Ontology-Based Requirement Conflicts Analysis in Class Diagrams

Chi-Lun Liu, Hsieh-Hong Huang

Abstract—**Requirements modeling and analysis are important in successful software engineering projects. Class diagrams are a useful standard for modeling static structures of information systems. Analyzing conflicts in software specifications is crucial when multiple stakeholder concerns need to be addressed. This work uses ontologies to analyze conflicts in the requirement specifications of class diagrams. The conflict analysis process and Twenty-one rules are proposed to detect four conflict issues: inconsistencies, redundancies, overrides, and missing parts. The proposed process and rules can help novices to analyze conflicts in class diagrams. The proposed rules are also feasible to be automatically executed by knowledge-based systems.**

Index Terms—**Requirements engineering, class diagram, ontology; conflicts analysis**

I. INTRODUCTION

LISTENING and modeling user requirements are important in successful software system development (He and King, 2008). Unified Modeling Language (UML) is a mainstream standard for requirements modeling. Class diagrams are commonly used for modeling the static aspects of information systems.

Analyzing conflicts in software models is crucial when multiple stakeholder concerns need to be addressed by software engineers (Savolainen and Männistö, 2010). Design inconsistencies are common in industries and often hard to be recognized (Egyed, 2006). Using ontologies to manage domain knowledge and support system development is emergent in the recent years (Nomaguchi and Fujita, 2007; Liu, 2010). However, none of the related works uses ontologies to analyze conflicts in class diagrams.

This work proposes a conflict analysis process and a set of rules to detect conflicts to reduce errors in class diagrams. The conflict analysis process are a four-step circle including modeling prior knowledge, modeling new requirements, detecting conflicts, and resolving conflicts. These rules handles four conflict issues: inconsistencies, redundancies, overrides, and missing parts. Scenarios in the electronic commerce context are also provides to preliminarily demonstrate and validate the proposed rules.

The advantage of the proposed process and rules are twofold. The process and rules can help students and novice software engineers to analyze conflicts in class diagrams by means of semantics in the ontology. On the other hand, the proposed rules and ontology is feasible to be stored and executed in knowledge-based systems to detect conflicts automatically.

The reminder of this paper is structured as follows: Section II discusses related works about conflicts analysis. Section III proposes the conflict analysis process. Section IV presents the proposed 21 rules for conflict detection based on the ontology. Scenarios are also provided for demonstrating how these rules works appropriately in this section. Finally, Section V discusses the conclusion.

II. RELATED WORKS

Table I summarizes the existing conflict analysis works in requirements engineering. Maxwell, Antón, and Swire (2011) and Roth et al. (2013) provides a taxonomy and process to identify and resolve conflicts in software requirements. Mohite et al. (2014) and Sapna and Mohanty (2007) analyze inconsistencies and conflicts in UML diagrams. Kaiya and Saeki (2005) and Liu (2010) use ontologies to analyze and resolve conflicts in requirements. These works reveal that conflicts occur in various requirement documentations. Two of these works use ontologies to analyze conflicts. Two of these works focus on UML diagrams. And no work use ontologies to analyze conflicts in class diagrams in Table I. Therefore conflict analysis for class diagrams is a valuable research issue.

TABLE I
EXISTING REQUIREMENTS CONFLICT ANALYSIS WORKS

	Document	Conflict Analysis Approach
Maxwell, Antón, and Swire (2011)	Legal Text and software requirements	Use legal cross-reference taxonomy for identify software requirement conflicts
Roth et al. (2013)	Enterprise architecture documentation	Provide a conflict resolution process
Mohite et al. (2014)	UML	Use Graph transformation systems to detect conflicts and dependencies between UML diagrams
Sapna and Mohanty (2007)	UML	Use a set of rules to detect inconsistencies among different kinds of UML diagrams
Kaiya and Saeki (2005)	X <u>is a instance of</u> Y, Subject-Verb-Object	Detect requirements conflicts according to predefined conflict relations in ontologies.
Liu (2010)	Activity Diagram	Use ontologies to automatically detect conflicts

Manuscript received July 22, 2014; revised February 13, 2015.

Chi-Lun Liu is with the Department of Multimedia and Mobile Commerce, Kainan University, Taoyuan 33857, Taiwan (corresponding author to provide phone: 886-3-341-2500#6070; fax: 886-3-341-2373; e-mail: tonyliu@mail.knu.edu.tw).

