Design Pattern Evolution and Verification Using Graph Transformation

Chunying Zhao
University of Texas at Dallas
Richardson, TX 75083
cxz051000@utdallas.edu

Jun Kong
North Dakota State University
Fargo, ND 58105
jun.kong@ndsu.edu

Kang Zhang
University of Texas at Dallas
Richardson, TX 75083
kzhang@utdallas.ed

Abstract

This paper presents a graph transformation based approach to design pattern evolution. An evolution of a design pattern includes modifications of pattern elements, such as classes, attributes, operations and relationships between classes. Compared with other techniques, graphical notation, as a natural and intuitive way in software modeling, is suitable to be used at the transformation stage. In this paper we focus on the automated evolution of design patterns using graph transformation. The rules for the potential design evolutions are defined. After the evolution process, a graph grammar based syntax parser is proposed to check the structural integrity of the evolved design patterns.

1. Introduction

In recent years, object-oriented design patterns [1] have been widely used in the development of software systems, as systems become increasingly complex and hard to maintain. As a micro-architecture of high level abstraction, a design pattern extracts the common and featured structural information from a system. Design patterns represent the successful and reusable practice and solve general problems in a particular context, which facilitates the development of large software systems.

Intensive research has been carried out on the implementation and application of design patterns. Software systems should be adaptable to the changes of users' requirements, which require the evolution of patterns to comply with the changes of a system design. This ensures systems to be extensible and flexible since we cannot know all the requirements and build a perfect system at the beginning [2]. This type of evolution includes modifying a software element in a system without changing the basic design of the system. More specifically, classes and relationships may be added or removed in a particular design pattern without changing its structural properties [3]. Manual modifications of pattern elements require designers to go to the details of a program, which is a tedious and error-prone process. To accelerate software evolutions, researchers separate a system into the abstraction level and the implementation level, as specified in the Model Driven Architecture [4]. In a Model Driven Architecture, it saves considerable time to modify a system design at a conceptual level before actually putting a design into implementation.

Graphical notation provides an intuitive and flexible approach to describing structural information. Compared with other formal methods, graph transformation [5, 6] is a visual pattern and rule based manipulation of graph models and is suitable for describing the evolution of design patterns. Graph transformation is theoretically well founded and many matured environments and tools [7, 8, 9] are available to support the design and implementation of graph transformation. Furthermore, since design patterns are represented as UML diagrams, an evolution process can be simulated with the application of transformation rules on these diagrams.

In this paper we define the graph transformation rules that cover the potential evolutions for all design patterns. After an evolution process, consistency checking is conducted using a syntax parser to verify the structural integrity of the modified design. More specifically, a supporting visual language environment [9] can be used to check the consistency of the evolved design.

To summarize, the contribution of this paper includes:

- A graph transformation based approach that is able to evolve design patterns and extend the system is presented.
- A Reserved Graph Grammar based syntax parser is proposed to verify the structural integrity of the evolved design.

The remainder of the paper is organized as follows. Section 2 briefly introduces the Reserved Graph Grammar formalism and the concept of graph transformation. Section 3 proposes the framework of our approach and focuses on the design pattern

Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS'07)
0-7695-2755-8/07 $20.00 © 2007 IEEE
1530-1605/07 $20.00 © 2007 IEEE

evolution process using graph transformation. Section 4 shows how to check the consistency of evolved design patterns using a graph grammar based syntax parser. Section 5 reviews related work. Conclusion and future work are given in Section 6.

2. Reserved Graph Grammar and graph transformation

In this section we give an overview of the Reserved Graph Grammar (RGG) formalism [10] and graph transformation, which are the theoretical foundation of the discussion in Sections 3 and 4.

2.1. Reserved Graph Grammar

Graph grammars extend the generative grammars of Chomsky into the domain of graphs. Different from string grammar expressing sentences in sequence of characters, graph grammars are suitable for specifying visual information in a multi-dimensional fashion.

