Ontology Modelling Notations for Software Engineering Knowledge Representation

Pornpit Wongthongtham[1], Elizabeth Chang[1] and Tharam Dillon[2]

[1]School of Information Systems, Curtin University of Technology, Perth, Western Australia,
e-mail : {Pornpit.Wongthongtham, Elizabeth.Chang}@cbs.curtin.edu.au
[2]Faculty of Information Technology, University of Technology Sydney, Australia
e-mail: tharam@it.uts.edu.au

***Abstract*—This paper is intended as an analysis of software engineering ontology and to highlight characteristics of software engineering ontology that represent important design considerations for the methodology of multi-site distributed software development. Multi-site teams use software engineering ontology to assist in defining information for the exchange of semantic project information. Some features and functions of the software engineering ontology are also provided. There is no standard graphical ontology representation in the literature so far. In this paper, we also present graphical notations of modelling software engineering ontology as an alternative formalism. The use of the formalism has made it a possible alternative to model the software engineering ontology. The main aim is not only to create a graphical representation making it easier to understand but, importantly, this model should be able to capture the semantic richness of the defined software engineering ontology.**

***Index Terms*—software engineering ontology, ontology modelling, ontology modelling notations.**

I. Introduction

As was seen in a number of papers [1-9], work on software engineering ontology has moved significantly towards solving multi-site distributed software development issues. Multi-site teams use software engineering ontology to assist in defining information for the exchange of semantic project information. Software engineering ontology poses important design consideration for the methodology of multi-site distributed software development.

This paper is structured as follows: Firstly, we offer a definition on software engineering ontology. Then we discuss on ontology modelling which we compares features of both software engineering ontology modelling and a well known object oriented model, Unified Modelling Language (UML). Our proposed notations for software engineering ontology modelling are discussed next. The last section provides a conclusion and future work.

II. Software Engineering Ontology

We have merged Gruber's [10], Borst's [11] and Studer's and colleagues [12] definitions of ontology as a basis for defining software engineering ontology. Hence, we define software engineering ontology as a formal, explicit specification of a shared conceptualisation in the domain of software engineering. The software engineering ontology is machine-understandable enabling better communication over software engineering domain knowledge between humans and machines. The type of software engineering concepts used, and their constraints used are explicitly defined. Software engineering ontology standardises and formalises the meaning of terms in software engineering through its concepts. The software engineering ontology specifies conceptual knowledge of software engineering which is public and accepted by a group of software engineers. An abstract model identifying the involved software engineering concepts implies the software engineering ontology.

The whole set of software engineering concepts are transformed into software engineering ontology as domain knowledge. In each software engineering project there will be project data or project agreement. Project data specially meet a particular project need and is required with the software engineering ontology to define instance knowledge. The instance knowledge varies depending on its use for a particular project and is literally defined according to domain knowledge in the software engineering ontology. Once all domain knowledge and instance knowledge are created it is available to be shared among project team members through the internet. All team members, regardless of where they are, can query the semantically linked project data and use it as the common communication and knowledge basis of raising discussion matters, questions, analysing problems, proposing revisions or designing solutions and the like.

III. Ontology Modelling

Various formalisms have been developed for modelling ontologies notably the Knowledge Interchange Format (KIF) [13] and knowledge representation languages descended from KL-ONE [14]. However, these representations are little success outside Artificial Intelligence (AI) research laboratories [15, 16] and require a steep learning curve. KIF provides a Lisp-like syntax to express sentences of first order predicate logic and descendants of KL-ONE includes description logics or terminological logics provide a formal characterisation of the representation [17]. Traditional AI knowledge representations have a linear syntax but no standard graphical representation. We present graphical notations of an ontology modelling ontology as an alternative formalism. It is the most appropriate representation to model the software engineering ontology.

Comparison of ontology modelling and UML is given here. We discuss the features the ontology modelling and the UML have in common and then the features have in on-

1-4244-0470-3/07/$20.00 ©2007 IEEE

tology modelling but not in UML.

