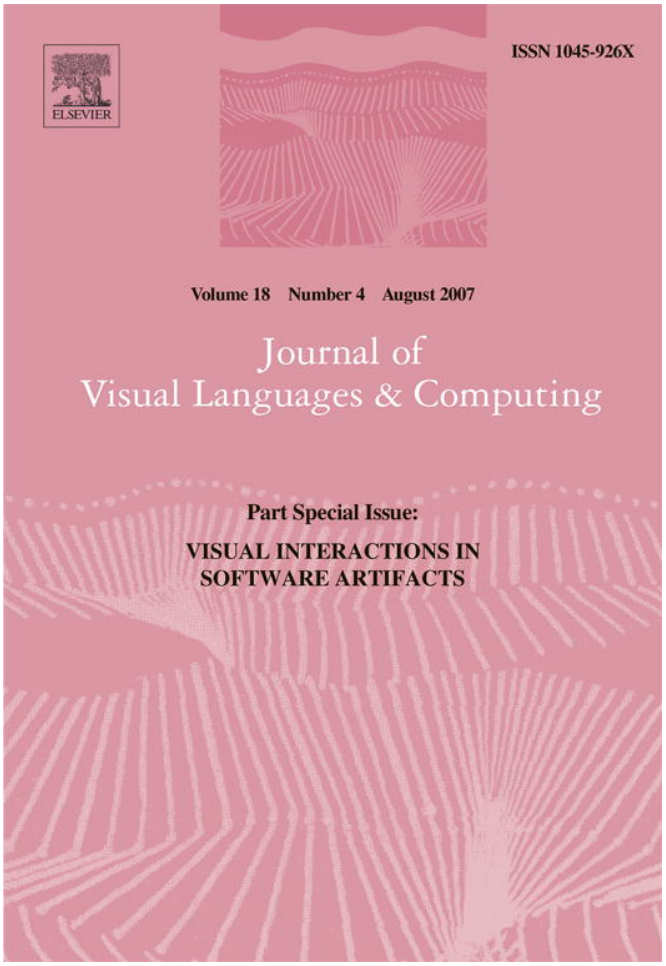

Available online at www.sciencedirect.com
ScienceDirect
Journal of Visual Languages and Computing
18 (2007) 378–398

Journal of Visual Languages & Computing

www.elsevier.com/locate/jvlc

Pattern-based design evolution using graph transformation

Chunying Zhao[a,*], Jun Kong[b], Jing Dong[a], Kang Zhang[a]

[a]*University of Texas at Dallas, Richardson, TX 75083, USA*
[b]*North Dakota State University, Fargo, ND 58105, USA*

Abstract

In recent years, design patterns gain more interest in software engineering communities for both software development and maintenance. As a template to solve a certain recurring problem, a design pattern documents successful experiences of software experts and gradually becomes the design guidelines of software development. Applying design patterns correctly can improve the efficiency of software design in terms of reusability and enhance maintainability during reverse engineering. Software can be evolved when developers modify their initial designs as requirements change. For instance, a developer may add/delete a set of design elements, such as classes and methods. Modifications on software artifacts can introduce conflicts and inconsistencies in the previously applied design patterns, which are difficult to find and time-consuming to correct. This paper presents a graph-transformation approach to pattern level design validation and evolution. Based on a well founded formalism, we validate a given design by a graph grammar parser and automatically evolve the design at pattern level using a graph-transformation system. Rules for potential pattern evolutions are predefined. The graph-transformation approach preserves the integrity and consistency of design patterns in the system when designs change. A prototype system is built and a case study on the Strategy pattern demonstrates the feasibility of pattern-based design validation and evolution using graph transformation techniques.
Published by Elsevier Ltd.

Keywords: Design pattern; Graph transformation; Graph grammar; Visual language; Software evolution

*Corresponding author.
E-mail addresses: cxz051000@utdallas.edu (C. Zhao), jun.kong@ndsu.edu (J. Kong).

1045-926X/$ - see front matter Published by Elsevier Ltd.
doi:10.1016/j.jvlc.2007.07.004

1. Introduction

Software systems become increasingly complicated and hard to maintain due to the massive information in the system and complex system structure. There has been an urgent need for an efficient and effective method to ease the tedious work at software development and maintenance stages. The emergence of object-oriented design patterns [1] makes software development more efficient in terms of reusability because design patterns document the successful experience of experts and provide solutions to certain reoccurring problems in particular contexts. Application of design patterns facilitates software development in that it allows software engineers to communicate and collaborate with each other efficiently. Well documented design patterns also assist the comprehension of legacy programs in reverse engineering.