A graph grammar is made up of a set of rewriting rules, called *productions*. Each production consists of two parts: a left hand side (LHS) and a right hand side (LHS). Applying a production in a graph instance (called *host graph*) is in the form of *L*-application or *R*-application. When the right hand side of a production is matched in a host graph, it will be replaced by the left hand side of the production which is called *R*-application. The reverse process is called *L*-application. Any transformation of graphs can be realized by applying a sequence of productions. If a host graph is eventually transformed into an initial graph, the parsing process is successful and the host graph is considered to represent a type of design sharing the structural properties specified by the graph grammar [11].

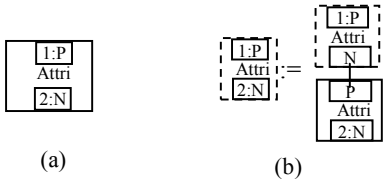

Figure 1. Node and production

The Reserved Graph Grammar is a context-sensitive graph grammar formalism, which is expressed in a node-edge format. A node is defined as a two-level hierarchy with a node itself and vertices embedded in it as shown in Figure 1(a). A node can denote any pattern element, e.g. classes, attributes and operations; an edge represents the relationship between nodes. A vertex in a node functions as a point attached to an edge.

The marking technique, which classifies a vertex as marked or unmarked, is used to deal with embedded issues: to update the connection between the replacing sub-graph and the surrounding of the replaced sub-graph in the host graph. Figure 1 (a) illustrates a node "attribute" with two vertices and (b) shows an RGG production. It denotes a group of connected attributes that can be reduced to one entity. The dashed rectangle represents a non-terminal graph symbol and the solid rectangle denotes a terminal graph symbol.

2.2. Graph transformation

Design patterns may be described by UML diagrams thus it is natural to use graphical notations to depict and transform them through graph transformation.

Graph transformation is the application of a sequence of rules on a given host graph. Slightly different from the parsing process of verification, in which the parser finally presents a parsing result, i.e. valid or not, the transformation process produces a new graph from the input host graph after applying transformation rules. Moreover, the transformation process terminates when no more transformation rules can be applied, while in the validation process the parser terminates only when the host graph is reduced to an initial graph.

Let *L* be the LHS of a grammar rule *r* and *R* be the RHS of the rule. Let *G* be a host graph. The transformation from graph *G* to graph *H* by rule *r* can be achieved through the following steps [13]:

1. Recognize sub-graph *L* in the host graph *G*.
2. Check if the transformation rule can be applied.
3. Replace sub-graph *L* by sub-graph *R*.
4. Connect the dangling edges if the vertex is marked to preserve the association with the original surrounding of the replaced sub-graph *L*.

According to Varró *et al*. [12], steps 1 and 2 can be classified as a pattern matching process and steps 3 and 4 are used to update the surrounding connections with the host graph. The example in Figure 2 shows a process of graph transformation:

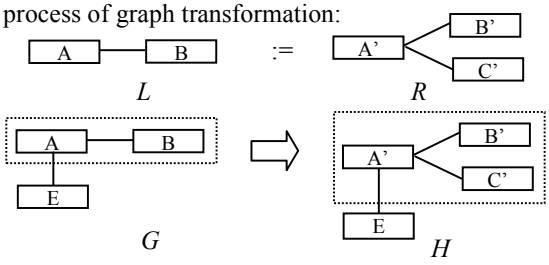

Figure 2. A graph transformation process

3. Pattern based design evolution

After introducing the concepts of the Reserved Graph Grammar and graph transformation in Section 2, we proceed to explain pattern based design evolutions, which involve the modifications of pattern elements.

3.1. Classification of design pattern evolution

In this section we focus on the evolution of pattern level design. It is very common for a designer to add/remove a software element into/from a system. For example a designer may need to add one class with its association with another class and extend the original design. Pattern elements of a software system include classes, attributes, operations and relationships, e.g. association, generation.

For all the patterns there may be addition or removal of operations, attributes and classes with corresponding relationships. From this perspective design pattern evolutions are classified into five main categories [3] as shown in Figure 3 to Figure 7.

1. **Independent**. An independent class and its relationships with other classes are added or removed. For example, in the Mediator pattern, a concrete class with a generation and an association to other classes can be added to the Mediator pattern as shown in the dashed line in Figure 3 without changing the structural integrity of this pattern.

