UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN MATHÉMATIQUES

ET INFORMATIQUE APPLIQUÉES

PAR

YANFEN SHEN

A FORMAL ONTOLOGY FOR DATA MINING: PRINCIPLES, DESIGN, AND EVOLUTION

(UNE ONTOLOGIE FORMELLE POUR LE FORAGE DE DONNÉES : PRINCIPES, CONCEPTION ET ÉVOLUTION)

AVRIL 2007

A FORMAL ONTOLOGY FOR DATA MINING: PRINCIPLES, DESIGN, AND EVOLUTION

Yanfen Shen

ABSTRACT

Data mining (DM) and decision support system (DS) are two relatively independent domains broadly applied to scientific research and business practice. Successful integration of technologies associated with both domains could an intelligent data mining assistant system, which is able to provide intelligent assistance beyond the numbers of DM methods and tools, is an essential step toward a better integration of DM and DS. However, the development of such a system is currently facing two major challenges: the support of non-expert data miner and the definition of DM knowledge. Formalized and computerized ontologies, as a new research area for knowledge conceptualization, possess a great potential to help resolve the above problems. Due to its powerful knowledge representation formalism and associated maintenance mechanism, integrating an ontology into a data mining assistant system will be an effective way of making the system more intelligent and helpful for decision makers.

The objective of the research is to develop an ontology-based approach for data mining. It includes a data mining ontology, which creates a complete data mining domain knowledge base, and an ontology evolution tool, which provides a mechanism to support the ontology development and updating. This research provides a fundamental part of a larger project which aims to develop an intelligent data mining assistant system.

Based on protégé (Stanford University) and the OWL language, a finely designed DM ontology is successfully established. The role of the DM ontology is to represent the data mining knowledge required in the system. Two types of knowledge are represented: data mining domain knowledge that consists of both the methodology and the detailed applicable knowledge of the entire data mining process, and system generated knowledge that consists of data annotation and CBR case representation. To provide more intelligent support for data mining activities, our DM ontology is further integrated with the other two system components: a data warehouse and a case-based reasoning system.

Furthermore, a new ontology evolution methodology is proposed and implemented as a Protégé plug-in. This methodology is based on the evolution tasks and the consequence of the change operations. The different change operations and evolution tasks are finely defined in the methodology. The plug-in groups and arranges the necessary steps of most commonly used evolution tasks. It can be used as a step-by-step wizard to guide decision makers to execute ontology-updating tasks.

The results of this research have led to the construction of a fundamental framework for our data mining assistant system and pave the way for a better integration of data mining and decision support system. Furthermore our versatile DM ontology evolution methodology will greatly improve the accuracy, consistency, and efficiency of the evolution tasks. More importantly, this evolution methodology possesses a great potential for further development.

UNE ONTOLOGIE FORMELLE POUR LE FORAGE DE DONNÉES : PRINCIPES, CONCEPTION ET ÉVOLUTION

Yanfen Shen

SOMMAIRE

Le forage de données (DM) et les systèmes d'aide à la décision (DS) sont deux domaines relativement indépendants qui sont largement appliquées dans la recherche scientifique et la gestion. L'intégration réussie des technologies associées à ces deux domaines pourra mener à la réalisation d'un système intelligent puissant pour soutenir la prise de décision. L'application d'un système intelligent d'aide au forage de données, qui peut fournir une aide intelligente au delà de nombre de méthodes et d'outils de DM, est une étape essentielle vers une meilleure intégration de DM et de DS. Cependant, le développement du système doit relever actuellement deux défis principaux : le support de foreur de données non expert et la définition de la connaissance de DM. Les ontologies formelles et informatisées, en tant que nouveau secteur de recherche pour la conceptualisation de la connaissance, possèdent un grand potentiel pour aider à résoudre les problèmes ci-dessus. En raison de son puissant formalisme de représentation de la connaissance et du mécanisme d'entretien associé, intégrer une ontologie dans le système d'aide au forage de données sera une manière efficace de rendre le système plus intelligent et utile pour des décideurs.

L'objectif de cette recherche est de développer une approche basée sur l'utilisation d'une ontologie dans le domaine du forage de données. Cette dernière constitue une base de connaissance du domaine de forage de données ; ainsi qu'une méthode d'évolution d'ontologie, laquelle fournit un mécanisme et un outil pour soutenir le développement et la mise à jour de l'ontologie. Cette recherche est une partie fondamentale d'un plus grand projet de développement d'un système intelligent d'aide au forage de données.

Basé sur Protégé (Université de Stanford) et le langage OWL, nous avons établi avec succès une fine ontologie de DM. Le rôle de l'ontologie de DM est de représenter la connaissance de forage de données nécessaire au système. Deux types de connaissance sont représentés : la connaissance du domaine de forage de données et la connaissance produite par le système. La connaissance du domaine de forage de données se compose de la méthodologie et de la connaissance détaillée applicable au processus de forage de données, alors que la connaissance produite par le système se compose de la représentation d'annotations de données et des cas du système à base de cas. Pour fournir un support plus intelligent pour des activités de forage de données, notre ontologie de DM reste intégrée avec les deux autres composants du système : l'entrepôt de données et le système de raisonnement basé sur les cas.

Nous avons aussi proposé une nouvelle méthodologie d'évolution d'ontologie, cette méthodologie est mise en application comme un plug-in de Protégé. Cette approche est basée sur les tâches de l'évolution de l'ontologie et la conséquence des opérations de changement. Cette approche définie avec grande précision les différentes opérations de changement et les tâches de l'évolution. Le plug-in groupe et arrange les étapes nécessaires des tâches de l'évolution les plus généralement utilisées. Il peut être employé comme un *wizard*, étape par étape, pour guider des décideurs pour exécuter les tâches de mise à jour de l'ontologie.

Les travaux accomplis dans cette recherche ont permis de construire la structure fondamentale pour notre système d'aide au forage de données et ont préparé le terrain pour une meilleure intégration du forage de données et des systèmes d'aide à la décision. En outre notre méthodologie d'évolution d'ontologie améliorera considérablement l'exactitude, l'uniformité et l'efficacité des tâches de l'évolution. À notre avis, cette approche évolutive possède un grand potentiel pour un développement ultérieur.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to Dr. Sylvain Delisle, my supervisor, for his constant encouragement and support throughout my study at UQTR. Without his ambition, creative ideas, clear expression and concern of the detail, this thesis would not have been possible.

I want to thank Dr. Ofelia Cervantes, Michel Charest, Marie-Chantal Denis, Francis Gasse, Remy Auclair, Michel D. Pepin and Dr. Mhamed Mesfioui for their great assistance and collaboration during my master's study.

Finally, I would like to thank my husband Changbin Mao, my little daughter Jenny Mao, my parents and my brother for their profound love and unwavering supports during my studies.

LIST OF FIGURES

LIST OF TABLES

Chapter 1 : INTRODUCTION

1.1 Background

Data mining (DM), the extraction of hidden predictive information from large databases, is a powerful new technology with great potential for applications in almost every area to support decision-making. Nowadays, with the tremendous growing competition, many organizations are facing serious challenges in data and information analysis when making decisions. One of the challenges is the increasing availability of large volumes of high-dimensional data occupying databases. Another is the competitive demand for the rapid construction and deployment of data-driven analysis. The third is the need to give end users analysis results in a form readily understandable, helping them gain the insights they need to make critical decisions. This trend requires intelligent support to deal with a very large volume of data and find new, interesting and useful patterns from data for effective decision-making. Data mining, as proposed by many researchers, is regarded as the best choice to assist decision makers to solve the "data rich but knowledge poor" problems.

Decision makers can make decisions based on the information and knowledge obtained through data mining processes, or in other cases, through decision support systems (DSS) for certain types of decisions. The decision support system is *"an interactive, flexible and adaptable computer-based information system especially developed for supporting the solution of a non-structured management problem for improved decision making"* [1]. It aims to increase the productivity of decision makers through implementing their abilities to manipulate knowledge, by facilitating problem solving, and by providing the assistance for non-structured problems. This requires a complex processing of data and information. The results depend largely on the quality of the applied data processes. Thus data mining and decision support systems are not two independent components to support decision-making. They interact with each other to combine useful data and information for decision-making process, improve the understanding of decision-making process, and generate new knowledge from decision-making process. Actually, they are interdependent; they can benefit from each other if they can be successfully integrated.

Proper integration of DM and DS will not only support required interaction between them but also present new opportunities for enhancing the quality of support provided by each system[2, 3, 4]. Mladenic *et al.* [2] proposed an equation, "data mining + decision tool = better business", which illustrates that integration can lead to more success in business. While integrated in DS, data mining will provide a strategic advantage in defining, developing, and deploying competitive business strategies. Data mining tools will predict future trends and behaviors, allowing business to make proactive, knowledge-driven decisions. They will also answer quickly and accurately business questions that were traditionally time consuming to resolve.

However, data mining and Decision Support System are currently not well integrated [5]. The problem of the poor integration has recently been clearly acknowledged in the scientific literature and so far little work has been done. The DSS mainly focuses on improving decision makers' ability in dealing with data, information and knowledge; while data mining methods and tools are supposed to be able to facilitate decision makers by finding interesting information from a sea of data and compacting it into a form easily amenable to decision making. However, currently tools provided by data mining cannot completely fulfill its task due to its lack of cooperation with DSS. As shown in Figure 1.1 [3],*"DM and DSS are two distinct components that do not usually interact through a computerized tool. Thus, DM and DSS are not integrated at all"*. [5] Therefore, the integration of DM and DSS remains one of the biggest challenges in the artificial intelligence field.

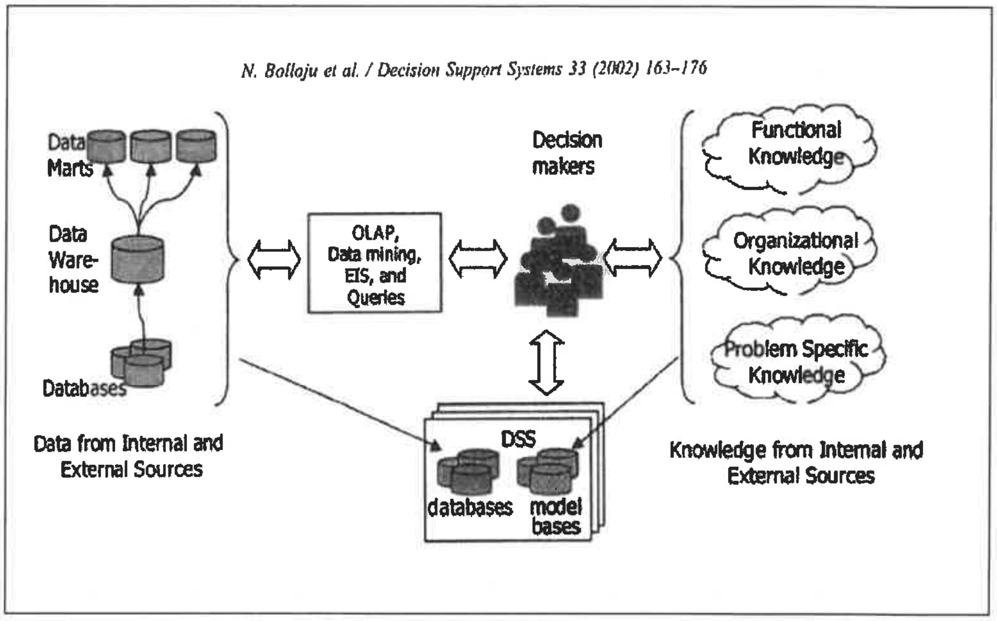

N. Bolloju et al. / Decision Support Systems 33 (2002) 163–176
Data Marts
Data Ware-house
Databases
Data from Internal and External Sources
OLAP, Data mining, EIS, and Queries
Decision makers
DSS
databases
model bases
Functional Knowledge
Organizational Knowledge
Problem Specific Knowledge
Knowledge from Internal and External Sources

Figure 1.1 Decision support and knowledge management activities

1.2 Challenges of intelligent data mining assistance

Data mining is not an easy, simple process; it is a discipline which brings together database systems, data warehouse, statistics, artificial intelligence, machine learning, parallel and distributed processing, and visualization. An intelligent data-mining assistant can make DM more accessible and effective for decision makers to support the application-oriented decision-support tasks. The application of the intelligent assistant systems is an essential step toward a better integration of DM and DSS, but it is currently facing the following challenges [6, 7, 8].

How to support the non-expert data miner

Data mining is a complicated process that ranges from specifying DM objects, data preprocessing, selecting algorithms, and models to evaluating DM results. For each step, some important decisions must be made. For example, how to convert business objectives into DM objectives; how to perform the data preparation phase; how to choose the most appropriate DM algorithm and its parameters; how to evaluate and interpret DM results, and so on. These decisions require a deep understanding of data mining concepts, and only expert data miners can handle this detailed knowledge. Unfortunately, most commercial products either do not offer any intelligent assistance or tend to offer only "wizard-like" interfaces that tend to assume a high level of background knowledge to use the system. In order to make DM tools more applicable and practical for the potential users of all levels, including ordinary decision makers, special consideration must be taken in how to effectively support the non-expert users when designing a data mining assistant.

How to define the DM knowledge

Over the past several decades, the field of statistics and machine learning has evolved at a tremendous pace. This results in a myriad of algorithms and associated knowledge available to data miners. The effective use of some algorithms requires a data miner to possess a great deal of basic knowledge to carry out a given step. This basic knowledge consists of domain knowledge describing DM concepts and tacit knowledge explaining the experience from the well-accomplished DM tasks.

Data mining domain knowledge is a basic requirement of DM intelligent systems. Often the amount of encapsulated knowledge determines the "level" of intelligence the system can provide. With the increasing growth of DM technologies, the corresponding DM knowledge becomes multidisciplinary covering more and more relative fields. The contents of the domain knowledge also become increasingly richer and deeper, and the accuracy and the versatility of knowledge interpretation become more and more important and difficult. Nevertheless, many DM methodologies are not capable of providing enough necessary, detailed knowledge for the novice miners. In most cases, they only specify the phases, tasks and activities that need to be carried out during a DM project. Thus, how to formalize the domain knowledge that can be further shared and reused by different applications is still a challenge.

Another source of knowledge is the tacit knowledge from the data miners. Tacit knowledge often deals with the practice experience and the personal knowledge of various DM tasks; it could be used to assist in answering important questions during the DM process. Most enterprises do not directly manage tacit knowledge in a form that can be stored, refined and reused. Therefore, how to externalize tacit knowledge to make it explicit remains another challenge.

1.3 Ontology

Ontology is currently a hot research area under development that is catching increasing interest in many industrial and academic fields, especially in artificial intelligence. Ontology is a knowledge representation mechanism for better structuring domain knowledge. Existing knowledge sources are mapped into the domain ontology and semantically enriched. This semantically enriched information enables better knowledge sharing and automatic processing and, implicitly, a better management of knowledge. Based on this characteristic, ontology-based systems seem to be the best choice for knowledge management systems.

In computer science, ontology is defined as a formal specification of a particular view on the important concepts within a respective domain. Typically, an ontology consists of a hierarchy of concepts with a specification of their characteristics and relationships. The idea of applying ontologies to knowledge management is due to the fact that computers can exploit the knowledge contained in an ontology to handle information in a way similar to humans (who share the same knowledge). The usage of ontology has several advantages [9]. Ontologies can facilitate interoperability between applications by capturing a shared understanding of a specific domain, they can provide a formalization of the shared understanding that makes them machine-processable, the explicit representation of the semantics of data through ontologies enables applications to provide a new level of services such as verification, justification etc.

Due to the powerful knowledge representation formalism and associated inference mechanism, adopting an ontology into the data mining intelligent assistant system will be an effecting way of making the system more powerful and helpful for decision makers. An ontology can be used to solve the knowledge definition problems as discussed in section 1.2 in the assistant system. As the data mining knowledge is a necessary background to data mining activities, the ontology can play an essential role as the backbone of the data mining assistant system. The ontology can structure and model DM domain knowledge and the relationships between different concepts. The knowledge represented in an ontology will not only give the intelligent system more accessible information, but also provide a common semantic agreement between different components to share and reuse. Therefore, the assistant system will be ontology based. On the other hand, from the point of view of users, the ontology can contribute more intelligence to the system. The contextual knowledge defined in the ontology may help data miners select the appropriate information, features or techniques, prune the space of hypothesis, represent the output in a most comprehensible way and improve the process. Ontology can also allow semantic search and combination of DM knowledge; this will enable users to incorporate necessary knowledge into the DM process. In a few words, formal and computerized ontologies offer a promising technology that has great potential for the data mining field, and an ontology-based data mining system could become a truly intelligent data mining assistant system

1.4 Research objectives

In order to provide better support for decision makers who will conduct data mining activities, our project aims to create an intelligent data mining assistant system. The whole project will be accomplished by our data mining research team at UQTR. The assistant system can empower data miners with the understanding of basic concepts and assist them to make right choices throughout various phases of the DM process for a particular data-mining task, and eventually, help them make better decisions. The architecture of our system will mainly consist of a DM ontology, a case-based reasoning system and a data warehouse.

The main objective of this work is to develop an ontological approach to data mining for decision makers. Our research work mainly includes the following three parts:

Part 1: To design and build a data mining ontology. This DM ontology will model and represent the various concepts of data mining process and data mining domain knowledge. It will also specify the relationships among the concepts. This ontology will facilitate the knowledge sharing and reuse among decision makers. Our DM ontology will be developed using the OWL language and based on the Protégé ontology editor [10] (Stanford University).

Part 2: To integrate a DM ontology into the intelligent data mining assistant system. Particularly, the ontology will interact with a case-based reasoning system and the metadata of a data warehouse. This integration will assist decision makers in specifying the data, the cases, and the necessary data mining technique more efficiently for a given DM task.

Part 3: To develop an ontology evolution tool. This new tool will support the users in dealing with the activities of ontology updating and maintenance. It will provide a step-by-step guide to help users, especially non-expert users, capture all the necessary works for the most commonly used evolution tasks. This tool will be integrated into Protégé as a new plug-in that can be used for any Protégé OWL ontologies.

Structurally, the thesis consists of seven chapters. Chapter 2 reviews briefly the concepts of data mining. Chapter 3 reviews the current state of ontology research relevant to this work, including the fundamental concepts, the OWL language, the ontology evolution strategies, and the ontology based application in data mining domain. Chapter 4 describes the procedures of the development of the new data mining ontology. This includes the definition of the functions, the design and the implementation of the DM ontology in the intelligent assistant system. Chapter 5 presents the conceptual solutions of Protégé Owl ontology evolution, and the details on development of the ontology evolution plug-in. The possible future works and general conclusions are given in chapter 6 and chapter 7 respectively.

Chapter 2 : DATA MINING FUNDAMENTALS

2.1 Basic concepts

2.1.1 Data mining definition

Data mining is the nontrivial extraction of implicit, previously unknown, interesting, and potentially useful information from data. The extracted knowledge is used to describe the hidden regularity of data, to make prediction, or to aid human users in other ways. It is usually in a form of knowledge patterns or models. From a business perspective, data mining is defined as a decision support process in which we search for patterns of information in data.

Data mining is usually classified into two categories: prediction and description. Prediction focuses on using some variables or fields in the database to predict unknown or future values of other variables of interest, while description or discovery focuses on finding hidden knowledge patterns or regularities without a predetermined idea or hypothesis about what the pattern may be.

Data mining is primarily used today in companies with a strong customer focus – retail, financial, communication, and marketing organizations. It enables these companies to determine relationships among internal factors such as price, product positioning, or staff skills, and external factors such as economic indicators, competition, and customer demographics. It also enables them to determine the impact on sales, customer satisfaction, and corporate profits. Finally, it enables them to "drill down" into summary information to view detail transactional data.

2.1.2 Data mining and KDD process

Knowledge Discovery in Databases (KDD) [11] is the overall process of discovering useful knowledge from data, which is widely adopted in the field of knowledge management and data mining. Fayyad et al [12] define the KDD as "the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data. Figure 2.1. It includes numerous steps, summarized as:

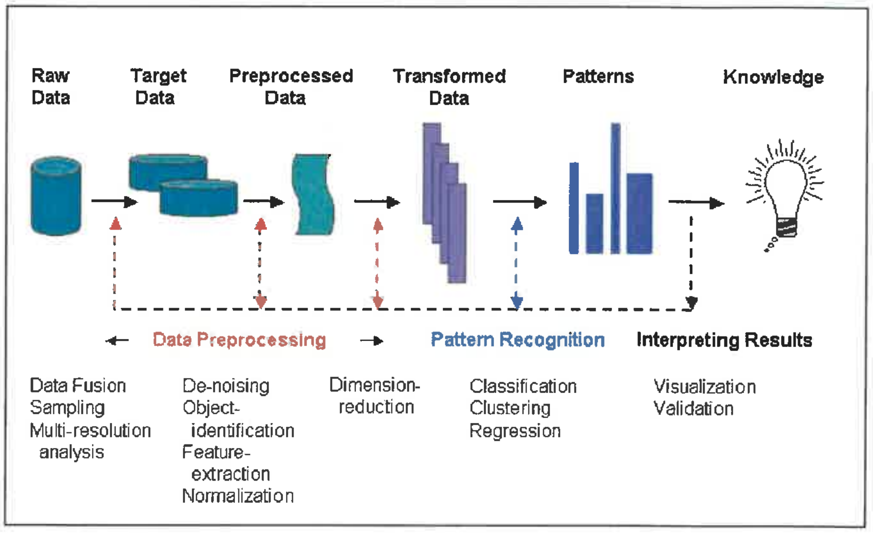


Figure 2.1 The KDD process

1. Learning the application domain: includes relevant prior knowledge and the goals of the application.
2. Creating a target dataset: includes selecting a dataset or focusing on a subset or data samples on which discovery is to be performed.
3. Data cleaning and preprocessing: includes basic operations such as removing noise or outliers, handling missing values, etc.
4. Data reduction and projection: includes finding useful features to represent the data, reduce the number of variables or find invariant representations for the data.
5. Choosing the function of data mining: includes deciding the purpose of the model derived by the data mining algorithm

6. Choosing the data mining algorithms: includes selecting methods to be used for searching for patterns in the data and matching a particular data mining method with the overall criteria of the KDD process.
7. Data mining: includes searching for patterns of interest in a particular representational form or a set of representations, such as classification, regression, clustering etc.
8. Interpretation: includes interpreting and visualizing the discovered patterns and translating the useful ones into terms understandable by users.
9. Using discovered knowledge: includes documenting and reporting the discovered knowledge, incorporating the knowledge into the performance system, and taking actions based on the knowledge.

Sometimes the two terms KDD and data mining are used interchangeably. However, from a research-oriented perspective in computer science, knowledge discovery in databases, or KDD, is aimed to set up an infrastructure for data mining at the organizational level. KDD is used to refer to the broad process of finding knowledge in data, while data mining refers to the actual algorithms used in the discovery process. Nevertheless, in business community the term data mining is used in a broader sense, because it refers to both the infrastructure and the algorithms. In addition, KDD implies the data reside in databases, while data mining could be conducted at data sets stored in any format. Since this work intends to concentrate on data mining aspects from a broad perspective and model data mining techniques for all the phases from data understanding to model evaluation, we will use the term data mining to refer to both infrastructure and algorithms.

2.1.3 Data mining and data warehouse

Data mining may involve data from multiple data sources, which may be located in a distributed database system. The complexity of distributed database systems makes data mining more difficult when dealing with data preparation. Data warehouses provide an excellent environment for database-centric data mining.

Data warehouse is an integrated environment, containing integrated data, detailed and summarized data, historical data, and metadata. An important advantage of performing data mining in such an environment is that the data miner can concentrate on mining data, rather than cleaning the integrating data. Data warehousing provides an effective approach to deal with complex decision support queries over data from multiple sites. A key advantage of the data warehousing approach is to create a copy of all the data at one location, and to use the copy rather than going to the individual sources. Data warehouses contain consolidated data from many sources, spanning long time periods, and augmented with summary information. Data warehouses are much larger than other kinds of databases, sizes are larger, typical workloads involve ad hoc, fairly complex queries, and fast response times are important. Data warehouses are usually integrated with OLAP (OnLine Analytical Processing) to benefit the data preparation phase of data mining.

2.2 CRISP-DM

CRISP-DM [13] is a comprehensive data mining methodology and process model. It provides not only guidance to all data miners from beginners to experts but also a generic process model that can be specialized according to the needs of any particular industry or company. CRISP-DM organizes the data mining process into six phases: business understanding, data understanding, data preparation, modeling, evaluation, and deployment. These phases help organizations understand the data mining process and provide a road map to follow while planning and carrying out a data mining project.

The whole data mining process is shown in Figure 2.2 [13]. The arrows indicate the most important and frequent dependencies between the phases, while the outer circle symbolizes the cyclical nature of data mining itself and illustrates that data mining process is an iterative process. Figure 2.3 [13]outlines each phase of the data mining process.

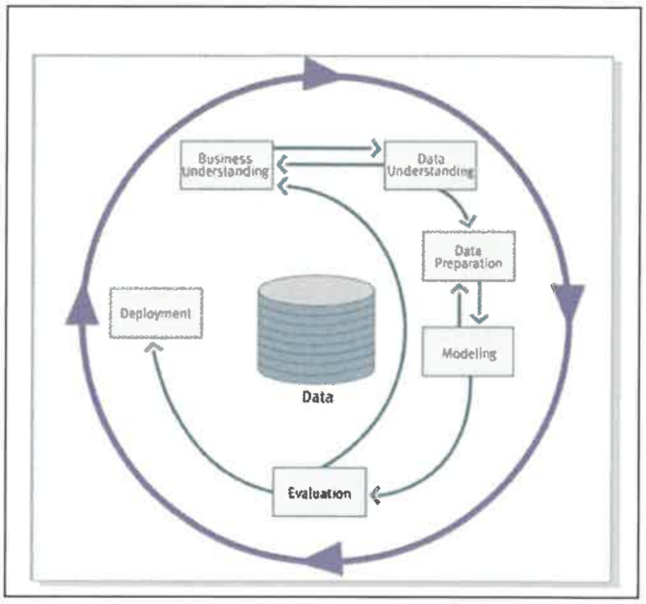
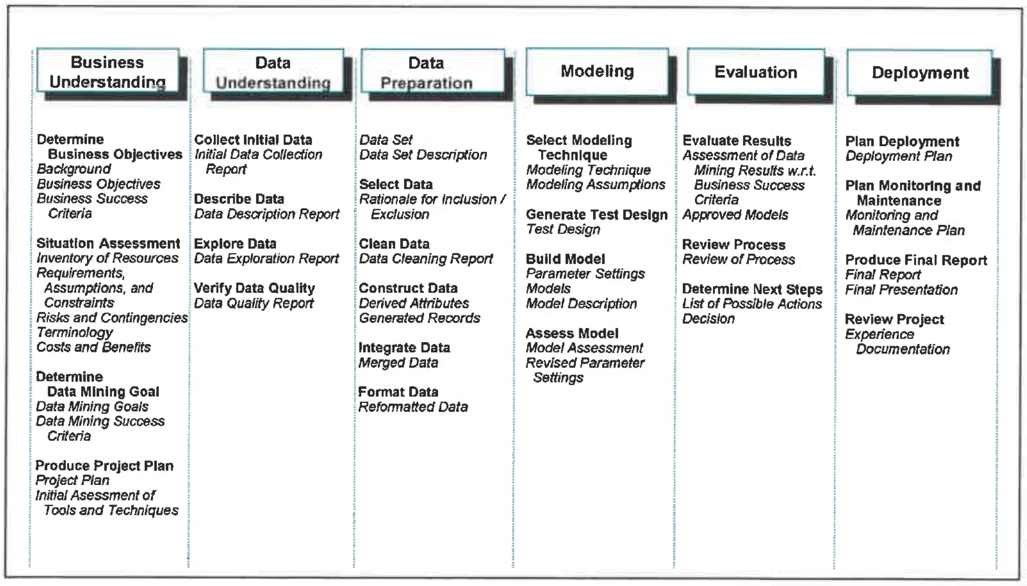


Figure 2.2 The CRISP-DM process



Business Understanding	Data Understanding	Data Preparation	Modeling	Evaluation	Deployment
Determine Business Objectives *Background* *Business Objectives* *Business Success Criteria* **Situation Assessment** *Inventory of Resources* *Requirements, Assumptions, and Constraints* *Risks and Contingencies* *Terminology* *Costs and Benefits* **Determine Data Mining Goal** *Data Mining Goals* *Data Mining Success Criteria* **Produce Project Plan** *Project Plan* *Initial Asessment of Tools and Techniques*	**Collect Initial Data** *Initial Data Collection Report* **Describe Data** *Data Description Report* **Explore Data** *Data Exploration Report* **Verify Data Quality** *Data Quality Report*	*Data Set* *Data Set Description* **Select Data** *Rationale for Inclusion / Exclusion* **Clean Data** *Data Cleaning Report* **Construct Data** *Derived Attributes* *Generated Records* **Integrate Data** *Merged Data* **Format Data** *Reformatted Data*	**Select Modeling Technique** *Modeling Technique* *Modeling Assumptions* **Generate Test Design** *Test Design* **Build Model** *Parameter Settings* *Models* *Model Description* **Assess Model** *Model Assessment* *Revised Parameter Settings*	**Evaluate Results** *Assessment of Data Mining Results w.r.t. Business Success Criteria* *Approved Models* **Review Process** *Review of Process* **Determine Next Steps** *List of Possible Actions* *Decision*	**Plan Deployment** *Deployment Plan* **Plan Monitoring and Maintenance** *Monitoring and Maintenance Plan* **Produce Final Report** *Final Report* *Final Presentation* **Review Project** *Experience Documentation*

Figure 2.3 The phases, tasks and outputs of the CRISP-DM process

Phase one: Business understanding. The initial business understanding phase focuses on understanding the project objectives from a business perspective, converting the project objectives into a data mining problem definition, and then developing a preliminary plan to achieve the objectives. This phase involves several steps, including determining business objectives, assessing the situation, determining the data mining goals, and producing the project plan.

Phase two: Data understanding. This phase starts with an initial data collection. The analyst then proceeds to increase familiarity with the data, to identify data quality problems, to discover initial insights into the data, or to delete interesting subsets to form hypotheses about hidden information. This phase involves four steps: the collection of initial data, the description of data, the exploration of data, and the verification of data quality.

Phase three: Data preparation. This phase covers all the activities to construct the final data set or the data that will be fed into the modeling tools from the initial raw data. The five steps in this phase are the selection of data, the cleaning of data, the construction of data, the integration of data, and the formatting of data.

Phase four: Modeling. In this phase, various modeling techniques are selected and applied and their parameters are calibrated to optimal values. Typically, several techniques exist for one data mining problem type, and some techniques have specific requirements on the form of data. Therefore, stepping back to the data preparation phase may be necessary. The modeling phase includes the selection of the modeling technique, the generation of test design, the creation of models, and the assessment of models.

Phase five: Evaluation. Before proceeding to final deployment of the model, it is important to thoroughly evaluate the model and review the model's construction to be certain it properly achieves the business objectives. Here it is critical to determine if some important business issue has not been sufficiently considered. The key steps for the evaluation phase are the evaluation of the results, the process review, and the determination of next steps.

Phase six: Deployment. Model creation is generally not the end of the project. The data mining results and the knowledge gained must be organized and presented in a way such that the decision makers can use it, which often involves applying “live” models within an organization’s decision-making processes.

Chapter 3 : A BRIEF STATE OF THE ART IN ONTOLOGY RESEARCH

3.1 Fundamentals of ontology

3.1.1 Some definitions

Ontology is a term that comes from Philosophy, where it means a systematic explanation of being. In the last decade, this word has become relevant for the knowledge engineering community. Ontology is used to facilitate knowledge representation, sharing and reuse. Ontology can take different meanings in different domains. One of the first definitions was given by Neches and colleagues [14], who defined an ontology as follows:

An ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary.

This definition describes how to build an ontology, identifies the basic terms and relations between terms, and also the rules to combine the terms.

The most widely quoted ontology definition is given by Gruber [15]:

An ontology is an explicit specification of a conceptualization.

Ontologies are used as a specification mechanism to represent knowledge based on a conceptualization. A conceptualization of the domain starts with the identification of the abstract or concrete objects and the relationships between them. A domain can be conceptualized differently from different viewpoints. In general there is no unique conceptualization of a domain. An ontology expresses a viewpoint on the knowledge of a domain. A specification of a conceptualization provides the foundations for building conceptual vocabularies for knowledge sharing. These shared vocabularies can become computable models if they are implemented in an ontology specification language.

Based on Gruber's definition, many definitions of what an ontology is were proposed. Borst modified slightly Gruber's definition as follows [14]:

Ontologies are defined as a formal specification of a shared conceptualization.

Gruber's and Borst's definitions have been merged and explained by Studer and colleagues as follows [14]

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.

In 1995, Guarino and Giaretta [14] collected and defined an ontology as *(1) a philosophical discipline*, (2) *an informal conceptual system, (3) a formal semantic account, (4) a specification of a conceptualization, (5) a representation of a conceptual system via a logical theory, (6) the vocabulary used by logical theory, and (7) a (meta-level) specification of a logical theory.*

Although the different definitions of ontology discussed above provide different and complementary points of view, they share some common features: ontologies aim to capture and model knowledge in a generic way, and they may be reused and shared across software applications and by groups of people.

3.1.2 Ontology components and types

Ontology components vary in different ontology languages, but they also share some common elements. According to the logic formalism of modeling techniques, ontology components can be classified into two groups. One is based on frames and first order logic. This group identifies five kinds of components: classes, relations, functions, formal axioms and instances. Another group is based on Description Logic (DL). DL systems allow the representation of ontologies with three kinds of components: concepts, roles, and individuals. Concepts represent classes of objects, roles describe binary relations between concepts that allow the description of properties of concepts, and individuals represent instances of classes. An example language in this group is OWL, which will be discussed further.

It is important to know that there are some connections and implications among the knowledge modeling components, the knowledge representation paradigms and the languages used to implement the ontologies under a given knowledge representation paradigm. That is, an ontology built with frames or description logics can be implemented in several frames or description logics languages.

There are various categories of ontologies. Ontologies can be classified with different criteria. The categorization can be made based on the subject of the conceptualization, the information that the ontology needs to express and the richness of their internal structure, and the level of dependence on a particular task or point of view.

Table 3.1 gives a detailed classification and description of ontologies based on the conceptualization subject.

Table 3.1 Types of ontology

Ontology type	Description	Examples
Knowledge Representation (KR) ontologies	Capture the representation primitives used to formalize knowledge under a given KR paradigm	- Frame Ontology (Gruber, 1993) - OKBC Ontology - OWL KR Ontology
General or common ontologies	Used to represent common sense knowledge reusable across domains.	Mereology Ontology (Borst, 1997) Standard-Units Ontology
Top-level or Upper-level ontologies	Describe very general concepts and provide general notions under which all root terms in existing ontologies should be linked.	IEEE Standard Upper Ontology (SUO) Cyc's Upper Ontology
Domain ontologies	Are reusable in a given specific domain. Provide vocabularies about concepts within a domain and the relationships, about the activities taking place in that domain, and about the theories and elementary principles governing that domain.	UNSPSC (the United Nations Standard Products and Services Codes) (for computer equipment) NAICS (North American Industry Classification System)
Task ontologies	Describe the vocabulary related to a generic task or activity by specializing the terms in the top-level ontologies.	Scheduling Task Ontology
Domain-task ontologies	Are task ontologies reusable in a given domain, but not across domains. Are application-independent.	Plan-surgery ontology
Method ontologies	Give definitions of the relevant concepts and relations applied to specify a reasoning process so as to achieve a particular task.	Scheduling by means of task decomposition
Application ontologies	Contain all the definitions needed to model the knowledge required for a particular application	Application ontology for travel agencies

The reusability-usability trade-off problem applied to the ontology filed states that the more reusable an ontology is, the less usable it becomes, and vice versa. Figure 3.1 [14] presents the reusability-usability trade-off of ontologies. Upper-level, general, and domain ontologies capture knowledge in a problem-solving independent way, whereas method, task, and domain-task ontologies are concerned with problem solving knowledge.

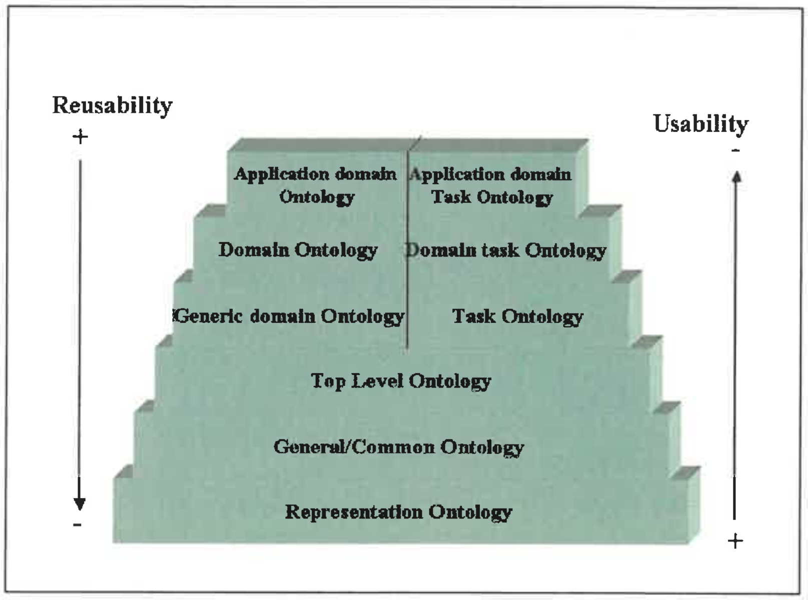


Figure 3.1 The reusability-usability trade-off

3.1.3 Ontology and semantic web

The Semantic Web [16, 17, 18] based on a vision of Tim Berners-Lee, is the next generation of the WWW. "It is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation." [16]. The great success of the current WWW leads to a new challenge: a huge amount of data is so unstructured that they can only be understood by humans, but the amount of data is so huge that they can only be processed efficiently by machines. The Semantic Web aims to build a WWW architecture that enhances the web contents with formal semantics, thus making data, information and knowledge machine-processable.

Ontologies are the backbone of the Semantic Web. They enable machine understandable information representation and information exchange. This means ontologies can establish common vocabularies; define shared and common domain concepts, their relationships and their semantics. They can help both human and machines to communicate concisely by supporting the exchange of semantics of data, information and knowledge rather than only the syntax. “The success of the deployment of the Semantic Web will largely depend on whether useful ontologies will emerge, allowing shared agreements about vocabularies for knowledge representation.” [19]. It is therefore important that any semantics of the web resources should be explicitly specified on ontologies. Only in this way can all the users reach a shared understanding by exploiting the contents of ontologies.

Berner-Lee believes that the application of ontologies on the web scale will greatly accelerate the development of the Semantic Web. In his vision, the Semantic Web will have several layers in its structure, as presented in Figure 3.2 [17].