In addition to efficient software development methodologies, software systems should always be adaptable to evolve upon modification requests from users or designers. This requires a system to be extensible and flexible since we cannot know all the requirements and build a perfect system at the beginning [2]. Arthur et al. [3] defined software evolution as "a continuous change from a lesser, simpler, or worse state to a higher or better state" for software system. To efficiently achieve such an evolution, modifications to a system generally start from the design level that hides massive programming information because the size of a design is typically much smaller and more manageable than that of codes.

Since Gamma et al. [1] first introduced the well recognized design patterns, intensive research has been carried out on the application of design patterns. The wide use of design patterns requires an effective mechanism to validate software designs and allow designers to make modifications on them as system requirements change. On one hand, it is necessary to guarantee that a design complies with the structural integrity of the design patterns that have been applied. On the other hand, any change on the design should not violate the structural properties of existing design patterns in the system, since a local modification may have a chain effect on the whole system. Manually checking the impact of a single modification, however, is a time-consuming and error-prone process. In general, design evolutions that happen at pattern level includes refactoring [4] and design pattern evolution [5]. Refactoring is to find the "bad smell" in the system and reconstruct it in order to achieve better efficiency with external behaviors preserved. Design pattern evolution is to refine a design pattern according to requirements changes while maintaining the pattern's properties. A design pattern normally includes changeable and stable parts. The changeable part of a design pattern can be potentially adapted to several different applications, while the stable part will remain. More specifically, pattern participants, e.g. classes and relationships, may be added to/removed from a particular design pattern without violating the pattern's structural properties and constraints. We defined such a process as a *pattern level design evolution*, in which pattern participants in a system design are modified but the fundamental properties of the design are preserved at the pattern level.

As a *de facto* standard to visually modeling software systems, UML provides a set of intuitive notations to represent design patterns and corresponding designs. The lack of formal semantics, however, prevents UML from automatic validation and evolution. Dong et al. [5] automated the design evolution process 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 an XMI format

to facilitate the transformations. To take advantage of the graphical characteristics UML diagrams, we explore a graph-transformation approach to pattern level design validation and evolution.

Graph transformation, which offers a computational paradigm of mathematical precision and visual specification [6], provides an intuitive yet formal means to interpret and manipulate visual languages. In general, graph transformation defines computation in a multi-dimensional fashion based on a set of rewriting rules, i.e. *productions*. Each production consists of two parts: a left graph and a right graph, the difference between which visually indicates the changes generated by a computation. By representing a design as a graph, the problem of design evolution is converted to graph evolution, which can be naturally realized through graph transformation: the left graph of a production indicates the pre-condition of an evolution and the right graph represents the post-condition after an evolution. Applying graph transformation to design evolution can guarantee that the design after evolution still observes the structural properties of its underlying patterns. A graph grammar system [7] extends a graph transformation system by (1) defining an initial graph and (2) classifying terminal and non-terminal objects. In this way, a graph grammar abstracts the essential structures shared by a set of graphs. Therefore, we formalize the structure of a design pattern into a graph grammar, and correspondingly validate the structural integrity of a design through a parsing process, i.e. a sequence of graph-transformations. Using visual representation to interpret software evolution and validation improves the expressiveness to human's understanding and communication.

In summary, this paper presents a graph-transformation framework for pattern level design validation and evolution, which is characterized by a syntax-oriented graphical design environment facilitated by an automatic design evolution tool. The graphical design environment can verify whether a design observes the structural integrity of a design pattern while the evolution tool can guarantee that a design after evolution is still consistent with the structural properties of its underlying patterns. Validation of a design is a parsing process on a graph representing the design. In order to improve the performance of the parsing process, each design pattern defines one key structure. We first locate the key structure in a design and then initiate a parsing process from the key structure. The key structure can narrow down the parser's searching scope and thus achieve a better performance. Based on the characteristics of GoF design patterns, we categorize pattern level design evolutions into five types [11] (e.g. independent evolution and packaged evolution), which are applicable to different design patterns. Each evolution style in a specific design pattern is defined through graph-transformation. When an evolution is carried out through graph transformation, the syntactic correctness of the evolved design is guaranteed with respect to a design pattern.

The contributions of this paper are threefold:

- Formalizing design pattern structures using graphical representation.
- Validating patterns in a design by the spatial graph grammars.
- Realizing pattern-based design evolutions through graph-transformations.