UML defines several types of diagram i.e. UML activity, UML collaboration, UML component, UML deployment, UML sequence, UML state chart, UML class and object and UML use case. All these diagrams can be used to model the static and dynamic behaviour of a system. Static model like UML class diagram has some similarities with ontology and can be used to show the ontology classes and their relationships to model software engineering ontology. In an UML class diagram, UML classes are represented by boxes with three parts. The first part contains the name of the UML class. The second part contains each attribute of the UML class specified orderly by visibility, name and type. The third part contains each operation of the UML class specified orderly by visibility, name, argument list and return type. For purposes of representing software engineering ontology, UML class can be considered to the ontology class with UML class attribute is more or less ontology class property. However, operations are not needed in ontology modelling.

Between UML classes there are several types of relationship i.e. generalisation, association, aggregation, dependency and composition. The only relationships the UML and ontology modelling have in common are generalisation and, more or less, association. In ontology modelling, we do not make aggregation relationships, dependency relationships or composition relationships. Association relationships are similar to object properties which may be annotated with a multiplicity of indicators giving a range of numbers denoting how many relations can be associated. Also an arrowhead and different line are used to specify property characteristics. Association class; that is an UML class, attached to an association can also be adopted for use in modelling ontology. It is used for relations that require more details or properties than just their name and characteristics.

In UML, there is a rectangle with folded corners called notes, pieces of text to provide informal clarification. In ontology modelling, the notes can be used to attach ontology model elements to annotate clarification, for example, classes, and properties to provide the elements definition.

Representing instances of ontology can be adopted from an UML object. It depicts instances, links between instances or instances of the relationship that hold among the linked instances' respective classes. For the class instance, the class of each instance must be specified and the instance must be named. For the property instance, the class (if it exists), the property of each instance and the values of the instance must be shown explicitly.

UML supports a fixed defined extent for an UML class called enumeration; ontology does so, called oneOf.

In ontology, there is a universal class, class Thing, whose extent is all instances in ontology model and all classes are subclass of the class Thing. This is outside the system in an UML model.

Unlike UML, in ontology permits a class to be defined as the set of instances which satisfy a restriction expression. The expressions can be a Boolean combination of other classes e.g. intersection classes, union classes, disjoint classes, decomposition classes or property restrictions e.g. allValueFrom, hasValue.

The way of representing ontology properties is different from presenting attributes of the UML class. This is because ontology has more to specify in the property like its characteristic and restriction.

There is no attached linkage between class and property like UML has attached linkage between class and attribute. A property may or may not be owned by one or more classes in ontology modelling. A property can have generalisation relationships with other properties; this is unacceptable for UML but is acceptable for ontology modelling

IV. Software Engineering Ontology Modelling Notations

In this section, graphical notations are presented to facilitate the software engineering ontology modelling process. Some concepts or terms represented by notation have multiple presentations.

Software engineering concepts are represented as ontology classes in the software engineering ontology. Software engineering ontology class is a collection of specific project data with common characteristics that satisfy a restriction expression of the class. The notation of software engineering ontology class is represented as a rectangle with two compartments. The top compartment is for labelling the class and the second compartment is used for presenting properties related to the class. It is mandatory to specify the word '<<Concept>>' above the class label in the top compartment. To appear in the same standard as software engineering ontology class names, the class names are written with the first letter capitalised in each word without spaces between e.g. 'ObjectAttribute'. The class Thing is the special class that represents the set containing all instances in the software engineering ontology. All classes in software engineering ontology are subclasses of class Thing. Its notation description is the same with an ontology class notation but the top compartment contains the word '<<Concept>>' and 'Thing' as its label and the second compartment is empty. Fig. 1 (a) shows ontology class *ObjectAttribute* and (b) shows an ontology class Thing.

