Using the Metamodel Mechanism To Support Class Refinement

Wuwei Shen and Weng Liong Low
Department of Computer Science, Western Michigan University
{wwshen,wllow}@cs.wmich.edu

Abstract

With modern software development being a complicated process, refinement has become an inevitable step in software development. To date, however, supporting a refinement process during software development has not received much attention in the research community. In this paper we present a metamodel mechanism which can support class diagram refinement based on a set of proposed rules, which is represented by a class diagram together with an Object Constraint Language (OCL) constraint. By applying our existing tool supporting the metadata architecture, a new tool can be generated to help software developers find some discrepancies between two class diagrams during software refinement. A web-based online learning system is illustrated in this paper to show how the metamodel mechanism can be applied to find discrepancies between the two models at two different levels.

1 Introduction

It is impossible for modern software development to take only one step from software requirements to a final software system. Software developers must apply a divide and conquer technique to software development, which means that complex software requirements should be broken up into a number of smaller and more controllable pieces, each of which can be further refined until an executable system can be derived. Finally all executable systems should be combined together to form a final software system. During the process to apply the divide and conquer technique, software refinement plays an important role.

During the software refinement, different artifacts, each of which usually concentrates on one aspect of software requirements, are generated. Since the different artifacts concentrate on different levels of details, maintaining the consistency between any two consecutive artifacts is crucial for software developers when they design a software system. Experience shows that it can save much more money, time and labor if errors between two consecutive artifacts can be detected in the earlier phase of software development instead of the later phase of software development.

Among the artifacts generated during software refinement, class diagrams are the most important since they represent some static structure of a software system. Different levels of class diagrams represent different levels of details which software developers are interested in. When two class diagrams at two different phases of development are designed, it is most likely to introduce some semantic discrepancies. Experience shows that some discrepancies are a major factor to crash a final system. So software developers should be aware of the discrepancies in their design.

A tool which helps software developers realize discrepancies can have the following benefits:

- It supports to find errors during software development. Some discrepancies are caused by developers’ mistakes. This kind of discrepancy is called a negative discrepancy which software developers should care about. It is the negative discrepancy that can crash the final software system. The goal of this research is to help software developers remove these negative discrepancies during their software development.

- It aids developers to know that their development is on the right track because some discrepancies are caused by the refinement process from an imprecise problem to a precise solution. Due to the limitation of the human brain, we cannot precisely consider the whole problem at the very beginning. As the refinement process continues, the more details we can consider, the more precise a model we can design. Sometimes, a discrepancy between a precise model and an imprecise model is not harmful. On the contrary, it indicates the progress made by software developers and we call it a positive discrepancy.

When software developers get overwhelmed with hundreds of classes, it is really hard for them to observe some semantic discrepancies between two sets of classes. While developers are eager to have such a tool supporting their software development, the current techniques are not good...
enough to support building this kind of tool. The main challenging issues come from the following aspects.

1. How to define refinement rules. Software development is a complicated process and different domain applications may have different methodologies while they are based on the object-oriented development. Because of different needs at different times during the refinement process, the refinement rules can have different versions even during one software development. How to support the varying refinement rules by one tool is a really challenging issue.

2. How to avoid a complicated path search. When a class diagram is mapped to a more complicated class diagram, whether the set of classes in the two diagrams still keeps the similar relation is an important problem. With the detailed class diagram having many intricate relations, the discrepancy checking has become a complicated graph search problem.

In this paper, we will introduce a new method which is based on the four-layer architecture, which UML is defined in terms of, to generate a tool which can report to software developers some semantic discrepancies. In the four-layer architecture, the M0-layer defines a specific information model which is an instance of a software model (M1-layer) designed by software developers. Any M1-layer model is regarded as an instance of the UML metamodel, i.e., an M2-layer model in the four-layer architecture. This method is called the metamodel mechanism.