The rest of the paper is organized as follows. Section 2 briefly introduces the Spatial Graph Grammar (SGG) [8] formalism and concepts of graph transformation. Section 3 explains the classification of design pattern evolutions. Section 4 presents the overview of our framework and investigates a graph-transformation approach to design pattern

validation and evolution. Section 5 shows a prototype system and illustrates an example using the Strategy pattern. Section 6 reviews related work. Section 7 concludes this paper and discusses the future work.

2. The spatial graph grammar and graph-transformation

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

The SGG formalism [8] is a context-sensitive graph grammar formalism, which allows the left graph to have multiple nodes, and is expressive in specifying various types of graphs. The SGG formalism is expressed in a node-edge format as presented in Fig. 1. In an SGG, nodes are organized into a two-level hierarchy, where a large rectangle representing the node itself is the first level with embedded small rectangles as the second level called *vertices*. Fig. 1(a) depicts a typical SGG node, which includes various vertices. In a node, each vertex is uniquely labeled. A node can be viewed as a module, a procedure or a variable, etc., depending on the design requirement and object granularity. A vertex functions as a port to connect other nodes by an edge. Edges denote communications or relationships between nodes.

With a well-established theoretical background a graph grammar can be used as a natural and powerful syntax-definition formalism [7] to specify visual languages and the parsing algorithm based on a graph grammar may be used to check the syntactic correctness. More specifically, applying a production to a host graph can be classified as an *L*-application (i.e. replacing the left graph of a production with the right graph of the production) or *R*-application (i.e. a replacement from the right graph to the left graph). The visual language, defined by a graph grammar, can then be derived by using *L*-applications from an initial graph. On the other hand, *R*-applications are used to verify the membership of a graph. If a host graph is eventually transformed into an initial graph, i.e. a special symbol λ, 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.

Due to the multi-dimensional nature of graphs, some mechanisms are needed to address the embedding issue in a graph transformation [9], i.e. establishing relationships between the surrounding of a replaced graph and its replacing graph in the host graph. Inherited from the reserved graph grammar [10], the SGG addresses the embedding issue by the *marking technique*. In a production, a vertex is marked by prefixing its label with a unique

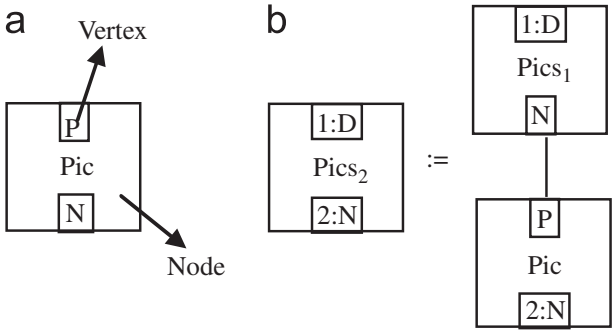

Fig. 1. Spatial graph grammar representation: (a) A node; (b) A production.

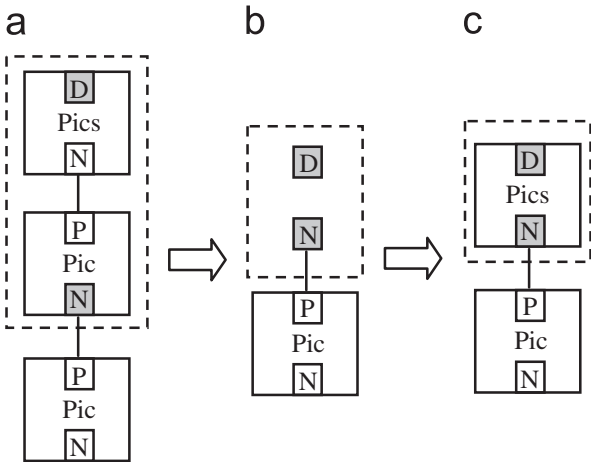

Fig. 2. The marking technique: (a) before transformation; (b) preserve context information; (c) after transformation.

integer. For example, in the production of Fig. 1(b), vertex D of $Pics_1$ is marked while the vertex N is unmarked. If a vertex v in a replaced sub-graph (i.e. a *redex*) maps to a marked vertex in a production, v will be preserved during graph-transformation to establish connections between the surrounding of the replaced sub-graph and a new replacing sub-graph. As the sub-graph (the dotted rectangle) in Fig. 2(a) is isomorphic to the right graph of the production in Fig. 1(b), the vertices in Fig. 2(a) that map to the marked vertices in Fig. 1(b) are highlighted with gray color. Those highlighted vertices will be preserved in a graph-transformation as shown in Fig. 2(b), and the transformed graph is shown in Fig. 2(c).