Hsieh-Hong Huang is with the Department of Information Science and Management Systems, National Taitung University, Jhiben Campus, Taitung, 369, Taiwan (e-mail: kory@nttu.edu.tw).

III. ONTOLOGY-BASED CONFLICT ANALYSIS PROCESS IN REQUIREMENTS EVOLUTION

This section extends the prior work (Liu, 2010) to propose the ontology-based conflict analysis process in requirements evolution depicted in Fig. 1. The proposed conflict analysis process is a circle and has four steps: modeling prior knowledge, modeling new requirements, detecting conflicts, and resolving conflicts. These steps are introduced as follows.

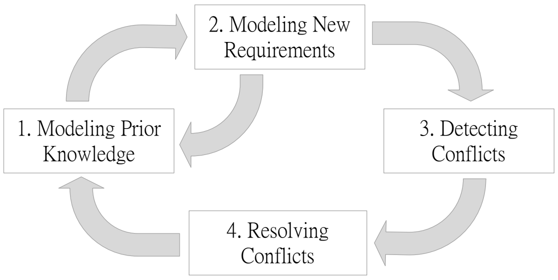

Fig. 1. Conflicts analysis process in requirements evolution

(1) Modeling prior knowledge: Users, software engineers, and knowledge engineers model the domain knowledge, approved existing requirements, and conflict detection rules. The terms related to the domain should be stored in the ontology. These terms in the ontology will be used to represent requirements specifications, such as class diagrams. This work proposes several conflict detection rules. New rules can also be added in this step.

(2) Modeling new requirements: Modeling new requirements in this step is based on the prior knowledge in step 1. Terms in the ontology, which are established in step 1, can be used to represent new requirements. The terms used in new requirements should exist in the ontology. If a new term is necessary to represent a new requirement, step 1 is performed to add this term in the ontology.

(3) Detecting conflicts: This step uses the ontology and rules to detect conflicts. Several rules and scenarios are provided to explain how to detect conflict in the next section.

(4) Resolving conflicts: Stakeholders should negotiate a solution for the conflicts in this step. If requirements and environments change, step 1 is performed to start these steps again.

IV. PROPOSED CONFLICT DETECTION RULES

Twenty-one rules are proposed for conflict detection in class diagrams. These rules detect inconsistencies, redundancies, overrides, and missing parts. This section introduces these rules and explain these rules with several scenarios in the electronic commerce context.

A. *Inconsistency Detection Rules*

Inconsistency detection rules focus on inconsistencies between two requirements and between requirements and the ontology. Rule$_{\text{ID}}$1-11 are proposed in this section. Scenarios are also provided to explain these rules.

Rule$_{\text{ID}}$1: There is a requirements generalization inconsistency if ClassM is a superclass of ClassN in ReqE, ClassO is a suprclass of ClassP in ReqF, an equality or a synonym relationship exists between concept ClassN and concept ClassO in the ontology, and an equality or a synonym relationship exists between concept ClassM and concept ClassP in the ontology.

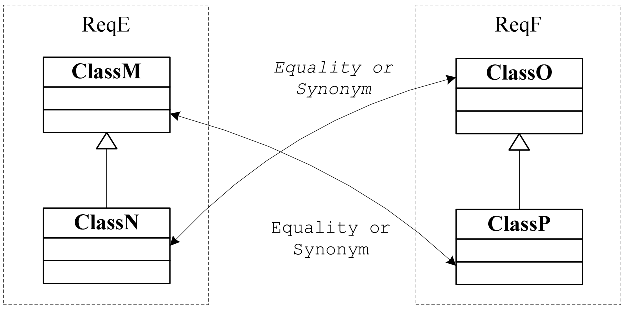

Fig. 2. Requirements generalization inconsistency

Rule$_{\text{ID}}$2: There is a requirements composition inconsistency if there is a composition relationship from ClassN to ClassM in ReqE, there is a composition relationship from ClassP to ClassO in ReqF, an equality or a synonym relationship exists between concept ClassN and concept ClassO in the ontology, and an equality or a synonym relationship exists between concept ClassM and concept ClassP in the ontology.

