International Journal of Hybrid Information Technology
Vol. 9, No.10 (2016), pp. 185-200
http://dx.doi.org/10.14257/ijhit.2016.9.10.19

Checking Relationship Consistency and Class Redundancy in a Class Diagram under Model-Driven Engineering

Liang Huang[1], Yucong Duan[2*], Honghao Gao[3], Hui Li[4], Caimao Li[5] and Zhiyang Lin[6]

[1,2,4,5,6]College of Information Science and Technology, Hainan University, China
[3]College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[1]11512460987@qq.com, [2]yucongduan@hotmail.com, [3]gaohonghao@zju.edu.cn, [4]hitlihui1112@163.com, [5]Lilcaim@126.com

Abstract

Model-Driven Engineering (MDE) tries to reduce the effort spent on software development by generating codes from models. People concentrate their minds on the transformation between models and models, or between models and codes. People also concentrate on checking consistency between different models such as consistency between a class diagram and a sequence diagram and consistency between a sequence diagram and a state machine diagram. Checking relationship consistency and class redundancy in a class model is still important but ignored in recent years. This paper concentrates on relationship problems between classes in a class diagram and proposes methods of checking various relationship problems. We address the redundancy of a class's operations and attributes. We identify a large range of the problems for class diagram. Our research is based on the relationship abstraction rules.

Keywords*: class diagram, relationship abstraction, circulation inconsistency, class redundancy*

1. Introduction

A model is an abstract representation of a real system or phenomenon [11]. Models are created to meet particular purposes, for example, to present a human understandable description of some aspect of a system or to present information in a form that can be mechanically analyzed [13]. Models and modeling are essential parts of every engineering endeavors [1]. We build models in order to get a better understanding of the structures and behaviors of our software systems [10]. Well constructed models make it easier to deliver large, complex enterprise systems on time and within budgets [7]. OMG launched the Model Driven Architecture (MDA) as a Model-Driven Engineering (MDE) standard in 2001 [9,12]. The promise of MDE is that the development and maintenance cost can be reduced by working with models instead of codes [14]. In traditional software development, codes are core elements. However software system of today's are increasing more and more complexity, distribution, heterogeneity and lifespan [5]. Traditional software development methods cannot meet the changing requirement of users and cannot catch the essence of the problem area. But in MDA, models become core elements in software development and are not only used as an assistant tool.

Our research on model checking is based on the Unified Modeling Language (UML). UML is a de-facto standard for object-oriented modeling [4]. Relevant information in different UML diagrams [2] under one project should be consistent with each other. In MDE we may need to check consistencies between diagrams.

Checking consistency between two diagrams (CMod). For a given software project (P), stakeholders propose a requirement (R). In order to model P, developers use two different diagrams (D1), (D2). In D1, the information used to describe R is (INFO1). In D2, the

$$\mathrm{CMod}: \forall p \in P, \exists d1 \in D1 \wedge \exists d2 \in D2; \forall r \in R, \exists info1 \in INFO1 \wedge \exists info2 \in INFO2;$$
$$(\mathrm{DIFF}(info1, info2) = \mathrm{True}) \Rightarrow (\mathrm{INCON}(d1, d1) = \mathrm{True});$$

information used to describe R is (INFO2). If INFO1 is different from INFO2 (DIFF), we conclude that D1 is inconsistent with D2 (INCON).

Consistency checking between models is an important part in developing models. Nuseibeh proposed that consistency checking is an activity focusing on comparing information in two or more views [8]. This implies that one important goal of view integration is to provide automatic assistance in identifying view inconsistencies [3].

Three types of inconsistency are identified:

1). *Inconsistency between two class diagrams.* It means the inconsistency between a low level class diagram and a high level class diagram. The high level class diagram is abstracted from the low level class diagram. For this kind of inconsistency, when we check a model using the method *CMod*, one of D1 and D2 is a high level class diagram and the other is a low level class diagram.

2). *Inconsistency between a class diagram and a sequence diagram.* It means the call between classes in sequence diagram doesn't match their relationship in class diagram. When using *CMod* to check that, one of D1 and D2 is a class diagram and the other is a sequence diagram.

3). *Inconsistency between a state machine and a sequence diagram.* It means the states in a state machine don't match the execution sequence in a sequence diagram. For example, an object sends a message to another object in a sequence diagram, but in a state machine diagram there is no state to describe the state change. When using *CMod* to check that, one of D1 and D2 is a state machine diagram and the other is a sequence diagram.

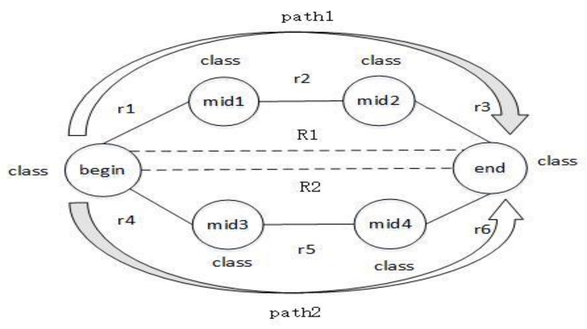


Figure 1. A Form of Circulation

In this paper, we investigate relationship problems in a single class diagram. We also investigate the redundancy of attributes and operations in a class diagram and circulation problems in a class diagram. The circulation may have problems shown in Figure 1. In Figure 1, there are two paths between class *begin* and class *end*. The two paths are *path1* and *path2*. *{ r1, r2, r3 }∈ path1. r1, r2* and *r3* are relationships. If we abstract *r*1, *r*2 and *r*3 in *path1*, we can get a direct relationship *R*1 between class *begin* and class *end*. If we abstract *r*4, *r*5 and *r*6 in *path2*, we can get a direct relationship *R*2 between class *begin* and class *end*. If *R*1 ≠ *R*2, we conclude that *path*1 is inconsistent with *path*2 and the circulation is not correct.

2. Checking Relationships

If the size of a class diagram is small, we can check relevant problems on relationships and classes manually. The checking includes consistency checking, redundancy checking and completeness checking. However, when the software system becomes larger and more complex, we are impossible to find out all the problems manually. Using a tool to discover the problems in a diagram helps us to correct the diagram easily. We find that relationship problems mainly exist in circulations as shown in Figure 1. Figure 2 is an activity diagram indicating the activities in relationship checking. The activities contain circulation finding, relationship consistency checking, completeness checking and redundancy checking. The progress of the activity diagram is:

1). *Using the depth-first search (DFS) method to find out all the circulations in the class diagram*.