3. Pattern based design evolution

3.1. Problem statement

Design patterns have been widely accepted as guidelines documenting the design of an object-oriented system since Gamma et al. [1] introduced the popular 23 design patterns. Each design pattern has a structure, behavior and semantic meaning. The structure indicates the structural relations among pattern elements; the behavior tells how pattern participants interact with each other and the semantic meaning defines the context of a problem addressed by the design pattern.

Therefore, each design pattern has some essential properties. For example, the Mediator pattern should have a concrete mediator referred to each of its concrete colleagues while concrete colleagues do not have references between each other. In real modeling, a design can evolve with variations to satisfy application requirements without violating the pattern's essential properties. For example, we introduce a new concrete colleague class to the mediator pattern as shown in Fig. 3. Such an evolution is not trivial since any local modification may cause the evolved design violating some underlying essential properties. If a designer ignores the neighboring elements influenced by the modification, the change can lead to inconsistency and a consequent malfunction. For instance, in Fig. 3, the introduction of class ConcreteColleague3 requires to establish one inheritance and one association with Colleague and ConcreteMediator (highlighted with dotted circles), respectively, in order to maintain the essential properties of the mediator pattern. Missing

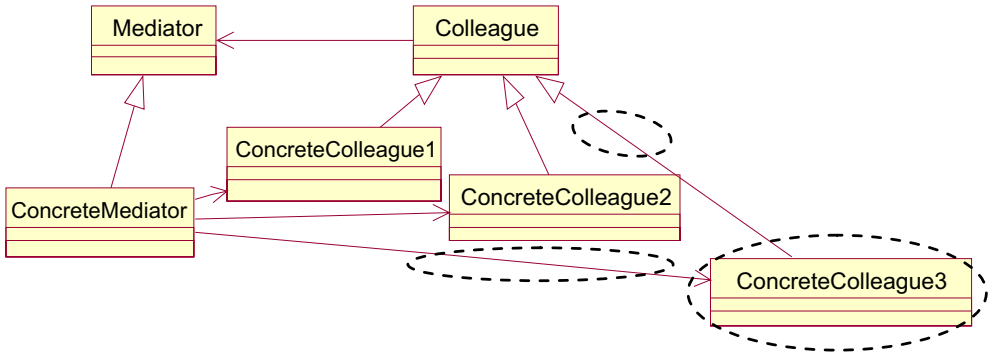

Fig. 3. The Mediator pattern.

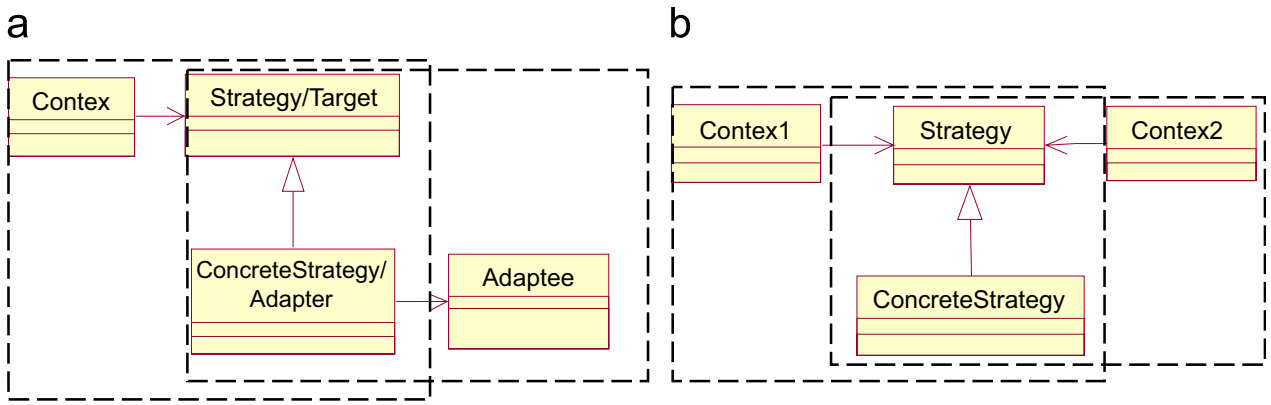

Fig. 4. Overlapping patterns.

any of these elements violates the constraints of the mediator pattern. However, it is tedious and error-prone to require designers to perform all the required modifications propagated within the design.