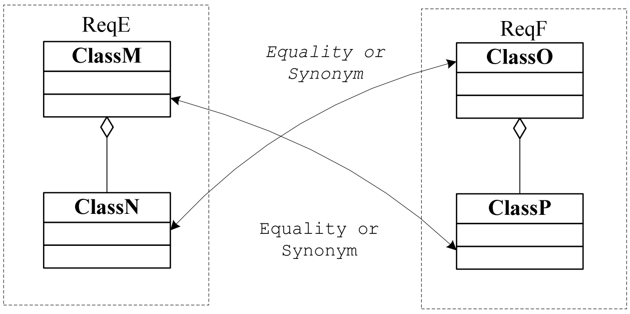

Fig. 3. Requirements composition inconsistency

Rule$_{\text{ID}}$3: There is a requirements aggregation inconsistency if there is an aggregation relationship from ClassN to ClassM in ReqE, there is an aggregation relationship from ClassP to ClassO in ReqF, an equality or synonym relationship exists between concept ClassN and concept ClassO in the ontology, and an equality or a synonym relationship exists between concept ClassM and concept ClassP in the ontology.

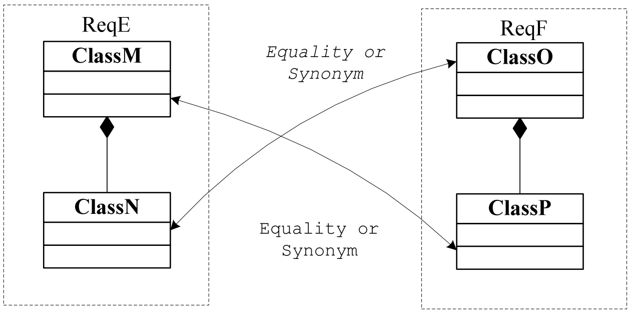

Fig. 4. Requirements aggregation inconsistency

Requirements generalization inconsistency detected by Rule$_{\text{ID}}$1 means that a class is not only a superclass but also a subclass of another class in a wrong class diagram. Fig. 2 depicts Rule$_{\text{ID}}$1. Requirements composition inconsistency detected by Rule$_{\text{ID}}$2 and aggregation requirements inconsistency detected by Rule$_{\text{ID}}$3 mean a class is a part and a whole of another class. Fig. 3 and 4 depicts Rule$_{\text{ID}}$2 and Rule$_{\text{ID}}$3. For example, Payment service (ClassM) is a superclass of Near Field Communication (ClassN) in ReqE in the payment system. NFC (ClassO) is a supserclass of Payment service (ClassP) in ReqF. ClassM equals ClassP. ClassN is a synonym of ClassO because NFC is the abbreviation of Near Field Communication. According to Rule$_{\text{ID}}$1, requirements generalization inconsistency occurs. The structures of Rule$_{\text{ID}}$1-3 are similar.

Rule$_{\text{ID}}$4: There is a method exclusion inconsistency if AttributeX is added in ClassM in ReqE, MethodI is not allowed in ClassN in ReqF, there is an equality, kind, part, or synonym relationship between MethodI and MethodJ, and there is an equality, kind, part, or synonym relationship between ClassM and ClassN.

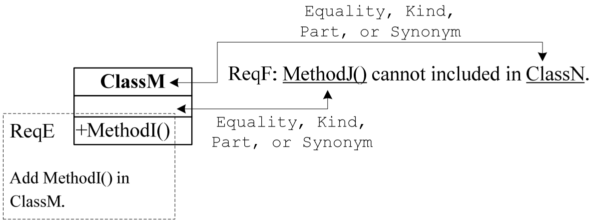

Fig. 5. Method exclusion inconsistency

Some behaviors in information systems are regulated by government laws and corporation policies. Method exclusion inconsistency detected by Rule$_{\text{ID}}$4 and illustrated in Fig. 5 means an undesirable method is added. For example, storing credit_card_number() (MethodI()) is added in credit card (ClassM). Credit card numbers cannot be stored in the database is a the corporation policy because stored credit card numbers has a security risk about hacking. Therefore Storing_credit_card_number() (MethodJ) cannot included in Any class (ClassN) in ReqF in the payment system. MethodI() equals MethodJ(). ClassM is a kind of ClassN. According to Rule$_{\text{ID}}$4, method exclusion inconsistency occurs.