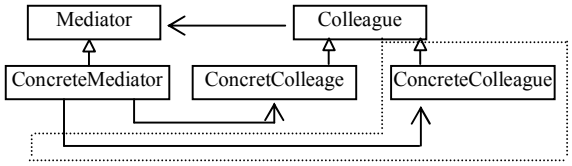

Figure 3. Independent evolution

2. **Packaged**. One independent class with attributes or operations and the corresponding relationships with other classes can be added or removed. Figure 4 shows an Observer pattern. The dashed part is the elements that can be added for extension purpose.

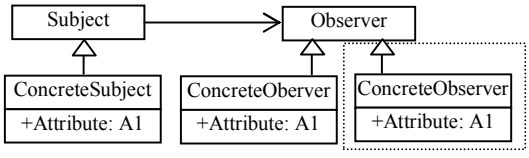

Figure 4. Package evolution

3. **Class group**. One attribute or operation can be added to/removed from several different classes consistently. Figure 5 shows an Observer pattern with a set of attributes added that are highlighted in the dashed rectangle.

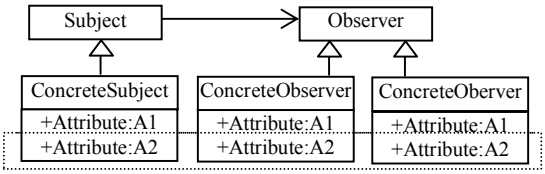

Figure 5. Class group evolution

4. **Correlated class**. A group of correlated classes are added or removed. Figure 6 shows an AbstractFactory pattern. Adding one ConcreteFactory class will be accompanied by the addition of two Product classes with the corresponding relationships as shown in the dashed areas in Figure 6.

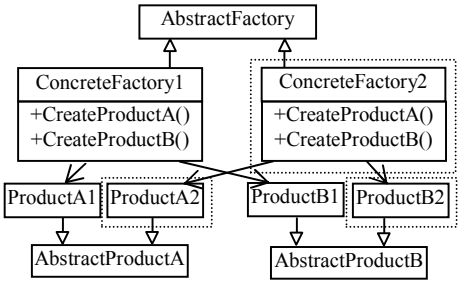

Figure 6. Correlated class evolution

5. **Correlated attribute/operation**. Adding or removing a group of classes should also add or remove the correlated attributes or operations. As shown in the dashed part of Figure 7, adding ProductB classes are accompanied by the addition of two correlated Create methods.

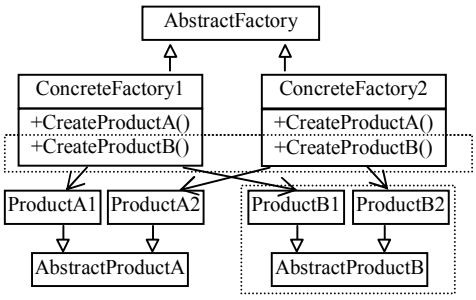

Figure 7. Correlated attribute/operation evolution

All the possible potential evolutions for each design pattern can be expressed by the above five types of transformations although different design patters have different structural properties. For example, the

possible potential evolutions of Adapter pattern are of the fourth and fifth types; the first type of evolution can occurs only in the Façade pattern and Mediator pattern [3]. Minor modifications to these basic rules are necessary when they are applied to different patterns for graph transformation. Therefore we only focus on the transformations of these five types of pattern evolutions using graph transformation.

3.2. A framework of our approach

In this section we describe the general framework of our approach. Design patterns are evolved in three steps as shown in Figure 8:

1. *Pattern generation*. Design patterns expressed in UML class diagrams are extracted and depicted in the graph editor. Each pattern is used as an input graph for the next step.
2. *Graph transformation.* The graph transformation engine consists of a set of predefined graph transformation rules for each type of pattern evolution. The input of the transformation engine includes the UML diagrams generated in the first step and the commands from users. The user should specify the desired modification, e.g. the class to be added or the type of design evolution he/she needs. The output of this step is an evolved design pattern.
3. *Consistency verification*. The consistency of the transformed pattern is examined by a syntax parser. Since any modification of a design should maintain its structural integrity, the evolved pattern should be transformed to an initial graph by a sequence of productions that represent the structural properties of a design pattern.

