A Model Transformation Approach for Design Pattern Evolutions

Jing Dong, Sheng Yang, Kang Zhang
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083, USA
{jdong,syang,kzhang}@utdallas.edu

Abstract

The evolution of a design pattern typically involves the addition or removal of a group of modeling elements, such as classes, attributes, operations, and relationships. However, the possible evolutions of each design pattern are often not explicitly documented. Missing part of the evolution process may result in inconsistent evolution. In this paper, we define the evolution processes of design patterns in terms of two-level transformations, thus making the possible evolutions of each design pattern explicit. In addition, we automate the evolution processes as XSLT transformations that can transform the UML model of a design pattern application into the evolved UML model of the pattern. Both the original and evolved UML models are represented in the XML Metadata Interchange (XMI) format to facilitate the transformations. Furthermore, we check the consistency of the evolution results using the Java Theorem Prover.

KEYWORDS
Design pattern, Model Transformation, XMI, XSLT, JTP, Design pattern evolution

1. Introduction

Computer-based systems are normally needed to be amenable to changes due to constant changes of user requirements, platforms, technologies and environments. Change is a constant theme of computer-based design and development. It can be a disaster if a single change may cause a large number of changes in the systems. It is important to localize the changes such that minimum efforts are needed. This requires the initial designers of a computer-based system to be aware of potential changes. Thus, the resulting systems are flexible and agile to future evolution.

One of the important goals of design patterns [9] is design for change. Design patterns capture expert design experience by partitioning software designs into a stable part and changeable part. By separating and encapsulating both parts, the change impact of a software design can be minimized. Thus, most of the design patterns encapsulate future changes that may only affect limited part of a design pattern. This evolution process can be achieved by adding or removing design elements in existing design patterns. In the document of each design pattern, however, the evolution information is generally not explicitly specified. When changes are needed, a designer has to read between the lines of the document of a design pattern to figure out the correct ways of changing the design. More importantly, the evolution process of a design pattern may involve the addition or removal of several parts of a design pattern. Misunderstanding of a design pattern may result in missing parts of the evolution process. The addition and removal of system parts should not violate the constraints and properties of design patterns. Thus, it is important to have, in the documentation of the design pattern, information about the evolution of the patterns. The evolution of a software system at the design level is less costly than it is at the implementation level.

To raise the level of abstraction, the Model Driven Architecture (MDA) [17] supports software development based on models as primary artifacts. Thus, the level of reuse is raised accordingly since high-level software models can be reused as well as software programs (libraries). In this way, models become assets in MDA. Consequently, technology that supports the transformation of models is considered as a key enabler of MDA. Design patterns are usually modeled in the Unified Modeling Language (UML) [2] which is considered to be the *de facto* standard for object-oriented modeling. The evolution of a pattern may be considered as a transformation of the design model.

The XML Metadata Interchange (XMI) [27] is an interchange format for metadata in terms of the Meta Object Facility (MOF) [17]. XMI specifies how UML models are mapped into a XML file. By representing a UML model in the XML format, the UML model can be manipulated since there are rich collections of XML related techniques and tools available. The extensible stylesheet language transformation (XSLT) [28] provides the transformation from a XML document to other types of documents (including XML). The use of

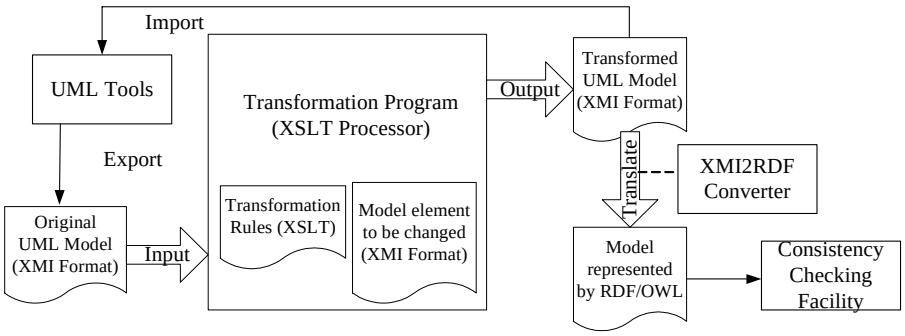

Figure 1 The Overall Architecture of the Approach

XMI and XSLT helps to automate model transformation process and enforces constraints of model implicitly.

To capture the evolution processes of design patterns, we define two-level evolutions: the primitive level and the pattern level. The primitive-level evolutions are the addition or removal of modeling elements, such as classes and relationships. The pattern-level evolutions characterize the recurring evolutions of each design pattern based on the primitive-level evolutions. Meanwhile, the evolution processes are implemented based on XMI format to transform the UML models of design pattern applications. Thus, the structure of a design pattern may be evolved in some prescribed ways based on the pattern-level transformation. The evolution processes of design patterns are specified as XSLT transformation rules. We also provide tool support to automate such transformations. In order to assure the essential properties of a design pattern still hold after the evolutions, we use the Java Theorem Prover (JTP) [8][31] to check the consistency of the designs after the evolution processes.

The remainder of this paper is organized as follows: the next section describes an overview of our approach. Section 3 defines the primitive-level and pattern-level evolutions. Section 4 presents the design pattern evolutions as model transformations based on XMI and XSLT transformation rules. Section 5 discusses the consistency checking using JTP. The last two sections cover related work and conclusions.