(a) Ontology class *ObjectAttribute* (b) Ontology class Thing

Fig.1 Ontology class symbol

Because the concept of generalisation in the software engineering ontology model is similar to the object-oriented model, we use the same generalisation symbol as in UML. The generalisation symbol appears as a line with one end empty and the other with a hollow triangular arrowhead. The empty end is always connected to the class being subsumed, whereas the hollow arrowhead connects to the class that subsumes. This symbol is also used to indicate generalisation of software engineering ontology properties. An example shown in Fig. 2 indicates classes *ObjectStructural*,

1-4244-0470-3/07/$20.00 ©2007 IEEE

ObjectDependency and *ObjectGeneralisation* are all subclasses of class *ObjectRelationship*; classes *ObjectAggregation*, *ObjectAssociation* and *ObjectComposition* are all subclasses of class *ObjectStructural*. From Fig. 2, properties of class *ObjectAssociation* are then properties inherited from its superclass of superclass. Additionally, *Constraint_Note*, *Related_Object* and *Relating_Object* are properties inherited from its superclass including *Related_Cardinality*, *Relating_Cardinality*, *Related_Role_Name* and *Relating_Role_Name* plus its own property, *Associated_Object*. This is because subclass properties contain properties inherited from superclass plus its own properties. Similarly, properties of class *ObjectDependency* are then *Constraint_Note*, *Related_Object*, *Relating_Object* and *Dependency_Type*.

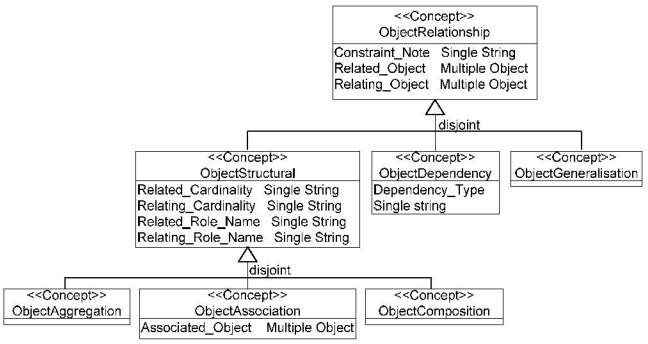


Fig.2 The subclasses of ontology class ObjectRelationship

In case subclasses of super class are disjointed, decomposed or partitioned, the words 'disjoint' or 'decompose' or 'partition' are attached to generalise symbol respectively. For disjoint classes, project data that has been asserted to be a member of one of the classes in the group cannot be a member of any other class in that group. Project data can be a member of a combination of classes in the group of decomposition classes. Project data of a class must be either project data of either one of the classes in the partition classes group. Fig. 2 shows that all the subclasses *ObjectStructural*, *ObjectDependency* and *ObjectGeneralisation* of class *ObjectRelationship* are disjointed. Likewise all the subclasses *ObjectAggregation*, *ObjectAssociation* and *ObjectComposition* of class *ObjectStructural* are disjointed.

Software engineering ontology properties including data type property, annotation property and object property, its characteristics, restrictions and association class attached property are defined notations. Data type property associates class to an XML schema data type value. Annotation properties have the same type of data type property used to add information which explains data about data. Object property associates classes to classes. To appear with the same standard, property names are written with the first letter of each word capitalised with an underscore (_) between the words e.g. 'Property_Name'.

A class with subclasses has properties which can have sub properties as well. Symbols of generalisation are the same for both class and property. Because they are applied to different entities they will not raise any confusion.

Data type properties of the software engineering ontology class can be expressed in the bottom compartment of class notation. That means the top compartment is known as its domain. In the bottom compartment, notation formats, as in the order of data type property name, its characteristic, its type (e.g. String, Integer) and its restriction. The type of data property is considered as a range. The characteristics of data type properties can be functional, inverse functional and non-functional. The notation of the characteristics is discussed later. An example shown in Fig. 3 expresses class *ClassOperation* has data type property *Class_Operation_Name* which is functional and its type is String. An oneOf in software engineering ontology is represented in curly brackets ({}). For example, Fig. 3 defines functional data type property *Class_Operation_Visibility* to relate a set of data value of 'public', 'private' and 'protected'.