The first two layers provide a common syntax. Uniform resource identifiers (URIs) provide a standard way to refer to entities, while Unicode is a standard for exchanging symbols. The XML layer formalizes the structure of the documents and XML Schema defines the grammars for valid XML documents. The RDF can be seen as the first layer where information becomes machine understandable. It is the foundation for processing metadata. RDF Schema defines a modeling language on top of RDF.

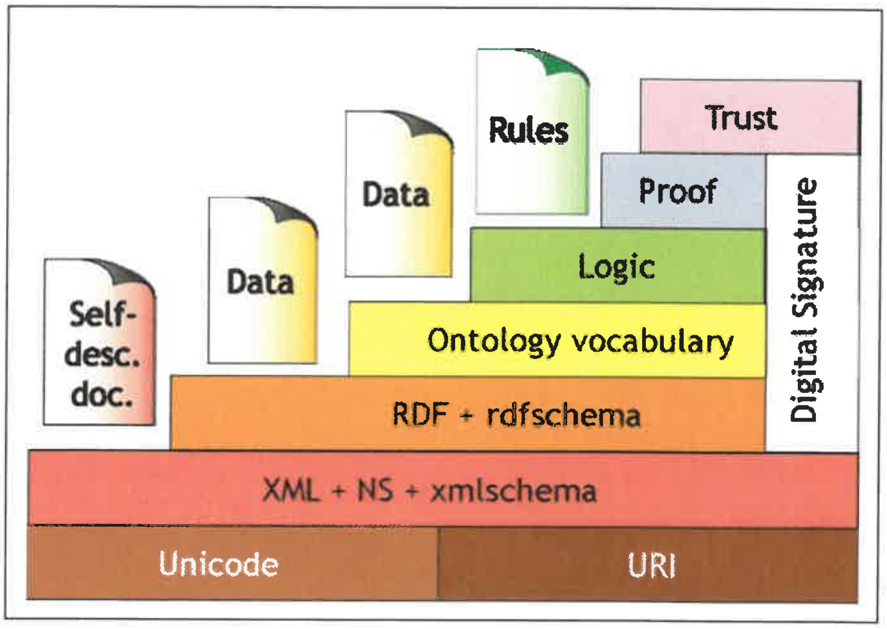


Figure 3.2 The architecture of the Semantic Web

The next layer is the ontology vocabulary, which is the main research area of the Semantic Web. Technologies such as XML, RDF and RDFS represent the basis of the ontology language, while the ontology vocabulary layer adds the semantic annotations to the web documents and represents the formal common agreement about the meaning of the data. The Semantic Web needs ontologies with a significant degree of structures. Most ontologies consist of a set of concepts, a hierarchy on them, and relations between concepts. Ontologies are also integrated with the logic layer because most ontologies allow for logic axioms. By applying logical deduction, one can infer new knowledge from the information that is stated implicitly in ontologies.

Proof and trust are the remaining layers. They check the validity of the statements made in the Semantic Web.

As the research on the Semantic Web keeps on growing, ontological engineering has become an essential part of ontology studies. Ontologies are facing some real challenges [20]: they must be developed, managed and endorsed by committed practice communities, and they must carefully define the data and allow interactions held in different formats.

3.2 OWL web ontology language

3.2.1 OWL language overview

The OWL Web Ontology Language [21, 22, 23] is a language for defining and instantiating web ontologies. It is the recommended standard language of W3C for describing the semantic web. An OWL ontology may include descriptions of classes, properties, instances and their relationships.

OWL is outstanding from other ontology languages in several aspects. OWL is designed for the Semantic Web, in which information is given explicit meaning, making it easier for machines to automatically process and integrate. OWL can formally describe the meaning of the terminology used in a Web application and the relationships between those terms; it has more facilities for expressing semantics than XML, RDF, and RDF-S. Thus OWL goes beyond these languages in its ability to represent machine interpretable content on the Web. OWL also supports more powerful reasoning techniques. There are a lot of available tools that can not only do some general works but also perform reasoning tasks about OWL ontologies.

The OWL language provides three increasingly expressive sub languages: OWL Lite, OWL DL and OWL Full. OWL Lite supports those users primarily needing a classification hierarchy and simple constraint features. OWL DL supports those users who want the maximum expressiveness without losing computational completeness and decidability of reasoning systems. OWL DL is so named due to its correspondence with description logic; it includes all OWL language constructs with restrictions. OWL Full is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. It is not actually a sub language since it contains all the OWL language constructs and provides free, unconstrained use of RDF constructs. Table 3.2 gives a brief comparison of these three sub languages.

Table 3.2 Comparison of three OWL sub languages

	OWL Lite	OWL DL	OWL Full
Usage	- Classification hierarchy, simple constraints	- Maximum expressiveness - High reasoning ability	- Maximum expressiveness - Free syntax - Unwarranted reasoning
Representation language	- Fundamental part - Subset of OWL DL	-All, but used under certain constraints -Based on description logic	-All, used freely without constraints -Extension of RDF
Reasoning	High efficiency	High efficiency	No warrantee

OWL is a powerful language with many language features. Figure 3.3 presents some main synopsis of its sub languages. This figure indicates that OWL Lite is the fundamental part of the OWL language, it also has more limitations on the use of the features than OWL DL or OWL Full. OWL DL and OWL Full expand OWL Lite in many important aspects.

The most important concepts in OWL ontology are classes, properties, instances of classes, and relationships between these instances. These concepts are discussed respectively in the next three sections.

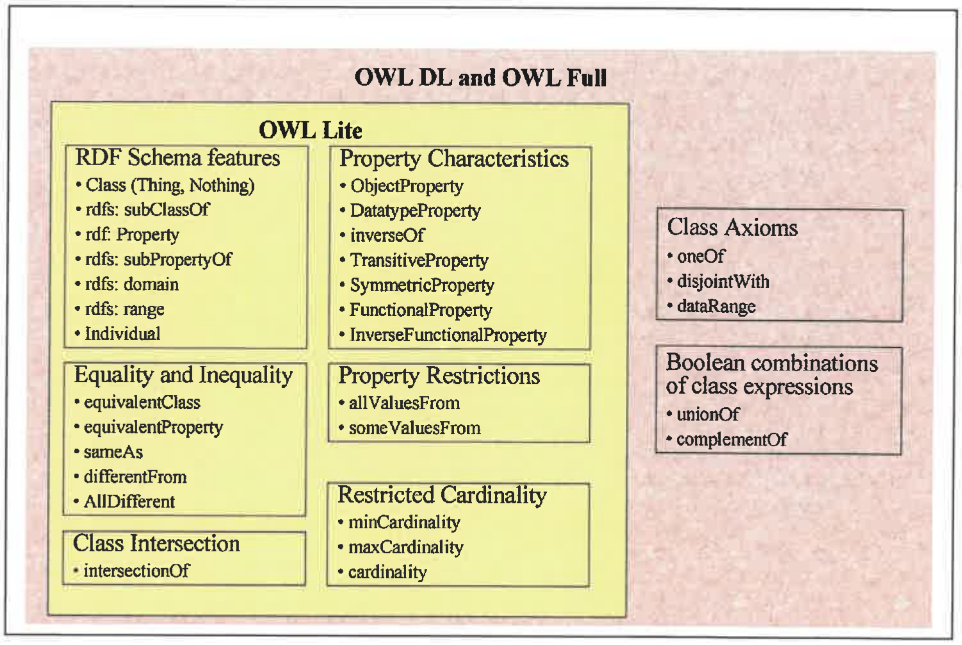


Figure 3.3 language synopsis of OWL sub languages

3.2.2 OWL classes

OWL classes provide an abstraction mechanism for grouping resources with similar characteristics. They can be used to represent the different concepts and their hierarchies in ontology. Every OWL class is associated with a set of individuals, called the class extension or instances. OWL classes are described through class descriptions and class axioms.

1. Class description

A class description describes an OWL class either by a class name or by specifying the class extension of an unnamed anonymous class.

OWL distinguish six types of class descriptions:

(1) A class identifier

(2) An exhaustive enumeration of individuals that together form the instances of a class

(3) A property restriction

(4) The intersection of two or more class description

(5) The union of two or more class description

(6) The complement of a class description.

Table 3.3 categorizes the language constructs with different class descriptions. Some language constructs are further described in detail right after the table.

Table 3.3 OWL class description type and its language constructs

Description type		**Language constructs**
Class identifier		- owl: Class - rdfs: subClassOf
Enumeration		- owl: oneOf
Property restrictions	Value constraints	- owl: allValuesFrom, - owl: someValuesFrom - owl: hasValue
	Cardinality constraints	- owl: maxCardinality - owl: minCardinality - owl: cardinality
Intersection, union, complement		- owl: intersectionOf - owl: unionOf - owl: complementOf

rdfs:subClassOf: Class hierarchies may be created by making one or more statements that a class is a subclass of another class.

owl:allValueFrom: It requires that for every instance of the class that has instances of the specified property, the value of the property are all members of the class indicated by the owl:allValuesFrom clause.

SomeValuesFrom: It is stated on a property with respect to a class. A particular class may have a restriction on a property such that at least one value for that property is the member of the class indicated by owl:someValuesFrom clause.

Owl:hasValue: It links a restriction class to a particular property value. A restriction containing a owl:hasValue constraint describes a class of all individuals for which the property concerned has at least one value semantically equal to the particular property value.

MinCardinality: If a minCardinality of 1 is stated on a property with respect to a class, then any instance of that class will be related to at least one individual by that property. This restriction is another way of saying that the property is required to have value for all instances of the class.

MaxCardinality: If a maxCardinality of 1 is stated on a property with respect to a class, then any instance of that class will be related to at most one individual by that property. A maxCardinality 1 restriction is sometimes called a functional property.

Cardinality: Cardinality is provided as a convenience when it is useful to state that a property on a class has both minCardinality 0 and maxCardinality 0 or both minCardinality 1 and maxCardinality 1.

2. Class axioms

Class descriptions form the building blocks for defining classes through class axioms. Class axioms typically contain additional components that state necessary and/or sufficient characteristics of a class. OWL contains three language constructs for combining class description into class axioms:

(1) rdfs: subClassOf: allows one to say that the class extension of a class description is a subset of the class description of another class description

(2) owl: equivalentClass: allows one to say that a class description has exactly the same class extension as another class description

(3) owl: disjointWith: allows one to say that the class extension of a class description has no members in common with the class extension of another class description

3.2.3 OWL properties

A property is a binary relation. OWL distinguishes two categories of properties: object properties that define the relations between instances of two classes and datatype properties that define the relations between instances of classes and data types. In other words, object properties link individuals to individuals while datatype properties link individuals to data values.

A property axiom defines the characteristics of a property. OWL supports the following constructs for property axioms:

(1) RDF Schema constructs: rdfs: subPropertyOf, rdfs:domain and rdfs:range

(2) Relations to other properties: owl:equivalentProperty and owl:inverseOf

(3) Global cardinality constraints: owl:FunctionalProperty and owl:InverseFunctionalProperty

(4) Logical property characteristics: owl:SymmetricProperty and owl: TransitiveProperty

rdfs:subpropertyOf: States that the property is a subproperty of some other property.

rdfs:domain: A domain of a property limits the individuals to which the property can be applied. If a property relates an individual to another individual, and the property has a class as one of its domains, then the individual must belong to the class. For example, the property hasChild may be stated to have the domain of Mammal. From this a reasoner can deduce that if Frank hasChild Anna, then Frank must be a Mammal.

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

owl:inverseOf: One property may be stated to be the inverse of another property. If the property P1 is stated to be the inverse of the property P2, then if X is related to Y by the P2 property, then Y is related to X by the P1 property.

owl:FunctionalProperty: Properties may be stated to have a unique value. If a property is a FunctionalProperty, then it has no more than one value for each individual (it may have no values for an individual).

owl:InverseFunctionalProperty: If a property is inverse functional then the inverse of the property is functional.

3.2.4 OWL individuals

Individuals are instances of classes, and properties may be used to relate one individual to another. Individuals are defined with individual axioms. There are two types of individual axioms:

(1) Axioms about class membership and property values of individuals

(2) Axioms about individual identity.

3.3 Existing ontology evolution methodologies

Due to the increasing number of ontologies in use and the increasing requirements associated with ontology updating and maintenance, ontology evolution becomes an oncoming hot topic nowadays. Stojavonic *et al.* [24] defines ontology evolution as "Ontology evolution is the timely adaptation of an ontology to the arisen changes and the consistent propagation of these changes to dependent artifacts." Ontology evolution is a quiet new question in the ontology research field; its importance has just been realized recently due to the tremendous growing speed of ontology applications. Even though many institutes and research groups are interested in how to maintain and evolve ontologies, there are not many fruitful results available now. Many areas of ontology updating remain to be explored and much more research is needed.

This section reviews some existing ontology evolution methodologies. Two well-developed evolution strategies are discussed in detail and concluded with their advantages and drawbacks. Some other strategies are also presented.

3.3.1 An ontology evolution process

Ontology consistency is an essential issue to be considered in the ontology evolution domain. As ontologies grow in size, the complexity of change management increases the difficulty of keeping ontologies consistent. Focusing on this problem, Stojanovic *et al.* [24, 25, 26] propose an ontology evolution process that enables resolving the given ontology changes and ensures the consistency of the underlying ontology and all its dependent applications.

The ontology evolution process contains six phases, occurring in a cyclic loop. Figure 3.4 [25] presents the entire evolution process. The process of ontology evolution starts with capturing changes either from explicit requirements or from the result of change discovery methods, which induce changes from existing data. In the *change representation* phase, a set of ontology changes is derived. Three levels of fine-grained changes are specified in this set: elementary changes, composite changes and complex changes. The *semantics of change* is the most important phase to deal with the effects of the change on the ontology consistency. It distinguishes syntax and semantic inconsistency. The possible problems that might be caused in the ontology by the identified changes are determined and resolved in this phase. For example, if a concept is removed, we should decide what to do with its instances. It enables the resolution of ontology changes in a systematic manner by ensuring the consistency of the ontology. The role of the *implementation phase* is to implement the changes identified in the previous two phases, to present the changes to the ontology engineer for final verification and to keep a log of the implementation changes. The *change propagation* phase ensures that all changes will be propagated to the interested parties and the consistency of dependent artifacts after an ontology update has been performed. Finally, the *change validation* phase allows the ontology engineer to review the changes and possibly undo them if desired.

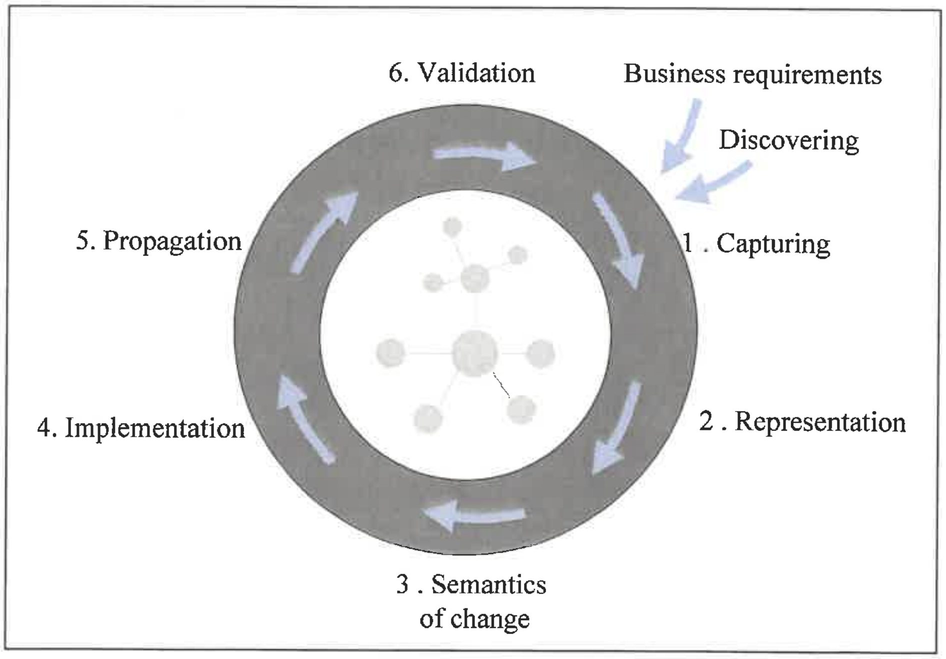


Figure 3.4 The ontology evolution process

To figure out the possible changes to be performed to one change request, one must consider the consistency of the ontology and its depending applications. There are many ways to achieve consistency after a change request. Thus, the concepts of the "evolution strategy" are introduced to customize this evolution process [24, 25]. To resolve a change, the evolution process needs to determine answers at many resolution points. The resolution points are the branches during change resolution where taking a different path will produce different results. Each possible answer at each resolution point is an elementary evolution strategy; a set of elementary evolution strategies is called an evolution strategy that defines how elementary changes will be resolved. For example, there are three strategies (or resolution points) to determine how to handle orphaned concepts: orphaned concepts are deleted, reconnected to their parents, or reconnected to the root concepts. Typically, a particular evolution strategy is chosen by the user at the start of the evolution process.

This methodology is implemented in the KAON framework [27]. KAON is an ontology management infrastructure allowing ontology management and application. Ontology evolution is realized through both KAON API and an UI application OI-modeller. OI-modeller supports ontology editing and evolution. In fact, the ontology is edited through an evolution process in this application.

The contribution of this methodology is that it defines a general, user-driven ontology evolution process that can be applied to different tools with different ontology languages. The process takes into account the consistency of ontology and its dependent applications; it can also make some suggestions to users. Another contribution is the concept of resolution strategy, which makes several paths to resolve one evolution problem possible, while other approaches only deal with one simplest solution. The third contribution is its excellent performance of reversibility, which is realized with a change ontology defined in KAON language and change log. However, our experience in using the software OI-modeller reveals that this approach has some drawbacks. First, it does not consider the interdependence of ontology change types. At least some evolution operations cause isolate results, an example is if we try to add a property into an existing class, this property could not be "transferred" to the instances of this class. Second, some actions of ontology evolution are not very well defined. For example, if we add an instance into a class, we do not know how to deal with its membership values and its property values, all we did is just adding an instance name. We believe these are serious drawbacks.

3.3.2 OWL ontology change management

When ontologies are built by several experts and used in a distributed and dynamic environment, the support for ontology evolution becomes extremely important, especially when there are at least two versions of the ontology available at the same time. Some essential issues must to be considered when dealing with the change of an ontology: the different change representations, the incomplete change information, the data transformation, the consistent reasoning between the two versions, etc.

Based on these considerations, a component-based framework for ontology evolution is proposed in [28, 29]. This methodology manages ontology evolution between two versions. It focuses on the change management of the ontology, assuming that two versions are already existing, but the changes information may be represented in different format and might be incomplete. The framework relates the available change information and provides mechanisms to derive new pieces of information from existing information.

The components of the framework are showed in Figure 3.5 [28]. Between the versions is the *minimal transformation set*, which provides a set of change operations that specify how V_{old} can be transformed into V_{new}. This is the kernel of the framework. The minimum transformation set is specified with the operations from the *ontology of change operations*, which defines a large number of standard changes to an ontology. Together with the minimal set, the *complex change operations* can be used to create data transformation scripts. The *structural diff* is designed for visualizing differences between ontologies, while the *conceptual relations* can facilitate data access by improving data interpretation and data source query.

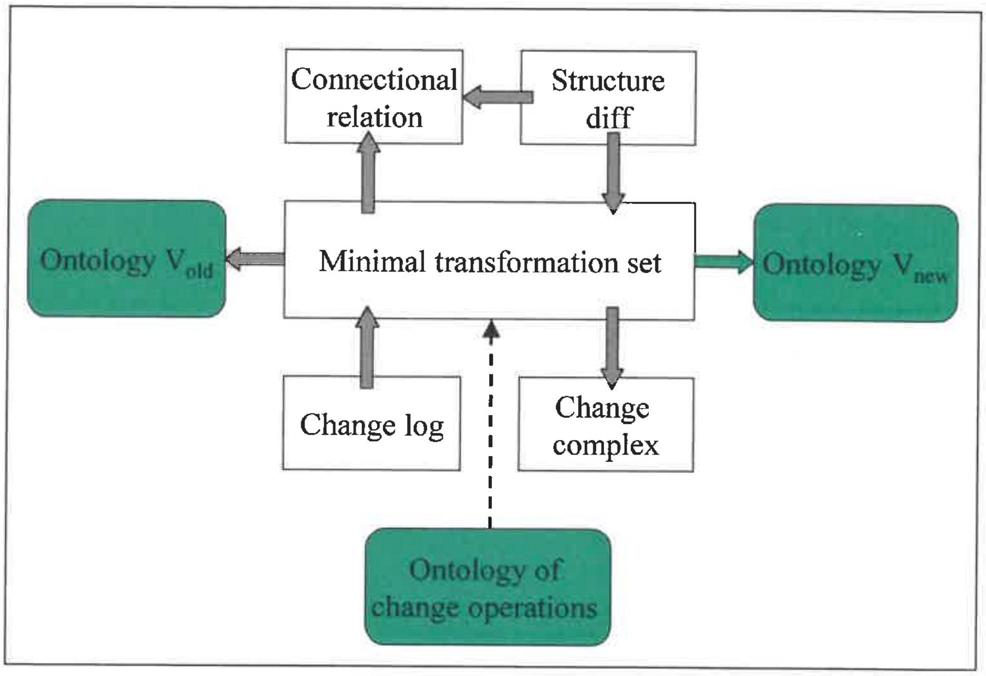


Figure 3.5 A schematic representation of the framework

The ontology of change operation is an essential element of the framework. It provides a vocabulary and syntax to express an accurate specification of change for an OWL ontology. The ontology of change distinguishes basic changes and complex changes. Each of the basic operations deals with only one specific feature of the ontology such as adding, removing, or value modifying the features. Complex operations provide a mechanism for grouping a number of basic operations that together constitute a logical entity. The change operations are modeled as a hierarchy of classes, where each class represents a specific type of change operation. The complex changes are defined as an extension of the basic changes. The complex operation can be distilled from a set of basic operations through a number of rules and heuristics.

The available tools for this approach are OntoView [30] and PrompDiff [31]. OntoView implements a change detection procedure for RDF-based ontologies. The role of this tool is to produce a transformation set. PrompDiff is a plug-in of Protégé, the change management strategy is implemented in two extensions of PrompDiff.

In conclusion, this methodology provides a framework for managing changes of a distributed ontology. It can complete the change information between two versions of the ontology by deriving the new change information from the existing incomplete information. Thus, it is possible to find the inconsistency of two versions, to ease the ontology updating and data access. The ontology of change operation is another contribution. It classifies two levels of change operation and proposes a hierarchy of the whole change operations to form an operation system. This ontology can be regarded as the representation format for OWL ontology changes. However, this methodology is based on the assumption that the ontology has already evolved. Although the change specifications are very useful for the distributed ontology-based applications, this methodology is not for ontology evolution, instead, it is rather a post-evolution methodology.

3.3.3 Other methodologies

A method of ontology evolution using document clustering is proposed for domain ontologies of the semantic web [32]. This method firstly search web documents based on a set of initial URLs, and characterizes the documents with these representative keywords. It then maps both the key words and web documents to the domain ontology concepts so that the web documents can be characterized again with the ontology concepts. A clustering algorithm is then applied to the web documents. The results of the clustering are used to derive proposed ontology changes

Another methodology of ontology evolution is to capture the change requirements and recommend change operations to a personalized ontology based on the usage information of the individual ontologies in a user community [33]. The relevance of ontology change operations are determined by a collaborative filtering algorithm taking into account the similarity of the user's ontologies.

All methods discussed above are all semi-automatic methodologyes; the ontology engineers or the users are involved in the evolution process. Due to some inconvenience of the interaction between human and tools, an idea of automatic evolution methodology is proposed in [34]. This idea aims to automatically perform ontology evolution process without human supervision. This research direction is based on the concepts of belief change and tries to exploit it into the ontology evolution field. However, this study does not provide any concrete solutions to the problem. This theoretical proposal needs more development and applications.

3.4 Ontology-based applications for data mining

Ontologies are applied in various fields such as artificial intelligence, the Semantic Web, software engineering, and information architecture. Recently, data mining research has paid more interest on using ontologies to improve the efficiency of data mining. The two main purposes of using ontologies in data mining are to represent domain knowledge and assist data miners to select appropriate DM process, algorithm or software. These two main reasons are interdependent; they interact with each other to facilitate a better understanding of data mining knowledge. Usually, the ontologies are used as the fundamental background support of the data mining project, they provide the knowledge background, the reasoning mechanism, and the advice based on the domain knowledge represented in them.

Integrating ontologies into data mining is a new research direction with great potential. Some data mining projects are leaders in this direction: they have already implemented some data mining related ontologies in their projects, and the results seem very encouraging. However, as a new research domain, how to properly integrate ontologies and data mining still remains a challenge; it needs more research. The rest of this section discusses some existing ontology-based applications in the data mining domain. The discussion is organized into two sub-sections: knowledge representation and assistance of a better selection.

3.4.1 Ontologies for knowledge representation

Knowledge representation is the most essential and important reason to introduce ontologies into data mining. Usually, a data-mining project, especially a data mining assistance project, requires well-defined and organized domain knowledge as its background engine to support the different functions of the project. According to their natural characteristics, ontologies can fulfill this kind of requirement.

One of the data mining ontologies is the DAMON ontology [35, 36], which is apparently the most complete DM ontology so far. This ontology distinguishes some basic terms as task, method, algorithm, software, suite, data source, and human interaction. These terms are implemented in ontology as the main concepts in first level of concept hierarchy, each concept are further divided into sub concepts to complete the concept hierarchy. For example, the concept "method" is divided into classification method, clustering method, deviation detection method, link analysis method, regression method, summarization method, and visualization method. The internal structure and the relationship of all the concepts in the concept hierarchy are defined by means of properties. Some axioms are also applied in the ontology; these axioms can provide the constraints on the properties and the facts about the relations among objects. By navigating the entire ontology, users can find the specified methods, algorithms, and software for a given task such as classification, association, etc.

Another DM ontology worth mentioning is the ontology in the Intelligent Discovery Assistants (IDAs) project [37]. This ontology groups the DM operators into three major groups: preprocessing, induction, and post-processing. Each of these groups is further sub-divided. The preprocessing group is subdivided into categorical attribute transformations, continuous attribute transformations, record sampling, and selecting features. The induction algorithm group is subdivided into classifiers, class probability estimators, and regressors. The post-processing group is subdivided into pruning, thresholding, and logical model transformations. The leaves of the concept hierarchy are the actual operators.

3.4.2 Ontologies for a better selection

The Knowledge Discovery process is one of the central notions of the field of Knowledge Discovery and Data Mining (KDD). Ontologies can be applied in different phases of the KDD process. With the domain knowledge modeled in them, ontologies can cooperate with other components to produce and provide useful advice to furthermore facilitate selecting the appropriate sources.

The IDAs project [37] focuses on helping users to choose a valid and appropriate data mining process. It defines the data mining process as a subset of the KDD process, which includes three stages: data preprocessing, algorithm induction, and model post-processing. It assumes that there are many possible choices for each stage, and only some combinations are valid and useful for a given data mining task. Taking into account this consideration, this project aims at providing users with (1) enumerations of valid DM processes, and (2) rankings of these valid processes by different criteria, to ease the choice of DM processes to execute. Its ontology is used to compose possible and valid processes in both the DM-Process planning stage and the heuristic ranking stage.

Ontologies can be also integrated in the KDD preprocessing. MiningMart [38] is an ontology-supported KDD preprocessing tool that models the data in two levels allowing data abstraction to ease the KDD process. In this project, data are represented on two levels: the logic level and the conceptual level. The logic level describes the database schema of tables, attributes and links, allowing a consistent access to the information. The conceptual level, also called the ontology level, is on the top of the logic level. Depending on a domain ontology, this level uses concepts with features and relationships to model data. Some mapping mechanisms are also provided to switch between these two levels. The ontology is used to describe conceptual domain knowledge and the data abstraction. The advantage is that all the data processing can be captured in an ontology, which gives a better understanding and reuse of the data. Another ontology-based project proposed by Phillips *et al.* [39] works on the feature construction of the KDD preprocessing. They believe that the composition of the useful constructed attributes depends on the semantic relationships among the attributes that usually do not exist in databases. The ontology is designed to hold this meta-knowledge. The ontology is used with a program (attribute annotations) to suggest and generate new attributes based on the predefined rules.

Modeling is the most important part of the KDD process that gains more attention in data mining assistance projects. How to help users to choose a right algorithm, model, or software is the main problem to be solved. Keith Rennolls [40] suggests using an ontology as part of the framework to classify DM models and their relations to make a better choice of a model. The concepts represented in the ontology are grouped firstly into supervised learning models and unsupervised learning models, and can be divided even further.

The DAMON ontology is designed for the knowledge grid [41]. It is used to suggest to the users the appropriate software on the basis of the user's requirements or needs. The structure of this ontology makes it clear to represent the features of the available data mining software, and to classify their main components. It thus offers to a data miner a reference model for the different kinds of DM tasks, methodologies and software available and the useful suggestions of software selection. Besides, this ontology can also simplify the development of distributed knowledge discovery applications on the Grid.

Chapter 4 : DEVELOPMENT OF A NEW DATA MINING ONTOLOGY

As ontology will play an important role in our data mining assistant system, the development of the DM ontology becomes an essential piece of work that influences the success of our system. The creation of an ontology is a complicated endeavor that involves several steps, from specifying the knowledge scope to implementation. This chapter presents the strategy, design and implementation of the DM ontology. It is organized as follows: Section 4.1 gives an overview of the architecture of our intelligent data mining assistant system, the roles of the DM ontology and the relationships between the DM ontology and other system components. The design procedure of knowledge representation is discussed in section 4.2, and section 4.3 explains how to implement the knowledge into the DM ontology. Section 4.4 compares the DM ontology with two other data mining related ontologies and gives some concluding remarks.

4.1 DM ontology in an intelligent data mining assistant system

4.1.1 Overview of the new system

In order to empower (novice) data miners throughout various data mining activities, our data mining team has built a new intelligent data mining assistant system. This hybrid system is mainly based on a DM ontology, a case-based reasoning system and a data warehouse. The new system is capable of providing support for the entire data mining process, from data preparation to result interpretation. More specifically, it can provide not only the general data mining knowledge and explanation but also the practical experience and recommendations for executing particular data mining tasks. Moreover, the new system also presents a synergistic methodology for leveraging the acquisition and representations of data mining knowledge. This system will eventually facilitate the integration of data mining activities into the decision-making process, thus help decision makers to make better choices—this is the long-term goal.

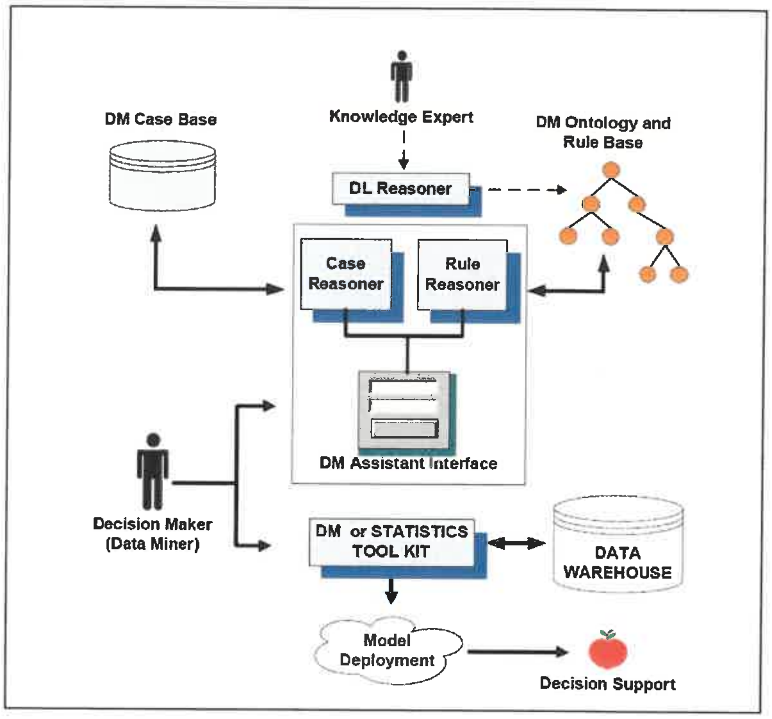


Figure 4.1 Intelligent DM assistant system architecture

As illustrated in Figure 4.1 above [8], our hybrid DM assistant system consists of seven components; a DM Case Base, a DM Ontology and rule base, a Case Reasoner, Rule Reasoner, a DL (Description Logic) Reasoner, a DM Assistant Interface, and a data warehouse. The DM ontology and CBR subsystems have well defined knowledge representation roles. The *DM Ontology and Rule Base* defines high-level concepts (i.e. tasks, activity types, algorithms, etc.) and case adaptation knowledge, while the *DM Case Base* holds detailed case information (i.e. data quality verification, data preparation steps, model parameters, etc.). Operation of the intelligent DM assistant is initiated by the user's query specifying a DM problem (i.e. problem characteristics). Subsequently, the *Case Reasoner* provides a subset of similar previously resolved DM cases to the user. Once a user has chosen a basic case, the adaptation cycle is carried forward (assisted) by the inference capabilities provided by the *Rule Reasoner* and the detailed domain knowledge within our *DM ontology*. The DM ontology (by formally capturing concepts, relationships, constraints and rules) is capable of complementing the CBR system and addressing this need for more detailed domain knowledge. For the moment, the *DL Reasoner* is strictly used for the purposes of ensuring the consistent evolution of the *DM ontology*.

4.1.2 Roles of the DM ontology

The DM ontology is an important component in our assistant system. It acts as the complementary knowledge source in the system, and determines the intelligence level of the system.

DM knowledge representation

The essential usage of the DM ontology is to represent data mining domain knowledge. In our assistant system, all the required conceptual data mining knowledge is defined in the DM ontology. The DM ontology captures and classifies the knowledge into four sections; each section presents a different type of knowledge. The concepts represented in the DM ontology are organized as a knowledge hierarchy, with which we can easily identify the different level of concepts, the class constraints and the relationships between them. With the ability of defining the semantics of the concepts, the DM ontology also provides a data mining vocabulary as well as a computerized specification of the meaning of terms used in the vocabulary.

Integration of DM ontology into the DM assistant system

The DM ontology will be integrated in to the assistant system, especially with the CBR system and data warehouse.

Both the CBR and the DM ontology subsystems have well defined knowledge representation roles, they can be related to each other. More specifically, the DM ontology holds high-level concepts (i.e., activity types, algorithms, etc.) while the CBR holds detailed case information (i.e., data preparation steps, model parameters, etc.). On the other hand, the CBR maps problems to solutions, while the DM ontology can semi-automatically learn concept dependencies and properties strictly within either the problem or solution spaces. When integrating DM ontology with CBR cases, the DM ontology can not only cooperate with CBR cases but also provide some fundamental supports to them. On one hand, as the DM ontology defines the conceptual DM knowledge; it can offer a solid background knowledge source to CBR cases. The DM ontology is also capable of providing recommendations and heuristic at various DM phases through CBR cases. On the other hand, the DM ontology represents the important features and semantics of cases; this will give users another way to understand, classify, navigate, and choose cases for a given DM task, especially for the task with a set of constraints about the attribute type, algorithm selection, model selection, result evaluation and so on. The newly returned cases from CBR paradigm can be added to the DM ontology to keep the case representation complete and updated.

The integration of the DM ontology and data warehouse is realized through the representation of the schema and its semantic. This data annotation can largely help users identify the structure of the data mart and its tables, the relationships between data marts, data marts and tables, and fact tables and dimension tables, thus helps users understand the whole structure of the data warehouse. The existing schema in the ontology can also guide users to construct new data in the data warehouse.

4.2 The design of the data mining knowledge representation

The first step of developing an ontology is to identify the domain and scope of the ontology. As a knowledge source of the DM assistant system, the conceptual data mining domain knowledge is the most significant part of knowledge that must be modeled in the DM ontology. This part is the most valuable and useful knowledge representation that can be reused and shared with other applications. However, the DM ontology needs to be integrated with other components. This requires other types of knowledge representation: the data annotation, the CBR cases and the data mining process. This section describes how these two types of knowledge are analyzed and how the knowledge representation structures are designed. Four sections, CRISP-DM, data sources, CRB cases, and data mining techniques are discussed respectively.

4.2.1 CRISP-DM

CRISP-DM is a general data mining methodology and process model which covers all the data mining phases that can be applied in various kinds of data mining activities. Considering this advantage, CRISP-DM was adopted in our assistant system as a basic infrastructure to guide the construction of CBR cases and DM ontology.

The CRISP-DM methodology is described in terms of a hierarchical process model, which consists a series of tasks described at four different levels, as shown in Figure 4.2. At the top level, the data mining process is organized into a number of phases; each phase consists of several second-level tasks. Each task contains some more detailed activities and each activity may have some outputs.

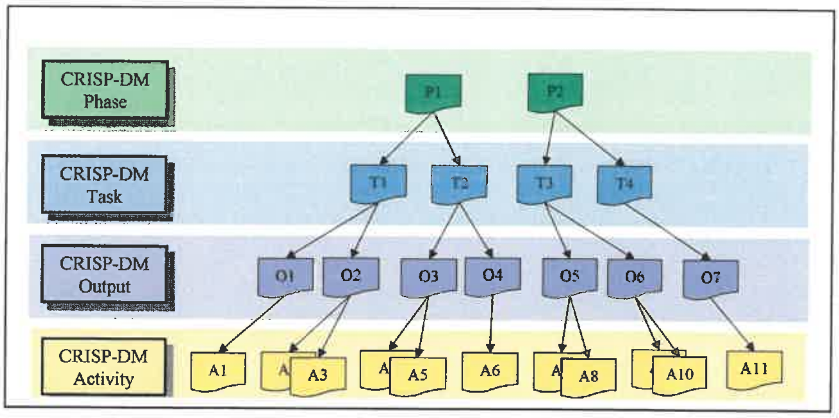


Figure 4.2 The hierarchy of CRISP-DM

The DM ontology represents the CRISP-DM as a section of knowledge source. This section of ontology will focus on the CRISP-DM knowledge itself, namely, the methodology and the process. It will provide a guide to the whole process and an explanation of the concepts involved in the methodology. Trying to keep it in agreement with the characteristics of ontology knowledge modeling, we follow the hierarchical representation of CRISP-DM in general but rearrange the important concepts. As shown in Figure 4.3, four main concepts are distinguished: phase, task, activity, and output. *Phase* is the high-level term for part of the process model, (e.g. business understanding, data understanding, data preparation, modeling, etc.); each phase consists of several related tasks. *Task* is the part of a phase which includes a series of activities to produce one or more outputs. *Activity* is the part of a task describing actions to perform a task, and the *Output* is the tangible result of performing a task. The relationships between the concepts are also presented in Figure 4.3; all the relationships are two ways. For example, a phase (P1) may include several tasks (T1, T2, T3,…), so these tasks are only included in this particular phase P1 (not in P2 or P3, or any other phase). Therefore, in the DM ontology, whenever a Phase is defined, its related task must be included. In the same manner, whenever a task is defined, its belonging phase must also be specified. In this way, the relationships between Phase and Task can be described properly in the DM ontology.