Rule$_{\text{ID}}$5: There is a multiple inheritance inhibition inconsistency if a generalization relationship from ClassO to ClassN is added in ReqE, there is a generalization relathionship from ClassO to ClassM in the existing requirements, and multiple inheritance is not allowed in ReqF.

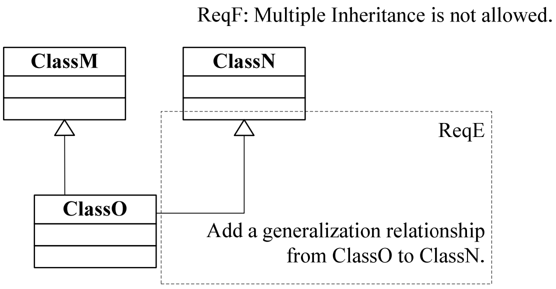

Fig. 6. Multiple inheritance inhibition inconsistency.

Some programming languages inhibit multiple inheritance, such as Java. Rule$_{\text{ID}}$5 detecting multiple inheritance inhibition inconsistency indicates that more than one superclass exists in a class diagram. Fig. 6 depicts Rule$_{\text{ID}}$5

Rule$_{\text{ID}}$6: There is a generalization and alternative inconsistency if a generalization relationship from ClassN to ClassM is added in ReqE and there is a equality, part, antonym, or synonym relationship between concept ClassM and concept ClassN in the ontology.

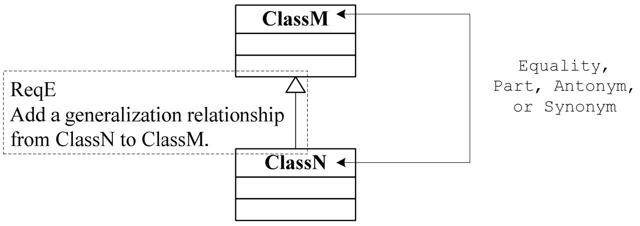

Fig. 7. Generalization and alternative inconsistency

Rule$_{\text{ID}}$6 detecting generalization and alternative inconsistency means there is an alternative semantic relationship other than a generalization relationship between two classes in the ontology. Fig. 7, 9, 11 depict Rule$_{\text{ID}}$6, Rule$_{\text{ID}}$8 and Rule$_{\text{ID}}$10. The structures of Rule$_{\text{ID}}$6, Rule$_{\text{ID}}$8 and Rule$_{\text{ID}}$10 are similar. For example, the ontology indicates that Payment service (ClassN) is a part of Electronic commerce website (ClassM). According to Rule$_{\text{ID}}$6, adding a generalization relationship from Payment service (ClassN) to Electronic commerce website(ClassM) causes a generalization and alternative inconsistency.

Rule$_{\text{ID}}$7: There is an inverse generalization inconsistency if a generalization relationship from ClassN to ClassM is added in ReqE and concept ClassN is a kind of concept ClassM in the ontology.

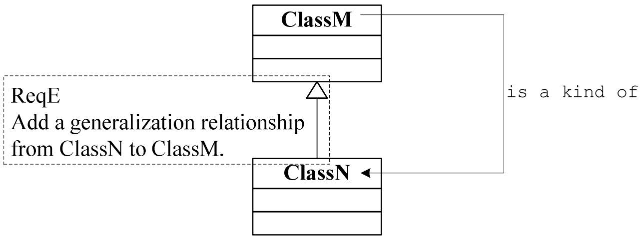

Fig. 8. Inverse generalization inconsistency.

Rule$_{\text{ID}}$7 detecting inverse generalization inconsistency means the direction of generalization relationship between two classes in a class diagram is inverse comparing to the ontology. Fig. 8, 10, 12 depict Rule$_{\text{ID}}$7, Rule$_{\text{ID}}$9 and Rule$_{\text{ID}}$11. The structures of Rule$_{\text{ID}}$7, Rule$_{\text{ID}}$9 and Rule$_{\text{ID}}$11 are similar. For example, the ontology shows NFC (ClassM) is a kind of Wireless connectivity (ClassN). According to Rule$_{\text{ID}}$7, adding a generalization relationship from Wireless connectivity (ClassN) to NFC (ClassM) which means wireless is a kind of NFC causes inverse generalization inconsistency.