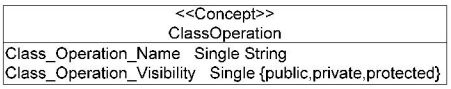
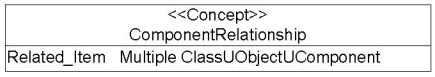


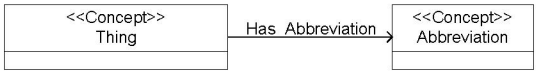
Fig.3 Data type properties presentation

Object properties of the software engineering ontology can be expressed in the bottom compartment of class notation like data type properties. In this manner, the top compartment is still called its domain. Notation format in the bottom compartment includes the object property's name, its characteristic, class name (its range), and its restriction. Class name expression as a range can assert the complex class description e.g. union, intersection. In addition, an object property can be expressed as an arrow with an open arrowhead and with text label of object property name. This is an alternative design for object properties. The arrow points from the domain of property to the range of the property. Its restrictions can be expressed in the bracket after its name. The characteristics of object property can be functional, inverse functional, non-functional, symmetric and transitive. The notation of the characteristics and the restrictions are discussed later. An example shown in Fig. 4(a) represents class *ComponentRelationship* which has a relationship with the union of classes *Class*, *Object* and *Component* through the property *Related_Item*. The domain of the property is class *ComponentRelationship* whereas the union of classes *Class*, *Object* and *Component* is the range of the property. The term 'Multiple' is the property's characteristic of non-functional. In this example, there is no property's restriction. Alternatively, object property can be presented as an arrow. An example shown in Fig. 4(b) represents an object property *Has_Abbreviation* relating any class in the software engineering ontology to the class *Abbreviation*. This is actually for any concept or term in software engineering having its abbreviation. Hence, it can be asserted that if software engineering concepts or terms have no abbreviation this object property will not appear for those ontology classes. In other words, the object property *Has_Abbreviation* is literally a stand alone property.

1-4244-0470-3/07/$20.00 ©2007 IEEE



(a) Object property *Related_Item* presentation



(b) Object property *Has_Abbreviation* presentation

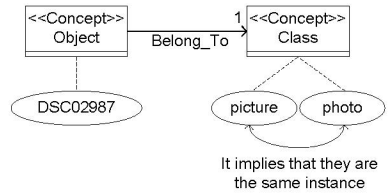
Fig.4 Object properties presentation

Characteristics of the software engineering ontology property are classified into five categories which are functional, inverse functional, non-functional, symmetric and transitive properties. A functional property has a maximum cardinality of 1 on its range while an inverse functional property has a maximum cardinality of 1 on its domain. A non-functional property has cardinality within the range of 0 to many. Properties can be declared symmetric for symmetric properties or transitive for transitive properties.

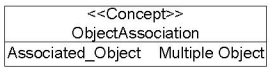
A functional property notation is represented as an open arrowhead with text label of property name and symbol number 1 near the arrowhead. A non-functional property notation is represented simply as an open arrowhead with text label of the property name which is similar to a functional property notation. The difference is that there is no symbol number 1 at near the arrowhead. However, at the arrowhead, the cardinality restriction can be presented at near the arrowhead (details in the part of property restriction notation). An inverse functional property notation is represented as an open arrowhead with the arrowhead pointing opposite to its inverse property notation point direction and a dot line placed in the middle linking the inverse functional property symbol and its inverse. As usual, the text label represents the property name and there is a symbol number 1 near the arrowhead. A symmetric property notation is represented simply as a solid line with a text label of the property name. Lastly, a transitive property notation is represented as a dot arrow with an open arrowhead and with a text label of its property name.

The property is expressed in the bottom compartment of the class notation. A functional property is represented as the word 'Single' whereas a non-functional property is represented as the word 'Multiple'. An inverse functional property and a symmetric property are represented as the words 'Inverse' and 'Symmetric' respectively.

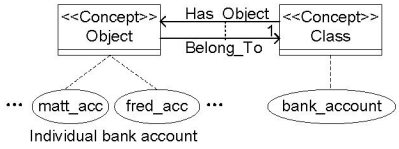
For example, object property *Belong_To* shown in Fig. 5(a) represents functional property related class *Object* to class *Class*. If instance *DSC02987* of class *Object* and instances *picture* and *photo* of class *Class*, because property *Belong_To* is functional, then it can be inferred that instances *picture* and *photo* must be the same instance. An example of a non-functional property illustrate in Fig. 5(b) representing object property *Associated_Object* related class *ObjectAssociation* to class *Object*. The term 'Multiple' represents the non-functional property. Fig. 5(c) shows an example of inverse functional property. Object property *Has_Object* is inverse functional i.e. object property *Belong_To*. From Fig. 5(c) *matt_acc* and *fred_acc* are objects belonging to UML class *bank_account*. UML class *bank_account* has many objects belonging to it. Object of UML class *bank_account* represents an individual bank account.