Using the instance-of relation between an M1-layer model and an M2-layer model, we can check whether the M1-layer model is valid based on the M2-layer model. For instance, the UML metamodel is defined by a set of class diagrams together with some well-formedness rules so any M1-layer model can be checked for syntactic correctness. It is the UML metamodel that makes it possible for software developers to find some errors in one model. But in this research we extend the UML metamodel to support refinement rules, i.e., finding the discrepancy between two different class diagrams when they are regarded as an instance of the extended UML metamodel. Using the metamodel mechanism, we can solve the above challenging issues when supporting software refinement.

The metamodel mechanism supporting class refinement can properly solve the first challenging issue. Depending on a set of refinement rules, users of this research can provide their extended UML metamodel which consists of a class diagram with some OCL constraints. Consequently, they can find whether an instance, i.e., two class diagrams, is valid based on the metamodel. Since the metamodel reflects the refinement relationship between two class diagrams, any invalid instance denotes some discrepancies between the two class diagrams.

The support of the metamodel mechanism is based on our existing tool [9], which supports model checking between a metamodel and its instance model. Our tool takes two inputs: a metamodel and its instance model. The output of the tool is whether the instance model is valid based on the metamodel. As an application of the metamodel mechanism, we will present a general set of refinement rules by extending the UML metamodel using a class diagram as well as OCL constraints. Once our existing tool reads the extended UML metamodel and two class diagrams, it returns to users whether the two class diagrams are valid based on the metamodel which supports class refinement.

On the other hand, because of our existing tool which supports the metamodel mechanism, developers can revise our set of refinement rules by presenting their class diagram with OCL constraints. For example, if users find the previous set of rules are not applicable in the later phase of software development. They can redesign a new metamodel which is based on the previous metamodel. After that, they can check whether two new class diagrams satisfy the new metamodel or not. Therefore, the first challenging issue can be easily solved thanks to our existing tool supporting the metamodel and model checking mechanism.

The second challenging issue can be simplified when all elements in the M1 model can be regarded as an instance element of an element in the M2 model (metamodel). Therefore all paths in the M1 model are regarded as instances of the meta-element in the M2 model. Consequently, there is no need to have any path search in the M1-layer model. In this paper, an example will be illustrated to show how to find some semantic discrepancy based on different sets of refinement rules during software refinement.

Since the positive or negative property of a semantic discrepancy heavily depends on the development context, the main objective of this paper is to propose a tool which can report to software developers any semantic discrepancies between two class diagrams during the refinement. However, either a positive or a negative property of a semantic discrepancy is left to the software developers to decide.

The metamodel mechanism has been used in many applications. One of the most important applications is proposed in paper [4] led by France. Paper [4] applies the metamodel mechanism to specify metamodels which characterize UML design models of pattern solutions. In this paper, we will propose to apply the metamodel mechanism to class refinement.

The paper is organized as follows. Section 2 shows how to apply the metamodel mechanism to class refinement. Section 3 introduces our existing tool and shows how a new tool based on the refinement rules is generated. At the same time, a web-based online learning system is shown throughout Section 2 and 3 to illustrate the application of our work. Section 4 compares our approach with some similar work.
Finally we draw some conclusions in section 5.

2. Refinement Rules for Class Refinement

In this section, we will first introduce the stereotypes used in the class refinement. Therefore, we will introduce a set of basic refinement rules using stereotypes in subsection 2.2. As an advantage of the metamodel which is applied to class refinement, we will discuss some variations of refinement rules in subsection 2.3.

2.1 UML Extension Mechanism: Stereotypes

In this research, we allow software developers to check the semantic relationship between one model element $A$ and its refined model elements which belong to two different class diagrams respectively when the model element $A$ is refined. Usually software developers can use a table explicitly representing a refinement relation for classes and relationships. For example, if a relationship $A$ in model $M_1$ is refined to relationships $A_1, A_2, \ldots, A_n$ in model $M_2$, then a table shown in Table 1 consisting of two columns can have the value of $A$ and $A_1, A_2, \ldots, A_n$ in the two columns respectively, indicating a refinement relation. Also, the table can be used to represent the class refinement relation in which software developers are interested.