This paper aims at developing a graphical design environment that supports automated pattern-based design evolutions, in which two challenging issues will be addressed: (1) the pattern structural integrity of a design needs to be validated since a syntactically correct design is the prerequisite of a successful pattern-based design evolution; (2) an automatic mechanism is desirable to perform pattern-based design evolution while keeping the evolved design correct and consistent.

There are some issues that need special attention when we perform pattern-based design validation and evolution. For instance, design elements, e.g. a class, may participate in multiple design patterns. Patterns sharing common elements can be the same or different as shown in Figs. 4(a) and (b) respectively. In Fig. 4(a), the Strategy pattern overlaps with the adapter pattern. In Fig. 4(b) two Strategy patterns share the same "strategy" and "concreteStrategy". If a modification occurs to the shared elements of multiple design patterns we need to consider the influences to all patterns involved.

3.2. Classification of pattern-based design evolution

The pattern-based design evolution involves the modifications of pattern participants. Pattern participants of a software system include classes, attributes, operations and

relationships, e.g. association and generalization. For example, a designer may introduce a new class to extend the original design. As discussed in Section 3.1, the modification to a pattern element may affect other pattern participants. In real world modeling, design patterns are applied in the system with variants to meet various application requirements. We explore the changeable part in each design pattern that allows future extensions. Pattern level design evolutions could change the structure and behavior of a system by means of addition/deletion of software artifacts while keeping the fundamental properties of its underlying pattern invariant.

Although different patterns have different properties, they share some common evolution requirements and actions. From this perspective, we summarize five pattern level evolutions that are recurring in different design patterns. For each evolution style, the changes will not violate the properties and constraints of the original design. We classify pattern-based design evolution into five main categories [11] as follows:

- *Independent*: The addition or removal of one independent class and the corresponding relationships between this class and the classes in the original pattern. The added or removed 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. For example, in Fig. 5(a), a concrete class with a generalization and an association to other classes in the dashed area can be added to the mediator pattern without changing the structural integrity of this pattern.