(a) Functional property *Belong_To*



(b) Non-functional property *Associated_Object*



(c) Inverse functional property *Belong_To*

Fig.5 Characteristics of properties presentation – functional, non-functional, and inverse functional properties

Basically, in software engineering ontology modelling the multiplicity of a relationship has to be specified in pairs because it can be indicated at the end of the relation line, one indicator for each relation only. Hence, if you want to specify multiplicity 'one to one' then you need to specify a functional property and an inverse functional property. Likewise, the multiplicity 'one to many' can specify a non-functional property and an inverse functional property.

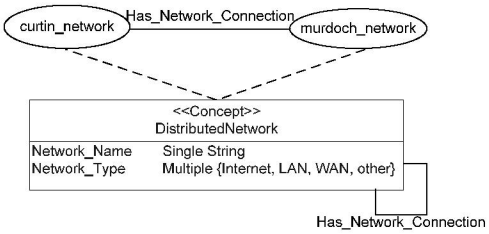
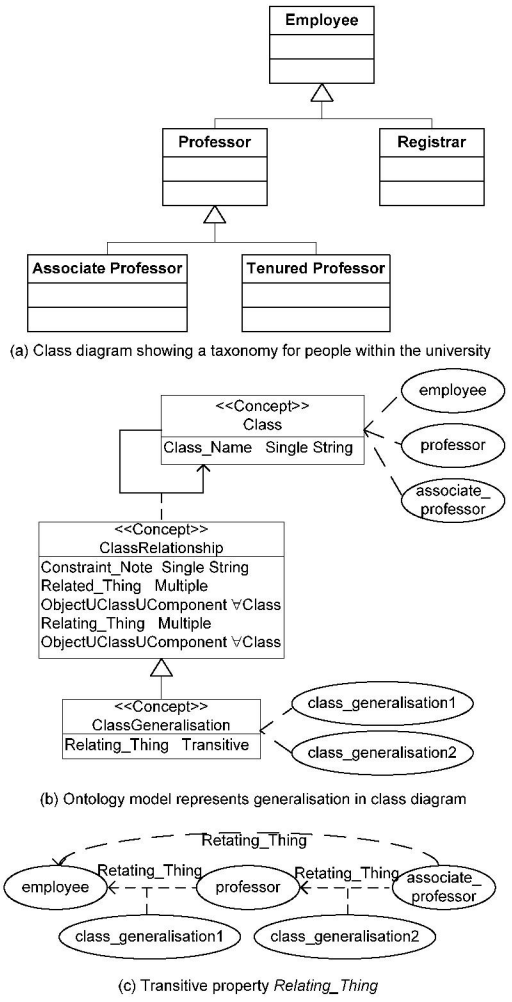


Fig.6 Symmetric property Has_Network_Connection presentation

Another example of property characteristics is shows in Fig. 6. Class *DistributedNetwork* has a symmetric relationship to itself. If *curtin_network* and *murdoch_network* are instances of class *DistributedNetwork* and the property *Has_Network_Connection*, that is symmetric, relates instance *curtin_network* to instance *murdoch_network* then it can be inferred that instance *murdoch_network* must also be related to instance *curtin_network* through the same property *Has_Network_Connection*.



