents in this plant. At the first sight, the inner infrastructure and architecture seems to be very complex to simulate. Therefore and in order to

simplify the engineering process, it is necessary for designers to design a data model first before starting to build the production line.

Once the production line is built and its functions don't match the customer's requirements well, it would certainly lead to severe problems. It would be almost impossible to make any large changes again, even a slight modification would certainly cost much money and delay the overall delivery time. In the worst case, a complete redesign of the data model and rebuild of the production line is necessary. To summarize, with the help of the data modeling languages, this scenario could be better observed through analyzing all the differences of UML and ontologies.

The goals of this thesis are a) to introduce the primary principles of UML, such as meanings and notations of building elements, a set of diagram types etc and Ontology such as meanings of primary building elements, three sublanguages of ontology language and three variations of OWL etc b) to present overviews of standard tools like Visual Paradigm for UML, such as overviews of all functions and specific features of the Entity Relationship Diagram while Protégé for ontologies such as overviews of all functions and specific features of the Protégé-OWL editor for data modeling; c) to introduce the characteristics of MAS, the components of test management tool MAST and the similarities and differences of MAST and the employed tool SAW in this project, analyze the main responsibilities of each kind agent, and show major differences between these two data modeling approaches regarding the use case scenario from multi-agent based production automation simulation; d) the four major research issues of this thesis are to evaluate the differences between data modeling using either UML or ontologies, compare their dissimilar model consistency checking capabilities, research the possibility of mapping UML to OWL, and explore the revolution of UML and OWL; and e) to suggest didactics skills for ontology-based modeling teaching improvement, like the creation of each concrete lecture unit and propose an optimal method that can help the UML engineers to under and command OWL in an efficient way. In order to answer these research issues, the overall data modeling process is performed using an example in the field of multi-agent systems for production automation simulation by means of an UML-based approach and an ontology-based approach. After the implementation of those two approaches, both approaches are evaluated regarding their visualization and expressions, consistency, performance and additional functions approximately. This evaluation characterizes and appraises the general features, advantages and limitations of UML and Ontology respectively, and additionally a detailed evaluation result is presented.

Thesis Structure

The remainder of this thesis is structured as follows: Section 2 "Related work" introduces the primary principles of UML and Ontology, overviews of the standard tool Visual Paradigm for UML and Protégé for Ontologies, and

introduction to multi-agent system simulation. Section 3 "Research Issues" discusses and defines four meaningful and potential issues. Section 4 "Use case description" defines five involved roles of this example in the field of production automation and their main functions and properties. Section 5 "Data modeling" demonstrates the overall data modeling process of a concrete example in the field of production automation based on UML and ontology approaches. Section 6 "Evaluation" compares and analyzes their advantages and disadvantages respectively, as well as indicates a detailed evaluation result. Section 7 “Didactics in ontology-based modeling" places emphasis on the didactic teaching of Ontology in high schools and universities and an optimal way for conversion from UML diagrams to Ontology notations. Section 8 "Discussion" offers a feasible solution to the above inquired research issues. Finally, section 9 "Conclusion" concludes and gives a future outlook.

2. Related work

The second section presents the related work necessary for this project, such as importance and primary principles of data modeling and a brief introduction to object-oriented modeling and knowledge modeling etc which make through the way to understand UML and Ontology better and help to build foundation steps for further research issues.

2.1 Importance of Data Modeling

One principal reason for the usage of data modeling which has been mentioned before is to offer a simplification of the complicated reality and helps software engineers better understand the system which is going to be developed. Therefore, it is relevant to outline the importance of the data modeling before starting with this project. Following are the four main aims that designers and software engineers should achieve in data models in general.

System visualization

A diagram in data modeling provides a vital display of customer's requirements to developers, which makes it possible for them to evaluate how much the real system will satisfy the customer’s needs at an early stage of development.

For example, if we would like to decorate our house, we would probably like to draw a draft first, indicate the colors and each room style in order to make the artisan know our needs and favors more exactly [11].

In UML, there exist several structure diagrams which are used for system visualization: like Class diagram, Composite structure diagram, Deployment diagram, Object diagram and Package diagram etc. The main features of UML diagrams will be introduced in more details in section 2.5.3.

System specification

When software engineers consider building up the data model, some indispensable sections like system architecture, functional and non-functional behaviors of the whole software system should be cleared up. In this way, the data model can also be used as system specifications according to the customer’s requirements. It can also facilitate the customers to verify the system specifications easily. Any changes or improvements could be carried out on the basis of the data models [11].

System template

System templates, also called prototypes, are much closer to the real system. They provide a precise guidance in constructing a system. When data modeling is concrete enough, system template can be set up to help both software engineers and customers to minimize the gap of requirements understood between each side. For example, a stereoscopic globe supports travelers a much better guide than a flat world map [11].

Documentation

There is a saying "a picture is better than thousand words". Data modeling provides more clear and accurate information than words, even only with a simple diagram. Sometimes, there exists only one way to demonstrate results and decisions by means of diagrams or formal languages. This kind of formalization could offer a unique and accurate solution which could be understood by experts without a word of explanation [11].

2.2 Principles of Data Modeling

As the long history of applying data modeling suggests the following four principles which are helpful to create an accurate modeling in an efficient way.

Precise model selection

Model selection is one of the difficulties in data modeling. Different choices on model selections will lead to different impacts on the whole project. The appropriate model which software engineers choose, will offer their insight into the correct solution, even may help finding the solution of the most challenging development problems. Otherwise, the wrong chosen models will mislead software engineers far away from the success, spending more waste time in the wrong direction.

In software engineering, the chosen models can also present or greatly affect the software engineers' views. Suppose that during the development life cycle of the system, the choice of business analysts would probably be the use case diagrams and class diagram models, while the choice of object-oriented developers would be the object-oriented models for the system etc [11].

Abstraction

Creating data models in an abstract way means to top-down derive logical data models from a subject, which normally all people can understand. This kind of method is opposite to the way of bottom-up creation of data models. Bottom-up models are often observed as the result of a reengineering effort, which usually start with already existing data structures forms.

A system model may be created and expressed at different levels of

precision, starting with the higher levels and adding more levels with more detail as more is understood about the system.

The best kinds of models are that those could be viewed at several levels depending on different roles or different situations. They could only show some information necessary for a certain role, but hide all the other unnecessary information. For instance, an analyst or an end user will rather focus on specifications; while a developer wants to focus on realizations.

In our example, it is relevant to establish different models for different distributed agents which could execute different independent tasks in most cases. To merge the different tasks at the beginning of design will increase the complexity of the system and make the software developers more confused [11] [15].

Connection to the reality

The basic requirement of data modeling is to bridge the real world with the computer system. Therefore, the best data models should have a connection to the reality. Since data modeling should also abstract the concepts of the real world, or even more than representing the reality, it should also simplify the reality, anyway, it should make sure not to hide any important details of reality [11].

If the designed data models do not correspond to the reality, this means that the design is not feasible. It would make no more sense to carry out the building processes of the data models. In this project, a data model of all the involved roles in production automation system will be created. All basic information of the production automation as well as the special requirements should be ensured to be included into our consideration [11].

Structured Analysis

For a simple and trivial system, it is easy to create a corresponding and accurate model; nevertheless, for every nontrivial system is not easy to make it happen, the best approach is to combine a set of nearly independent models.

There are three main view types that could support a structured analysis of the system. The primary view is the functional view which consists of the architectural elements that specifies the system's functionalities, providing the primary structures of the solution, such as use case descriptions. The data view, also called a static structural view, consists of entity relationship diagrams, etc. The dynamic view consists of e.g. state chart diagrams, which defines for instance, what happens under certain conditions [15].

In this project, in order to understand the system architecture well, the primary view is required at least, primary view of each agent which helps exposing the requirements of the system.

2.3 Object-oriented Modeling

UML is one of the most powerful representative methods for object-oriented modeling. Here explains a close view of some benefits to object-oriented modeling.

In software engineering, there are two common approaches: either to create a model from an algorithmic perspective or from an object-oriented perspective.

The object-oriented perspective helps software engineers to address the complexity of a problem domain by considering the problem not as a set of functions that can be performed, but primarily as a set of related, interacting objects. The modeling task is specifying for a specific context, those objects and their respective set of properties and methods, shared by all objects members of the class [28].

The main building element of the object-oriented modeling is either an object or class. An object is a unique thing, generated from the vocabulary of the problem domain, each object has its unique identity; states represents the attributes of the object, and behavior represents the methods operated on the states of an object. A class is a description of a group of objects which have the same set of states and behaviors. The relationships among classes called class hierarchy and inheritance, should be also modeled [11] [28].

2.4 Knowledge Modeling

Knowledge modeling is a systematic approach of representing information and logic representations in a digitally reusable format for purpose of capturing, sharing and processing knowledge to simulate intelligence. Ontologies share or reuse the knowledge base that can be used as the basis for knowledge acquisition tools for gathering domain knowledge or for generating databases or expert systems.

Knowledge models contain three knowledge levels: task knowledge, inference knowledge and domain knowledge. Correspondingly, there are three steps to create knowledge modeling, i.e., knowledge identification, knowledge specification and knowledge refinement [1].

In knowledge identification which plays the preparation stage for realizing the customers' requirements, all useful information sources like the task knowledge and the domain knowledge should be identified. Since the task knowledge is often goal-oriented, potential functional components should be decomposited and listed in a hierarchical structure. In additional, domain schema and knowledge base of domain knowledge, such as domain types, domain rules and domain facts should be also determined [1].

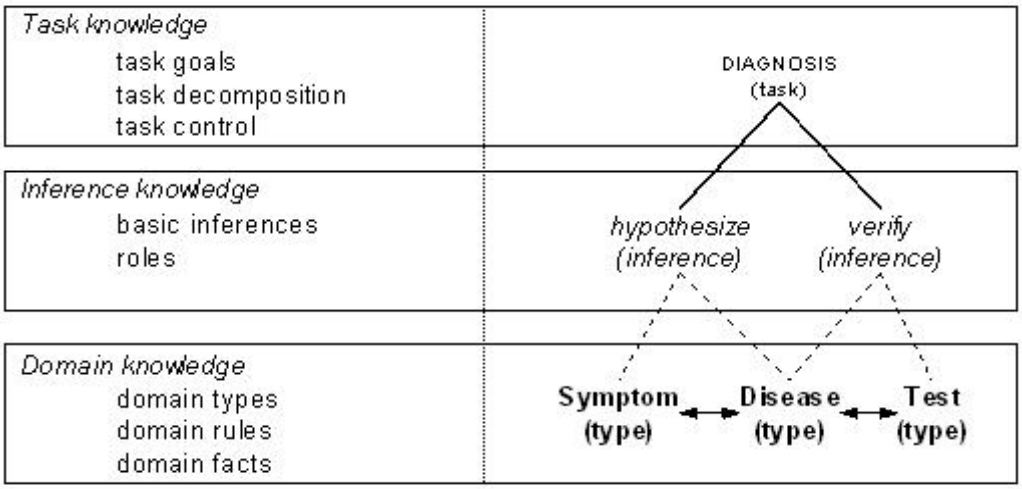

Figure 1: Overview of Knowledge Model [1]

Knowledge specification is defined as the construction of a specification of the knowledge model. The knowledge developer could start with the development of the task knowledge and domain knowledge, and later link both with the inference knowledge, which represents the basic information reasoning steps. The knowledge base representing the domain knowledge could be finished in the next step [1].

Knowledge refinement is defined as the step of validating and completing the knowledge model. The knowledge base of the domain knowledge should be completed by inserting a set of knowledge instances. The validation is to check whether the knowledge model could fulfill the defined goals by simulations [1].

2.5 Introduction to UML

UML is one of the most popular specifications issued by the Object Management Group (OMG). The following will introduce about UML's history, its definition, diagrams and other more detail knowledge.

2.5.1 History

In about 1990s, more than 50 methods appeared in the software market at that time, each of them has its own set of notations and processes. However, none of them was able to provide a complete satisfaction to users. In industry, people always would like to require a standard method and approach to analysis their requirements [25].

The development of UML started in late 1994, three designers were Grady Booch, Jim Rumbaugh, and Ivar Jacobson. They were trying to unify their three well recognized methods in the world at that time; which were

Rumbaugh's OMT (Object Modeling Technique), Grady Booch's Booch method, and Jacobson's OOSE (Object Oriented Software Engineering). Each method had its own value and emphasis, such as OMT was powerful in analysis aspect and weaker in the design, which was more suitable for object-oriented analysis (OOA). Booch's method was relative strong in design and weaker in analysis aspect, which was more suitable for object-oriented design (OOD). By contrast, OOSE was stronger in behavior analysis and had shortcomings in the other areas [25].

In 1996, a few organizations realized the large strategic value and impact of UML on their business. Later the Object Management Group (OMG) provided A Request for Proposal (RFP), and the achievement of UML version 1.0 was successful in 1997 [25].

The current version of UML 2.1.2 specification concludes two complimentary parts: the UML infrastructure specification defines the foundational language of a core meta model that specifies the abstract syntax of the UML, such as the set of UML modeling concepts, their attributes and relationships, as well as the combining rules. The UML Superstructure specification defines the notation and semantics for diagrams and their model elements, how the UML concepts are going to be realized by computers [5].

2.5.2 Definition

The official OMG (Object Management Group) proposed a standard and comprehensive definition for UML: "The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components" [24].

UML consists of three basic building blocks: Things, Relationships and Diagrams. Things are component parts of the UML, Relationships get the Things together and Diagrams are the mutual groups of related Things.

2.5.2.1 Things

Things are generally used to write well-formed models. There are four species of things basically extended from object-oriented models: Structural Things, Behavioral Things, Grouping Things and Annotational Things [11].

Structural things

The structural things represent the nouns of the UML models, play the static roles in a model, representing either the conceptual or physical elements

[11]. They are the most commonly used elements in data modeling, seven kinds of Structural Things totally.

- Class
  A class is a collection of a set of objects with the same attributes, operations, and relationships. This concept comes from Object-Oriented Analyzing and Object-Oriented Design. It focuses on the presentation of the basics attributes and the relationships of the objects in the real word, in additional abstract of the common objects. It is rendered as a rectangle with its name, attributes, and operations etc [11].
- Interface
  An interface gathers a set of operations that define a service of a class or components. An interface only specifies the concepts of a set of operations; these operations can only be implemented in a specified class. It is rendered as a circle along with its name [11].
- Collaboration
  A collaboration illustrates the cooperative interactions between each role in a society. Collaborations have both structural and behavioral dimensions in general. It is rendered as an ellipse with dashed lines, including its name [11].
- Use case
  A use case is a description of system functions, which generate results to a certain actor. It shows the sequence of actions of the system, without a detail internal system structure. The functions described by use case should be a complete process, and during the creation of use case, users can find undefined classes and precise function sequences, therefore creation of use case is a very important part in data modeling. In UML it is rendered as an ellipse with solid lines, including its name [11].
- Active class
  An active class contains active objects owning one or more threads that initiate the control activity. It is rendered like as a rectangle with heavy lines, including its name, attributes, and operations [11].
- Component
  A component is a modular and replaceable part of a system that encapsulates such as classes, interfaces and collaborations. Besides, Java Beans is a good example of Component. It is rendered as a rectangle with tabs, including its name [11].
- Node
  A node is a computational resource always with memory and processing capability that represents the physical elements applied at run time. A set of components can be interconnected through communication paths. It is rendered as a cube with its name [11].

Behavioral things

Behavioral things represent the verbs of UML models, play the dynamic roles in a model, representing behaviors over time and space. There are two species of Behavioral things: Interaction and State machine [11].

- Interaction
  An interaction is a behavior that depicts a few of messages communicated among objects in a defined society in order to achieve a certain task. It comprises other elements, such as messages, action sequences, and connections. It is rendered as a directed line with the name of its operation [11].
- State machine
  A state machine is a behavior that describes all sequences of an object or an interaction' states reacting to events within its lifetime. It comprises other elements, such as states, transitions, events, and activities. It is rendered as a rounded rectangle with its name and substates [11].

Grouping things

Grouping things play the connected roles of UML models. There is only one species Grouping things: Package [11].

- Package
  A package groups for elements or other packages with certain purposes packing into groups, providing a hierarchical order. A package can comprise Structural things, like classes, objects and use cases etc; Behavioral things, and other Grouping things. It is rendered as a tabbed folder, with its name and contents. Other variations of packages are existed, such as frameworks, models, and subsystems [11].

Annotational things

Annotational things play the interpretive roles of UML models. One species Annotational things: the Note can be applied for elements in a model to display with comments and constraints etc [11].

- Note
  A note is a symbol for an element to display or illustrate comments and its constraints with a textual or a graphical comment. It is rendered as a rectangle with a dog-eared corner [11].

2.5.2.2 Relationships

UML is especially good at describing relationships between classes. There are four kinds of relationships below: Dependency, Association, Generalization and Realization.

Dependency

A dependency is a semantic connection between two things, which includes class with class, package with package, use case with use case, model with model and so on. The change to the independent thing may lead to the change of the dependent thing. It is rendered as a dashed line, the directed arrowhead indicates the dependent thing [11].

Association

An association is a set of structural connections among objects, especially different parts inside component, class and objects. The structural relationship between a whole and its parts called aggregation. It is rendered as a solid line, with other notations in most cases, for example multiplicity and role names [11].

The difference between “Dependency” and “Association” is that objects with “Dependency” relationship still can exist without each other, but objects with “Association” cannot.

Generalization

A generalization is a generalization connection describes in which objects of the specialized element are substitutable for objects of the generalized element. It is rendered as a solid line with a hollow arrowhead pointing to the generalized element [11].

Realization

A realization is a semantic connection between two classifiers normally. One classifier specifies and assigns a task, so that another classifier is supposed to implement the assigned task. It is rendered as a dashed line with a hollow arrowhead pointing to the classifier implementer [11].

2.5.3 Diagram

A diagram provides the graphical notation of a set of components, groups interrelated collections of things and relationships, rendered as a connected graph of vertices (things) and arcs (relationships).

UML 2.0 has 13 types of diagrams that can be categorized hierarchically in

the following figure 2.

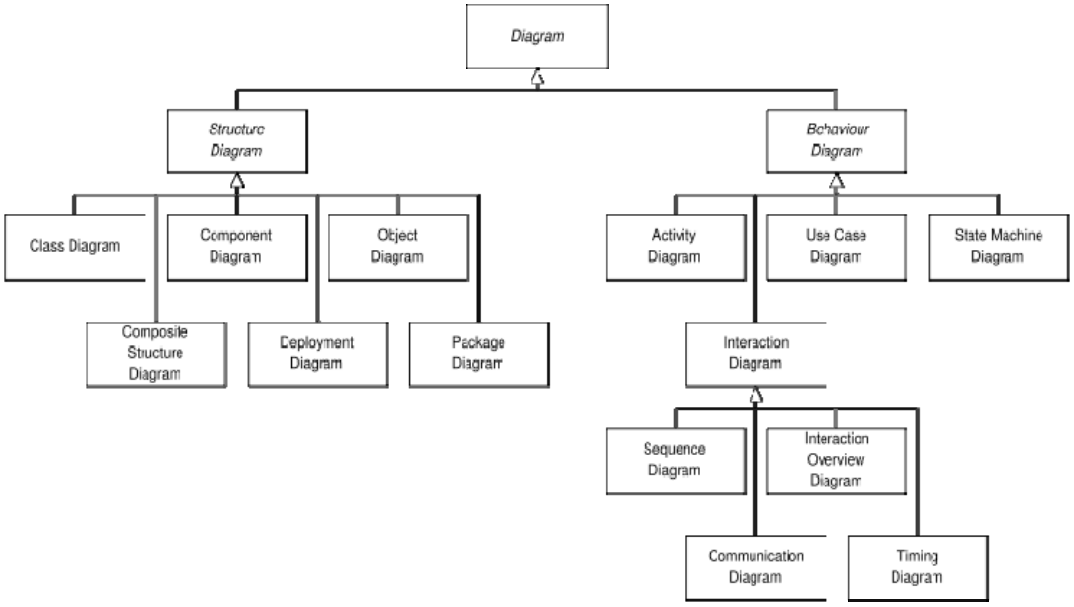

Figure 2: UML 2.0 Diagrams [3]

Six types of Structure Diagram represent static application structures; three types of Behaviour Diagram represent dynamic behaviours; and other four kinds of Interaction Diagram represent different aspects of interactions.

2.5.3.1 Structural Modeling Diagrams

Structure modeling diagrams are often used to define the static architecture of a system. They model all the elements that make up a system, for instance, the classes, attributes, interfaces and the relationships between elements [37].

- Class diagram
  Class diagrams define the classes, attributes, operations and relationships between classes in general that required to construct a system [37].
- Composite Structure diagram
  Composite Structure diagrams define an overview of an element's internal structure with special focus on its inner details, and relationships between variables [37].
- Component diagram
  Component diagrams define the components and their dependencies that consist of a complex system and drive the system run [37].
- Deployment diagram
  Deployment diagrams model the physical hardware, system

environment and other significant artifacts applied to real-world settings [37].

- ❖ Object diagram
  Object diagrams specify the dependencies within instances of a system's structure at a particular run-time [37].
- ❖ Package diagram
  Package diagrams define the dependencies among the Grouping things by means of dividing the system into logical packages [37].

2.5.3.2 Behavioral Modeling Diagrams

Behavioral modeling diagrams comprise the behavioral features of functionalities and business process among components [37].

- ❖ Activity diagram
  Activity diagrams define the overall business workflow of the components in a system, including the significant decision points and actions [37].
- ❖ Use Case diagram
  Use Case diagrams define the functionalities and relationships of a set of actors, including requirements and constraints in the context of scenarios [37].
- ❖ State Machine diagram
  State Machine diagrams define the possible states or events of a model's behaviors and the specific conditions or transitions that may trigger the variation of states [37].

2.5.3.3 Interactive modeling Diagrams

Interactive modeling diagrams are a subset of behavioral modeling diagrams, with the focus on tracking the workflow of control and interactions among the components [3].

- ❖ Sequence diagram
  Sequence diagrams define the communicative sequences of messages among components in accordance with their life spans [37].
- ❖ Communication diagram
  Communication diagrams define the communicative sequences of messages among components at run-time, providing a combination of classes, sequence and use case diagrams [3] [37].
- ❖ Interaction Overview diagram
  Interaction Overview diagrams combine each activity diagram with the decision points in a workflow [37].
- ❖ Timing diagram

Timing diagrams are a kind of interaction overview diagrams, providing especially the timing constraints of a component and the corresponding interactions [3] [37].

2.5.4 Architecture

UML provides various perspectives for analyzing system architecture. Among them there are five main views, such as Use case view, Design view, Process view, Implementation view and Deployment view. Each view has its own special focus on the view point of the system's structure.

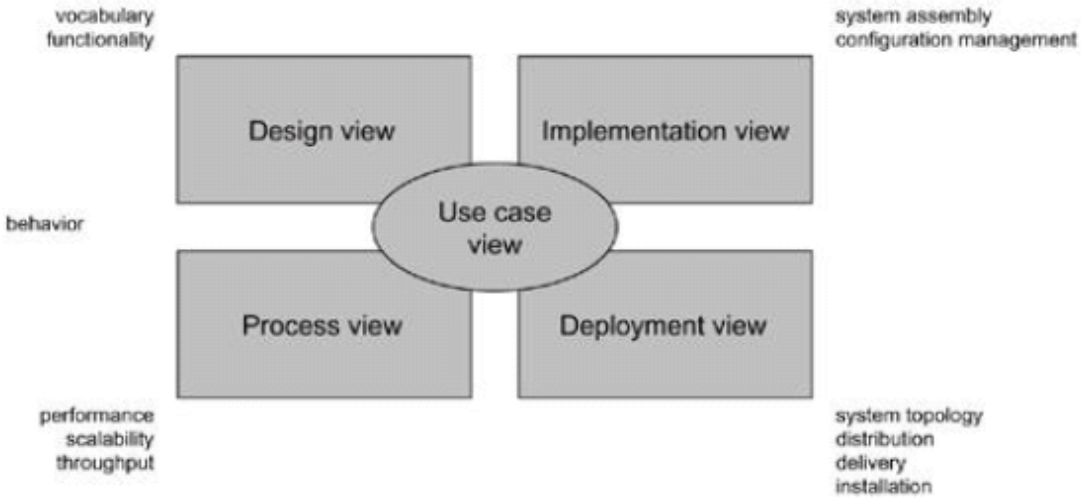

Figure 3: A system's Architecture [11]

Use case view

The use case view of a system provides its end users, analysts and testers an overview of use cases that describe the architecture and functionalities of the system. A use case view always illustrates a typical interaction between a user and a software system, captures a functionality of the system to be used by a user, and shows a typical user goal planed to be achieved.

A use case view may contain use cases, actors, classes, use class diagrams and object diagrams for demonstration of its static aspects; state diagrams, sequence diagrams and collaboration diagrams for demonstration of its dynamic aspects [11].

Design view

The design view of a system collects all the parts of elements, such as classes, interfaces and so on that form the vocabulary of the requirements of the system and its proposals. This view primarily generalizes the functional requirements of the system required by its end users planed to be realized.

A design view may contain class diagrams, and object diagrams for demonstration of its static aspects; state diagrams, interaction diagrams and

activity diagrams for the demonstration of its dynamic aspects [11].

Process view

The process view of a system contains the threads and processes that ensure the system's concurrency and synchronization mechanisms. This view primarily guarantees the performance, scalability, and maximal throughput of the system.

A process case view is familiar with the design view, but emphasizes on the active classes that depict the threads and processes [11].

Implementation view

The implementation view of a system depicts the way how components and code files are gathered to assemble the physical system. This view focuses on the configuration management of the system mainly, variations to combine independent components together working on the system.

An implementation view may use component diagrams for demonstration of its static aspects; interaction diagrams and state diagrams for demonstration of its dynamic aspects in common [11].

Deployment view

The deployment view of a system collects the nodes that construct the system's hardware topology. This view illustrates the connections of various devices or parts of the system involved in the environment of the physical system installed.

A deployment view may contain nodes, artifacts and deployment diagrams for demonstration of its static aspects; interaction diagrams and state diagrams are usually used for demonstration of its dynamic aspects [11].

2.5.5 Class Diagram

Since class diagram plays a relevant role in UML diagrams and will be applied for a wide range of pragmatic applications in data modeling. Here will present a brief introduction of the class diagram.

A class diagram is a special kind of structural modeling diagrams, it shows the static structure of the system. The essential elements of the class diagram include the system's classes, their attributes, operations and relationships between classes.

A class is usually made up of three elements, a class name, attributes, and operations. The first stack of the class diagram's rectangle is the class name. The second stack is for attributes that state the data properties of the classes, including attribute names, type, default value and visibility. The third stack contains operations that depict the functionalities for the objects of the classes,

including operation names, parameters names, parameter types, parameter visibilities, and return types [8].

There are several kinds of relationships, for example, instance level relationship includes external links, association, aggregation, and composition etc; class level relationship includes generalization and realization etc; general relationship includes dependency and multiplicity [8].

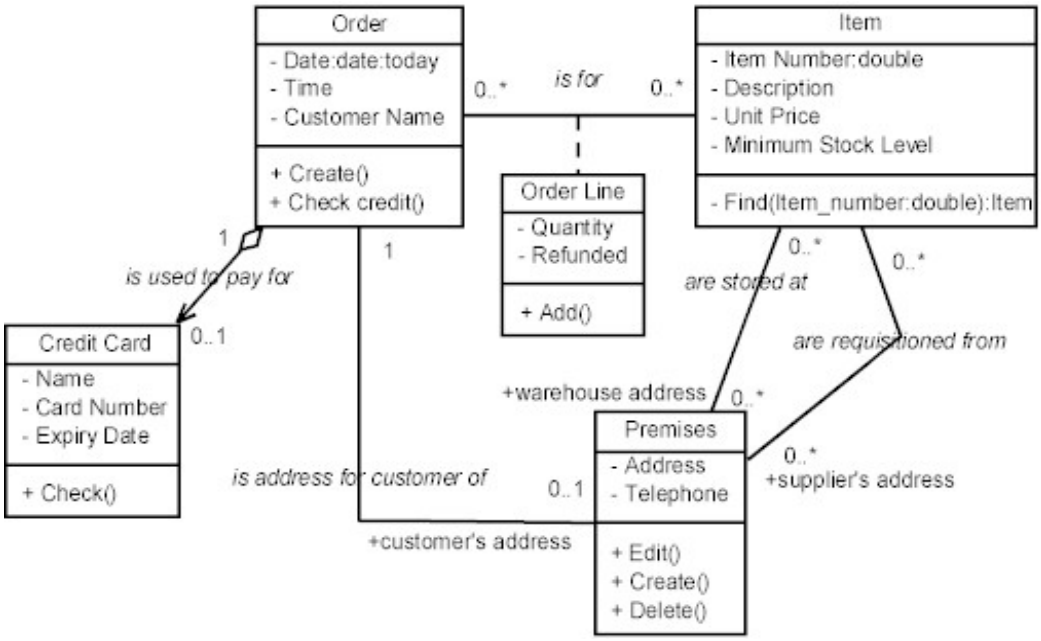

Figure 4: An example of class diagram [5]

2.6 Introduction to UML Tool Visual Paradigm

Visual Paradigm is a powerful and very popular UML tool, I have chosen this tool to implement the project. Therefore here will present a concise overview of Visual Paradigm and features of creating ER (Entity-Relationship) diagrams.