Rule$_{ID}$8: There is an aggregation and alternative inconsistency if an aggregation relationship from ClassN to ClassM is added in ReqE and there is a equality, part, antonym, or synonym relationship between concept ClassM and concept ClassN in the ontology.

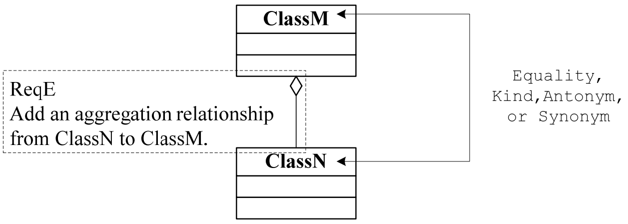

Fig. 9. Aggregation and alternative inconsistency.

Rule$_{ID}$9: There is an inverse aggregation inconsistency if an aggregation relationship from ClassN to ClassM is added in ReqE and concept ClassN is a kind of concept ClassM in the ontology.

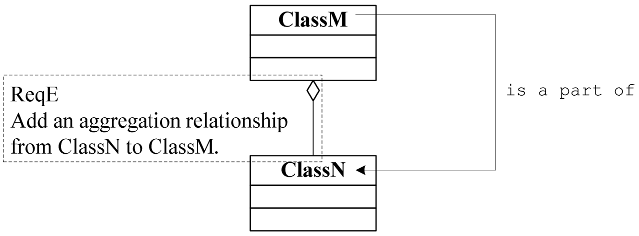

Fig. 10. Inverse aggregation inconsistency

Rule$_{ID}$10: There is a composition and alternative inconsistency if a composition relationship from ClassN to ClassM is added in ReqE and there is a equality, part, antonym, or synonym relationship between concept ClassM and concept ClassN in the ontology.

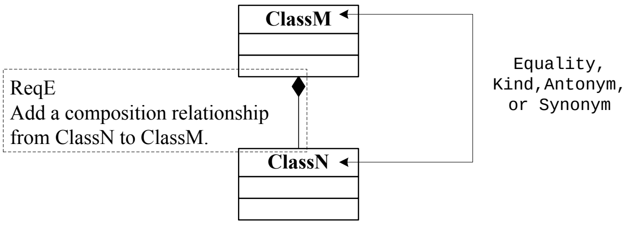

Fig. 11. Composition and alternative inconsistency.

Rule$_{ID}$11: There is an inverse composition inconsistency if a composition relationship from ClassN to ClassM is added in ReqE and concept ClassN is a kind of concept ClassM in the ontology.

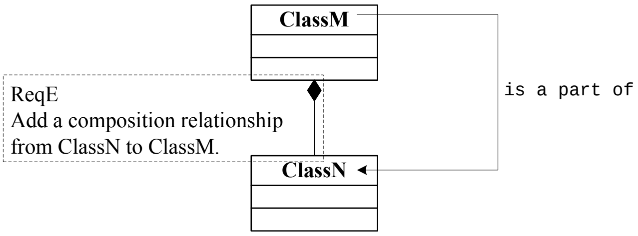

Fig. 12. Inverse composition inconsistency.

B. Redundancy Detection Rules

The ontology provides the domain knowledge to detect redundancies about classes, attributes, and methods in redundancy detection rules. Rule$_{RD}$1-6 are proposed and explained as follows.

Rule$_{RD}$1: There is an attribute redundancy if AttributeY is added in ClassM in ReqE and there is an equality, kind, part, or synonym relationship between concept AttributeX and concept AttributeY in the ontology.

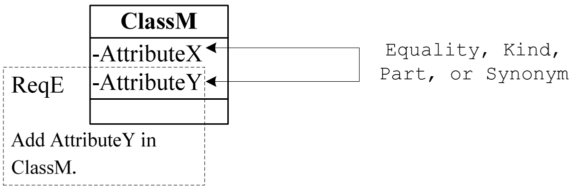

Fig. 13. Attribute redundancy.

Attribute redundancy in Rule$_{RD}$1 means two attributes are the same, similar, or overlap. Method redundancy in Rule$_{RD}$2 means two methods are the same, similar, or overlap. The structures of Rule$_{RD}$1 and Rule$_{RD}$2 are similar. Fig. 13 depicts Rule$_{RD}$1 and Fig. 14 depicts Rule$_{RD}$2. For example, Expired date (AttributeX) exists in Credit card (ClassM). According to Rule$_{RD}$1, adding Date (AttributeY) causes attribute redundancy because Expired date (AttributeX) is a kind of Date (AttributeY). The name of AttributeY is inappropriate and needs to be modified.