2). *Choosing two classes for each circulation*. For the two selected classes, we use relationship abstraction rules to abstract the relationships between them. We get two relationship results *R1* and *R2* of two classes in Figure 1, because there are two paths namely *path1* and *path2* between them.

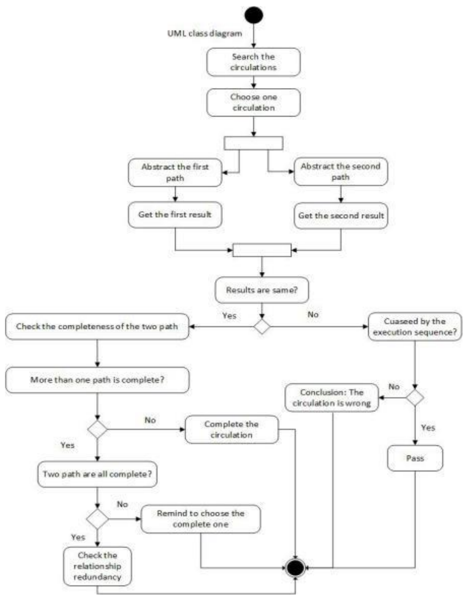

UML class diagram
Search the circulations
Choose one circulation
Abstract the first path
Abstract the second path
Get the first result
Get the second result
Results are same?
Yes
No
Check the completeness of the two path
Cuaseed by the execution sequence?
More than one path is complete?
Conclusion: The circulation is wrong
No
No
Complete the circulation
Yes
Yes
Pass
Two path are all complete?
No
Remind to choose the complete one
Yes
Check the relationship redundancy

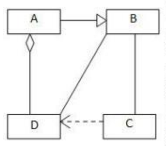
Figure 2. The Activity Diagram of Finding and Checking Circulations

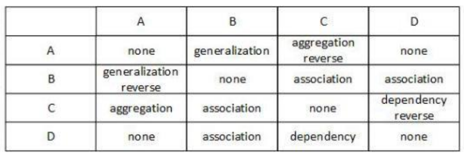
3). *Comparing the two relationship results*. If the two results are identical, we then check the completeness of information for each path. If the information of two paths is complete, we check the relationship redundancy in the circulation. If the information of one path is complete and the other one is not complete, we choose the path in which the information is complete to satisfy the requirements of stakeholders. If both two paths are not complete, we may create a new direct relationship between the two classes.

4). *Finding The reason of their difference*. The first reason may be the different execution order. The second factor may be the path inconsistency. If the relationship difference is caused by the second reason, the circulation is not correct. We ignore the first reason. In the section 3, we will analyze the difference caused by the rules' execution sequences.

2.1. Searching Circulations in a Class Diagram

Figure 3 is a simple class diagram containing circulations. The left part of Figure 3 shows a class diagram. The right part of Figure 3 shows the relationships in the class diagram. We consider the class diagram as a graph. We use the DFS method to get circulations in the class diagram.





	A	B	C	D
A	none	generalization	aggregation reverse	none
B	generalization reverse	none	association	association
C	aggregation	association	none	dependency reverse
D	none	association	dependency	none

Figure 3. A Class Diagram Contains Circulations

2.2. Abstraction and Abstraction Rules

If we want to get the direct relationship between two classes, we need rules to abstract the paths between the two classes. Abstraction contains relationship abstraction and classifier abstraction. Classifier abstraction uses the classification and clustering method to get a high level abstraction. The package diagram in UML introduces the method of using classifier abstraction. Relationship abstraction abstracts the relationships in the class diagram. For example, relationship abstraction finds the relation between two relationships and abstract the two relationships into one. The relationship abstraction uses rules to abstract the class diagram and abstracts a low level class diagram to a higher level class diagram.

1). *Grammars of relationship abstraction and classifier abstraction*. Grammars of classifier abstraction are like *P1:[class x (relationship) x class →class]. For* example, if in a scene class A and class B has a relationship according to grammar *P1*, the scene can be abstracted on class A. The grammar of relationship abstraction is *P2:[(relationship) x class x (relationship) → relationship]*. For example, if in a scene class A has a relationship with class B and class B has a relationship with class C according to *P2*, the scene indicates that class A has a relationship with class C. Figure 4 is a scene describing the relationship between class Human, Mammals and Animals. Human is a kind of mammals and a mammal is a kind of animals. The scene can be described as *S1:[Human x (Generalization) x Mammals x (Generalization) x Animals]*. According to the classifier abstraction grammar*:[Human x (Generalization) x Mammals → Human]*, We abstract the scene S1 and we get the abstracted scene which describes human and animal is *S2:[Human x (Generalization) x Animal]*. If we use the relationship abstraction to abstract the scene, according to the relationship abstraction *rule:[(Generalization) x Class x (Generalization) → Generalization]*, we can get an *Generalization* relationship. And finally we can get the abstracted scene which describes Human and Animal is *S3:[Human x (Generalization) x Animal]*. Relationship abstraction means that the model is abstracted through relationship abstraction rules rather than classifier abstraction rules. The relationship abstraction is used to get a higher-level abstraction of a class diagram and the relationship abstraction is used to abstract paths.

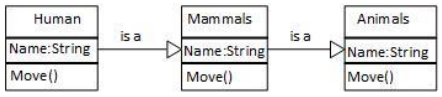


Figure 4. Animal Family Example

2). *Input and output for relationship abstraction and classifier abstraction*. Figure 5 shows the input and output of relationship abstraction and classifier abstraction. The upper part of Figure 5 indicates the input and output for the classifier abstraction. The upper part of Figure 5 matches the grammar of classifier abstraction *P1:[Class x (Relationship) x Class → Class]*. If we input *[Class1 x (Relationship) x Class2]*, the output is *Class1*. The lower part of Figure 5 indicates the input and output for relationship abstraction. The lower part of matches the grammar of relationship abstraction *P2:[(Relationship) x Class x (Relationship) →Relationship]*. If we input *[(Relationship1) x Class x (Relationship2)]*, the output is *Relationship1 or Relationship2*. The output of a relationship abstraction depends on the relationship abstraction rules.