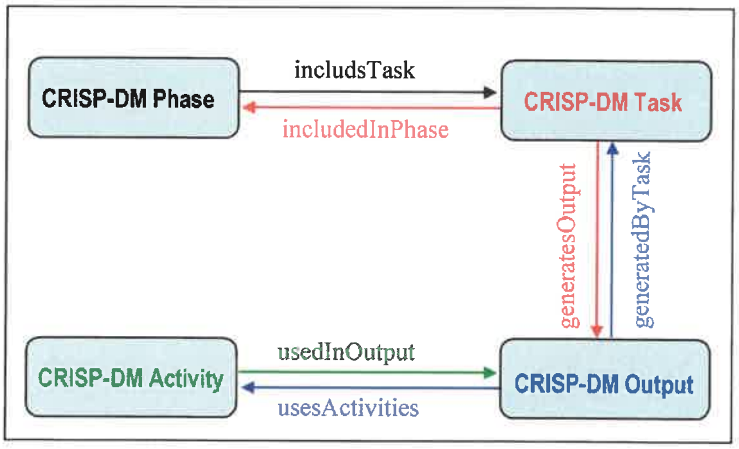


Figure 4.3 The knowledge representation of CRISP-DM

4.2.2 Data source

The data warehouse integrated in our data mining assistant system mainly represents the data of UQTR and ICOPE (a source of data investigating the information of undergraduate student in University of Quebec, including academic and socio-demographic characteristics, living conditions, preparation of studies, intentions, motivations, interest and knowledge of the program), mainly about student-related information. The data warehouse contains several data marts; each data mart represents the information of one sector of the university, such as student, application, registration, and so on. The data mart may consist of several tables, including one fact table and some dimension tables. There may be some overlap of data marts; one table may belong to two or more data marts.

One section of the DM ontology will represent the data source stored in the data warehouse. This data annotation will capture and model the metadata (data dictionary) of the data warehouse. More significantly, the data source section of the DM ontology provides the meaningful semantics of the concepts defined in the data warehouse system, table, attribute, etc. This includes two aspects. One is how the concepts map to the real world. For example, the attribute CD_PGM may refer to the program code of the university's programs of studies. The other is about the schema itself: what are its structure and its meaning. Furthermore, the data source section of the DM ontology is capable of linking tables to the CBR cases in which the tables are involved. This capability gives a better integration of data source and CBR cases.

The data source section of the DM ontology takes a hierarchical structure to represent the different concepts of the data warehouse. Figure 4.4 gives an overview of the structure. The first level concept *System* defines the entities such as student, application, registration and so on, corresponding to the different data marts. A system may contain several tables, which is the second level concept. The concept *Table* describes the characteristics (schema) of each table; all the tables will be classified according to the system to which it belongs. *Table* is further divided into fact table and dimension table. The concept *Attribute* describes the characteristics of each attribute such as the attribute name, type and constraint, etc.

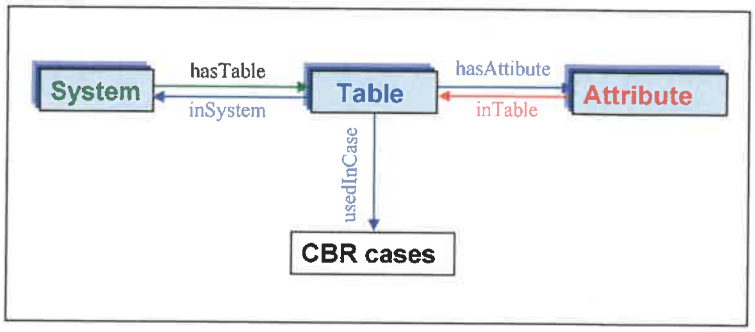


Figure 4.4 The knowledge representation of Data Source

4.2.3 CBR cases

CBR cases are created to capture and present the useful practical experience of executing various data mining activities, thus helping data miners better understand the data mining process. The design is based on meta-learning and CRISP-DM, and implemented in the case-based reasoning paradigm. The CBR system maps DM problems to solution space through CBR cases. Each CBR case has 53 features, while 15 of them are indexes. The features are divided into three parts: problem description, solution space, and activity output. As the knowledge represented in CBR cases is in a very detailed level, the DM ontology will capture only some main characteristics of cases that are important to represent the case structure and usage.

The CBR cases section of the DM ontology will define the semantics of each selected feature to represent the cases. This representation can give users a global outlook of CBR cases and their functions. The DM ontology will also link the cases to other relevant concepts that are also represented in the ontology. More specifically, a case can have relations with one or several particular tables and attributes, which connect the

CBR cases and Data source together. Moreover, a case may choose a special program as its solution; this illustrates how cases can be linked with DM techniques. The integration of CBR cases and the DM ontology will largely facilitate the representation, sharing and reuse of DM knowledge.

4.2.4 DM techniques

4.2.4.1 Some differences in definition of data mining knowledge

With the increasing growth of machine learning techniques and statistical analysis, data mining knowledge has tremendously evolved these recent years. Data mining knowledge is not a simple set of algorithms; it has become sophisticated, it covers much more different domains than ever, and the developed algorithms become even more complicated. The resources of data mining knowledge presentation are also very abundant. A sea of data mining books, web sites, lectures, and courses are available nowadays. Usually, they all describe the general, common data mining knowledge, some resources dig the mathematical and statistical methods a little deeper, and others extend the data mining knowledge with other domains. There are lots of well organized and finely presented data mining books and lectures [42, 43, 44, 45], from any of these resources we can get a general idea and a good understanding of data mining. However, since every resource has its own focus and ideas, there exist some differences in knowledge definition and categories among them. These differences in terms and categories make things complicated and sometimes are genuinely confusing.

a. Different categorizations

Generally, data mining concepts can be classified as classification, association, clustering and so on. These main categories are adopted by almost all the data mining books and relevant literature. But how about the sub-categories and others concepts besides the main categories? Some books put the regression separate from classification, while others make regression a subsection of classification. When implementing the knowledge into the ontology, the different categorizations pose a major problem of concept classification. Since the DM ontology strictly demands an accurate concept hierarchy, such differences must be resolved and a reasonable, unified categorization of data mining knowledge must be specified.

b. Mix-up of terms

Many terms are involved in data mining knowledge representation. The mixed use of data mining terms within different books really confuses some readers, especially non-expert data miners. This problem can be divided into the following two types:

(1) Several terms refer to the same concept. For example, the term *data object* is a collection of data set; other terms of data object are record, point, vector, pattern, event, sample, observation, or entity. Another example is the term *attribute*; the synonyms of attribute are variable, characteristic, field, feature, or dimension. Different terms are used in different books, or in different categories of data mining concepts.

(2) Disagreement of terms. Model, algorithm, and method are the most common terms, but their definitions are unclear when checking their meanings in different books. Model may refer to the output in one book, which is exactly the meaning of algorithm in another material! Method can be used in both generalization and specification. This misuse of terms decreases the clarity of data mining concepts and becomes more serious when trying to classify data mining knowledge and unify data mining vocabularies.

4.2.4.2 Data mining knowledge representation

The DM techniques section of the DM ontology aims to model systematically the data mining domain knowledge. In order to represent DM knowledge, we must firstly resolve the problems discussed above: the involved terms must be well selected, their meanings must be clarified, and the categorization must be properly structured. This section of the DM ontology will be used as a reference of data mining knowledge.

The structure of knowledge representation of DM techniques is designed based on CRISP-DM. The CRISP-DM section represents the methodology and process of data mining while the DM Techniques section represents the actual applicable knowledge for each phase of the process. More specifically, the CRISP-DM section models the high level directions of "what to do" for a data mining task while the DM Techniques section models the required domain knowledge of "how to do" for the task. It can include various data sampling techniques, different data mining algorithms, available programs, and so on.

Trying to parallel the different phases of the data mining process, the knowledge representation of the DM Technique section is divided into three sub-sections: data understanding, data preparation and modeling, corresponding to the same phases described in CRISP-DM. The other three phases: business understanding, evaluation and deployment are not presented since these sections are really business dependent and cannot be covered by data mining domain knowledge. The data understanding subsection of the DM technique section describes the characteristics of the data. For example, the various attribute types, the general errors found in data, how the data is collected and exploited, etc. The data preparation subsection presents the techniques dealing with how the data is prepared for a data-mining task, such as data cleaning, sampling, data construction, data integration, and so on. The modeling subsection is the most complicated part of the DM techniques section, which represents the kernel of data mining knowledge. This subsection classifies the most important and useful concepts in a concept hierarchy linked with their relationships. It also identifies the most detailed knowledge (instances) for each concept. Figure 4.5 shows the top-level concepts in the modeling subsection.

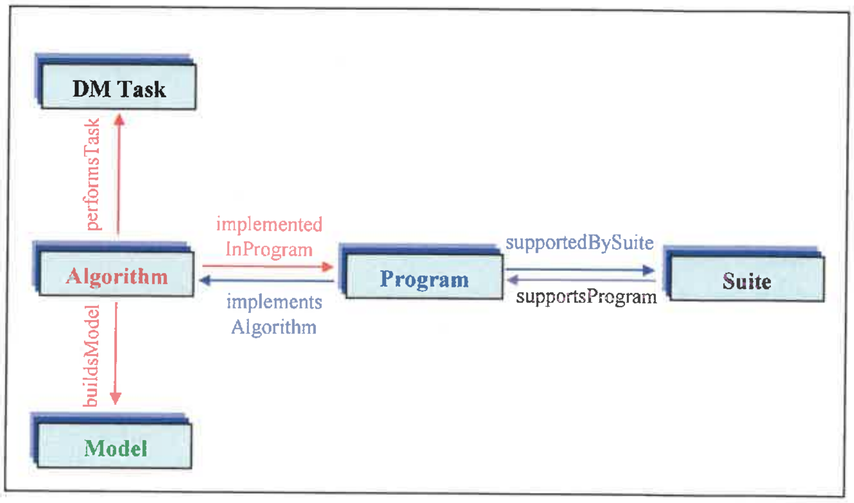


Figure 4.5 The knowledge representation of the DM Techniques

As illustrated in the figure above, the modeling subsection defines five main concepts: task, algorithm, program, model, and suite, which are explained below.

Data mining task: It is a general problem for which data mining is called in to generate a model. The data mining task consists of classification, regression, association and clustering, and so on.

Data mining algorithm: An algorithm accepts structured data and returns a model of the relationships within a data set (if there are any!). It is a mechanism in which a data mining task is performed. The algorithm's performance is measured by its accuracy, training/testing time, training/testing resource requirements, and the model's understandability.

Data mining model: A model is the output of an algorithm. It can be a decision tree, a linear equation, a set of rules, etc., and it can be descriptive or predictive. A descriptive model helps understanding of an underlying process or behavior. A predictive model makes it possible to predict an unseen or unmeasured value from other known values. As models are often presented with statistical terms, the evaluation and interpretation of the results is often very difficult for data miners.

Data mining program: A program is the implementation of an algorithm. It can be an independent software, but most programs are packaged into data mining suites.

Data mining suite: A suite is a set of programs usually packaged in an integrated software environment. Each program may perform different tasks and may use different algorithms to achieve the goal.

The concepts described above are all top-level concepts; they can be further divided to make a more detailed concept hierarchy for knowledge representations. As classification, association, regression, and clustering are the four most common and most applied data mining tasks, the DM techniques section of the DM ontology will mainly focus on these four categories. The concept hierarchies of data mining task, algorithm, model and program are illustrated respectively in Figure 4.6, Figure 4.7, Figure 4.8, and Figure 4.9. To simplify the layout of the concept hierarchies and to give a better understanding, only some specializations of classification are shown here.

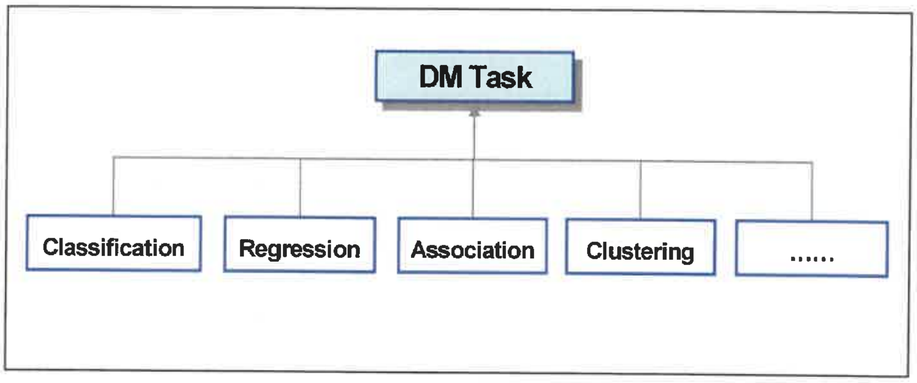


Figure 4.6 The concept hierarchy of *DM Task*

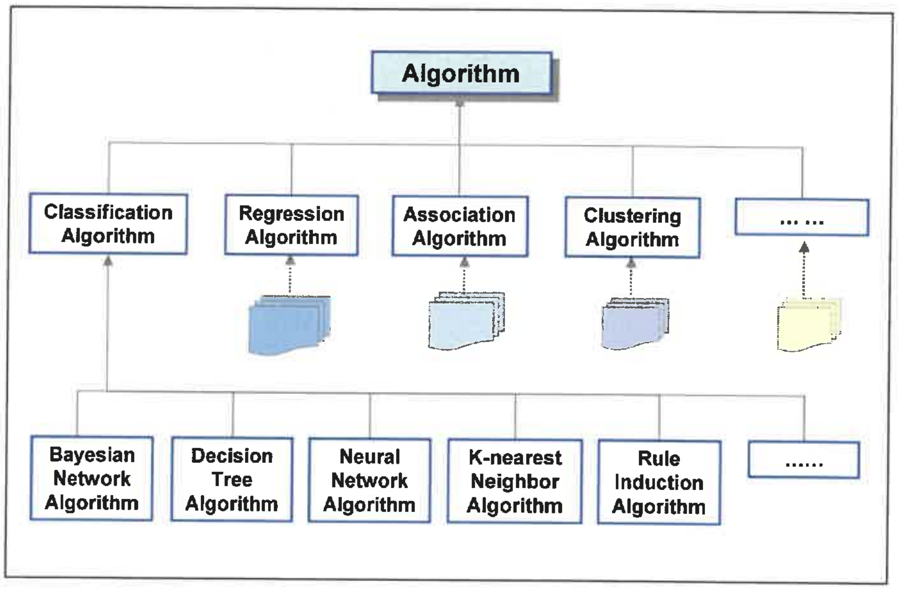


Figure 4.7 The concept hierarchy of *Algorithm*

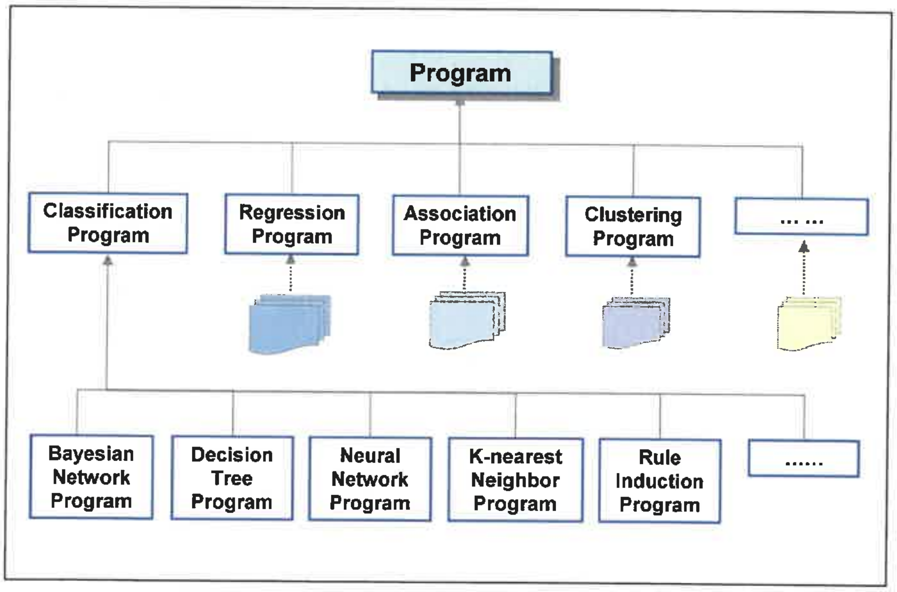


Figure 4.8 The concept hierarchy of *Program*

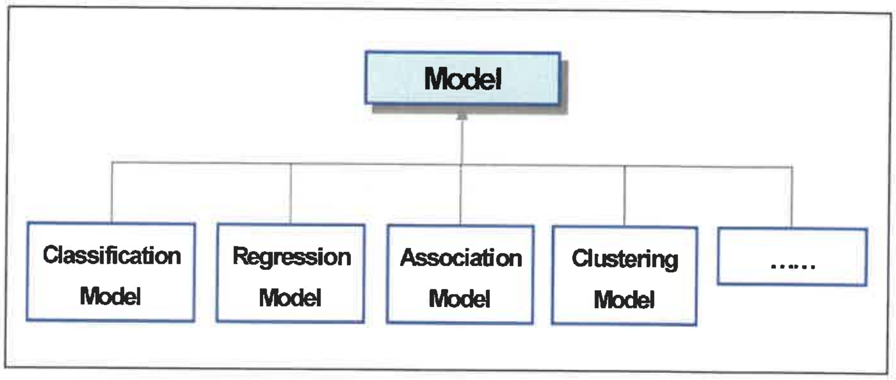


Figure 4.9 The concept hierarchy of *Model*

Let us also remark that Figure 4.5 above also presents the relationships among the concepts. These relationships connect interrelated concepts together so that all the concepts can be modeled in a taxonomy. This makes the knowledge representation more meaningful regarding to its semantics. The instances of the concepts can also be described more precisely. For example, C4.5 is a decision tree algorithm, which is in the classification algorithm category, performing the classification task (*performsTask*), building a decision tree model (*buildsModel*), and is implemented in the j48 program (*implementedInProgram*).

4.3 Design and implementation of the DM ontology

Now that the scope and structure of knowledge to be represented in the ontology are well defined, the next step is to build the DM ontology and implement the knowledge representation into the ontology. The DM ontology describes the data mining concepts and relationships required in our data mining assistant system, providing a complete set of unified DM terms and their specified meanings.

The DM ontology is built upon the OWL DL language and developed with the ontology editor *Protégé* [10]. Protégé is a free, open-source platform developed at Stanford University which provides a suite of tools to construct domain models and knowledge-based applications with ontologies. It supports the creation, visualization, and manipulation of ontologies in various representation formats. The Protégé-OWL editor is an extension of Protégé that supports the OWL language. OWL is the most recent development in standard ontology languages, endorsed by the World Wide Web Consortium (W3C) to promote the Semantic Web vision. The Protégé-OWL editor enables users to edit and visualize classes, properties and instances of an ontology, define logical class characteristics as OWL expressions, and execute reasoners such as description logic classifiers.

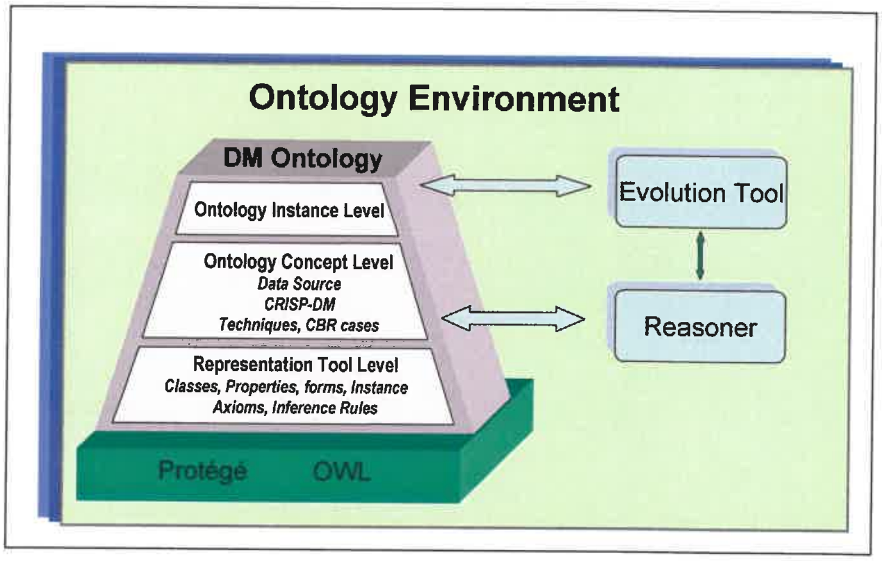


Figure 4.10 The DM ontology environment

Figure 4.10 illustrates the ontology environment in the assistant system. A pyramid structure is adapted for the structure of the DM ontology. The basic level is the knowledge representation tool level, which provides the models used to represent knowledge such as class, property, instance, and axioms. The second level is the ontology concept level, which defines all the data mining concepts including CRISP-DM, data source, CBR cases and DM techniques. The ontology instance level is the finest level of ontology, which represents the concrete knowledge of data mining. The DM ontology is also integrated with a reasoner and an ontology evolution tool. The reasoner is used to check the ontology consistency. The ontology evolution tool is a new Protégé plug-in developed by the author to support OWL ontology updating and maintenance. This tool will be further discussed in Chapter 5.

4.3.1 Class definition and hierarchies

Class is one of the most important components of an OWL ontology. Classes group the resources with similar characteristics. They are typically used to represent the different concepts and their hierarchies involved in a knowledge base. In OWL, classes are built up from descriptions such as properties and constraints that specify the conditions that must be satisfied by their own instances. OWL classes may also have several instances in them to represent the particular individual concepts sharing the same structure.

In our DM ontology, classes represent the theoretical concepts of DM knowledge and their internal structure. The design and categorization of classes follow the classification of knowledge discussed in Section 4.2. The highest level of class hierarchy maps the four main sections of knowledge: CRISP-DM, CBR cases, data source and data mining techniques, as shown in Figure 4.11. The class CRISP-DM, data source and data mining techniques are also divided into a series of subclasses to model the complete structure of data mining knowledge. A class that contains subclasses is annotated with a black/grey triangle at the left of the class name.

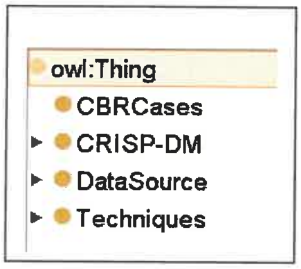


Figure 4.11 The highest level of class hierarchy in DM ontology

The CRISP-DM class is further divided into CRISP-DM phase, CRISP-DM task, CRISP-DM activity, and CRISP-DM output subclasses, corresponding to the CRISP-DM knowledge representation form. As presented in Figure 4.12, these four subclasses are further expanded into more detailed levels.

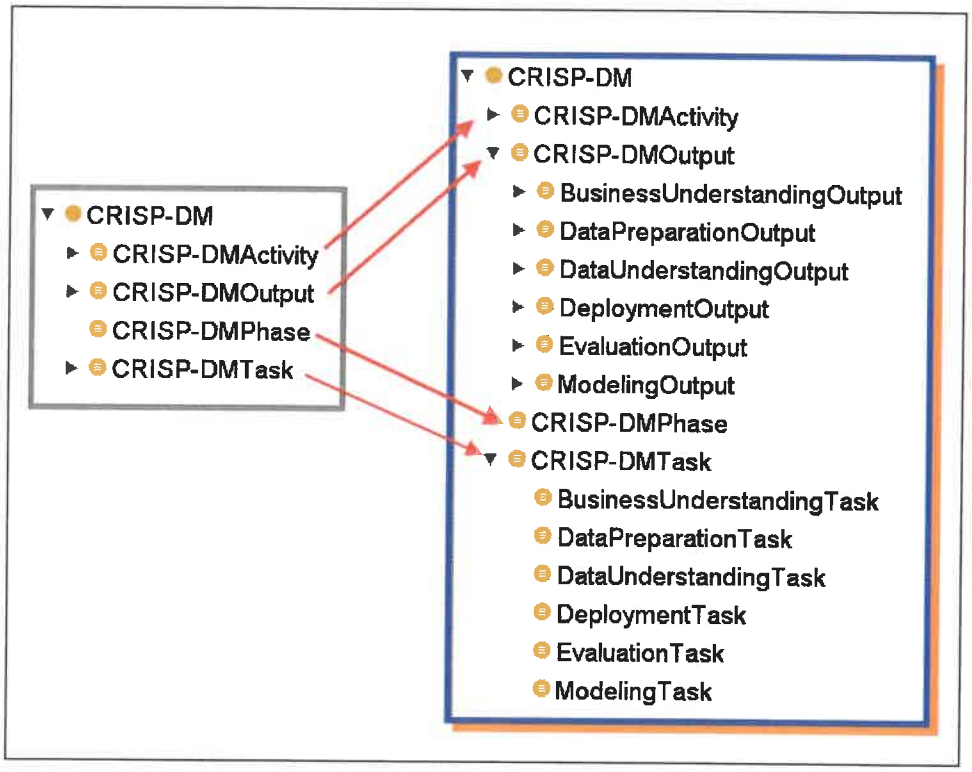


Figure 4.12 The CRISP-DM classes

Similar to the knowledge representation of data source section, three subclasses are created in class *DataSource*: *System*, *Attribute*, and *Table*. Figure 4.13 shows the class hierarchy of data source section.

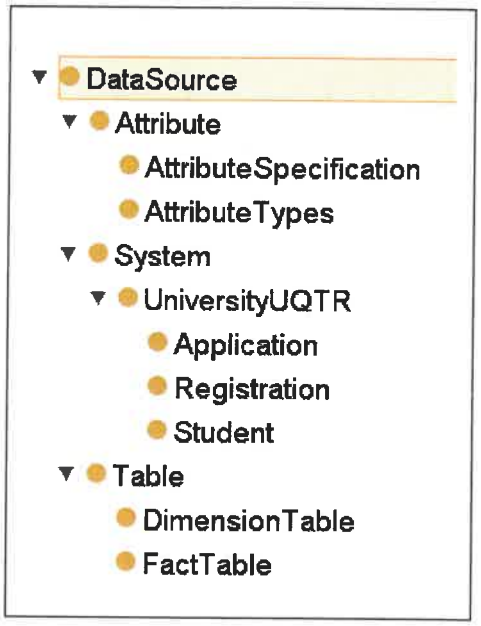


Figure 4.13 The class hierarchy of *DataSource*

Techniques is the largest, essential class in the DM ontology, which represents the systematic part of data mining domain knowledge. It is further divided into data preparation, data understanding, and modeling subclasses. The relationships between the CRISP-DM methodology and the DM ontology are illustrated in Figure 4.14. The class CRISP-DM models the high-level knowledge about the data mining methodology and process, while the class *Techniques* represents the detailed applicable knowledge of a special data mining task. The subclasses of *Techniques* represent the concrete base-level knowledge of each corresponding phase.

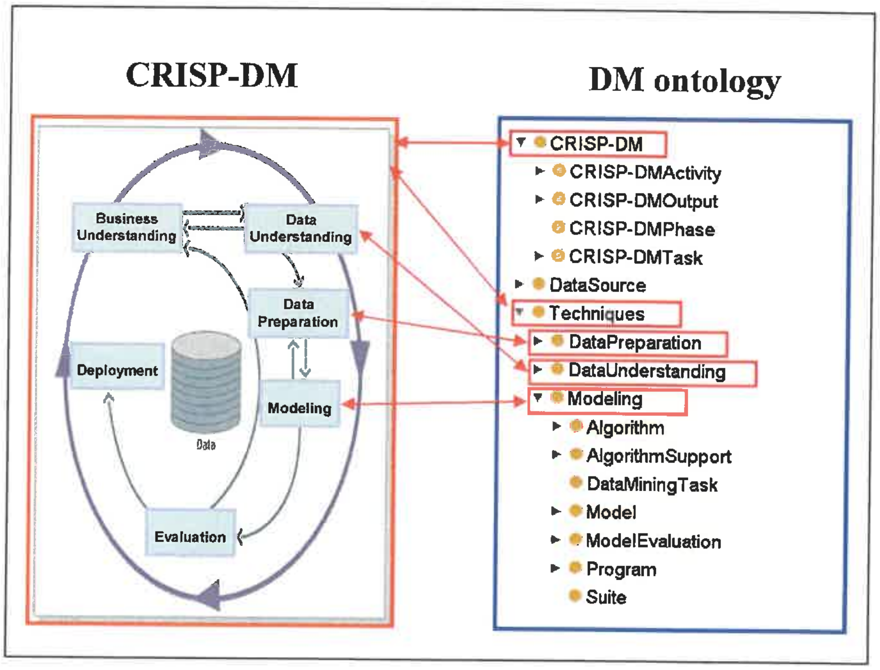


Figure 4.14 Relationships between CRISP-DM and DM ontology

Figure 4.15 shows the subclasses of *Modeling*. The class Modeling consists of two types of subclasses. One is the main class corresponding to the concepts described in Section 4.2.4.2; this type includes the classes *DataMiningTask, Algorithm, Model, Program*, and *Suite*. The other one is the utility classes, which includes the class *AlgorithmSupport* and *ModelEvaluation*. This type of class depends on the main classes, it provides some additional knowledge source to the main classes, thus making the knowledge represented in the main classes more complete and semantically meaningful. For instance, the class *AlgorithmSupport* can provide the definition and explanation of the pruning technique and the splitting criteria used to describe a decision tree algorithm. When an instance of the *DecisionTreeAlgorithm* class, say C4.5, needs some knowledge source to fill its properties "*usesPrunningTechnique*" and "*usesSplittingCriteria*", the Protégé ontology editor can select the knowledge represented in the utility class *AlgorithmSupport*.

The class *ModelEvaluation* defines the evaluation methods and metrics of the data mining results. Most often, the data mining results are presented with statistics, which makes them difficult to be understood by non-expert data miners, and only few (if any) DM books and lectures try to address the problem of explaining the interpretation of statistics in data mining. In order to provide better support for decision makers, our DM ontology specifically defines a class to represent the statistical knowledge. The class *ModelEvaluation* presents the definition and description of some commonly used statistical terms as well as indications to facilitate results interpretation. For example, the correlation coefficient is a widely used metric to evaluate the performance of numeric prediction by comparing predicted values on the test records $p_1, p_2, \ldots p_n$, and the actual values $a_1, a_2, \ldots a_n$. It ranges from 1 for perfectly correlated results, to 0 when there is no correlation, to -1 when the results are perfectly correlated negatively. The larger the correlation coefficient value is, the better the performance of the classification model will be. This metric is mainly used in evaluation of the linear regression algorithms and models.

As a complicated class hierarchy, the subclasses of Modeling also contain subclasses, making the hierarchy several levels deep. The expansion of the classes *Algorithm*, *Model* and *Program* are respectively presented in Figure 4.16, Figure 4.17, and Figure 4.18.

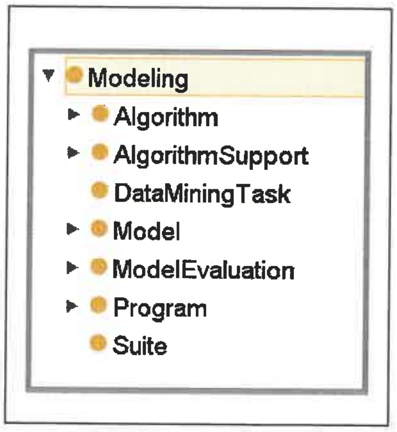


Figure 4.15 The class *Modeling* and its subclasses

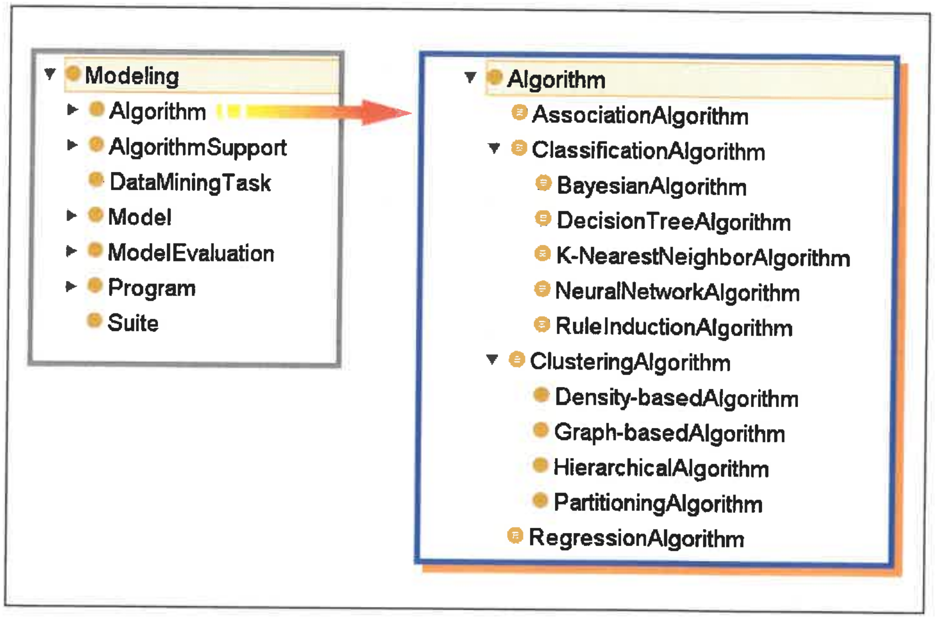


Figure 4.16 The class *Algorithm* and its subclasses

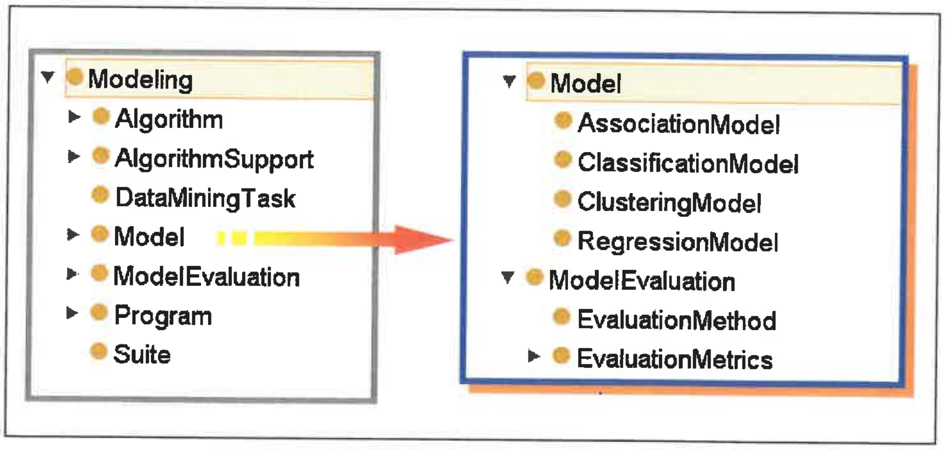


Figure 4.17 The class *Model* and its subclasses

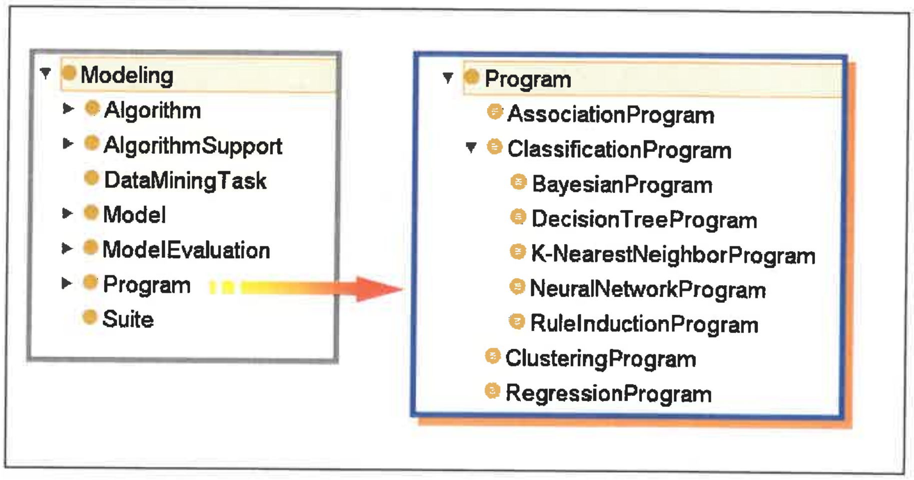


Figure 4.18 The class *Program* and its subclasses

4.3.2 Associated properties and class conditions

The classes alone will not provide enough information to represent domain knowledge. Once the classes have been defined, we must describe the internal structure of the concepts. OWL properties provide a mechanism to link the related classes and their instances as they describe the binary relations between the instances of two different classes or between the instances and data types. The appropriate definition of properties has two advantages. One is that they can make the elements of the ontology (e.g., classes, instances) more connected as a network; this will ease the information querying especially when several classes are involved in a single query. The other benefit is that they can define a structure for describing the characteristics of the classes and their instances. The more complete the structure is, the more meaningful and significant the instances are. The properties associated with the sections CRISP-DM, CBRCases, DataSource, and Techniques are given in Table 4.1, Table 4.2 , Table 4.3, and Table 4.4 respectively.

Table 4.1 Properties of *CRISP-DM*

Class	Property name	Property type
CRISP-DMPhase	description	datatype
	name	datatype
	includesTask	object
CRISP-DMTask	name	datatype
	description	datatype
	includedInPhase	object
	generatesOutput	object
CRISP-DMOutput	name	datatype
	description	datatype
	generatedByTask	object
	usesActivities	object
CRISP-DMActivity	description	datatype
	usedInOutput	object

Table 4.2 Properties of *CBRCases*

Class	Property name	Property type
CBRCases	name	datatype
	pbmBusinessArea	object
	pbmDataAttributes	object
	pbmDataFormat	datatype
	pbmDmActivity	object
	pbmTargetDataType	object
	pbmUsesTool	object
	sltChosenTable	object
	sltChosenAttibute	object
	sltSelectedProgram	object
	outAchievedCriteria	datatype
	outGUM	datatype

Table 4.3 Properties of *DataSource*

Class	Property name	Property type
System	description	datatype
	name	datatype
	hasDimensionTable	object
	hasFactTable	object
Table	name	datatype
	description	datatype
	inSystem	object
	hasAttribute	object
	usedInCases	object
	primaryKey	object
	foreignKey	object
Attribute	name	datatype
	description	datatype
	inTable	object
	accessService	datatype
	attributeConstraint	datatype
	attributeLabel	datatype
	attributeType	datatype

Table 4.4 Properties of *Techniques*

Class	Property name	Property type
DataUnderstanding	description	datatype
DataPreparation	description	datatype
	description	datatype
	buildsModel	object
	handlesAttributesType	object

Modeling	implementedInProgram	object
	performsTask	object
	Pseudo-code	datatype
	usesPrunningTechnique	object
	usesSplittingCriteria	object
	usesConjunctChoosingCriteria	object
	usesGrowingStrategy	object
	usesStoppingCriteria	object
	usesDensityDefinitionApproach	object
	usesClusterProximity	object
	usesPointProximity	object
	evaluatesProgram	object
	author	datatype
	programmingLanguage	datatype
	version	datatype
	implementsAlgorithm	object
	supportedBySuite	object
	usesEvaluationMetrics	object
	supportsProgram	object
	targetAttributeType	object

Relational property and descriptive property

An OWL property can be relational or descriptive when considering its role in an ontology. A relational property defines the relationships between instances of two classes while a descriptive property describes the characteristics of the classes and their instances in which the property is applied. Usually, the object properties are used to define the relations, and the data type properties are used to describe the characteristics. Taking the class Algorithm as an example, Table 4.5 lists the descriptive and relational properties and their functions in the Algorithm class.