2.6.1 Overview

Visual Paradigm is a powerful, cross-platform and yet the most easy-to-use visual UML modeling and CASE tool. It is especially designed for the following actors, such as software engineers, system analysts, business process analysts, and system architects. Visual Paradigm is developed by Visual Paradigm International Ltd. from Hong Kong, China, and it is becoming more and more popular all over the world with a rapid growth these years [43].

This tool mainly focuses on providing a reliable data modeling and analysis tool for object-oriented system. Visual Paradigm supports the latest Java standard and UML diagrams, moreover, it can be integrated with other

software develop tools, such as Eclipse and IBM WebSphere.

In the new version of Visual Paradigm, users can use custom picture to replace the traditional UML symbols, O/R Mapping Diagrams, in order to improve the diagram support of Robustness. The latest version of Visual Paradigm supports the new deployment of UML up to 2.1 version. Visual Paradigm enables the modeling in visualization display in order to fulfill the requirements of today's software technology and communications [43].

Visual Paradigm supports various software development languages in Code Generation and Reverse Engineering as well as on programming languages Java, C++, .NET, PHP, XML Schema and so on. Visual Paradigm provides Smart Development Environment and DB Visual ARCHITECT for many main software development IDEs, like Eclipse, IntelliJ IDEA, NetBeans/Sun ONE, JBuilder, JDeveloper, and Weblogic Workshop [44].

Visual Paradigm provides a synchronization support with Java code. From Visual Paradigm, Java code can be generated based on the model and establish models without Java code. Any changes occurred in the existing codes can trigger the change of model, vice versa [44].

Visual Paradigm provides fast and convenient methods at a whole software development process from creation of UML diagrams for data modeling till code generation in different IDEs. This solution is much better than the traditional Model-Code-Deploy software development process [44].

2.6.2 Features

Visual Paradigm for UML offers a wide range of functionalities in various aspects, for instance, UML Modeling, Database Modeling, Object-Relational Mapping, Interoperability, IDE Integration, Requirement Modeling, Business Process Modeling, Team Collaboration, Code Engineering and Documentation Generation [44].

For creating an Entity Relationship Diagram by means of Visual Paradigm, there are a few ways available, such as Creating Data Model, Reverse Database Engineering, Creating Array Table in Data Model, Creating Partial Table in Data Model, Copying SQL Statements, Mapping Data Model to Object Model, Mapping Data Model to Enterprise JavaBeans Model and so on [46].

Regarding the main way Creating Data Model, Visual Paradigm offers some other possible features, for instance, creating a new entity element to the ERD, modifying the entity specification, adding new column to the entity, adding relationship to the entities, and editing relationship specification as well [46].

For further detailed information, please see the reference [43] [44].

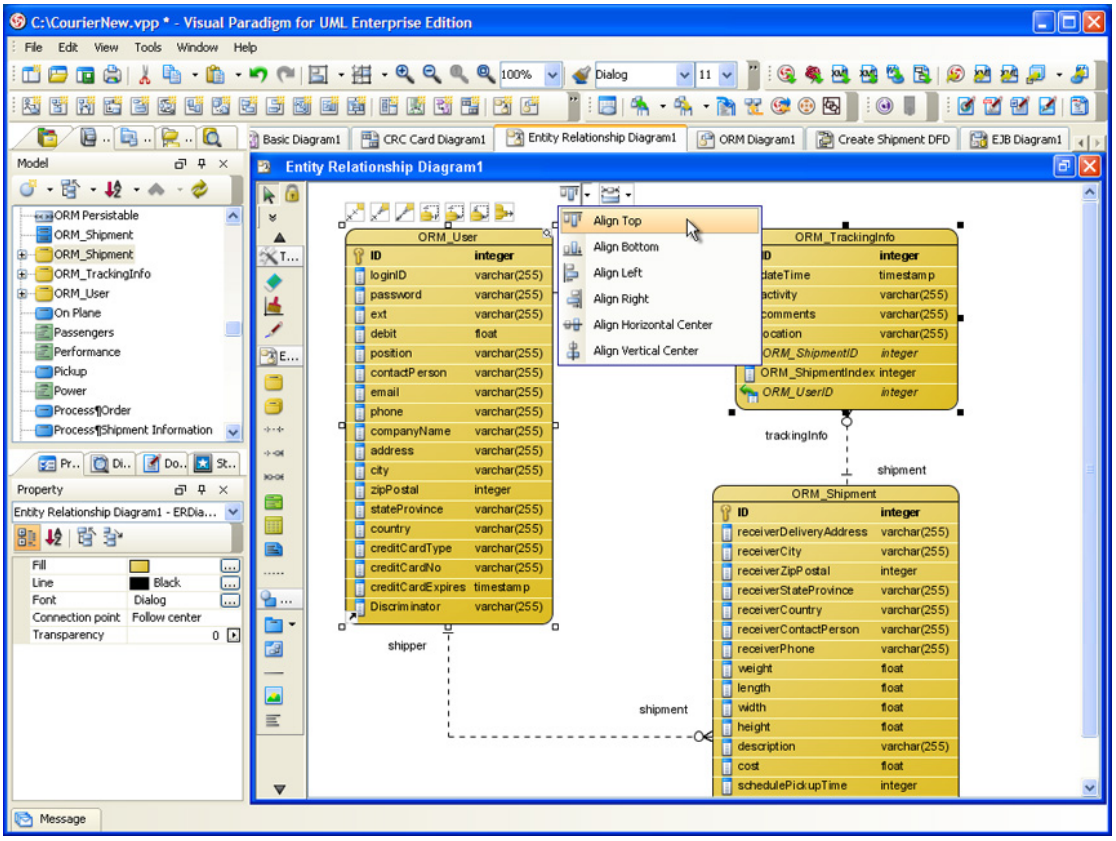

Figure 5: Screenshot of Entity Relationship Diagram in VP [45]

2.7Introduction to Ontology

The following introduces more about ontology's history, definition, three variations of ontology languages, such as OWL, RDF/RDF Schema and DAML+OIL. In additional, three sublanguages of most popular ontology language OWL and a brief overview of the standard tool Protégé.

2.7.1 History

The term Ontology was first originated from the philosophy, the Greeks raised the question "what is the essence of the things through the changes?" One Greek philosopher Aristole created a set of categorizations for being, such as substance, color, relation, quantity, and state etc. Those categorizations help to discover the changes of the things, this theory was approved until the eighteenth century. In the beginning of 1990s, Tom Gruber has changed the concepts of an ontology from philosophy into a technical term by defining the ontology as a formal specification of concepts [48].

The relevant differences between the philosophical term "Ontology" and the technical term "Ontology" in computer science are that the technical term "Ontology" is a machine readable language, it should be also more specific than the philosophical term. Furthermore, the reusable and sharable features

of the technical term "Ontology" are more essential than the philosophical term.

In recent years, Ontologies have been widely applied in such areas as software engineering, artificial intelligence, knowledge engineering and information retrieval etc [4].

2.7.2 Definition

There is a standard and comprehensive definition for the ontology from the view of an ontology engineer. "An Ontology is a formal, explicit specification of a shared conceptualization. Conceptualization refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. Explicit means that the type of concepts used, and the constraints on their use, are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared reflects the notion that an ontology captures consensual knowledge, that is, it is not private of some individual, but accepted by a group" [4].

The followings represent the four main components of an ontology.

Classes

Classes represent a set of concepts that make up to define a certain domain. Each class contains a few instances. Classes should be listed in a taxonomy, namely superclass-subclass hierarchy. Generally, subclasses are defined to derive from subclasses, for instance, the class "Child" is a subclass of the class "People" [4].

Relations

Relations, also called Properties, represent a set of connections between concepts in the domain. They are usually demonstrated as binary relations, in a word to say, Relations bind two instances together. For instance, the Relation "hasChild" can link the class "Father" to the class "Son" [4].

There are three types of Properties: Object properties, Datatype properties and Annotation properties, which the first two are the main types. Nevertheless, the difference is easy to discover. Object properties bind an instance to another instance while Datatype properties bind an instance to an XML schema or an RDF schema. Annotation properties help to insert metadata information to classes, instances and Object/Datatype properties [4].

Formal axioms

Formal axioms represent the formal logical rules that describe domains in an ontology. They help to check and ensure the consistency of the ontologies. Furthermore, they are also helpful for concluding new knowledge [4].

Instances

Instances represent individuals or instances of classes in a certain domain. For example, an instance of the class "Person" is the instance "Mary" [4].

2.7.3 Ontology language

Till now there are various ontology languages available, the followings will introduce three common ontology languages which two of them has appeared in ontology's history and one is currently dominant: OWL, DAML+OIL, and RDF, DRF Schema.

OWL

The OWL (Web Ontology Language) is a knowledge representation language proposed by the World Wide Web Consortium for defining web ontologies. OWL ontologies are always denoted in RDF/XML syntax, it facilitates the machine read and understand the web information better. OWL has three expressive sub-languages: OWL Lite, OWL DL, and OWL Full. This technology is now currently wide used in the Semantic Web [36].

RDF/RDF Schema

Both RDF (Resource Description Framework) and RDF Schema are lightweight meta modeling languages using URI and XML technologies for knowledge exchange in the Semantic Web. Moreover, RDF Schema is an extension of RDF, working to structure RDF resources, such as classes and properties. DAML+OIL has replaced RDF and RDF Schema later by providing more expressiveness [6] [26].

DAML+OIL

DAML is also an language for defining ontologies, it was created as an extension of RDF and XML in order to provide complicated classifications and properties. The latest release is DAML+OIL, congregates the both features. DAML stands for DARPA Agent Markup Language while OIL stands for Ontology Interface Layer. However, OWL has replaced DAML+OIL later [40].

2.7.4 OWL Sub-Languages

There are three sub-languages of Web Ontology Language (OWL): OWL Full, OWL DL and OWL Lite. OWL Lite is the least comprehensive language; OWL DL can be viewed as an extension of OWL Lite, OWL DL is the average comprehensive one; OWL Full is the most comprehensive variant, it can be viewed as an extension of OWL DL [13].

OWL Full

OWL Full guarantees users to take advantage of the maximum expressiveness of OWL. However, it is not possible to run the logic computation to verify the completeness, since no reasoning software is strong enough to support all features of OWL Full [13] [21].

OWL DL

OWL DL is less expressive than OWL Full, but it contains all OWL structures. Additionally, DL stands for Description Logics that is Description Logics are one part of first-order predicate logics that enable computational completeness and decidability. Furthermore, it is also able to verify inconsistencies and ensure the correct classification hierarchy in an ontology automatically [7] [13] [21].

OWL LITE

OWL Lite is the simplest expressive sub-language with lower formal complexity. It is suitable for providing users a simple classification hierarchy and constraints. It is relatively easier to realize the tool support automatically than other two sub-languages, a quick verification for thesauri and other taxonomies is accessible [13] [21].

2.8 Introduction to Ontology Tool Protégé

2.8.1 Overview

Protégé is an open source software developed by Stanford University in cooperation with the University of Manchester. It provides a suite of tools and other useful plug-ins by third parties that help to construct domain models and knowledge-based frameworks with Ontologies [30].

Protégé comprises a set of knowledge model implementations and supports the definition, visualization, documentation and manipulation of ontologies. Protégé also provide friendly GUI that make customer define knowledge models and input data more much more convenient. Protégé is a Java-based Application Programming Interface, so that it can be extended to comprise more functionality for defining knowledge models and applications through third parties plug-ins [30].

There are two main ways of modeling Ontologies supported by the Protégé platform: the Protégé-Frames editor and the Protégé-OWL editor. The Protégé-Frames editor allows users to create frame-based ontologies according to the Open Knowledge Base Connectivity Protocol (OKBC). The Protégé-OWL editor allows users to create ontologies especially for the

Semantic Web, in OWL (Web Ontology Language). For further detailed information, please see the reference [31].

2.8.2 Features

The Protégé-OWL editor will be applied in the following data modeling of the project. The following figure shows the screenshot of the GUI accessible in the Protégé-OWL editor.

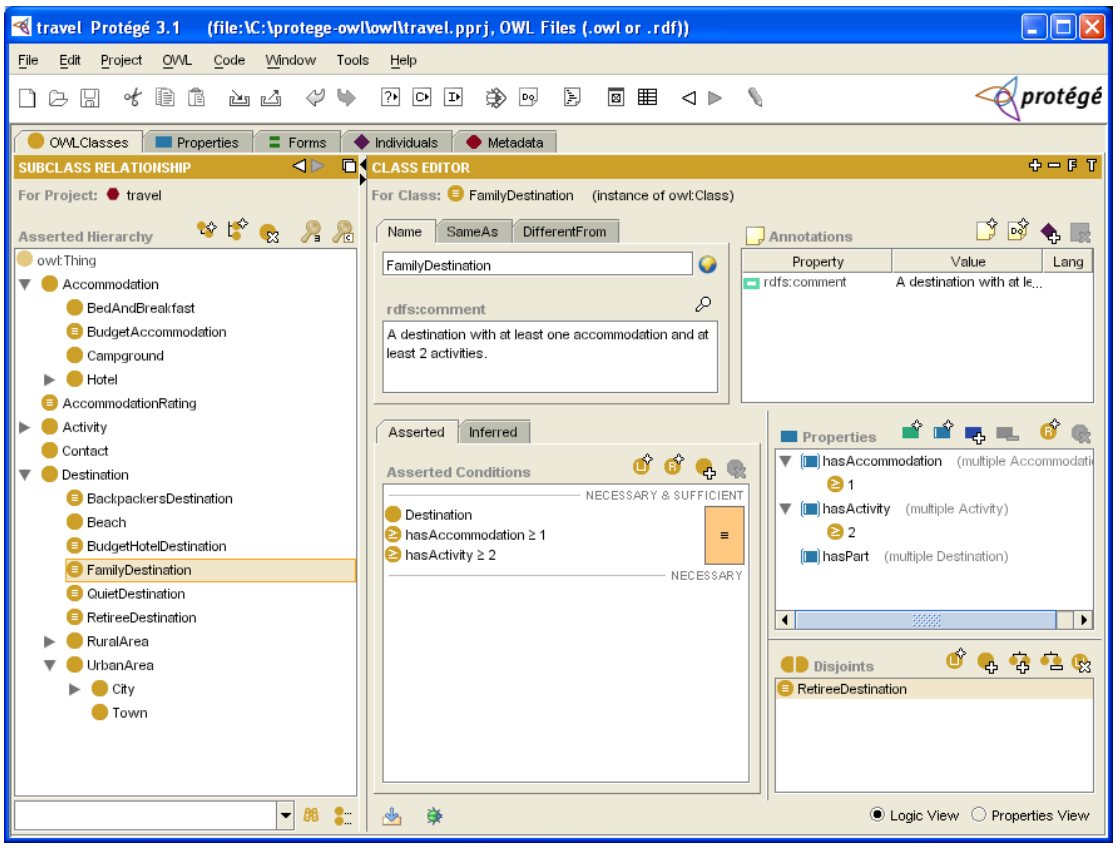

Figure 6: Screenshot of OWL Classes in Protégé [33]

The Protégé-OWL editor supports the standard ontology language OWL for the Semantic Web. The primary features are that one side it enables users to create/import OWL and RDF ontologies, export OWL ontologies to other formats like Clips, N triple, and show RDF/XML source code; on the other side to edit and display classes, properties and SWRL rules by Jambalaya, Ontoviz and OWL Viz etc.; moreover, to define logical class features as OWL expressions; to run reasoners to classify taxonomy, compute inferred types, or execute the third party plug-in DIG Reasoner in order to check consistency and completeness etc as well. Besides, it also supports to generate Java code, Java Schema class etc, automatically.

For further detailed information, please see the references [31] [32].

2.9 Introduction to Multi-Agent System Simulation

A multi-agent system (MAS) is a system composed of multiple heterogeneous intelligent agents, based on decentralized system architecture. Each agent, acting on behalf of users with different goals and motivations, will require the ability to cooperate, coordinate, and negotiate with each other. Multi-agent systems can be used to build complex systems, solve problems which are difficult or impossible for an individual agent or monolithic system to solve [23].

The agents in a multi-agent system should possess a few relevant characteristics, such as:

- Autonomy: each agent is capable of acting independently, exhibiting control over their internal state.
- Local views: for each agent, it is only a local view of the system available.
- Decentralization: there is no one controlling agent (or the system is effectively reduced to a monolithic system).
- Flexibility: each agent is flexible to undertake other tasks which are not defined in the schedule.
- Learning ability: each agent has the capability to adapt to the newly reconfigured environment, react to the prompt changes.
- Social abilities: each agent has the capability to cooperate, coordinate and negotiate with others [17].

For further detailed information regarding the related support tool MAST (Manufacturing Agent Simulation Tool), please refer to the section 4.1.

3. Research issues

This section introduces the four major research issues nowadays existed in the related research area both in software engineering and knowledge engineering.

3.1 Evaluation of UML and OWL

Each modeling language, no matter UML or OWL, has its own focuses and original creation attempts. Both two have some common features; despite sometimes their representations might look quite different.

One goal of this research study is to summarize their similarities and differences, for instance, UML class diagrams can define entities, its state-charts and activity diagrams are appropriate for service and process related ontologies while OWL provides additional prediction description language that the UML couldn't provide. It will present the result of which approach is appropriate for a certain scenario.

Meantime, we will create two data models for this multi-agent system in the field of manufacturing system management by means of UML and OWL approaches. Later, we will discover and evaluate the advantages and disadvantages of those two data models.

3.2 Model consistency checking

Nowadays, the object-oriented software design has already taken an important place in the software engineering. To guarantee the high software quality in the software development, it is crucial to assure the consistency of the UML models. However, it is often hard and undecidable for the UML to ensure its logical consistency and syntax errors checking. Due to the meaning of the UML dependency and their specializations (abstractions, binding, usage, permission) as well as their stereotypes are still not precisely defined. It raises the problem how to understand and how to check consistency between modeling artifacts.

Therefore, OWL and its language tool support this automatic consistency checking function much better than the UML, for example, the tool Protégé and its plug-ins.

It is unrealistic and always wasteful that a lot of efforts and costs required on the manual consistency checking. So the second goal is to find out whether it is conceptually possible to check UML models for their consistency in an automated way and furthermore there is any tool available for the UML that

supports its automatic consistency checking. We will analyze this phenomenon, compare the differences on the conceptual possibilities and external tools with OWL and propose to these questions.

3.3 Mapping from UML to Ontology

The existing problem is that an ontology can't be sufficiently represented in UML. Based on the results of the first goal, we would understand the differences between UML and ontology. So the question is that how to reconcile the gap between UML and OWL, and find out any solution that supports ontology development and conceptual modeling in one standard representation language.

The mapping from UML model to ontology enables the conversion of an arbitrary UML model into OWL ontology, for instance, UML classes are mapped into OWL classes, attributes into data type property, associations into object property, etc. It takes a UML model as input and produces an ontology conforms to the OWL meta model. The transformation can produce an OWL model in a core format, in this way it plays a central role for bridging Model Driven Architecture based standards and Semantic Web technologies.

The third goal is to explore the conceptual possibilities of mapping from UML to OWL, enumerates the concrete OWL concepts that could be mapped to UML and discover its benefits and limitations.

3.4 Extensions of UML and OWL

The initial purpose of creating the UML is to design a widely recognized and standard visualizing language. Therefore, its strong expressiveness and powerfulness has been recognized by the software developers. However, there are still some critics on the high complexity of the UML. UML has 13 diagrams and its constructs contain somewhat redundancy that hinders some junior software developers to learn and adopt UML. At the same time, UML doesn't have a formal semantics like OWL, semantic web not supportable, neither supports properties as first-class a model elements like OWL.

The last goal is also the question that most of software developers concern, to list a few limitations which can be addressed using OWL and its benefits.

4. Use Case Description

The fourth section provides an overview of manufacturing agent simulation tool, as well as a detailed use case description of each involved roles and their collaborations.

4.1 Manufacturing Agent Simulation Tool

A multi-agent system is a system composed of distributed intelligent agents that each agent can carry out his individual local jobs and collaborate together to finish a complicated task in common [23]. The Manufacturing Agent Simulation Tool (MAST) exploited by Rockwell Automation Research Center in Prague, supports the simulation and demonstration of material handling tasks using multi-agents systems [35]. It offers a clear visual overview of advantages and shortcomings during the whole process in the industrial manufacturing domain.

The MAST is programmed in Java language and built on the open source agent platform JADE (Java Agent Development Framework). The initial idea is established on the implementation of the scenario that all the agents perform the whole manufacturing process in a virtual environment with high complexity precisely at a lower cost. The features and benefits of this simulation tool are to manage the real time control, for instance, the user is able to add or change additional functions or the value of parameters like conveyor speed, machine set up and processing time in order to check the influence of these changes. At the same time, it is also feasible to imitate the real situation by using different scheduling algorithms, for instance, alteration of the physical condition or priority could bring minor or huge differences that the user could learn from. It facilitates a better production planning and scheduling in the future. Meantime, it offers the opportunity to integrate a seamless sales network among supplier, customers and operators, and so on. Moreover, it helps to discover of potential failures and assists the designer in establishing a solid system [18] [29].

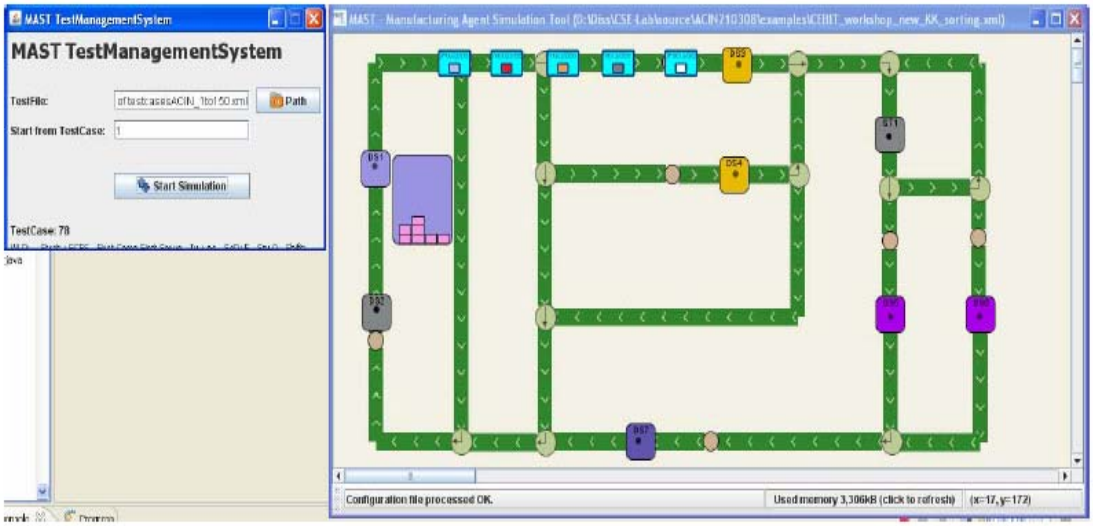

Figure 7: Overview of MAST Test Management System [19]

The above figure 7 provides an overview of a MAST system, it consists of the following components:

The library of agent classes

This component represents the domain of each agent, such as properties of business manager, and the primary material handling objects like a machine, conveyor belt etc. Each agent is supposed to do his autonomous tasks, on demand for accomplishment of a common task needs the mutual cooperation among those agents. For example, the optimal transportation should be created based on the right setting up communications among those material handling objects [19].

The simulation engine

This component simulates the behavioral functions of the agent-based system and verifies the performance of the physical system via monitoring the virtual movement of the products from source to destination, as well as the normal activation or deactivation of triggers and sensors etc [19].

The GUI

This component enables the dynamic demonstration of the multi-agents system in order to inform the user about the current process in a graphic way. Furthermore, the user is also able to monitor the negotiation signals among the agents in case of dynamic scheduling, such as how agents make decisions on choosing alternative routings especially when emergent situations occur [19].

The control interface

This component is a PLC-based control interface, which bridges an

integration of agent classes and the simulation engine. During the simulation, in detecting failures and emergency situations, the control interface could inform the related agents, and the agents could make their choices on dispatching routes, e.g., to avoid delay caused by accidents [19].

Similarities and differences between MAST and SAW

The SAW (Simulator of Assembly Workshops) is an extension and a practical application of MAST. For this project it will be employed in the followings: the components such as "the simulation engine" and "the GUI" remain the same as MAST. SAW is created based on the idea to create an improvement in MAST. The main differences between MAST and SAW are first within the component "the library of agent classes", both simulators possess the component agents, like conveyor agents, machine agents etc, furthermore, in SAW there are more agents that could be added according the software developers or customers' descriptions, such as product agents, and strategy agents etc. In the control interface, there is no PLC-based control interface available in SAW, it appears only possibly in the real MAST system.

4.2 Use Case Description

The following sections include a detailed use case description of the role business manager, plant manager, shop manager, operation manager as well as system developer.

4.2.1 Business manager

The use case description of the role business manager will describe his responsibilities and derive all entities, their properties and relationships.

4.2.1.1 Description of the involved role business manager

The responsibilities of a business manager encompass drawing up a production plan, win more contracts from customers, transform the format from a contract to an order, input the order information into the system to satisfy the corresponding production plan. The data has to be offered from the side of business manager are the general order information, concluding the kind of products needed by customers, the quantity of each needed product and their delivery dates. There are also several status have to be confirmed by the business manager, for example, execution possibility whether we are able to supply all the products with the warrantable qualification required by the customers within the expected dates. Order status, the business manager has to check the status from time to time, production not started, in progress or

finished. The interaction should be happened between him and the plant manager [42].

4.2.1.2 Entities and their Properties

The entity **Business Manager** represents the basic information of the role business manager, should contain the properties, such as the unique ID of business manager, the full name of business manager, the age of business manager, the telephone number of business manager, the contact address of business manager, could be his office or home address, the short description of his main responsibilities.

The entity **Business Order** represents the basic information of business order, should contain the properties, such as the unique ID of business order, the delivery date of required products, the expected date to finish the production of required products, the current process status.

The entity **Client** represents the basic information of the client who gives the business order to the business manager, should contain the properties, such as the unique ID of client, the full name of client, like legally registered company name, the name of contact person, the telephone number of contact person, the office address of contact person.

The entity **Product** represents the basic information of the product which the client has ordered, should contain the properties, such as the unique ID of the required product, the name of the required product, the type of the required product in case it is available, the expected date to finish the production of this kind of product, the process status of this kind of product.

The entity **ProductTree** represents the basic information of the product decomposition into product tree, should contain the properties, such as the unique ID of the product tree, and the complexity of the product tree.

The entity **ProdTreeItem** represents the basic information of each product item that composes the whole product tree, should contain the properties, such as the unique ID of the product tree item, the name of the product tree item, the amount of the product tree item, the father of this product tree item, the x position (width) of the product tree item, the y position (depth) of the product tree item, and the brothers of this product tree item.

The entity **Quantity** should contain the properties, such as the unique ID of business order's quantity, and the required number of required product.

4.2.1.3 Relationship

One **Business Manager** could hold several **Business Orders**. One **Client** could order several **Business Orders** at the same time or different time. One **Business Order** might have several **Products**. Each **Product** decomposes a **ProductTree**, which is composed of several **ProdTreeItems**. Each **Business Order**/ **ProdTreeItem** has a corresponding **Quantity**.

4.2.2 Plant manager

The use case description of the role plant manager will describe his responsibilities and derive all entities, their properties and relationships.

4.2.2.1 Description of the involved role plant manager

The responsibilities of a plant manager encompass checking whether all needed resources or raw materials are available or sufficient, calculating whether there exist enough capacities to carry out all the working steps in time, arranging the scheduling algorithms to individual machine and set up the appropriate priority. The data from the plant manager has to be offered is the division of a business order into several work orders, information about current inventory and free production capacities, actual shop layout. The plant manager has to decide on the acceptance of executing the work orders, the product scheduling line for the next shifts and so on [42].