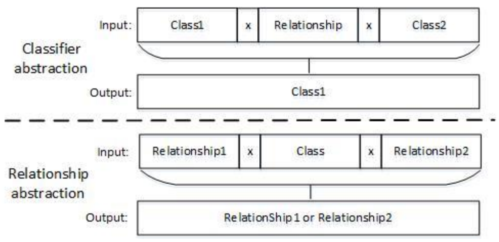
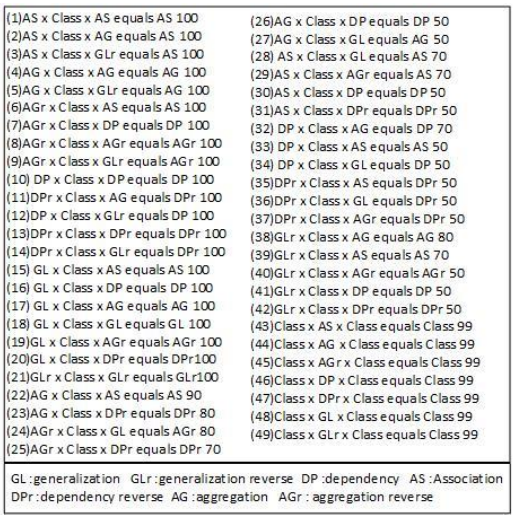


Figure 5. Input and Output for Abstraction



(1)AS x Class x AS equals AS 100
(2)AS x Class x AG equals AS 100
(3)AS x Class x GLr equals AS 100
(4)AG x Class x AG equals AG 100
(5)AG x Class x GLr equals AG 100
(6)AGr x Class x AS equals AS 100
(7)AGr x Class x DP equals DP 100
(8)AGr x Class x AGr equals AGr 100
(9)AGr x Class x GLr equals AGr 100
(10) DP x Class x DP equals DP 100
(11)DPr x Class x AG equals DPr 100
(12)DP x Class x GLr equals DP 100
(13)DPr x Class x DPr equals DPr 100
(14)DPr x Class x GLr equals DPr 100
(15) GL x Class x AS equals AS 100
(16) GL x Class x DP equals DP 100
(17) GL x Class x AG equals AG 100
(18) GL x Class x GL equals GL 100
(19)GL x Class x AGr equals AGr 100
(20)GL x Class x DPr equals DPr100
(21)GLr x Class x GLr equals GLr100
(22)AG x Class x AS equals AS 90
(23)AG x Class x DPr equals DPr 80
(24)AGr x Class x GL equals AGr 80
(25)AGr x Class x DPr equals DPr 70
(26)AG x Class x DP equals DP 50
(27)AG x Class x GL equals AG 50
(28) AS x Class x GL equals AS 70
(29)AS x Class x AGr equals AS 70
(30)AS x Class x DP equals DP 50
(31)AS x Class x DPr equals DPr 50
(32) DP x Class x AG equals DP 70
(33) DP x Class x AS equals AS 50
(34) DP x Class x GL equals DP 50
(35)DPr x Class x AS equals DPr 50
(36)DPr x Class x GL equals DPr 50
(37)DPr x Class x AGr equals DPr 50
(38)GLr x Class x AG equals AG 80
(39)GLr x Class x AS equals AS 70
(40)GLr x Class x AGr equals AGr 50
(41)GLr x Class x DP equals DP 50
(42)GLr x Class x DPr equals DPr 50
(43)Class x AS x Class equals Class 99
(44)Class x AG x Class equals Class 99
(45)Class x AGr x Class equals Class 99
(46)Class x DP x Class equals Class 99
(47)Class x DPr x Class equals Class 99
(48)Class x GL x Class equals Class 99
(49)Class x GLr x Class equals Class 99

GL :generalization GLr :generalization reverse DP :dependency AS :Association
DPr :dependency reverse AG :aggregation AGr : aggregation reverse

Figure 6. The Rules for Abstraction

3). *Abstraction rules and reliability*. UML relationships include dependency, aggregation, association, generalization and composition. The former four kinds of relationships are frequently used. The combination with repetition of any two relationships can reflect in an input structure. And the input structure belongs to relationship abstraction input structure. If two classes are connected with a relationship, a combination of the two classes and relationship can reflect on an input structure. And this input structure belongs to the classifier abstraction input structure. Figure 6 shows relevant rules for relationship abstraction and classifier abstraction. The part before "equals" in each rule is similar to the part before "→" in grammar. The part meets the input structure. The part after "equals" is similar to the part after "→" in grammar. The part meets with the output structure. The number at the end of the rule shows the reliability of the abstraction rule. We get the final reliability after running two or more rules. For example, the reliability of rule (23) is 80 and the reliability of rule (30) is 50. The ultimate reliability is 80*50/100=40. The lower part of Figure 6 shows the short hands for the relationships. In Figure 6, there are 42 relationship abstraction rules and 7 classifier abstraction. The rules ranging from (1) to (42) belong to relationship abstraction rules. The rules ranging from (43) to (49) belong to classifier abstraction rules. In the relationship abstraction rules, there are 21 rules ranging from (1) to (21) whose reliability are 100, taking nearly half of the relationship abstraction rules.

2.3. Problems in Circulation

A relationship circulation in a class diagram requires that there are two paths from one class to another class. If class A wants to get the information of class B, there will be two paths for class A to choose from. Circulation often contains problems. For example, we may select the wrong path. We get two relationships between class A and class B after we abstract the two paths. If the two relationships are significantly different, the diagram contains inconsistency. Another problem is relationship redundancy in the circulation.

1). *Relationship consistency:*

Check relationship consistency in a circulation (CkCir). We suppose that class diagram (D) contains circulation (C). C contains two classes (CL1) and (CL2). Between CL1 and CL2, there are two paths (P1) and (P2). Using abstraction rules (Abs), we abstract P1 and get a direct relationship (R1) between CL1 and CL2. We abstract P2 and get a direct relationship (R2) between CL1 and CL2. If R1 ≠ R2, we say that P1 is inconsistent with P2 (Inc).

$$\begin{aligned}
&\mathrm{Ckcir}: \forall d \in D, \exists c \in C; \forall cl1 \in CL1, \forall cl2 \in CL2; \\
&\exists p1 \in P1, \exists p2 \in P2; \exists r1 \in R1, r1 = \mathrm{Abs}(p1); \exists r2 \in R2, r2 = \mathrm{Abs}(p2); \\
&(r1 \neq r2) \Rightarrow (\mathrm{Inc}(p1,p2) = \mathrm{True}).
\end{aligned}$$