Table 4.5 The properties associated to the class *Algorithm*

	Property	Property Type	Function
Descriptive	description	Datatype property string	General description of the algorithm
	pseudo-code	Datatype property string	Pseudo-code of the algorithm
Relational	performsTask	Object property	Links Algorithm to DMTask
	handlesAttributeType	Object property	Links Algorithm to AttributeType
	buildsModel	Object property	Links Algorithm to Model
	implementedInProgram	Object property	Links Algorithm to Program

Domain and range

Domain and range are two axioms associated with properties. A domain of a property limits the instances to which the property can be applied. For example, if the domain of the property *performsTask* is defined as the class *Algorithm*, then only the instances of the class *Algorithm* can use the property *performsTask*. If the domain of the property *performsTask* is defined as the union of the class *Algorithm* and *Program*, then *performsTask* can be applied to instances from both *Algorithm* and *Program*. The range of a property limits the instances that the property may have as its value. If the range of the property *performsTask* is set as the class *DMTask*, then only the instance from *DMTask* can be selected as the value of the property. Table 4.6 lists the domain and the range of the properties associated with the class *Algorithm*.

Properties may have a domain and a range specified. Properties link instances from domain to instances from the range. For example, as the domain is *Algorithm* and the range is *DMTask*, the property *performsTask* links the instances belonging to the class *Algorithm* to the instances belonging to class *DMTask*.

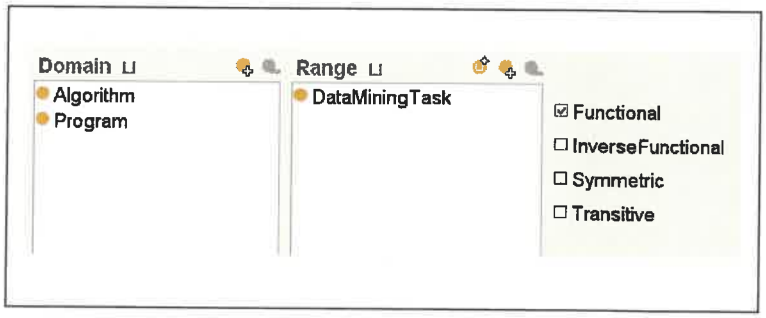
Table 4.6 Domain and range of the properties associated with *Algorithm*

Property	**Property type**	**Domain**	**Range**
description	Datatype property string	Algorithm	string
pseudo-code	Datatype property string	Algorithm	string
performsTask	Object property	Algorithm	DMTask
handlesAttributeType	Object property	Algorithm	AttributeType
buildsModel	Object property	Algorithm	Model
implementedInProgram	Object property	Algorithm	Program

Inverse and functional properties

Properties have a direction from domain to range. In practice, it is useful to define relations in both directions. If a property links instance A to instance B, then its inverse property will link B to A. For example, the inverse property of *implementsAlgorithm* is *implementedInProgram*. If a program j48 implements C4.5, then because of the inverse property we can infer that C4.5 must be an algorithm and C4.5 is implemented in the program j48.

If a property is functional, for a given instance, there must be at most one instance that is related to the instance via the property. In other words, a functional property can only have one unique value for each instance. For example, a particular algorithm can only perform one data-mining task; it cannot perform a classification task and a clustering task at the same time. Therefore, the property *performsTask* should be defined as functional. Figure 4.19 shows the constructs of the property *performsTask.*



Domain ⊔
Algorithm
Program

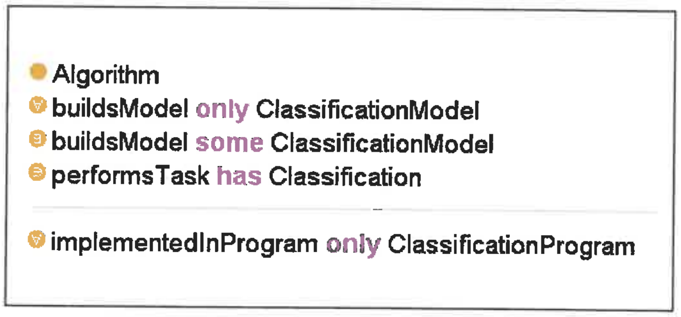
Range ⊔
DataMiningTask

☑ Functional
☐ InverseFunctional
☐ Symmetric
☐ Transitive

Figure 4.19 The functional property *performsTask*

Property restrictions and Class constraints

In OWL, properties are used to create restrictions. As the name may suggest, restrictions are used to restrict the instances that belong to a class. Property restrictions can be regarded as a special kind of class description. Value constraint is one kind of property restrictions that sets constraints on the range of the property when applied to this particular class description. There are three types of value constraints that are broadly applied in ontology constructions: *allValuesFrom*, *someValuesFrom* and *hasValue*. In our DM ontology, the three constraints are applied in many classes to specify the class conditions. Figure 4.20 shows an example using constraints in the *classificationAlgorithm* class, which is a subclass of the *Algorithm* class.



Algorithm
buildsModel only ClassificationModel
buildsModel some ClassificationModel
performsTask has Classification

implementedInProgram only ClassificationProgram

Figure 4.20 The constraints of the class *ClassificationAlgorithm*

The property *buildsModel* has applied both *allValuesFrom* and *someValuesFrom* constraints. The *allValuesFrom* restriction requires that for every instance of the class that has instances of the specified property, the values of the property are all members of the class indicated by the *allValuesFrom* clause. In this case, *allValuedFrom* means that for all the classification algorithms, if they can build models, all the models they build are classification models. However, this constraint does not restrict the number of classification models when filling out the property for an instance, the value of this property can be empty. This is why the property *buildModel* is also restricted with *someValuesFrom* constraint. The *someValuesFrom* requires that there must be at least one model that is a classification model, but there may be some models that are not classification models. When using the two constraints together, it indicates that a classification algorithm must build at least one model, and all the models the algorithm builds are classification models. This double restriction can guarantee that the relation between the classification algorithm and the classification model is particularly specified.

The property *performsTask* is restricted with the *hasValue* constraint. The *hasValue* links a restriction class to a particular instance. In our example, *classification* is an instance of the class *DMTask*, and all the classification algorithms must perform the *classification* task. Hence, the filler of the property *performsTask* has only one value, which is the instance *classification.*

Inherited properties

When class A is defined as a subclass of class B, class A will inherit the properties and constraints from class B. In the meantime, class A can also possess its own properties and more specified constraints. This mechanism of generalization and specification keeps the class hierarchies more organized and can avoid double definitions of the class, subclass and properties.

In our DM ontology, the class *Algorithm* has a subclass *ClassificationAlgorithm*, which in turn has a subclass *DecisionTreeAlgorithm*. The class *DecisionTreeAlgorithm* inherits all the six properties defined within the class *Algorithm* (because there is no added properties to the class *ClassificationAlgorithm*, the class *Algorithm* and *ClassificationAlgorithm* share the same properties). Besides, it also has three properties associated only with the class *DecisionTreeAlgorithm*, as shown in Figure 4.21. Therefore, the class *DecisionTreeAlgorithm* has nine properties.

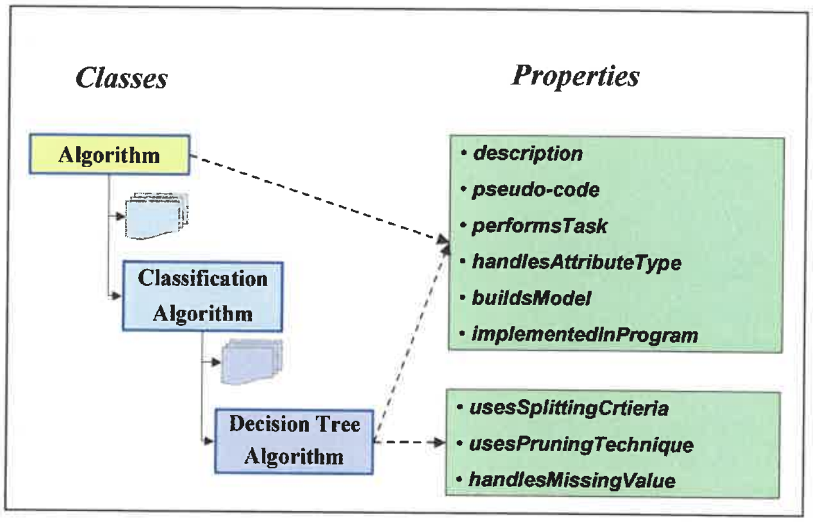


Figure 4.21 Inherited and created properties of the class *DecisionTreeAlgorithm*

4.3.3 Added instances

Individual instances are the most specific concepts modeled in a knowledge base; they are at the lowest level of granularity in the knowledge representation structure [46]. Deciding whether a particular concept is a class in an ontology or an individual instance depends on the potential applications of the ontology. The main goal of our DM ontology is to represent data mining knowledge in a more detailed level so that the users can search the most specific knowledge about a particular concept. An example is the particular algorithm C4.5: the DM ontology should provide its description, its pseudo-code, what data mining task it can perform, what model it can build, and what programs implement it, etc. This level of granularity is the finest level of data mining knowledge representation. The knowledge modeled in this level describes the "real" comprehension of data mining concepts and should be defined as instances.

The ontology without instances has only an empty structure of its classes; it lacks the concrete knowledge and its semantics. Figure 4.22 shows the boundary of classes and instances in the Algorithm concept hierarchy. The class hierarchy is a pattern that defines the category and the structure of the algorithm, while the instances describe the individual algorithms in a particular class. Thus, for all the instances in the same class, they share the common features of the class such as its category, constraints, and related properties; but each instance possesses its own specific characteristics.

Some instances of the class *DecisionTreeAlgorithm* in the DM ontology are presented in Figure 4.23. All the instances of the class share the same properties, but the value of one particular property for each instance can be different. For example, the instance C4.5 with its property fillers is shown in the figure. The value of the property *handlesAttributeType* for instance C4.5 is nominal and numeric, while that for instance CART is numeric.

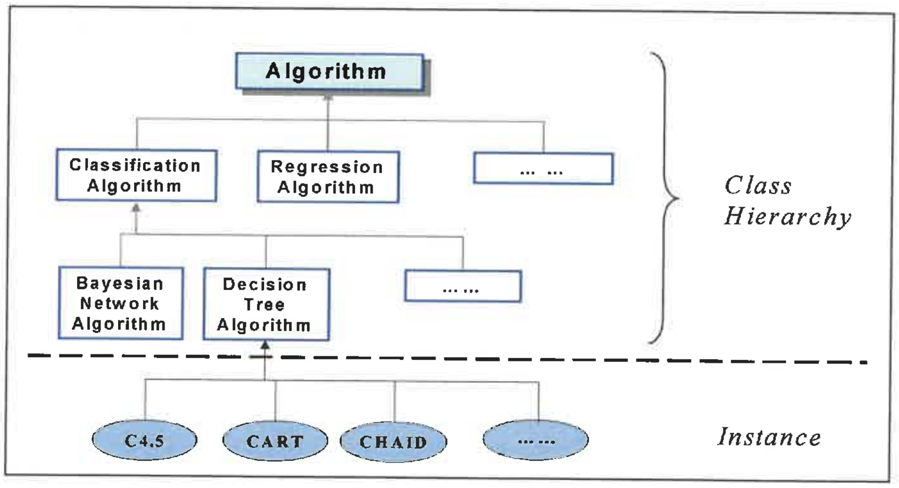
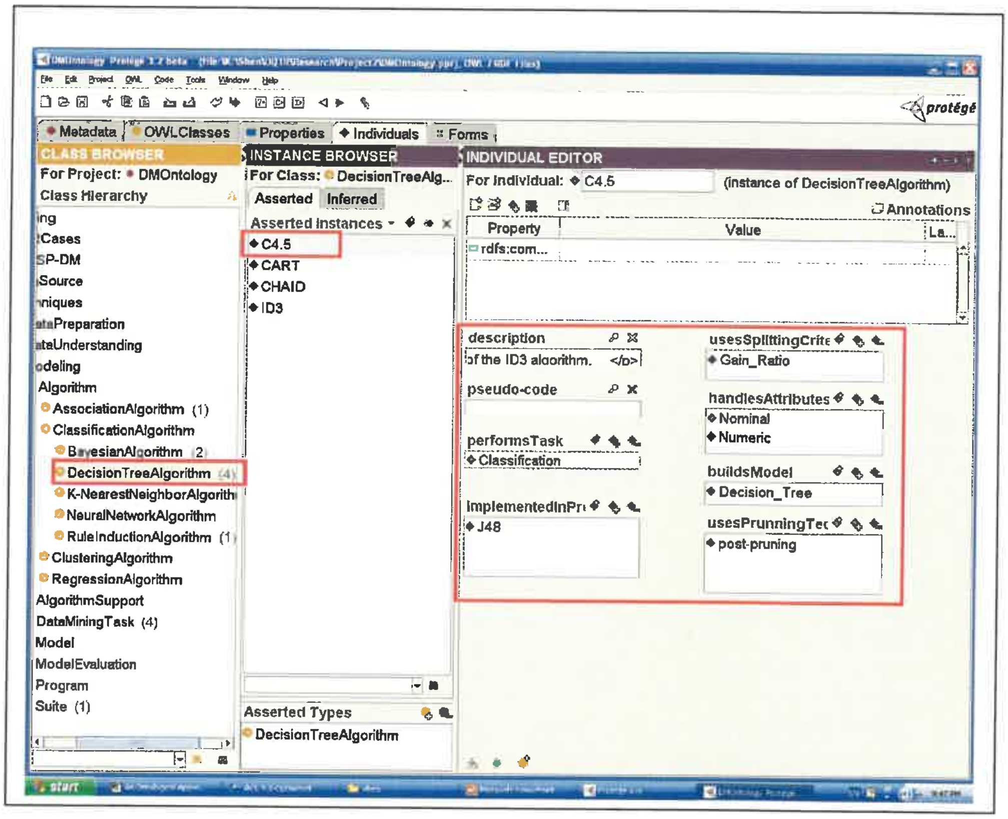


Figure 4.22 The boundary of classes and instances



For Project: DMOntology

Class Hierarchy

ing
Cases
SP-DM
Source
niques
ataPreparation
ataUnderstanding
odeling
Algorithm
AssociationAlgorithm (1)
ClassificationAlgorithm
BayesianAlgorithm (2)
DecisionTreeAlgorithm (4)
K-NearestNeighborAlgorithm
NeuralNetworkAlgorithm
RuleInductionAlgorithm (1)
ClusteringAlgorithm
RegressionAlgorithm
AlgorithmSupport
DataMiningTask (4)
Model
ModelEvaluation
Program
Suite (1)

For Class: DecisionTreeAlg...

Asserted | Inferred

Asserted Instances

- C4.5
- CART
- CHAID
- ID3

Asserted Types

- DecisionTreeAlgorithm

For Individual: C4.5 (instance of DecisionTreeAlgorithm)

Annotations

Property	Value	La...
rdfs:com...		

description: of the ID3 algorithm. </b>

pseudo-code:

performsTask: Classification

implementedInPr: J48

usesSplittingCrite: Gain_Ratio

handlesAttributes: Nominal, Numeric

buildsModel: Decision_Tree

usesPrunningTec: post-pruning

Figure 4.23 Instances of the class *DecisionTreeAlgorithm*

4.3.4 Reasoning

OWL-DL is based on Description Logics (DL) [47], which are decidable fragments of First Order Logic. Because an OWL-DL ontology can be translated into a Description Logic representation, it is possible to perform automated reasoning over the ontology using a Description Logic Reasoner. A Description Logic reasoner performs various inferencing services, such as computing the inferred super-classes of a class, deciding whether or not the classes are consistent, deciding whether or not one class is subsumed by another, etc.

The Protégé OWL plug-in provides access to reasoners such as Racer [48] and Pellet [49, 50]. The current interface of the plug-in supports two types of DL reasoning: consistency checking and classification (subsumption). Consistency checking determines whether a class is consistent. Based on the conditions (constraints) of a class, the reasoner can check whether or not it is possible for the class to have any instances. A class is deemed to be inconsistent if it cannot have any instances. Inconsistent classes are marked with a red-bordered icon. Classification is to test whether or not one class is a subclass of another class. By performing such tests on all the classes it is possible for a reasoner to compute the inferred ontology class hierarchy.

Pellet is the reasoner used in DM ontology. It is a Java-based OWL DL reasoner that can be integrated in a Protégé OWL plug-in. It is based on the tableaux algorithms [51] developed for expressive Description Logics. Since a Protégé OWL plug-in only provides T-Box reasoning techniques, which means reasoning on classes, we cannot check instances at this time. Thus Pellet is mainly used to check the class consistency of the DM ontology.

4.4 Discussion

Our DM ontology was developed to provide the core data-mining knowledge to our data-mining assistant system. Meanwhile, it is also a complete data-mining domain knowledge reference that can be reused and shared by other applications. From this point of view, the DM ontology is a hybrid data mining domain ontology. Comparing with the two other data mining domain ontologies DAMON [35, 36] and IDAs [37], our DM ontology holds several advantages.

Firstly, our DM ontology possesses a very large knowledge scope that covers all the theoretical and practical knowledge related to the data mining domain. The knowledge represented in the DM ontology can be classified into two parts: the conceptual data mining domain knowledge and the system generated knowledge. The data-mining domain knowledge contains the methodology and the detailed applicable knowledge of the whole data mining process from data understanding to model evaluation, while the system-generated knowledge consists of data annotation and case representation. It puts emphasis on (1) the whole data mining process, (2) the mapping of methodological process and corresponding applicable knowledge, (3) the model evaluation methods and criteria, and (4) the integration with other system components. The IDAs ontology covers data pre-processing, induction algorithm and post-processing, but it does not cover data understanding and results evaluation and interpretation. The DAMON ontology has a good knowledge representation about algorithms, methods, tasks, and software, but that corresponds only to the modeling sub-section of our DM ontology. Considering the knowledge scope, and as far as we know, our DM ontology can be regarded as the most complete and comprehensive data mining domain ontology.

Secondly, our DM ontology has a well-organized and specifically constrained representation structure: it is a DL-based formal ontology. It classifies the concepts into four sections according to the different knowledge types; each section holds a clear knowledge scope and a well-defined class hierarchy. For each level of the class hierarchy, the relationships and constraints are precisely defined. The most detailed knowledge is put into the instance level in order to make the DM ontology even more semantically meaningful.

Chapter 5 : A NEW TOOL TO MANAGE THE EVOLUTION OF A PROTÉGÉ OWL ONTOLOGY

Ontology evolution is a relatively new but very important area in ontology research. The major reason is the increasing number of ontologies and fast development in ontology-based projects. Furthermore, the increasing costs associated with adapting them to changing requirements makes ontology evolution a key step in making ontology a truly versatile tool in practical applications. Developing ontologies and their applications is expensive, but evolving them is even more expensive. The importance and challenges of ontology evolution have been realized recently and a great amount of work is needed in this area.

This chapter presents a new methodology and software tool to manage the evolution of an ontology. This methodology is based on the evolution tasks and is implemented as a Protégé plug-in. The importance and the difficulties of ontology evolution is described in section 5.1, the task-driven evolution methodology is presented in section 5.2, and section 5.3 presents the Protégé plug-in, called the Ontology Evolution Tab. Section 5.4 discusses this methodology with other two methodologies and gives some concluding remarks.

5.1 Challenges of ontology evolution

Change is a constant and continual factor in ontology-based applications. The changes may come from the ontology itself and the dependent environment. Thus, to improve the performance and reduce the costs of their modification, the changes have to be reflected in the underlying ontology. If the underlying ontology is not up-to-date, then the reliability, accuracy and effectiveness of the system will decrease significantly [52]. As ontologies are increasingly used in many fields, the need for ontology evolution becomes inevitable. The task of the ontology evolution is to interpret formally all requests for changes coming from different sources and to perform them on the ontology and its depending applications while keeping consistency of all of them.

5.1.1 The importance of ontology evolution

Most of the work conducted so far in the field of ontologies has focused on ontology construction issues. It is assumed that domain knowledge encapsulated in an ontology does not evolve in time. However, in a more open and dynamic business environment, the domain knowledge and the knowledge-based applications change continually. These changes include the updating of domain knowledge, the organization of the information in a better way, the additional functionality of different users' needs, the modification in the application domain or in the business strategy, etc. Three basic sources that can cause ontology changes are described in details in what follows.

As ontology is used to represent domain knowledge, the change of knowledge itself is the most important task of ontology evolution. With the development of the particular domain and the usage of the application system, the underlying domain knowledge may grow rapidly. The growth of knowledge can be the changed or updated information from the existing knowledge, the new knowledge continually accumulated, the management of the out-of-date knowledge, and the changed representation structure. How to keep the ontology updated and accurate is definitely a challenging problem.

Another source of change is the different requirements from different users. Users' requirements often change after the system has been built. There may be different opinions, needs and operations with the same ontology. The adaptation of the system is required for this kind of change.

The third source of change is the environment. The environment in which the ontology-based system operates can change, thereby invalidating the assumptions initially made when the system was built. The change of environment should be transferred into the ontology.

5.1.2 The problems of ontology evolution

Ontology evolution is not a trivial process, due to the variety of sources and consequences of changes. It cannot be done manually by an ontology engineer since he/she is not able to understand all the consequences of a change. Therefore, an evolution tool that is responsible for maintaining evolution is needed. Building such a tool has proven to be a difficult task, since there is almost a complete lack of suitable methodology and techniques. Particularly, there are three challenges for the efficient realization of an ontology evolution software tool.

1. Complexity. An ontology model is an explicit specification of a conceptualization that often has a complicated representation structure. The structure is rich in classes, properties, instances, axioms, constraints and internal relationships; these interdependent concepts make the structure like a complex network. Working with this interwoven structure, it is very difficult for an ontology engineer to interpret the necessary changes for the corresponding ontology operations. Moreover, when a change is applied, this change leads to a series of consequent changes. Even when the effects of a change are minor, the cumulative impact of all the changes can be enormous.

2. Consistency. Consistency is one of the most essential factors that must be considered during the evolution of the ontology. An inconsistent ontology can cause serious (sometimes contradictory) problems to its dependent applications and systems. The challenges of consistency checking lies in two aspects. One is consistency checking within a single ontology, which does not import or export other ontologies. When an evolution task is involved, a simple change may generate a list of consequent changes to keep the ontology consistent. The complexity of ontology evolution increases when the ontology becomes large and rich. The other aspect concerns the ontologies reusing and extending other ontologies. Changes in an ontology may affect the ontologies that are based on it.

Therefore, changes between dependent ontologies are interrelated, and the immediate synchronization between dependent ontologies is required.

3. Dependent applications. The ontology and its dependent applications are two interdependent parts, thus the changes applied to one part may cause some corresponding changes to the other. On one hand, when the ontology evolves, its dependent applications must be updated to maintain consistency. On the other hand, when some changes are applied to the applications, the changes must be transferred to the ontology.

5.2 Conceptual solutions for OWL ontology evolution

Ontology evolution is the timely adaptation of an ontology to the changes in the business requirements, to trends in the ontology instances and the patterns of the usage of the ontology based application, as well as the consistent management and propagation of these changes to dependent applications [24]. There are two major issues involved in ontology evolution. The first issue is the understanding of how an ontology can be changed; the second issue is the decision of when and how to modify an ontology to keep its consistency. We now discuss the main elements of our original contribution to the problem of managing the evolution of an OWL ontology.

5.2.1 OWL change operations

To resolve the changes of an OWL ontology, different change operations must be identified and represented in an appropriate format. Based on the OWL DL language, we propose three types of change operations: basic changes, composite changes and complex changes.

Definition 1: A ***basic change*** is an ontology change that *creates, deletes or modifies* an atomic element of an ontology such as axioms, constructs, constraints, property values, etc. This type of change only performs one simple task; it cannot be divided further. The examples of basic changes are creating a class identifier (name), identifying a class axiom *subClassOf*, defining a value constraint *allValuesFrom*, etc.

Table 5.1 gives a list of some basic changes for the OWL DL language. These changes are classified by different levels of concepts on which the changes will operate: class level, property level and instance level. The changes at the class level deal only with the operations related to the class description such as class identifier, class axioms, and class conditions. The changes at the property level and the instance level deal with the operations related to the property description and instance description respectively.

Table 5.1 Some basic change operations of an OWL ontology

Level		**Change**	**Operations**
Class	Identifier	Class name	create, delete, modify (value)
	Axioms	subClassOf	create, delete, modify (relation)
		equivalentClass	create, delete, modify (relation)
		disjointWith	create, delete, modify (relation)
	Conditions (Property restrictions)	Value constraints	
		allValuesFrom	create, delete, modify (filler, make ∀, make ∃)
		someValuesFrom	create, delete, modify (filler, make ∀, make ∃)
		hasValue	create, delete, modify (filler, make ∀, make ∃)
		Cardinality constraints	
		maxCardinality	create, delete, modify (value)
		minCardinality	create, delete, modify (value)
		cardinality	create, delete, modify (value)
	Intersection, union, complement	intersectionOf	create, delete, modify (relation)
		unionOf	create, delete, modify (relation)
		complementOf	create, delete, modify (relation)
	Identifier	property name	create, delete, modify (value)
		type	create, delete, modify (type)

Property	Constructs	domain	create, delete, modify (filler)
		range	create, delete, modify (filler)
		subPropertyOf	create, delete, modify (relation)
	Relations	equivalentProperty	create, delete, modify (relation)
		inverseOf	create, delete (Boolean)
	Global cardinality restriction	functionalProperty	create, delete (Boolean)
		InverseFunctionalProperty	create, delete (Boolean)
	Logical characteristics	TransitiveProperty	create, delete (Boolean)
		SymmetricProperty	create, delete (Boolean)
Instance	Identifier	name	create, delete, modify (value)
	Values	Class membership and property values	create, delete, modify (value)

However, this granularity for ontology changes is not always appropriate. Often, the intended changes may be expressed on a higher level. For example, we may need to generate a new class and make the new class a subclass of a given class. This task involves several basic changes such as the class name identifier and the axioms subClassOf. If we add some constraints to this class, we also need some other basic changes. In a flexible ontology evolution environment, it should be possible to define changes on a coarser level.

Definition 2: A ***composite change*** is an ontology change that *creates, deletes, adds, removes, or modifies* an ontology class, a property or an instance. This type of change is composed with different related basic changes.

The composite changes represent a group of basic changes applied together. While a basic change can be seen as an isolated modification of an ontology, a composite change defines a "context" of the evolution in a more practical fashion. The composite changes are further divided into three parts: changes for a class, changes for a property, and changes for an instance. These three parts group different basic changes to accomplish a higher-level task. Changes for a class deal with the class identifier, axioms, conditions and constraints. Changes for a property handle the property identifier, constructs, relations, restrictions, and logical characteristics, while changes for an instance work on instance identifier and values.

Table 5.2 describes the change operations applied to the composite changes. The "√ " at the intersection indicates that the operation at the row can be applied at the corresponding column. These five operations are defined according to the operations provided by the Protégé ontology editor so that the evolution tool can be well integrated with Protégé. Note that there is a subtle but important difference between "create" and "add", "delete" and "remove". "Create" means to build a completely new one, while "add" indicates to add an existing one from the list, "delete" means to take out a selected concept, while "remove" indicates to take a selected concept away from its container, but this selected concept will always exist in the list. The "add" and "remove" operations can only be applied to properties. The operation "modify" means we can change the values of all the basic changes included in each level-related changes.

Table 5.2 Change operations applied to the composite changes

	Class	**Property**	**Instance**
Create	√	√	√
Delete	√	√	√
Add		√	
Remove		√	
Modify	√	√	√

In order to keep the ontology consistent during the evolution phase, we must consider the change consequence for each operation at each level. The change consequence will be executed automatically when the user chooses a certain operation. Table 5.3 provides the detailed explanation for each possibility.

Table 5.3 The change operations and their consequences

Change operation	Change consequence
Create a class	The created class will inherit all the properties and conditions from its super-classes, all its subclasses will inherit the properties and conditions associated with the created class.
Create a property	If the property is created inside a class, this class will be the domain of the created property.
Create an instance	This instance will inherit all the properties and conditions defined within the class to which the created instance belongs.
Delete a class	If the selected class and its subclasses have no instances, this class and its subclasses can be all deleted.
Delete a property	This property will be deleted from the property list, and will also be deleted from all the classes (including their instances) that are the domain of the deleted property.
Delete an instance	This instance will be deleted and the property values of other instances who point to the deleted instance will also be deleted.
Add a property	This will only be executed inside a selected class. This class will be the domain of the added property.
Remove a property	This will only be executed inside a selected class. This class will be removed from the domain of the removed property.
Modify	All the basic changes involved in the related class, property, and instance can be modified. The modification can be value, filler, relation, necessary and sufficient condition, etc. This operation is more complicated then the others. It may introduce inconsistency in the ontology.

This is essential for ontology consistency but not sufficient for ontology semantics. The reason is that ontology evolution is a very complex problem, even at the instance level. The change sequence can generate a fully or partially consistent ontology syntax (structure). The remaining part of the ontology syntax, which requires the user's contribution, must be provided and the semantics of each involved concept must be fulfilled by the user. In other words, our methodology to ontology evolution is semi-automatic. Remember that during the whole evolution process, the change sequence (and maybe other techniques) will be applied to guarantee the ontology consistency.

When dealing with a group of classes, for example moving them one level higher in the class hierarchy, the composite changes are not efficient because they can only handle one class at a time. We need changes at a higher level that can manage multiple concepts together.

Definition 3: A ***complex change*** is an ontology change that *moves, merges, splits* ontology class or sibling classes. This type of change can be decomposed into several basic and composite changes. Complex changes can be regarded as an extension of composite changes.

5.2.2 A task-based evolution methodology

Ontology evolution is designed to manage the changes of the ontology; it is often regarded as an updating task applied to the ontology. When applying changes to a class, a property or an instance, the tasks are not isolated because there is a large number of relationships and interactions among them. In some cases, one change in a class may generate some consequent changes in its associated properties and instances. In fact the evolution tasks are greatly "internal dependent". The word "dependence" can be further explained with two meanings: one is consistency dependence; the other is usage dependence.

Consistency is an important issue in ontology evolution and its dependent applications. One change to a class (for example, creating a new property) can trigger a series of necessary changes (the constraints of the new created property, the domain, the range of this property, the instances belonging to the class, etc.). These changes must be considered and well applied to preserve the ontology's consistency.

Regarding usage dependence, we may consider some common cases of developing an ontology. When we create a new class in the ontology, we usually will not leave this class "empty" in the ontology. The class is created to describe something new: new concepts, new relations, and new instances. In other words, the newly created class will be described with axioms, conditions, linked with properties, and filled with instances. The finest level of change is the instances of this class. Another case is to create a new property. This property will surely not exist alone; it will be linked to one or several classes as its domain and range. So how to deal with the classes which are linked to this property as its domain? For these classes and all their dependent instances, one new property is added. To keep the ontology more meaningful, richer in semantics, it is better to fill the value of this property in all affected instances. This means that the change of a property will also produce some changes in the affected classes and their dependent instances. The third case to consider is the instance itself. When we create a new instance, we will not only create its name, but also define its class memberships and property values. It may be concluded from these problems that:

(1) From the usage point of view, one task may require several changes, and all these changes mostly occur one by one, or series by series. This is crucial for ontology evolution development. The well-organized change series will guide users to understand the dependence of changes, and thus save time and improve accuracy.

(2) The changes associated with the classes, properties and instances are interwoven together. The changes related to the class level and property level will eventually cause the changes at the instance level. The property-related change will trigger the changes to affected classes and corresponding instances, while the class-related changes will cause the changes to classes properties and all dependent instances.

Considering the change series and the evolution tasks we have discussed above, what we actually propose here is to create a new task-based evolution methodology. This methodology concentrates on the ontology evolution tasks. It aims to group all the necessary actions and the corresponding changes for the most commonly-used tasks of ontology development and updating. It can help users, especially non-expert users, recognize the reason why the actions of a particular task are grouped together in a particular order, thus helping them to further understand the techniques and skills involved in the evolution of an ontology. This methodology is designed for an OWL DL ontology; it mainly focuses on the composite changes and their operations.

The fundamentals of our methodology are based on the different layers of evolution tasks. Figure 1.1 shows the main concepts of the methodology. The evolution tasks of an OWL DL ontology are classified into three layers: class related tasks, property related tasks, and instance related tasks.

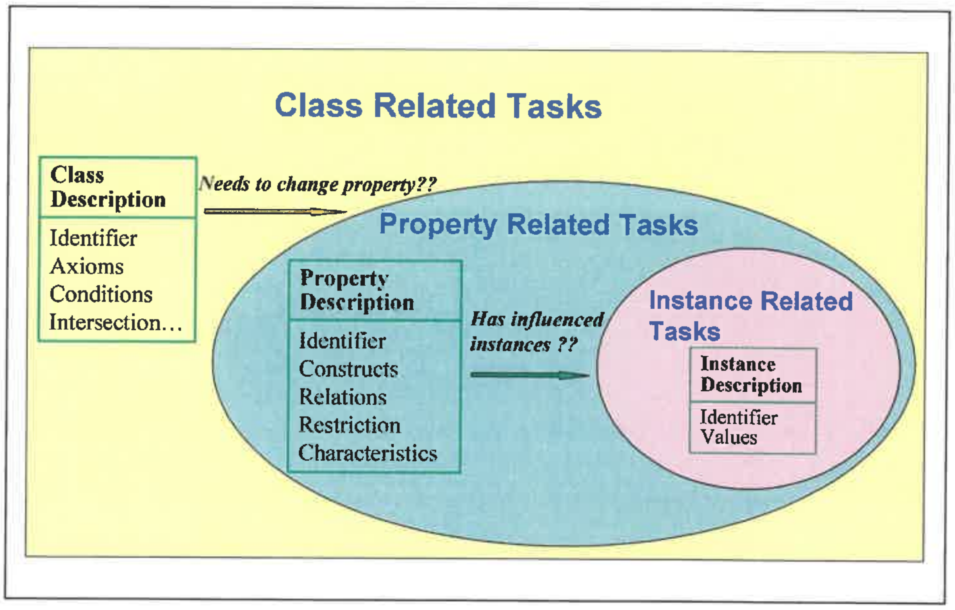


Figure 5.1 The strategy of ontology evolution

Instance-related tasks is the smallest layer that handles only the changes applied to the instances. It consists of the creation, deletion and modification of instances, including the instance name and values. Although the instance-related task is simple, it is the most useful layer of the whole methodology. On the one hand, instance changing is the most widely applied action of ontology updating and maintenance. When the structure of an ontology is well defined, it usually does not change a lot, but the accumulated new instances may need to be added into the ontology and the old instances may need to be updated. On the other hand, the instance-related task is the basis of the other two tasks; class-related tasks and property-related tasks will all generate some instance changes.

The middle layer of the structure is the property-related task. The task begins with the identification of a property change, which can be creating a new property, deleting or modifying an existing one. For example, as illustrated in detail in Figure 5.2, if a new property is required, the property descriptions must be changed. Creation of the property name, type, domain, range, restrictions, etc., is needed. It is very important to note that when some classes are specified as the domain of this property, the property-related tasks is connected to the instance-related tasks. There is a new property added to the classes defined as the property domain, all the instances of these classes have been influenced by the newly added property. If the property has some affected instances, the task goes to the instance layer; a value of this property should be added to the affected instances.

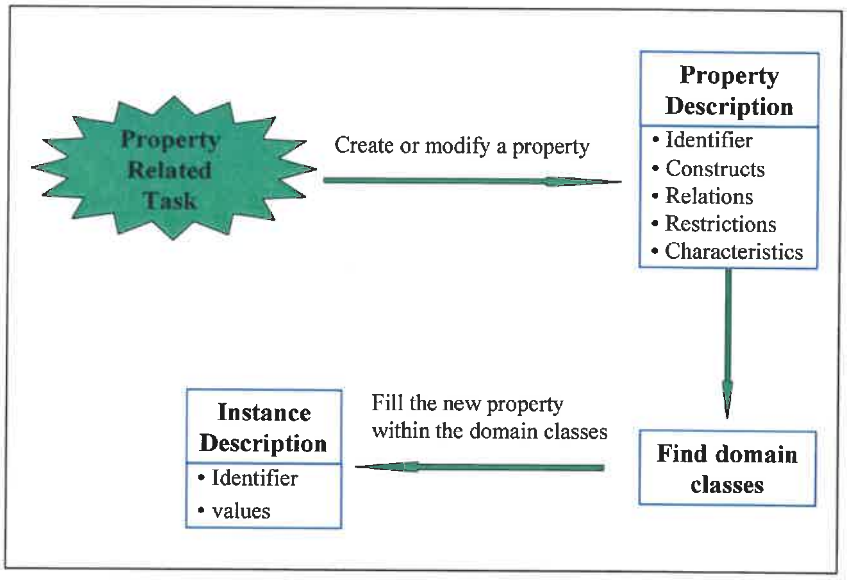


Figure 5.2 The property related task

Class-related tasks is the outer layer of the structure, which involves both property-related tasks and instance-related tasks. Figure 5.3 shows an example of creating a new class. To create a new class, the class name should be identified and at least one property should be added because the class conditions and constraints are specified through the properties associated to the class. The added properties can also make the class more meaningful in the ontology. If some properties are added, the class-related tasks points to the property-related tasks, which means the property descriptions should be defined in this step. When creating instances in this class, the value of the properties should be identified. These instances can be considered as the affected instances, and the task thus reaches the instance layer.