(a) Class diagram showing a taxonomy for people within the university

(b) Ontology model represents generalisation in class diagram

(c) Transitive property *Relating_Thing*

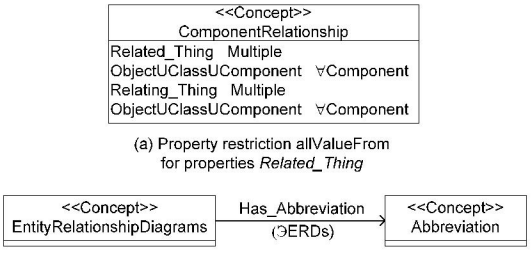
Fig.7 Transitive property presentation

Fig. 7 shows an example of a transitive property. Fig. 7(a) shows generalisation in the class diagram of the taxonomy for people working within the university. From the class diagram it refers to UML class Associate Professor as a subclass of UML class Professor and UML class Professor is a subclass of UML class Employee. In other words, both associate professor and tenured professor are a professor; both professor and registrar are an employee. From here it means that all of associate professor, tenured professor and registrar are employees. This generalisation relationship specifies of transitive property in ontology modelling. Hence, in Fig. 7(b) *employee*, *professor* and *associate_professor* are set as instances of class *Class*. Association class *ClassGeneralisaion* represents generalisation relationship between UML classes. For example, its instance *class_generalisation1* relates instance of class *Class*, *employee*, to instance of class *Class*, *professor*, and its instance *class_generalisation2* relates instance of class *Class*, *professor*, to instance of class *Class*, *associate_professor*. Property *Relating_Thing* of class *ClassGeneralisaion* is transitive, hence, it can be inferred that instance *associate_professor* relates to instance *employee* through the property *Relating_Thing* as shown in Fig. 7(c).

An upside down A symbol ∀ represents restriction allValueFrom. A *backwards* facing E symbol ∃ represents restriction someValueFrom. Property range can be restricted to a class only, called allValueFrom restriction, or one part of a class, called someValueFrom restriction, when the property is applied to the domain class. A symbol denoted by ∋ represents restriction hasValue describing the set of project data that has at least one relation along a specified property to a specific project data. For the cardinality restriction, symbols equal (=), greater than and equal (>) and less than and equal (<) represent respective cardinality specifying the exact number, minimum cardinality specifying the minimum number and maximum cardinality specifying the maximum number. Cardinality constraints can be replaced with the format of x..y where 'x' is the value of minimum cardinality and 'y' is the value of maximum cardinality. The asterisks (*) is used as part of the specification to indicate the unlimited upper bound. For example, specifying 'a > 2' is equivalent to '2..*', specifying 'a < 2' is equivalent to '0..2', specifying 'a = 2' is equivalent to '2..2' or just '2'. Table 1 summarises the potential indicators matching with ontology property characteristics or restriction.

Table 1 The potential indicators with ontology property characteristics or restriction

Indicator	Property Characteristic/Restriction	Meaning
1	Functional property	One only
0..*	Non-functional property	Zero or more
1..*	someValueFrom restriction	At least one
n	Cardinality restriction	Only n where n > 1
0..*	Maximum cardinality restriction	At most n where n > 1
n..*	Minimum cardinality restriction	At least n where n > 1



(a) Property restriction allValueFrom for properties *Related_Thing*

(b) Property restriction hasValue for property *Has_Abbreviation*

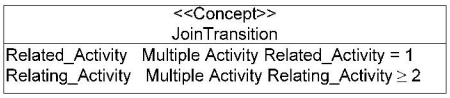
Fig.8 Property restrictions presentation

For example, object properties shown in Fig. 8(a) *Related_Thing* and *Relating_Thing* restrict that instances of class *ComponentRelationship* relating only to instances of class *Component*. The hasValue restriction is for example applied to property *Has_Abbreviation* depicted in Fig. 8(b) relating class *EntityRelationshipDiagrams* to a particular instance *ERDs* that is instance of class Abbreviation. The term ERDs stands for entity relationship diagrams.

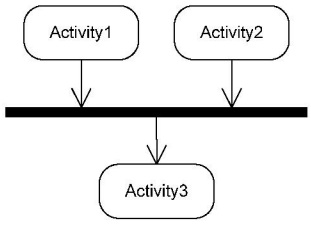
An example of the cardinality restriction is shown in Fig. 9(a). From Fig. 9(a) it can be inferred that at least two *Relating_Activity* relationships relate instance *JoinTransi-*

1-4244-0470-3/07/$20.00 ©2007 IEEE

tion to class *Activity*. For the property *Related_Activity* restricts instance from class *JoinTransition* to exactly one instance of class *Activity*. In other words, in join transition (from activity diagram) there are at least two relating activities transited into one related activity as shown in Fig. 9(b).