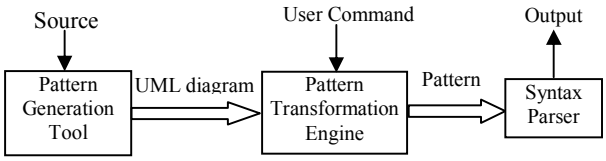

Figure 8. A framework of pattern evolution process

3.3. Graph based design pattern evolution

According to the definition and classification of pattern evolutions, we define graph transformation rules for each type of evolution respectively, as shown in Figure 9. For space limitation, we only present the addition of software elements. The principle is similar to that of the removal process if the rules are applied reversely.

In rule (a), a new independent class and its relationships with its surrounding are added. We can match the graph of LHS of this rule with the host graph and replace the matching part with the RHS. Similarly, in rule (b), a new concrete class with its attributes is added. In rule (c), a new attribute is added to a set of concrete classes. Rules (d) and (e) appear more complicated since the additions of some classes are accompanied by the additions of other elements (classes or methods) besides the relationships with other classes.

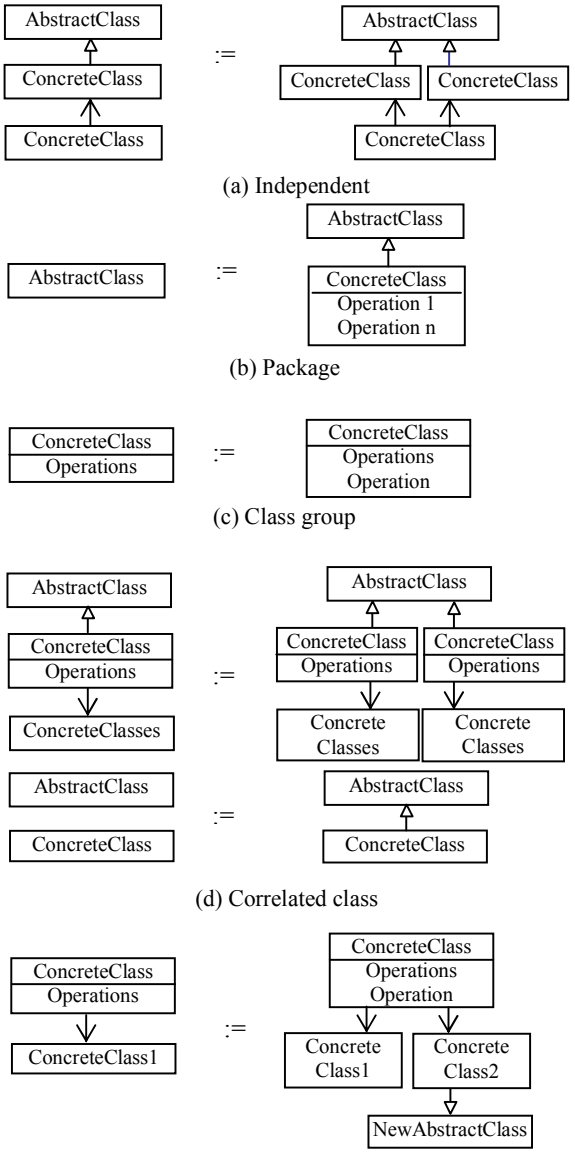

Figure 9. Pattern evolution rules

For instance, to realize a Correlated class evolution, we need to add a new ConcreteFactoray class with corresponding addition of some new Product classes thus the transformation rules in Figure 9 (d) can be used. This transformation consists of two steps. First, a new ConcreteFactory class and its corresponding concrete product classes are added. Second, the inheritance relationship between AbstractClass and ConcreteClass is added. To illustrate a sample evolution process, we use the independent rule in Figure 9(a) to show how to extend the Mediate pattern. The shaded elements of graph *G* in Figure 10 are matched by the LHS of the independent rule and replaced by the RHS of the rule. The graph *H* is the evolved pattern.