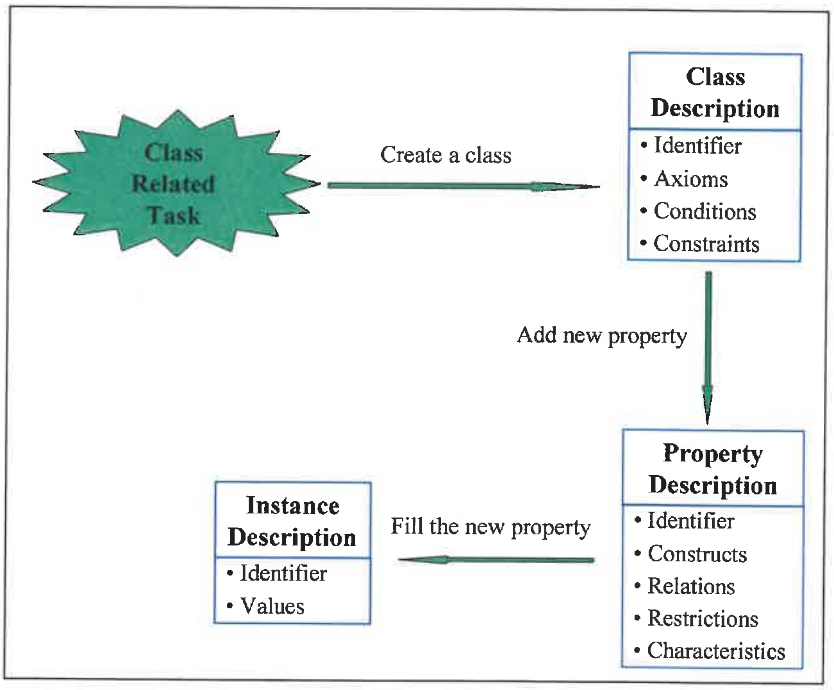


Figure 5.3 The class related task

5.3 The ontology evolution tab: a new Protégé plug-in

5.3.1 Overview of the new ontology evolution tab

The Ontology Evolution Tab is a new tool for Protégé OWL ontology evolution integrated in the Protégé platform. This tab implements the task-based methodology discussed above to fulfill the ontology evolution tasks. Its main purpose is to guide the user, especially the non-expert user, to complete the basic common tasks of developing and updating an ontology step by step.

The Ontology Evolution Tab is based on the OWL ontology language and works as an extension of the Protégé OWL plug-in. Usually, when we create a class in an ontology just by defining its name, properties and restrictions using the “Class” widget provided by Protégé OWL, the job is not totally completed—the class is “empty” although its frame is well defined. We need to create some instances to make the class more semantically meaningful. Although adding instances can be done with the “Instance” widget, some problems do exist for the users who don’t know Protégé well enough to follow all the steps of their task. The same problem exists when we create a property. For this end, the newly developed Ontology Evolution Tab aims at grouping and arranging the necessary steps of each basic task of ontology evolution. It can be used as a wizard to deal with the tasks on ontology classes, properties and instances such as “create”, “delete” and “modify” from the very beginning (e.g. choose the task) to the end (e.g. edit the instance in the created class).

5.3.2 Implementation of the ontology evolution tab

There are five sub-tabs in the Ontology Evolution Tab: Create Class, Delete Class, Create Property, Delete Property, and Create/Delete Individual. Before executing some actual tasks, this tool needs users to understand the requirements and the category of the task. For example, if a user wants to add a new class to the ontology, create a new property for this class and create some instances in this class, then the task is class related and the user should go to the Create Class sub-tab. If a user wants to create a new property, then the Create property sub-tab is the appropriate sub-tab to use. This sub-tab will also guide the user to complete all the involved steps to make the newly created property more meaningful in the ontology. The detailed description of these five sub-tabs is presented below.

Create class

The Create Class sub-tab performs the class related task: it first creates a class, then defines its properties and constraints, and finally creates instances in it. The task performed in this sub-tab is divided into five steps: (1) Choose the location where the new class will be created. The location can be a subclass or a sibling class of a selected class. (2) Edit the class. The properties and class conditions and axioms can be added to the created class. The definition of the class should be complete when this step is finished. (3) The newly created class will be automatically checked and selected from the class browser that allows users to verify whether it is the created class. Note that only the selected class in the class hierarchy is allowed to perform the consequent actions such as creating instances. (4) Create/delete instances in this class, and (5) Edit the created instances.

Figure 5.4 shows that a new class *Restaurant* is created, a property *sellsPizza* and its corresponding constraint are added to the class. The class is checked and selected automatically by the tab, as shown in Figure 5.5, then some instances are added to the class *Restaurant* shown in Figure 5.6.

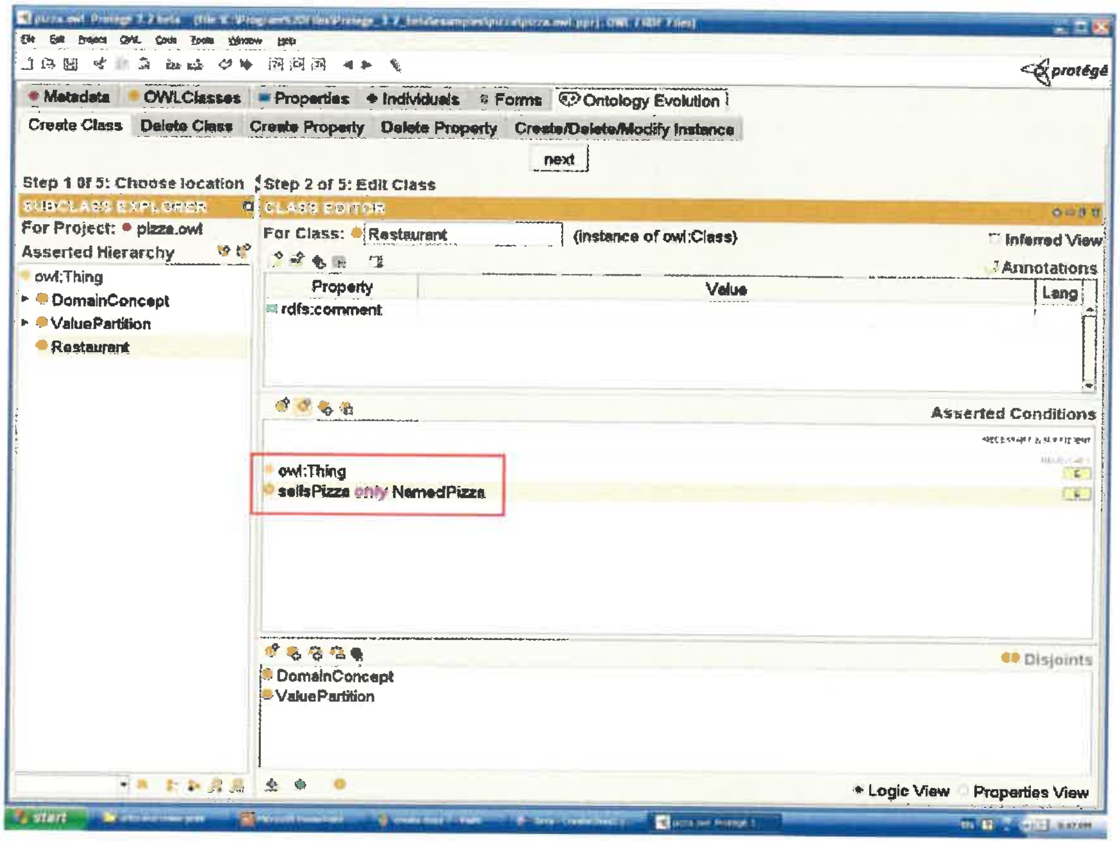


Figure 5.4 Create and edit a class Restaurant

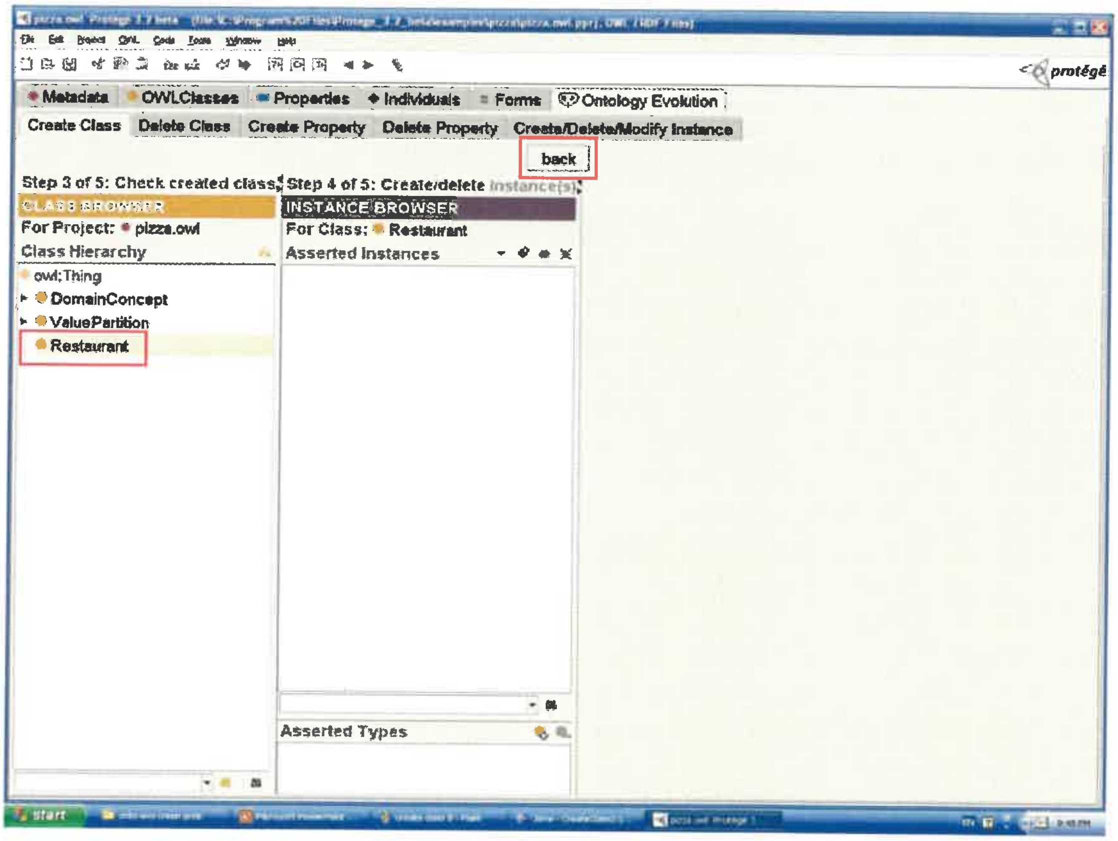


Figure 5.5 Check the created class restaurant

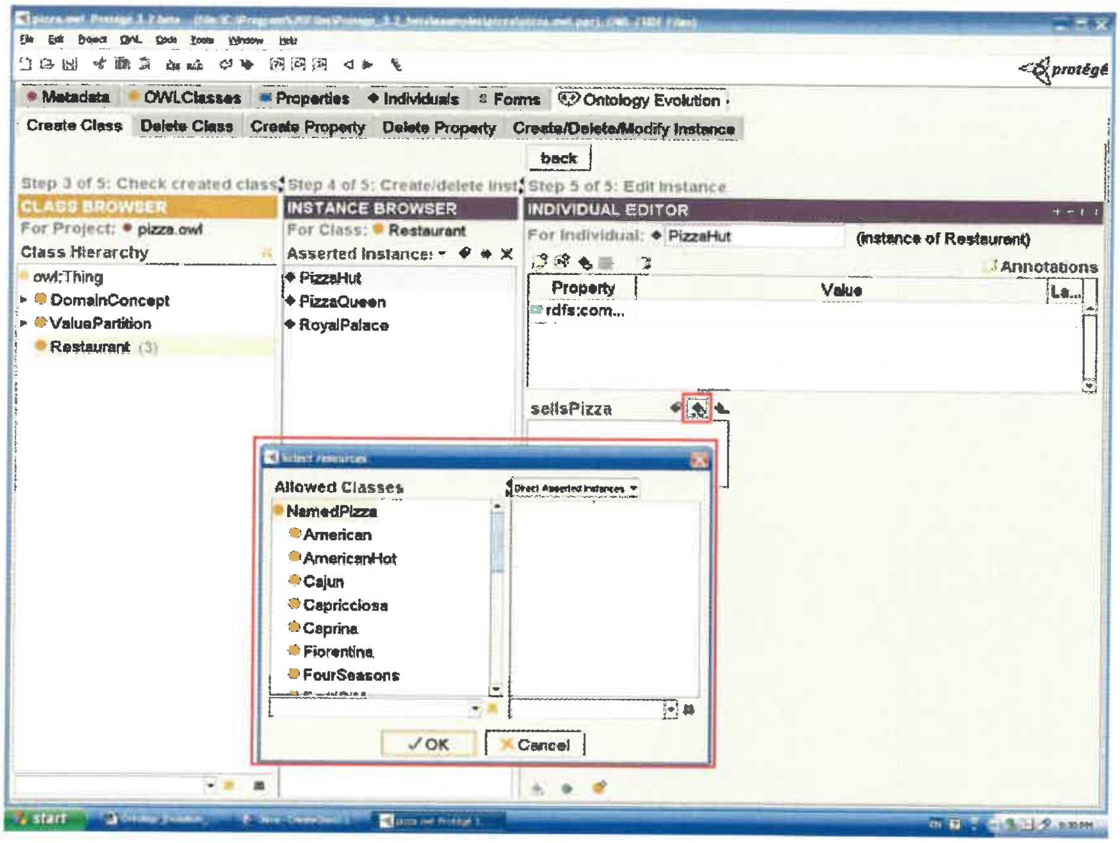
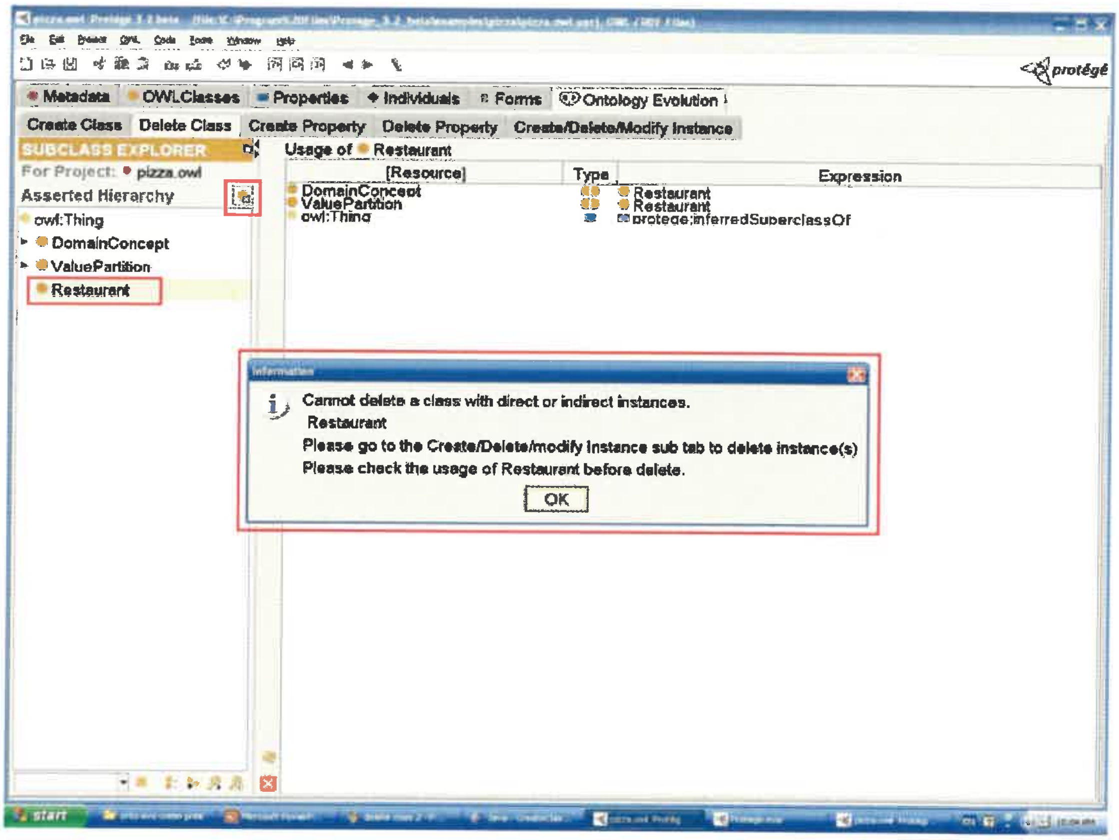


Figure 5.6 Create and edit instances in the class Restaurant



Usage of Restaurant

[Resource]	Type	Expression
DomainConcept		Restaurant
ValuePartition		Restaurant
owl:Thing		protege:inferredSuperclassOf

Information

Cannot delete a class with direct or indirect instances.
Restaurant
Please go to the Create/Delete/modify Instance sub tab to delete instance(s)
Please check the usage of Restaurant before delete.

OK

Figure 5.7 Try to delete the class Restaurant

Delete class

The Delete class sub-tab deals with the deletion of the selected class. If the class has direct or indirect instances, it is not allowed to be deleted. Users may go to the *Create/Delete/Modify Instance* sub-tab to delete its instances first. When selecting a class to be deleted, the list of the usage of this class is presented at the right part of the screen. This function allows users to check the class usage carefully before the deletion action since deleting a class, especially a class with subclasses, can be very problematic. Indeed, the changed ontology may lose not only the deleted classes from the class hierarchy, but also the filler of some properties, and some restrictions by a wrong deletion. The screen shot of deleting the class *Restaurant* is presented in Figure 5.7.

Create property

The Create property sub-tab performs the property related task: it (1) Creates a property. (2) Defines its domain, range, inverse property and other characteristics. (3) Finds the changed classes caused by the created property, and (4) Edits the instances of the changed classes.

The task of this sub-tab is further divided into five steps: (1) Choose the property type (object, dataytpe, or annotation) and create a new property. (2) Edit the created property. The annotations, domain, range, and some property characteristics (ex. Functional, reverse) can also be defined in this step. (3) Choose the changed class to edit its instances. The classes that have been added as the domain of the created property will be automatically checked and highlighted in red in the class hierarchy panel; these classes are considered as the changed classes since there is one more property added to their property list. This change can be reflected either from the *Property and Restrictions* widget in the Create Class sub-tab or from the *Individual Editor* in the Create Property sub-tab. The instances of the changed class are also changed accordingly: they have one more property added and the filler of this property is still empty. The purpose of choosing the changed class is to edit its changed instances. (4) Choose instance(s) from the changed classes, and (5) edit the selected instance(s).

Figure 5.8 shows that a property *makesPizza* is created, its domain is set as the class *Restaurant* and its range is set as the class *NamedPzza*. The changed class *Restaurant* is highlighted automatically, as illustrated in Figure 5.9, since the property makesPizza is added to the instances of the class *Restaurant,* as presented in Figure 5.10.

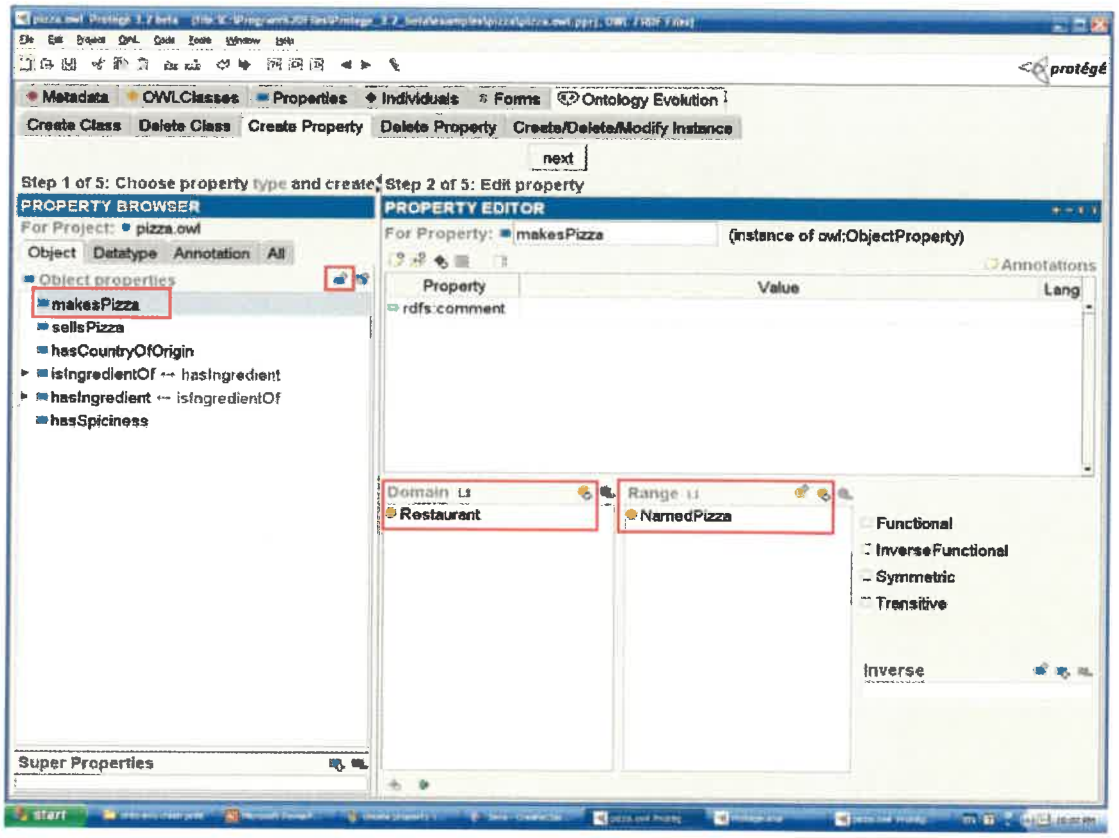
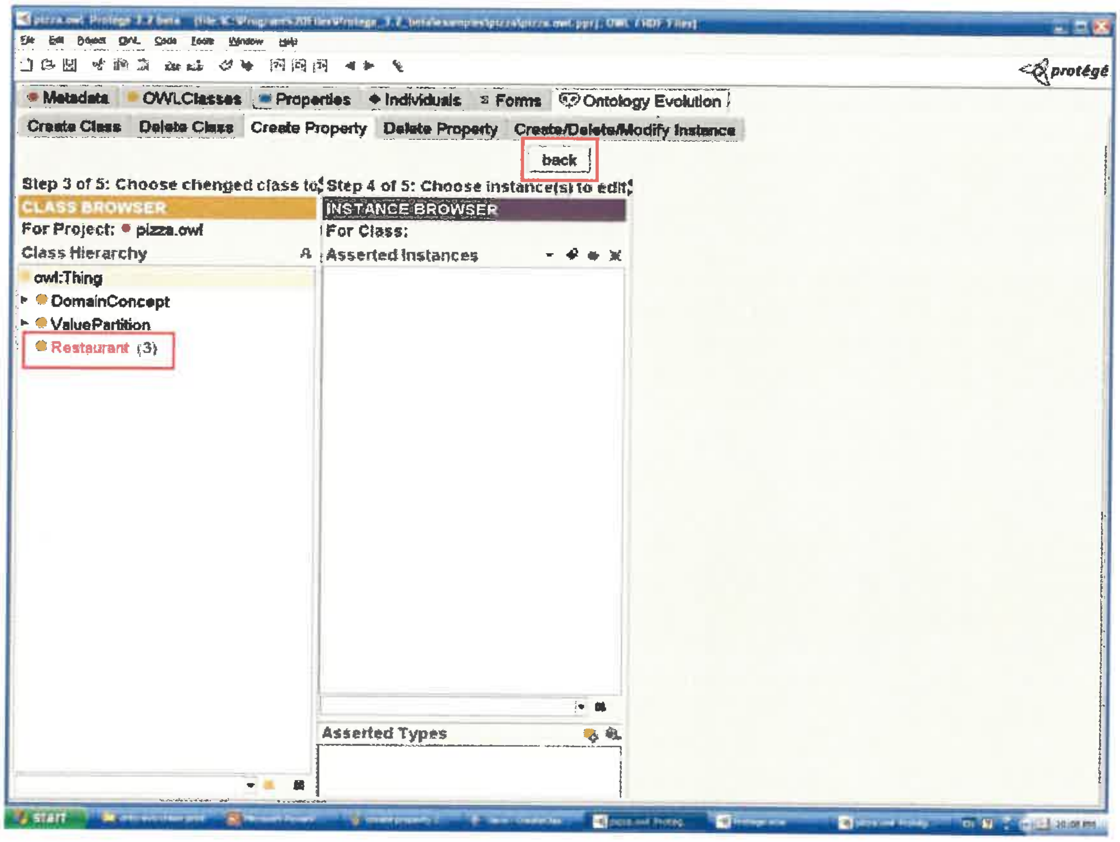


Figure 5.8 Create a property *makesPizza*



Step 3 of 5: Choose chenged class to

Step 4 of 5: Choose instance(s) to edit

CLASS BROWSER

For Project: pizza.owl

Class Hierarchy

owl:Thing

DomainConcept

ValuePartition

Restaurant (3)

INSTANCE BROWSER

For Class:

Asserted Instances

Asserted Types

Figure 5.9 The changed class *Restaurant*

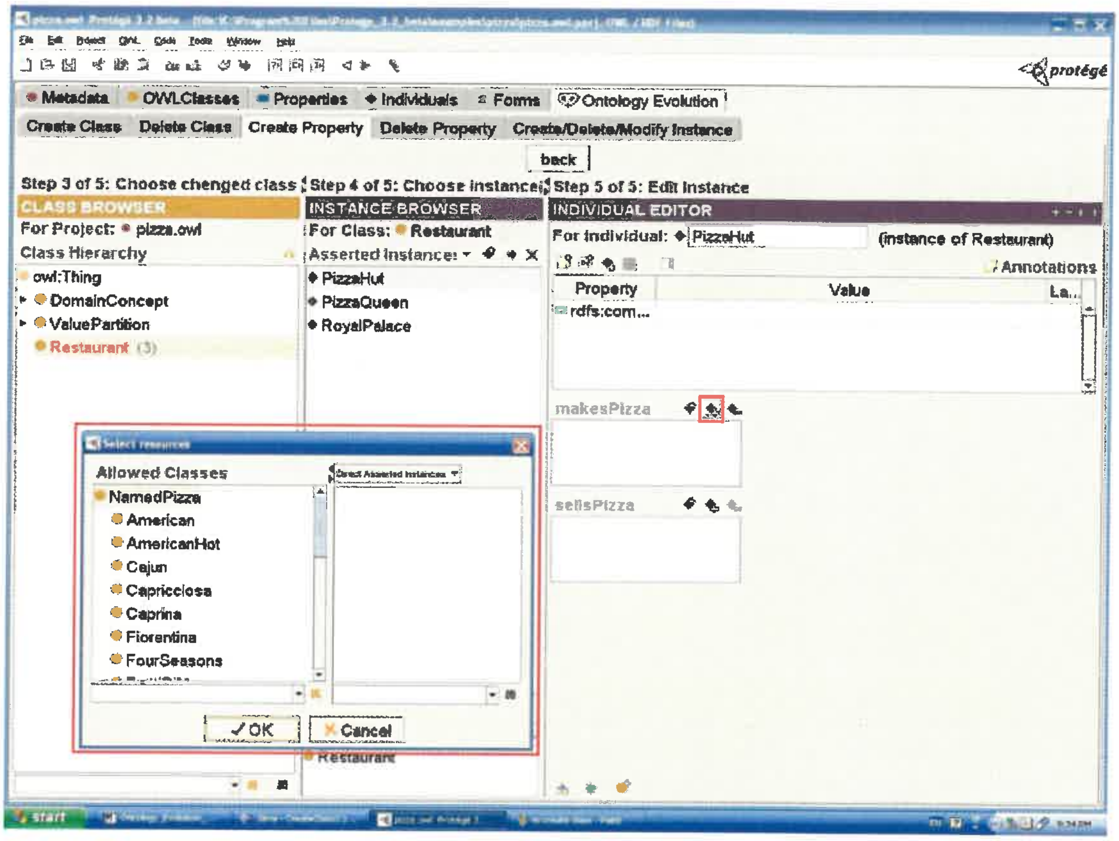


Figure 5.10 The property *makesPizza* is added to the instances of *Restaurant*

Delete property

This sub-tab deals with property deletion. Before deleting a property, make sure to check the usage of this property. When selecting a property to be deleted, the list of usages of the property is also presented at the right part of the screen. The presented information list allows users to check the usage of the selected property before deleting it. Figure 5.11 shows the screen shot of deleting the property *sellsPizza*.

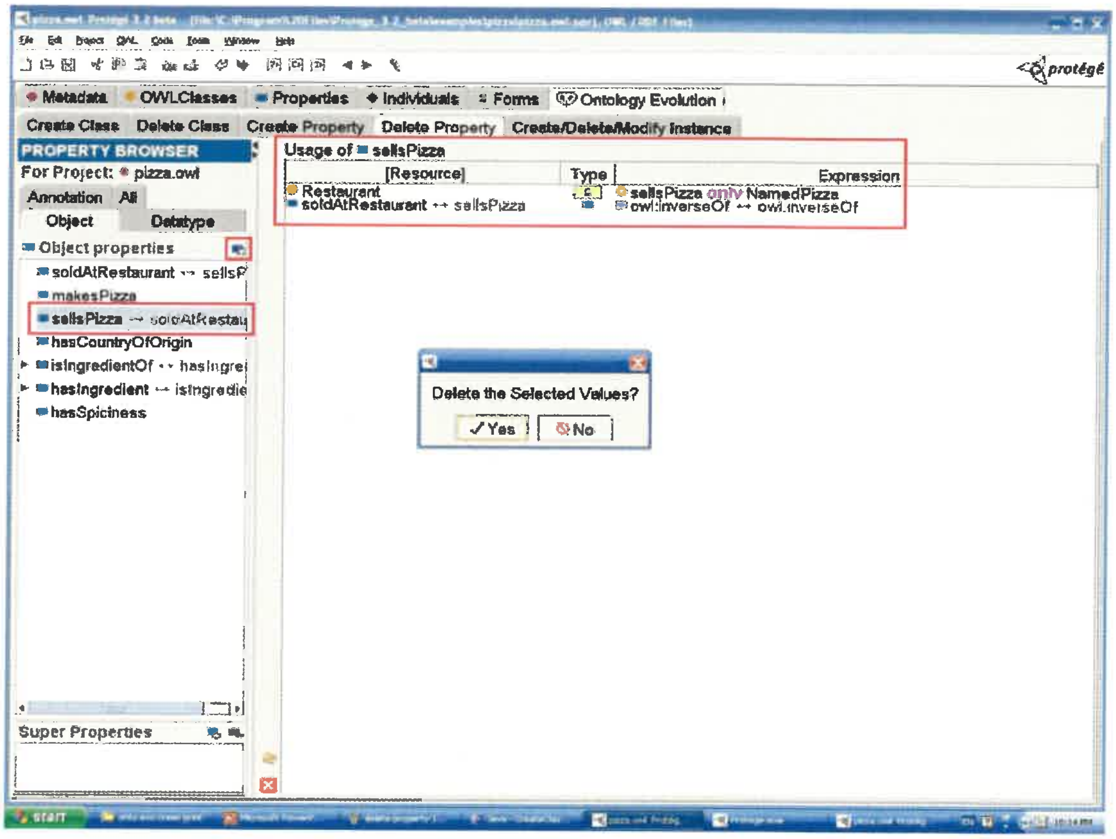


Figure 5.11 Delete the property *sellsPizza*

Create/delete/modify instance

This sub-tab provides the functions to create, modify or delete instance(s) from an existing class. The task is further divided into three steps: (1) Choose an existing class to edit its instances. (2) Create some new instances or delete some existing instances. (3) Edit the selected instance(s). When creating an instance, *edit instance* means to add, delete or modify the filler of each property of this instance. Note that only the name of the instance and the filler of the properties describing the instances can be modified. Figure 5.12 shows the screen shot of editing the instances of the class *Restaurant*.

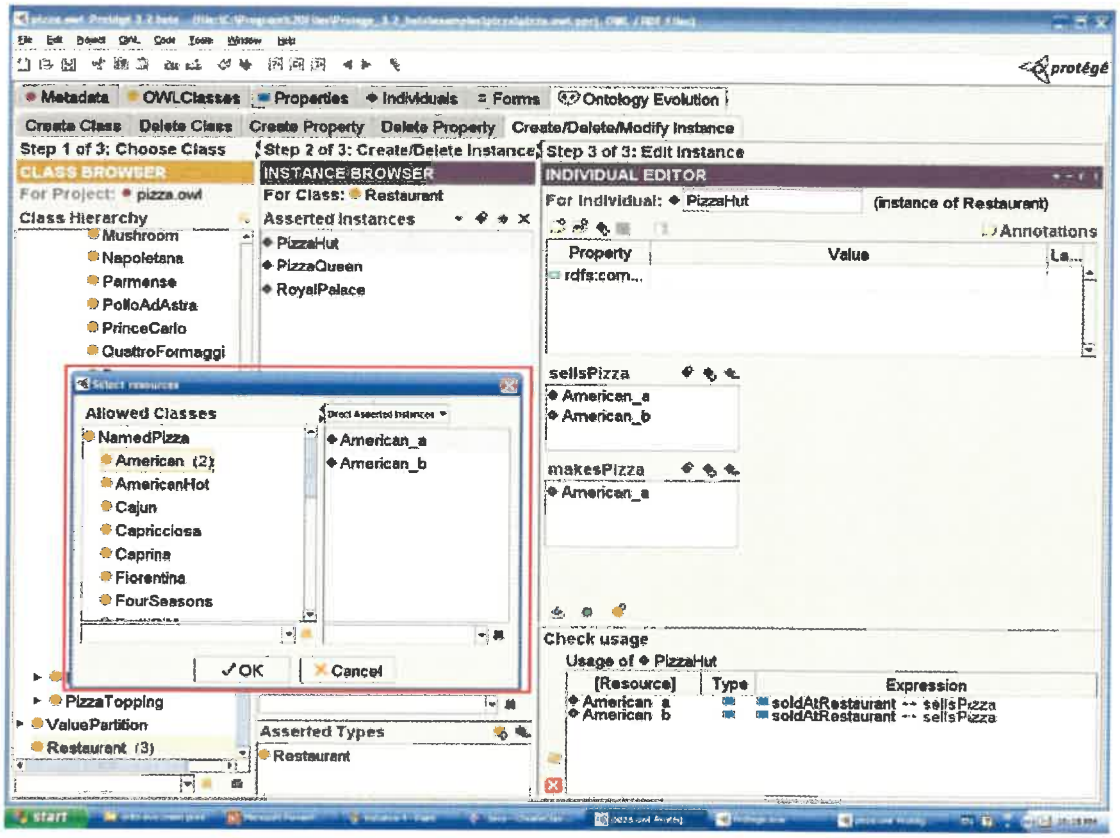


Figure 5.12 Edit the instance of the class *Restaurant*

The Ontology Evolution Tab is the first step of the implementation of the evolution methodology. However, we must emphasize that, in our opinion, it constitutes a significant and original contribution to the current state of the art in ontology evolution, especially in the context of OWL Protégé for which no similar plug-in existed before. Also, it must be pointed out that although we have applied our proposal to a DM ontology, the plug-in is actually domain-independent and could thus be re-used in any application domain. The plug-in was designed as a step-by-step wizard to guide users to accomplish some basic evolution tasks. It supports the creation and deletion tasks involving classes, properties and instances. When applying these two categories of change operations, it is capable of automatically locating the created class, identifying the changed class and influenced instances for a created property, and showing the usage information about a class or a property to be deleted. Modification is another category of change operations discussed in the methodology. This is the most complicated and dynamic category because we can modify almost everything in the ontology. The difficulty dramatically increases when capturing this type of changes while keeping the ontology consistent. The plug-in only supports the same modification operations as Protégé OWL provides for the time being, this can be done through the *Create Class*, *Create Property*, and *Create/Delete/Modify Instance* sub-tabs. More sophisticated and intelligent modification operations are needed in the future.

5.4 Discussion

As ontologies and ontology-based applications have developed tremendously in various domains in recent years, the updating and maintaining of ontologies has become a very important issue. Ontology evolution is a challenging endeavor that faces several difficulties. Among them, the ontology complexity and consistency are the two most important difficulties to be considered. Research in this area is still in its initial phase: the importance of ontology evolution is relatively new and relatively little research focuses on it. The results are quite limited: most of the research groups working on this problem tend to propose theoretical ideas and only a few approaches are actually implemented as actual ontology evolution tools.

The two well-defined methodologies with implemented tools are Stojavonic's methodology [24, 25] and Klein's methodology [28, 29]. Stojavonic *et al.* propose a general user-driven ontology evolution process that can be applied to different evolution tools. They also define the concepts of resolution points that give users flexibility to choose the proper change consequence. However, the corresponding tool of this methodology has some drawbacks: it does not consider the dependence of change consequence; the new changes are not totally integrated into the existing ontology. Klein *et al.* propose a component-based framework to manage changes of distributed evolution between different versions. The methodology defines the change operations of the OWL language and an ontology of change operations. This ontology is used as the main component of the framework. However, this framework is designed to compare and find change information between two ontology versions, it is not suitable for ontology evolution tasks.

Our ontology evolution methodology is based on the evolution tasks and their activities. It takes into account the fact that the tasks of maintaining an ontology are mutually dependent and emphasizes the consistency dependence and usage dependence of ontology evolution tasks. This task-based methodology for ontology evolution possesses four remarkable advantages:

1. Our methodology defines basic, composite and complex changes at three different granularities. Basic changes deal with the atomic, non-dividable operations, composite changes deal with the change operations related to the class, property and instances, and complex changes manage the change operations of a group of classes. The definition of basic change in our methodology is similar to the elementary change in Stojavonic's process and basic change in Klein's framework, but the other two levels are different although we use the same or similar terms. These three layers of change operations finely define the relationship between evolution tasks and their containing activities. They provide a solid infrastructure of building our evolution strategy and its corresponding tool.

2. All related activities (or change operations) are linked and organized in an appropriate order to fulfill a task. The results of an evolution task, especially the newly created classes and properties, do not exist separately in the ontology. On the contrary, they are already integrated into the concept hierarchy of the ontology when the task is finished. Comparing to Stojavonic's methodology [24, 25], which leaves the added concepts isolated from the main body of the ontology, our methodology is capable of managing the relationships between

change consequence and usage independence. This mechanism can help resolve problems of ontology consistency.

3. The ontology evolution tool focuses mainly on the structure and optimized change consequence of the evolution tasks. It can be regarded as a step-by-step guide that provides the precise directions of "what to do next" to assist users to complete the evolution tasks throughout various steps. The wizard-like interface can make the process more predicable and controllable. These characteristics are particularly useful for non-expert ontology users.

4. Ontology evolution is regarded as a process of knowledge adaptation, which occurs when some changes are demanded. The results of the evolution task can be an updated ontology or a new version of an existing ontology and its adapted dependent applications. Klein's methodology [28, 29] focuses on two ontology versions: it assumes that the two versions already exist and tries to find and derive all change information between two versions. It is a post evolution tool. Our methodology is designed for the evolution process itself, it can be used for both developing and updating an ontology, and can generate an up-to-date ontology.

However, as a beginning step towards the solution of ontology evolution, this work still has several areas that need to be further improved. The methodology and the plug-in are all semi-automatic; it requires the users' supervision and interaction. The users are supposed to know the basic knowledge about ontology development and Protégé OWL. They should also be able to identify the needs of the evolution task, select the appropriate task layer and understand each change they made in the activity series. Another area is consistency checking. Since the ontology consistency is not completely guaranteed for all the possible changes, manual checking the consistency with a reasoner is required. The plug-in was developed for the Protégé OWL editor: it uses Protégé's *undo* and *redo* buttons from its main menu for reverse functions. This is not suitable for complex changes, some other new functions to improve the ontology reversibility are needed.