4.2.2.2 Entities and their Properties

The entity **Plant Manager** represents the basic information of the role plant manager, should contain the properties, such as the unique ID of plant manager, the full name of plant manager, the age of plant manager, the telephone number of plant manager, the contact address of plant manager, could be his office or home address, and the short description of his main responsibilities.

The entity **Work Order** represents the basic information of the work order which converts from the business order, should contain the properties, such as the unique ID of the work order, the name of work order, the process status of this work order, the expected date to finish this work order, the actual date to finish the work order in case the finishing time has been postponed, otherwise, the value could be empty.

The entity **BillOfMaterial** represents the bill of the material required for the product production, should contain the properties, such as the unique ID of material bill, the name of the required material, and the amount of this required material.

The entity **BoMItem** represents the bill of the material item required for the product production, should contain the properties, such as the unique ID of the material item, the name of the material item, and the amount of this required material item.

The entity **Inventory** represents the basic information of the current inventory's situation, should contain the properties, such as the unique ID of the inventory, the name of the inventory, the current status of the inventory's availability, and the amount of inventory in case the current inventory is

available.

The entity **Failure** represents the generally potential of occurred failures, should contain the properties, such as the unique ID of potential failure, the name of the potential or occurred failure, other related effect or delay caused by the potential or occurred failure, the short description of this occurred or potential failure, for instance, like cause or severity of this kind of situation etc, and the short description of the solution.

The entity **MachineFailure** represents the specific failures caused by the machine, should contain the properties, such as the unique ID of the broken machine, and the name of the machine failure.

The entity **ConveyorFailure** represents the specific failures caused by the conveyor, should contain the properties, such as the unique ID of the broken conveyor, and the name of the conveyor failure.

The entity **Shift** represents the basic information of arranging the shift for the operators in turn, should contain the properties, such as the unique ID of the shift, the name of the shift, for instance, day shift or night shift, etc, the period time for switching a shift, for instance, each four hours per shift, and the work load of each shift.

The entity **Machine** represents the basic information of the machine, should contain the properties, such as the unique ID of the machine, the name of machine, the current status of machine, for instance, busy or idle etc, the short description of the machine, for instance, functions, conditions etc.

The entity **MachineItem** represents the basic information of the machine item, should contain the properties, such as the unique ID of the machine item, the name of the machine item, the current status of machine, the run step of the machine item during the machine operation period, the short description of the machine item, for instance, functions, conditions, etc.

The entity **Strategy** represents the basic information of the feasible strategy that could be applied for the machine operation, should contain the properties, such as the unique ID of the strategy in use, the name of the strategy in use, for instance, FCFS (First Come, First Served) or EDD (Earliest Due Date) etc, the type of the strategy in use, for instance, static or dynamic scheduling etc, the priority of the strategy in use, for instance, high, average, low etc.

The entity **Capacity** represents the basic information of the machine capacity, should contain the properties, such as the unique ID of the machine capacity, the amount of the occupied machine capacity at the moment, the amount of the full provided machine capacity, the available percentage of the machine capacity, for instance, 1 minus the result of the occupied capacity divides the provided capacity by calculation in general.

4.2.2.3 Relationship

One **Plant Manager** could hold several **Work Orders**. Each **Work Order**

has one **BillOfMaterial**, one **BillOfMaterial** may have several **BoMItems**. Each **BoMItem** has one **Inventory**. One **Plant Manager** might assume several **Failures**. Each **Failure** is either a **MachineFailure** or **ConveyorFailure** according to its characteristics. There are several **Shifts** available working on one **Work Order**. Each **Work Order** could be applied one **Strategy** according to the various requirements. One **Plant Manager** could monitor several **Machines**, each **Machine** is composed of several **MachineItems**. Each **Machine** can have one **Capacity**.

4.2.3 Shop manager

The use case description of the role shop manager will describe his responsibilities and derive all entities, their properties and relationships.

4.2.3.1 Description for the involved role shop manager

The responsibilities of a shop manager encompass decomposition of the whole production process into every single procedure steps, discovery of the optimal sequence of the production process and its sub processes for the shift; plan the agile adaption to the shop layout. The shop manager has to offer the data about the time and resource scheduling for the actual shift, actual shop layout, and state problems of actual execution process. The shop manager is responsible to estimate the work steps to assemble the products, set-up time and costs for adapting shop layout should be taken into consideration [42].

4.2.3.2 Entities and their Properties

The entity **Shop Manager** represents the basic information of the role shop manager, should contain the properties, such as the unique ID of shop manager, the full name of shop manager, the age of shop manager, the telephone number of shop manager, the contact address of shop manager, could be his office or home address, and the short description of his main responsibilities.

The entity **Machine** please see the section 4.2.2.2.

The entity **MachineItem** please see the section 4.2.2.2.

The entity **Transport** represents the basic information of the product transportation, should contain the properties, such as the unique ID of the product transport, the product that would be transported, the place from where the products would be obtained, the place to where the products should be transported, the short description of the product transport, and the graphical notation of this product transport.

The entity **ArrivalSequence** represents the basic information of the product's arrival sequence, should contain the properties, such as the unique

ID of the arrival sequence, and the chronological sequence of the product arrival.

The entity **Testcase** represents the basic information of the test cases in case the shop manager needs to verify the machine or transport, should contain the properties, such as the unique ID of test case, and the short description of test case.

4.2.3.3 Relationship

One **Shop Manager** could observe several **Machines** and meantime supervise several **Transports**, like one **Product** is transfered from the source to the destination. Each **Machine** is composed of several **MachineItems**. Each **MachineItem** contains exactly one **ArrivalSequence**. Each **Transport** contains exactly one **ArrivalSequence**. Each **MachineItem** could perform several **Testcases**, in other words it means that one or several **Testcases** can be designed to verify whether each **MachineItem** works correctly. Each **Transport** could perform several **Testcases**, in other words it means that one or several **Testcases** can be designed to verify whether each **Transport** works correctly.

4.2.4 Operation manager

The use case description of the role operation manager will describe his responsibilities and derive all entities, their properties and relationships.

4.2.4.1 Description for the involved role operation manager

The responsibilities of a operation manager encompass controlling and inspecting the coordination of all the procedure steps, rapid reaction on possible appearing problems like power failures, machine failures and so on, balancing the utilization frequency of the machines in order to achieve the best throughput. The operation manager has to deal with the work steps defined by the shop manager to fabricate the products, fill the requirements for adapting shop layout. His responsibility is to arrange the efficient sequence of working steps for each machine, set up the correct shop layout, record the finished products, and output their log files of the simulation for future references [42].

4.2.4.2 Entities and their Properties

The entity **Operation Manager** represents the basic information of the role operation manager, should contain the properties, such as the unique ID of operation manager, the full name of operation manager, the age of operation

manager, the telephone number of operation manager, the contact address of operation manager, could be his office or home address, the short description of his main responsibilities.

The entity **Palette** represents the basic information of the palettes in a conveyor, should contain the properties, such as the unique ID of palette, the length of the palette, the current status of the palette, for instance, busy or idle, the unique ID of other palette that should be followed by this palette, the short description of this palette's situation.

The entity **Route** represents the basic information of the palette's route, should contain the properties, such as the unique ID of the route, and the short description of the scheduled route, like its environment etc.

The entity **Conveyor** represents the basic information of the conveyor, should contain the properties, such as the unique ID of the conveyor, the start point of conveyor, the end point of conveyor, the length of the conveyor that equals the distance from the start point to the end point, the amount of palettes in this conveyor, and the run speed of this conveyor.

The entity **Transport** please see the section 4.2.3.2.

The entity **Failure** please see the section 4.2.2.2

The entity **MachineFailure** please see the section 4.2.2.2.

The entity **ConveyorFailure** please see the section 4.2.2.2.

The entity **FinishedProduct** represents the basic information of the product that has been already finished, should contain the properties, such as the unique ID of the finished product, the name of the finished product, the amount of the finished product, and the log files of the finished product during its whole production process.

4.2.4.3 Relationship

One **Operation Manager** could supervise several **Palettes**. Each **Palette** follows one **Route**. One **Conveyor** can contain several **Palettes**. Each **Conveyor** can operate one **Transport**, like one **Product** is transferred from the source to the destination. One **Operation Manager** could supervise several **Failures**, each **Failure** is either a **MachineFailure** or a **ConveyorFailure** according to its characteristics. One **Operation Manager** could check several **FinishedProducts**, namely which **Products** have been finished and their amounts etc.

4.2.5 System developer

The use case description of the role system developer will describe his responsibilities and derive all entities, their properties and relationships.

4.2.5.1 Description for the involved role system developer

The responsibilities of a system developer encompass bridging a background platform for each layer, providing basic information for the simulation of the production, including products, shop layout, etc. The system developer handles the changes, and basic data of each role involved during the production process. He is responsible to monitor, provide information of the production line, and meantime also guarantee that all the roles and process are good in operation. He has the interaction with all the roles over all the levels, is available for all roles. In case of any change, the four roles should inform the system developer at their earliest convenience, and the system developer is obligatory to response promptly [42].

4.2.5.2 Entities and their Properties

The entity **System Developer** represents the basic information of the role system developer, should contain the properties, such as the unique ID of system developer, the full name of system developer, the age of system developer, the telephone number of system developer, the contact address of system developer, could be his office or home address, and the short description of his main responsibilities.

The entity **Product** please see the section 4.2.1.2.

The entity **ProductTree** please see the section 4.2.1.2.

The entity **ProdTreeItem** please see the section 4.2.1.2.

The entity **Function** represents the basic information of the general function, should contain the properties, such as the unique ID of the function, and the short description of the function.

The entity **MachineFunction** represents the basic information of the specific machine function, should contain the properties, such as the unique ID of the machine function, the minimum required time for this machine to finish processing the particular task, the maximum required time for this machine to finish processing the particular task, the expected time for this machine to finish processing the particular task, and the required time for this machine to unload.

The entity **TransportFunction** represents the basic information of the specific transport function, should contain the properties, such as the unique ID of the transport function, the place from where the products would be obtained, the place to where the products should be transported, the minimum required transportation time for products, the maximum required transportation time for products, and the expected transportation time for products.

The entity **ShopLayout** represents the basic information of the shop layout, should contain the properties, such as the unique ID of the shop layout, the costs required when the shop layout is changed, and the time required when

the shop layout is changed.

The entity **Conveyor** please see the section 4.2.4.2.

The entity **Crane** represents the basic information of the transportation vehicle crane, should contain the properties, such as the unique ID of the crane, and the free space of the crane.

The entity **Inventory** please see the section 4.2.2.2.

The entity **Machine** please see the section 4.2.2.2.

The entity **Diverter** represents the basic information of the diverter, should contain the properties, such as the unique ID of diverter, and the direction of the conveyor's junction, for instance, left, right or straight.

4.2.5.3 Relationship

One **System Developer** maintains the basic information of several **Products**. Each **Product** is composed following a **ProductTree**, which is composed of several **ProdTreeItems**. One **System Developer** operates several **Functions**, each **Function** is either a **MachineFunction** or a **TransportFunction** according to its characteristics. One **System Developer** adapts several **ShopLayouts** and vice versa, several **System Developers** can adapt one **ShopLayout**. There are several **Conveyors** available in each **ShopLayout**. Every **Conveyor** is connected to exactly two **Nodes**. Each **Node** is equal to either a **Crane** or a **Inventory**. In a **Crane** or a **Inventory**, it can contain several **Machines**. One **Conveyor** comprises several **Diverters,** such as turn to forward, backward, left or right directions.

4.3 Collaborations among the roles

The following figure 8 represents the simplest situation of the mutual collaborations among the roles. The collaboration takes place among those five roles.

For example, the business manager should notify the plant manager of business order and product related information etc, meanwhile, the plant manager is also responsible to report the newest status of the work order process to him. The plant manager is ready to give the shop manager the draft of shop layout, the shop manager is assigned to execute the shop layout and offer the feedbacks as well. The operation manager has to implement each single work step made by the shop manager; simultaneously, the operation manager should submit the log file of whole implementation. The system developer is needed to collect and store the raw data for all the other roles, if any data is changed or system failure occurred etc, the other roles should inform him.

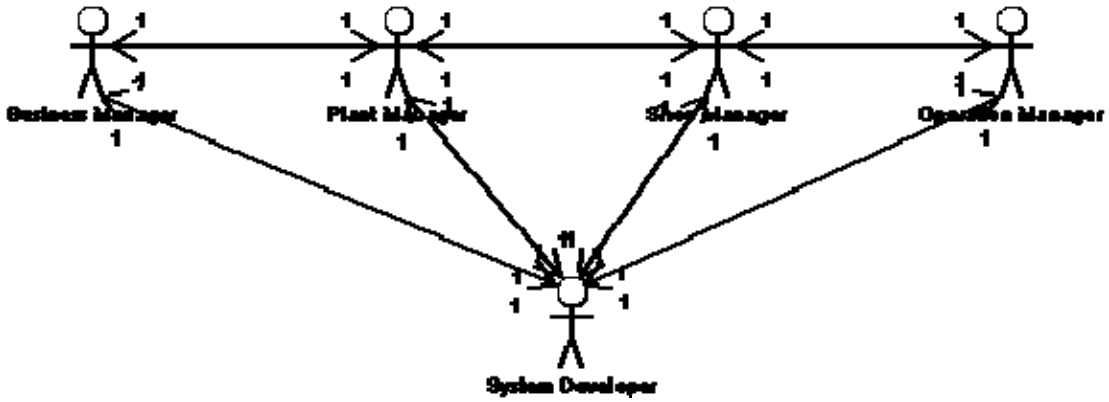

Figure 8: Mutual collaborations among each role

Furthermore, this use case can be extended for a large and complex system. For example, there are multiple business managers, plant managers or other agents available in a complicated large system. Thus, the multiplicity should be 1 to n, n to 1 or n to n etc.

5. Data Modeling for SAW with UML and Ontology

The fifth section describes two data modeling approaches based on UML and Ontology approaches and finally compares the two approaches.

5.1 UML-based approach

The data modeling based on UML approach implements the use case descriptions defined in above section 4, demonstrates five ER diagrams for each role, business manager, plant manager, shop manager, operation manager and system developer, in production automation multi-agents system.

5.1.1 ER Diagram for Business Manager

This section presents the ER Diagram for business manager and its corresponding descriptions.

5.1.1.1 Diagram

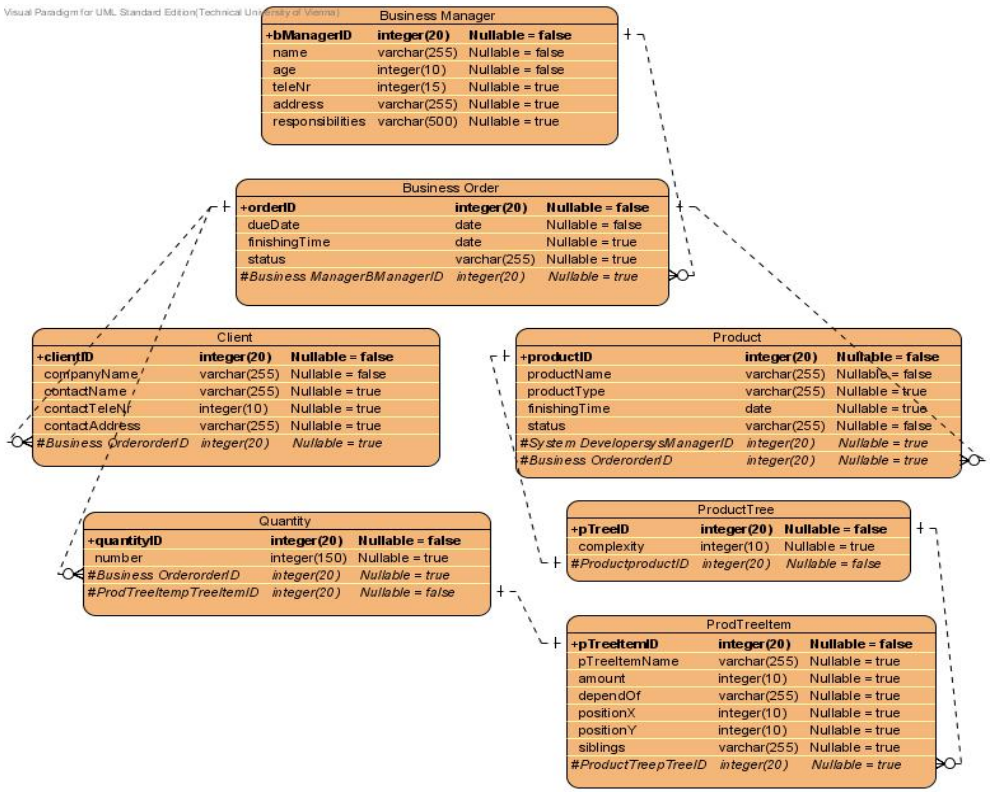

Figure 9: ER Diagram for business manager

5.1.1.2 Description

The above figure 9 "ER Diagram for business manager" implements the use case description defined in section 4.2.1.2 "Entities and their properties". Each defined Entity has been created in ER Diagram and inserted their corresponding properties. Each property's type and its length like integer(20), varchar(255) etc and the initial value of null able true or false should be estimated for the physical database. During the modeling, the primary key, for instance *bManagerID*, and the foreign keys should be defined additionally. The relationship among each Entity and the multiplicity of relationship implement "Relationship" defined in section "4.2.1.3" respectively. The limitation of ER Diagram is that the instance of each Entity could not be shown.

The entity **Business Manager** represents the basic information of the role business manager, should contain the properties, such as *bManagerID* that indicates the unique ID of business manager, *name* indicates the full name of business manager, *age* indicates the age of business manager, *teleNr* indicates the telephone number of business manager, *address* indicates the contact address of business manager, could be his office or home address, *responsibilities* indicates the short description of his main responsibilities.

The entity **Business Order** represents the basic information of business order, should contain the properties, such as o*rderID* that indicates the unique ID of business order, *dueDate* indicates the delivery date of required products, *finishingTime* indicates the expected date to finish the production of required products, s*tatus* indicates the current process status, for instance, 3 choices, including not started, in progress and finished.

The entity **Client** represents the basic information of the client who gives the business order to the business manager, should contain the properties, such as *clientID* that indicates the unique ID of client, *companyName* indicates the full name of client, like legally registered company name, *contactName* indicates the name of contact person, *contactTeleNr* indicates the telephone number of contact person, *contactAddress* indicates the office address of contact person.

The entity **Product** represents the basic information of the product which the client has ordered, should contain the properties, such as *productID* that indicates the unique ID of the required product, *productName* indicates the name of the required product, *productType* indicates the type of the required product in case it is available, *finishingTime* indicates the expected date to finish the production of this kind of product, *status* indicates the process status of this kind of product, for instance, 3 choices, including not started, in progress, and finished.

The entity **ProductTree** represents the basic information of the product decomposition into product tree, should contain the properties, such as

pTreeID that indicates the unique ID of the product tree, *complexity* indicates the complexity of the product tree.

The entity **ProdTreeItem** represents the basic information of each product item that composes the whole product tree, should contain the properties, such as *pTreeItemID* that indicates the unique ID of the product tree item, *pTreeItemName* indicates the name of the product tree item, *amount* indicates the amount of the product tree item, *dependOf* indicates the father of this product tree item, positionX indicates the x position (width) of the product tree item, positionY indicates the y position (depth) of the product tree item, *Siblings* indicates the brothers of this product tree item.

The entity **Quantity** should contain the properties, such as *quantityID* that indicates the unique ID of business order's quantity, *number* indicates the required number of required product

5.1.2 ER Diagram for Plant Manager

This section presents the ER Diagram for plant manager and its corresponding descriptions.

5.1.2.1 Diagram

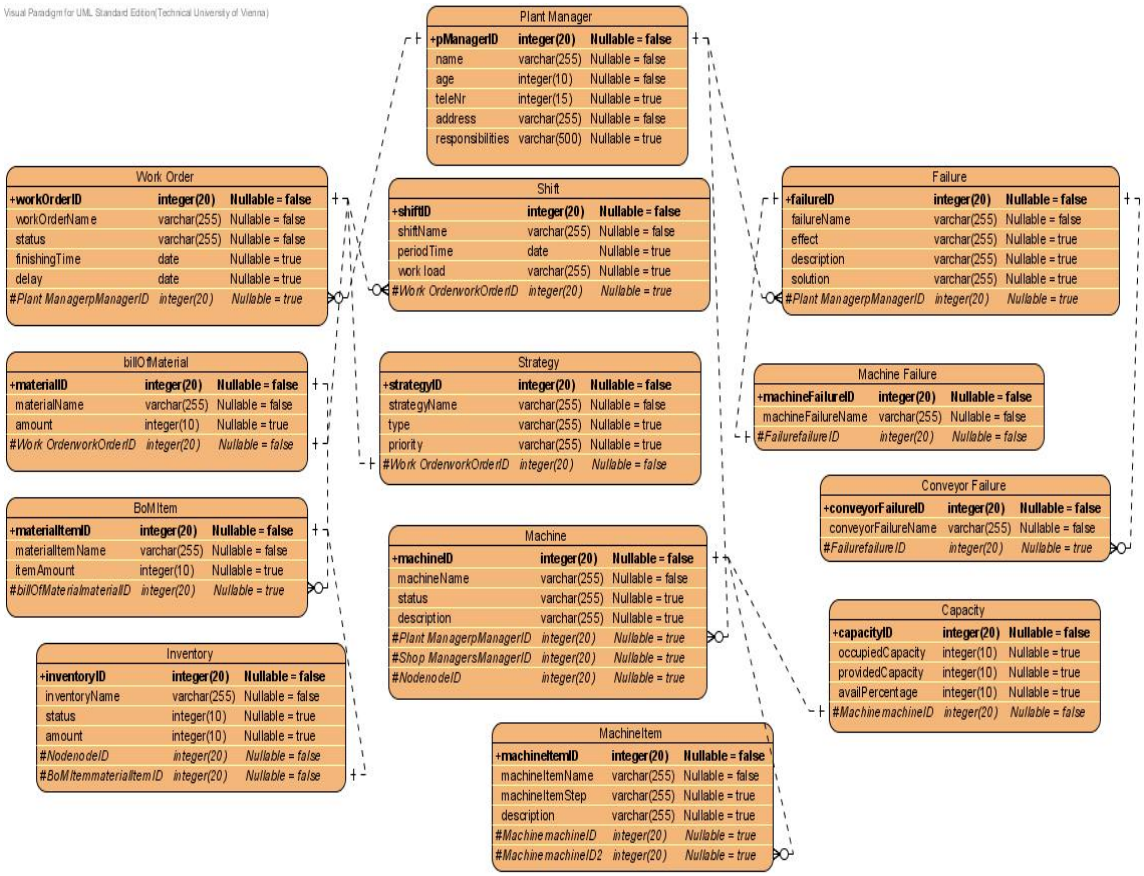

Figure 10: ER Diagram for plant manager

5.1.2.2 Description

The entity **Plant Manager** represents the basic information of the role plant manager, should contain the properties, such as p*ManagerID* that indicates the unique ID of plant manager, *name* indicates the full name of plant manager, *age* indicates the age of plant manager, *teleNr* indicates the telephone number of plant manager, *address* indicates the contact address of plant manager, could be his office or home address, *responsibilities* indicates the short description of his main responsibilities.

The entity **Work Order** represents the basic information of the work order which converts from the business order, should contain the properties, such as *workOrderID* that indicates the unique ID of the work order, *workOrderName* indicates the name of work order, *status* indicates the process status of this work order, for instance, not started, in progress, and finished, *finishingTime* indicates the expected date to finish this work order, *delay* indicates the actual date to finish the work order in case the finishing time has been postponed, otherwise, the value could be empty.

The entity **BillOfMaterial** represents the bill of the material required for the product production, should contain the properties, such as *materialID* that indicates the unique ID of material bill, *materialName* indicates the name of the required material, *amount* indicates the amount of this required material.

The entity **BoMItem** represents the bill of the material item required for the product production, should contain the properties, such as *materialItemID* that indicates the unique ID of the material item, *materialItemName* indicates the name of the material item, *itemAmount* indicates the amount of this required material item.

The entity **Inventory** represents the basic information of the current inventory's situation, should contain the properties, such as *inventoryID* that indicates the unique ID of the inventory, *inventoryName* indicates the name of the inventory, *available* indicates the current status of the inventory's availability, for instance, yes or no, *amount* indicates the amount of inventory in case the current inventory is available.

The entity **Failure** represents the generally potential or occurred failures, should contain the properties, such as *failureID* that indicates the unique ID of potential failure, *failureName* indicates the name of the potential or occurred failure, *effect* indicates other related effect or delay caused by the potential or occurred failure, *description* indicates the short description of this occurred or potential failure, for instance, like cause or severity of this kind of situation etc, *solution* indicates the short description of the solution.

The entity **MachineFailure** represents the specific failures caused by the machine, should contain the properties, such as m*achineFailureID* that indicates the unique ID of the broken machine, *machineFailureName* indicates the name of the machine failure.

The entity **ConveyorFailure** represents the specific failures caused by the

conveyor, should contain the properties, such as *conveyorFailureID* that indicates the unique ID of the broken conveyor, *conveyorFailureName* indicates the name of the conveyor failure.

The entity **Shift** represents the basic information of arranging the shift for the operators in turn, should contain the properties, such as *shiftID* that indicates the unique ID of the shift, *shiftName* indicates the name of the shift, for instance, day shift or night shift, etc, *periodTime* indicates the period time for switching a shift, for instance, each four hours per shift, *workload* indicates the work load of each shift.

The entity **Machine** represents the basic information of the machine, should contain the properties, such as *machineID* that indicates the unique ID of the machine, *machineName* indicates the name of machine, *status* indicates the current status of machine, for instance, busy or idle etc, *description* indicates the short description of the machine, for instance, functions, conditions, etc.

The entity **MachineItem** represents the basic information of the machine item, should contain the properties, such as *machineItemID* that indicates the unique ID of the machine item, *machineItemName* indicates the name of the machine item, *status* indicates the current status of machine, for instance, busy or idle etc, *machineItemStep* indicates the run step of the machine item during the machine operation period, *description* indicates the short description of the machine item, for instance, functions, conditions, etc.

The entity **Strategy** represents the basic information of the feasible strategy that could be applied for the machine operation, should contain the properties, such as *strategyID* that indicates the unique ID of the strategy in use, *strategyName* indicates the name of the strategy in use, for instance, FCFS (First Come, First Served) or EDD (Earliest Due Date) etc, *type* indicates the type of the strategy in use, for instance, static or dynamic scheduling etc, *priority* indicates the priority of the strategy in use, for instance, high, average, low etc.

The entity **Capacity** represents the basic information of the machine capacity, should contain the properties, such as *capacityID* that indicates the unique ID of the machine capacity, *occupiedCapacity* indicates the amount of the occupied machine capacity at the moment, *providedCapacity* indicates the amount of the full provided machine capacity, *availPercentage* indicates the available percentage of the machine capacity, for instance, 1 minus the result of the occupied capacity divides the provided capacity by calculation in general.

5.1.3 ER Diagram for Shop Manager

This section presents the ER Diagram for shop manager and its corresponding descriptions.