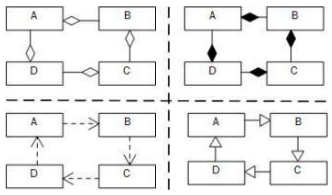


Figure 7. The Circulations

Figure 7 shows 4 kinds of circulation with relationship inconsistency problems. These circulations are aggregation circulation, composition circulation, dependency circulation and generalization circulation. For each kind of circulation, there are two relationship results between class A and class C and the two relationships are different. Take the aggregation circulation as an example, if we use the relationship abstraction rule (4) to abstract *path1:[A x (AG) x D x (AG) x C]*, we get *result1:[A x (AG) x C]*. However, if we use rule (4) to abstract *path2:[A x (AGr) x B x (AGr) x C]*, we get *result2:[A x (AGr) x C]. result1 ≠ result2*, so *path1* is inconsistent with *path2*. Figure 8 shows a simple class diagram of the ATM system. The class diagram contains four classes namely Session, CustomerInformation, TransactionRecord and Transaction. According to *rule (32):[(DP) x Class x (AG) equals DP 70]* given in Figure 8, we abstract the scene as *S1:[Session x (DP) x CustomerInformation x (AG) x TransactionRecord]*. We get result *r1:[Session x (DP) x TransactionRecord]*. The reliability of *r1* is 70. The description of *d1:[(AGr) x Class x (DPr)]* is the same as the description of *d2: [(DP) x Class x (AG)]*. According to *rule (32):[(DP) x Class x (AG) equals DP 70]*, we abstract the scene *S2:[Session x (AGr) x Transaction x (DPr) x TransactionRecord]*. We get result *r2:[Session x (DPr) x TransactionRecord]*. The reliability of *r2* is 70. We conclude that *S1* is conflict with *S2* since *r1≠r2*. The circulation contains inconsistency problem.

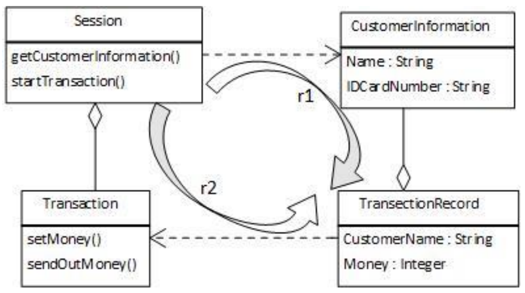


Figure 8. The ATM System Example

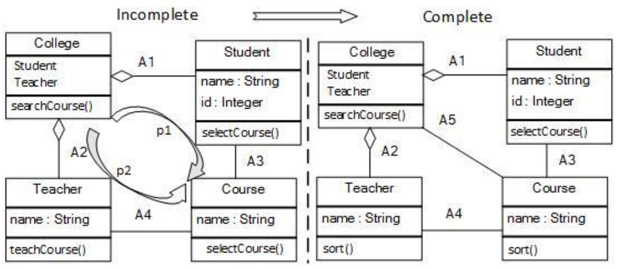


Figure 9. A Relationship Circulation Contains Both Aggregation and Association

2). *Completeness of information in circulation.* The completeness problem here mainly refers to the reference integrity. For example, in order to meet the requirements of stakeholders, class A need to get the information in class C. But class A is not directly connected to class C. Class A is connected to class B and class B is connected to class C. So class A gets the information in class C through class B. But sometimes the information is not correct or complete, so the information cannot satisfy all relevant requirements. This kind of problem is the completeness problem. The completeness problem is quite common in circulation. Class diagram in Figure 9 shows relationships between class College, Student, Teacher and Course. Suppose there is a need to find out all the courses in a college. In Figure 9 (left) we find two paths from College to Course. They are *p1:[College x (AGr(A1)) x Student x (AS(A3)) x Course]* and *p2:[College x (AGr(A2)) x Teacher x (AS(A4)) x Course]*. The courses we get may not only belong to the college, because students in college can choose the courses of other colleges. If we choose *p2*, the courses we get may not only belong to the college for there are some visiting professors in the college. Visiting professors may teach some lessons for college students, but they also teach lessons that do not belong to the college. We get a result that no matter through which path *p1 or p2*, we cannot fully satisfy the requirement. The result is a completeness problem in the circulation. When dealing with this kind of problem, we can add a direct relationship between the two classes. For Figure 9, we can add a direct relationship between class College and Course shown in Figure 9(Right). Using *rule (2):[AS x Class x AG equals AS 100]*, we abstract *p1* and get result *r1:[College x AS x Course]*. We abstract *p2* and get result *r2:[College x AS x Course]*. *r1 = r2* and the relationship between a college and a course is *AS* in *r1* and *r2*. The type of the direct relationship (A5) between class College and Student is *AS a*ccording to *r1 and r2.*

3). *Relationship redundancy in circulation.* Redundancy in circulation means that some relationships are redundancy. Relationship redundancy is also common in a circulation. Relationship redundancy does not cause software errors, but waste the resources such as time and space. We can save resources if we identify and reduce them. Figure 10 shows the relationship redundancy in circulation in Amazon Web Services. Figure 10 illustrates class AwsService, AwsServiceDefinition, AmazonWhishListService and AmazonWhishListServiceDefinition. According to *rule (17):[(GL) x Class x (AG) equals*

AG 100], we abstract *p1: [AmazonWhishListServiceDefinition x (GL) x AwsServiceDefinition x (AG) x AwsService]* and get result *r1:[AmazonWhishListServiceDefinition x (AG) x AwsService]* with reliability 100. According to *rule (27):[(AG) x Class x (GL) equals AG 50]*, we abstract *p2: [AmazonWhishListServiceDefinition x (AG) x AmazonWhishListService x (GL) x AwsService]* and get result *r2:[AmazonWhishListServiceDefinition x (AG) x AwsService]* with reliability 50. After checking the information completeness of two paths between AmazonWhishListServiceDefinition and AwsService, we find that the information is complete. Because *r1=r2*, we think that there may be relationship redundancy in the circulation. Because the reliability of *r1* is bigger than *r2*, we suggest that redundancy is existing in *p2*. We check *p2* and speculate that the relationship AG in *p2* is redundant and delete AG. Then we check information completeness again. We finally find that there are no problem in the class diagram so we are sure that AG in *p2* is a relationship redundancy.