<table>
<thead>
<tr>
<th>Refined Element</th>
<th>Refining Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$A_1, A_2, \ldots, A_n$</td>
</tr>
</tbody>
</table>

Table 1. Using a table to represent a refinement relation.

But a table, as a way used to attach additional refinement information to UML diagrams, places some burdens on software developers. It is also hard to check a table syntax based on the UML metamodel mechanism unless some rules about a table structure are proposed. In view of the disadvantages of a table, we use stereotypes to represent the model elements whose refinement relation software developers are interested in.

While UML is a very comprehensive modeling language applicable to a variety of software systems, it is impossible for software developers to apply UML to all different domain-based software systems. So UML provides a mechanism to extend the UML notation. Among the techniques used in the extension mechanism, stereotypes are the most powerful way to extend the UML metamodel.

A stereotype can be defined by either using the predefined dependency stereotype to show a relation between a newly defined stereotype and its base class or listing a table where a stereotype, its base class and some other operational constraints are given. We extend the UML metamodel by introducing stereotypes, i.e. new model metalevels, which extend the existing meta-element. Based on the refinement rules we can give an OCL constraint for the new stereotype. Therefore, any $M_1$-layer model can be regarded as an instance of the extended UML metamodel which includes the new stereotypes.

In this paper, we choose the former one, which is compatible with the UML metamodel. A stereotype is a user-defined metaclass which is a child class of a base class. Besides all features inherited from its base class, a stereotype can define tags. An instance of a stereotype should have a value for each tag defined in the stereotype. It is the tag value that we can represent a refinement relation between two models. Software developers only use stereotypes attached to elements in class diagrams, showing that these elements’ refinement relation should be studied. Then our tool can check whether any semantic discrepancies among these stereotyped elements exist or not. Besides a tag, a stereotype can define constraints which must be satisfied by its instance. If a constraint is not satisfied, then a semantic discrepancy is found and reported.

2.2 Rules for Refinement

In a class diagram, there are two kinds of elements, i.e. classes and relationships. A relationship mainly includes generalization, association and dependency. An association can be further divided into a unidirectional association, bidirectional association, aggregation, or composition. A class is an abstraction that specifies the attributes and behaviors of a set of objects. A class will become a meaningless entity when it has no relationship with other classes. It is the relationships that connect all classes to achieve the requirements of a problem. So the relationship refinement plays an important role during class diagram refinement. It is the relationship refinement that results in changes of classes, for example new classes can be introduced or a class is refined to a set of new classes. Hence, studying the relationship refinement and then considering a relation between the related classes caused by the relationship refinement are the key ideas in this research.

To study the refinement relation for a relationship and its related classes during the class diagram refinement, we use a stereotype to represent a relationship or a class whose refinement relationship is what software developers are interested in. A stereotype is a user-defined meta-element which is used to extend an existing UML metalelement, called a base class. In this paper, we use a graphical notation, i.e. using a special dependency called «stereotype» to show a stereotype and its base class.

In order to show the refinement relation between two class diagrams, we extend the UML metamodel by intro-
producing seven stereotypes, which are Refined_Assoc, Refining_Assoc, Refined_Gen, Refining_Gen, Refined_Dep, Refining_Dep and Refine_Class. A stereotype whose name starts with Refined_ represents an element to be refined, i.e. its instance belongs to a high-level model. A stereotype whose name starts with Refining_ represents an element which is used to refine an element in its high-level model. Another stereotype we define is Refine_Class. This stereotype is used to represent the classes related to a relation during the class diagram refinement. We define a tag mapping_name which is used to represent a mapping relation between the two classes during the class diagram refinement. If the two classes have the same value for mapping_name, then these two classes have some refinement relation during the class diagram refinement.