2. Overview of the approach

Figure 1 depicts the overall architecture of our approach for automated evolutions of design patterns based on model transformation and thereafter checking the system consistency.

A software design with the applications of design patterns is normally modeled using the UML and drawn using *UML tools*, such as IBM Rational Rose [29] or ArgoUML [17]. Since UML diagrams are typically saved in proprietary formats of the corresponding UML tools, a standard XMI format of the UML diagrams has been defined and the plug-in of these UML tools has been developed to export UML diagrams into the XMI format. For example, the plug-in of IBM Rational Rose is called UniSys which can translate UML diagrams into XMI format.

In order to automatically evolve design patterns, we define two-level evolutions (primitive level and pattern level) as discussed in the next section. We also specify these two-level evolutions as *transformation rules in XSLT*. These transformation rules define which modeling elements are changed (added or removed) and where and how these elements are changed.

The *transformation program* is an *XSLT processor* which can automatically transform the UML model of a design pattern based on the transformation rules. The input of the XSLT processor is the *original UML model* which has been converted into XMI format. It records all the model elements, such as classes, attributes, operations, and their relationships in the original UML model of a design pattern. The XSLT processor has built-in transformation rules which are written in XSLT.

The *transformation rules* specify how the model elements are changed (added or removed) from the original system design. The XSLT processor takes the input and performs the following two actions. First, all *model elements to be added into or removed from* the UML model are generated. Second, the generated model elements are added into or removed from the original system design at the required positions according to the built in XSLT transformation rules. The output of the XSLT processor is the *transformed UML model* of a design pattern in XMI format, which can be imported and transformed into a UML class diagram that can be displayed by, e.g., IBM Rational Rose.

Table 1 The Primitive-Level Evolutions

Model Elements	Parameter List	Descriptions
Class	className	Add or remove a class with name "className" into a pattern
Attribute	attributeName, className, type, accessibility	Add or remove an attribute with name "attributeName", type of "type", accessibility of "accessibility" into the class "className"
Operation	operationName, className, returnType, accessibility, para1, paraType1…	Add or remove an operation with name "operationName", type of "type", accessibility of "accessibility", and arguments list para1 with type "paraType1" into the class "className"
Association	className1, className2	Add or remove an association between classes "className1" and "className2" into a pattern
Generalization	child, parent	Add or remove a generalization relationship into a pattern, with subclass "child" and superclass "parent"
Aggregation	part, whole	Add or remove an aggregation relationship into a pattern, "part" class is a part of "whole" class
Composition	part, whole	Add or remove a composition relationship into a pattern, "part" class is a part of "whole" class
Realization	fromName, toName	Add or remove a realization relationship from class "fromName" to class "toName" into a pattern
Dependency	fromName, toName	Add or remove a dependency relationship from class "fromName" to class "toName" into a pattern

When a software system is evolved and changed, the integrity and consistency of the system should be maintained. Some properties may not hold anymore after the changes and evolutions. Thus, we need to check that the evolved system has the same properties as the original one. In order to check the consistency of the evolved design, as shown in Figure 1, the evolved design is first converted into *RDF/OWL* [32] format from the XMI format. Therefore, the *consistency checking facility*, such as JTP [31], can be used to perform the system consistency checking.

3. Two-level design pattern evolutions

A design pattern normally encapsulates some particular ways for future changes. After a design pattern is applied in a software application, the designer may change the application design in the particular ways directed by the design pattern. The evolution information of each design pattern allows the designers to change the system design with minimum impact of other parts of the system. For example, Figure 2 shows the class diagram of the Mediator pattern [9]. When the Mediator pattern is applied initially, there may be only two concrete colleague classes. A change may request an additional concrete colleague class later as shown in Figure 3. However, such evolution information of each design pattern is normally implicit to the description of the pattern. A designer has to search the document of the pattern to find the guidance on evolution. Misunderstanding and mistakes of changing the design patterns may compromise the benefits of using these patterns and have huge impact on the system designs.

In this section, we describe a classification of design pattern evolutions in terms of two-level transformations: the primitive-level evolution and the pattern-level evolution.

3.1 Primitive-Level Evolutions

The primitive-level evolution describes the basic transformations that can be performed during the evolution process of a design pattern. These basic transformations include the addition or removal of a modeling element, such as class, operation, attribute, association, generalization, aggregation, composition, realization, and dependency. These basic transformations become the building blocks of the pattern-level evolution.

We identify nine modeling elements that can be added or deleted as the basic transformations in the pattern evolution process. The general format of adding a modeling element is Add (ME (PL)). The model elements (ME) and the parameter list (PL) are shown in Table 1. For example, adding a class named "Leaf" can be specified: Add (Class (Leaf)). Similarly, the removal of a modeling element can be specified: Delete (ME (PL)). The replacement of a model element with another is conducted by first removing the modeling element and then adding a new modeling element. It can be defined: Delete (ME1 (PL1)) + Add (ME2 (PL2)).