5.1.3.1 Diagram

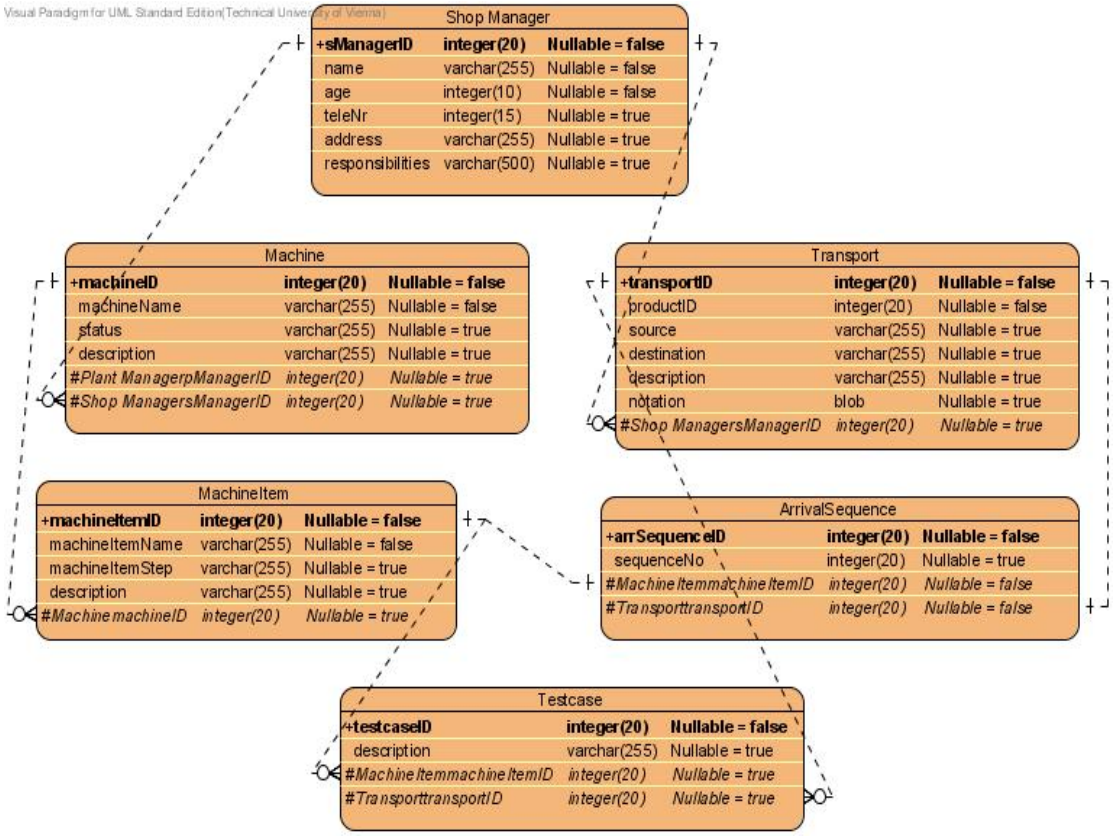

Figure 11: ER Diagram for shop manager

5.1.3.2 Description

The entity **Shop Manager** represents the basic information of the role shop manager, should contain the properties, such as s*ManagerID* that indicates the unique ID of shop manager, *name* indicates the full name of shop manager, *age* indicates the age of shop manager, *teleNr* indicates the telephone number of shop manager, *address* indicates the contact address of shop manager, could be his office or home address, *responsibilities* indicates the short description of his main responsibilities.

The entity **Machine** please see the section 4.2.2.2.

The entity **MachineItem** please see the section 4.2.2.2.

The entity **Transport** represents the basic information of the product transportation, should contain the properties, such as *transportID* that indicates the unique ID of the product transport, *productID* indicates the product that would be transported, *source* indicates the place from where the products would be obtained, *destination* indicates the place to where the products should be transported, *description* indicates the short description of the product transport, *notation* indicates the graphical notation of this product

transport.

The entity **ArrivalSequence** represents the basic information of the product's arrival sequence, should contain the properties, such as *arrSequenceID* that indicates the unique ID of the arrival sequence, *sequenceNo* indicates the chronological sequence of the product arrival.

The entity **Testcase** represents the basic information of the test cases in case the shop manager needs to verify the machine or transport, should contain the properties, such as *testcaseID* that indicates the unique ID of test case, *description* indicates the short description of test case.

5.1.4 ER Diagram for Operation Manager

This section presents the ER Diagram for operation manager and its corresponding descriptions.

5.1.4.1 Diagram

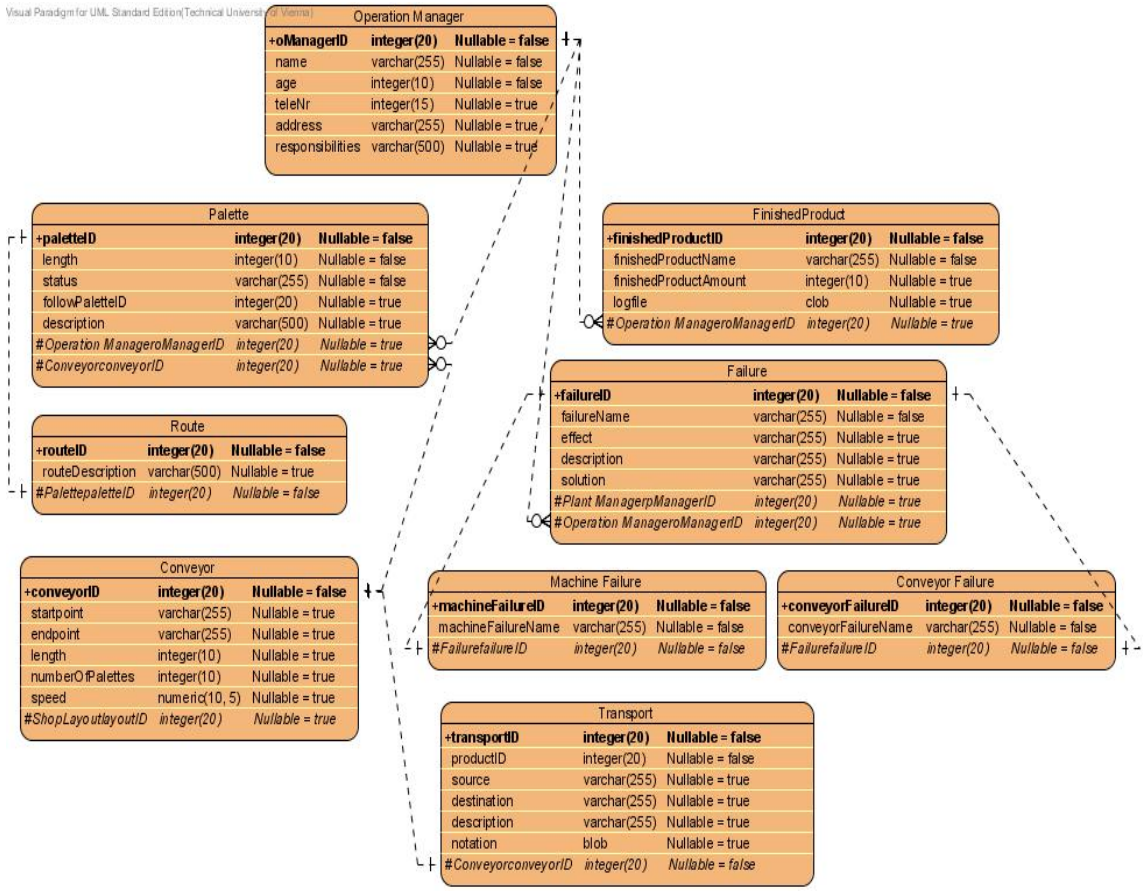

Figure 12: ER Diagram for operation manager

5.1.4.2 Description

The entity **Operation Manager** represents the basic information of the role operation manager, should contain the properties, such as o*ManagerID* that indicates the unique ID of operation manager, *name* indicates the full name of operation manager, *age* indicates the age of operation manager, *teleNr* indicates the telephone number of operation manager, *address* indicates the contact address of operation manager, could be his office or home address, *responsibilities* indicates the short description of his main responsibilities.

The entity **Palette** represents the basic information of the palettes in a conveyor, should contain the properties, such as *paletteID* that indicates the unique ID of palette, *length* indicates the length of the palette, *status* indicates the current status of the palette, for instance, busy or idle, *followPaletteID* indicates the unique ID of other palette that should be followed by this palette, *description* indicates the short description of this palette's situation.

The entity **Route** represents the basic information of the palette's route, should contain the properties, such as *routeID* that indicates the unique ID of the route, *routeDescription* indicates the short description of the scheduled route, like its environment etc.

The entity **Conveyor** represents the basic information of the conveyor, should contain the properties, such as *conveyorID* that indicates the unique ID of the conveyor, *startpoint* indicates the start point of conveyor, *endpoint* indicates the end point of conveyor, *length* indicates the length of the conveyor that equals the distance from the start point to the end point, n*umberOfPalettes* indicates the amount of palettes in this conveyor, *speed* indicates the run speed of this conveyor.

The entity **Transport** please see the section 4.2.3.2.

The entity **Failure** please see the section 4.2.2.2

The entity **MachineFailure** please see the section 4.2.2.2.

The entity **ConveyorFailure** please see the section 4.2.2.2.

The entity **FinishedProduct** represents the basic information of the product that has been already finished, should contain the properties, such as *finishedProductID* that indicates the unique ID of the finished product, *finishedProductName* indicates the name of the finished product, *finishedProductAmount* indicates the amount of the finished product, and *logfile* indicates the logfile of the finished product during its whole production process.

5.1.5 ER Diagram for System Developer

This section presents the ER Diagram for system developer and its corresponding descriptions.

5.1.5.1 Diagram

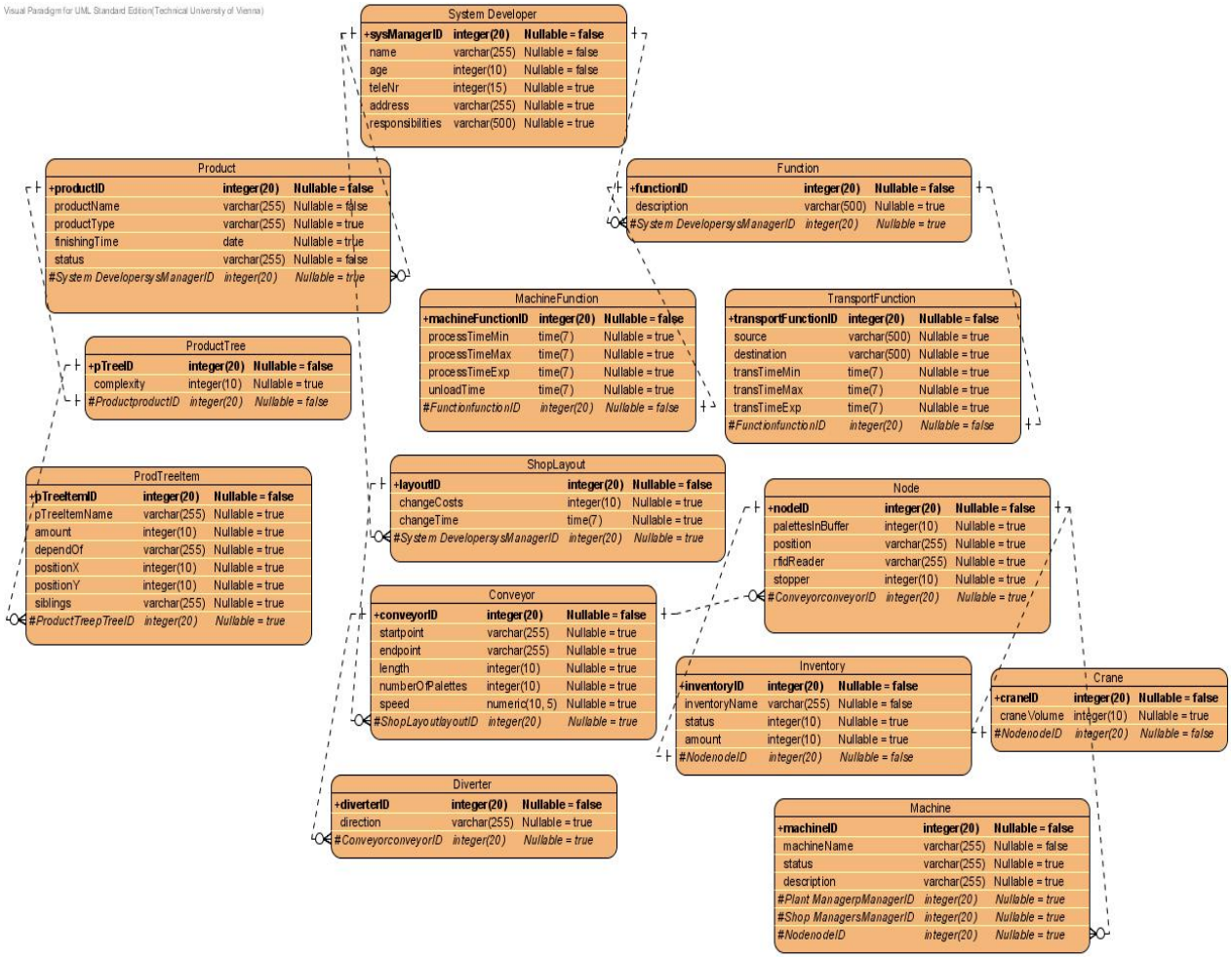

Figure 13: ER Diagram for system developer

5.1.5.2 Description

The entity **System Developer** represents the basic information of the role system developer, should contain the properties, such as *sysDeveloperID* that indicates the unique ID of system developer, *name* indicates the full name of system developer, *age* indicates the age of system developer, *teleNr* indicates the telephone number of system developer, *address* indicates the contact address of system developer, could be his office or home address, *responsibilities* indicates the short description of his main responsibilities.

The entity **Product** please see the section 4.2.1.2.

The entity **ProductTree** please see the section 4.2.1.2.

The entity **ProdTreeItem** please see the section 4.2.1.2.

The entity **Function** represents the basic information of the general function, should contain the properties, such as *functionID* that indicates the unique ID of the function, *description* indicates the short description of the

function.

The entity **MachineFunction** represents the basic information of the specific machine function, should contain the properties, such as *machineFunctionID* that indicates the unique ID of the machine function, *processTimeMin* indicates the minimum required time for this machine to finish processing the particular task, *processTimeMax* indicates the maximum required time for this machine to finish processing the particular task, *processTimeExp* indicates the expected time for this machine to finish processing the particular task, *unloadTime* indicates the required time for this machine to unload.

The entity **TransportFunction** represents the basic information of the specific transport function, should contain the properties, such as *transportFunctionID* that indicates the unique ID of the transport function, *source* indicates the place from where the products would be obtained, *destination* indicates the place to where the products should be transported, *transTimeMin* indicates the minimum required transportation time for products, *transTimeMax* indicates the maximum required transportation time for products, *transTimeExp* indicates the expected transportation time for products.

The entity **ShopLayout** represents the basic information of the shop layout, should contain the properties, such as *layoutID* that indicates the unique ID of the shop layout, *changeCosts* indicates the costs required, if the shop layout is changed, *changeTime* indicates the time required, if the shop layout is changed.

The entity **Conveyor** please see the section 4.2.4.2.

The entity **Crane** represents the basic information of the transportation vehicle crane, should contain the properties, such as *craneID* that indicates the unique ID of the crane, *craneVolume* indicates the free space of the crane.

The entity **Inventory** please see the section 4.2.2.2.

The entity **Machine** please see the section 4.2.2.2.

The entity **Diverter** represents the basic information of the diverter, should contain the properties, such as *diverterID* that indicates the unique ID of diverter, *diverterDirection* indicates the direction of the conveyor's junction, for instance, left, right or straight.

5.2 Ontology-based approach

The data modeling based on Ontology approach implements the use case descriptions defined in above section 4, demonstrates five Ontoviz diagrams for each role, business manager, plant manager, shop manager, operation manager and system developer, in production automation multi-agents system.

5.2.1 Business Manager

This section presents the Ontoviz Diagram for business manager and its corresponding descriptions.

5.2.1.1 Diagram

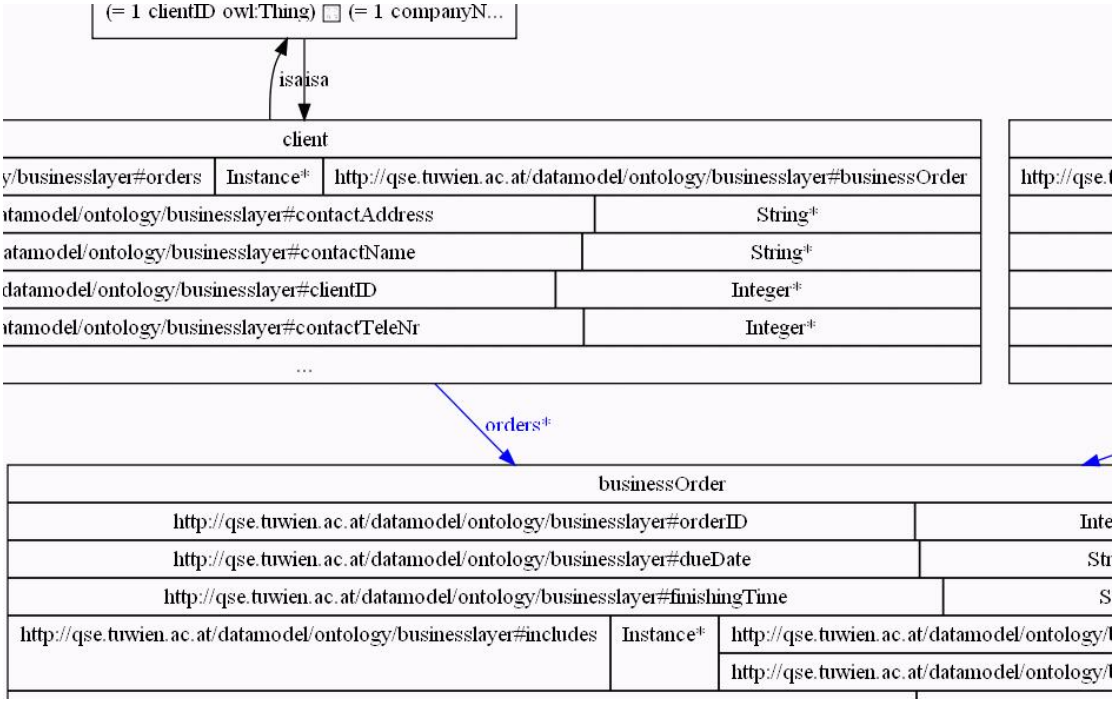

Figure 14: Part of Ontoviz diagram for business manager

5.2.1.2 Description

The building elements of OWL are defined differently and each with a different name, unlike the elements of UML. In OWL, there are some own components available, such as individuals, datatype properties of each individual, restriction on each datatype property, object properties links two individuals together and restriction on each object property etc.

The individuals here are businessManager, businessOrder, client, product, productTree, productTreeItem, and quantity.

The datatype properties of businessManager are *bManagerID*, *name*, *age*, *teleNr*, *address*, *responsibilities*. Restriction on slots are the restriction on *bManagerID* exactly one, *name* exactly one, *age* exactly one, *teleNr* exactly one, *address* exactly one, *responsibilities* exactly one. All properties and their cardinalities of individual businessManager can be represented in a formal logic language like *bManagerID* exactly 1 AND *name* exactly 1 AND *age* exactly 1 AND *teleNr* exactly 1 AND *address* exactly 1 AND *responsibilities* exactly 1 AND holds MIN 1 businessOrder.

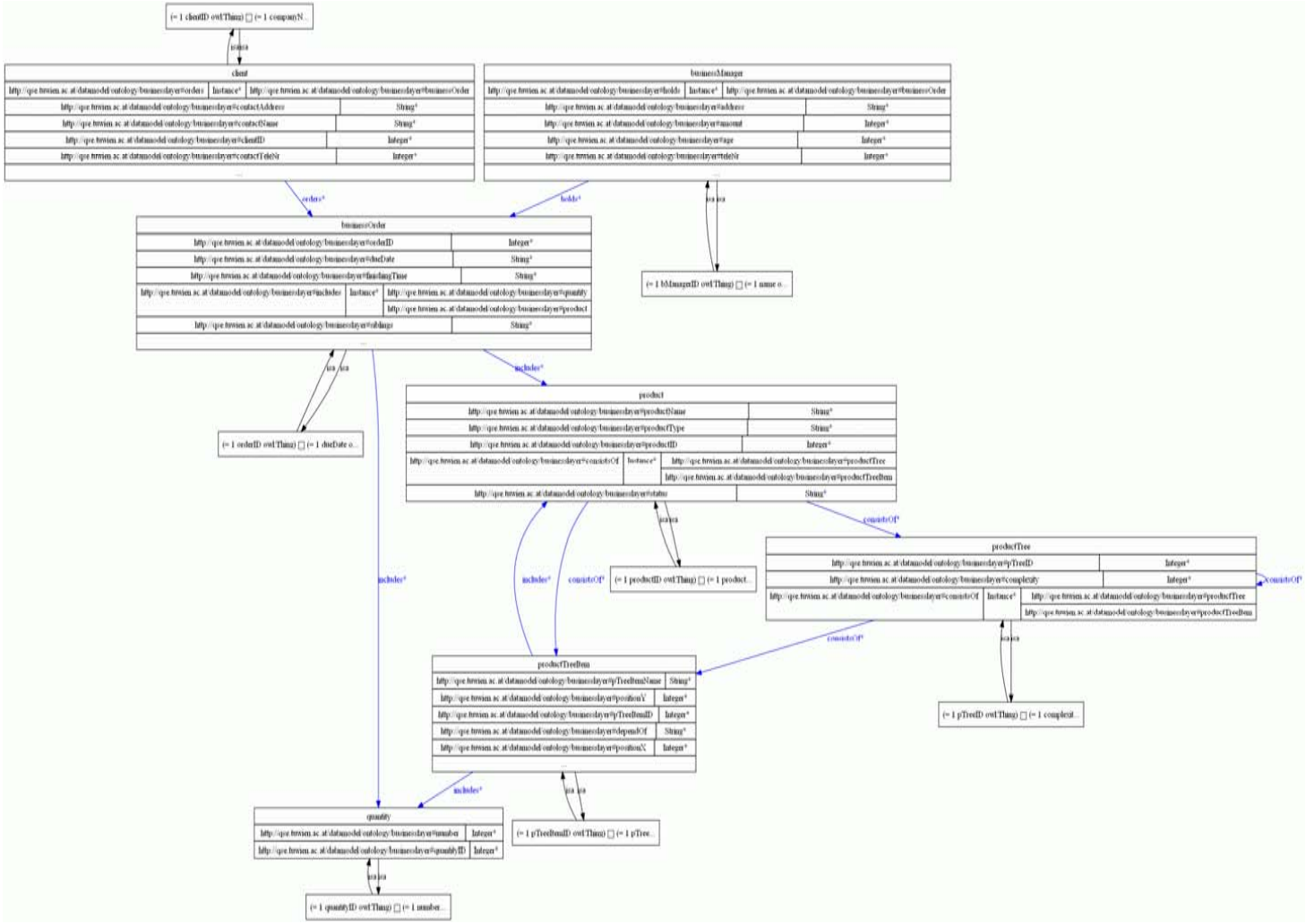

Figure 15: The whole Ontoviz diagram for business manager

The object property *consistsOf* links the individual product to the individual productTree, restriction is product *consistsOf* exactly one productTree, and links the individual productTree to the individual productTreeItem, restriction is productTree *consistsOf* minimum one productTreeItem. The object property *holds* links the individual businessManager to the individual businessOrder, restriction is businessManager *holds* minimum one businessOrder.

The individuals are equal to the entities in UML, such as businessManager, businessOrder etc. The datatype properties are equal to each entity's properties, such as the datatype properties of businessManager are *bManagerID*, *name*, *age*, *teleNr*, *address*, *responsibilities*. Restriction on slots are equal to the *bManagerID* exactly one, *name* exactly one, *age* exactly one, *teleNr* exactly one, *address* exactly one, *responsibilities* exactly one. The object property is equal to multiplicity, such as the object property *consistsOf* links the individual product to the individual productTree.

5.2.2 Plant Manager

This section presents the Ontoviz Diagram for plant manager and its corresponding descriptions.

5.2.2.1 Diagram

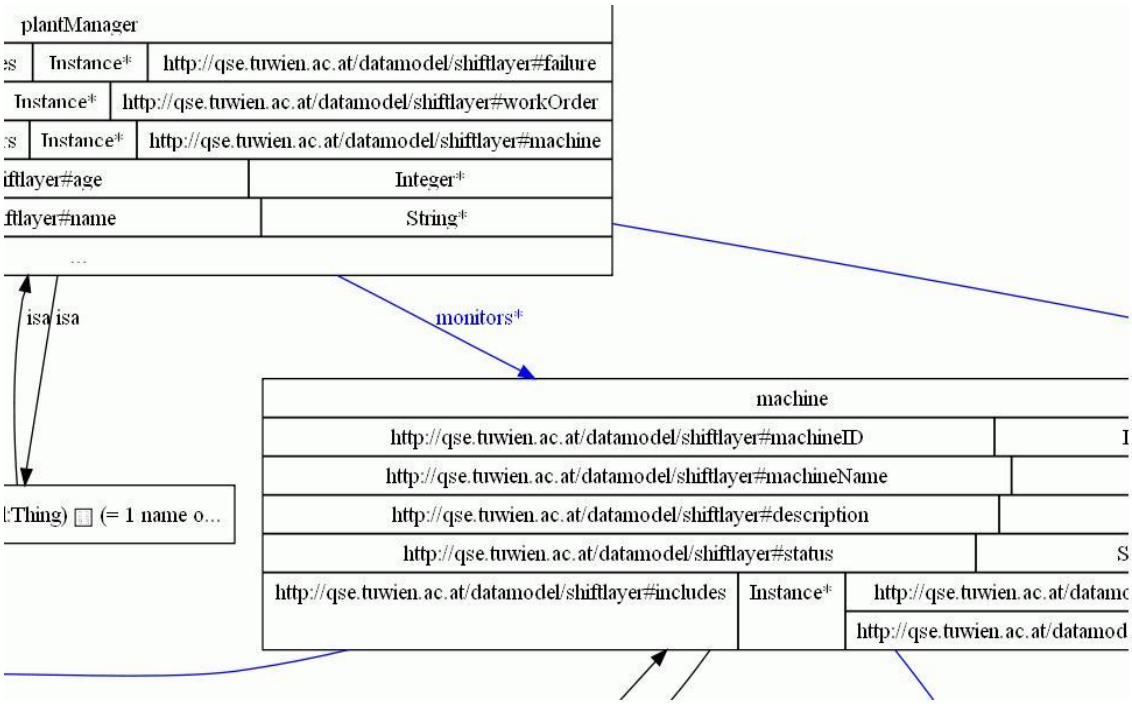

Figure 16: Part of Ontoviz diagram for plant manager

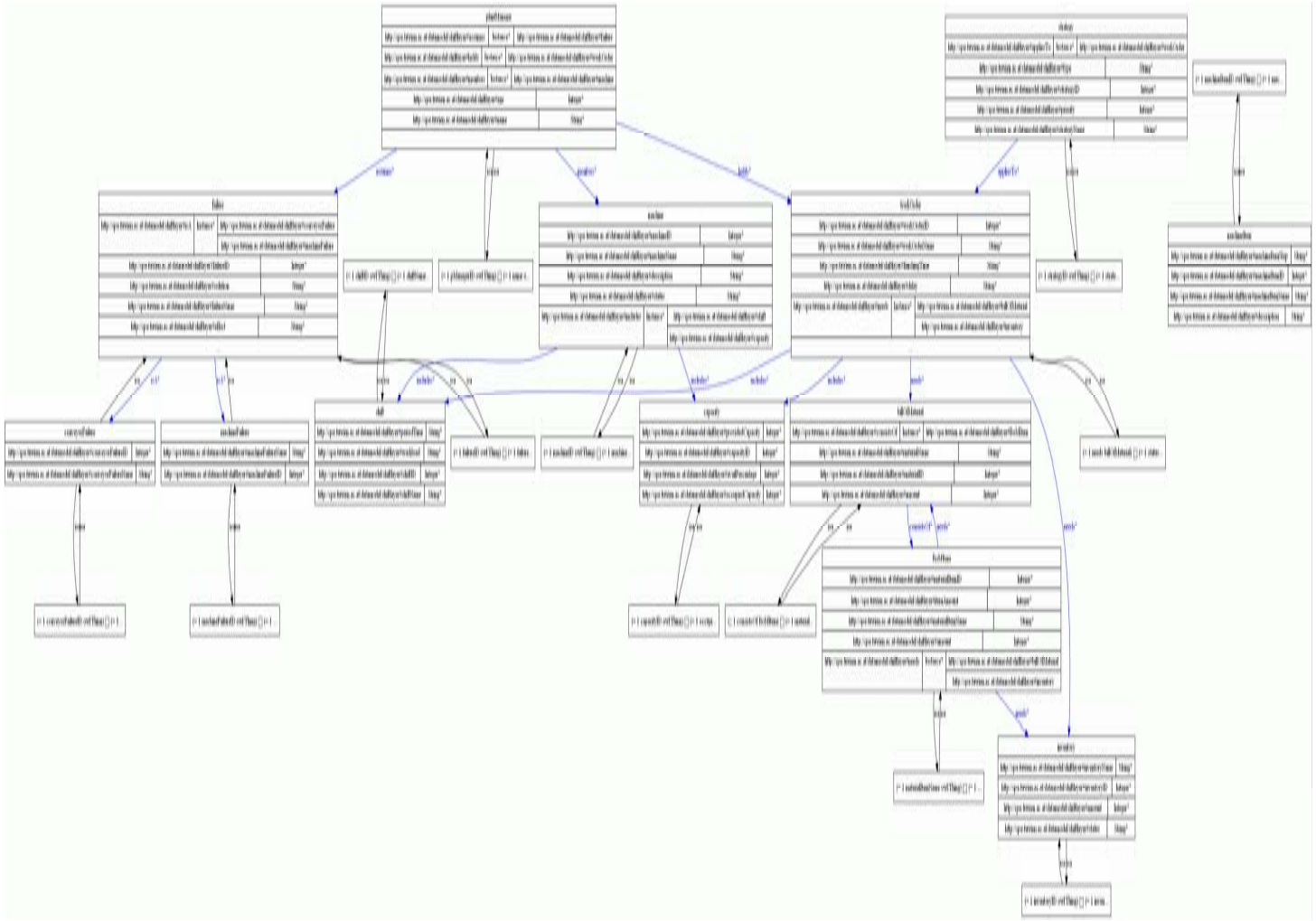

Figure 17: The whole Ontoviz diagram for plant manager

5.2.2.2 Description

The individuals are billOfMaterial, BoMItem, capacity, failure, inventory, machine, machineItem, plantManager, shift, strategy, workOrder.