The refinement relation in two class diagrams can be represented by a mapping between a single relationship in a higher-level class diagram and more than one relationship in a lower-level class diagram, i.e. a one-to-many mapping. Figure 1 shows the UML extended metamodel which includes stereotypes and the relationships we need to represent the class refinement.

In the following we use a web-based online learning system [11] to illustrate our refinement rules. The web-based online learning system is used to provide a learning environment without time and location constraints. A student first registers for the system. After then, the student can learn the course material through the internet, i.e. obtaining the course materials such as lecture notes and homework assignments, inquiring about grades etc. through the web site. These course materials must be designed by an instructor. Based on the above description, we can design a higher-level class diagram shown in Figure 2(a).

As the development continues, more details related to the program domain are considered. The course materials are refined to two sets of classes. One set consists of classes LectureNote, Homeworksheet and ExamPaper. These classes are accessed by the registered students. The other set consists of class Lesson, Homework and Exam, which are controlled by a class Instructor. The further relationship among these classes can be found in Figure 2(b). In addition, we introduce class Person which is a parent class of Student and Instructor. Class Instructor is a child class of Administrator because an instructor can not only change some login records, which can also be changed by an administrator, but also control classes related to course materials while an administrator cannot. The whole design model is shown in Figure 2(b).

In the following we will consider refinement rules for generalization, bidirectional association, and unidirectional association. Since the tool tries to find any semantic discrepancy and leaves software developers to decide whether it is positive or negative, our rules are based on the semantic consistency during the relationship refinement. According to [2], a new user-defined stereotype consists of the name, description, tagged values and constraints besides the graphical notation and we will follow this to introduce each stereotype.

1. Bidirectional Association: At level one, we assume that there is an association between class A and B and this association is tagged as Refined_Assoc in Figure 3(a). The association between class A and B can be refined to a set of associations, tagged as Refining_Assoc in Figure 3(b), with a set of helper classes. Using different values for the tag mapping_name of classes, developers can represent the refinement relation between the related classes. Figure 3(b) represents that class A is mapped to X and class B to Y because class A and X have the same value for mapping_name and so do class B and Y. The models shown in Figure 3(I) belong to the M1-layer while Figure 3(II) shows the M2-layer model elements taken from the metamodel shown in Figure 1. We use the dotted line with an arrow to represent instance-of relation between an element in M1-layer and its counterpart in M2-layer. Using the instance-of relation, we can check whether an M1-layer model satisfies all constraints given in the M2-layer model.

As an application of the rule shown in Figure 3, in the web-based online learning system example, an association between class Instructor and Course_Materials can be refined to two associations, i.e. an association between class Instructor and class CourseEditor and an association between class CourseEditor and class Lesson, Homework or Exam, which are the refined classes of Course_Materials.

Based on the above description, we define three stereotypes: Refined_Assoc, Refining_Assoc and Refine_Class.

Description: The stereotype Refined_Assoc is used to represent an association defined in a higher-level class diagram which will be refined in a lower-level class diagram.

Constraint: According to Figure 3, the stereotype Refined_Assoc should satisfy restrictions between itself and its corresponding refined associations.

- The refined association should be refined to a set (chain) of associations, and
- The two end classes on the refined association correspond to the two classes at both ends of the chain of the refining associations, and
Figure 1. A part of the extended UML metamodel related to the class refinement.

Figure 2. Two UML class diagrams for the web-based online learning system.
• Two consecutive bidirectional associations should have the same class as their end class.