3.2 Pattern-Level Evolutions

The pattern-level evolution characterizes the recurring evolution processes which occur in many design patterns. It is described in terms of a sequence of

the basic primitive-level evolutions. Each design pattern may perform some of the pattern-level evolutions, which can be added in the document of the pattern. Thus, the designer may choose a potential pattern-level evolution and apply the corresponding transformations when changes are required.

In this section, we characterize five pattern-level evolutions that are recurring in different design patterns. Note that we do not claim this to be a complete list of all possible pattern-level evolutions. Nevertheless, new pattern-level evolutions can be easily added into the list specified by the primitive-level evolutions.

The first pattern-level evolution is called *independent* change which is a simple addition or removal of one independent class and the corresponding relationships between this class and the classes in the original pattern. This class is independent in the sense that the addition or removal of the class does not cause any effects on the existing classes of the design. This kind of pattern-level evolution can be expressed in the primitive level evolutions as follows[1]:

```
Add ( Class (className)) +
Add ( Relationship (className, existingClassName))
```

where className is the name of the class which is added into the pattern. Relationship includes association, generalization, aggregation, composition, realization, and dependency. The existingClassName is the name of the class from the original pattern. There may be multiple relationships added into the pattern with the addition of a class.

This kind of evolution appears in several design patterns as, for example, in the Mediator and Facade patterns. Figure 2 is the class diagram of the Mediator pattern describing a possible application containing two ConcreteColleague classes. A potential evolution of this pattern application is to add a new ConcreteColleague class, which can be defined as the following transformations based on the primitive-level evolutions:

```
Add ( Class (ConcreteColleague)) +
Add ( Generalization (ConcreteColleague, Colleague)) +
Add ( Dependency (ConcreteMediator, ConcreteColleague))
```

where a new ConcreteColleague class is added with two new relationships: generalization and dependency. The generalization relationship is with the Colleague class. The dependency relationship is on the ConcreteMediator class. The result of this evolution is shown in Figure 3.

[1] Since the addition and removal have the same format and the only difference is the transformation names (Add and Delete), we omit the evolutions of removing modeling elements. We also omit the specifications of the next four kinds of evolutions. We refer to [7] for interested readers.

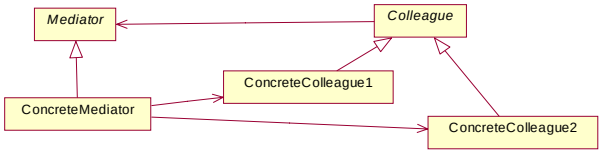

Figure 2 Mediator Pattern with Two Concrete Colleagues

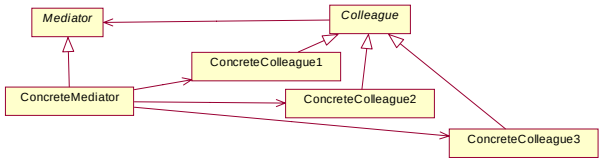

Figure 3 Mediator Pattern with Three Concrete Colleagues

The second pattern-level evolution is called *packaged* change which is the addition or removal of one independent class and the corresponding relationships between this class and the classes in the original pattern. In addition, certain attributes and/or operations of this class are added and removed accordingly. For example, Figure 4 is the class diagram of the Observer pattern describing a possible application containing one ConcreteSubject and two ConcreteObserver classes. A potential packaged evolution of this pattern application is to add a new ConcreteObserver class (ConcreteObserver3) with its attributes (s1 and s2) as shown in Figure 5.

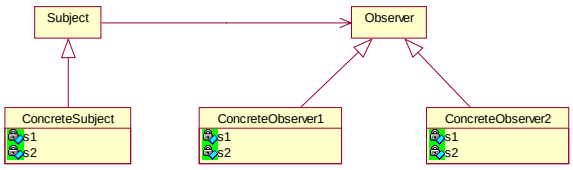

Figure 4 Observer Pattern with Two Concrete Observers

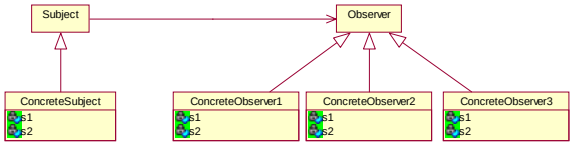

Figure 5 Observer Pattern with Three Concrete Observers

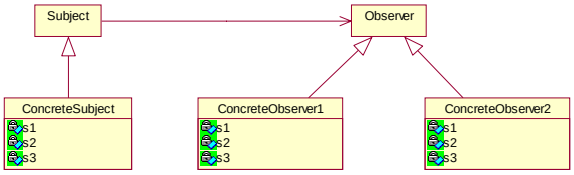

Figure 6 Observer Pattern with Three Attributes

The third kind of pattern-level evolution is called *class group* change which is the addition or removal of one attribute/operation in several different classes consistently. In this case, a certain set of classes, instead of a single class, are affected by the addition or removal of the attribute or operation. One potential class group evolution of the Observer example in Figure 4 is to add one attribute called s3 as a new data to be observed by

the observers. Thus, this attribute needs to be added in all ConcreteSubject and ConcreteObserver classes, which is shown in Figure 6.