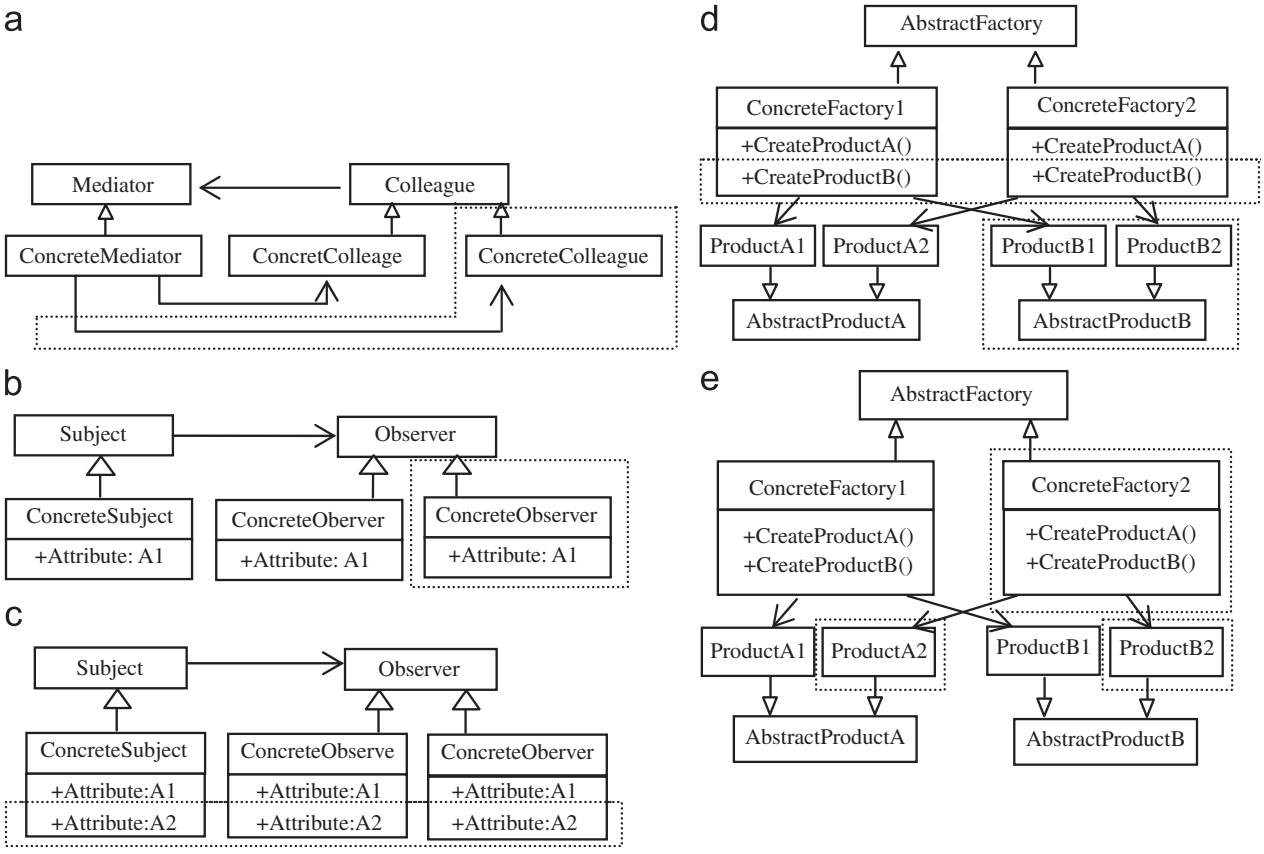

Fig. 5. Classification of possible evolutions.

- *Packaged*: 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. Fig. 5(b) shows the observer pattern. The dashed rectangle shows the addition of one ConcreteObeserver class with one attribute.
- *Class group*: 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. Fig. 5(c) shows the observer pattern with an additional set of attributes that are highlighted in the dashed rectangle. This pattern indicates that a new data is observed by all the observers.
- *Correlated class*: 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. Fig. 5(d) shows an AbstractFactory pattern. Adding one ConcreteFactory class is accompanied by the addition of two product classes with the corresponding relationships as shown in the dashed areas.
- *Correlated attribute/operation*: The addition or removal of a group of classes, that requires the addition or removal of some attributes or operations in the classes of the original pattern applications. As shown in the dashed part of Fig. 5(e), adding ProductB classes has to be accompanied by the addition of two correlated CreateProductB() methods to the classes in the original design.

The category of design pattern evolution offers a guideline for developers as they evolve a software system and guarantees pattern consistency. Table 1 summarizes the relationships between the pattern evolutions and GoF design patterns. For example, the CorrelatedClass and CorrelatedAttribute/Operation evolutions can be applied to the adapter pattern.

Based on the classification of pattern evolutions, this paper presents a graphical environment with a graph-transformation engine that supports an automatic design evolution. Given a design pattern, each applicable evolution is specified through a sequence of graph-transformation rules. By explicitly representing the changes caused by a design evolution in the graph-transformation process, a complete evolution can be carried out without missing any modifications on the affected pattern participants. Therefore we avoid the tedious work of manually checking each evolved design.

Table 1
Potential evolutions of design patterns [11]

Type	Evolution name	Design pattern name
1	Independent	Mediator, Façade
2	Packaged	Prototype, Bridge, Composite, Decorator, Interpreter, Observer, State, Strategy, TemplateMethod, Visitor, Chain of responsibility
3	ClassGroup	Decorator, Observer
4	CorrelatedClass	AbstractFactory, Builder, FactoryMethod, Adapter, Proxy, Command, Iterator
5	CorrelatedAttribute/ Operation	AbstractFactory, Builder, Adapter, Visitor

4. Graph-transformation approach to design pattern evolution

4.1. Overview

This section describes a general framework of our approach as shown in Fig. 6. Given a design, a graph-transformation-based design pattern evolution proceeds in three steps [12]:

Design representation: Using a graphical design environment, a designer can apply design patterns to a system design represented in UML class diagrams. Alternatively, design patterns can be retrieved from source codes using reverse engineering techniques. In either case, a design pattern expressed in a UML class diagram can be automatically transformed into a node-edge representation, which can be recognized by the SGG parser. Then, a design pattern in a node-edge format can be validated and evolved through graph-transformation in the next two steps.