Chapter 6 : FUTURE WORK

The DM ontology and the ontology evolution tool is an essential component of our intelligent data mining assistant system, which is used to represent and keep updated the data mining knowledge. From the data mining point of view, this is the first phase of integrating data mining and the decision support system to support successful decision making. From the ontology point of view, this is the first fruitful result of applying ontological technique into data mining. This work provides an interesting ontological platform to manage complex knowledge with very good performance. However, as the field of ontologies is currently a new and non-mature research area, several future work items can be identified to improve the results and performance of our DM ontology and our ontology evolution tool (plug-in).

1. Ontology completion

The DM ontology is a specification of data mining knowledge. As the structure and hierarchy of DM classes are precisely defined in the DM ontology, the next step would be to add more instances into the classes. Although a great number of current concepts are already represented in the ontology, data mining knowledge is increasing tremendously and the new knowledge should be added into the DM ontology to keep it complete and up-to-date. For the Technique section of the DM ontology, the new instances are mostly the new algorithms, new programs and new model evaluation criteria. For the Data Source section, as new tables and attributes are added into the data warehouse, their metadata should be added as new instances. For the CBR cases section, when new cases are created, they could eventually be promoted into the ontology as new instances.

2. Improvement of the ontology evolution tool

Currently, the Ontology Evolution Tab supports basic and composite change operations; the support of complex change operations would be the next step to be further developed. The complex changes handle the operations applied to a group of classes such as moving some sibling classes one level higher or changing the super class of some sibling classes. This level of operation requires a deeper understanding of the ontology language and editing tools. The implementation of the complex change operations would also facilitate the ontology evolution tasks.

Another potential improvement of the Ontology Evolution Tab is the enhanced support of the modification of classes and properties. Presently, the activities of modification are covered by the Create Class sub-tab and the Create Property sub-tab. These two sub-tabs can only deal with simple modifications manually; some other more intelligent types of support are needed. Moreover, how to automatically check the consistency of the ontology is another question to be solved. It could be possible to integrate the evolution tool with a reasoner so that the consistency checking and the concept inference could be carried out to enhance the performance of the evolution tool.

3. Performance evaluation

As one of the fundamental components of our data mining assistant system, the DM ontology and the ontology evolution tool will be implemented and evaluated through the DM assistant system. Our DM assistant is initially deployed to support a strategic decision support department within a university setting. Specifically, the objectives of the assistant consist of analyzing large amounts of actual student academic details found within a data warehouse and deriving various predictive and explanatory models using data mining tools. It mainly focuses on three categories: student admission process, student retention and student follow-up.

The evaluation of the DM ontology would mainly focus on the ability of knowledge representation. This ability could be assessed by the ontology completeness, consistency, and conciseness. As the DM knowledge increases, the DM ontology could be hardly complete; the term completeness is a relative concept. In our case, the DM ontology could be considered complete if it covers all the common and potentially useful DM knowledge, including both theoretical and practical knowledge, for a data miner who deals with various data mining activities. The DM ontology is currently consistent; ontology consistency checking could be useful when some new classes or instances are added into the ontology. The conciseness of the DM ontology could also be assessed by checking whether there are some unnecessary or useless definitions in the ontology.

The evaluation of the ontology evolution tool would mainly focus on its performance as a wizard. This tool is initially designed to guide the non-expert users to manage ontology-updating tasks. The ability of directing users through different steps, helping users find the consequence of activities, and reducing the time and effort for a given task could be some important criteria to evaluate its performance.

Chapter 7 : CONCLUSION

The well-designed data mining ontology and the new ontology evolution methodology (and tool/plug-in) are the main results of this research work, which is a fundamental part of our larger project on intelligent data mining assistance.

The main objective achieved in this work is the development of an ontology-based methodology to data mining to support non-expert data miners and decision makers to make better choices during various data mining tasks. We have resolved at a satisfactory level two important problems in developing the data mining assistant system: the support of non-expert data miners and the definition of DM knowledge. The core of the work accomplished is a DM ontology that provides a relatively complete data-mining knowledge base to the data mining assistant system. Another essential part is the new ontology evolution methodology to support ontology updating and maintenance. Here are more details on our main accomplishments.

Firstly, based on the Protégé (Stanford University) software and the OWL language, a new data mining ontology has successfully been developed. As a knowledge source for the system, the role of the DM ontology is the knowledge representation. Two types of knowledge are represented in the DM ontology: data mining domain knowledge and system generated knowledge. The data mining domain knowledge consists of both the methodology and the detailed applicable knowledge of the entire data mining process, which are represented respectively in the section of the *CRISP-DM* and *Techniques* sections in the DM ontology. The system-generated knowledge consists of data annotations and CBR case representation, which are represented in the *Data Source* and *CBR Cases* sections in the DM ontology.

Particularly, our DM ontology puts emphasis on the whole data mining process, the mapping of theoretical aspects, including methodological ones, and corresponding applicable knowledge, the detailed description of data mining algorithms and programs, and the statistical interpretation of model evaluation methods and criteria. We believe these characteristics make our DM ontology the most complete and comprehensive data-mining domain ontology available. It also gives data miners and decision makers a better understanding of data mining knowledge, which can greatly facilitate parts of a decision making process.

Secondly, our DM ontology is further integrated into the DM assistant system. The DM ontology is one of the three main components in the system; it cooperates with the CBR system and the data warehouse to provide more intelligent support for data mining activities. The integration of the DM ontology and CBR cases is realized through two aspects. One is the ontological representation of the structure and semantics of the cases; the other is the relationships between cases and related DM knowledge. It can help data miners to understand, classify, navigate and choose appropriate cases as well as provide heuristic recommendations for a given DM task.

The DM ontology and data warehouse is integrated through the ontological representation of the metadata of the data warehouse, which involves the dictionary of data marts, tables and attributes. The relationships between data source, CBR cases and DM knowledge are also precisely defined in the DM ontology. Integrating the DM ontology into the assistant system can greatly assist data miners to better understand the requirements, activities and outputs of each phase of data mining tasks, thus making better choices for the given tasks.

Thirdly, a new ontology evolution methodology was proposed in this work. With the accumulated new knowledge from numerous applications and the changes in the DM field, the DM ontology needs to be updated. This new methodology defines three levels of change operations: basic change, composite change and complex change. The strategy of evolution is based on the evolution tasks and their associated activities. The evolution tasks focus on the composite change operations and are classified at three layers according to the different task scopes: instance related tasks is the inner layer that deals with the changes only applied to the instances; property related tasks is the middle layer that handles the changes applied to the properties and their influenced instances; while class related tasks is the outer layer that manages the changes of classes, their associated properties and instances.

This evolution strategy is implemented as a Protégé plug-in: the ontology Evolution Tab, which can be used for any Protégé OWL ontologies, whatever the application domain. The plug-in groups and arranges the necessary steps of most commonly used evolution tasks into five sub-tabs: create class, delete class, create property, delete property and create/delete/modify instance. It is used as a wizard to deal with the tasks on ontology classes, properties and instances. It is capable of guiding the user, especially the non-expert user, to complete the basic common tasks of developing and updating an ontology step by step.

As one fundamental component of our data mining assistant system, the DM ontology and the ontology evolution tool will be evaluated through the application of the assistant system.

Our work puts forward the foundations for our data mining assistant system and paved the way for a better integration of data mining and decision support system. Furthermore, our versatile DM ontology evolution strategies will greatly improve the accuracy, consistency, and efficiency of the evolution tasks. More importantly, this evolution methodology possesses a great potential for further development. We think it clearly demonstrates the potential success of integrating ontology into data mining and thus opens a brand new research area in data mining development.

References

1 Turban, E. (1995) *Decision support and expert systems : management support systems.* Englewood Cliffs, N.J. Prentice Hall. ISBN 0-024-21702-6

2 Mladenic, D., Lavrac, N., Bohanec, M. Moyle, S., (2003) *Data mining and Decision support: integration and collaboration.* Kluwer Academic Publishers, Boston/Dordrecht/London

3 Bolloju, N., Khalifa, M., Turban, E. (2002) *Integrating knowledge management into enterprose envirnments for the next generation decision support.* Decision Support Systems

4 Chen, Z (2001) *Data mining and uncertain reasoning: an integrated approach.* John Wiley & Sons, inc. ISBN 0471388785

5 Delisle, S. (2005) *Integrating data mining and decision support via computational intelligence.* PMKD-DEXA Workshop, Philosophies and methodologies for knowledge discovery. Copenhagen, Denmark

6 Charest, M. Delisle, S. Cervantes, O, Shen, Y. (2006) *Intelligent data mining assistance via CBR and ontologies.* DEXA-PMKD Workshop, Poland

7 Charest, M. Delisle, S. (2006) *Ontology-guided intelligent data mining assistance : combining declarative and procedural knowledge.* 10th International Conference on Artificial Intelligence and Soft Computing

8 Charest, M., Delisle, S., Cervants, O., (2006) *Design considerations for a CBR-based intelligent data mining assistant.* MCSEAI, Agadir, Morocco

9 Uschold, M., Gruninger, M. (1996) *Ontologies: principles, methods and applications.* Knowledge Engineering Review, volume 11, number 2, pp93-155

10 The Protége project http://protege.stanford.edu/

11 Fayyad, U. Piatetsky-Shapiro, G. Smyth, P. (1996) *The KDD process for extracting useful knowledge from volumes of data.* Communications of the Acm November 1996, Vol 89, No.11

12 Fayyad, U., Piatetsky-Shapiro, G., Smyth, P. (1996) *From data mining to knowledge discovery : an overview.* In Advances in Knowledge Discovery and Data Mining. AAAI/MIT Press, Cambridge, Mass

13 CRISP-DM 1.0 (2000) *A step-by-step data mining guide.* www.crisp-dm.org

14 Geomez-Perez, A., Corcho, O., Fernandez-Lopez, M. (2005) *Ontological engineering: advanced information and knowledge processing.* Springer ISBN 1852335513

15 Gruber, T.R. (1993) *Towards principles for the design of ontologies used for knowledge sharing.* Formal Ontology in Conceptual Analysis and Knowledge Representation. Deventer, Netherlands, Kluwer

16 Berners-Lee, T., Hendler, J., Lassila, O. (2001) *The semantic web.* Science American

17 Bervers-Lee, T.(2000) *XML 2000 Semantic Web talk* http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html

18 Fensel, D., Hendler, J.A., Lieberman, H., Wahlster (Eds), W. (2003) *Spinning the Semantic Web : bringing the World Wide Web to its full potential.* MIT Press 2003. ISBN : 0-262-06232-1

19 Lassila, O., McGuinness, D. (2001) *The role of frame-based representation on the semantic web.* Electronic Transactions on Artificial Intelligence (ETAI)

20 Shadbolt, N., Berners-Lee, T., Hall, W. (2006) *The semantic web revisited.* IEEE Intelligent Systems

21 *OWL web ontology language overview.* (2004) http://www.w3.org/TR/2004/REC-owl-features-20040210/

22 *OWL web ontology language guide.* (2004) http://www.w3.org/TR/2004/REC-owl-guide-20040210/

23 *OWL web ontology language reference.* (2004) http://www.w3.org/TR/2004/REC-owl-ref-20040210/

24 Stojavonic, L. (2004). *Methods and tools for ontology evolution.* PhD thesis, University of Karlsruhe

25 Stojavonic, L., Maedche, A., Motik, B., Stojavonic, N. (2002) *User-driven ontology evolution managemen*t. European Conf. Knowledge Eng. and Management. (EKAW 2002)

26 Hasse, P., Sure, Y. (2004) *State-of-the-art on ontology evolution.* Institute AIFB, University of Karlstuhe. http://www.aifb.uni-karlsruhe.de/WBS/ysu/publications/SEKT-D3.1.1.b.pdf

27 The KAON project http://kaon.semanticweb.org/

28 Klein, M., Noy, N.F. (2003). *A component-based framework for ontology evolution.* Proceeding of Workshop on Ontology and Distributed System, IJCAI'03, Acapulco, Mexico

29 Klein, M. (2004). *Change management for distributed ontology.* PhD thesis, Vrije University, Amsterdam, Holland

30 Klein, M., Kiryakov, A., Ognyanov, D., Fensel, D. (2002) *Ontology versioning and change detection on the web.* In 13th International Conference on Knowledge Engineering and Knowledge Management. (EKAW 02) Siguenza, Spain

31 Noy, N., Klein, M. (2002) Prompdiff : a fixed-point algorithm for computing ontologie versions. In 18th National Conference on Artificial Intelligence AAAi 2002, Edmonton, Canada

32 Tsatsaronis, G., Pitkanen, R., Vazirgiannis, M. (2005) *Clustering for ontology evolution.* 29th Annual Conference of the German Classification Society, P167, 9-11 March, Magdeburg, Germany

33 Haase, P., Hotho, A., Schmidt-Thieme, L., Sure, Y. (2005) *Collaborative and usage-driven evolution of personal ontologies.* LWA Saarbrucken, Germany

34 Flouris, G., Plexousakis, D., Antoniou G. (2006) *Evolving ontology evolution.* SOFSEM 2006: Theory and Practice of Computer Science, Proceedings lecture notes in Computer Science 3831: p14-29, Germany

35 The DAMON Project www.icar.cnr.it/kgrid

36 Cannataro, M., Comito, C. (2003) *A data mining ontology for grid programming.* Workshop on semantics in peer-to-peer and grid computing. Budapest, Hungary

37 Bernstein, A., Hill, S., Provost, F. (2005) *Intelligent assistance for the data mining process: an ontology-based approach.* IEEE Transactions on Knowledge and Data Engineering

38 Euler, T., Scholz, M. (2004) *Using ontologies in a KDD workbench.* Workshop on knowledge discovery and ontologies. ECML/PKDD Conference, Pisa, Italy, September, 2004

39 Phillips, J., Buchanan, B. G. (2001) *Ontology-based knowledge discovery in databases.* International Conferences on Knowledge Capture. Victoria, Canada

40 Rennolls, K. (2005) *An intelligent framework (O-SS-E) for data mining, knowledge discovery and business intelligence.* Proceedings of 16th International Workshop on Database and Expert Systems Applications

41 Cannataro, M, Talia, D. (2003) *Knowledge grid: An architecture for distributed knowledge discovery.* CACM, Volume 46, No. 1 pp. 89-93

42 Tan, P., Steinbach, M., Kumar, V. (2006) *Introduction to data mining.* ISBN : 0-321-32136-7 http://www.aw-bc.com/computing

43 Witten, I.H., Frank, E. (2005) *Data mining: practical machine learning tools and techniques.* Second edition Morgan Kaufman Publishers

44 Kantardzic M. (2003) *Data mining: concepts, models, methods, and algorithms.* IEEE Press. ISBN: 0-471-22852-4

45 Han, J., Kamber, M.(2000) *Data mining : concepts and techniques.* Morgan Kaufmann Publishers

46 Noy, N.F., McGuinness, D.L. (2001) *Ontology development 101: a guide to creating your first ontology.* http://protege.stanford.edu/publications/ontology_development/ontology101.pdf

47 Nardi, D., Brachman, R.J. (2002) *An introduction to description logics.* In the Description Logic Handbook. Cambridge University Press

48 Racer manager http://www.sts.tu-harburg.de/~r.f.moeller/racer/

49 The Pellet project http://www.mindswap.org/2003/pellet/

50 Sirin, E., Parsia, B., Grau. B. C., Kalyanpur, A., Katz, Y. (2006) *Pellet: a practical OWL-DL reasoner.* Journal of Web Semantics

51 Baader, F., Sattler, U., Logica, S. (2000) *An overview of tableaux algorithms for Description Logics.* Studia Logica pp5-40

52 Klein, M., Fensel, D. (2001) *Ontology versioning for the Semantic Web.* Processing of the 1st International Semantic Web Working Symposium (SWWS). Stanford University, California, USA

Appendix

Ontology Evolution Plug-in User Manual

Ontology Evolution Tab is a new Protégé tab widget plug-in that allows you to update an existing Protégé Owl ontology or to create a new Protégé OWL ontology.

Ontology Evolution Tab is based on the OWL ontology language and works as an extension of Protégé OWL. The main purpose of Ontology Evolution Tab is to guide the user, especially the non-expert user to complete the basic common tasks of developing and updating an ontology step by step. Usually, when we create a class in an ontology just by defining its name, properties and restrictions using the "Class" widget provided by Protégé OWL, the consequent work is not totally completed---- the class is "empty" although its frame is well defined. We need to further create some instances to make the class more semantically meaningful. Although adding instances can be done in "Instance" widget, some problems do exist for the users who don't know Protégé well enough to follow all the steps of their task. The same problem exists when we create a property. For this end, the newly developed Ontology Evolution Tab aims at grouping and arranging the necessary steps of each basic task of ontology evolution. It can be used as a wizard to deal with the tasks about ontology classes, properties and instances such as "create", "delete" and "modify" from the very beginning (e.g. choose the task) to the end (e.g. edit the instance in the created class).

Ontology Evolution Tab integrates with the Protégé OWL plug-in. It inherits all the functions of OWL language from Protégé OWL plug-in. It allows you to use the same main menu provided by Protégé OWL plug-in; it also adapts the same "look and feel" of Protégé OWL plug-in so that you will feel easy and comfortable to use.

For the time being, the Ontology Evolution Tab mainly supports the creation and deletion tasks involving classes, properties and instances. The plug-in is semi-automatic; it requires the users' supervision and interaction. The users are supposed to know the basic knowledge about ontology development and Protégé OWL. They should also be able to identify the needs of the evolution task, select the appropriate task and understand each change they made in the activity series.

1. How to install

Ontology Evolution Tab is designed for Protégé 3.2 beta.

1. Put the Ontology Evolution folder in the following directory: <Protégé_installation_dir>/plugins (replacing the Protégé_installation_dir with your Protégé installation directory).
2. Run Protégé.
3. Choose "Project" from the main menu, then choose "Configure", in the Tab widget list, select OntologyEvolutionTab, click OK, the Ontology Evolution Tab will appear.

2. How to use

There are five sub tabs in Ontology Evolution Tab: Create Class, Delete Class, Create Property, Delete Property, and Create/Delete Individual. For each sub tab, some screenshots and examples are given from step to step to help you become familiar with its functions.

2.1 Create Class:

Create Class sub tab performs the class related task: it creates a class, defines its structure and creates instances in it. Create Class sub tab has two pages linked with *next* button and *back* button. Double click the proper button to turn the pages.

Step 1 of 5: Choose location

Two options are provided to create a class: *create subclass* button which creates a new class as subclass of the selected class, and *create sibling class* button which creates a new class at the same hierarchy level of the selected class.

Figure 1 presents the initial screenshot of the Create Class sub tab. Only the step 1 of the task is shown when firstly clicking this sub tab. When the location of the new class is chosen by clicking either *create subclass* button or *create sibling class* button, the second step can be shown on the screen. In this way, you can easily follow the steps one by one. Clicking the highlighted "next" button can switch to the next page of this sub tab.

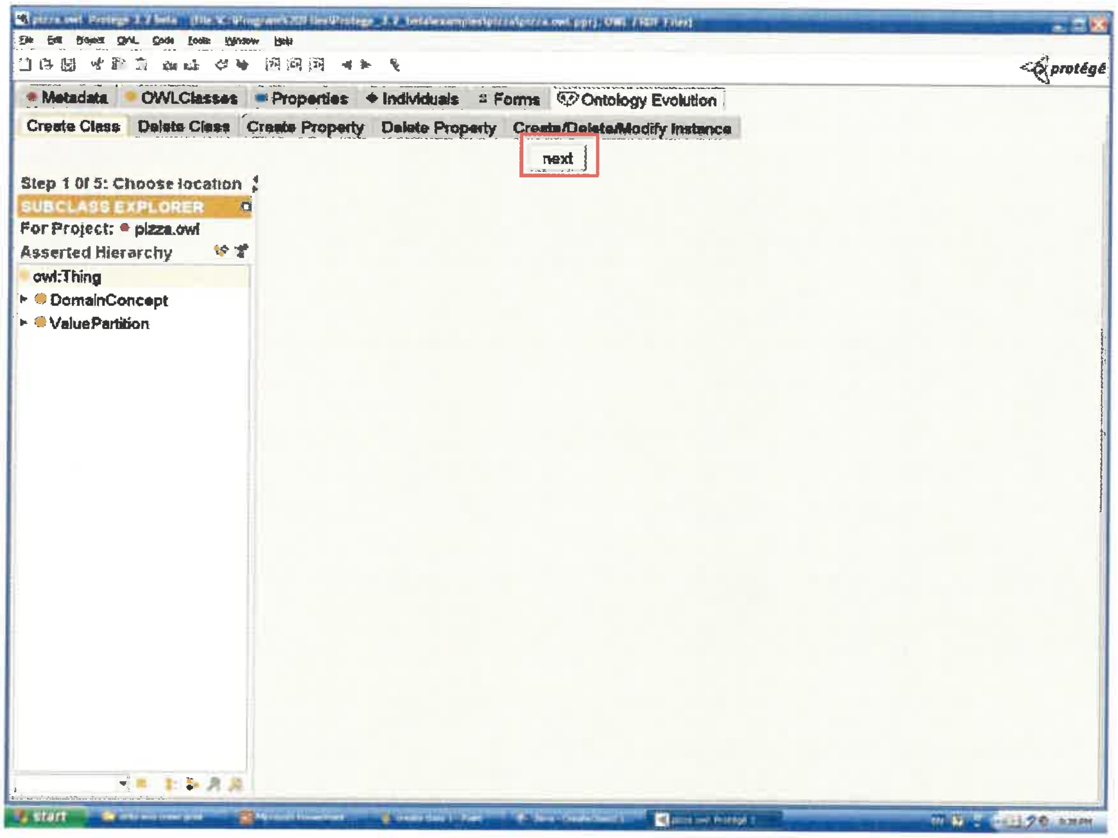


Figure 1 The initial screenshot of Create Class sub tab

Step 2 of 5: Edit class

You can edit the current created class in the class editor. The class editor contains two switchable views: logic view and property view. The logic view widget provides four widgets: *Class name* widget (to change class name), *Annotations* widget (to add, modify or remove class annotations), *Asserted Conditions* widget (to add, modify or remove asserted conditions) and *Disjoints* widget (to modify or remove the disjoint classes). The property view provides *Property and Restrictions* widget (to create, modify or delete the properties and restrictions), *Class name* widget, *Annotations* widget, and *Disjoints* widget.

In Figure 2, a new class is created by clicking *create subclass* button. The following figures present how a class can be edited within step 2. From the logic view, the class is firstly renamed as "Restaurant" in Figure 3, then all its sibling classes ("DomainConcept" and "ValuePartition" are added to its disjoint classes as indicated in Figure 4. Figure 5 shows the result of the disjoint classes of the class "Restaurant". In order to create or add properties to this class, we must switch the logic view to the property view. A new property of the class "restaurant" is created as shown in Figure 6. This property is named "sellsPizza", its domain is set as the class "restaurant", and its range is set as the class "NamedPizza". Then we'd like to add a necessary condition based on the property "sellsPizza". After switching back to the logic view, as presented in Figure 7, the necessary condition of the class "Restaurant" is set by defining the filler of the "sellsPizza" property as allValuesFrom the instances of the class "NamedPizza". Figure 8 shows the result of the added condition.

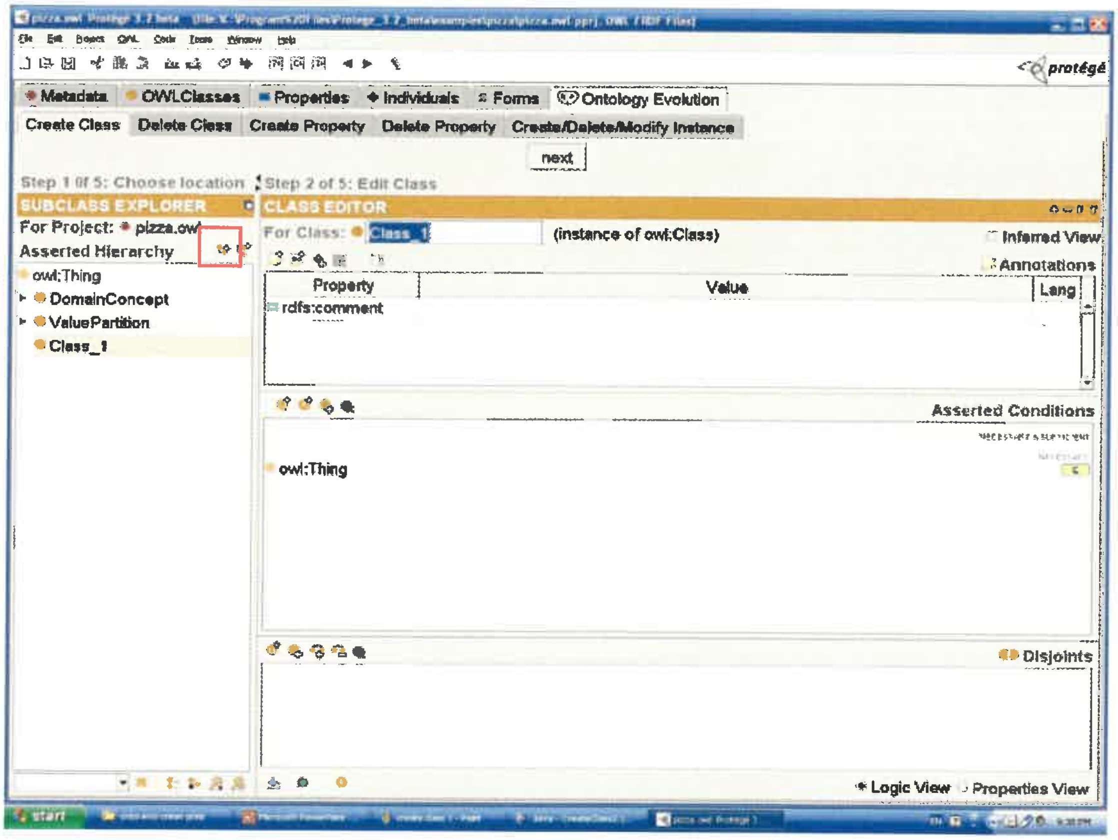


Figure 2 Create a new class

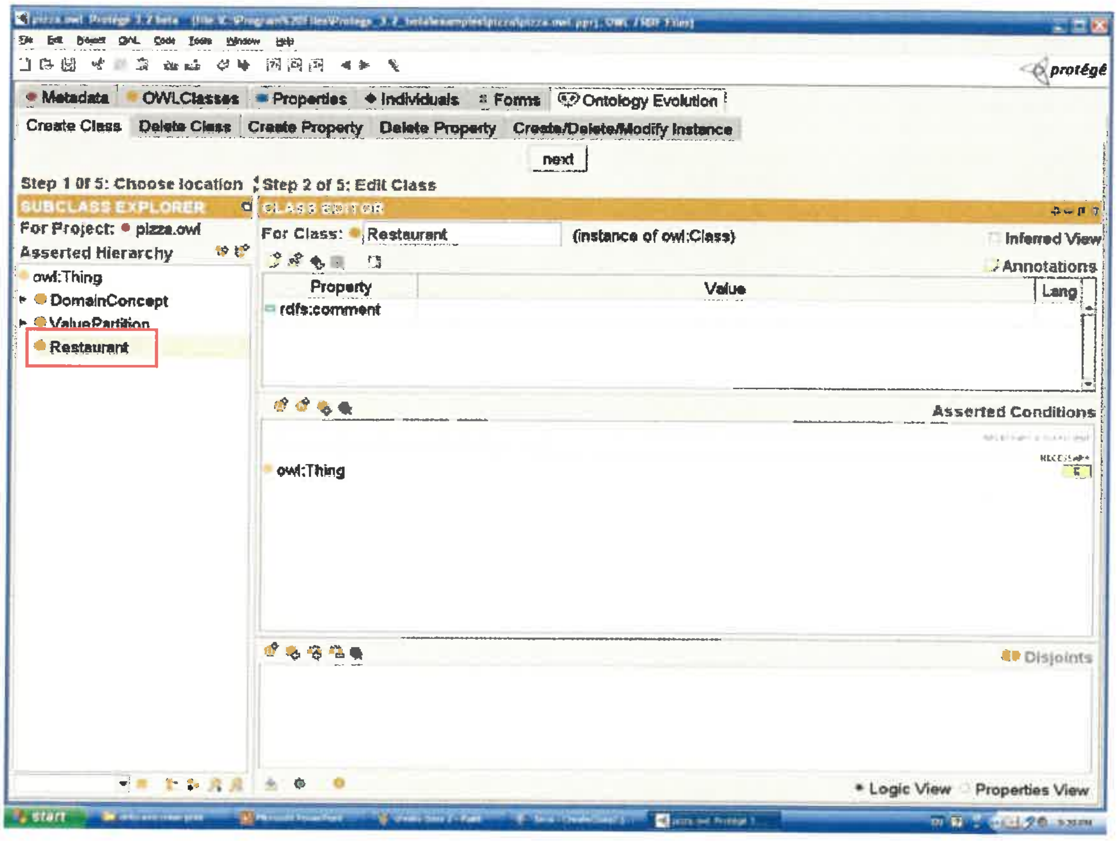
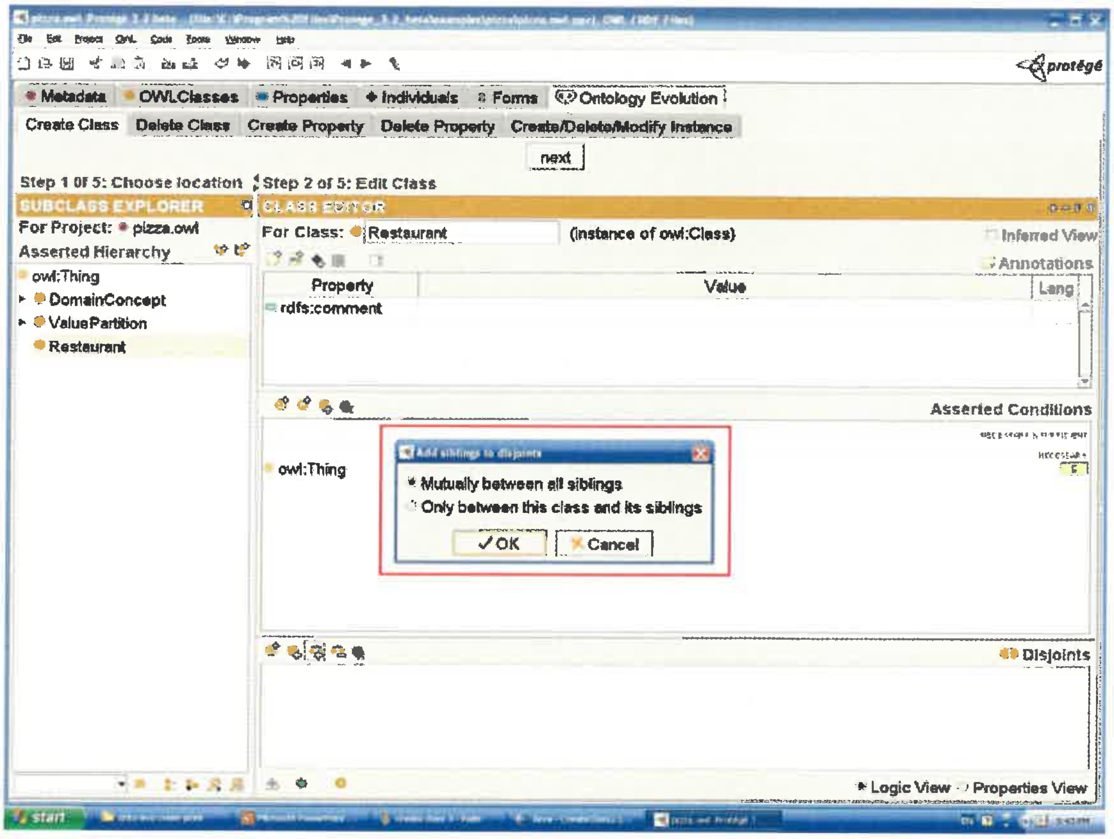


Figure 3 Rename the new class as "Restaurant"



[✓ Mutually between all siblings]
Only between this class and its siblings

Figure 4 Set disjoint classes of "Restaurant"

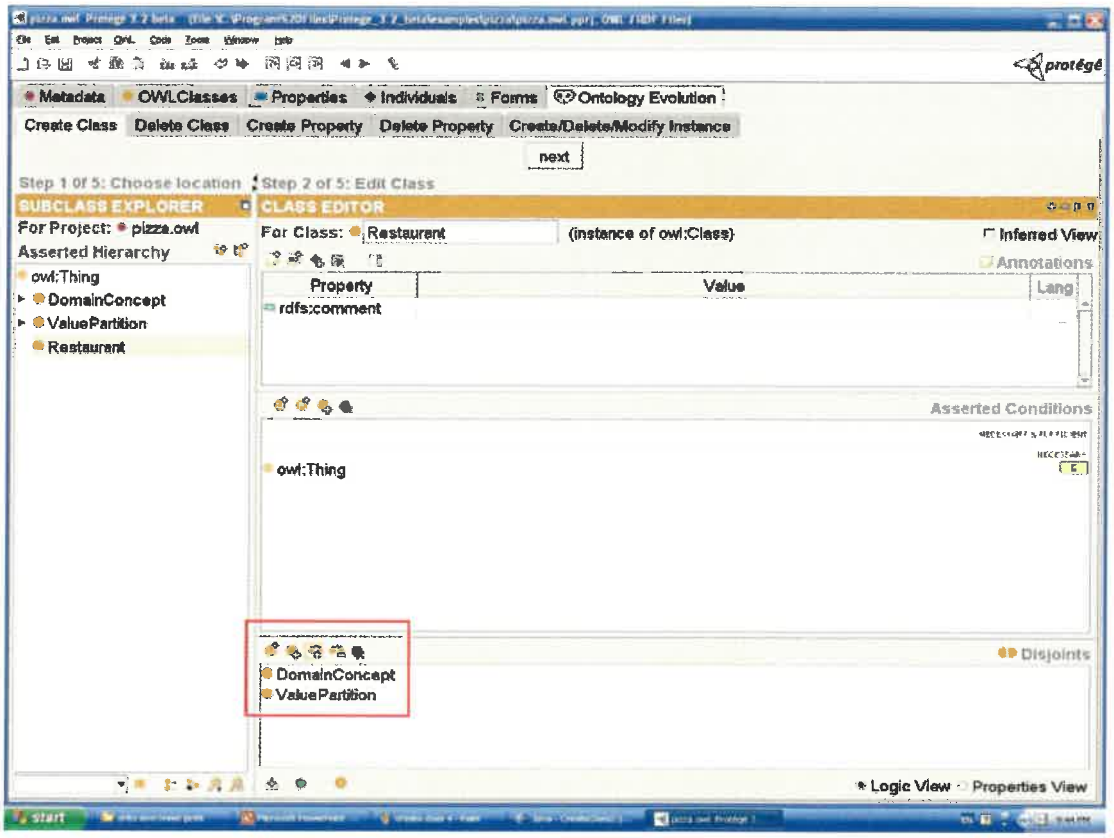


Figure 5 The results of disjoints of the class "Restaurant"

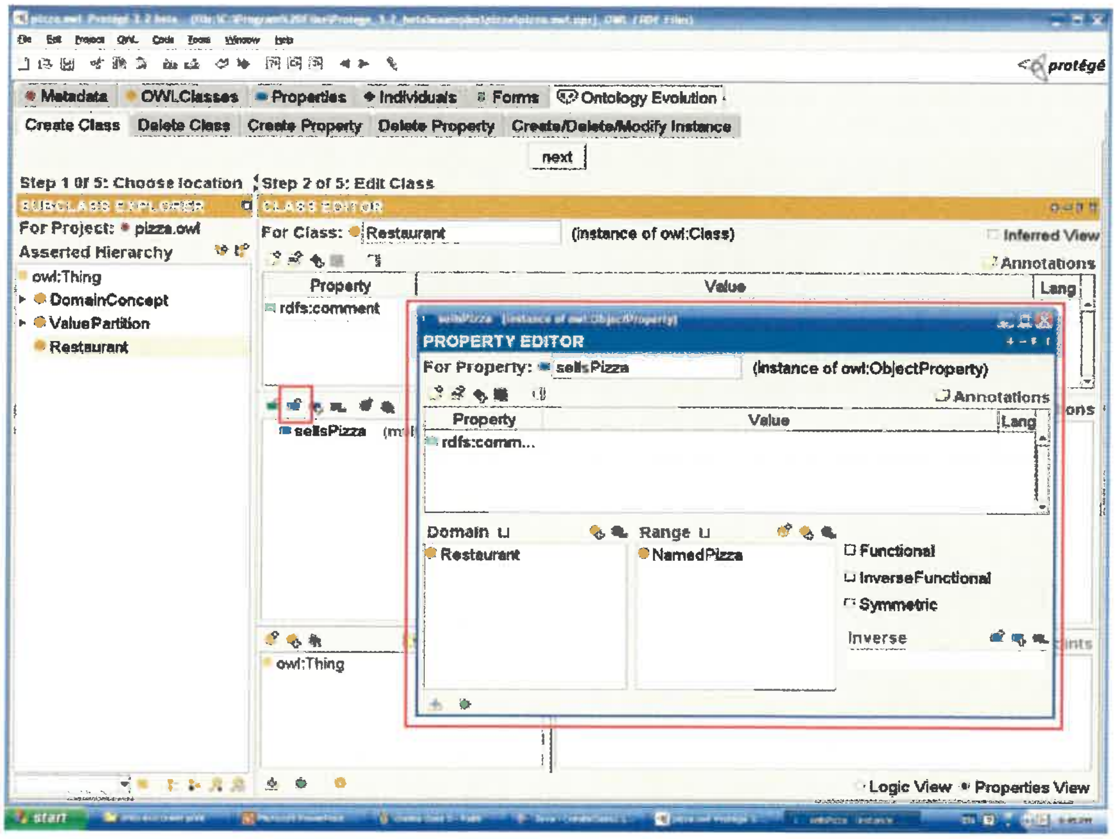


Figure 6 Create a new property “sellsPizza” for class “Restaurant”

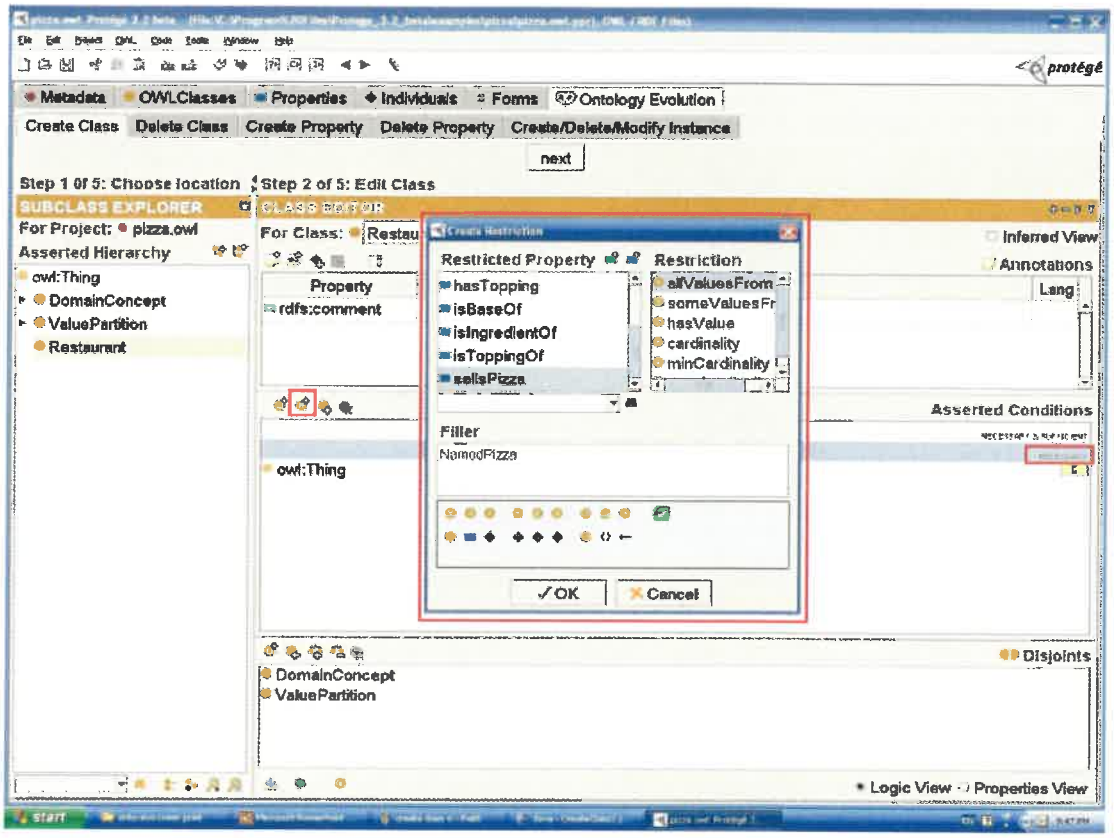


Figure 7 Set a necessary condition of “Restaurant”

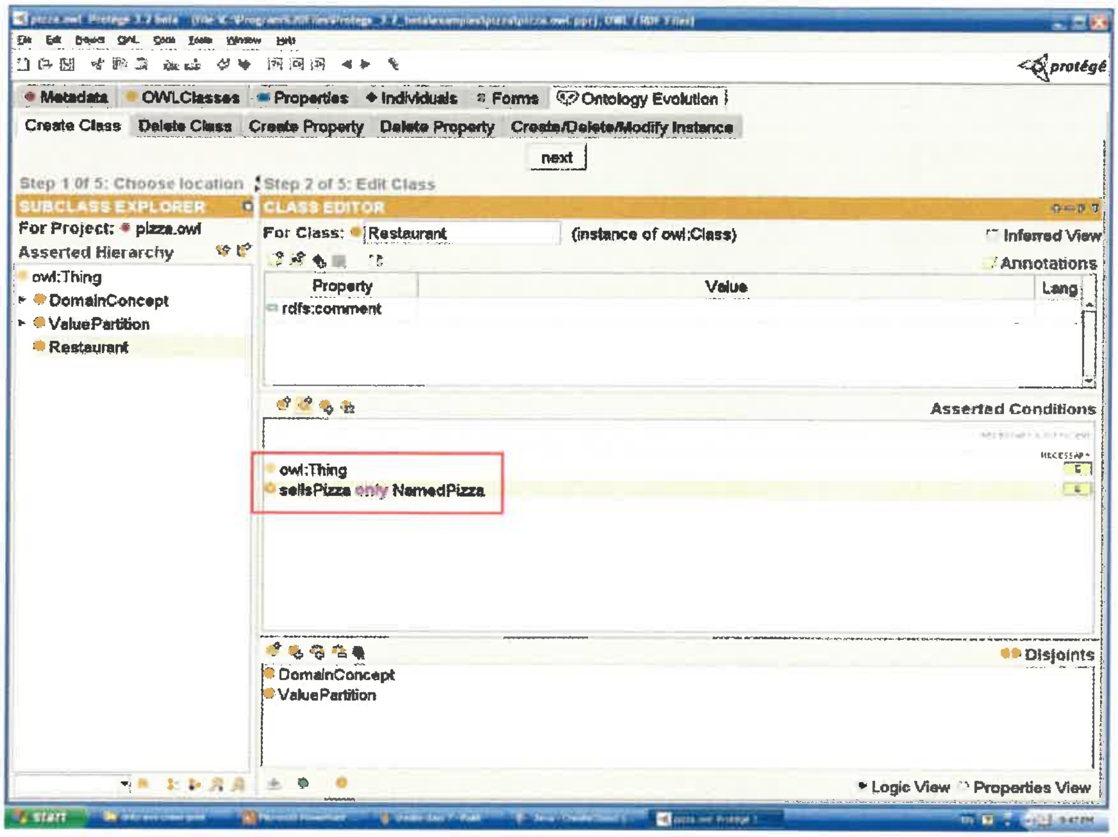


Figure 8 The result of Setting a necessary condition

Step 3 of 5: Check created class

The newly created class will be automatically selected from the class browser that allows you to verify whether it is the created class. Figure 9 gives the screenshot of this step. Clicking the "back" button can go back to the first page of this sub tab.