The datatype properties of plantManager are pManagerID, name, *age*, teleNr, address, *responsibilities*. Restriction on slots are the restriction on *pManagerID* exactly one, *name* exactly one, *age* exactly one, *teleNr* exactly one, *address* exactly one, *responsibilities* exactly one. All properties and their cardinalities of individual plantManager can be represented in a formal logic language like *pManagerID* exactly 1 AND *name* exactly 1 AND *age* exactly 1 AND *teleNr* exactly 1 AND *address* exactly 1 AND *responsibilities* exactly 1 AND holds MIN 1 workOrder AND assumes MIN 1 failure AND monitors MIN 1 machine.

The object property *monitors* links the individual plantManager to the individual machine, restriction is plantManager monitors minimum one machine.The object property *holds* links the individual plantManager to the individual workOrder, restriction is plantManager holds minimum one workOrder. The object property *assumes* links the individual plantManager to the individual failure, restriction is plantManager assumes minimum one failure.

The individuals are equal to the entities in UML, such as plantManager, workOrder etc. The datatype properties are equal to each entity's properties, such as the datatype properties of plantManager are *pManagerID*, *name*, *age*, *teleNr*, *address*, *responsibilities*. Restriction on slots are equal to the *pManagerID* exactly one, *name* exactly one, *age* exactly one, *teleNr* exactly one, *address* exactly one, *responsibilities* exactly one. The object property is equal to multiplicity, such as the object property *monitors* links the individual plantManager to the individual machine.

5.2.3 Shop Manager

This section presents the Ontoviz Diagram for shop manager and its corresponding descriptions.

5.2.3.1 Diagram

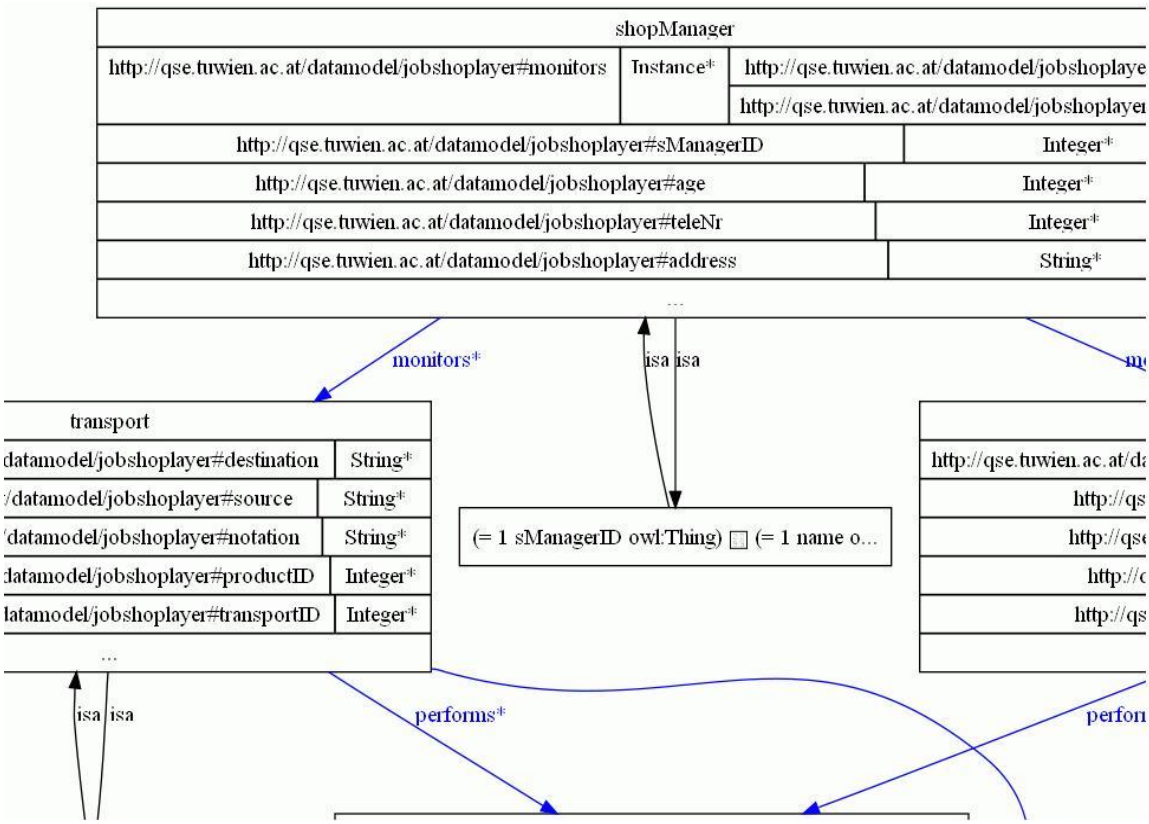

Figure 18: Part of Ontoviz diagram for shop manager

5.2.3.2 Description

The individuals are arrivalSequence, machine, machineItem, shopManager, testcase, transport.

The datatype properties of Shop Manager are *sManagerID*, *name*, *age*, *teleNr*, *address*, *responsibilities*. Restriction on slots are the restriction on *sManagerID* exactly one, *name* exactly one, *age* exactly one, *teleNr* exactly one, *address* exactly one, *responsibilities* exactly one. All properties and their cardinalities of individual shopManager can be represented in a formal logic language like *sManagerID* exactly 1 AND *name* exactly 1 AND *age* exactly 1 AND *teleNr* exactly 1 AND *address* exactly 1 AND *responsibilities* exactly 1 AND monitors MIN 1 machine AND monitors MIN 1 transport. The datatype properties of transport are *transportID*, *productID*, *source*, *destination*, *description*, *notation*. Restriction on slots are the restriction on *transportID* exactly one, *productID* exactly one, *source* minimum one, *destination* minimum one, *notation* exactly one.

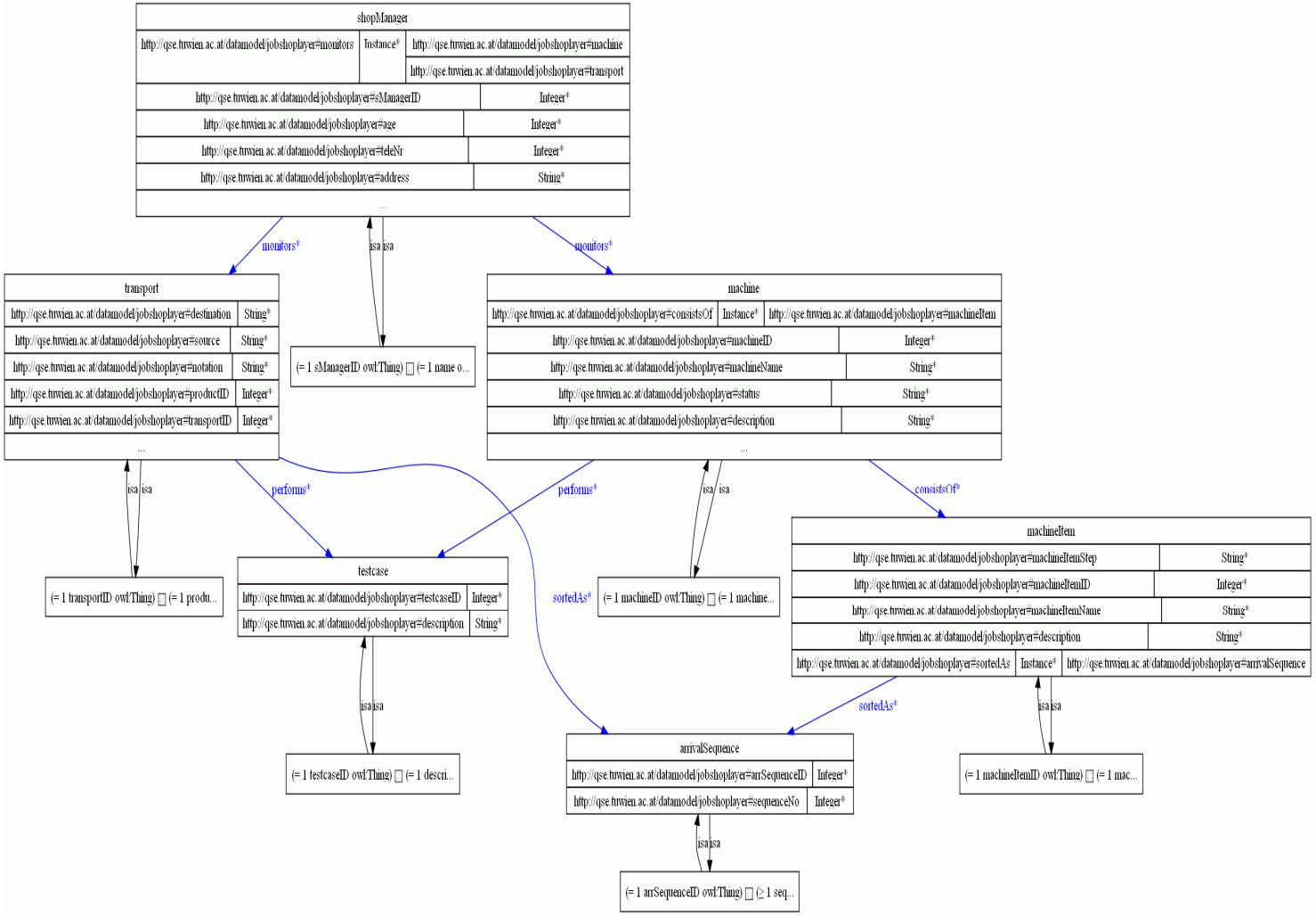

Figure 19: The whole Ontoviz diagram for shop manager

The object property *monitors* links the individual shopManager to the individual machine, restriction is shopManager *monitors* minimum one machine, and links the individual shopManager to the individual transport, restriction is shopManager *monitors* minimum one transport

The individuals are equal to the entities in UML, such as shopManager, transport etc. The datatype properties are equal to each entity's properties, such as the datatype properties of shopManager are *sManagerID*, *name*, *age*, *teleNr*, *address*, *responsibilities*. Restriction on slots are equal to the *sManagerID* exactly one, *name* exactly one, *age* exactly one, *teleNr* exactly one, *address* exactly one, *responsibilities* exactly one. The object property is equal to multiplicity, such as the object property *monitors* links the individual shopManager to the individual machine and links the individual shopManager to the individual transport.

5.2.4 Operation Manager

This section presents the Ontoviz Diagram for operation manager and its corresponding descriptions.

5.2.4.1 Diagram

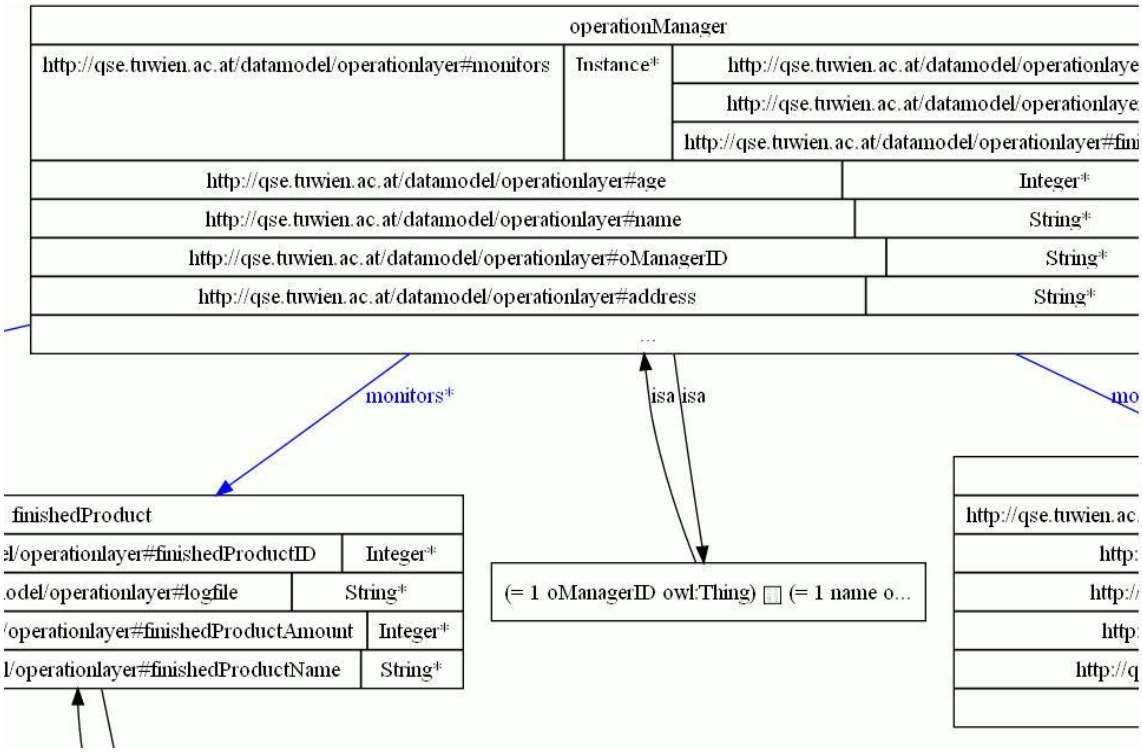

Figure 20: Part of Ontoviz diagram for operation manager

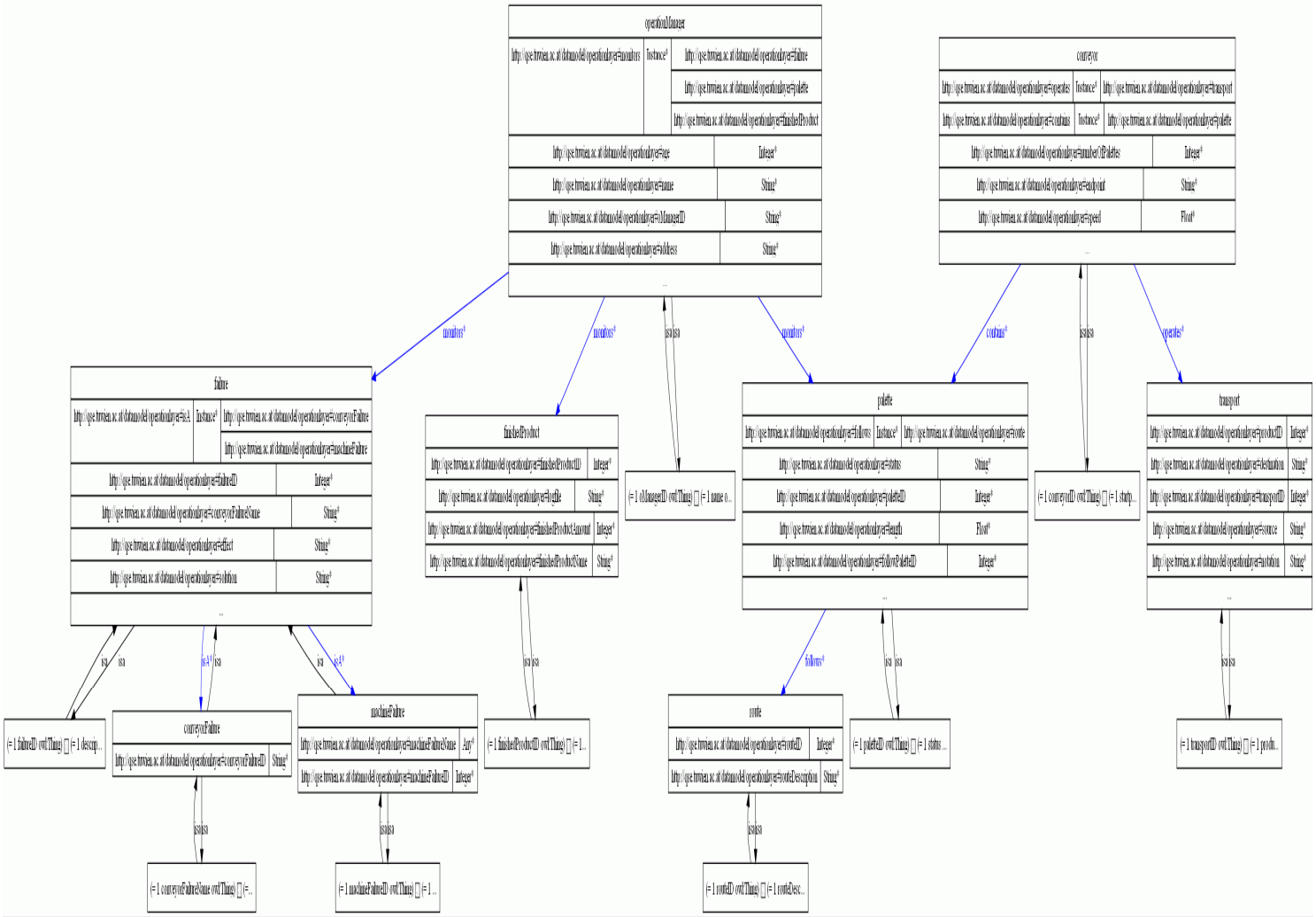

Figure 21: The whole Ontoviz diagram for operation manager

5.2.4.2 Description

The individuals are conveyor, failure, finishedProduct, operationManager, palette, route, transport.

The datatype properties of operationManager are *oManagerID*, *name*, *age*, *teleNr*, *address*, *responsibilities*. Restriction on slots are the restriction on *oManagerID* exactly one, *name* exactly one, *age* exactly one, *teleNr* exactly one, *address* exactly one, *responsibilities* exactly one. All properties and their cardinalities of individual operationManager can be represented in a formal logic language like o*ManagerID* exactly 1 AND *name* exactly 1 AND *age* exactly 1 AND *teleNr* exactly 1 AND *address* exactly 1 AND *responsibilities* exactly 1 AND monitors MIN 1 failure AND monitors MIN 1 palette AND monitors SOME finishedProduct. The datatype properties of finishedProduct are *finishedProductID*, *finishedProductName*, *finishedProductAmount*, *logfile*. Restriction on slots are the restriction on *finishedProductID* exactly one, *finishedProductName* exactly one, *finishedProductAmount* exactly one, *logfile* exactly one.

The object property *monitors* links the individual operationManager to the individual failure, restriction is operationManager *monitors* minimum one failure, and links the individual operationManager to the individual palette, restriction is operationManager *monitors* minimum one palette, and links the individual operationManager to the individual finishedProduct, restriction is operationManager *monitors* some finishedProduct.

The individuals are equal to the entities in UML, such as operationManager, transport etc. The datatype properties are equal to each entity's properties, such as the datatype properties of operationManager are *oManagerID*, *name*, *age*, *teleNr*, *address*, *responsibilities*. Restriction on slots are equal to the *oManagerID* exactly one, *name* exactly one, *age* exactly one, *teleNr* exactly one, *address* exactly one, *responsibilities* exactly one. The object property is equal to multiplicity, such as the object property *monitors* links the individual operationManager to the individual failure, restriction is operationManager *monitors* minimum one failure, and links the individual operationManager to the individual palette, restriction is operationManager *monitors* minimum one palette, and links the individual operationManager to the individual finishedProduct, restriction is operationManager *monitors* some finishedProduct.

5.2.5 System Developer

This section presents the Ontoviz Diagram for system developer and its corresponding descriptions.

5.2.5.1 Diagram

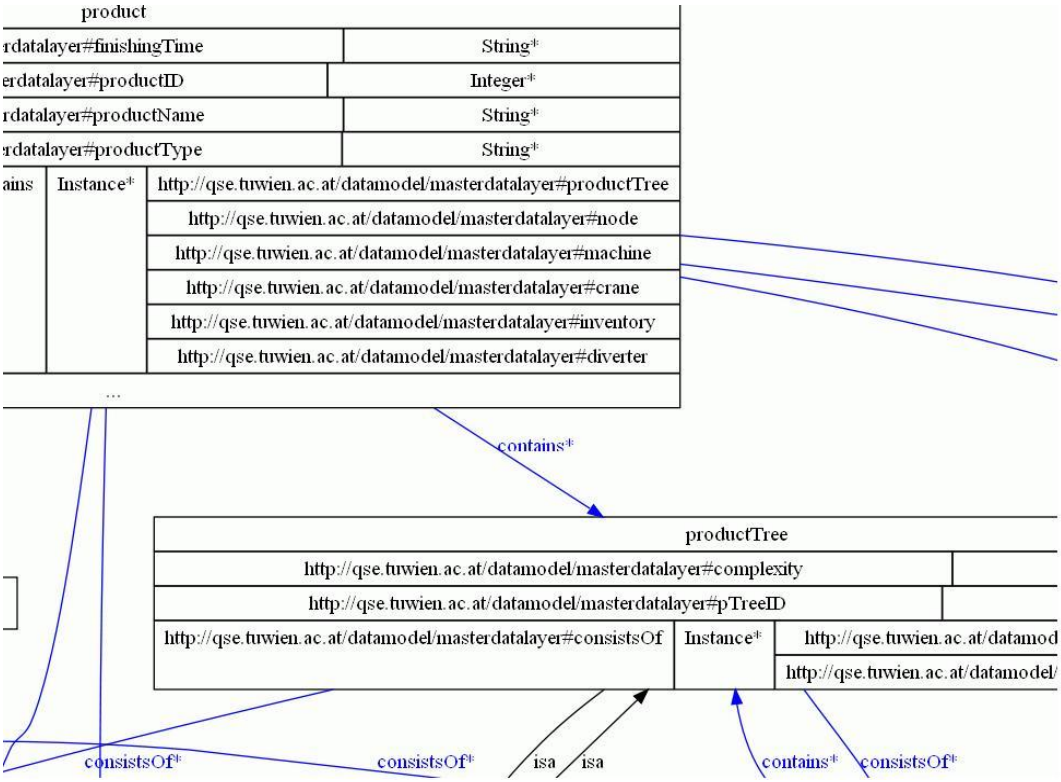

Figure 22: Part of Ontoviz diagram for system developer

5.2.5.2 Description

The individuals are conveyor, crane, diverter, function, inventory, machine, node, prodTreeItem, product, productTree, shopLayout, systemDeveloper.

The datatype properties of systemDeveloper have the properties such as *sysManagerID*, *name*, *age*, *teleNr*, *address*, *responsibilities*. Restriction on slots are the restriction on *sysManagerID* exactly one, *name* exactly one, *age* exactly one, *teleNr* exactly one, *address* exactly one, *responsibilities* exactly one. The datatype properties of product are *productID*, *productName*, *productType*, *finishingTime*, *status*. Restriction on slots are the restriction on *productID* exactly one, *productName* exactly one, *productType* exactly one, *finishingTime* exactly one, *status* exactly one. All properties and their cardinalities of individual systemDeveloper can be represented in a formal logic language like sys*ManagerID* exactly 1 AND *name* exactly 1 AND *age* exactly 1 AND *teleNr* exactly 1 AND *address* exactly 1 AND *responsibilities* exactly 1 AND contains EXACTLY 1 productTree. The datatype properties of productTree are *pTreeID*, *complexity*. Restriction on slots are the restriction on *pTreeID* exactly one, *complexity* exactly one.

The object property *contains* links the individual product to the individual productTree, restriction is product *contains* exactly one productTree.

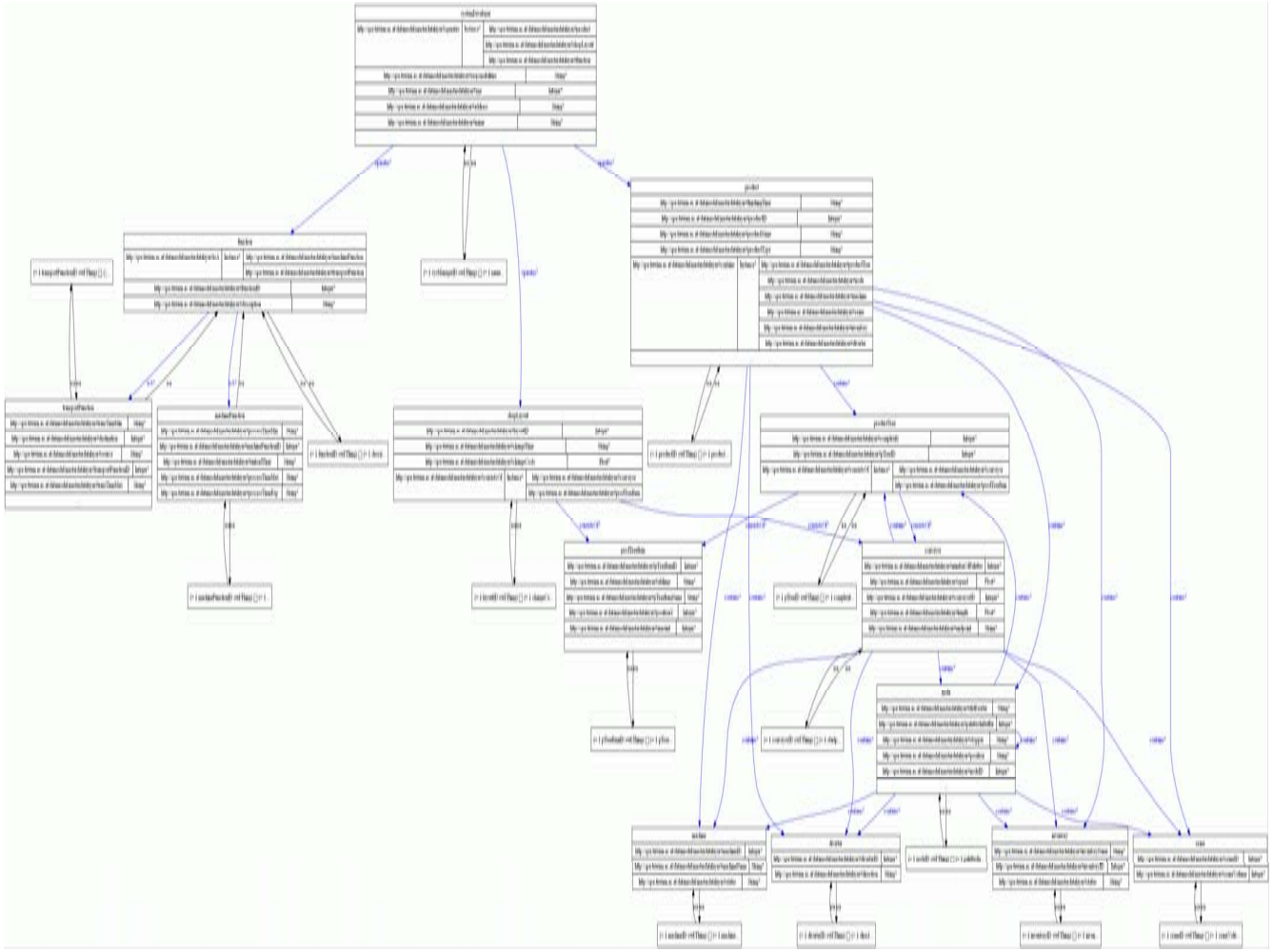

Figure 23: The whole Ontoviz diagram for system developer

The individuals are equal to the entities in UML, such as systemDeveloper, product, productTree etc. The datatype properties are equal to each entity's properties, such as the datatype properties of systemDeveloper are *sysManagerID*, *name*, *age*, *teleNr*, *address*, *responsibilities*. Restriction on slots are equal to the *sysManagerID* exactly one, *name* exactly one, *age* exactly one, *teleNr* exactly one, *address* exactly one, *responsibilities* exactly one. The object property is equal to multiplicity, such as the object property *contains* links the individual product to the individual productTree, restriction is product *contains* exactly one productTree

5.3 Similarities and Differences

This section analyzes and compares four main evaluation criteria regarding UML and ontologies based on the data models created above using these two approaches.

5.3.1 Visualization & Expression

UML provides a wide range of diagrams for visualization, for instance, use case diagrams, class diagrams etc. Users could decide for the suitable diagrams according to their needs, either use case diagram is appropriate for the use case description or class diagram and entity relationship diagram are appropriate for creating data models. In this case with the help of the UML tool Visual Paradigm, the entity relationship diagram is chosen to visualize the use case description. In section 5.1 above, the diagrams shown present all the necessary entities as rectangles in a graph for each role, their corresponding properties are shown inside the rectangles, concluding type, value range etc, relationships between entities and their cardinalities are represented as directed lines connected with the entities.