Rule$_{RD}$2: There is a method redundancy if MethodJ() is added in ClassM in ReqE and there is an equality, kind, part, or synonym relationship between concept MethodI() and concept MethodJ() in the ontology.

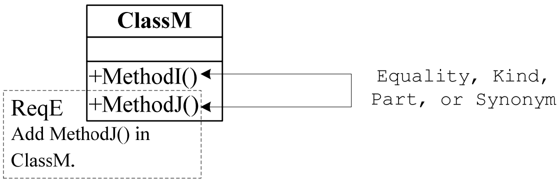

Fig. 14. Method redundancy.

Rule$_{RD}$3: There is a generalization relationship redundancy if a generalization relationship from ClassN to ClassM is added in ReqE and concept ClassN is not a kind of ClassM in the ontology.

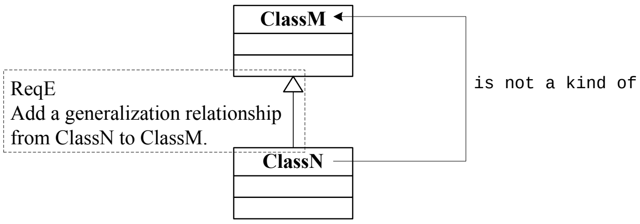

Fig. 15. Generalization relationship redundancy.

Generalization relationship redundancy in Rule$_{RD}$3 means the meaning of generalization relationship between two classes in a class diagram does not appear in the ontology. Aggregation relationship redundancy in Rule$_{RD}$4 means the meaning of aggregation relationship between two classes in a class diagram does not appear in the ontology. Composition relationship redundancy in Rule$_{RD}$5 means the meaning of composition relationship between two classes in a class diagram does not appear in the ontology. The structures of Rule$_{RD}$3, Rule$_{RD}$4, and Rule$_{RD}$5 are similar. Fig. 15-17 depict Rule$_{RD}$3-5. For example, stakeholders propose ReqE: "Debit card (ClassN) is a subclass of Payment service (ClassM)". The domain knowledge which is "Debit card is a kind of Payment service" cannot be found in the ontology. According to Rule$_{RD}$3, generalization relationship redundancy occurs and the domain knowledge should be updated in this case.

Rule$_{RD}$4: There is an aggregation relationship redundancy if an aggregation relationship from ClassN to ClassM is added in ReqE and concept ClassN is not a part of ClassM in the ontology.

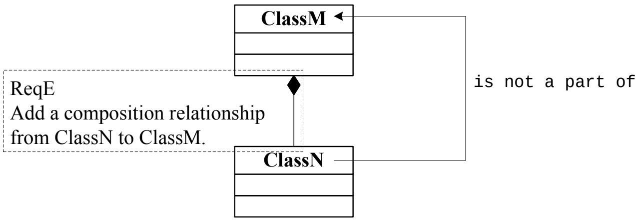

Fig. 16. Aggregation relationship redundancy.

Rule$_{RD}$5: There is a composition relationship redundancy if a composition relationship from ClassN to ClassM is added in ReqE and concept ClassN is not a part of ClassM in the ontology.

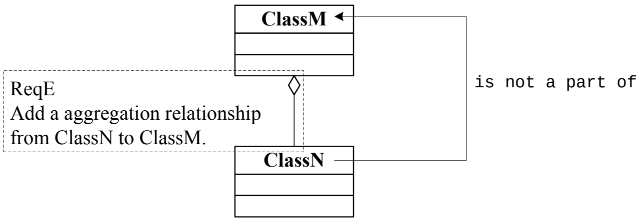

Fig. 17. Composition relationship redundancy.

Rule$_{RD}$6: There is a class redundancy if ClassN is added in ReqE and there is an equality or synonym relationship between concept ClassM and ClassM in the ontology.

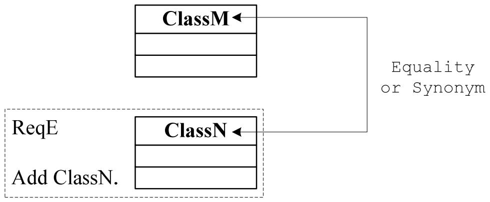

Fig. 18. Class redundancy.