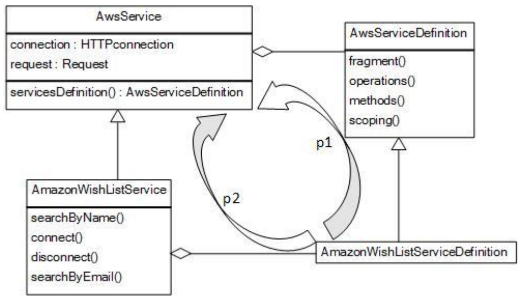


Figure 10. The Class Diagram of Amazon Web Services

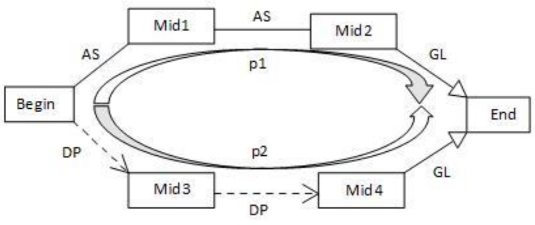


Figure 11. A Circulation Example

2.4. Checking Problems in Circulation

Figure 11 is a simple circulation. In the circulation, between class *Begin* and class *End* there are two paths. In order to obtain the relationship between class *Begin* and class *End*, we need to abstract *p1* and *p2*. In order to abstract the two paths and apply the abstraction rules automatically, we use an automaton. Figure 12 shows the method of using an automaton to abstract the two paths. The upper part of Figure 12 shows the automaton for abstracting *p1*. State *S0* is the initial state. We describe *S0* as *S0:[(Begin) x (AS) x (Mid1) x (AS) x (Mid2) x (GL) x (End)]*. Firstly, we use rule (1) to abstract *S0*, then the state S0 changes to *S1*. *S1* can be described as *S1:[(Begin) x (AS) x (Mid2) x (GL) x (End)]*. Secondly, we use rule (28) to abstract *S1* and the state *S1* changes to *S2*. *S2* can be described as *S2:[(Begin) x (AS) x (End)]*. The reliability of *S2* is *100*70/100=70*. The relationship between class *Begin* and class *End* is AS through *p1* and the reliability of the relationship is 70. The lower part of Figure 12 shows the automaton for abstracting *p2*. *S3* is the initial state of *p2*. *S3* can be described as *S3:[(Begin) x (DP) x (Mid3) x (DP) x (Mid4) x (GL) x (End)]*. Firstly, we use rule (10) to abstract *S3* and get *S4:[(Begin) x*

(DP) x (Mid4) x (GL) x (End)]. Secondly, we use rule (34) to abstract *S4* and get *S5:[(Begin) x (DP) x (End)]*. The relationship between class *Begin* and class *End* is DP through *p2*. And the reliability of the relationship of *S5 is 100*50/100=50*. After using the automaton to abstract the two paths, The relationship between class *Begin* and class *End* in *p1* is AS, in *p2* is DP. (AS)≠(DP), so *p1* is inconsistent with *p2*.

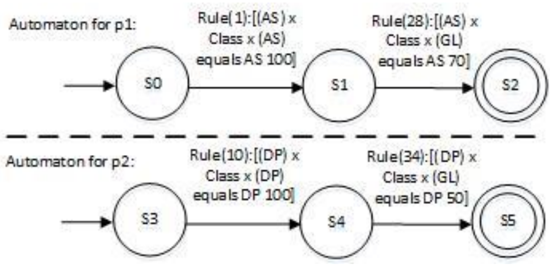

Automaton for p1:
Rule(1):[(AS) x Class x (AS) equals AS 100]
Rule(28):[(AS) x Class x (GL) equals AS 70]
S0
S1
S2
Automaton for p2:
Rule(10):[(DP) x Class x (DP) equals DP 100]
Rule(34):[(DP) x Class x (GL) equals DP 50]
S3
S4
S5

Figure 12. Automata for Abstracting Paths

3. Inconsistency in a Path

In the method of using automaton, we apply rules in an iterative way. Through the iterative approach, the final reliability may not be biggest. So we need to find out a new running order to get the biggest reliability. There are two execution orders after we run two rules. Running the first rule firstly or running the second rule firstly leads to two different execution orders. Different execution orders may lead to different results. Figure 13 shows the two execution orders. The lower part of Figure 13 shows the first execution order. The lower part of Figure 13 executes the *rule:[(RelationshipA) x Class2 x (RelationshipB) equals RelationshipD1]* firstly and executes the r*ule:[(RelationshipD1) x Class2 x (RelationshipC) equals RelationshipE1]* secondly. The ultimate relationship of first order between *Class1* and *Class4* is *RelationshipE1*. Reliability of *RelationshipE1* is *FinalReliabilitya*. The upper part of Figure 13 shows the second execution order. The execution apply the *rule:[(RelationshipB) x Class3 x (RelationshipC) equals RelationshipD2]* firstly and apply the *rule:[(RelationshipA) x Class2 x (RelationshipD2) equals Re- lationshipE2]* secondly. The ultimate relationship of second execution order between *Class1* and C*lass4* is *RelationshipE2*. Reliability of *RelationshipE2* is *FinalReliabilityb*. The *RelationshipE1* and *RelationshipE2* may be different. At the same time the *FinalReliabilitya* and *FinalReliabilityb* may be different too. If *RelationshipE1* ≠ *RelationshipE2*, the path contains relationship inconsistency. If *FinalReliabilitya* ≠ *FinalReliabilityb*, the path contains reliability inconsistency.

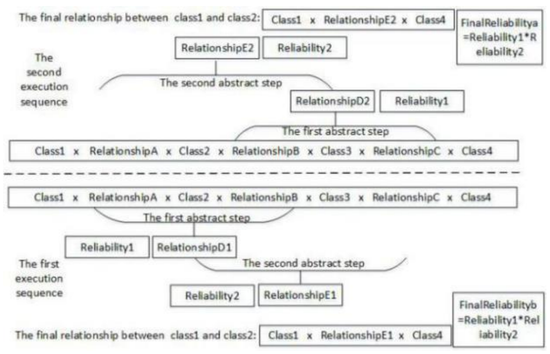

The final relationship between class1 and class2:
Class1 x RelationshipE2 x Class4
FinalReliabilitya =Reliability1*Reliability2
RelationshipE2
Reliability2
The second execution sequence
The second abstract step
RelationshipD2
Reliability1
The first abstract step
Class1 x RelationshipA x Class2 x RelationshipB x Class3 x RelationshipC x Class4
Class1 x RelationshipA x Class2 x RelationshipB x Class3 x RelationshipC x Class4
The first abstract step
Reliability1
RelationshipD1
The second abstract step
The first execution sequence
Reliability2
RelationshipE1
FinalReliabilityb =Reliability1*Reliability2
The final relationship between class1 and class2:
Class1 x RelationshipE1 x Class4