Ontology also offers the possibility for visualization by installing the ontology tool protégé plug-ins, for instance, Ontoviz, Jambalaya, OWLViz etc. Jambalaya is an interactive visualization tool that helps users to visualize, navigate, understand a sophisticated knowledge based system while OWLViz allows the visualization of asserted and inferred classification hierarchies. In this case with the help of the tool Protégé, the nested view of Jambalaya is chosen to visualize the ontologies of the use cases. In section 5.2 above, the diagrams shown present all the necessary classes and instances as nodes in a graph, their corresponding properties as arcs, concluding type, range, OWL syntax, relationships between concepts and instances shown as directed edges. A node contained within another node indicates a child-parent relationship. Arcs have types as well, roughly corresponding to slots, or properties in OWL. Jambalaya groups arcs (properties or slots) according to their function. If you click on a group of arcs, you can hide them all or show

them all [20].

In comparison, ER diagram from UML and Jambalaya diagram from ontology don't have too much difference basically. Generally speaking, UML offers a more powerful visualization than ontology because of its wide range of specified diagram support and easy use of manipulation on diagrams in UML. Users could choose their preferred diagrams depend on their functions and requirements. On the other hand, ontology is more appropriate for creating a better vocabulary, strong expressiveness on data modeling in knowledge domain than UML. Ontology has its own OWL syntax that show the constraints on the properties, each owl element specify the cardinality of the classes and instances' properties which UML is not able to provide.

5.3.2 Consistency

The editor tool Visual Paradigm guarantees the consistency feature in diagrams for UML, for example, both in the roles plant manager and shop manager could monitor several machines, and meantime each machine is composed of several machineItems. Suppose that the user has finished the establishment of the entities machine, machineItem and the assignment of the properties values for the plant manager. Later the user would like to set a machine or machineItem entity, it is redundant to retype all the properties of machine or machineItem, the point is to keep the entity name consistently. If anything would be changed in either side, it would be adjusted automatically; the both sides remain the same.

Ontology does also guarantee this consistency feature, nevertheless represented in another form. With the assistance of the DIG (Description Logic Implementers Group) interface, it enables the computation and examination on the subsumption relationships between classes and detects inconsistent classes. Reasoners (like Racer) can be used to checks whether or not all of the statements and definitions in the ontology are mutually consistent and can also recognize which concepts fit under which definition.

In comparison, UML provides a set of consistent diagrams for different roles; ontology might provide a set of inconsistent diagrams for different roles triggered by manual causes. Only an entire ontology file could be imported under Protégé's support, an individual element can not be imported into another ontology file directly, so the model engineer would create it manually, the errors might occur meanwhile. The UML users don't have to worry about the consistency problem in diagrams, but the disadvantage is the lack of the model consistent checking while ontology could make it happen [21].

5.3.3 Needed Effort

Establishing the diagrams in UML using Visual Paradigm is easy and

convenient, for example, both in the roles plant manager and shop manager could monitor several machines, and meantime each machine is composed of several machineItems. When the user finishes the establishment of the entities machine, machineItem and their properties in plant manager, it is not necessary to retype the properties of machine and machineItem which saves a lot of time and efforts.

However, it does cost more time and efforts in order to establish the diagrams in ontology using Protégé. For example, the user has to retype the Classes machine, machineItem, their properties and restrictions on properties in both roles plant manager and shop manager, although the user has already typed once. In this case, this kind of procedure raises the possibility of faults caused by human being.

In comparison, it is more wasteful both in time cost and artificial efforts to establish the diagrams in ontology than in UML. Under rough estimation, the time required to establish all the diagrams in UML is twice as much as the time required to establish all the diagrams in ontology considering this example of multi-agent systems. For more complicated situations and systems, the time cost and efforts keep increasing in accordance with the system's complexity. Regarding the time and efforts needed for model checking, ontologies save definitely a lot of time than UML, because of its automatic model consistency checking that has been mentioned in section 5.3.2. For inconsistency discovery, ontology does have dominant advantage.

5.3.4 Additional functions

UML is famous because of its strong support and expressiveness of diagrams. Besides, it provides also other additional functions, like XMI (XML Metadata Interchange) standard which is designed to facilitate the interchange of UML models. Visual Paradigm tool can reverse engineer 9x, C++, Java, IDL, PHP and Python source code, XML and XML schema files, databases (with JDBC), and .NET .exe and .dll files etc.

Protégé supports not only the visualization, but also other pragmatic functions in ontology. Actually, for instance, it provides the possibility to generate XML schemas which are a means for defining the structure, content and semantics of XML documents, express shared vocabularies and allow machines to carry out rules made by people. RDF/XML source code allows RDF models to be sent easily from one computer application to another in a common XML format. In addition it offers also the query tools, for example, SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions, it also supports extensible value testing and constraining queries by source RDF graph [10][30][47].

In comparison, Protégé offers more plug-ins and additional functions in ontology than Visual Paradigm in UML. It enhances the strong ability in ontology and widens its user range. However, UML is constrained by diagram functions which narrow its popularity to some degree.

6. Evaluation

The section 6 characterizes and evaluates the fundamental features of UML and OWL in four main aspects, visualization & expression, consistency, performance and additional functions.

6.1Visualization & Expression

This section will present the evaluation results of UML and OWL in both common and uncommon features respectively.

6.1.1 Evaluation table of common features

Table 1: The evaluation table of common features

UML features	OWL features
class	class
instance	individual
ownedAttribute binary association	property
Subclass generalization	Subclass subproperty
N-ary association association class	Class property
enumeration	oneOf
navigable, non-navigable	Domain range
disjoint, cover	disjointWith unionOf
multiplicity	minCardinality maxCardinality inverseOf
package	ontology
dependency	reserved name RDF: properties

6.1.2 Description

Table 1 shows the equivalent features both in UML and OWL. For instance, UML and OWL are all based on *class*. In UML it is a basic element stand for an object, it could be any person, concept to the system, while in ontology class

consists of the ontology library.

In UML the extent of a class is denoted as a set of *instances*, corresponding, in OWL the extent of a class is a set of *individuals* [16].

An association represents the relationship among classes. There are two kinds of association: *binary association* and *n-ary association*. In UML *binary association* is the structural connection between two classes, appears as a straight line while in OWL it is represented as *owl:ObjectProperty*. There are two kinds of property in OWL, such as *owl:ObjectProperty* and *owl:DatatypeProperty*. The *ownedAttribute* indicates a bundle of owned properties of a class, if the *ownedAttribute* is a unique key, then it is equivalent to *owl:ObjectProperty*, otherwise it is equivalent to *owl:DatatypeProperty* [16].

Both two languages support the subclass relationship, it is represented as *generalization* in UML and *rdfs:subClassOf* in OWL, regarding the property hierarchies, it is represented as *generalization of association* in UML and *rdfs:subPropertyOf* in OWL [16].

The other association *n-ary association* in UML is the connection among three or more classes, appears as a set of lines connected to a central diamond. The similar element is OWL classes with bundles of binary *owl:FunctionalProperty* [16].

The enumeration of the individuals or instances that consist of the class is denoted as *enumeration* in UML and in OWL is *owl:oneOf* [16].

UML has two options for binary associations are navigable that means the association can be traversed or queried and non-navigable that means the association is bi-directional. OWL properties are always binary and with two distinguished ends, one is *rdfs:domain* that limits the individuals to which the property can be applied and *rdfs:range* that limits the individuals that the property may have as its value [16].

Both UML and OWL allow multiple inheritances, that is to say, a class can be a subclass of more than one class. *Disjoint* of the subclasses in UML is equal to *owl:disjointWith* statement asserts that two subclasses of a class involved have no individuals in common. The concept cover which a collection of subclasses to be declared to cover a super class is equal to the *owl:unionOf* property links a class to a list of class descriptions. *An owl:unionOf* statement describes an anonymous class for which the class extension contains those individuals that occur in at least one of the class extensions of the class descriptions in the list [16].

Multiplicity in UML means that an association can have minimum and maximum cardinalities. In comparison with OWL, a property can be constrained by cardinality restrictions such as the minimum number of instances *owl:minCardinality*, and maximum number of instances *owl:maxCardinality.* The *owl:inverseOf* construct can be used to define such an inverse relation between properties, from domain to range and vice versa [16].

The module structure called *package* in UML that organizes elements in

groups, establishing ownership of elements etc and an OWL *ontology* may include such elements as descriptions of classes, properties and their instances etc [16].

In UML the relationship that the change of a supplier element may affect the client's element is represented as *dependency* and could be translated to a reserved name *rdf: properties* whose domain and range are both rdf:class in OWL [16].

6.1.3 Evaluation table of uncommon features

Table 2: The evaluation table of uncommon features

Feature	**UML**	**OWL**
Boolean combination of class expressions	No	intersectionOf, unionOf, complementOf
Property value restriction	No	hasValue, equivalentClass, allValuesFrom, someValueFrom
Unique name assumption	No	allDifferent, sameAs, differentFrom
Unique name assumption of classes and properties	No	equivalentClass, equivalentProperty
Thing element	No	Yes
Symmertic/Transitive Property	No	Yes
Behavioral features	operations, responsibilities, interface/abstract/active classes	No
Composite structure	structured classifier, connector, collaboration, port	No
Part-of relationship	composition, aggregation	No
Derive element	Yes	No
Access control	public, protected, private, package	No
Keywords	<<interfaces>>	No

6.1.4 Description

Table 2 shows the evaluation of uncommon features in UML and OWL. There are some features existed in OWL but are not included in UML, vice

versa. For instance, OWL provides additional boolean combination of class expressions, such as the *owl:intersectionOf* element defines that the class consists of exactly all the objects that are common to all class expressions from the list, is similar to the logical conjunction, the *owl:unionOf* element defines that the class consists of all the objects that belong to at least one of the class expression from the list, is similar to the logical disjunction. The *owl:complementOf* element defines that the class consists of exactly all objects that do not belong to the class expression, is similar to the logical negation. Both *owl:unionOf* and *owl:complementOf* are not parts of OWL Lite [16].

In additional, OWL provides some kinds of property value restrictions, such as the *owl:hasValue* element defines that a property must have at least one value which could be either an individual or a data value. The *owl:allValuesFrom* element requires the particular class has a local range restriction associated with it and the *owl:someValuesFrom* restriction requires the particular class may have a restriction on a property that at least one value for that property is of a certain type [16].

OWL does not have this unique name assumption, but it allows expressing explicitly that two names refer to distinct entities. For example, the *owl:sameAs* element states that two given named individuals have the same identity, by contrast, the *owl:differentFrom* element states that two given named individuals have different identities. Furthermore, the *owl:AllDifferent* element states that a number of individuals should be mutually distinct [16].

Regarding the unique name assumption of classes and properties in OWL, it also allows to express explicitly. The *owl:equivalentClass* element is used to indicate that two classes have precisely the same instances, and the *owl:equivalentProperty* element indicates that two properties are the same.

It is special and only in OWL that a universal class *owl:Thing* is available, all classes are subclasses of *owl:Thing* [16].

There are two subclasses of *owl:ObjectProperty*, one is the *owl:SymmetricProperty* element asserts that a property is symmetrically valid in two directions. The other is the *owl:FunctionalProperty* element asserts that a property can only have one unique value or semantically equal [16].

Considering UML, it provides the behavioral features additionally, including *operation* that is a specification of a transformation or query that an object may be called to execute; *responsibility* that indicates the obligation of a class or other elements, expressed as comments; *abstract class* indicates that a class may not be instantiated; *active class* is the class whose instances are active objects etc [16].

The *composite structure* is also a special feature of UML that describes the interconnection of objects within a context to form an entity. *Composite structures* include structured *classifiers* and *collaborations*, *connectors* and *ports* etc [16].

There are several kinds of part-of relationship between classes, for example, an *aggregation* specifies a whole-part relationship between an

aggregate and a constituent part. A *composition* relationship represents a whole–part relationship and is a type of aggregation [16].

In the aspect of access control, UML identifies four types of visibility for each attribute and operation, for instance *public*, *protected*, *private*, and *package*.

UML allows a target element to be derived from other source elements, denoted with the keyword *derive*, for example a composition derived from a generalization [16].

A UML *keyword* is a textual adornment to categorize a model element. Most *keywords* are shown in «». The *keyword* categorizes that a classifier box is an interface shown as «interface» [16].

6.1.5 Conclusion

In a word, OWL supports more features than UML, such as *boolean combination of class expressions*, *property value restriction*, *unique name assumption*, *thing element* and *symmetric/transitive property*. The intent of OWL is to produce machine readable codes, process and integrate information automatically. Since OWL semantics are based on description logics, relation to description logics has to be represented as *intersection*, *union*, and *complement*. *Property value restriction* limits the value range of classes in OWL while in UML the value of classes are shown in other forms, like *aggregation*, *composition* etc. *Unique name assumption* is not defined in OWL, so it is necessary to mark them explicitly. In UML there is no need for *unique name assumption*, because all the classes are defined by default that two equivalent class names refer to the same class. In the theory of ontology, all other individuals are defined to derive from *thing element* that in comparison not necessary in UML that based on object-oriented data modeling primarily. *Symmetric/transitive property* is the typical features of logical language, so they are not supposed to appear in UML [16].

By contrast, UML supports additional features than OWL, such as *behavioral features*, *composite structure*, *part-of relationship*, *derive element*, *access control*, *keywords*. UML is designed to be a graphical notion modeling language originally, facilitates the human beings to understand the system model. UML's statechart diagrams are used to describe dynamic behaviors of a system, such as the possible states of an object as events occur. Therefore, OWL is a knowledge representation of ontology, provides a static ontology vocabulary, thus *behavioral features* are surplus in OWL. *Composite structure* is a set of interconnected elements that collaborate at runtime to achieve some purpose. *Relationships* in UML are for instance aggregation, association, composition that in OWL are represented as *boolean combination of class expressions*. Classes, variables and methods could be declared as four kinds of *access controls public*, *private*, *protected*, and *package* in UML that could be in accordance with java syntax, enables the UML tool to generate

corresponding consistent codes. Otherwise, OWL is written in RDF/XML codes, so it is useless to define access controls. *Keywords* in UML could be used to distinguish abstract class and interface, the aim is also to be in accordance with java syntax [16].

6.2Consistency

This section will present the evaluation results of UML and OWL in the diagram or model consistency aspects.

6.2.1 Evaluation table of consistency

Table 3: The evaluation table of consistency features

Consistency	UML	Ontology
Diagram consistency		
1. Class consistency	Yes	No
2. Attribute consistency	Yes	No
3. Relationship consistency	Yes	No
Model consistency		
1. Class consistency	No	Yes
2. Property consistency	No	Yes
3. Instance consistency	No	Yes

6.2.2 Description

Table 3 shows the evaluation of the consistency checking features in UML and ontology.

For instance, regarding the diagram consistency, UML tool Visual Paradigm provides the diagram consistency checking that ontology tool Protégé doesn't provide it. Visual Paradigm keeps all the classes' names and their attributes consistent, if any change occurs, they will be changed automatically. Suppose there exists a kind of relationship between two classes, if one of the both classes are deleted, the relationship would be deleted automatically.

Regarding the model consistency, UML doesn't provide this kind of function. In OWL it doesn't allow to create two individuals/properties with the same name, furthermore, the tool Protégé could examine whether the required instance has been created. With installation of an additional plugin Racer (Renamed Abox and Concept Expression Racer), ontology could carry out model consistency checking [34].

The description logic reasoner offers the possibility of OWL to deduce their correct logical inheritance in order to determine whether a class is consistent

or not, decide whether one class is subsumed by another or not, etc [34].

Racer is so to say a robust server for scalable ontology reasoning. The theory of Racer is to combine large Aboxes with large and expressive Tboxes. It provides highly optimized standard and non-standard inference services for sophisticated ontology applications. Racer offers much more than OWL by supporting rules, constraint reasoning, and expressive query answering (e.g., in SPARQL syntax), etc [34].

6.2.3 Conclusion

UML provides *class consistency*, *attribute consistency* and *relationship consistency checking* in diagram checking. UML aims at to offer a strong robust diagram system, so that it doesn't allow appearing inconsistency in different diagrams. In OWL, all the diagrams are generated automatically based on the created ontology models, thus for OWL it is more important to keep models consistent.

OWL provides *class consistency*, *property consistency* and *instance consistency checking* in model checking. UML doesn’t provide model consistency checking basically, but this function could be realized through one additional plug-in "MCC". More details would be explained later in section 6.4.

6.3Needed Effort

This section will present the evaluation of UML and OWL's needed Effort for creation and consistency checking.

6.3.1 Evaluation table of needed effort

Table 4: The evaluation table of needed effort

Function	Time cost in UML	Time cost in Ontology
Creation		
Creation of classes	1 min/class	1 min/class
Creation of attributes	2 min/attribute	4 min/attribute
Creation of relationships	0.5 min/relationship	4 min/relationship
Creation of instances	No	2 min/instance
Diagram export	1 min/diagram	5 min/diagram
Creation of one agent	1 hour/agent	2 hour/agent
Creation of five agents	3 hours/five agents	6 hours/five agents
Consistency checking		
Class consistency	0.5 min/class	0.5 min/class
Attribute consistency	0.5 min/attribute	0.5 min/attribute
Relationship consistency	0.5 min/relationship	0.5 min/relationship

Instance consistency	No	1 min/instance

6.3.2 Description

Table 4 shows the evaluation of the functional performance in UML and OWL. For instance, creation of one class in UML is quite easy, the user is required to create an entity and rename it, and this operation needs 1 minute. In comparison with OWL in Protégé, creation of one class needs also 1 minute. The user is first required to click "create subclass of thing" and then rename it.

In creation of one attribute in UML, the user is required to add "new column" to the corresponding entity, and then click "open column specification", fill out some mandatory options like name, type, length etc. It requires the user about 2 minutes. In comparison with OWL in Protégé, the user has to first create a "datatype property" in Property Browser and rename it, input its domain and range. Then the user turns back to "OWL Classes", click "edit class description" in order to assign the "datatype property" to the corresponding class, input its attributes and their cardinality in OWL syntax. All operations require the user about 4 minutes.

In creation of one relationship in UML, the user is required to choose the right relation among a few options, like "one to one relationship", "one to many relationship", "many to many relationship" etc, and then assign this relation to two appropriate entities. It requires about 0.5 minute. In comparison with OWL in Protégé, the user has to first create an "object property" in Property Browser and rename it, input its domain and range. Then the user turns back to "OWL Classes", click "edit class description" in order to assign the "object property" to the corresponding class, input its attributes and their cardinality in OWL syntax. All operations require the user about 4 minutes.

In creation of one diagram for each agent, it is convenient to click export "Diagrams as Image", and then the user is required to choose "output destination", "export type", "quality" etc. All operations require about 1 minute. In comparison with OWL in Protégé, considering creation of the diagram, the user has to choose the right tab under "project configure", there are a few diagram options, such as Jambalaya Tab, Ontoviz Tab or OWL Viz Tab etc. It requires the user about 5 minutes.

In creation of one agent in UML, for about 7 entities, 30 attributes of the agent business manager, reference in section 5.1.1, the needed effort for all the operations requires about 1 hour. In comparison with OWL in Protégé, it requires about 2 hours.

In creation of five agents in UML, for about 50 entities, 200 attributes of five agents business manager, plant manager, shop manager, operation manager and system developer, reference in section 5.1.1, 5.1.2, 5.1.3, 5.1.4 and 5.1.5. This needed effort requires about 3 hours. In comparison with OWL in Protégé, it requires about 6 hours.

Regarding consistency checking in general, the needed effort for class

consistency checking in OWL is 0.5 min/class, attribute consistency requires 0.5 min/attribute, relationship consistency requires 0.5 min/relationship, instance consistency requires 1 min/instance. Similarly, in UML class consistency checking requires 0.5 min/class, attribute consistency requires 0.5 min/attribute, relationship consistency requires 0.5 min/relationship. Unfortunately, there is no instance defined in UML diagram, so it is impossible to compare the needed effort.

6.3.3 Conclusion

The needed effort for creation of classes both in UML and OWL are almost equal. The difference occurs on creation of attributes and creation of relationships, in UML the time cost is shorter than in OWL. For in OWL, for example, creation of a class requires the user to form all the attributes and their cardinality in logic expressions which cost more time than UML. UML is a graphical language, its models are all produced as diagrams, and by contrast, in OWL the user has to decide for what kind of diagram type to generate that needs more time.

Suppose creation of one agent as one unit in comparison with creation of five agents as five units, the efforts invested in creation of models in OWL raise according to the project's range.

The needed effort for consistency checking in parts class consistency, attribute consistency and relationship consistency are no big difference between UML and OWL. But OWL has more advantages of import a few ontology models and checks their consistency simultaneously.

6.4 Additional functions

This section will present the evaluation of UML and OWL in additional functions.

6.4.1 Evaluation table of additional functions

Table 5: The evaluation table of additional functions

Reasoning	**UML**	**Ontology**
Check consistency	in Tool MCC feasible	Yes
Classify taxonomy	in Tool MCC feasible	Yes
Compute inferred types	in Tool MCC feasible	Yes
SPARQL Query	in Tool MCC feasible	Yes
Codes		
Generate EMF Java Interfaces	Yes	Yes
Generate Java Schema class	Yes	Yes

Generate Java code	Yes	Yes
Show DIG Code	No	Yes
Show RDF/XML Source code	Yes	Yes

6.4.2 Description

Table 5 shows the evaluation of additional features both in UML and OWL. In the reasoning part, Protégé provides reasoning plug-in to check consistency, classify taxonomy, compute inferred types, and SPARQL Query allows for a query to consist of triple patterns, conjunctions, disjunctions, and optional patterns, could be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware.

In the codes part, both UML and Protégé provide generation of EMF Java interfaces, Java schema class and RDF/XML Source code. In additional, Protégé provides Java code, and DIG code.

6.4.3 Conclusion

The plug-in reasoning engine Racer (Renamed ABox and Concept Expression Reasoner) supports the ontology editor Protégé to verify the model consistency of ontologies. The Racer guarantees such functionalities as check the model consistency of OWL ontologies, discover implicit subclass-superclass relationships, ensure consistent resources, and enable an OWL-QL query processing system.

The possibility to realize the automatic consistency checking of UML models is offered by the tool MCC (Model Consistency Checker). It is based on Description Logic, implemented as a plug-in of Poseidon for UML, applies Racer as the reasoning engine. The MCC consists of the three components like Fact Extractor, Visual Query Interface and Query Processor basically and link the Poseidon for UML to Racer. First, the Poseidon loads the MCC plug-in and any UML model can be loaded by the Fact Extractor while any user events can be loaded by the Visual Query Interface. The Query Processor translates the model consistency analysis from a user-friendly and communicates with the Racer, and then it translates the reasoner results from Racer to well understandable information for users.

The similarities of UML and Ontology are that both automatic model consistency checking are implemented by the Racer finally, it means no matter the models are checked directly or converted to be verified by the Racer.
The differences are that since the Ontology models has a logic semantics and be expressed machine readable, so it can be executed without any transformation. The UML models are always shown in the UML diagrams, such as class diagrams. The class diagrams should be translated first into the Description logics.

7. Didactics in ontology-based modeling

UML has been payed a close attention for a long time in software engineering. Since it is widespread and has become an indispensable didactics part in high schools and universities. There is already a wide range of didactic teachings of UML available. So now it is important to suggest some didactics improvements on ontology-based modeling.

7.1 Definition of didactic elements

At the beginning of didactics planning, it is absolutely necessary to answer several relevant questions of didactics in general. The followings define 9 characteristic elements, namely the primary questions of didactics:

1. Teacher

This part of didactics is appropriate for teachers whose responsibilities are for software engineering and knowledge engineering learning or junior teachers who have less experience in ontology-based modeling teaching.

2. Content

The contents of this ontology learning should conclude two main parts, the first part is didactics of basic ontology's theories, and the second part is didactics of ontology-based modeling in practice. The more details of each lecture unit would be described in section 7.2.

3. Time

The time schedule of this ontology-based modeling didactics is supposed to last for one semester. Each lecture unit is supposed to last for 1.5 hour.

4. Students

The learners are aimed at students of high-schools and universities niveau or model engineers and software engineers who are interested in data modeling using ontologies, and software engineers with knowledge in the traditional data modeling area who want to analyze the advantages and possible limitations of switching to ontology-based modeling approach. Knowledge of data modeling in advance is required, meanwhile, knowledge of ontology is not required, but knowledge of UML is preferred, and knowledge of software/knowledge engineering is also a plus.

5. Location

The location could be classrooms of high schools or universities, vocational schools, and meeting rooms of companies' train bases etc. The practical exercises could take place in computer labs.

6. Method

The teacher can use traditional didactic methods like presentation, face-to-face explaining, demonstrating as the main techniques. Otherwise, it is also commendatory that the teacher could use questioning, brain-storming, collaborating and learning by teaching etc. The didactic methods are dependent of students' abilities and levels of classes.

7. Medium

The teacher can use such media like slides, beamer as the main media tools for presentation, questioning and so on. It is commendatory for the teacher to apply the e-learning platform "moodle" [9]. For example, the teacher could upload presentation slides and learning material, creating forum also increases the communication possibilities for teacher and students.

8. Learning motivation

Ontology has become one of potential data modeling languages in software engineering. The learning motivation is the effectiveness of ontology-based data modeling based on its automatic consistency verification and strong logic expressivity used in the data modeling process.

9. Educational objective

The target of ontology-based modeling didactics is to provide students and interested engineers a fundamental overview of ontology and command of ontology-based data modeling by means of the tool Protégé.

7.2 Structure of didactics material

This section plans each unit of lectures for ontology didactics approximately and each unit comprises the whole didactics material required and depicts the involved relevant didactic elements.

7.2.1 Overview of ontology

1. Motivation for students

This part is to introduce the overview of ontology to students. For students

without previous ontology knowledge, it is good to know about ontology's history, definition and application ranges etc in the beginning.

2. Content

The content should contain ontology's history, including its origin and development phases. Another part is definition of ontology, especially should point out three primary components of ontology. For application ranges, it is easy for students to memory by enumeration of some concrete examples of applications.

3. Precondition of students

Basically no specific knowledge is required, but knowledge of software engineering (UML) or knowledge engineering is an advantage for both students and teacher.

4. Method

At the beginning it is commendatory for the teacher to use brain-storming in order to test the level of students which facilitates the continued lecture plan. After obtaining students' feedbacks, the teacher could use the traditional teaching method, namely face-to-class presentation.

5. Medium

The teacher could use some flip charts or cards during brain-storming, moreover, he could use slides and beamer to present his presentation.

6. Exercises for students

It is benefit for students to solidify the knowledge they have learned from this lecture unit. Two students make up one team, their tasks are to find out an example of ontology in practice by themselves. Each team should describe the scenario of this example, explain the reasons of their choice, decompose and analyse their components and usage, etc. Length: 2-3 pages.

7. Time cost

The lecture unit is supposed to be 1.5 hour. The time cost by the exercises for each team is supposed to be 1-1.5 hour.

7.2.2 Theory enhancement

1. Motivation for students

The first step "overview of ontology" serves a good preparation for students. Since ontology is an important theory in knowledge engineering and now it takes a relevant place in software engineering by and by. So it is demandable for students to enhance and command the knowledge of ontology.

2. Content

In order to enhance the memory of ontology, it is relevant for students to practice the knowledge they have learned before by themselves. One side, presentation of more details about ontology for the teacher; on the other side, brain-storming for students to find out the differences between UML and Ontology, focus only on the principal theories.

3. Precondition of students

A few knowledge of ontology is required; moreover, knowledge of UML is an advantage for students and the teacher.

4. Method

The teacher could use presentation to state more details about ontology in software engineering. In addition, face-to-face explaining and questioning are also useful in case any student has questions. At last, brain-storming will be applied for the comparison between UML and Ontology.

5. Medium

The teacher could use slides and beamer to present his presentation; moreover, he could also use some flip charts or slides during brain-storming etc.

6. Exercises for students

The tasks for each team are to research or conclude reasons which cause the differences between UML and Ontology deeply and write a document about the results. Length: 2-3 pages.

7. Time cost

The lecture unit is supposed to be 1.5 hour. The time cost by the exercises for each team is supposed to be 1-1.5 hour.

7.2.3 Ontology Tool Protégé

1. Motivation

Protégé is a free, open source and one authorized tool for ontology editor. For model engineers who use ontology-based modeling are indispensible to command the use of Protégé tool, furthermore, with the help of its other feasible plug-ins.