Class redundancy in Rule$_{RD}$6 indicates two classes are the same. Fig. 18 illustrates Rule$_{RD}$6. For example, Debit card (ClassM) is already in the class diagram. According to Rule$_{RD}$6, adding a Check card (ClassN) causes class redundancy because check card is a synonym of debit card.

C. Override Detection Rules

Override is an essential characteristic in Object-Oriented Programming. The appropriateness of override should be concerned in software engineering processes. The two proposed rules for override detection remind software engineers about potential overrides. Rule$_{OD}$1-2 are proposed and discussed as follows.

Rule$_{OD}$1: There is a possible override during generalization relationship addition if a generalization relationship from ClassN to ClassM is added in ReqE and there is an equality, kind, composition, or synonym relationship between concept MethodI() in ClassM and MethodJ() in ClassN in the ontology.

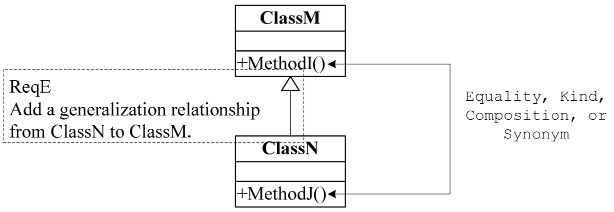

Fig. 19. Possible override during generalization relationship addition

Rule$_{OD}$1 and Rule$_{OD}$2 shows possible overrides to remind software engineers about overrides of methods in class diagrams. Fig. 19-20 depicts Rule$_{OD}$1-2. In Rule$_{OD}$1, adding a generalization relationship reminds software engineers about possible override. In Rule$_{OD}$2, adding a method reminds software engineers about possible override. For example, VIP_member (ClassN) is a subclass of Member (ClassM) in Fig. 20. Storing_membership_application_form() (MethodI()) is in Member (ClassM). Stakeholders need a new method in VIP_member to store VIP members' membership application data. Therefore Storing_vip_membership_application_form() (MethodJ()) is added in VIP_member (ClassN). Obviously, Storing_vip_membership_application_form() (MethodJ()) is a kind of Storing_membership_application_form() (MethodI()). According to Rule$_{OD}$2, MethodJ() (Storing_vip_membership_application_form()) in ClassN can be renamed as Storing_membership_application_form() to override MethodI() in ClassM.

Rule$_{OD}$2: There is a possible override during attribute addition if MethodJ() is added in ReqE, ClassM is a superclass of ClassN, and there is an equality, kind, composition, or synonym relationship between concept MethodI() in ClassM and MethodJ() in ClassN in the ontology.

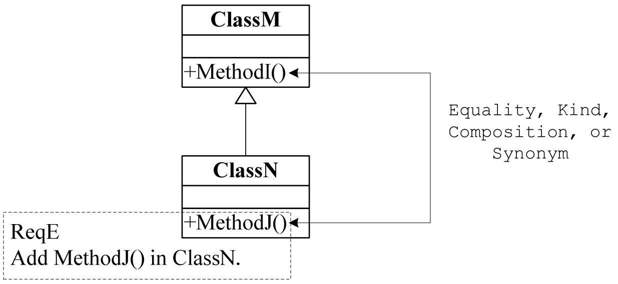

Fig. 20. Possible override during attribute addition

D. Missing Part Detection Rule

The proposed missing part detection rules use the ontology to find the classes that have been omitted. Rule$_{MPD}$1-2 are introduced and discussed as follows.

Rule$_{MPD}$1: There is a possible missing class if ClassN is added in ReqE, Concept i in the ontology equals ClassN, and Concept i has precise (child and part), general (parent and whole), or sibling concepts.

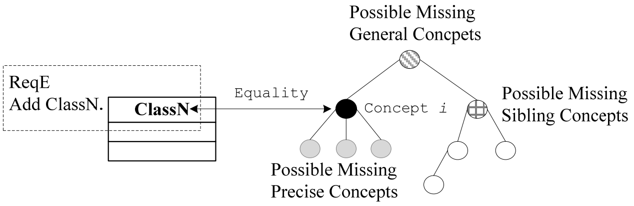

Fig. 21. Possible missing class.