(a) Cardinality restrictions in join transtition



(b) Join transition in activity diagram

Fig.9 Property cardinality restriction for properties Related_Activity and Relating_Activity

Association class can be used to participate in further associations for property in software engineering ontology. To specify association classes a dot line is used to attach the property notation with class notation.

An example shown in Fig. 10 illustrates that class *ClassAggregaion* is an associate class of object property that relates class *Class* together. Association class *ClassAggregaion* can participate in further associations with bundles of properties i.e. *Contraint_Note*, *Related_Thing*, *Relating_Thing*, *Related_Cardinality*, *Relating_Cardinality*, *Related_Role_Name* and *Relating_Role_Name*.

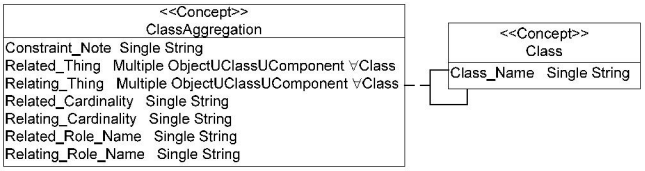


Fig.10 Association class ClassAggregation presentation

Instances represent specific project data in software engineering ontology. An instance notation is represented as an ellipse with a dot line attached to its class or property. If it is an instance of property then in the ellipse it contains the property name followed by a colon mark and then instance name. Unlike an instance of class, in the ellipse only the instance name is there. To make it easy to read, a dot line is attached to most of the class name or property name of its class instance or property instance. To be in the same standard, instance name is written all in small letters and an underscore (_) can be used as spaces between the words e.g. 'instance_name'.

For example, to populate the UML class diagram shown in Fig. 11 into an ontology model of the class diagram shown in Fig. 12. From the UML class diagram it defines UML classes Company, Person and Job and association relationship, hence, they will be defined as instances of class *Class* and class *ClassAssociation* as shown as class instances in Fig. 12. Theoretically, subclass is inheriting properties from its superclass and in Fig. 12 it also shows instances of properties and inherited properties. Fig. 13 shows all class instances and property instances related together. To make it easy to understand, it is in the same form of the class diagram in Fig. 11.

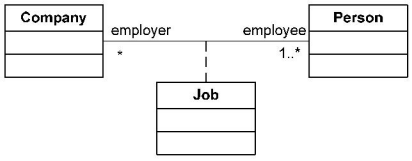


Fig.11 UML class diagram shown association relationship

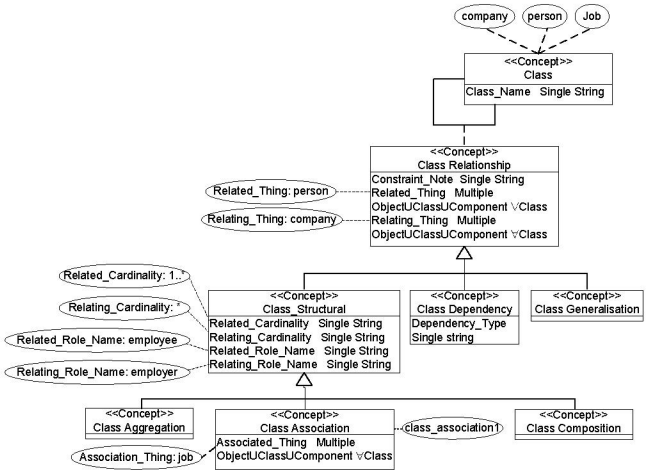


Fig.12 Ontology model of the class diagram

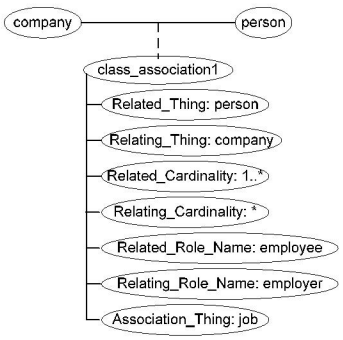


Fig.13 Instances of classes and properties of ontology model of the class diagram

V. Conclusion

This paper has discussed software engineering ontology and its modelling. Generally, we adopted UML for software engineering ontology representation as an alternative formalism. The characteristic of UML and our UML based modelling graphical notations facilitating the software engineering ontology were compared and discussed in detail. Our future work aims at apply this graphical ontology representation to any other domain knowledge across different domain disciplines.

