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 handle four conflict issues: inconsistencies, redundancies, overrides, and missing parts. Scenarios in the electronic commerce context are also provided 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 work 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>
<thead>
<tr>
<th>Document</th>
<th>Conflict Analysis Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maxwell, Antón, and Swire (2011)</td>
<td>Legal Text and software requirements</td>
</tr>
<tr>
<td>Roth et al. (2013)</td>
<td>Enterprise architecture documentation</td>
</tr>
<tr>
<td>Mohite et al. (2014)</td>
<td>UML</td>
</tr>
<tr>
<td>Sapna and Mohanty (2007)</td>
<td>UML</td>
</tr>
<tr>
<td>Kaiya and Saeki (2005)</td>
<td>X is a instance of Y, Subject-Verb-O bject</td>
</tr>
<tr>
<td>Liu (2010)</td>
<td>Activity Diagram</td>
</tr>
</tbody>
</table>
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.

1. Modeling Prior Knowledge
2. Modeling New Requirements
3. Detecting Conflicts
4. Resolving Conflicts

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. RuleID1-11 are proposed in this section. Scenarios are also provided to explain these rules.

RuleID1: There is a requirements generalization inconsistency if ClassM is a superclass of ClassN inReqE, ClassO is a supclass 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

RuleID2: 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

RuleID3: 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 RuleID1 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 ID1. Requirements composition inconsistency
detected by Rule ID2 and aggregation requirements
inconsistency detected by Rule ID3 mean a class is a part and a
whole of another class. Fig. 3 and 4 depicts Rule ID2 and
Rule ID3. For example, Payment service (Class M) is a
 superclass of Near Field Communication (Class N) in Req E
in the payment system. NFC (Class O) is a superclass of
Payment service (Class P) in Req F. Class M equals Class P.
Class N is a synonym of Class O because NFC is the
abbreviation of Near Field Communication. According to
Rule ID1, requirements generalization inconsistency occurs.
The structures of Rule ID1-3 are similar.

**Rule ID4:** There is a method exclusion inconsistency if
AttributeX is added in Class M in Req E, Method I is not
allowed in Class N in Req F, there is an equality, kind, part, or
synonym relationship between Method I and Method J, and
there is an equality, kind, part, or synonym relationship
between Class M and Class N.

Some behaviors in information systems are regulated by
government laws and corporation policies. Method exclusion
inconsistency detected by Rule ID4 and illustrated in Fig. 5
means an undesirable method is added. For example, storing
credit card number() (Method I) is added in credit card
(Class M). Credit card numbers cannot be stored in the
database if the corporation policy because stored credit card
numbers has a security risk about hacking. Therefore
Storing credit card number() (Method J) cannot included in
Any class (Class N) in Req F in the payment system. Method
equals Method I(). Class M is a kind of Class N. According to
Rule ID4, method exclusion inconsistency occurs.

**Rule ID5:** There is a multiple inheritance inhibition
inconsistency if a generalization relationship from Class O to
Class N is added in Req E, there is a generalization
relationship from Class O to Class M in the existing
requirements, and multiple inheritance is not allowed in
Req F.

Some programming languages inhibit multiple inheritance,
such as Java. Rule ID5 detecting multiple inheritance
inhibition inconsistency indicates that more than one
superclass exists in a class diagram. Fig. 6 depicts Rule ID5

**Rule ID6:** There is a generalization and alternative inconsistency if a generalization relationship from Class N to
Class M is added in Req E and there is a equality, part,
antonym, or synonym relationship between concept Class M
and concept Class N in the ontology.

Rule ID6 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 ID6,
Rule ID8 and Rule ID10. The structures of Rule ID6, Rule ID8
and Rule ID10 are similar. For example, the ontology indicates that
Payment service (Class N) is a part of Electronic commerce
website (Class M). According to Rule ID6, adding a
generalization relationship from Payment service (Class N) to
Electronic commerce website(Class M) causes a
generalization and alternative inconsistency.

**Rule ID7:** There is an inverse generalization inconsistency
if a generalization relationship from Class N to Class M is
added in Req E and concept Class N is a kind of concept
Class M in the ontology.

Rule ID7 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 ID7, Rule ID9 and Rule ID11.
The structures of Rule ID7, Rule ID9 and Rule ID11 are similar. For example, the ontology shows NFC (Class M) is a kind of
Wireless connectivity (Class N). According to Rule ID7,
adding a generalization relationship from NFC (Class M) to
Wireless connectivity (Class N) as a kind of NFC causes inverse generalization inconsistency.
RuleID8: 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.]

RuleID9: 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.]

RuleID10: 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.]

RuleID11: 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. RuleID1-6 are proposed and explained as follows.

RuleID1: 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 RuleID1 means two attributes are the same, similar, or overlap. Method redundancy in RuleID2 means two methods are the same, similar, or overlap. The structures of RuleID1 and RuleID2 are similar. Fig. 13 depicts RuleID1 and Fig. 14 depicts RuleID2. For example, Expired date (AttributeX) exists in Credit card (ClassM). According to RuleID1, 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.

RuleID2: 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.]

RuleID3: 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 RuleID3 means the meaning of generalization relationship between two classes in a class diagram does not appear in the ontology. Aggregation relationship redundancy in RuleID4 means the
meaning of aggregation relationship between two classes in a
class diagram does not appear in the ontology. Composition
relationship redundancy in RuleRD5 means the meaning of
composition relationship between two classes in a class
diagram does not appear in the ontology. The structures of
RuleRD3, RuleRD4, and RuleRD5 are similar. Fig. 15-17 depict
RuleRD3-5. 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 RuleRD5, generalization relationship redundancy occurs
and the domain knowledge should be updated in this case.

**RuleRD4**: 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.](image)

**RuleRD5**: 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.](image)

**RuleRD6**: 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.](image)

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. RuleOD1-2 are proposed
discussed as follows.

**RuleOD1**: 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 ClassM in the ontology.

![Fig. 19. Possible override during generalization relationship
addition](image)

RuleOD1 and RuleOD2 shows possible overrides to remind
software engineers about overrides of methods in class
diagrams. Fig. 19-20 depicts RuleOD1-2. In RuleOD1, adding a
generalization relationship reminds software engineers about
possible override. In RuleOD2, 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() (MethodI()) is added in VIP_member (ClassM). Obviously,
Storing_vip_membership_application_form() (MethodI()) is a kind of Storing_membership_application_form()
(MethodI()). According to RuleOD2, MethodJ() (Storing_vip_membership_application_form()) in ClassN
can be renamed as Storing_membership_application_form() to override MethodI() in ClassM.

**RuleOD2**: 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](image)
D. Missing Part Detection Rule

The proposed missing part detection rules use the ontology to find the classes that have been omitted. RuleMPD1-2 are introduced and discussed as follows.

RuleMPD1: 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.

RuleMPD2: 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 class.

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