The fourth kind of pattern-level evolution is called *correlated classes* change which is the addition or removal of a group of correlated classes. When certain classes are added or removed, some other classes have to be added or removed accordingly. These correspondence relations are important since missing transformations may cause inconsistency. In addition, the corresponding relationships between this group of classes and other classes are added or removed. The attributes and operations of the classes of this group are also added or removed. The addition or removal of this group of classes may not affect the internal of other classes in the original design pattern applications. For example, Figure 7 shows an application of the Abstract Factory pattern with two kinds of products (AbstractProductA and AbstractProductB). Each kind of products has two concrete products: ProductA1/ProductB1 and ProductA2/ProductB2, respectively. Thus, there are two concrete factories: ConcreteFactory1 and ConcreteFactory2. A potential correlated classes evolution can be the addition of a new kind of concrete products (ProductA3 and ProductB3). This requires the addition of a new concrete factory (ConcreteFactory3) to create the corresponding newly added concrete products. This new concrete factory class also has the same operations (createProductA and createProductB) as the other two concrete factory classes as shown in Figure 8.

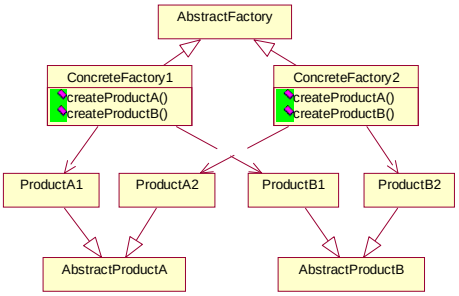

Figure 7 Abstract Factory Pattern with Two Kinds of Products Created by Two Concrete Factories

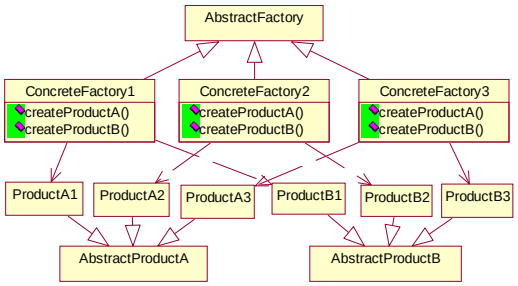

Figure 8 Abstract Factory Pattern with Three Kinds of Concrete Products Created by Three Concrete Factories

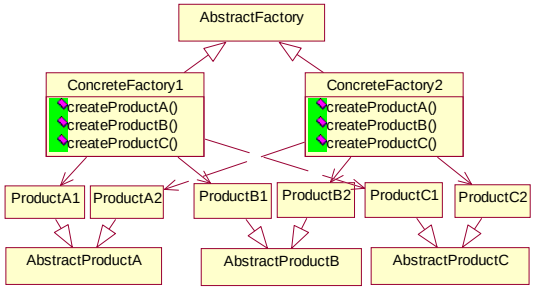

Figure 9 Abstract Factory Pattern with Three Kinds of Products Created by Two Concrete Factories

The fifth kind of pattern-level evolution is called *correlated attributes/operations* change which is the addition or removal of a group of classes. This change also requires the addition or removal of some attributes or operations in the classes of the original pattern applications. For the same example shown in Figure 7, a potential correlated attributes/operations evolution can be the addition of a new kind of product (AbstractProductC with ProductC1 and ProductC2). This requires the addition of the createProductC operation in all concrete factory classes (ConcreteFactory1 and ConcreteFactory2). The corresponding generalization and realization relationships are also added as shown in Figure 9.

All five kinds of pattern-level evolutions are summarized in Table 2.

Table 2 Summary of Pattern Level Evolution

#	Evolution Names	Description
1	Independent	Addition or removal of one independent class and the corresponding relationships between this class and the classes in the original pattern.
2	Packaged	Addition or removal of one independent class with attributes and/or operations and the corresponding relationships between this class and the classes in the original pattern.
3	Class group	Addition or removal of one attribute/operation in several different classes consistently.
4	Correlated classes	Addition or removal of a group of correlated classes.
5	Correlated attributes/operations	Addition or removal of a group of classes and addition or removal of some attributes or operations in the classes of the original pattern applications.

Table 3 The Evolutions of Design Patterns

Design Pattern Name	Pattern-Level Evolutions
Abstract Factory	4,5
Builder	4,5
Factory Method	4
Prototype	2
Singleton	N/A
Adapter	4,5
Bridge	2
Composite	2
Decorator	2,3
Façade	1
Flyweight	2
Proxy	4
Chain of Responsibility	2
Command	4
Interpreter	2
Iterator	4
Mediator	1
Memento	3
Observer	2,3
State	2
Strategy	2
Template Method	2,3
Visitor	2,5

3.3 Categorization of Pattern Evolutions

We studied the types of pattern evolutions of the design patterns listed in [9]. The result is shown in Table 3. For each design pattern, all possible evolution types of the design pattern are listed in the "Pattern-Level Evolutions" column. Consider the Abstract Factory pattern, for instance, one possible evolution is shown in Figure 8, which is classified as the fourth type of pattern-level evolution (correlated classes in Table 2). In this type of evolution, the addition of a new set of concrete products (ProductA3 and ProductB3) results in the addition of ConcreteFactory3 class. The other possible evolution is the fifth type (correlated attributes/operations) of pattern-level evolution as, for example, depicted in Figure 9. The addition of a new set of concrete products (ProductC1 and ProductC2) results in the addition of AbstractProductC class and the addition of operation (createProductC()) in the existing classes, ConcreteFactory1 and ConcreteFactory2. Thus, the Abstract Factory pattern may have the fourth and fifth types of possible pattern-level evolutions. Since the application of the Singleton pattern is not typically intended to evolve, it is labeled "N/A".