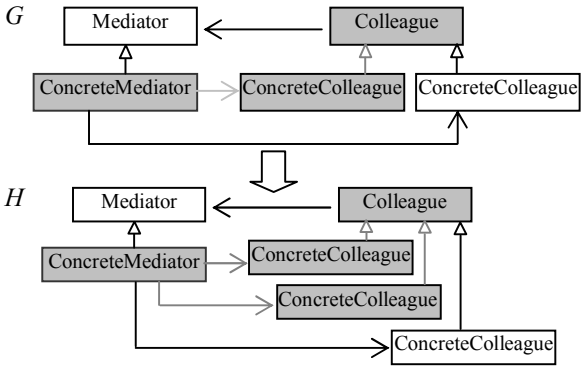

Figure 10. A graph transformation process

4. Consistency checking

After the design is evolved, we need to check the consistency to verify that the structural properties and constraints of the changed design are preserved. Consider the transformation process in Figure 10, when a ConcreteColleague class is added to the graph *G*, the association with ConcreteMediator class and the generation with Colleague class should also be added. If any of these relationships is missing, the transformed pattern will no longer be a Mediator pattern and the transformation is inconsistent.

Based on the Reserved Graph Grammar formalism, a generic parsing approach is used to check the consistency. A set of productions are defined for each pattern. If a pattern can be reduced to an initial graph by a sequence of productions, the evolved pattern is considered to conform to the structural properties of a particular design pattern represented by the graph grammar and the evolution of the design is proved correct.

We define relationships: association, generation, composition, etc, as terminal symbols. Pattern elements such as classes, attributes and operations are represented as non-terminal symbols determined by their syntactic meanings. An edge between two vertices represents a connection of two nodes. A dashed rectangle represents a non-terminal node and a solid rectangle denotes a terminal node. For brevity, we only define the productions for three design patterns mentioned earlier in this paper, as shown in Figure 11.

Production 1 shows that an attribute can be reduced from a group of attributes or one single attribute. Similarly production 2 defines a set of operations. Production 3 is a class that consists of a class name, operations and/or attributes. These three productions can be viewed as the building blocks of design patterns. Productions 4 - 6 specify an Observer pattern. Productions 7 - 9 represent a Mediator pattern. Productions 10 - 13 define an AbstractFactory pattern.