Note that only the selected class in class hierarchy panel (class browser) is allowed to perform the consequent actions such as creating instances.

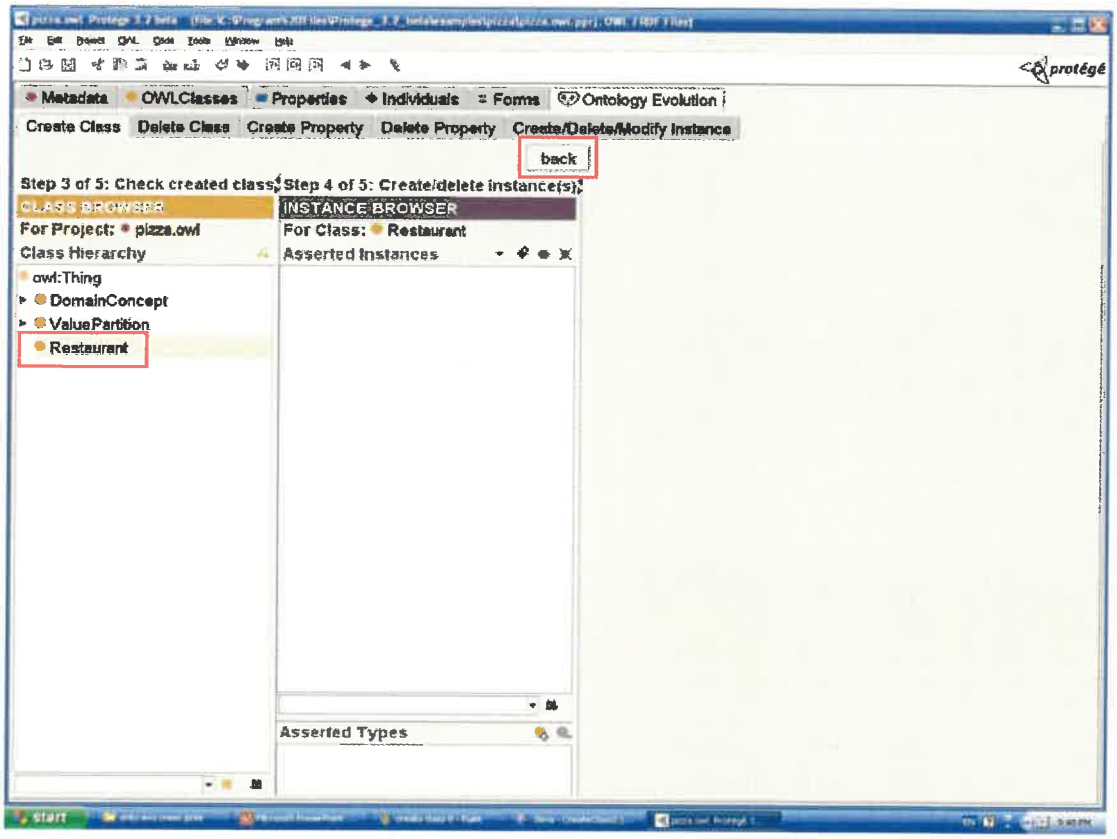


Figure 9 Check the created class "Restaurant"

Step 4 of 5: Create/Delete instance(s)

Once created and checked the class, you may want to create some instances in this class to make it more semantically meaningful. Click the *Create instance* button in the instance browser to create instance(s) of the selected class.

Step 5 of 5: Edit Instance(s)

You can select each time an instance from the instance browser to edit (add, delete or modify) the filler of its properties.

In the example presented in Figure 10, three instances of the class "Restaurant" have been crested: PizzaHut, PozzaKing and RoyalPalace. With the asserted condition for the property "sellsPizza" defined in step 2, (sellsPizza allValuesFrom namedPizza), when trying to fill the property "cellsPizza" of the instance "PizzaKing", the only allowed classes are the class "NamedPizza" and its subclasses. Note that there is no instance in the allowed classes; thus this property cannot be filled at this moment. It will be filled later.

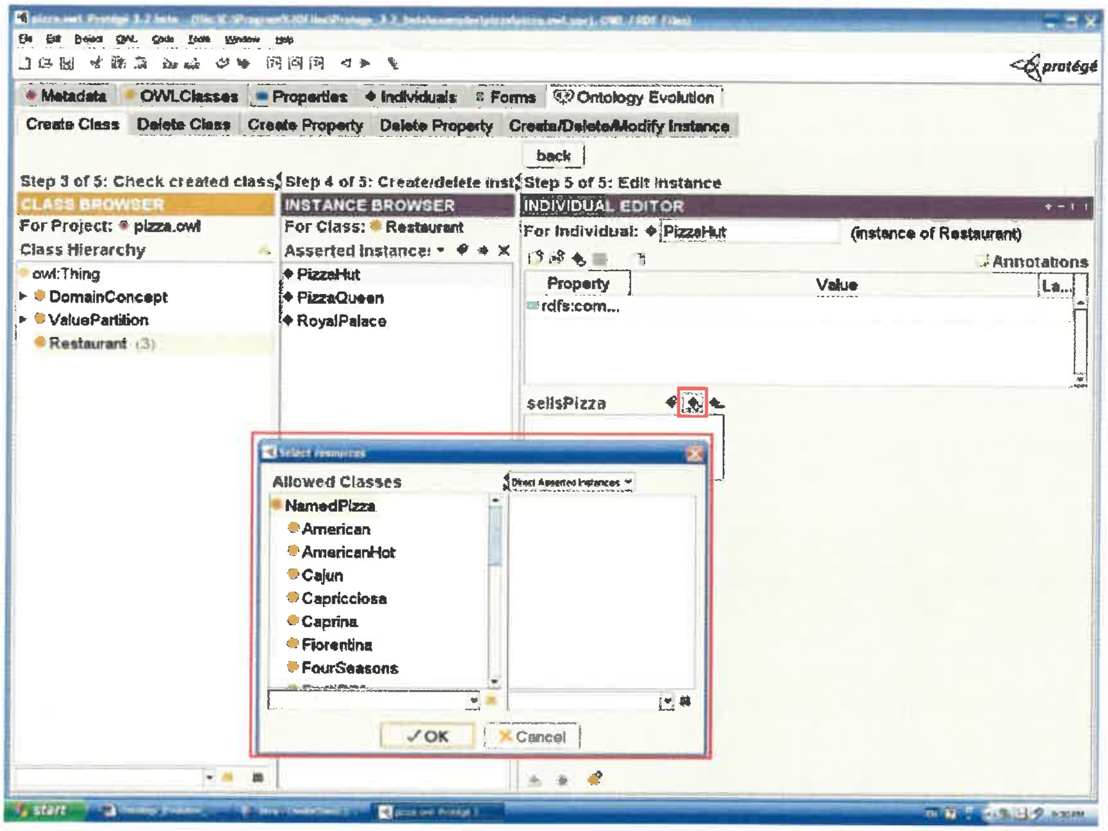


Figure 10 Create and edit instances for class "Restaurant"

2.2 Delete Class

Delete class sub tab deals with the deletion of the selected class. If the class has direct or indirect instances, it is not allowed to be deleted. You may go to the *Create/Delete/Modify Instance* sub tab to delete its instances firstly. Before deleting a class, make sure checking the usage of this class at the right part of the sub tab carefully since deleting a class especially a class with subclasses can be very dangerous: you may lose not only the deleted classes from the class hierarchy, but also the filler of some properties, and some restrictions!

Followings are some icons used in the usage panel:

≡ OWL equivalent class

∀ RDFS range

● OWL disjoint classes

⊑ RDF subclassof

🟦 OWL object property

🟩 OWL datatype property

The initial screenshot of Delete Class sub tab is shown in Figure 11.

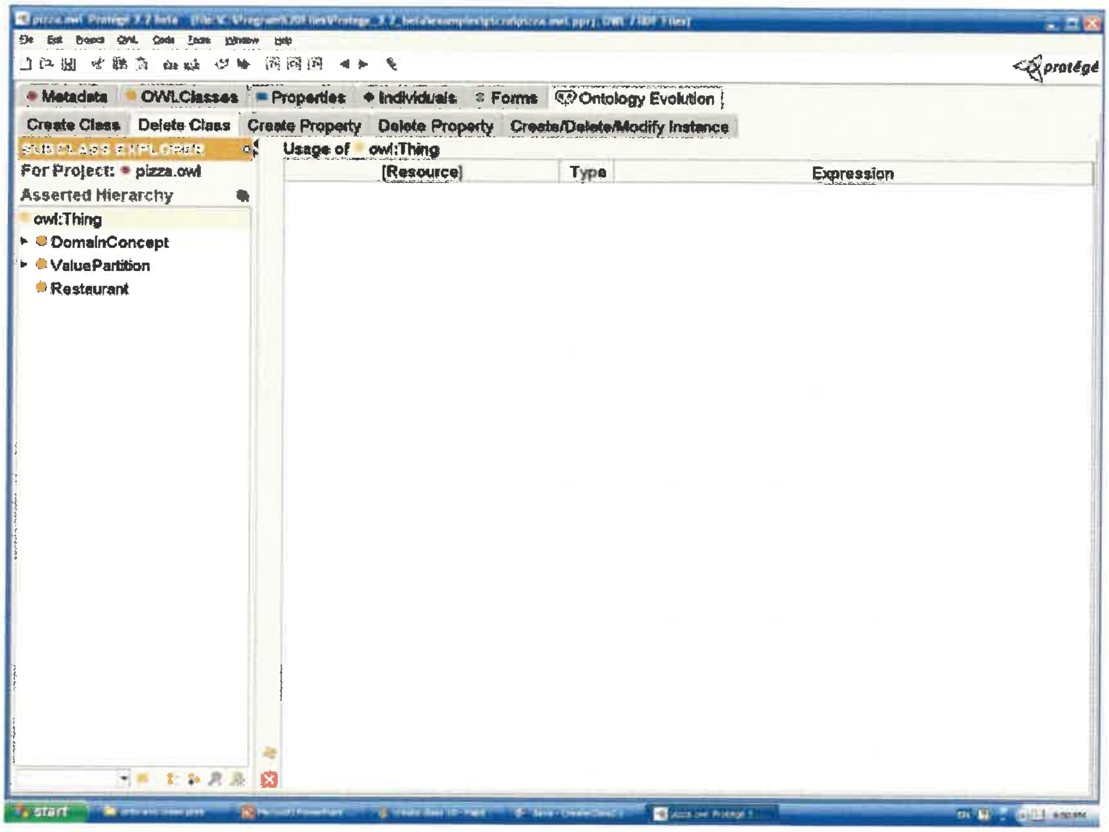
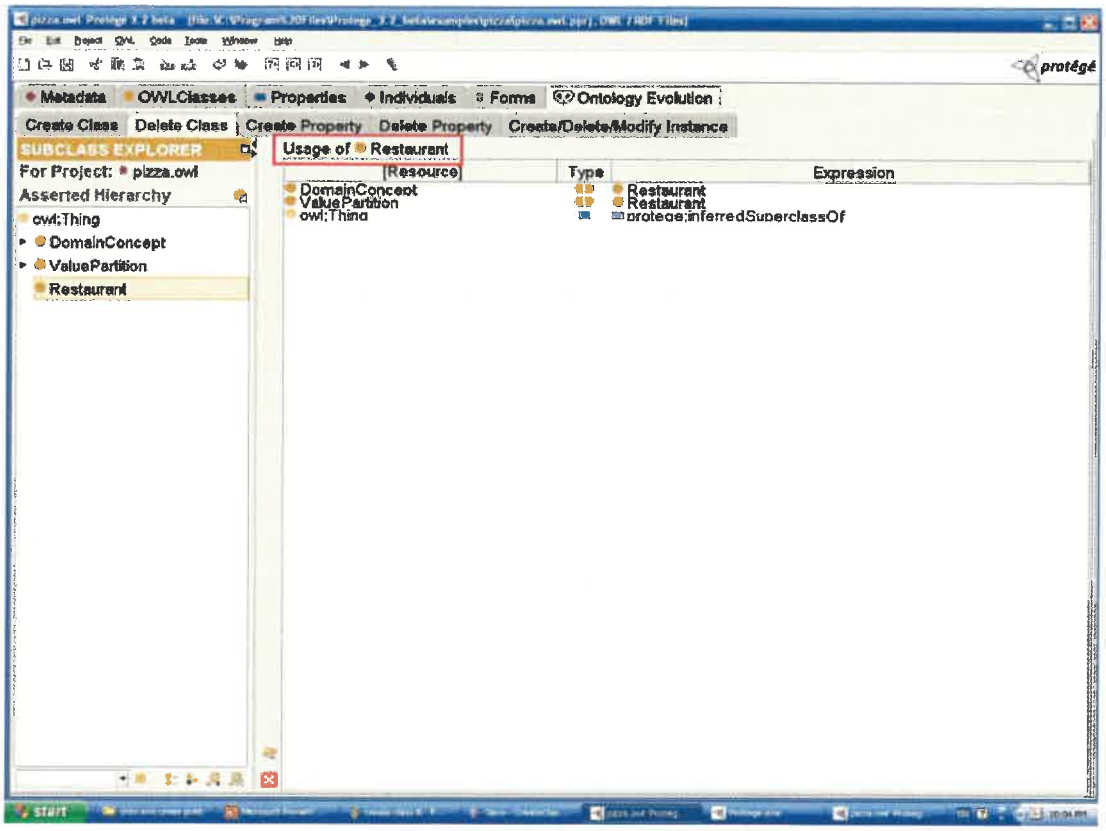


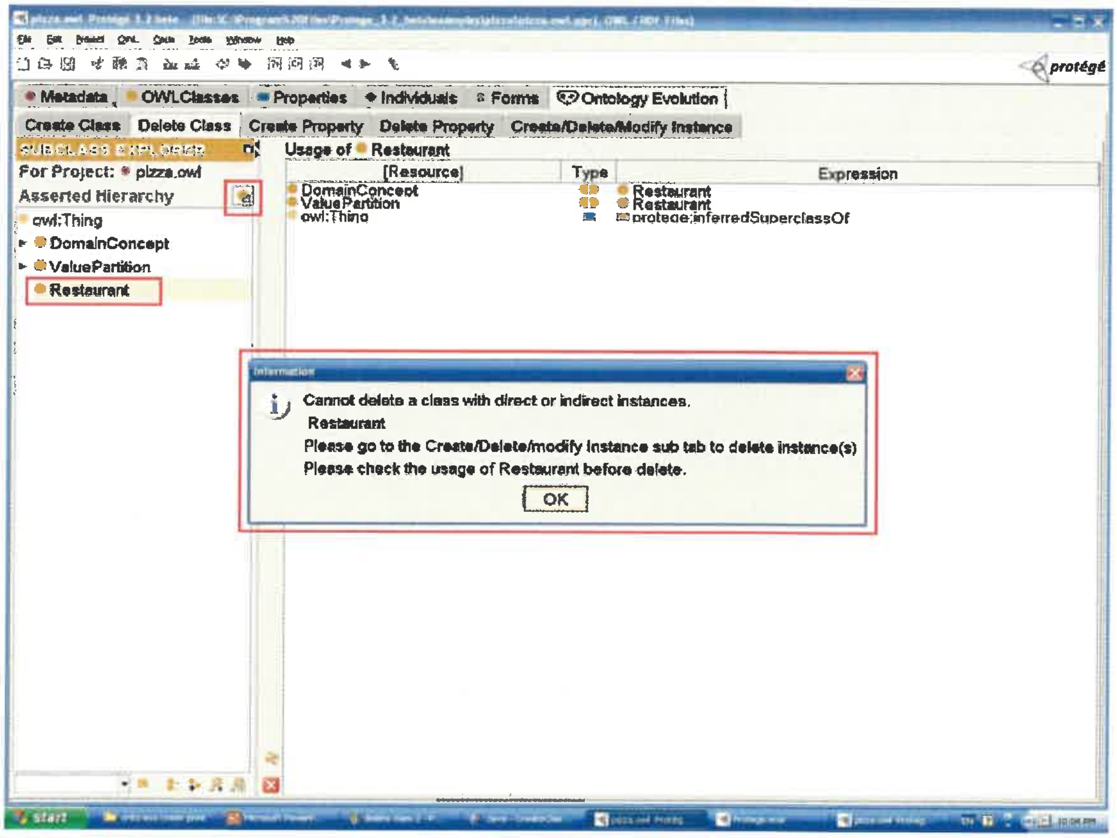
Figure 11 The initial screen shot of the Delete Class sub tab

When clicking the class “Restaurant”, the list of the usage of this class is presented at the right hand part of the screen, as shown in Figure 12. If we try to delete the class “Restaurant” by clicking the delete class button, a pop up window with the warning information will appear. The screenshot is presented in Figure 13. The class “Restaurant” cannot be deleted because it has three instances. We may go to the Create/Delete/modify instance sub tab to delete the instances firstly and go back to the Delete Class sub tab to delete this class. Although there are some inconveniences when deleting the classes with direct or indirect instances, this deletion mechanism really helps the user to protect the potential useful information from occasionally wrong operations. On the other hand, deleting a class without instance is direct and simple. Figure 14 shows that the class “Hot” is to be deleted. After checking the confirmation message and clicking “yes”, this class can be deleted. The result of deleting the class “Hot” is given in Figure 15. However, this operation can be undone by using the *undo* icon in Protégé main toolbar.



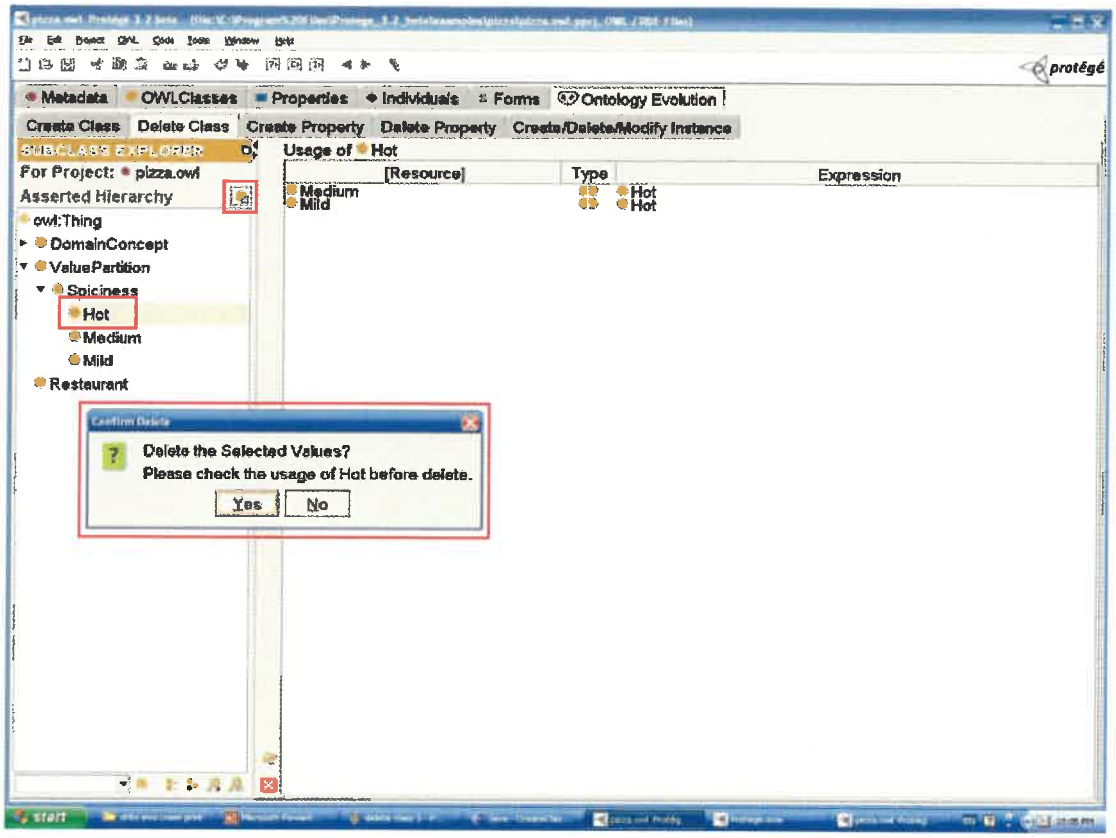
[Resource]	Type	Expression
DomainConcept		Restaurant
ValuePartition		Restaurant
owl:Thing		protege:inferredSuperclassOf

Figure 12 The usage of the class “Restaurant”



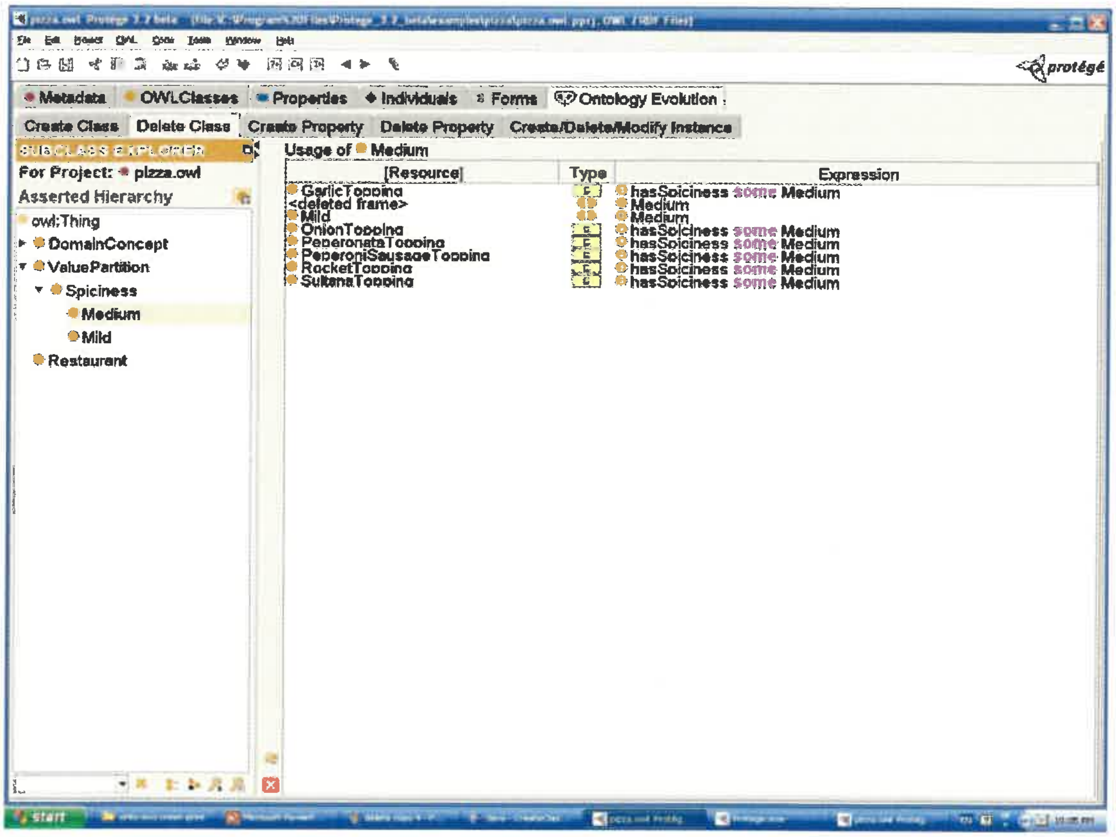
Cannot delete a class with direct or indirect instances.
Restaurant
Please go to the Create/Delete/modify Instance sub tab to delete instance(s)
Please check the usage of Restaurant before delete.

Figure 13 Try to delete the class "Restaurant"



Delete the Selected Values?
Please check the usage of Hot before delete.

Figure 14 Delete the class “Hot”



Usage of Medium

[Resource]	Type	Expression
GarlicTopping	⊑	hasSpiciness some Medium
<deleted frame>	● ●	Medium
Mild	● ●	Medium
OnionTopping	⊑	hasSpiciness some Medium
PeperonataTopping	⊑	hasSpiciness some Medium
PeperoniSausageTopping	⊑	hasSpiciness some Medium
RocketTopping	⊑	hasSpiciness some Medium
SultanaTopping	⊑	hasSpiciness some Medium

Figure 15 The result of deleting the class "Hot"

2.3 Create Property

Create property sub tab performs the property related task: it (1) creates a property, (2) defines its domain, range, inverse property and other characteristics; (3) finds the changed classes caused by the created property and (4) edits the instances of the changed classes.

Create property sub tab has two pages linked with next and back button. Double click the proper button to turn the pages.

Step 1 of 5: Choose property type and create.

There are four types of creating property: object property, datatype property, annotation property and all. Clicking the *create object property* button, *create datatype property* button, or *create annotation property* button will create a new property with its appropriated type; clicking *create subproperty* button will create a new property as the sub property of the selected property. Figure 16 shows the initial screenshot of this sub tab.

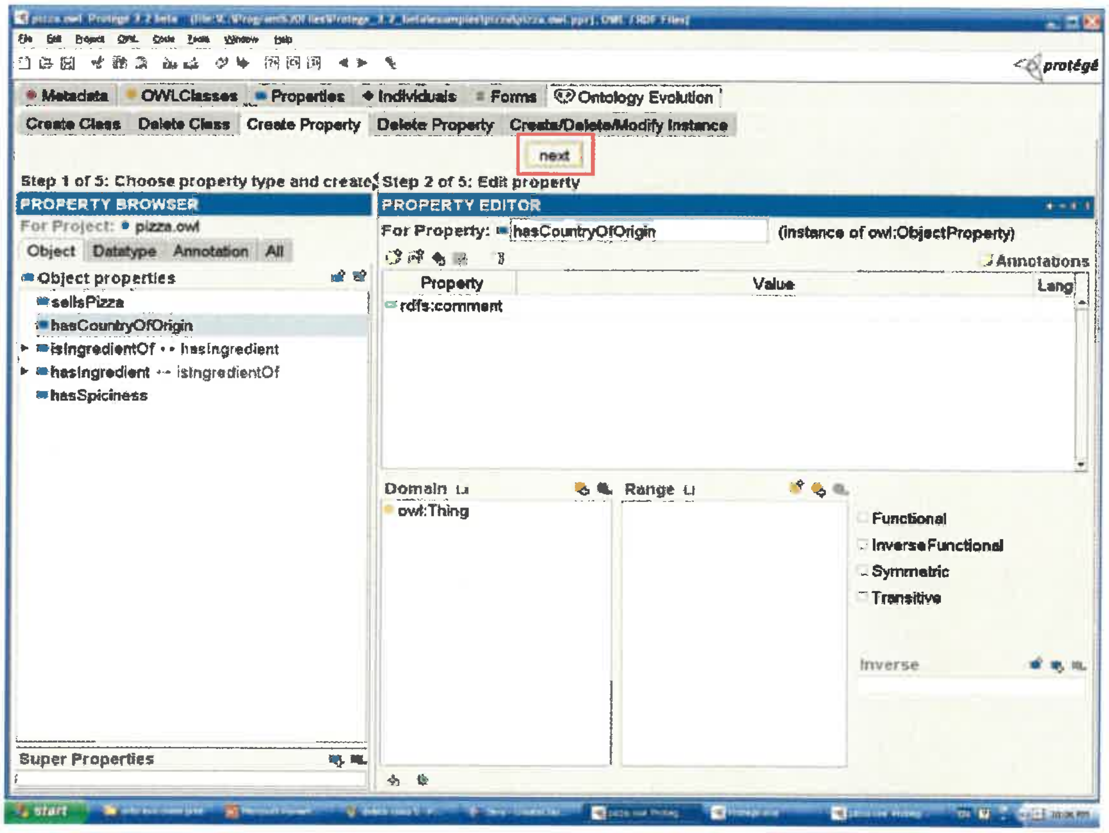


Figure 16 The initial screen shot of the Create Property sub tab

Step 2 of 5: Edit property.

The newly created property can be edited from the property editor. The annotations, domain, range, and some property characteristics (ex. Functional, reverse) can be defined in this step.

Figure 17 shows that a new property "makesPizza" is created. This property is created by clicking *create property* button in step 1, its domain is set as the class "Restaurant" and its range is set as the class "NamedPizza" in property editor.

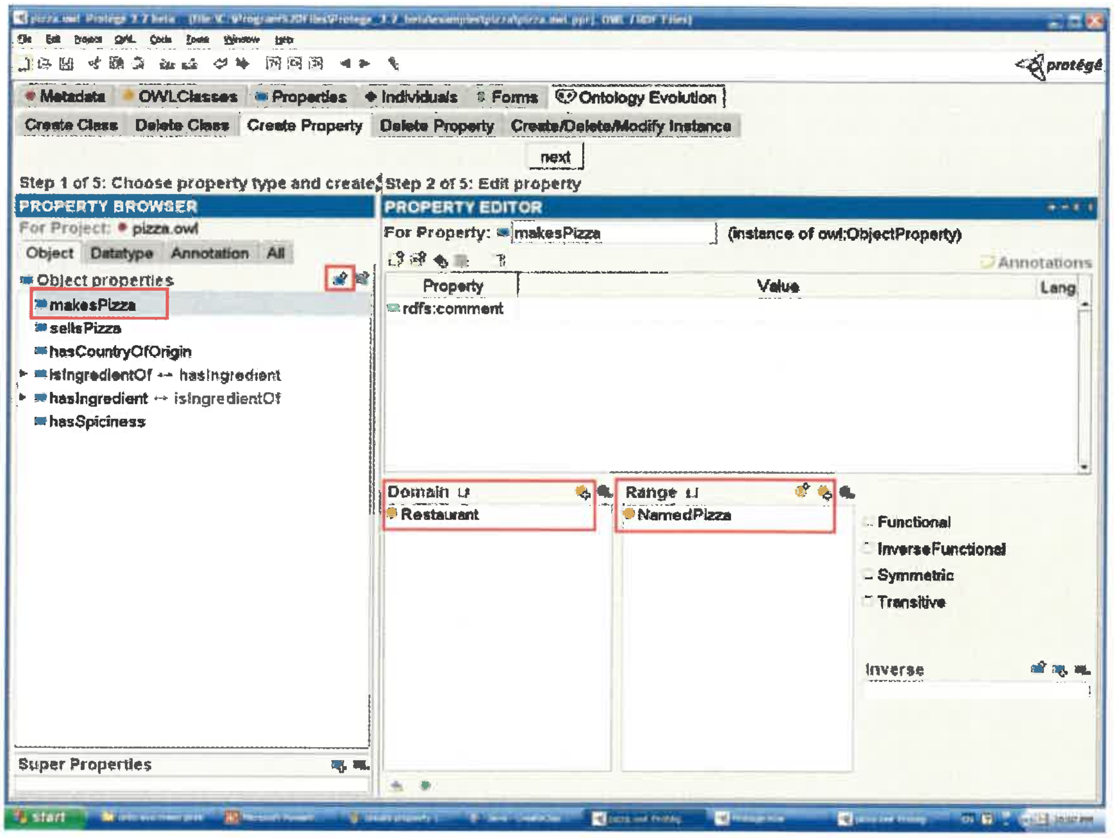


Figure 17 Create a property "makesPizza"

Step 3 of 5: Choose the changed class to edit its instances

The classes who have been added as the domain of the created property will be highlighted in red colour in the class hierarchy panel; these classes are considered as the changed classes since there is one more property added to their property list. This change can be reflected either from the *Property and Restrictions* widget in the Create Class sub tab or from the *Individual Editor* in the Create Property sub tab. The instances of the changed class also changed accordingly: they have one more property added and the filler of this property is still empty. The purpose of choosing the changed class is to edit its changed instances.

The screenshot of the changed classes is shown in Figure 18. The Create Property sub tab checks the changed classes automatically and highlights them with red colour. Note that the domain of the property "makesPizza" is defined as the class "Restaurant", thus, as indicated in Figure 18, the changed class is the class Restaurant, which is in red.

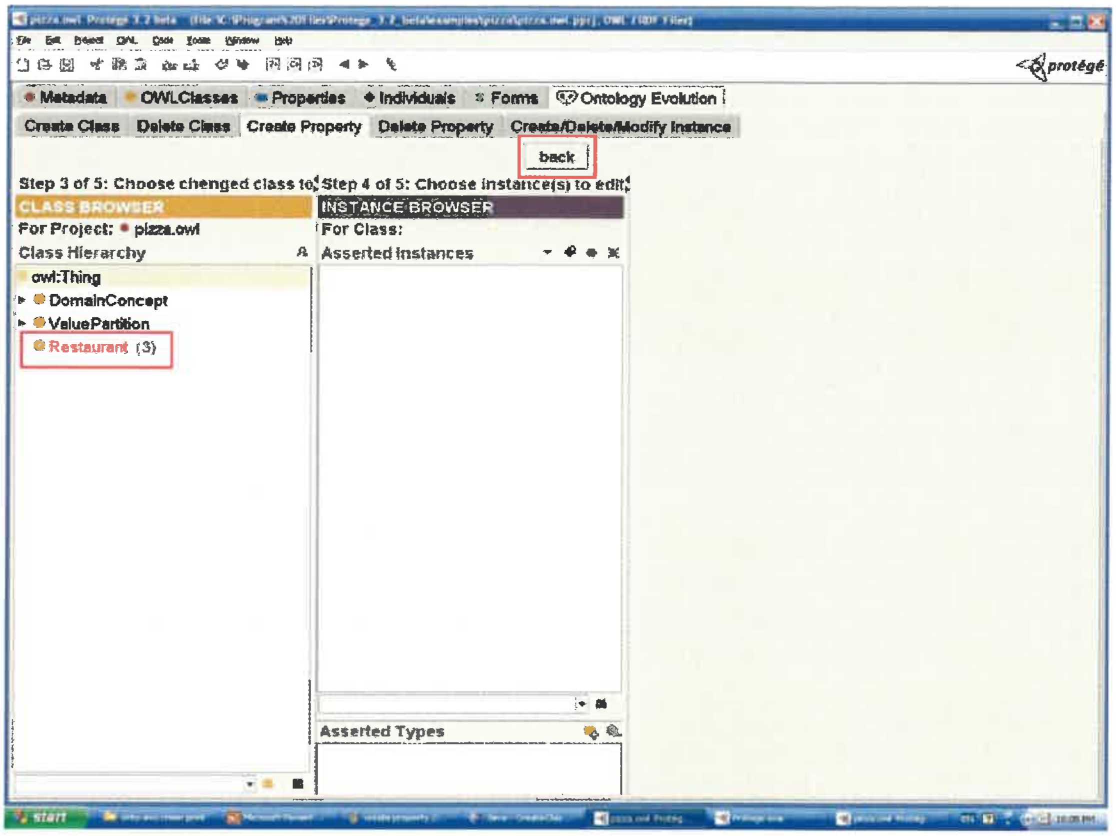


Figure 18 the changed class “Restaurant”

Step 4 of 5: Choose instance(s) to edit.