Figure 13. Two Execution Orders

3.1. Reliability Inconsistency

The inconsistency between *FinalReliabilitya* and *FinalReliabilityb* belongs to reliability inconsistency. Figure 14 is a scene describes the relationship between class *Vegetable*, *Human*, *Mammal* and *Animal*. We can describe the scene as *S:[Vegetable x (AS) x Human x (GL) x Mammal x (GL) x Animal]*. Two relationship abstraction rules in *S* are *rule (28):[(AS) x class x (GL) equals AS 70]* and *rule (18):[(GL) x class x (GL) equals GL 100]*. We use the automaton to abstract *S* iteratively, as the first sequence in Figure 13 shows. Firstly, we apply rule (28) and get the output scene *S1:[Vegetable x (AS) x Mammal x (GL) x Animal]*. Secondly, we still run rule (28) to abstract *S1* and get the output scene *S2:[Vegetable x (AS) x Animal]*. The reliability of *S2* is 70*70/100=49. However, if we firstly run the rule of which reliability is higher, the results may be different. For the rules in Figure 14, we find that the reliability of rule (18) is 100 and the reliability of rule (28) is 70. The reliability of rule (18) is higher than the reliability of rule (28), so we run rule (18) to abstract *S* firstly. The second sequence shown in Figure 13. We get the output scene *S3: [Vegetable x (AS) x Human x (GL) x Animal]*. We run rule (28) to abstract *S3* secondly and get the output scene *S4:[Vegetable x (AS) x Animal]*. Finally the reliability of *S4* is 100*70/100=70, the value is obviously higher than 49. Algorithm 1 shows the method of applying the rules of which the reliability is higher firstly and through this method we can get a more reliable result.

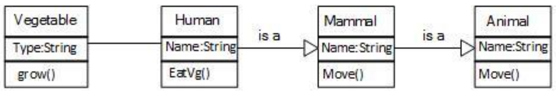


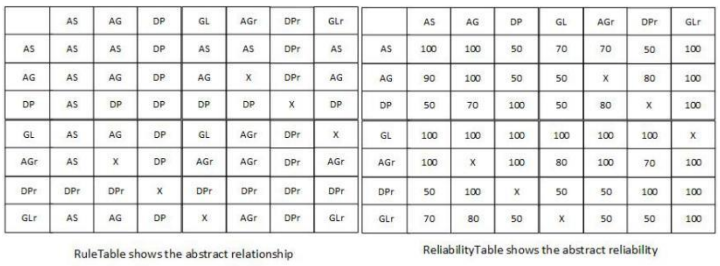
Figure 14. A Sample for the Analysis of Rule Running Sequence

Algorithm 1 Get a More Reliable Result

```
Require: A path begins from class A and ends with class B
Ensure: Relationship between class A and class B and the reliability
1: while The final relation between class A and class B is not found. do
2:     Find all the relationship rules in the path.
3:     Rank the relationship rules from big to small according to the reliability.
4:     Choose the highest reliability rule and abstract the path.
5: end while
6: Output the relationship between class A and class B and the reliability
```

3.2. Relationship Inconsistency

The result we get after abstracting the scene shown in Figure 14 only contains the reliability inconsistency. The consequences of the two relationships are same. The scene does not include relationship inconsistency. However, if *S2≠S4*, we come to a conclusion that different execution orders have influence on relationship inconsistency. We should consider whether the relationship inconsistency is caused by different execution orders or caused by inconsistency between two paths. If the relationship inconsistency is caused by the different execution orders, we need to eliminate the impact of different execution orders. If the relationship inconsistency is caused by inconsistency between two paths, we draw a conclusion that there are some problems in circulation and we need to modify the information in two paths. In order to eliminate the influence of different execution orders, we need to get all scene types which contain 3 relationships. A scene contains 3 relationships and contains 2 rules. A scene contains two rules and allows two different execution orders. And after analyzing the two execution orders in each scene, we can know the impact of different execution orders on circulation checking results. If different execution orders have no impact on scenes which contain two rules, execution orders will not affect the scenes which contain 4 or more relationships. The form of the scene is *S:[Class x (Relationship1) x Class x (Relationship2) x Class x (Relationship3) x Class]*. *Relationship1, Relationship2 and Relationship3* are selected from relationship collection C={ AG, GL, DP, AS, AGr, DPr, GLr }. So if we choose one relationship from C and choose three times, we can get all scenes which contains 2 rules, we find that there are 7 relationships in C, so there are 7*7*7=343 scenes require analyzing. For each scene type, we compare the two results got from two execution methods. The left of Figure 15 shows the abstracted relationship between row and column. For example, the result (RuleTable[AS][AG]=AS) is built according to *rule (2):[(AS) x Class x (AG) equals AS 100]*. The right of Figure 15 shows the reliability of each abstraction rule. By searching the two tables we can get the results of two execution orders for 343 scene types.Figure 16 is a



	AS	AG	DP	GL	AGr	DPr	GLr
AS	AS	AS	DP	AS	AS	DPr	AS
AG	AS	AG	DP	AG	X	DPr	AG
DP	AS	DP	DP	DP	DP	X	DP
GL	AS	AG	DP	GL	AGr	DPr	X
AGr	AS	X	DP	AGr	AGr	DPr	AGr
DPr	DPr	DPr	X	DPr	DPr	DPr	DPr
GLr	AS	AG	DP	X	AGr	DPr	GLr

RuleTable shows the abstract relationship

	AS	AG	DP	GL	AGr	DPr	GLr
AS	100	100	50	70	70	50	100
AG	90	100	50	50	X	80	100
DP	50	70	100	50	80	X	100
GL	100	100	100	100	100	100	X
AGr	100	X	100	80	100	70	100
DPr	50	100	X	50	50	100	100
GLr	70	80	50	X	50	50	100

ReliabilityTable shows the abstract reliability

Figure 15. Rule Table

flow chart used to analyze the 343 scene types. Firstly, we select three relationships *{R1, R2, R3}* from the relationship collection C={ AG, GL, DP, AS, AGr, DPr, GLr } in order. And the relationships *{ R1, R2, R3 }* can be repetitively selected. And there are 343 kinds of combinations of *{ R1, R2, R3 }* in order. A *{ R1, R2, R3 }* order reflect a scene, for a scene, there are two execution orders. We make one order apply *rule:[R1 x Class x R2 equals R4]* firstly and apply *rule:[R4 x Class x R3 equals Ra]* secondly. The application of rules uses the method of checking the rule table and reliability table as shown in Figure 15. We make the another order apply *rule:[R2 x Class x R3 equals R5]* firstly and apply *rule:[R1 x Class x R5 equals Rb]* secondly. Finally the two results of the two orders we got are *Ra and Rb*. If *Ra = Rb*, for the circulation composed of this scene type, different execution orders will not result in relationship inconsistency in circulation. If the 343 scene types are not all analyzed, we choose another combination of *{ R1, R2, R3 }* to analyze.