Design validation: Since a syntactic-correct design pattern is the prerequisite for a successful evolution, we need to validate the structural and behavioral properties of the given pattern. An SGG based parsing approach is used to examine the fundamental properties of a pattern. More specifically, we formalize the fundamental properties of a design pattern into a graph grammar. Based on the graph grammar, if a design pattern can be finally reduced to an initial graph, i.e. a special symbol λ, through a parsing process, the design satisfies all essential properties enforced by the pattern. Given a host graph representing a design, the start point of a parsing is essential to a successful design validation. For example, based on the graph grammar in Fig. 9, the parser will fail in the validation of a design in Fig. 8 if the first class analyzed is not the "Strategy". One brutal-force approach is to start the parsing process exhaustively for each symbol in the host graph once [13]. In order to improve the parsing performance, we employ the key structure in each design pattern, which characterizes the essential property in a design pattern. To validate the structure and behavior properties of a design, we first locate the key structure in the design and then start the parsing process from the key structure. In summary, the validation module consists of two steps:

1. Locate the key structure in a design.
2. Start parsing from the key structure.

The first step can be realized by a linear searching algorithm which will be explained shortly. The searching space is small since there are normally a limited number of nodes in

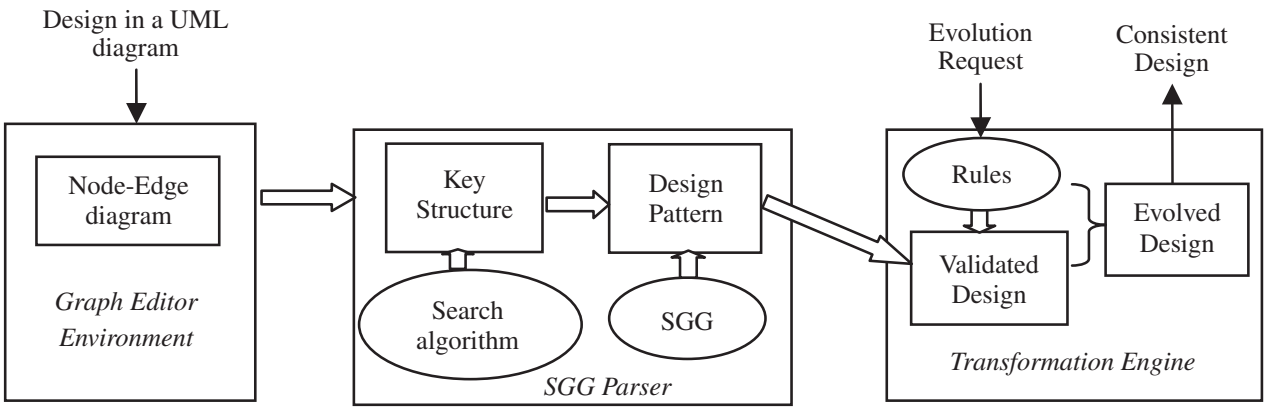

Fig. 6. Overall framework.

each pattern. At the second step, we apply a graph grammar to a node-edge diagram corresponding to the user-defined design.

Design evolution: The graph-transformation engine consists of a set of predefined graph-transformation rules for each type of pattern evolutions. The input to the transformation engine includes the node-edge diagram of a design, which has been verified in the second step, and the modification request specified by the user, e.g. adding/removing a class. According to the modification request, a set of predefined graph-transformation rules are selected to evolve the design. The graph-transformation rules apply all the changes to the evolving design in order to keep its fundamental properties invariant. In order words, the productions define both the changes to the design elements specified by a user and other related changes caused by the user's changes. Therefore, using the graph-transformation engine for the evolution will guarantee the consistency of the evolved pattern.

4.2. Design representation

As described in Section 2, we employ the SGG formalism to specify pattern-based design validation and evolutions. Both the system under study and the evolving design are represented as graphs. This section introduces preliminary concepts that are intrinsic to graph grammars and design representations.

Class diagram, one of the most popular diagrams in UML, visually models the static structure of a system in terms of classes and their relationships. In order to verify the structure of a class diagram, we translate a class diagram (Fig. 7(a)) into a node-edge format (Fig. 7(b)), on which the SGG parser operates. Nodes denote classes and relationships, e.g. *Association* in a UML class diagrams. Each node includes a set of vertices. Each node and vertex has a label to represent its meaning.

In class diagrams, classes are represented by compartmentalized rectangles including classes, attributes and operations. In a node-edge diagram, a node labeled *Class* denotes the first compartment containing the class name. A set of nodes labeled *Attri* represents attributes in the second compartment. These nodes are sequenced by linking two adjacent attributes in the order as they are displayed in the compartment, and the sequence of attributes is attached to a class name by linking the first *Attri* node with the *Class* node.