1-4244-0470-3/07/$20.00 ©2007 IEEE

VI. REFERENCES

[1] Wongthongtham, P., E. Chang, and T.S. Dillon, Ontology based multi-agent system to software engineering. a Special Issue of the Journal of System Architecture, 2005.

[2] Wongthongtham, P., E. Chang, and T.S. Dillon. Information Engineering of a Software Engineering Ontology. in 3rd International IEEE Conference on Industrial Informatics. 2005. Perth, Australia.

[3] Wongthongtham, P., E. Chang, and T.S. Dillon. Towards 'Ontology'-based Software Engineering. in 3rd International IEEE Conference on Industrial Informatics. 2005. Perth, Australia.

[4] Wongthongtham, P., E. Chang, and T.S. Dillon. Software Engineering Sub-ontology for Specific Software Development. in 29th Annual IEEE/NASA Software Engineering. 2005. Washington DC, USA.

[5] Wongthongtham, P., et al. Software Engineering Ontologies and their Implementation. in (accepted for presentation) The IASTED International Conference on SOFTWARE ENGINEERING, February 15-17. 2005. Innsbruck, Austria.

[6] Wongthongtham, P., E. Chang, and T.S. Dillon. Ontology based multi-agent system to multi-site software development. in Proceedings of International ACM Conference SIGSOFT 2004/FSE-12. 2004. California, USA.

[7] Wongthongtham, P., E. Chang, and T.S. Dillon. Methodology for multi-site software engineering using ontology. in Proceedings of the International Conference on Software Engineering Research and Practice. 2004. Las Vegas, USA.

[8] Wongthongtham, P., E. Chang, and T.S. Dillon. Intelligent communication through software agent and ontology for multi-site software engineering. in Proceedings of the 3rd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems. 2004. Edinburgh, UK.

[9] Wongthongtham, P., et al. Ontology based solution proposal for multi-site distributed software development. in Proceedings of the 16th International Conference on Software and Systems Engineering and their Applications. 2003. Paris , France.

[10] Gruber, T.R. Toward principles for the design of ontologies used for knowledge sharing. in International Workshop on Formal Ontology in Conceptual Analysis and Knowledge Representation. 1993. Padova, Italy: Kluwer Academic Publishers, Deventer, The Netherlands.

[11] Borst, W., Construction of Engineering Ontologies. 1997, Centre of Telematica and Information Technology, University of Tweenty: Enschede, The Netherlands.

[12] Studer, R., V. Benjamins, and D. Fensel. Knowledge Engineering: Principles and Methods. in IEEE Transactions on Data and Knowledge Engineering. 1998.

[13] Genesereth, M.R. Knowledge Interchange Format - draft proposed American National Standard. 1998 [cited; Available from: http://logic.stanford.edu/kif/dpans.html.

[14] Brachman, R.J. and J.G. Schmolze, An overview of the KL-ONE knowledge representation system, in Cognitive Science. 1985. p. 171-216.

[15] Farquhar, A., R. Fikes, and J. Rice. The Ontolingua Server: A Tool for Collaborative Ontology Construction. in 10th Knowledge Acquisition for Knowledge-Based Systems Workshop. 1996. Banff, Canada.

[16] MacGregor, R., Inside the LOOM classifier. SIGART bulletin, 1991. 2(3): p. 70-76.

[17] Genesereth, M.R. and R.E. Fikes, Knowledge Interchange Format Version 3 Reference Manual. 1992, Stanford University Logic Group.

1-4244-0470-3/07/$20.00 ©2007 IEEE