4. Automating the Pattern Evolution by Model Transformation

In the previous section, we introduce two levels of evolutions: primitive level and pattern level. We also explicitly describe the evolution processes of several design patterns in terms of the two-level evolutions. In order to automate the evolution process, we use the XSLT processor to transform from one UML model of a design pattern into the evolved UML model of the design pattern. More specifically, we translate the UML model of a design pattern into the XMI format and describe our two-level evolutions as XSLT transformation rules. Using an XSLT processor, design pattern evolutions can be automated by transforming from the original UML model of a design pattern to the destination UML model of the pattern.

Although there are several model transformation languages available currently, such as MDR (MetaData Repository from Sun [22]) and EMF (Eclipse Modeling Framework from IBM [30]), we choose XSLT to be our modeling transformation language due to the following reasons. First, current modeling languages are mostly based on MOF QVT, which is not yet finally standardized. Each modeling transformation language interprets the QVT in a different way. Second, since we are dealing with the primitive-level transformation, which is the basis of higher level transformations, i.e., pattern-level transformations, it is intuitive to use some basic type of model transformation language such as XSLT. Thus, we can define the semantic meaning of these primitive-level pattern transformations.

Table 4 The Primitive-Level Evolutions in XMI

Evolutions	Subtags of <XMI.Add> and <XMI.Delete>
class	<UML:Class name = "…" />
attribute	<UML:Attribute attributeName = "…" className = "..." />
operation	<UML:Operation operationName = "…" className = "..." />
association	<UML:Association associationEnd1 = "…"associationEnd2 = "..." />
generalization	<UML:Generalization child = "…" parent = "..." />
aggregation	<UML:Aggregation wholeName = "…" partName = "..." />
composition	<UML:Composition wholeName = "…" partName = "..." />
realization	<UML:Realization fromName = "…" toName = "..." />
dependency	<UML:Dependency fromName = "…" toName = "..." />

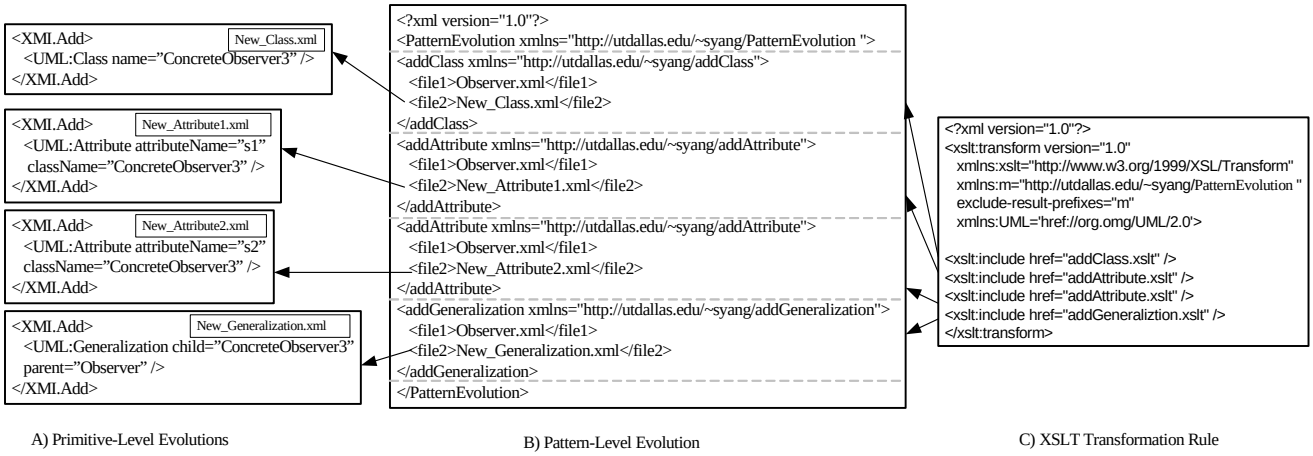

Figure 10 Pattern-Level Evolution of the Observer Pattern in XSLT

4.1 Primitive-Level Evolutions in XSLT

In this section, we describe the representations of the primitive-level evolution processes based on XSLT transformation rules. In particular, every primitive-level evolution shown in Table 1 can be represented in XMI by using XMI tags. More specifically, the addition or removal of a modeling element can be implement in XMI by the tags <XMI.Add> Subtags </XMI.Add> and <XMI.Delete> Subtags </XMI.Delete>, respectively. Table 4 shows the subtags corresponding to the primitive-level evolutions in Table 1. For instance, the first primitive-level evolution Add(Class(className)) can be implemented in XMI as follows:

```
<XMI.Add>
    <UML:Class name = "…" />
</XMI.Add>
```

Therefore, suppose the new class name is "ConcreteObserver3". The first primitive-level evolution is implemented in the XML file "New_Class.xml" in Figure 10(A) that also shows some other primitive-level evolutions, such as the additions of two attributes and one generalization relationship.