Then the OCL constraint for Stereotype Refined_Assoc is as follows:

\[
\text{self.associationEnd} \rightarrow \exists (p,q \mid p \leftrightarrow q \text{ and self.associationEnd} \rightarrow \exists (x,y \mid x \leftrightarrow y \text{ and self.associationEnd} \rightarrow \exists (a \mid a.\text{classifier}.\text{mapping_name} = p.\text{classifier}.\text{mapping_name} \text{ and y.associationEnd} \rightarrow \exists (b \mid b.\text{classifier}.\text{mapping_name} = q.\text{classifier}.\text{mapping_name} \text{ and (self.\text{mapping_assoc} \rightarrow \text{size()} = 2 \text{ and x.associationEnd} \rightarrow \exists (h1 \mid y.\text{associationEnd} \rightarrow \exists (h2 \mid h2.\text{classifier} = h1.\text{classifier})})) \text{ or (self.\text{mapping_assoc} \rightarrow \text{size()} > 2 \text{ and self.\text{mapping_assoc} \rightarrow \text{forAll}(z1 \mid (z1 \leftrightarrow x \text{ and z1} \leftrightarrow y) \text{ implies (z1.associationEnd} \rightarrow \text{forAll}(v \mid \exists (z2 \mid z2 \leftrightarrow z1 \text{ and z2.associationEnd} \rightarrow \exists (u \mid u.\text{classifier} = v.\text{classifier})))))))}
\]

As an example shown in Figure 2, it is not correct for the association learn shown in Figure 2 to be refined to classes Student, Person, WebServer and Registration and the relations between them, shown in Figure 2. The reason is that the refinement relation makes the above OCL constraint to be false.

**Description:** The stereotype Refining_Assoc is used to denote an association, defined in a lower-level class diagram, which is used to refine an association in its corresponding higher-level class diagram.

**Constraint:** Since the constraint is related to the stereotype Refined_Assoc and already given in Refined_Assoc, there is no constraint for the stereotype Refining_Assoc.

2. Unidirectional Association: At level one, we assume that there is a unidirectional association from class A to class B, which is tagged as Refined_Assoc shown in Figure 4(a). This unidirectional association is refined to a set of unidirectional associations, each of which is tagged as Refining_Assoc in Figure 4(b). Similarly, we can draw the instance-of relations between these classes and their metamodel elements but due to space we skip them here.

In the following we explain the stereotypes related to the unidirectional association refinement.

**Description:** The stereotype Refined_Assoc is the same as the stereotype Refined_Assoc defined for the bidirectional association.

**Constraint:** Besides the constraint given in the bidirectional association, we should add the restriction on navigable property for association ends, which is shown in the following.

- The refined association should be refined to a set (chain) of unidirectional associations, and
- The class at the navigable end of the refined association should correspond to the class at the navigable end of one end of the chain of the refining associations, and
The class at the non-navigable end of the refined association should correspond to the class at the non-navigable end of the other end of the chain of the refining associations, and

- Two consecutive unidirectional associations should have the same class as their end class and this class is at the navigable end of one association and the non-navigable end of the other association.

Then the OCL for Stereotype Refined_Assoc is as follows:

```
self.associationEnd->exists(p,q | p<>q and self.mapping_assoc->forall(x | x.associationEnd->exists (a,b | self.mapping_assoc->forall(y | y.associationEnd->forall(u | (u<>a and u<>b) implies (u.classifier <>a.classifier and u.classifier<>b.classifier) and (p.classifier.mapping_name=a.classifier.mapping_name and q.classifier.mapping_name = b.classifier. mapping_name and p.isNavigable=false and a.isNavigable=false and q.isNavigable=true and b.isNavigable=true) and (y.associationEnd->exists (v | u<>v and u.classifier=v.classifier and u.isNavigable<>v.isNavigable)))))
```

**Description:** The stereotype Refining_Assoc is the same as the stereotype Refining_Assoc defined in the bidirectional association.

**Constraint:** No constraint for the stereotype Refining_Assoc.

Due to space, the other rules related to the association refinement are skipped here and interested readers are referred to [6].

3. Generalization: At level one, we assume that class A is a subclass of class B. Then at level two which is a refined level of level one, the generalization can be refined into two generalizations with helper classes. But the relationship between the class A and B at level one should be kept. Figure 5 shows the rule for generalization abstraction.