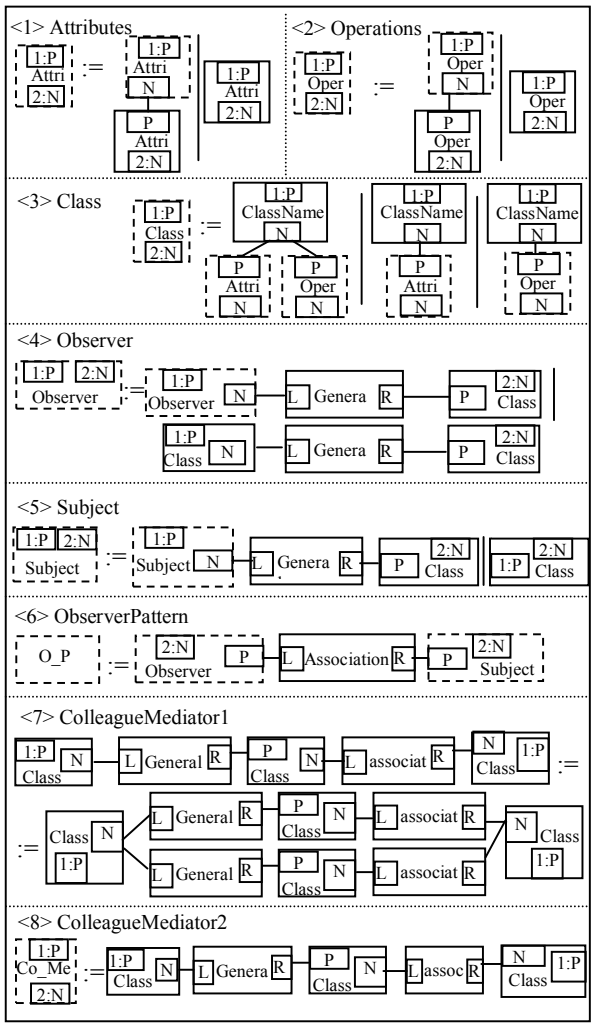

Figure 11-1. Design pattern productions

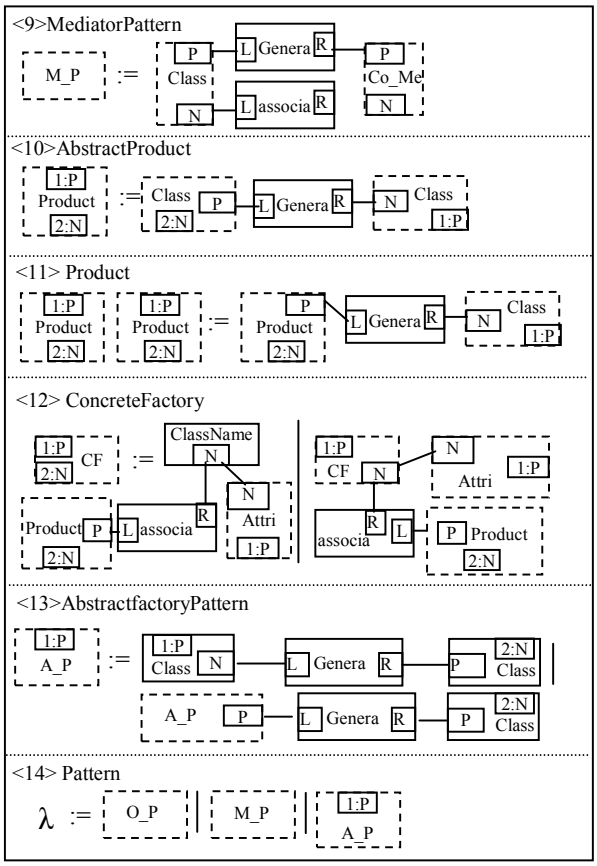

Figure 11-2. Design pattern productions

5. Related work

Graph transformation has been successfully used in many applications [14, 15, 16].

Dong *et al*. [3] proposed that the potential evolution of all design patterns could be categorized into five basic types. An XMI-based approach to design pattern evolutions was used to change the UML class diagram. This requires users to manually specify all the pattern elements that relate to a pattern evolution, i.e. classes, attributes, operations or relationships, and group these elements in one transformation. However, the user has to keep in mind of all the possible elements modified without having an intuitive scenario of the transformation process. Performing this task by hand will be tedious and error-prone. Also, to conduct the consistency checking, Java Theorem Prover (JTP) was deployed to verify the system. The XMI files had to be converted to an RDF/RDFS format before verification, and only circular inheritance was checked in the system. In contrast, the approach presented in this paper specifies all the necessary manipulations in a graph grammar rule, which enhances the expressiveness and accuracy of the evolution. For consistency checking, we do not need any intermediate transformation of pattern elements, since an RGG parsing process can be performed directly on the evolved pattern diagrams generated by graph transformation. Not only circular inheritance but also the structural integrity of the evolved design pattern is examined.

Kobayashi *et al*. [17] considered pattern evolutions from the viewpoint of software development. They evolved Analysis Patterns in the requirements analysis step to Design Patterns in the design step. The process of evolving customers' requirements into a final design was considered as a software development. The evolved patterns did not maintain the structural property of the original pattern. To achieve the target transformation from one pattern to another, designers need to group a set of operations on the diagrams. The idea of encapsulating a sequence of operations as one transformation on a diagram is similar to that of Dong *et al*. [3] and also error-prone. No consistency checking of this transformation was mentioned.

Costagliola *et at*. [18] proposed a design pattern recovery approach and patterns were expressed in terms of visual grammars. Design patterns were retrieved by a pattern recognition parser. This parser used an attributed-based representation of XPG grammar, which is not as expressive as the RGG. The RGG reserves the structural information by linking to other components through edges and vertices.

6. Conclusion and future work

Evolution of design pattern represented in UML diagrams is a common and important practice in the development of software systems.