In our approach, every primitive-level evolution has a corresponding XSLT transformation rule which can add/remove the corresponding modeling element to the original UML model in XMI format. Thus, the pattern-level evolution is a sequence of primitive-level evolutions as, for example, defined in Figure 10(B), where the <addClass> tag describes the addition of a class by merging the original UML model (<file1>) and the new class (<file2>). Similarly, the <addAttribute> and <addGeneralization> tags describe the additions of an attribute and a generalization relationship, respectively.

4. Pattern-Level Evolutions in XSLT

The pattern-level evolutions are represented in XMI similarly by grouping the XMI specifications of the corresponding primitive-level evolutions. For example, the second kind of pattern-level evolution (packaged evolution) can be implemented in XMI as shown in Figure 10 (B), where a new class named "ConcreteObserver3" is added into the original application of the Observer pattern. The two attributes: s1 and s2 of the ConcreteObserver3 class are also added. In addition, the generalization relationship between the ConcreteObserver3 and Observer classes are added with the ConcreteObserver3 class as a child and the Observer class as a parent. Every model element to be added is expressed in a separate XML file, e.g., the "ConcreteObserver3" class is defined in the file named "New_Class.xml", which corresponds to the primitive-level evolution of "add a class" defined in Table 4.

For each pattern-level evolution, there are a group of XSLT transformation rules associated, which can add/remove the model elements corresponding to this pattern-level evolution into/from the original UML model in XMI format. Figure 10(C), for instance, shows the XSLT transformation rules of the packaged evolution for the Observer pattern, which adds four model elements, a class, two attributes, and a generalization relationship, into the original application of the Observer pattern as shown in Figure 4. The evolution result is shown in Figure 5. The "addClass.xslt" file describes the transformation rules on how to add the new class (New_Class.xml) into the original Observer pattern application (Observer.xml). It locates these two XML files by looking for the <file1> and <file2> tags in the <addClass> tag from the pattern-level evolutions as, e.g., described in Figure 10(B). Similarly, the "addAttribute.xslt" and "addGeneralization.xslt" files describe the transformation rules for the additions of the attributes and generalization, respectively. These XSLT files can be reused in any corresponding transformation requests.

For brevity, we omit the detailed contents of these XSLT files.

5. System Consistency Check

After a system design evolves, we need to ensure that the evolved design persists the same structural properties as the original system (assuming that the original system design has proper structural properties). There are two possible ways that the structural properties of the system are affected due to the system evolutions. First, the structure properties of design patterns applied in the system design no longer hold after the system evolution. When a new concrete product is added into the Abstract Factory pattern application, for example, the corresponding Concrete Factory class should also be added into the pattern application. Otherwise the structure properties of the Abstract Factory pattern no longer hold. Our approach can avoid this kind of problem by defining standard pattern-level evolutions for each design pattern. When the pattern application is evolution, the user may choose a particular pattern-level evolution of the design pattern and apply our XSLT transformations which include a group of tasks. For the Abstract Factory pattern example, the addition of a concrete product is grouped with the addition of the corresponding concrete factory such that the user may not miss either of them. Second, the evolution of a design may introduce the structural inconsistencies into the system even though the structural properties of a certain pattern still hold. For example, the introduction of a group of classes and generalization relationships into a design may result in a circular generalization in the design, i.e., class A is a subclass of class B which is a subclass of class C which is a subclass of A. Our transformation rules do not prevent this kind of inconsistencies. Thus, we need to perform system inconsistency check after the system evolution.

We use the Java Theorem Prover (JTP) [8][31], a knowledge based reasoning system, to perform consistency checking. In JTP, a system design model is first represented as a knowledge base in the Resource Description Framework (RDF) [32] and Resource Description Framework Schema (RDFS) format. The system properties are then expressed in JTP in term of queries. These system properties can be proved or disproved using the JTP reasoner.

In order to use JTP to prove system properties, we need to convert the XMI file corresponding to the system design into RDF/RDFS format. In RDF/RDFS, a class is defined in <rdfs:Class> tag with rdf:ID as its attribute. The value of the attribute of <rdfs:Class> defines the class name. The generalization relationship is defined in <rdfs:subClassOf> tag. The association relationship is considered as a property of the class. For instance, Figure 11 shows a partial knowledge base of the Observer pattern in RDF/RDFS format. Lines 1 and 2 define classes Subject and Observer, respectively. Lines 3 to 5 define class ConcreteSubject, which is a subclass of Subject. Lines 6 through 9 define an association between classes Subject and Observer.

```
1  <rdfs:Class rdf:ID="Subject"/>
2  <rdfs:Class rdf:ID="Observer"/>
3  <rdfs:Class rdf:ID="ConcreteSubject">
4    <rdfs:subClassOf rdf:resource="#Subject"/>
5  </rdfs:Class>
6  <rdf:Property rdf:ID="{Subject-Observer}_Subject-to-Observer">
7    <rdfs:domain rdf:resource="#Subject"/>
8    <rdfs:range rdf:resource="#Observer"/>
9  </rdf:Property>
```