In our web-based online learning system example, the generalization between class Person and Instructor can be refined to two sub-generalizations which are a generalization between Person and Administrator, and a generalization between Administrator and Instructor. The two generalizations are needed because part of the work done by class Instructor can be moved up to a helper class Administrator. Figure 2 shows the above generalization refinement using Rational Rose.

**Description:** The stereotype Refined_Gen is used to represent a generalization, defined in a higher-level class diagram, which will be refined in the next level class diagram.

**Constraint:** The stereotype Refined_Gen should satisfy the restrictions between itself and its corresponding refining generalizations.

- The refined generalization should be refined to a set of ordered generalizations, and
- The child of the refined generalization should correspond to the child of the first refining generalization, and
- The parent of the refined generalization should correspond to the parent of the last refining generalization, and
- Two consecutive generalizations should have the same class as their end class, and this class is the parent class of one generalization and the child class of the other generalization.

Then the OCL constraint for Stereotype Refined_Gen is as follows:

1 Due to space, we combine the two class diagrams to one class diagram.
self.child.mapping_name = self.mapping_gen→first.
child.mapping_name and self.parent.mapping_name = self.mapping_gen→last.parent.mapping_name and
Sequence1..(self.mapping_gen→size( ) - 1) →
forall(i| self.mapping_gen[i].parent =
self.mapping_gen[i + 1].child)

Additional Operation: Because the order of the mapping generalizations derived from a refined generalization is important, the extended UML metamodel gives an operation to compute the order for each mapping generalization as follows:

Operation for Refined_Gen:
self.mapping_gen[i] =
if (i=0) then
  self.mapping_gen→exists(c| c.child.
  mapping_name=self.child.mapping_name)
else if(i=self.mapping_gen→size()) then
  self.mapping_gen→exists(c| c.parent.
  mapping_name = self.parent.mapping_name)
else
  self.mapping_gen→exists(c| c.child =
  self.mapping_gen[i-1].child)

Description: The stereotype Refining_Gen is used to denote a generalization defined in a lower-level class diagram which is used to refine a generalization in its higher-level class diagram.

Constraint: Since the constraint is related to the stereotype Refined_Gen and already given in Refined_Gen, there is no constraint for the stereotype Refining_Gen.

2.3 Advantage of the Metamodel Methodology

In the previous section, we discussed the most conservative refinement rules, i.e. keeping the transitive property for each relationship. In reality, developers usually can apply some “aggressive” strategy to software development. For example, Figure 6 shows that an association in a higher-level class diagram can be refined to a set of generalizations followed by a set of associations in the lower-level class diagram. While this rule is not applicable to all cases during software development, its application can avoid many positive discrepancies due to the transition from less information to more information during the refinement.

One important advantage of the metamodel methodology is that developers can easily design their metamodel to reflect the refinement rules they want. Furthermore, a refinement rule can be represented by different metamodels. But the metamodel designed by developers should reflect the refinement rule, i.e. if an instance model (two class diagrams) does not satisfy the metamodel, then their developers should realize this discrepancy in it. Figure 7 is the metamodel which represents the refinement rule shown in Figure 6.

In Figure 7, we introduce new tags refinement_value and mapping_order in the stereotype Refining_Assoc and Refining_Gen and refinement_value in stereotype Refined_Assoc and Refined_Gen. Using tag mapping_order, we can generate a sequence of relations which are used to refine an association in a higher-level class diagram, shown in Figure 6. Tag refinement_value is used to represent the refinement relation between two different layer model elements. The same value represents the same refinement relation. For example, all the generalizations and associations whose refinement_value is v1 are used to refine the association whose refinement_value is v1 between classes A and B in Figure 6.