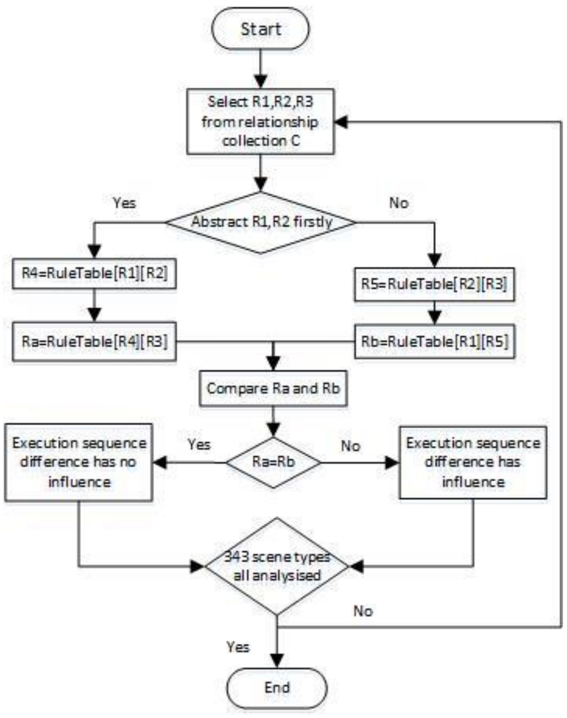

Start
Select R1,R2,R3 from relationship collection C
Abstract R1,R2 firstly
Yes
No
R4=RuleTable[R1][R2]
Ra=RuleTable[R4][R3]
R5=RuleTable[R2][R3]
Rb=RuleTable[R1][R5]
Compare Ra and Rb
Ra=Rb
Execution sequence difference has no influence
Execution sequence difference has influence
343 scene types all analysised
End

Figure 16. Execution Orders Flowchart

Table 1. Comparison Results

Comparison of two results	Number	Percentage
Equal	255	0.74
Unequal	0	0.00
Invalid	88	0.26
Total	343	1

Table 1 shows the result of the experiment. The number of equal type equals to 255. The number of unequal type equals to 0. The number of invalid type equals to 88. For the equal type and the invalid type we show an example of each type as shown in Figure 17. The top of Figure 17 shows a case that the two results got from two execution orders are equal. The two execution orders are shown in Figure 13. The bottom of Figure 17 shows an invalid type.

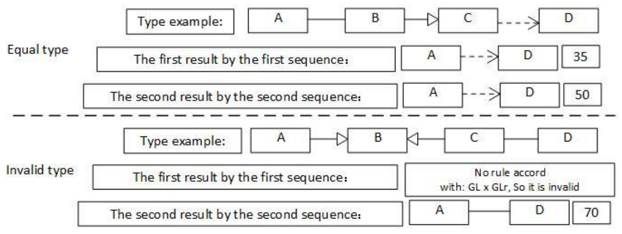

Equal type
Type example:
A
B
C
D
The first result by the first sequence:
35
The second result by the second sequence:
50
Invalid type
No rule accord with: GL x GLr, So it is invalid
70

Figure 17. Examples of Different Result Types

After the experiment, we note that there are no cases of unequal type in which the two results of the two execution orders are different. For the invalid type, one execution order is invalid. The bottom of Figure 17 shows a case of invalid type. For this case, we cannot abstract the scene *S:[A x (GL) x B x (GLr) x C x (AS) x D]* by the first execution order, because there is no rule for the *scene:[A x (GL) x B x (GLr) x C]*. So for invalid type, we have only one execution order and can only get a relationship result. So for a circulation contains this kind of scene execution orders will not cause the relationship inconsistency. We conclude that different execution orders will not affect the relationship results of abstraction. In the circulation, If results of abstracting two paths are different, we come to a conclusion that there exists relationship inconsistency between the two paths and the circulation has problems.

4. Reducing Redundancy and Abstracting a Class Diagram

4.1. Checking Redundancy

If classes in a class diagram have many similarities, redundancy may exist in the class diagram. For example, if two or more classes have same attributes and operations, the same attributes and operations may be redundancy. Figure 18 shows the classes that we used in drawing a graph. The classes contain Ellipse, Polygon, Polyline, PolygonConnector and

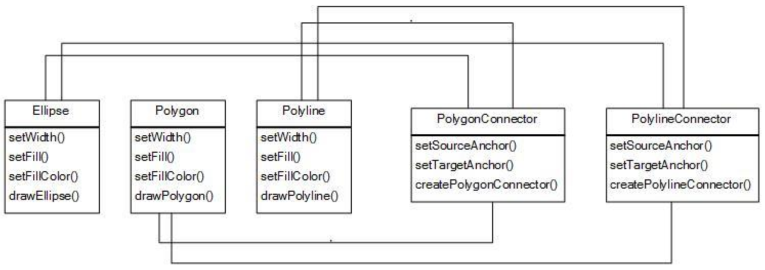


Figure 18. A Simple Graphical Class Diagram

PolylineConnector. Ellipse, Polygon and Polyline has similar operations. We take their same operations out and put them in a new class Shape. Then we make class Ellipse, Polygon and Polyline inherit class Shape. We do the same thing with class PolygonConnector and class PolygonConnector. If the limit of a operation is different, we choose the largest number of limit. For example, if the method *setWidth()* in Ellipse is private, in Polygon is public and in Polyline is public, the number of the limit which is public equals to 2, the number of the limit which is private equals to 1. The largest number of limit equals to 2. So we set the limit of the method *setWidth()* in Shape public. Through taking same attributes and operations in different classes out, we can reduce redundancy in class diagram. As Figure 18 shows that we needn't code *setWidth(), setFill()* and *setFillColor()* three times in class Ellipse, Polygon and Polyline. The relationship between class Shape and class Connector is based on the relationships between class Ellipse, Polygon, Polyline and class PolygoConnector, polylineConnector.