2. Content

The first part contains the introduction of the tool Protégé, and theory of its basic functions. The second part is to demonstrate an example of creating ontologies. The third part is about the introduction of some feasible useful plug-ins, such as Racer, etc.

3. Precondition of students

Knowledge of tool Protégé is not required, but knowledge of basic ontology' theories is a must, furthermore, knowledge of data modeling is an advantage for the teacher and students.

4. Method

The teacher could use presentation to explain basic functions of tool Protégé as the first part. For the second part, the teacher could use demonstrating to perform the creation of ontologies by Protégé. In addition, face-to-face explaining and questioning are also useful in case any student has questions. At last, the teacher should allow students to get used with Protégé in computer labor.

5. Medium

The teacher could use screenshots of tool Protégé in slides and beamer to present his presentation; moreover, he could also perform the demonstration of Protégé by applying a teacher-students network in computer labor.

6. Exercises for students

The tasks of each team are to install the tool Protégé and its plug-ins, get familiar with the tool environment.

7. Time cost

The lecture unit is supposed to be 1.5 hour. The labor unit is supposed to be 45 minutes. The time cost by the exercises for each team is supposed to be 1 hour.

7.2.4 Ontology-based Data Modeling

1. Motivation

The purpose of this lecture unit is to enable students to command how to create a data model based on ontology, then get an overview of several typical data modeling diagrams in ontologies.

2. Content

The contents should contain demonstration of ontology-based data modeling, and introduction of its features, advantages and disadvantages of several data modeling diagrams in ontologies, such as OWL, OntoViz etc.

3. Precondition of students

Knowledge of data modeling theory, basic ontology's theory and usage of Tool Protégé are required for students.

4. Method

The teacher could use demonstrating to perform ontology-based data modeling. For the introduction of data modeling diagrams in ontologies, the teacher could use presentation to present features of data modeling diagrams. In addition, face-to-face explaining and questioning are also useful in case any student has questions.

5. Medium

The teacher could perform the demonstration of ontology-based data modeling by applying a teacher-students network in computer labor, moreover, he could also use screenshots of Protégé in slides and beamer to present his presentation.

6. Exercises for students

The tasks of each team are to define a scenario, and implement it by means of tool Protégé. Each team should construct ontologies according this scenario. The results should conclude the screenshots of one data modeling diagram in ontologies, and enumerate the problems occurred in their works. Length: 2-3 pages.

7. Time cost

The lecture unit is supposed to be 1.5 hour. The labor unit is supposed to be 45 minutes. The time cost by the exercises for each team is supposed to be 1-1.5 hour.

7.2.5 Comparison

1. Motivation

The purpose of this part is to compare the differences of data modeling based on UML and Ontology. Since the UML approach is already familiar for some other software engineers, so it is ingenious to find a easy and optimal way to convert a data model from UML to Ontology.

2. Content

The contents should contain methods about how to convert from a class diagram or an entity relationship diagram of UML into a tuple notation of ontology and demonstrate the theory through a concrete example.

Example

The following figure 24 shows the example cut from ER Diagram of Business Manager partly for conversion from the ER Diagram to the Ontology's notation tuples.

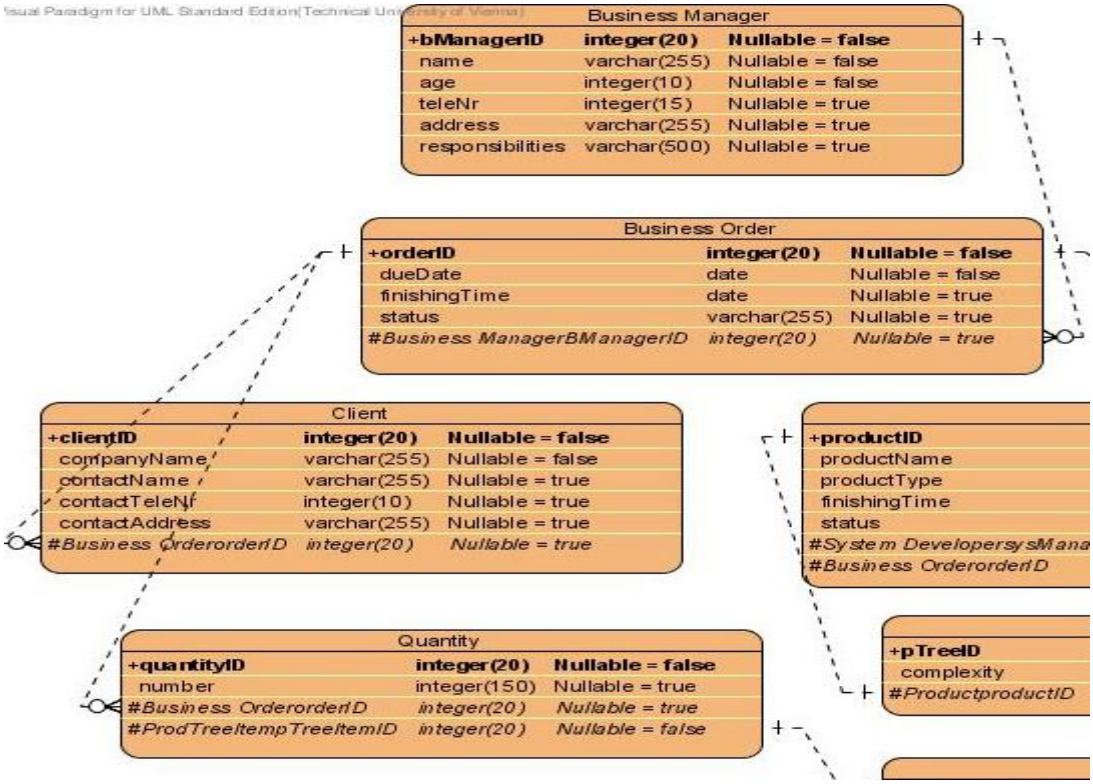

Figure 24: Example of conversion from UML ER Diagram into Ontology's notation

According to the responsibilities of business manager defined in section 4.2.1, it is easy for UML software engineers and other model engineers to

understand this ER Diagram. The meaning of this ER Diagram can be understood like one "Business Manager" could hold several "Business Orders". One "Client" could order several "Business Orders" at the same time or different time. Each "Business Order" has a corresponding "Quantity".

One Client instance can be notated in ontologies as Client [C1, Company1, Contact1, 0043123456, ContactAddress1, (BO1, BO2)]. One Business Order instance can be notated as [BO1, 01.July.2009, 2 Months, in process, BM1]. Another Business Order instance can be notated as [BO2, 01.October.2009, 2 Months, new, BM2]. One Business Manager instance can be notated as [BM1, Anna, 30, 0699 123456, Karlsplatz 13, responsible for BO1]. Another Business Manager instance can be notated as [BM2, Amy, 35, 0676 123456, Karlsplatz 13, responsible for BO2].

The Ontology's tuple notations insert the concrete configuration information of instances in each entity. The instances of each class can be presented as tuples, containing concrete data of elements in formal formulations for better machine planning and reasoning. Otherwise, within each tuple notation, the relationships between each entity and instance are represented.

3. Precondition of students

Knowledge of data modeling theory, basic ontology's theory and usage of Tool Protégé are required for students. Knowledge of UML's data modeling, either theory or praxis is a plus for students.

4. Method

The teacher could first use demonstrating to explain the use case diagram or entity diagram, for example, class's names, attributes and operations, then demonstrate the way to convert from UML diagram to Ontology's notation. Furthermore, face-to-face explaining and questioning are also useful in case any student has questions.

5. Medium

The teacher could use slides and beamer to present his presentation. Moreover, he could also use some flip charts during questioning etc.

6. Exercises for students

Each team has already defined a scenario in the last exercise. The tasks of each team are to find a partial scenario, implement it using either a class diagram or an entity relationship diagram containing at least four classes or entities, and transform this diagram to Ontology's tuple notations. Each team should conclude the transformation of the notations in 1-2 pages, additionally, summarize the differences and their experience in 1-2 pages.

7. Time cost

The lecture unit is supposed to be 1.5 hour. The time cost by the exercises for each team is supposed to be 1-1.5 hour.

7.2.6 Conclusion

1. Motivation

The purpose of this part is to recall the memories of students and remind of all the learning materials in order to strengthen their knowledge and command of ontologies.

2. Content

The contents of this part is to first repeat all the relevant information of ontology, then make a conclusion, and forecast its future development in software engineering.

3. Precondition of students

Knowledge of basic ontology's theory, usage of Tool Protégé and ontology-based data modeling are required for students.

4. Method

The teacher could use presentation to repeat all the important parts of ontology. Furthermore, the teacher should prepare some questions about the learning materials in order to test the learning results of students. Face-to-face explaining is helpful in case any student has questions.

5. Medium

The teacher could use slides and beamer to present his presentation; moreover, he could also use some flip charts or slides during questioning etc.

6. Exercises for students

The tasks of each team are to conclude the important knowledge of ontology which they have learned during this semester. Each team should present their results on the scenario of last exercise unit to the class. Presentation: 5-10 minutes

7. Time cost

The lecture unit is supposed to be 1.5 hour. The time cost by the exercises for each team is supposed to be 1-1.5 hour.

8. Discussion

The section 8 discusses the research issue regarding the model consistency check in reconfiguration. It describes the main process steps of UML-supported and Ontology-supported reconfiguration life cycle respectively. Three application scenarios are designed in order to compare based on UML- and Ontology-approach in various criteria, and conclude each strength and weakness of UML- and Ontology-based approach.

8.1 Model consistency check in reconfiguration

In consideration of high competitions in e-commerce nowadays, a major challenge of multi-agent system (MAS) in production automation is to face with numerous potential and unexpected changes. A MAS should have the capabilities to adapt to reconfigurations agilely and efficiently with the minimum of risks in causing and not discovering defects. A MAS is divided into two levels in general, its requirements and functionalities of each different level are described semantically. One is the domain level, it indicates all the development activities for a reusable set of software components in the "Component Tool Box", concluding the step "Component Development", such as agent identification, functionality definition etc. The other one is the production-line level, it indicates all the iterative configuration activities of domain-level agents for a specified product production, concluding such steps as "Component Analysis", "New Design" and "Testing and Simulation" [38] [39].

8.1.1 UML-supported reconfiguration life cycle

This section presents the four key process steps of UML-supported reconfiguration life cycle.

Step 1 Component Development

According to the stakeholder requirements on reconfigurations, UML-supported development will first analyze and generate a use case model that contains system functionalities as use cases. The Quality Assurance check point verifies whether the generated use case model is compatible with the original requirements, the Quality Assurance test is done manually in general. If the use case model has passed the Quality Assurance test, it will be stored to "Component Tool Box". If any design errors occurred and failed in the Quality Assurance test, they will be reported to the developer of "Component

Development". It is often difficult to create and maintain a complete system sequence chart diagram for modeling the whole system behavior. Moreover, no method in UML is available to define the configurations of agent instances and their properties [39].

Step 2 Component Analysis

The entire system model is composed of a variety of distribute agent models. Furthermore, each agent chooses the corresponding type of UML diagrams to present some certain aspects, such as class diagrams show basic parameters of an agent. When a large number of agents exist, it is always difficult to model the whole system behavior. For instance, it is particularly hard to handle and understand the overall system through state charts diagrams and sequence diagrams, then to measure inconsistencies and violated dependencies between models. The required agents should be identified and selected from all available agents which "Component Tool Box" could provide, in order to fulfill the global system functionalities and meantime, the corresponding XML files would be generated. The Quality Assurance check point verifies the model consistencies between current models and defined requirements manually. Here exist external model checkers, such MCC (Model Consistency Checker), but in most cases, not well populated and well-integrated in the UML methodology. When the Quality Assurance test has been passed successfully, it will transfer to next step; otherwise, the design errors will be returned to step 2 "Component Analysis" [39].

Step 3 New Design

According to UML models, some UML modeling tools support to generate some parts of implementation automatically, such as basic information of agent classes. The "Component Tool Box" which provides reusable agents could export some behaviors of the agents. It is dependent of requirements on reconfigurations to modify reusable agents slightly or implement some specific features manually. Thus, a few UML modeling tools are possible to generate codes from some parts of state charts and sequence charts regarding agents' behaviors. XML configuration file contains instances of agents, specific parameters etc additionally. The Quality Assurance check point verifies whether new configuration results match the requirements; otherwise, errors will return to Step 3 [39].

Step 4 Testing and Simulation

A MAS in production automation area is required to ensure high system quality and performance, agile reactions to failure scenarios etc. As mentioned before, UML modeling tools offer the possibilities of simulations for configuration tests. Tools can generate system test cases from the use case model and sequence charts, agent unit test cases from state charts and

sequence charts as well. The Quality Assurance check point verifies whether the new configurations and its generated XML configuration file could pass the simulation test successfully and fulfill the criteria of safety-critical systems. If any configuration errors occurred, the current system configuration will be reported to the step 2 "Component Analysis" and step 3 "New Design" [39].

The following figure 25 shows the whole process flow of UML-supported Quality Assurance in accordance with the above descriptions.

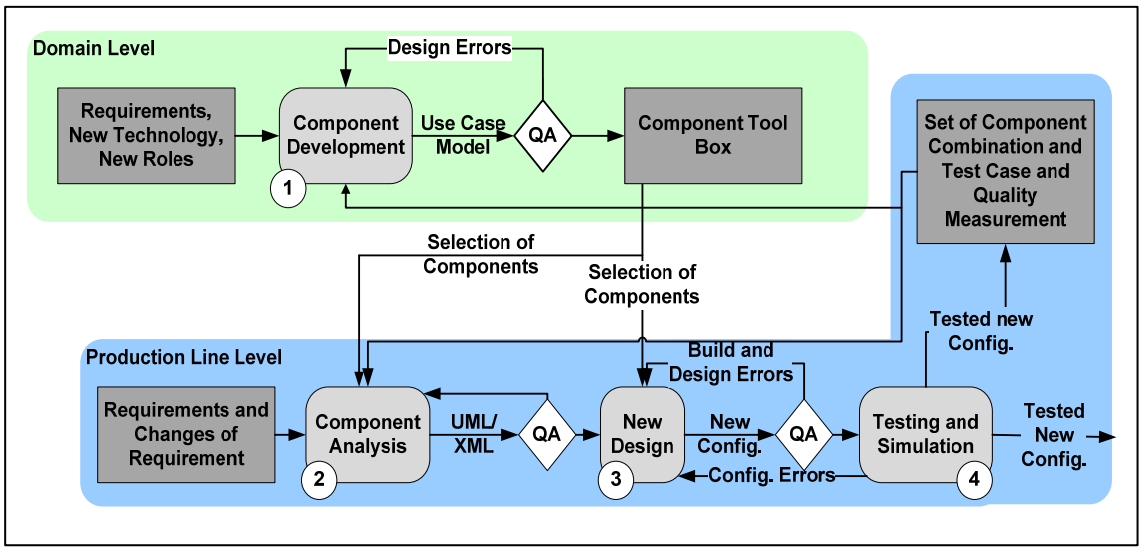

Figure 25: UML-supported Quality Assurance

8.1.2 Ontology-supported reconfiguration life cycle

This section presents the four key process steps of ontology-supported reconfiguration life cycle.

Step 1 Component Development

As long as any requirement has been changed, or new technology and new roles are planned to be added, it requires to develop those components such as agent identification, relationship identification, task definition and so on that are going to be used in the newly changed production automation system. The first Quality Assurance check point based on ontology-supported approach will generate some domain test instances automatically, such as unit tests for a particular use case or system tests for component dependencies and security-critical problems etc, and verify whether the developed components could fulfill requirements' criteria. If such components could pass the tests successfully, they will be added to "Component Tool Box"; otherwise errors will be reported to the developers and returned to the step "Component Development" [38] [39].

Step 2 Component Analysis

After step 1, new requirements and components or any changes with the impact of global system functionalities would trigger the step 2 "Component Analysis". Its input can be selection of components in the "Component Tool Box" that indicates all possible combinations of reusable components would be selected in order to satisfy the global system requirements and functionalities. Those ontologies contain both static and dynamic structures of a MAS system, for instance, static information on components of production automation system, behavioral information on models of dynamically generated and updated instances. The second Quality Assurance check point is responsible to verify whether each parameterized combination satisfies changed requirements, ensure the formal consistency and semantic validation of the designed model; any design errors will be returned as feedbacks to the step 2 "Component Analysis" [38] [39].

Step 3 New Design

During the design phase, it takes the responsibilities to make decisions for the right combination of components that satisfies non-functional requirements, such as production time, cost etc; besides, ontologies of components can be directly used as inputs and filled by concrete instances and property values. The production system is able to interpret this configuration view which has been transformed from the selected combination. The third Quality Assurance check point will emphasis on verifying on the completeness or correct syntax of new reconfigurations, any build errors will be returned as feedbacks to the step 3 "New Design", design errors will be returned to the step 2 "Component Analysis" or "New Design" [38] [39].

Step 4 Testing and Simulation

During the Testing and Simulation phase, for production automation system it is relevant to ensure the safety and high quality of systems. The simulations check the relevant properties of the actual system; the built-in monitoring functionality is well to help producing representable monitoring data in order to be evaluated by step 2 "component analysis" for selection of components. The fourth Quality Assurance check point verifies whether the tested new configuration fulfills the defined quality measurement. Any build and design errors will be returned as feedbacks to the step 3 "New Design" [38] [39].

The following figure 26 shows the whole process flow of Ontology-supported Quality Assurance in accordance with the above descriptions.

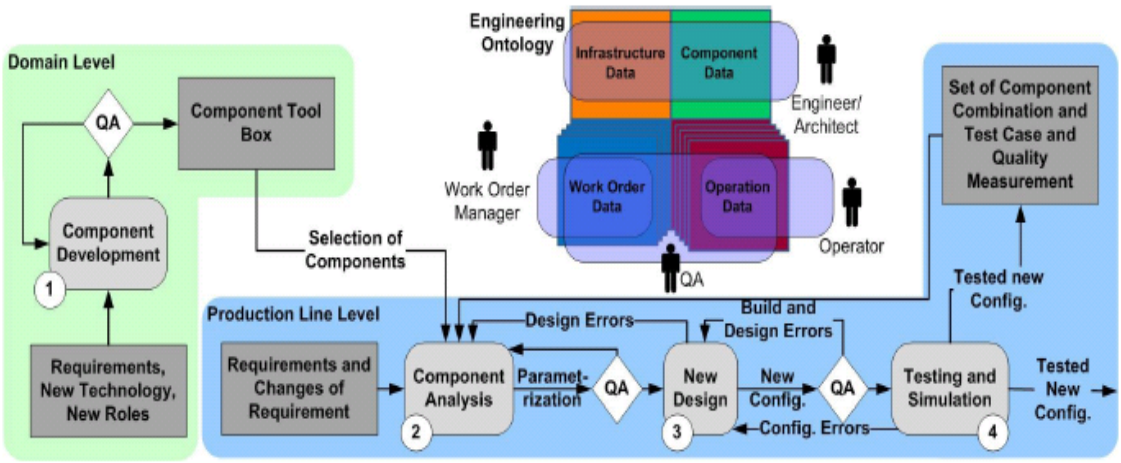

Figure 26: Ontology-supported Quality Assurance [38]

8.1.3 Concrete Example

This section concludes the relevant differences between UML and Ontology-based approach on reconfiguration processes. Through the three application scenarios, for instance, "adding a machine", "machine removal", and "change of the machine status", concludes the efforts on UML- and Ontology-approach in model complexity, modeling effort and quality risk. Furthermore, it derives the conclusion of each strength and weakness based on UML- and Ontology-approach.

8.1.3.1 Differences in reconfiguration process

The relevant differences between UML and Ontology-supported reconfiguration processes are concluded as two main parts. The first one is the extraction of reusable agents from "Component Tool Box", as we can see from the following figure 27, it shows that in Ontology all the reusable components from "Component Tool Box" will be imported in step 1 "Component Development" and be selected in step 2 "Component Analysis"; in comparison with UML, reusable components from "Component Tool Box" will be imported in step 1 "Component Development" and be extracted in step 2 "Component Analysis" and step 3 "New Design". This phenomenon indicates that UML Diagram has violated dependencies among each component. Suppose that if any component in "Component Tool Box" is required to change, such as additional functions should be added to a "Conveyor" component, it will relate step 1, 2, and 3, which would cause potential failures more easily.

In Quality measurement and assurance, the following figure shows the second main difference that errors may occur in measurements of system quality and performance. In Ontology, during the step 4 "Testing and Simulation", new configurations will be tested, if any error occurred, it will be

returned to step 2 "Component Analysis". In comparison with UML, any error of new configurations will be reported to step 1 "Component Development" and step 2 "Component Analysis".

The following figure 27 shows the differences between the process flow of UML and Ontology-supported Quality Assurance in accordance with the above descriptions.

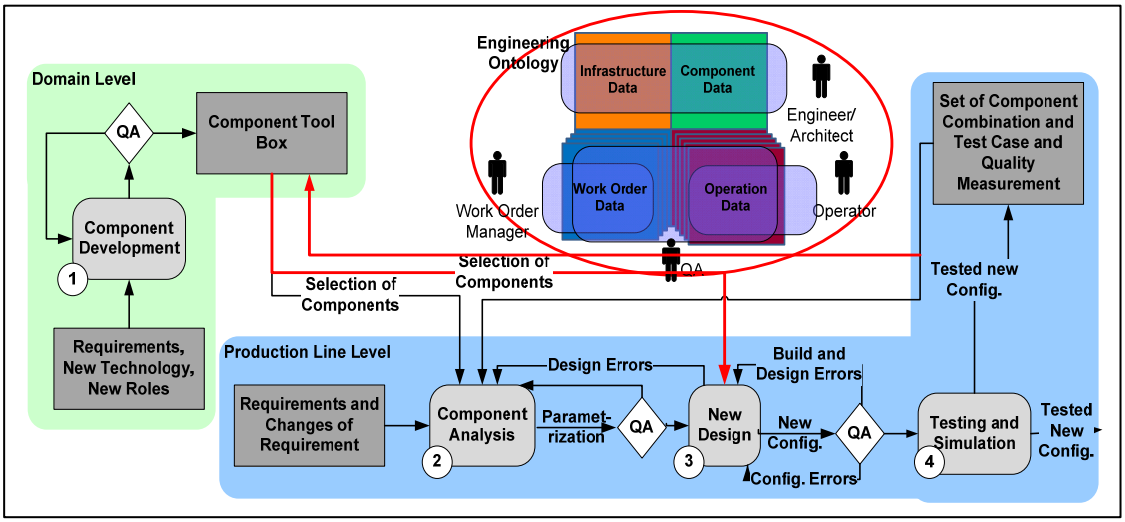

Figure 27: Differences between UML and Ontology-supported Quality Assurance

8.1.3.2 Application scenario

Scenario 1: Adding a machine

The application scenario "adding a machine" indicates the situation that a new component "Machine" is planned to be added in order to increase the throughput and effectiveness of the system.

UML

The new requirements "adding a machine" will trigger step 1 "Component Development", the use case model will be manually changed in conformity with stakeholders' requirements. If this newly changed use case model has passed the Quality Assurance test, it will be added to "Component Tool Box". In step 2 "Component Analysis", this component should be exported and used from "Component Tool Box", it will generate UML/XML codes and transfer to step 3 "New Design", as long as no inconsistencies are found. In step 3, new configurations will be reconfigured; some parts of behaviors of the "Machine" can be derived from the "Component Tool Box" into XML configuration file, concluding "Machine" classes, number of instances, and additional behavioral

parameters etc. In step 4 "Testing and Simulation", new configurations of "Machine" will be tested in a simulation environment. If any error occurs, it will be returned to step 1 "Component Development" or step 2 "Component Analysis" [39].

Ontology

The new requirement "adding a machine" should first trigger "Component Development", it will analyze whether the component "Machine" should have any new functionalities. If yes, the component "Machine" should be redefined in step 1 "Component Development", after successfully Quality Assurance check, a new "Machine" component will be added to "Component Tool Box"; otherwise since the component "Machine" has been already applied to the multi-agent production automation system, the "Machine" could be selected directly from "Component Tool Box", so it means the component "Machine" would be reused in step 2 "Component Analysis". The component "Machine" would be parameterized and carried out by the automatic Quality Assurance test. In step "New Design", Quality Assurance test will check whether the component "Machine" is compatible with the whole complex system. If any error occurs, it will be returned to step 3 "New Design". Finally, it will transfer to the step 4 "Testing and Simulation", its logical rules and potential hardware failures etc. will be tested and monitored, errors will returned to step 2 "Component Analysis" [39].

Scenario 2: machine removal

The application scenario "machine removal" indicates the situation that one component "Machine" is planned to be deleted.

UML

The changes of requirement "machine removal" will trigger the step 2 "Component Analysis", the use case model will be changed according to the current requirement changes, and then it will generate a UML or XML configuration file. In step 3 "New Design", it requires the new design to be compatible with global functionalities of the system; the Quality Assurance test will check if the new configurations are correct manually executed by designers. If any error occurred in step 4 "Testing and Simulation", it will back to "Component Analysis" [39].

Ontology

The change of requirements "machine removal" will trigger also the step 2 "Component Analysis", in this step, it will analyze whether the removal of component "Machine" will affect other functionalities. Above all, the instance of "Machine" class and the responsible software agents, moreover, the surplus

connections between the component "Machine" with other components should be also removed, unless it will cause design errors. In step 3 "New Design", it will check if the new configurations are correct. During the step 4 "Testing and Simulation", any errors caused by new configurations, will be reported to step 2 "Component Analysis" [39].

Scenario 3: change of the machine status

The application scenario "change of the machine status" indicates status of the machine has been changed according to the different situations of the multi-agents production automation system.

UML

The change of requirements "change of the machine status" works similarly like scenario2 will trigger the step 2 "Component Analysis", a new UML or XML configuration file with the newly changed machine status will be generated. In step 3 "New Design", it will check again by designer or software developer whether the overview of the whole system is correct. In step 4 "Testing and Simulation", the related errors will be reported to either step 1 or step 2 [39].

Ontology

The change of requirements "change of the machine status" will trigger first the step 2 "Component Analysis", it has to check whether the change of machine status will affect other functionalities. For example, the change of the machine status will change some values of Component "Capacity", properties like occupied capacity, available capacity etc.

Moreover, some potential failures will occur because of the reconfiguration of the machine status, but through the logical Quality Assurance test, the built reconfiguration errors will be discovered soon [39].

8.1.3.3 Comparison

Table 6: Comparison of the three scenarios based on UML- and Ontology-approach

	UML-based Approach	Ontology-based Approach
Scenario 1 - adding a machine		
Model complexity	medium	low
Modeling effort	Model changes: medium Dependency analysis: high Quality assurance: high	Model changes: low Dependency analysis: medium Quality assurance: low
Quality risk	medium	low

Scenario 2 - machine removal		
Model complexity	medium	low
Modeling effort	Model changes: medium Dependency analysis: high Quality assurance: high	Model changes: low Dependency analysis: medium Quality assurance: low
Quality risk	medium	low
Scenario 3 - change of the machine status		
Model complexity	high	low
Modeling effort	Model changes: high Dependency analysis: high Quality assurance: high	Model changes: low Dependency analysis: medium Quality assurance: low
Quality risk	medium	low

Conclusion

As mentioned in section 8.1.3.2, the use case model will be manually changed in conformity with stakeholders' requirements by means of the UML-based approach. For scenario 1 "adding a machine", the effort of Model changes is medium, because the component "Machine" could be reused from the "Component Tool Box" and designers only need to insert the "Machine" instance in the XML configuration file when no further functionalities of "Machine" required by customers. The Dependency analysis is high, because the entire model has to be checked manually whether this added activity has an impact on other functions and components. The Quality assurance is also high, because designers should ensure whether this added component integrates well with the current design pattern of the system, and the whole integrated system could pass the failure tests. The Quality risk is medium, because it relates strong with the ability and accuracy of the software engineers, and furthermore, it relates also with the risk possibilities of Dependency analysis and Quality assurance, which should be executed very seriously in order to reduce mistakes caused manually [39]. The other two scenarios work actually like the scenario 1 and have the same scalabilities.

By means of the Ontology-based approach, a new instance of "Machine" class should be added and inserted its properties according to the requirements in the Ontology, therefore, the effort of Model changes is low. The Dependency analysis is medium, because all the errors introduced by adding a new machine, such as no available agent is responsible for this machine etc could be detected by Ontology logic reasoner automatically. The Quality risk is also low, because the automatic tool check and low cost on Quality assurance help to avoid the large manual modeling efforts and errors and then reduce the Quality risk to low [39].