The constraint for this refinement rule is as follows:

- The refined association should be refined to a sequence of generalizations followed by a sequence of associations, and
- The two end classes on the refined association correspond to the two classes at both ends of the sequence of the refining associations, and

Figure 5. A rule for generalization abstraction.
Figure 6. A rule for the revised bidirectional association refinement.

Figure 7. A part of the extended UML metamodel related to the class refinement.
The two end classes on a refining relation, i.e., either a generalization or association, should connect to the same class, and

- The parent class of the last generalization in the sequence should connect to the class which also serves as an end class for the first association in the sequence.

Similarly, we can attach an OCL constraint which is translated from the above constraint for the stereotype Reﬁned_Assoc to the revised metamodel shown in Figure 7. As an application of this rule, the association learn shown in Figure 2 is correctly reﬁned to classes Student, Person, WebServer and Registration and the relations between them, shown in Figure 2, because the reﬁnement satisﬁes the above constraints.

3 A Tool Supporting the Metadata Architecture

The four layer metadata architecture is the classical framework for metamodeling. Model $M_1$ can be regarded as an instance of Model $M_{i+1}$. Any metamodel can affect a model designed at layer $M_i$. Usually the metamodel at layer $M_2$ consists of constructs, such as Classes, Association etc., which are all deﬁned in a class diagram. On the other hand, these constructs from a class diagram cannot be enough to represent all needs required by a metamodel. The Object Constraint Language (OCL) is used to represent those restrictions missed by a class diagram. So, a tool supporting a class diagram and OCL can also be used to support a metamodel at layer $M_2$. Therefore the tool can ﬁnd any inconsistency in a model at layer $M_1$ based on the metamodel.

The tool we developed consists of four modules as shown in Figure 8. This tool is based on the Abstract State Machine Language (AsmL) [7], developed by Microsoft. It is an executable speciﬁcation language based on the theory of Abstract State Machines [5]. We use the AsmL as our runtime environment.

The ﬁrst module is the Specification Model Parser, which reads a class diagram given in XML Metadata Interchange (XMI) [8] format. The XMI is proposed by the Object Management Organization to interchange a UML Model between the different UML Computer-Aided Software Engineering (CASE) tool vendors. Therefore, the advantage of using XMI to represent a class diagram in this tool is that the tool can accept any UML CASE tool as long as it can produce ﬁles in the XMI format.

The Specification Model Parser will translate the class diagram into its AsmL representation. The OCL constraints given in a class diagram are extracted by the module Extract OCL. After that, these are fed into the second module OCL Parser. The OCL parser translates these constraints into AsmL code that uses an OCL library module AsmL Library for OCL written in AsmL. The last module is Instance Model Parser that translates an instance of a speciﬁcation model into AsmL code that creates the objects, initializes their attributes, and calls the veriﬁcation methods. The single AsmL source ﬁle containing all the translated code can be compiled and executed to perform the model checking on the instance diagram.

The tool performs several checks on the instance diagram before translating it into AsmL. In particular, the tool ensures that each object in the instance diagram is an instance of an existing class in the speciﬁcation diagram. It also ensures that all attributes of the class have corresponding slots in the object and vice versa. Graphical constraints in the speciﬁcation diagram, such as multiplicity constraints, are automatically checked by the tool as well.

Depending on a metamodel fed to our tool, developers can check whether a model is a valid instance or not. In this paper, we feed the UML metamodel which describes the class diagram reﬁnement. Therefore software developers can input two class diagrams designed at two different phases during software development and then check whether these two class diagrams satisfy the metamodel, which means that the two class diagrams are treated as an instance model. As mentioned before, we design the metamodel which serves to support the class diagram reﬁnement. So any error reported by this tool indicates some discrepancy between the two diagrams. Software developers will decide whether the discrepancy is positive or negative.