Figure 11 Representing Design Pattern in RDF/RDFS

The properties of the system are expressed as queries in JTP, which are represented by a triple. More complicated queries may have logical operators, such as “and”, “or”, “not”. Figure 12 shows an example of the query in JTP, which tests whether there is circular inheritance in the design, i.e., the class ConcreteSubject is a subclass of the class Subject which is also a subclass of the class ConcreteSubject.

```
(and (|http://www.w3.org/2000/01/rdf-schema#|::|subClassOf| |file:/F:/syang/JTP/
observer.xml#|::|ConcreteSubject| |file:/F:/syang/JTP/observer.xml#|::|Subject|)
(|http://www.w3.org/2000/01/rdf-schema#|::|subClassOf| |file:/F:/syang/JTP/
observer.xml#|::|Subject| |file:/F:/syang/JTP/observer.xml#|::|ConcreteSubject|) )
```

Figure 12 A Query in RDF/RDFS

The proofs of the properties of a system can be done by invoking the “ask” command in JTP which asserts the query based on the knowledge base of the system design. The result of the query shows either success or failure. Figure 13 shows the proof of the query shown in Figure 12. The query is failed according to the system, which means that there is no circular inheritance in the system design.

6. Related Work

The evolution processes of design patterns have been studied in [1][6], where Prolog [4] is used to capture the structural evolution processes of design patterns. The structural aspect of a design pattern is described in terms of Prolog facts. Thus, the evolution of a design pattern application can be achieved by the addition or removal of new or old Prolog facts. The evolution processes are defined as Prolog rules. In this paper, we describe the evolution process as model transformations based on XMI and XSLT in terms of two-level evolutions.

Design pattern evolutions in software development processes are also discussed in [13], where software development processes are considered as the evolutions of analysis and design patterns. The evolution rules are specified in Java-like operations to change the structure of patterns. Although some primitive-level evolution

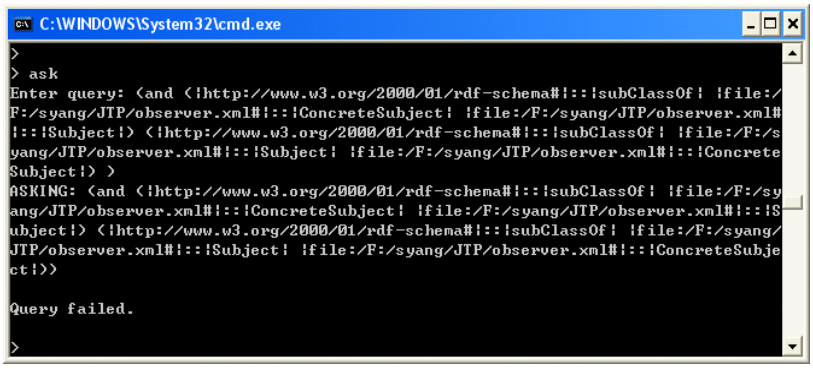

Figure 13 Property Proving in JTP

rules are introduced, there is no discussion on pattern-level evolution rules. In addition, our approach automates the evolutions based on model transformations by XSLT.

Noda et al. [16] consider design patterns as a concern that is separated from the application core concern. Thus, an application class may assume a role in a design pattern by weaving the design pattern concern into the application class using Hyper/J [21]. Due to the separation of concerns, an application class may assume different roles in different design patterns. The change of roles that an application class plays i.e., the change of design patterns, becomes a relative simple task. The main goal of their evolution of design pattern is to the replacement of one pattern by another. In contrast, our design pattern evolution refers to the internal changes of a design pattern application. In addition, the practical application of their approach is left as a mystery.

Improving software system quality by applying design patterns in existing systems has been discussed in [3]. When the user selects a design pattern to be applied in a chosen location of a system, automated application is supported by applying transformations corresponding to the minipatterns. The main goal of their software evolution is to apply design patterns in existing systems, whereas our evolution goal is to change the design patterns that have already applied in a system.

A generic XMI-based transformation infrastructure of UML models has been presented in [14]. This allows the user to select a predefined generic XML-based transformation and configure its parameters. Unlike this work, we concentrate on the XMI-based transformations for design pattern evolution.

The tool support for UML model evolution is provided in [12], which is also based on XMI. The design and development of the tool applies several design patterns. In contrast, we focus on the evolution of design patterns, instead of the evolution of UML models.

Experiments have been conducted in [5] to show that XMI can be used to transform the UML models into other modeling languages, such as SQL. The implementation of the XMI-based transformation uses XSLT. We base on these experiments and use XSLT to implement the transformations.

Kalnins et al. [10][11] proposed a graphical procedural transformation language MOLA. The model transformation defined by MOLA is a sequence of graphical statements linked by arrows. MOLA is more suitable to the transformation between two models, such as transformation from UML diagram to RDBMS schema. Other model transformation languages, such as QVT-Merge[25], ATL[18], MTF[24], Tefkat[26], and Fujaba Story diagrams (SDM)[20], which are either textual or graphical languages, address the transformation from one model to another. Muller et al. [15] also proposed a model transformation language (Kermeta) to better describe the behavioral aspect of model transformation. In contrast, our purpose is to describe the pattern evolution and automate this process. Thus, it is better to use some low lever transformation language such as XSLT to achieve the goal.