This paper has presented a graph transformation approach to design pattern evolution and a consistency verification process using a Reserved Graph Grammar parser. Based on the classification of design pattern evolutions, we define graph transformation rules to manipulate the pattern elements and extend the system. To verify the consistency after transformation, a graph grammar parser is proposed and productions are defined for each design pattern.

The transformation rules in the transformation stage and the grammar productions in the verification stage are applied in sequential steps specified by designers. To accelerate the process, an automated system that integrates the current graph transformation environments with the pattern evolution rules, will be further investigated. More experiments will be conducted on complex systems that include compound

design patterns. We will also conduct empirical studies in the future work.

7. References

[1] E. Gammar, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[2] M. Ó Cinnéide, P. Nixon, "Automated Software Evolution Towards Design Patterns", *Proceedings of the 4th International Workshop on Principles of Software Evolution*, pp. 162-165, 2001.

[3] J. Dong, S. Yang, and K. Zhang, "A Model Transformation Approach for Design Pattern Evolutions", *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, pp. 80-92, 2006.

[4] Model Driven Architecture, *http://www.omg.org/mda.*

[5] G. Rozenberg (Ed.), *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations*, vol. 1, World Scientific, 1997.

[6] L. Baresi and R. Heckel, "Tutorial Introduction to Graph Transformation: A Software Engineering Perspective", *International Conference on Graph Transformation*, LNCS 2505, Springer, pp. 402-439, 2002.

[7] S. Burmester, H. Giese, J. Niere, M. Tichy, J. P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf, "Tool Integration at the Meta-Model Level: the Fujaba Approach", *International Journal on Software Tools for Technology Transfer,* vol. 6, pp. 203-218, 2004.

[8] A. Schürr, A. Winter, A. Zündorf, "Graph Grammar Engineering with PROGRES", *Proceedings of Europe Software Engineering Conference*, pp. 219-234, 1995.

[9] K. Zhang, D. Q. Zhang, and J. Cao, "Design, Construction, and Application of a Generic Visual Language Generation Environment", *IEEE Transaction on Software Engineering*, vol. 27, pp. 289-307, 2001.

[10] D. Q. Zhang, K. Zhang, and J. Cao, "A Context-sensitive Graph Grammar Formalism for the Specification of Visual Languages", *Computer Journal*, vol. 44, pp.187-200, 2001.

[11] J. Kong, K. Zhang, J. Dong, and G. Song, "A Graph Grammar Approach to Software Architecture Verification and Transformation", *Proceedings of the 27th Annual International Computer Software and Applications Conference*, pp. 492-497, 2003.

[12] G. Varró, A. Schürr, and D. Varró, "Benchmarking for Graph Transformation", *Proceedings of the 2005 IEEE Symposium on Visual Language and Human-Centric Computing*, pp. 79-90, 2005.

[13] Y. Zhao, Y. Fan, X. Bai, Y. Wang, H. Cai, and W. Ding, "Towards Formal Verification of UML Diagrams Based on Graph Transformation", *Proceedings of IEEE International Conference on E-Commerce Technology for Dynamic E-Business*, pp. 180-187, 2004.

[14] J. Kong, G. Song, and J. Dong, "Specifying Behavioral Semantics through Graph Transformation", *Workshop on Visual Modeling for Software Intensive Systems*, pp.51-58, 2005.

[15] D. Le Métayer, "Describing Software Architecture Styles Using Graph Grammars", *IEEE Transaction on Software Engineering*, vol. 24, pp. 521-533, 1998.

[16] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr, "Application of Graph Transformation to Visual Languages", *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Language and Tool*, Vol.2, World Scientific, pp. 105-180, 1999.

[17] T. Kobayashi and M. Saaki, "Software Development Based on Software Pattern Evolution", *Proceedings of the 6th Asia-Pacific Software Engineering Conference*, pp. 18-25, 1999.

[18] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino and M. Risi, "Design Pattern Recovery by Visual Language Parsing", *Proceedings of the 9th European Conference on Software Maintenance and Reengineering*, pp. 102-111, 2005.