4 Discussion and Comparison

In this paper we discuss a metamodel to support class diagram reﬁnement during software development. We do not consider the multiplicity related to an association and its reﬁned associations. Actually, to support multiplicity reﬁnement, we can easily revise a metamodel according to [1]. After the metamodel is fed to our metamodel tool, software developers can check whether two class diagrams are a valid instance or not.

There are some research works related to class diagram transformation. Egyed in [3] proposed a methodology to abstract a class diagram to a high-level class diagram, which is called "reverse engineering", and a tool based on this methodology exists. Contrary to abstracting a class diagram from a low-level to a high-level, we proposed a metamodel which can support the class diagram reﬁnement during software development. Refining a class diagram is a necessary step during software development because forward engineering is the process of transforming a requirement model into code through a series of intermediate models.

The main difference between Egyed’s work [3] and ours
is that relationship refinement is the main focus in our work while class abstraction is the focus in Egyed’s work. We think forward engineering is a process to refine a relationship between objects in the real world. Refining a relationship can result in more classes. Each class is an abstraction of objects in the real world. However, reverse engineering is a process of transforming code into models. The code is written in some object-oriented languages such as C++. The class structure in code is obvious while a relationship between classes is not obvious. Therefore, the rules proposed in [3] are based on class abstraction while our rules proposed in the metamodel are based on relationship refinement.

Ambiguities can be caused by the serial abstraction in Egyed’s work due to the multiple paths between two classes. For example in [3], class Hotel and Guest are connected by two helper classes, i.e. Reservation and Room, shown in Figure 9(a). After abstraction, the two associations are left to connect Hotel and Guest shown in Figure 9(b). These two abstractions cannot be abstracted any more and they are declared as the “AND/OR” notation. But in forward engineering, this kind of ambiguity does not exist. At the highest level, there is only one association between Hotel and Guest, shown in Figure 9(c). As the development continues, this association can be refined to two associations, with a helper class added to each of them shown in Figure 9(a) via step Figure 9(b). All the relationships during software development are unambiguous.

Using stereotypes can help developers pinpoint the discrepancies during software refinement. For example, by mistake, developers refine association Reserves to the link connecting Hotel, Room and Guest and refine Lives to the link connecting Hotel, Reservation and Guest. When using the tool presented in [3] to abstract classes from (a) to (b) in Figure 9, developers cannot find the error during development. But using stereotypes tagged to the corresponding relationships and classes, our tool can pinpoint the above discrepancies immediately during the software refinement.

Transformation of class diagrams has been proposed by Whittle in paper [10] where he proposed a set of transformation rules. Based on these rules, software developers can check whether two class diagrams are refactorings of each other. Although the purposes of his work and ours are different, we have the similar goal, i.e. keeping the semantic consistency between two class diagrams. Contrary to using the textual format to represent rules as well as the related constructs such as class and association, we extend the UML metamodel which presents the class diagram refinement rules. So, software developers can check whether one class diagram refines another at the same time as the time they check whether these two class diagrams are a valid instance of the UML metamodel.

On the other hand, if two class diagrams are a valid instance based on a UML metamodel, then an object diagram reflecting the instance property can be generated. Any element in these two class diagrams can be regarded as an instance of a meta-element in the metamodel. Therefore, if developers provide the first class diagram, then using an object diagram we can produce the second possible class diagram based on the metamodel. This can implement an automatic class diagram transformation. All of these work can be done within our tool.

5 Conclusion

Our contribution is not only to propose the UML metamodel using stereotypes to represent class diagram refinement but also give software developers a mechanism to design their metamodel and then check whether their instance model is valid or not. Using the UML metamodel mechanism, it is possible for UML tool developers to find some discrepancies between the two class diagrams at different
levels. Depending on the different phase of software development, software developers can adjust the metamodel to reflect the different consistency rules in their class diagram design.

For future work, we can propose a UML profile which can support a software development process. Using the existing tool supporting the metadata architecture, we can generate another tool which can help software developers make sure that the model developed satisfies the development process. More real applications will be applied to our system in the future.

References