Select the instance one at a time from instance browser to edit.

Step 5 of 5: Edit instance(s)

The most common task for this step is to specify the filler of the created property by clicking the *Add...* button. If the range of this property is well defined in Step 2, the allowed classes of the property filler can be much specified.

Clicking the class “Restaurant”, its three instances appear in the instance browser, and the new property “makesPizza” is added for each of them in the instance editor, as shown in Figure 19. As the range of the property “maksPizza” is defined as the class “NamedPizza”, only the instances of the class “NamedPizza” or the instances of its sub classes are allowed as the filler of the property makesPizza.

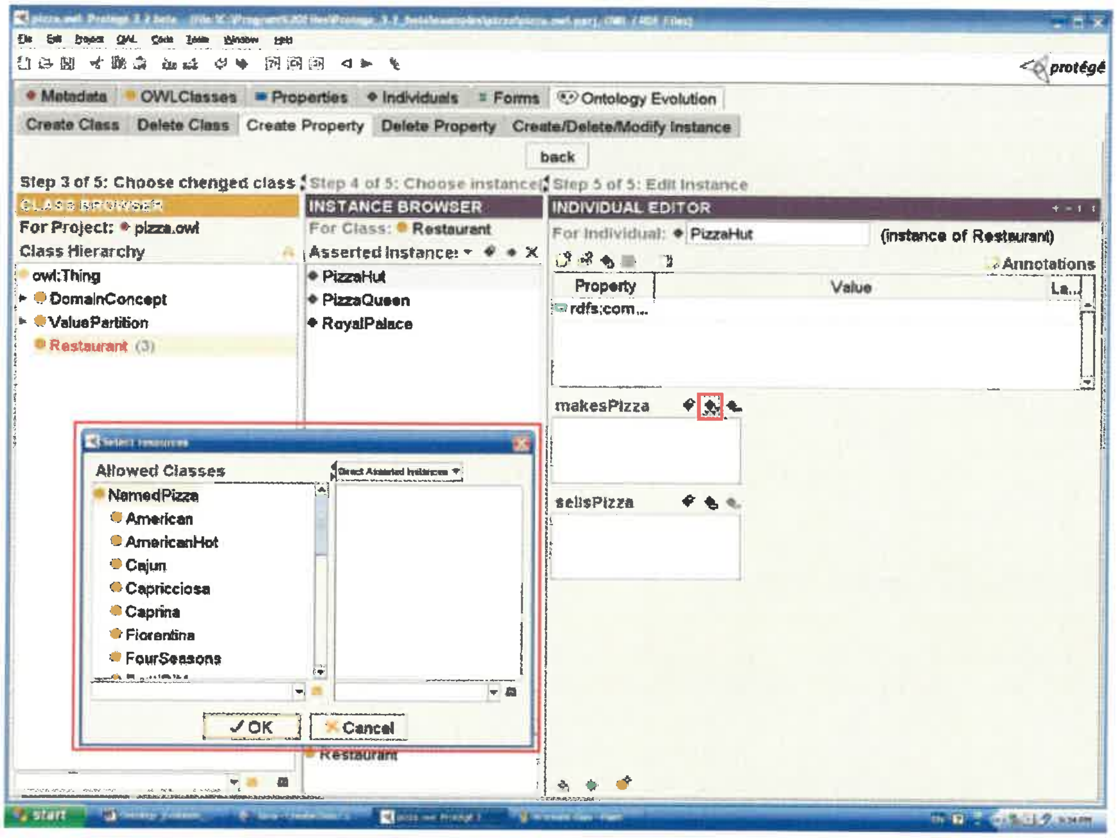


Figure 19 A new property is added to the instances of "Restaurant"

Here is another example illustrating the changed classes. Going back to the first page of this sub tab, we create another property "soldAtRestaurant", as shown in Figure 20. We then define its domain as the class "NamedPizza" and its range as the class "Restaurant". Also we set its inverse property as "sellsPizza". The function of the inverse property will be discussed later. As its domain is the class "NamedPizza", when going to the next page, we can see that the class "namedPizza" and all its sub classes become red as shown in Figure 21 since these classes are considered the changed classes. All the instances of the changed classes will have an additional new property "soldAtRestaurant" added. These instances can be edited further in the following steps in this sub tab.

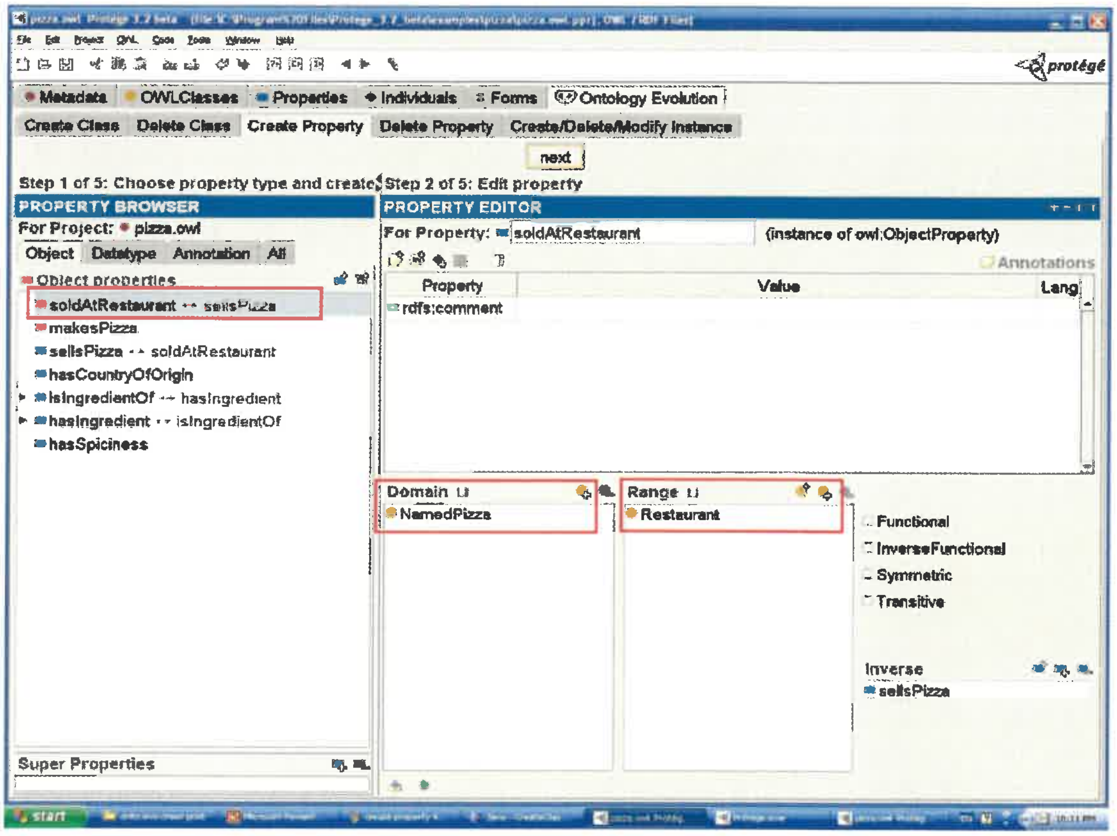


Figure 20 Create another property “soldAtRestaurant”

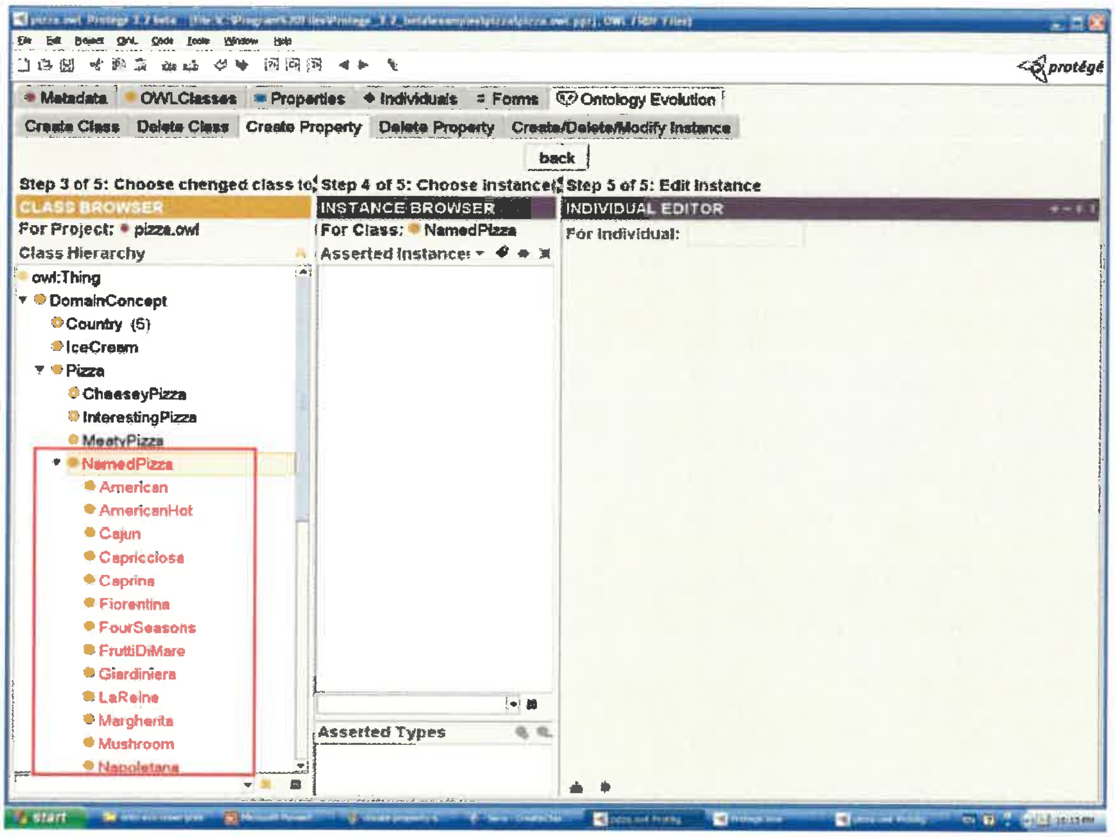
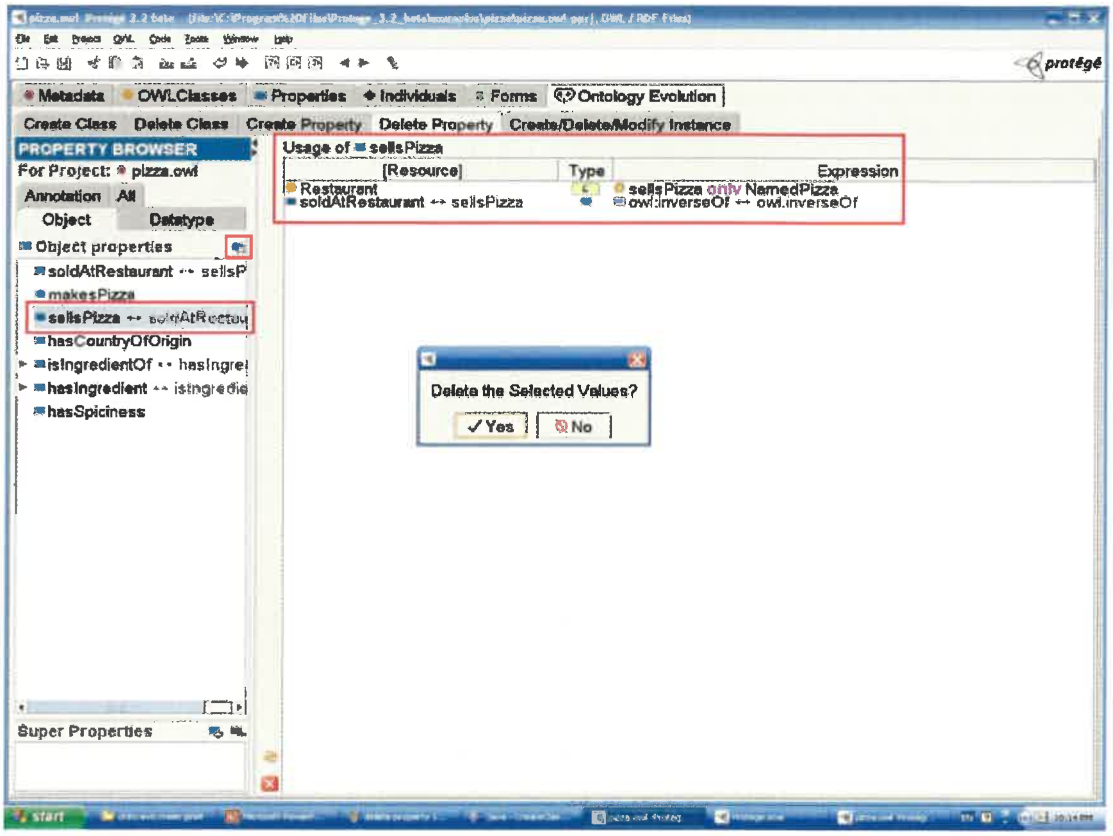


Figure 21 The changed classes of "soldAtRestaurant"

2.4 Delete Property

This sub tab deals with the deletion of the property. Before deleting a property, make sure to check the usage of this property. The icons used in the usage panel are almost same as those used in the Delete Class sub tab.

Figure 22 shows the screenshot of the Delete Property sub tab. The property "sellsPizza" is to be deleted, and the list of usage of this property is shown at the right hand part of the screen.



Usage of sellsPizza

[Resource]	Type	Expression
Restaurant		sellsPizza only NamedPizza
soldAtRestaurant ↔ sellsPizza		owl:inverseOf ↔ owl:inverseOf

Delete the Selected Values?

✓Yes | No

Figure 22 Delete the property “sellsPizza”

2.5 Create/Modify/Delete Instance

This sub tab provides the functions to create, modify or delete instance(s) from an existing class.

Step 1 of 3: Choose class

Choose the class from the class hierarchy panel in which you want to create, modify, or delete the instance(s). The initial screenshot of this sub tab is given in Figure 23.

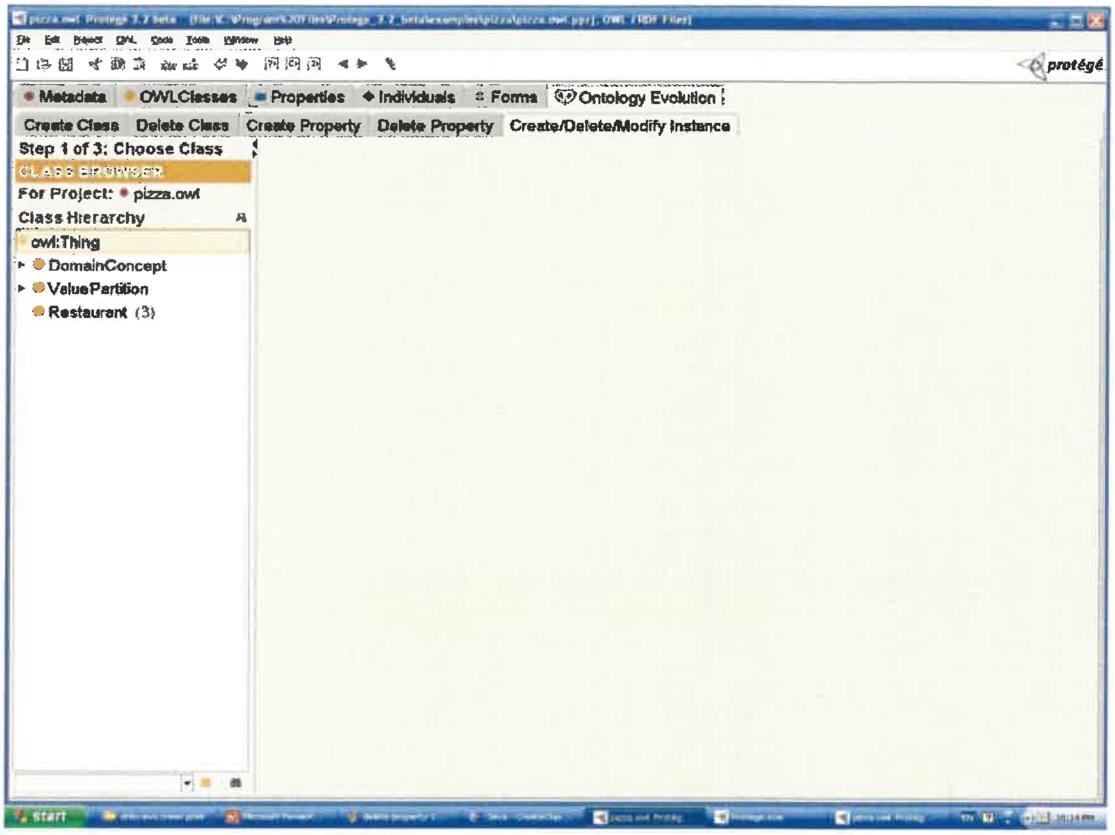


Figure 23 The initial screenshot of the Create/Delete/Modify instances sub tab

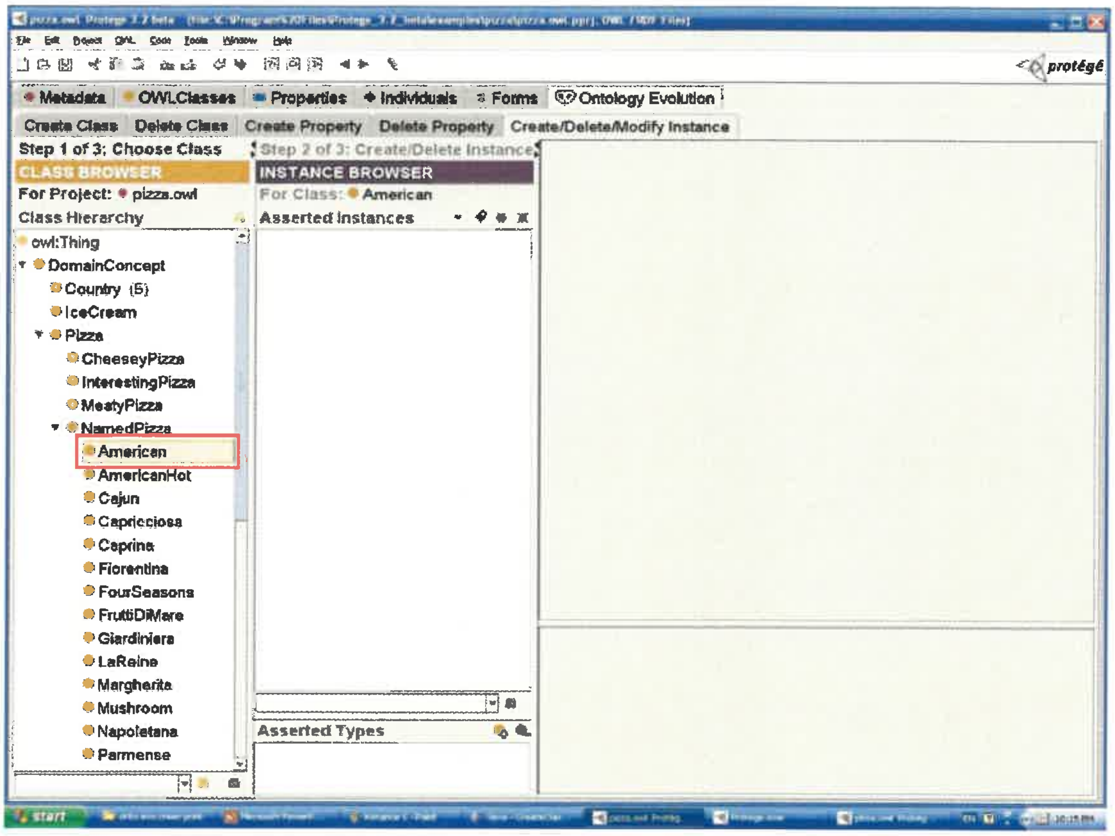


Figure 24 Choose the class "American" to add instances

Figure 24 indicates that the class "American" is chosen to add some instances. The instances of the class "American" will be used later in our example as the fillers of the property "makesPizza" and "sellsPizza".

Step 2 of 3: Create/Delete individual

Click the *Create instance* button to create a new instance; click the *Delete instance* button to delete an existing instance; or click the *Copy instance* to make a copy of an existing instance.

Step 3 of 3: Edit instance

When creating an instance, *edit instance* means to add, delete or modify the filler of each property of this instance. Note that only the name of the instance and the filler of the properties describing the instances can be modified.

Figure 25 shows that two instances are created for class "American": "American_a" and "American_b". The property "hasBase" and "hasTopping" of the instance "American_a" are automatically highlighted with the red rectangles by Protégé because the two properties must be filled according to the asserted conditions of the class "American". The asserted conditions of "American" inherited from its super class "Pizza" defines that "hasBase" must have at least one base that is the instance of the class "PizzaBase" and "HasTopping" must have at least one topping that is the instance of the class "PizzaTopping". Thus, if these two properties are not filled, it is considered inconsistent in Protégé. For the property "soldAtRestaurnat", two instances "PizzaHut" and "PizzaQueen" from the class "Restaurant" are selected as its fillers for "American_a", and one instance "PizzaHut" is selected for "American_b ".

Now, let edit the instances of the class "restaurant". Since the property "soldAtRestaurant" is set as the inverse property of "sellsPizza", we can see from Figure 26 that the property "sellsPizza" of the instance "PizzaHut" is filled automatically. This can also be proved from the usage of "PizzaHut". Figure 27 presents that another property "makesPizza" is filled with the instance "American_a" from the class "American".

Before deleting an instance, always check the usage of the instance in the usage panel. Figure 28 shows that an instance "PizzaQueen" is to be deleted. The result of the deletion operation is given in Figure 29.

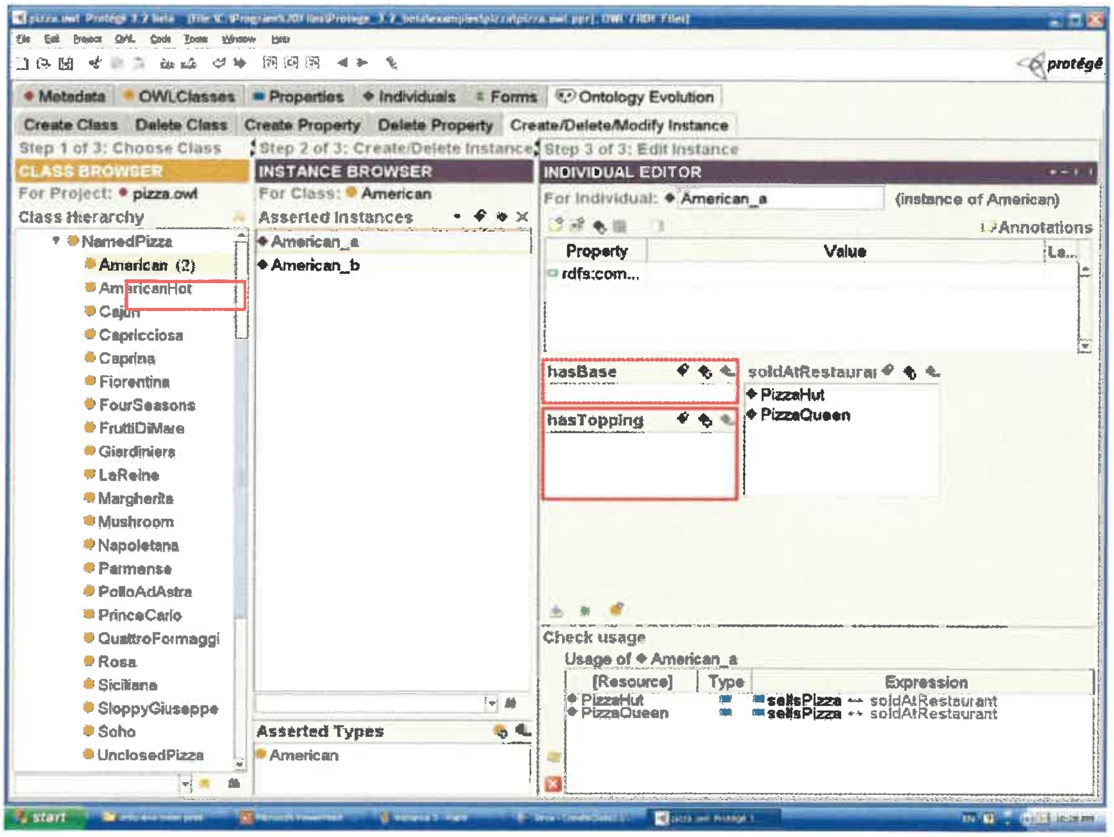
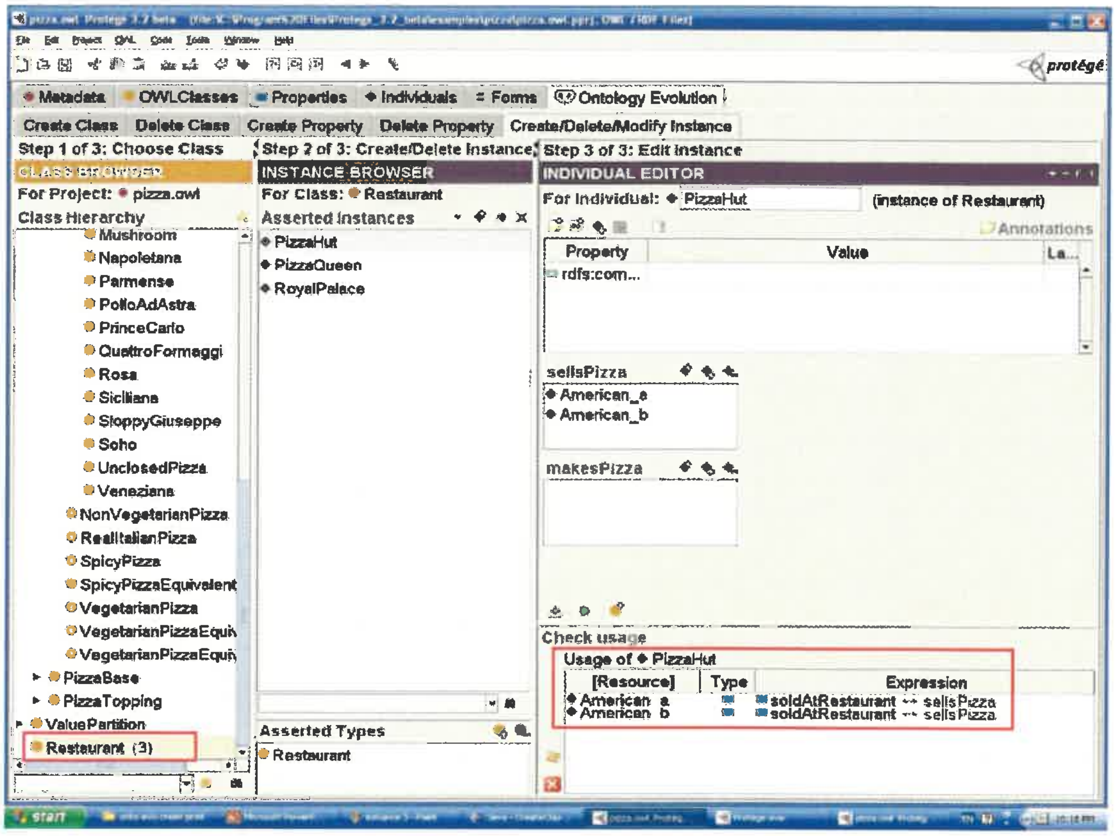


Figure 25 Create and edit the instances of "American"



For Project: pizza.owl

Class Hierarchy

- Mushroom
- Napoletana
- Parmense
- PolloAdAstra
- PrinceCarlo
- QuattroFormaggi
- Rosa
- Siciliana
- SloppyGiuseppe
- Soho
- UnclosedPizza
- Veneziana
- NonVegetarianPizza
- RealItalianPizza
- SpicyPizza
- SpicyPizzaEquivalent
- VegetarianPizza
- VegetarianPizzaEquiv
- VegetarianPizzaEquiv
- PizzaBase
- PizzaTopping
- ValuePartition
- Restaurant (3)

For Class: Restaurant

Asserted Instances

- PizzaHut
- PizzaQueen
- RoyalPalace

Asserted Types: Restaurant

For Individual: PizzaHut (instance of Restaurant)

Property	Value	La...
rdfs:com...		

sellsPizza

- American_a
- American_b

makesPizza

Usage of PizzaHut

[Resource]	Type	Expression
American_a		soldAtRestaurant ↔ sellsPizza
American_b		soldAtRestaurant ↔ sellsPizza

Figure 26 Edit the instances of the class "Restaurant"(1)

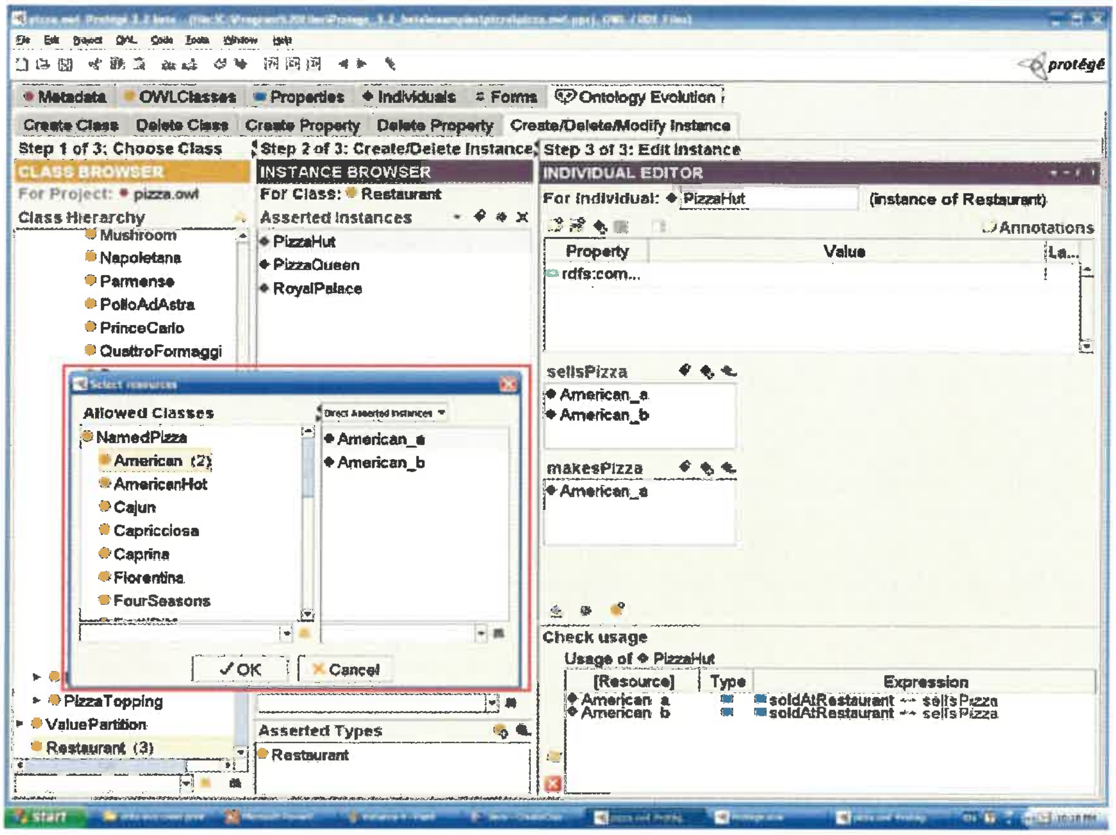


Figure 27 Edit the instances of the class "Restaurant"(2)

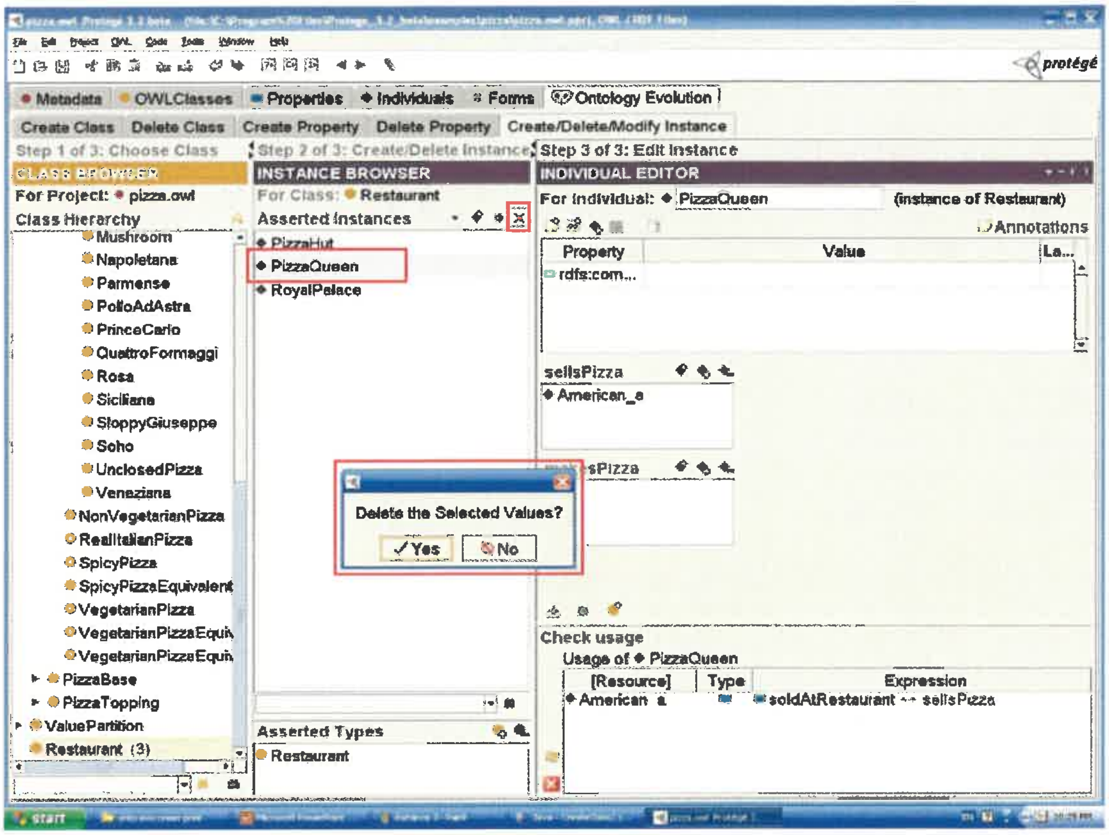
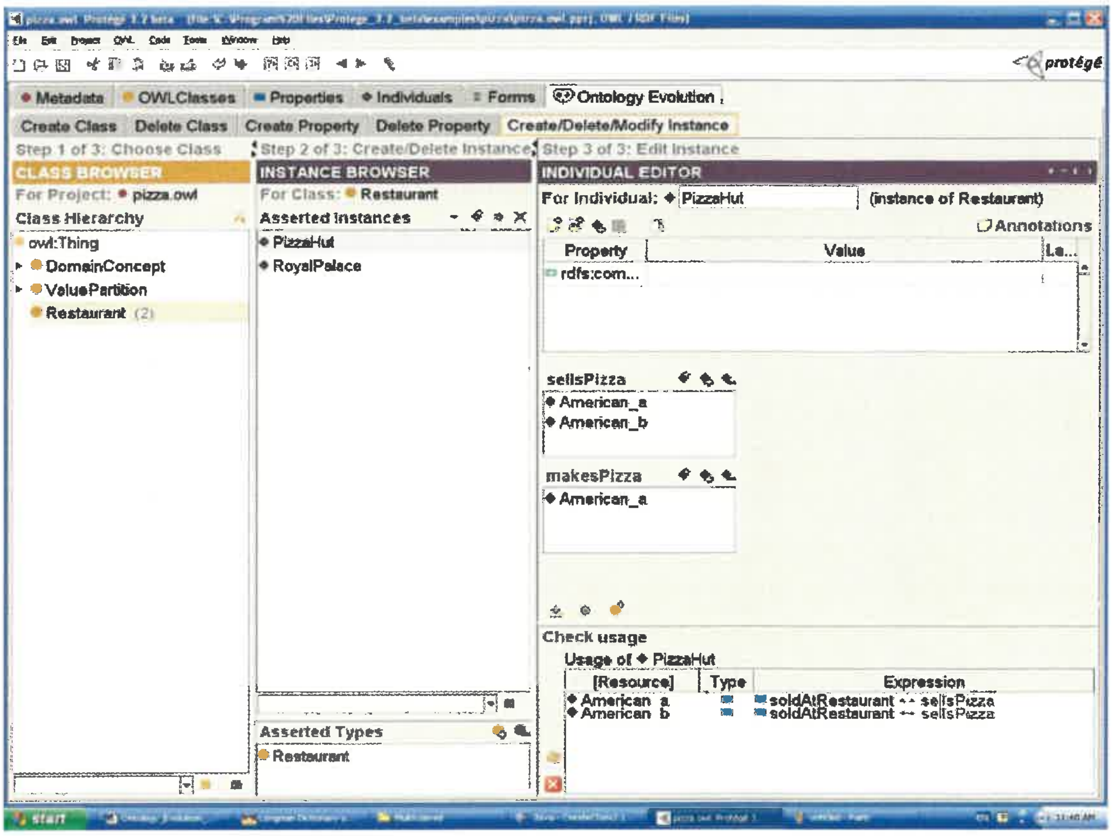


Figure 28 delete the instance “PizzaQueen” from the class “restaurant”



For Project: pizza.owl

owl:Thing
DomainConcept
ValuePartition
Restaurant (2)

For Class: Restaurant

PizzaHut
RoyalPalace

Restaurant

For Individual: PizzaHut (instance of Restaurant)

Property	Value	La...
rdfs:com...		

sellsPizza
American_a
American_b

makesPizza
American_a

Usage of PizzaHut

[Resource]	Type	Expression
American_a		soldAtRestaurant ↔ sellsPizza
American_b		soldAtRestaurant ↔ sellsPizza

Figure 29 The result of deleting the instance "PizzaQueen"