Rule$_{MPD}$1 indicates a possible missing class in a class diagram. Possible missing class means a class is added in a class diagram and this class has parent, child, whole, part, or sibling concept in the ontology. The structures of Rule$_{MPD}$1 and Rule$_{MPD}$2 are similar. Fig. 21-22 depicts Rule$_{MPD}$1-2. For example, Monthly installment is a kind of Installment in the ontology. The electronic commerce website wants to offer new installment service and adds Installment (ClassN) in the class diagram. According to Rule$_{MPD}$1, Monthly installment (ClassM) is suggested because Monthly installment is a precise concept of Installment in the ontology.

Rule$_{MPD}$2: There is a possible missing attribute if AttributeX is added in ClassM in ReqE, Concept i in the ontology equals AttributeX, and Concept i has precise (child and part), general (parent and whole), or sibling concepts.

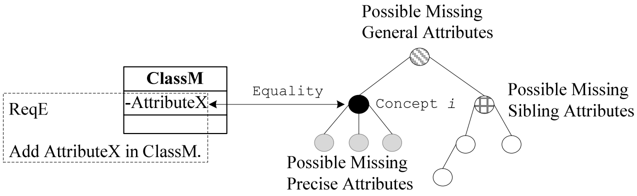

Fig. 21. Possible missing attribute.

V. CONCLUSION

This work proposes a process and a set of rules for conflict analysis in class diagrams to help software engineers to reinforce requirements analysis tasks. Several figures and scenarios are also provided to explain and validate the proposed rules in the preliminary stage.

This work has two advantages and a main limitation. In the first advantage, clear rules in this work can help novices to design class diagrams more easily. The second advantage is that structured domain knowledge can be stored in the ontology. Structured domain knowledge and rules can facilitate automatic conflict detection by means of knowledge-based systems. The main limitation is the ontology maintenance effort. Stakeholders should maintain the shared domain knowledge in the ontology together.

REFERENCES

[1] J. He and W.R. King, "The Role of User Participation in Information Systems Development: Implications from a Meta-analysis," *Journal of Management Information Systems*, vol. 25, no. 1, pp. 301–331, summer 2008.

[2] J. Savolainen and T. Männistö, "Conflict-Centric Software Architectural Views: Exposing Trade-Offs in Quality Requirements," *IEEE Software*, vol. 27, no. 6, pp. 33-37, Nov./Dec. 2010.

[3] A. Egyed, "Instant Consistency Checking for the UML," In *Proc. of 28th International Conference on Software Engineering*, New York, 2006, pp. 381–390.

[4] Y. Nomaguchi and K. Fujita, "DRIFT: A Framework for Ontology-based Design Support Systems," In: *Proc. of Semantic Web and Web 2.0 in Architectural, Product, Engineering Design Workshop*, Aachen, 2007, pp. 1-10.

[5] C.-L. Liu, "CDADE: Conflict Detector in Activity Diagram Evolution Based on Speech Act and Ontology", *Knowledge-Based Systems*, vol. 23, no. 6, pp. 536-546, Aug. 2010.

[6] J. C. Maxwell, A. I. Antón, and P. S. Swire, "A Legal Cross-References Taxonomy for Identifying Conflicting Software Requirements," In *Proc. of IEEE 19th International Requirements Engineering Conference*, New York, 2011, pp. 197-206.

[7] Roth, S., Hauder, M, Michel, F., Münch, D., Matthes, F., "Facilitating Conflict Resolution of Models for Automated Enterprise Architecture Documentation", In *Proc. of the 19th Americas Conference on Information Systems, Chicago*, Berkeley, 2013, pp. 1-11.

[8] S. Mohite, R. Phalnikar, M. Joshi, S. D. Joshi, S. Jadhav, "Requirement and Interaction Analysis Using Aspect-Oriented Modeling," In Proc. of IEEE International Advance Computing Conference, New York, 2014, pp. 1448 - 1453.

[9] P. G. Sapna, H. Mohanty, "Ensuring Consistency in Relational Repository of UML Models," In Proc. of *10th International Conference on Information Technology*, New York, 2007, pp. 217–222.

[10] H. Kaiya, M. Saeki, "Ontology based Requirements Analysis: Lightweight Semantic Processing Approach," In *Proc. of 5th International Conference on Quality Software*, New York, 2005, pp. 223–230.