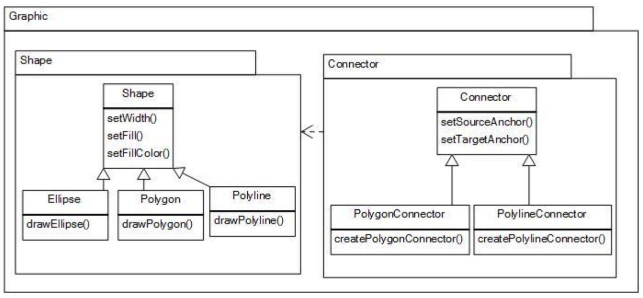


Figure 19. The Class Diagram after Reducing Redundancy

4.2. Classifying Classes and Abstracting a Class Diagram

A package in UML can be viewed as a folder if classes are viewed as files. Package is used to classify classes. In order to get the organizational layer of a software and help us understand the software better, packages are put in place to get the big picture of a software. Figure 20 shows a package diagram abstracted from the class diagram shown in Figure 19. According to the generalization relationships between classes, we can get the scope of a package. But the relationship between packages needs to solve manually.

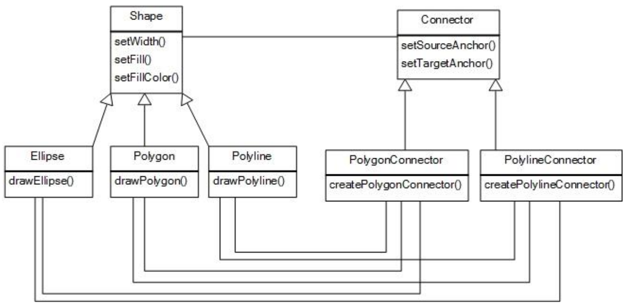


Figure 20. Using a Package to Classify Class Diagrams

5. Conclusions

Model abstraction is an extremely important part in MDE. The abstraction of class diagram takes a big part of model abstraction. The abstraction of a class diagram can help us to understand the software system. We investigate the relationship between classes in a UML class diagram. And we investigate the redundancy of class' attributes and operations. We discover that the circulations in class diagram often cause problems. We investigate the problems in a circulation. These problems include relationship inconsistency, completeness problem and relationship redundancy. We investigate how to check these problems and how to address these problems. In the future, we will investigate the cooperation between models. First, we will study the cooperation cases between class diagrams and sequence diagrams. Second, we will investigate the consistency between class diagrams and sequence diagrams.

Acknowledgments

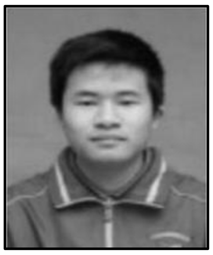
The authors acknowledge the support of the NSFC of China (No. 61363007 and No. 61462022) and Hainan NSF (No. 20156234) , and Postdoctoral Science Foundation of Zhejiang Province of China under Grant No.BSH1502119. * refers to the corresponding author.

References

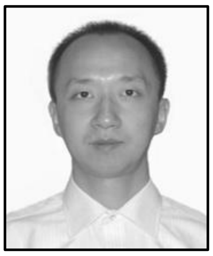
[1] D. Brugali, "Model-driven software engineering in robotics: Models are designed to use the relevant things, thereby reducing the complexity and cost in the field of robotics", IEEE Robotics Automation Magazine, vol. 22, no. 3, **(2015)**, pp. 155–166.
[2] Y. Duan, S.-C. Cheung, X. Fu and Y. Gu, "A metamodel based model transformation approach", in Software Engineering Research, Manage- ment and Applications, 2005. Third ACIS International Conference on. IEEE, **(2005)**, pp. 184–191.
[3] A.F. Egyed, "Heterogeneous view integration and its automation", Ph.D. dissertation, University of Southern California, **(2000)**.
[4] R.B. France, D.-K. Kim, S. Ghosh and E. Song, "A uml-based pattern specification technique", Software Engineering, IEEE Transactions on, vol. 30, no. 3, **(2004)**, pp. 193–206.
[5] P. Grunbacher, A.Egyed and N. Medvidovic, "Reconciling software requirements and architectures: the cbsp approach", in Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on. IEEE, **(2001)**, pp. 202–211.
[6] D. Harel and B. Rumpe, "Modeling languages: Syntax, semantics and all that stu", **(2000)**.

[7] A.G. Kleppe, J. Warmer, W. Bast and M. Explained, “The model driven architecture: practice and promise”, **(2003)**.
[8] B.A. Nuseibeh, “A multi-perspective framework for method integra- tion,” Ph.D. dissertation, Imperial College, **(1994)**.
[9] M.OMG, “Guide version 1.0. 1”, Object Management Group, vol. 62, **(2003)**, pp. 34.
[10] P. J.Puczynski, “Checking consistency between interaction diagrams and state machines in uml models”, **(2012)**.
[11] H. Schichl, “Models and the History of Modeling”, Springer US, **(2004)**.
[12] R. Soley, D. Frankel, J. Mukerji and E. Castain, “Model driven architecture-the architecture of choice for a changing world”, Technical report, OMG, http://www. omg. org, Tech. Rep., **(2001)**.
[13] H. Stachowiak, “Allgemeine modelltheorie”, Zitiert auf, **(1973)**,p p. 35.
[14] A. Van.Deursen, E. Visser and J. Warmer, “Model-driven software evolution: A research agenda”, Delft University of Technology, Software Engineering Research Group, Tech. Rep., **(2007)**.

Authors



Liang Huang, he is learning Computer Science in Hainan University. His research interests include software engineering, service-oriented computing and big data.



Yucong Duan, he received his PhD in software engineering from Institute of Software, Chinese Academy of Sciences, China in 2006. He is currently a full professor and vice director of Computer Science Department in Hainan University, P.R.China. His research interests include software engineering, service computing, cloud computing and big data. He is a member of IEEE, ACM and CCF (China Computer Federation).

Honghao Gao, he received his PhD degree in computer application technology from the School of Computer Engineering and Science of Shanghai University, Shanghai, China, in 2012. His research interests include Web service and model checking.

Hui Li, he received his PhD degree of communication and information system in Department of Electronics and Information Engineering from Harbin Institute of Technology in 2006. He is currently working as a professor in Hainan University. His research interests include advanced wireless networks, space communication and maritime communications.

Caimao Li, he received his master's degree in computer application technology from Southeast University in 2004. In 2010, he became an Associate Professor in Department of Computer Science and Technology of Hainan University. His research interests include software engineering, object oriented technology, information security, trusted computing, cloud computing, service computing, web technology and application.

Zhiyang Lin, he received his master's degree in communication and information systems from Hainan University, Haikou, P.R China, in 2008. He is currently teaching in the College of Information Science and Technology in Hainan University. His research interests include mobile communication, image processing and mobile information technology.