The "machine removal" activity will require a few instances to be removed,

such as the responsible agent for this machine instance and the instance of "Machine" class and other connections link to other agents and machines should be removed. The efforts of Model change are also relatively low. The Dependency analysis is low, because it works similar like scenario 1 that all the errors introduced by the machine removal, such as no available agent is responsible for this machine etc, could be detected by Ontology logic reasoner automatically. The Quality risk is low, it works also similar like scenario 1, automatic tool checks ensure the high quality and reduce the related risks [39].

Table 7: Conclusion of each strength and weakness based on UML- and Ontology-approach

	UML-based Approach	Ontology-based Approach
Strength	❖ UML diagrams provide clear and well understandable visualizations for software designers and engineers ❖ UML diagrams provide a detailed and well understandable overview of all the agents' classes, their properties and relationships.	❖ Creation of data models and instances in an integrated ontology model ❖ Logical text-based syntax supports machines for automatic dependency analysis and consistency checking ❖ Synchronization risk is low because ontology model can be used both at design and run time
Weakness	❖ External model checker to carry out automatic model reviews ❖ High error potential and cost in manual model reviews ❖ No diagram available to present the instance and configuration information	❖ Higher complexity and large volume of ontology model ❖ A general overview and visualization is hard for human to understand ❖ Preconditions on understanding of the domain required

Conclusion

The strengths of UML-based approach are that UML diagrams provide clear and well understandable visualizations for software designers and engineers, such as each kind of UML diagram has its own strong focus on various business processes and workflows etc. Software engineers can get an overview of the agent tool box containing all reusable components for reconfiguration processes on the other hand. Moreover, UML provide a well overview of component classes, their properties and relationships etc [39].

The strengths of Ontology-based approach are that an ontology model is

able to integrate all the detailed configuration information, such as instances of the classes and dependencies, their schema and data. The automatic consistency check and quality assurance of the ontology model are able to be realized by logical tool. The ontology model is able to be used not only at the design time but also at the run time, such as, run-time reconfiguration checks [39].

The weaknesses of UML-based approach are that the modeling effort is higher for the UML-based approach than for the ontology-based approach. Because UML requires manual dependency analysis and quality assurance which cost more efforts and leads to more errors, otherwise, Ontology is supported by automated reasoning. Each agent has its own components in separate diagram with a fractured view, the sequence diagram that models all the collaborations of agents could lead to large complexity. Finally, there is no type of UML diagrams is able to present the instance of classes and their configuration information [39].

The weaknesses of Ontology-based approach are that the ontology model integrates always all detailed reconfiguration information; therefore the model is too large with higher complexity. The visualizations that provided by the standard tool Protégé are not well to visualize an overview of a certain domain, for example, in Ontoviz some expressions are interpreted in logics. Finally, it is difficult for designers and software engineers to figure out the contribution of each ontology element without the preconditions on understanding of the domain [39].

9. Conclusion

This section summarizes the whole thesis, mention the most important points of this project. Besides, it concludes the relevant explored results of the research issues. Finally, it suggests some interesting and meaningful future works.

9.1 Summary

In the beginning of the thesis, I have depicted the motivations of this project, for example, the importance of using data modeling during the whole software lifecycle, introduction to data model theory and instances, and mostly the extension to the abstract. At last, the thesis structure provides a superficial overview of the whole thesis structure.

In "Related Work", I have described the principles of creating an accurate data modeling in an efficient way, in additional the brief introduction to the two data modeling approaches, including the basic concepts and their main building elements of object-oriented data modeling and knowledge modeling respectively. Later, I have explained the UML's development history shortly and introduced the OMG's standard and official definition for UML, and primary principles of UML's specifications, such as its three basic building blocks, their classifications and components of each classification, 13 kinds of notation diagrams with focus on the details of the class diagram, various views for analyzing system architecture as well, so that those kind of knowledge can help build a strong foundation to whose have already a smattering in UML and otherwise help some UML users to recall the knowledge. Moreover, there are a lot of assistant visualization tools available for UML-based approach, Visual Paradigm is one of the most powerful comprehensive and easy-to-use tools, it presents a brief of its overviews and some features and specific functionalities for creating an ER Diagram.

Since ontology is a relative newly concept appeared in software engineering and knowledge engineering, it is recommended to have a look at the ontology's development history, and definitions including the main components which consist of an ontology. I chose three ontology languages, two of them RDF/RDF Schema and DAML+OIL were preceding releases, now both have been replaced by the current popular ontology language OWL. Furthermore, OWL has been developed to its three sublanguages OWL Full, OWL DL and OWL Lite regarding different usage requirements and situations, where OWL DL is relatively common used in most cases. Protégé is a standard tool for definition and manipulation of ontologies, therefore, it is necessary for software engineers to get an overview, screenshots and main

features of Protégé. At the end of this section, it is still worthy to mention the theory of the multi-agent system simulation.

According to some research and papers, four research issues have been defined to be discussed in a deep level. The issue "Evaluation of UML and Ontology" presents such as the creation of two data models for this multi-agent production automation system by means of UML and OWL, and then gain the evaluation of UML and Ontology on their common and uncommon features, finally derive the summary of their similarities and differences. The issue "Model consistency checking" is to explore the similarities and differences of model consistency checking life cycle supported from UML and Ontology, to find out whether there is any external tool available for UML that supports its automatic consistency checking. The third issue is to find out the process of the mapping from UML to OWL and discover its benefits and limitations. The last issue is to discover the possibilities to improve the UML on strengthen its disadvantages, reduce its redundancy, lessen the gap between UML and Ontology, and find an optimal combination of UML and Ontology.

Since the audience have already got a superficial impact of the multi-agent system in section 2, section 4 "Use Case Description" introduced the specific tool Manufacturing Agent Simulation Tool for the test management of the multi-agent system with special emphasis on its technical assembled components, moreover, it explained shortly the main differences between the MAST and SAW which is an improvement of MAST employed in this project. It is very exigent to define the use case descriptions before any data modeling. Therefore, I defined the responsibilities of each agent's job, derive the generalized use cases for each six involved roles. Collaborations among all the involved roles are also relevant that help the users observe the whole multi-agent system in a centralized view.

The definitions of use cases serve a solid foundation to the section "Data Modeling for SAW with UML and Ontology". I employed the tool Visual Paradigm to create the data modeling for UML, and chose to implement the use cases defined above in Entity Relationship Diagrams. The essential parts of ER Diagram, such as each entity's name, their properties, each property's type, the initial value of each property, relationship between each entity and relationship multiplicity should be denoted. The descriptions below explain the notations of ER Diagram and help users to understand well.

Meanwhile, I employed the tool Protégé to create the data modeling for ontologies and chose to implement the use cases in OWL DL displayed with Ontoviz Diagrams. The essential parts of OWL ontologies, such as individuals, two kinds of properties: datatype properties and object properties, and restrictions on slots should be denoted. The Ontoviz diagrams are too large to be displayed, therefore, I cut each diagram into two parts, the upper part and the lower part. The descriptions below help to explain the notations of Ontoviz diagrams better.

According to the whole processes of the data modeling for UML and

Ontology-based approaches which have been demonstrated in the previous sections, it is time for the evaluation of their main features in various significant aspects. The first aspect "Visualization & Expression" lists the evaluation results of both common and uncommon features in two tables respectively for a clear demonstration, and concluded each similarity and difference in the description below. The second aspect "Consistency" splits into 2 parts "Diagram consistency" and "Model consistency". The evaluation table summarizes the supported or non-supported consistency features of UML and Ontology. In the third aspect "Needed Effort" compares each approximate time cost required for creation and consistency checking on UML and Ontology. The last aspect "Additional Features" lists the available additional features, mainly "Reasoning" and "Codes" aspects provided by UML or Ontology, and their external plug-ins and support tools etc.

Since UML occurred for several decades, there have yet existed a lot of valuable researches and skills in its didactic aspect, so I focused on giving some suggestions and improvements on ontology-based data modeling in didactics. First, I defined the seven characteristic elements of didactics for this scenario. During the establishment for the structure of didactic materials, for example, the concrete tasks of the didactic elements, lecture units like overview of ontology, theory enhancement, ontology tool Protégé, ontology-based data modeling, comparison and conclusion. One of the most important points is the conversion way from UML Class Diagrams or ER Diagrams into Ontology's tuple notations; it facilitates the UML software engineers to find a easy and well understandable way to grasp of using ontology as soon as possible.

The section "Discussion" emphasized on answering the research issue "model consistency check in reconfiguration". It describes and compares the key process steps of UML and Ontology-supported reconfiguration lifecycle. The comparison results have been demonstrated by a concrete example of three scenarios, like adding a machine, machine removal and change of the machine status. According to the exploration of the three scenarios, it lists two tables that conclude the main modeling efforts of the three scenarios, each strength and weakness based on UML- and Ontology-approach as well.

9.2 Results

The main results of this work can be found in section 6, 7 and 8. The section 6 comprises the evaluation results of the four aspects, "Visualization & Expression", "Consistency check", "Needed Effort" and "Additional features". The first aspect list two tables, one table contains the corresponding common features of UML and OWL. The descriptions explain the detailed common features visualized and expressed in UML and OWL. The other table contains all the features that can be either expressed in UML or only in OWL. The descriptions explain the uncommon features. The conclusion of this aspect

gives a brief overview of all specified features. UML is designed to be a graphical notion modeling language originally, facilitates the human beings to understand the system model. UML's state chart diagrams are used to describe dynamic behaviors of a system, such as the possible states of an object as events occur. Therefore, OWL is a knowledge representation of ontology, provides a static ontology vocabulary, thus behavioral features are surplus in OWL.

The "Consistency check" lists the evaluation table of the results. For UML it aims to be a robust diagram system, so it provides class consistency, attribute consistency and relationship consistency checking in diagram checking. For Ontology it is more important to ensure models consistent, so it provides class consistency, property consistency and instance consistency checking in model checking.

The "Needed effort" list each time cost for UML and OWL creations, and for consistency checking of UML and OWL. Time cost for creation of classes both in UML and OWL are almost equal while the time cost for creation of attributes and creation of relationships required in UML is shorter than in OWL.

The "Additional Features" list the additional features such as "Reasoning" and "Show/generate codes", the "Reasoning" functions for example UML are not supposed to be designed to have such functions originally, but all reasoning features can be realized by applying the third party plug-in MCC.

The section 7 "Didactics in ontology-based modeling" gives a solution and suggestions for improvements to the didactics in ontology-based modeling. The normal didactics lecture units can be established like "Overview of ontology" introduces the students to some ontology's development history, definitions and its practical applications. "Theory enhancement" helps the students get a deep insight into more details of ontology. "Tool Protégé" enables the students to grasp using the support tool Protégé for ontology-based data modeling. "Ontology-based data modeling" teaches the students how to create a data model based on ontology, and get to know an overview of several typical data modeling diagrams, such as OntoViz, OWL etc. "Comparison" is the most important point of this didactics part, explains differences of data modeling based on UML and Ontology and introduce a method and its example about transformation from UML to Ontology. In urgent situations, for example, UML software engineers have no plentiful time to learn ontology step by step, they can use the transformation method to understand and denote OWL. Finally, "Conclusion" recalls the memories of students and reminds of all the learning materials in order to strength their knowledge and command ontologies.

The section 8 "Discussion" describes the model consistency checking in UML-supported and Ontology-supported reconfiguration life cycles respectively. The whole life cycle can be divided into 4 steps, "Component Development", "Component Analysis", "New Design" and "Testing and Simulation". The relevant differences between UML and Ontology-supported

reconfiguration process are the extraction of reusable agents from "Component Tool Box" in ontology will be imported in step 1 "Component Development" and be selected in step 2 "Component Analysis"; in comparison with UML, reusable components from "Component Tool Box" will be imported in step 1 "Component Development" and be extracted in step 2 "Component Analysis" and step 3 "New Design". Another significant difference is that in case errors may occur in measurements of system quality and performance, in Ontology during the step 4 "Testing and Simulation", new configurations will be tested, if any error occurred, it will be returned to step 2 "Component Analysis". In comparison with UML, any error of new configurations will be reported to step 1 "Component Development" and step 2 "Component Analysis". At the end, it lists the evaluation of comparison of the three scenarios like adding a machine, machine removal and change of the machine status, based on UML- and Ontology-approach and conclusion of each strength and weakness based on UML- and Ontology-approach.

9.3 Future work

Regarding the limited space and range of my thesis, there are still some future work required and would be very interesting for this area.

In Section 6.4.3, I stated the theory possibilities for the automatic model consistency checking of UML models by means of integrating the MCC and Racer together. Such empirical experiments and proofs regarding the runtime complexity of MCC are firmly recommended.

In Section 7 "Didactics in ontology-based modeling", I suggested the didactic units of lectures for ontologies and an easy understandable way for UML software engineers to grasp the ontology, but empirical studies and experiments are also required in order to find a more acceptable and better way in the future. In section 8 "Discussion", I discussed the theory of model consistency checking in reconfiguration life cycle both using UML and Ontology aspects. Corresponding empirical studies should be proved to raise the correctness and sufficient performance of model consistency checking and reconfigurations [38].

A challenging question is for some organizations which have been already employing a traditional software development approach such as UML, when to determine the crucial time point and how to introduce a new development approach, like ontology-based approach. Researches and empirical studies regarding benefits, limitations and how to achieve the best results are required according to different situations [38]. The other issue in the current research fields, for example, mapping from UML to Ontology concerning the question how to bridge and transfer the UML to Ontology. The section 7 which suggested the transformation from UML diagrams to Ontology notations could be used as reference; more specific transformation rules from UML models to Ontology models should be researched and defined in the future.

References

1. A. Th. Schreiber and B. J. Wielinga (2008): Knowledge Model Construction, http://ksi.cpsc.ucalgary.ca/KAW/KAW98/schreiber/
2. American National Standards Institute (1975): ANSI/X3/SPARC Study Group on Data Base Management Systems
3. Armin Zimmermann (2007): Stochastic Discrete Event Systems: Modeling, Evaluation, Applications
4. Coral Calero (2006): Ontologies for software engineering and software technology. Springer
5. Crag Systems (1997-2004): A UML Tutorial Introduction, http://www.cragsystems.co.uk/ITMUML/the02/08the02.htm
6. Dan Brickley, R.V. Guha (2004): RDF Schema, W3C, http://www.w3.org/TR/rdf-schema/
7. Deborah L. McGuinness, Frank van Harmelen (2004): OWL Overview, W3C, http://www.w3.org/TR/owl-features/
8. Donald Bell (2004): UML basics: The class diagram, http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/
9. Edumoodle (2009), http://www.edumoodle.at/moodle/
10. Eric Prud'hommeaux, Andy Seaborne (2008): SPARQL Query Language for RDF, W3C, http://www.w3.org/TR/rdf-sparql-query/
11. Grady Booch, James Rumbaugh, Ivar Jacobson (1999): The Unified Modeling Language User Guide. Addison-Wesley
12. Hans-Jörg Happel, Stefan Seedorf (2006): Applications of Ontologies in Software Engineering, Workshop on Semantic Web Enabled Software Engineering,
http://km.aifb.uni-karlsruhe.de/ws/swese2006/final/happel_full.pdf
13. IBM, Sandpiper Software, Inc. (2004): Ontology Definition Metamodel, www.omg.org/docs/ad/05-08-01.pdf
14. Jocelyn Simmonds, M.Cecilia Bastarrica (2005): A Tool for Automatic UML Model Consistency Checking, Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering
15. Jon Holt, Institution of Electrical Engineers (2004): UML for Systems Engineering: watching the wheels. IET
16. Lewis Hart, Patrick Emery, Bob Colomb, Kerry Raymond, Sarah Taraporewalla, Dan Chang, Yiming Ye, Elisa Kendall, Mark Dutra (2004): OWL Full and UML 2.0 Compared, OMG TFC Report, http://www.itee.uq.edu.au/~colomb/Papers/UML-OWLont04.03.01.pdf
17. Liviu Panait, Sean Luke (2005): Cooperative Multi-Agent Learning: The State of the Art. Autonomous Agents and Multi-Agent Systems, Springer Netherlands

18. M. Merdan, T Moser, D. Wahyudin, S. Biffl (2008): Performance Evaluation of Workflow Scheduling Strategies Considering Transportation Times and Coveyor Failures. Proceedings of 6th international Workshop on Software Quality"
19. M. Merdan, T Moser, D. Wahyudin, S. Biffl, P. Vrba (2008): Simulation of workflow scheduling strategies using the MAST Test management system. Proceedings 10th International Conference on Control, Automation, Robotics and Vision
20. Margaret-Anne Storey, Robert Lintern, Neil Ernst, David Perrin (2004): Visualization and Protégé, 7th International Protégé Conference, http://protege.stanford.edu/conference/2004/abstracts/Storey.pdf
21. Matthew Horridge, Holger Knublauch, Alan Rector, Robert Stevens, Chris Wroe (2004): A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and COODE Tools Edition 1.0, The University Of Manchester Stanford University, http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf
22. Michael R. McCaleb (1999): A Conceptual Data Model of Datum Systems. Journal of Research of the National Institute of Standards and Technology
23. Michael Wooldridge (2002): An Introduction to MultiAgent Systems. John Wiley & Sons
24. Object Group Management (2007): UML Version 2.1.2 Specifications, http://www.omg.org/spec/UML/2.1.2/
25. Object Oriented Analysis and Design Team of Kennesaw State University (2000): History of UML, http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/history_of_uml.htm
26. Ora Lassila, Ralph R. Swick (1999): Resource Description Framework
27. Oscar Corcho, Asuncion Gomez-Perez (2000): A Roadmap to Ontology Specification Languages. Springer Berlin
28. Osmar Zaiane (1998): The Object-Oriented Data Model, http://www.cs.sfu.ca/CC/354/zaiane/material/notes/Chapter8/node3.html
29. Pavel Vrba (2003): MAST Manufacturing agent simulation tool. Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA '03. IEEE Conference
30. Protege (2009): http://protege.stanford.edu/
31. Protege (2009): what is Protege, http://protege.stanford.edu/overview/
32. Protege (2009): what is protégé-owl?, http://protege.stanford.edu/overview/protege-owl.html
33. Protege (2009): Screenshots of protégé-owl, http://protege.stanford.edu/plugins/owl/images/OWLClasses-FamilyDestination.png
34. [49] Racer Systems (2004): Overview of RacerPro, http://www.racer-systems.com/products/racerpro/index.phtml
35. Rockwell Automation Research Center (2009):

http://www.rockwellautomation.com/
36. Smith, Michael K.; Chris Welty, Deborah L. McGuinness (2004): OWL Web Ontology Language Guide, W3C, http://www.w3.org/TR/owl-guide/
37. Sparx Systems(2008): UML 2.1 Tutorial: http://www.sparxsystems.com.au/resources/uml2_tutorial/
38. Steffan Biffl, Thomas Moser, Richard Mordinyi, Dindin Wahyudin (2008): Ontology-Supported Quality Assurance for Component-Based Systems Configuration, Proceedings of 6th international Workshop on Software Quality
39. T Moser, K. Kunz, K. Matousek, D. Wahyudin (2008): Investigating UML- and Ontology-Based Approaches for Process Improvement in Developing Agile Multi-agent Systems, 34th EUROMICRO Conference on Software Engineering and Advanced Applications, IEEE Computer Society
40. The DARPA Agent Markup Language Homepage (2006), http://www.daml.org/
41. Tim McLellan (1995): An Introduction to Data Modeling http://www.islandnet.com/~tmc/html/articles/datamodl.htm
42. Uwe Szabo (2008): EER-model
43. Visual Paradigm (2009): http://www.visual-paradigm.com/
44. Visual Paradigm (2009): Why Visual Paradigm for UML? http://www.visual-paradigm.com/product/vpuml/
45. Visual Paradigm (2009): UML CASE Tool Screenshots, http://www.visual-paradigm.com/product/vpuml/vpumlscreenshots.jsp
46. Visual Paradigm (2009): Data Model, http://content.europe.visual-paradigm.com/media/documents/vpuml60ug2/html/Chapter_14_Data_Model/Chapter_14_Data_Model.html
47. W3C (2008): XML Schema, http://www.w3.org/XML/Schema
48. Wikipeida (2009): Ontology, http://en.wikipedia.org/wiki/Ontology

Appendix

1. RDF/XML Source Codes of business manager

The followings are the RDF/XML source codes of the agent business manager. They could be generated automatically by the tool support Protégé according to the ontology model of business manager. Within the codes, they contain some relevant basic structures of business manager and represent in a formal logical formalization, such as each datatype property, its domain and range etc. For further understanding on source codes, please see reference [7].

```
<rdf:RDF
xmlns="http://qse.tuwien.ac.at/datamodel/ontology/businesslayer#"
  xml:base="http://qse.tuwien.ac.at/datamodel/ontology/businesslayer"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:swrl="http://www.w3.org/2003/11/swrl#"
  xmlns:Protégé="http://Protégé.stanford.edu/plugins/owl/Protégé#"
  xmlns:xsp="http://www.owl-ontologies.com/2005/08/07/xsp.owl#"
  xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<owl:Ontology rdf:about=""/>
<owl:AllDifferent>
<owl:distinctMembers rdf:parseType="Collection"/>
</owl:AllDifferent>
<owl:DatatypeProperty rdf:ID="address">
<rdfs:domain rdf:resource="#businessManager"/>
<rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="age">
<rdfs:domain rdf:resource="#businessManager"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="amount">
<rdfs:domain rdf:resource="#businessManager"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="bManagerID">
<rdfs:domain rdf:resource="#businessManager"/>
<rdfs:range rdf:resource="&xsd;int"/>
```

```
</owl:DatatypeProperty>
<owl:Class rdf:ID="businessManager">
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
<owl:Restriction>
<owl:onProperty rdf:resource="#address"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#age"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#bManagerID"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#holds"/>
<owl:minCardinality rdf:datatype="&xsd;int">1</owl:minCardinality>
<owl:valuesFrom rdf:resource="#businessOrder"/>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#name"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#responsibilities"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#teleNr"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
</owl:intersectionOf>
</owl:Class>
</owl:equivalentClass>
</owl:Class>
<owl:Class rdf:ID="businessOrder">
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
<owl:Restriction>
<owl:onProperty rdf:resource="#dueDate"/>
```

<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#finishingTime"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#includes"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
<owl:valuesFrom rdf:resource="#quantity"/>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#includes"/>
<owl:minCardinality rdf:datatype="&xsd;int">1</owl:minCardinality>
<owl:valuesFrom rdf:resource="#product"/>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#orderID"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#status"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
</owl:intersectionOf>
</owl:Class>
</owl:equivalentClass>
</owl:Class>
<owl:Class rdf:ID="client">
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
<owl:Restriction>
<owl:onProperty rdf:resource="#clientID"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#companyName"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#contactAddress"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>

```
<owl:Restriction>
<owl:onProperty rdf:resource="#contactName"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#contactTeleNr"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#orders"/>
<owl:minCardinality rdf:datatype="&xsd;int">1</owl:minCardinality>
<owl:valuesFrom rdf:resource="#businessOrder"/>
</owl:Restriction>
</owl:intersectionOf>
</owl:Class>
</owl:equivalentClass>
</owl:Class>
<owl:DatatypeProperty rdf:ID="clientID">
<rdfs:domain rdf:resource="#client"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="companyName">
<rdfs:domain rdf:resource="#client"/>
<rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="complexity">
<rdfs:domain rdf:resource="#productTree"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:ObjectProperty rdf:ID="consistsOf">
<rdfs:domain>
<owl:Class>
<owl:unionOf rdf:parseType="Collection">
<owl:Class rdf:about="#product"/>
<owl:Class rdf:about="#productTree"/>
</owl:unionOf>
</owl:Class>
</rdfs:domain>
<rdfs:range>
<owl:Class>
<owl:unionOf rdf:parseType="Collection">
<owl:Class rdf:about="#productTree"/>
<owl:Class rdf:about="#productTreeItem"/>
</owl:unionOf>
```

```
</owl:Class>
</rdfs:range>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="contactAddress">
<rdfs:domain rdf:resource="#client"/>
<rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="contactName">
<rdfs:domain rdf:resource="#client"/>
<rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="contactTeleNr">
<rdfs:domain rdf:resource="#client"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="dependOf">
<rdfs:domain rdf:resource="#productTreeItem"/>
<rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="dueDate">
<rdfs:domain rdf:resource="#businessOrder"/>
<rdfs:range rdf:resource="&xsd;date"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="finishingTime">
<rdfs:domain rdf:resource="#businessOrder"/>
<rdfs:range rdf:resource="&xsd;date"/>
</owl:DatatypeProperty>
<owl:ObjectProperty rdf:ID="holds">
<rdfs:domain rdf:resource="#businessManager"/>
<rdfs:range rdf:resource="#businessOrder"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="includes">
<rdfs:domain>
<owl:Class>
<owl:unionOf rdf:parseType="Collection">
<owl:Class rdf:about="#businessOrder"/>
<owl:Class rdf:about="#productTreeItem"/>
</owl:unionOf>
</owl:Class>
</rdfs:domain>
<rdfs:range>
<owl:Class>
<owl:unionOf rdf:parseType="Collection">
<owl:Class rdf:about="#product"/>
```

<owl:Class rdf:about="#quantity"/>
</owl:unionOf>
</owl:Class>
</rdfs:range>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="name">
<rdfs:domain rdf:resource="#businessManager"/>
<rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="number">
<rdfs:domain rdf:resource="#quantity"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="orderID">
<rdfs:domain rdf:resource="#businessOrder"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:ObjectProperty rdf:ID="orders">
<rdfs:domain rdf:resource="#client"/>
<rdfs:range rdf:resource="#businessOrder"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="positionX">
<rdfs:domain rdf:resource="#productTreeItem"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="positionY">
<rdfs:domain rdf:resource="#productTreeItem"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:Class rdf:ID="product">
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
<owl:Restriction>
<owl:onProperty rdf:resource="#consistsOf"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
<owl:valuesFrom rdf:resource="#productTree"/>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#finishingTime"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#productID"/>

<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#productName"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#productType"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#status"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
</owl:intersectionOf>
</owl:Class>
</owl:equivalentClass>
</owl:Class>
<owl:DatatypeProperty rdf:ID="productID">
<rdfs:domain rdf:resource="#product"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="productName">
<rdfs:domain rdf:resource="#product"/>
<rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:Class rdf:ID="productTree">
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
<owl:Restriction>
<owl:onProperty rdf:resource="#complexity"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#consistsOf"/>
<owl:minCardinality rdf:datatype="&xsd;int">1</owl:minCardinality>
<owl:valuesFrom rdf:resource="#productTreeItem"/>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#pTreeID"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
</owl:intersectionOf>

```
</owl:Class>
</owl:equivalentClass>
</owl:Class>
<owl:Class rdf:ID="productTreeItem">
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
<owl:Restriction>
<owl:onProperty rdf:resource="#amount"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#dependOf"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#includes"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
<owl:valuesFrom rdf:resource="#quantity"/>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#positionX"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#positionY"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#pTreeItemID"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#pTreeItemName"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#siblings"/>
<owl:minCardinality rdf:datatype="&xsd;int">1</owl:minCardinality>
</owl:Restriction>
</owl:intersectionOf>
</owl:Class>
</owl:equivalentClass>
</owl:Class>
```

```
<owl:DatatypeProperty rdf:ID="productType">
<rdfs:domain rdf:resource="#product"/>
<rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="pTreeID">
<rdfs:domain rdf:resource="#productTree"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="pTreeItemID">
<rdfs:domain rdf:resource="#productTreeItem"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="pTreeItemName">
<rdfs:domain rdf:resource="#productTreeItem"/>
<rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:Class rdf:ID="quantity">
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
<owl:Restriction>
<owl:onProperty rdf:resource="#number"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
<owl:Restriction>
<owl:onProperty rdf:resource="#quantityID"/>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
</owl:intersectionOf>
</owl:Class>
</owl:equivalentClass>
</owl:Class>
<owl:DatatypeProperty rdf:ID="quantityID">
<rdfs:domain rdf:resource="#quantity"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="responsibilities">
<rdfs:domain rdf:resource="#businessManager"/>
<rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="siblings">
<rdfs:domain>
<owl:Class>
<owl:unionOf rdf:parseType="Collection">
```

```
<owl:Class rdf:about="#businessOrder"/>
<owl:Class rdf:about="#productTreeItem"/>
</owl:unionOf>
</owl:Class>
</rdfs:domain>
<rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="status">
<rdfs:domain>
<owl:Class>
<owl:unionOf rdf:parseType="Collection">
<owl:Class rdf:about="#businessOrder"/>
<owl:Class rdf:about="#product"/>
</owl:unionOf>
</owl:Class>
</rdfs:domain>
<rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="teleNr">
<rdfs:domain rdf:resource="#businessManager"/>
<rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
</rdf:RDF>
```