7. Conclusions

Since the evolution information of a design pattern is generally implicit in the descriptions of the pattern, a designer has to dig into the pattern descriptions to understand the particular ways of evolutions encapsulated in design patterns. There are several problems when the evolution information is implicit: first, it is hard for the designer to take advantage of the benefits of using a design pattern when changes are needed. Second, the evolution of a design pattern generally involves several classes and relationships. Missing one part may cause inconsistencies and errors in the design which are difficult to find and correct. Third, the evolution processes are not reusable if not documented. As discussed previously, many of the evolution processes recur in different patterns.

In this paper, we characterize two-level transformations: the primitive level and the pattern level and explicitly describe design pattern evolutions using these two-level transformations. We also implement the transformation based on XMI using a XSLT processor. The evolutions of each design pattern are defined as XSLT transformation rules. In addition, we use JTP to check the constraints of evolutions of each design pattern after evolutions.

We will investigate the model transformation techniques based on Query, View, Transformation (QVT) that is a forthcoming OMG standard allowing users to query, establish and maintain views, and transform MOF models. Many groups have submitted their proposals and they are still competing. There are also model transformation tools available, such as Model Transformation Framework from IBM [30] of which we can take advantage. When the QVT is standardized by OMG, we will apply the QVT techniques for the transformation of design patterns. Moreover, a case study to illustrate our approach is in our forthcoming work.

Acknowledgement

The authors would like to thank the anonymous reviewers for the helpful comments.

References

[1] P. Alencar, D. Cowan, J. Dong, and C. Lucena, A Pattern-Based Approach to Structural Design Composition, *Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference*, pp160-165, Phoenix USA, October 1999.

[2] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[3] M. Ó Cinnéide and P. Nixon. Automated Software Evolution Towards Design Patterns, *Proceedings of the International Workshop on the Principles of Software Evolution, pp162-165, Vienna*, Austria, September, 2001.

[4] W. F. Clocksin and C.S. Mellish. Programming in Prolog. Berlin: Springer-Verlag, 1987.

[5] B. Demoth, H. Hussmann, and S. Obermaier. "Experiments with XML-based Transformations of Software Models", *Workshop on Transformations in UML (ETAPS 2001 Satellite Event)*, Genova, Apr. 2001.

[6] J. Dong, P. Alencar, and D. Cowan, Ensuring Structure and Behavior Correctness in Design Composition, *Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems(ECBS)*, pp279-287, Edinburgh UK, 2000.

[7] J. Dong, S. Yang and D. T. Huynh, Evolving Design Patterns Based on Model Transformation, *Proceedings of the Ninth IASTED International Conference on Software Engineering and Applications (SEA), pp 344-350*, USA, Nov. 2005.

[8] R. Fikes, J. Jenkins, and G. Frank, JTP: A System Architecture and Component Library for Hybrid Reasoning. *Proceedings of the Seventh World Multiconference on Systemics, Cybernetics, and Informatics*. Orlando, Florida, USA. July 27 - 30, 2003.

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[10] A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA, *Proceedings of MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004)*, pp. 14-28, Linkoeping, Sweden, June 2004

[11] A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA: Extended Patterns. *6th International Baltic Conference DB@IS 2004, IOS Press, FAIA* vol. 118, 2005, pp. 169-184

[12] F. Keienburg and A. Rausch, Using XML/XMI for tool Supported Evolution of UML Models, *Proceeding of International Conference Hawaii International Conference on System Science*, Maui, Hawaii, Jan. 2001.

[13] T. Kobayashi and M. Saeki. Software Development Based on Software Pattern Evolution, *Proceedings of the Sixth Asia-Pacific Software Engineering Conference (APSEC)*, pp 18-25, Takamatsu, Japan, 1999.

[14] J. Kovse and T. Harder, Generic XMI-Based UML Model Transformations, *Proceedings of the International Conference on Object-Oriented Information Systems*, pp. 192-198, Montpellier, Sept. 2002, Springer-Verlag.

[15] P.-A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, J.M Jezequel, On Executable Meta_Languages Applied to Model Transformations, *Proceedings of INRIA Workshop of Model Transformations In Practice*, Jamaica, October 2005

[16] Natsuko Noda, Tomoji Kishi. Design pattern concerns for software evolution, *Proceedings of the 4th International Workshop on Principles of Software Evolution*, pp 158-161, Vienna, Austria, 2001.

[17] ArgoUML, http://argouml.tigris.org/

[18] ATL http://www.sciences.univ-nantes.fr/lina/atl/

[19] The Attributed Graph Grammar System (AGG) http://tfs.cs.tu-berlin.de/agg/

[20] Fujaba User Documentation http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf

[21] Hyper/J, http://www.alphaworks.ibm.com/tech/hyperj

[22] MetaData Repository, http://mdr.netbeans.org/

[23] Model Driven Architecture. http://www.omg.org/mda/

[24] MTF http://www.alphaworks.ibm.com/tech/mtf

[25] QVT-Merge, http://www.omg.org/docs/ad/05-03-02.pdf

[26] Tefkat http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/

[27] W3C, Extensible Markup Language (XML), http://www.w3.org/

[28] W3C, XSL Transformations (XSLT), http://www.w3.org/

[29] Rational Rose website. http://www.rational.com/

[30] IBM, http://www-128.ibm.com/developerworks/rational/library/05/503_sebas/

[31] JTP, http://www.ksl.stanford.edu/software/JTP/

[32] RDF, http://www.w3.org/RDF/