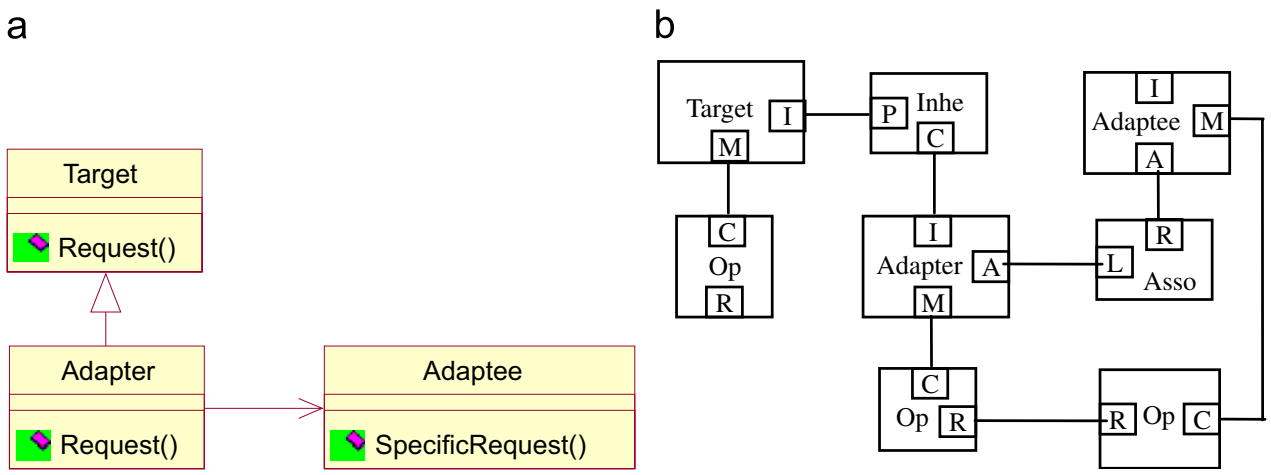

Fig. 7. The adapter pattern and its node-edge representation.

Operations in the third compartment are processed in the same way as attributes by replacing *Attri* with *Oper* nodes.

An association denoted by straight or diagonal lines in UML carries information about reference relationships between two classes. In a node-edge diagram, a node labeled *Asso* is used to symbolize an association between two nodes. To indicate the navigability of *association*, there are two vertices labeled *L* and *R* inside the node *Asso*. The vertices labeled *L* and *R* inside an *Asso* node are connected to a referencing *Class* node and a referenced *Class* node, respectively. On the other hand, if a relationship is undirected in the representation of a design pattern, we ignore the difference between *L* and *R*. *Aggregation* and *Composition*, two special types of associations, are translated in the same way as associations.

A generalization specifies a hierarchical relationship between a general description and a specific description. In the node-edge representation, a node *Inhe* is used to denote the generalization between two classes. Similarly to the *Asso* node, there are two vertices inside *Inhe* labeled *P* and *C*, which are used to connect the parent class and the child class, respectively.

Definition 1. Given a vertex set Ω, a *node* is 3-tuple $N = (V, l, name)$, where V is a set of vertices, $l: V \to \Omega$ is an injective function to label vertices, and *name* is the node label.

In particular, there are two kinds of vertices in a node that represent a class in a design pattern: *relationship* vertex and *method call* vertex. As shown in Fig. 7, the graph defines the adapter pattern. The node named "adapter" represents a class name and has two vertices labeled '*I*' and '*A*' to connect with the *generalization* and *association* relationships, respectively. The vertex '*M*' connects to a *method call* node.

Definition 2. A directed *host graph* is a 4-tuple $G = (N, E, s, t)$, where N is the node set, $E \in N \times N$ is the edge set, and $s, t: E \to V.N$ are two functions that specify the source and target points of an edge. $V.N$ is the set of vertices within N.

Fig. 7 is a host graph instance representing the adapter pattern in the node-edge format. Nodes denote classes, relationships and interacting method calls. Vertices imply directions of relationships. For instance, the inheritance node named *Inhe* has two vertices *P* and *C* that connect to a superclass and a subclass, respectively, impling a class 'adapter' inherits a class 'target'.

Definition 3. Two nodes n_1 and n_2 are *isomorphic*, denoted as $n_1 \approx n_2$, iff

$$\exists f((f : n_1.V \to n_2.V) \wedge \forall v \in n_1.V(n_1.l(v) = n_2.l(f(v))) \wedge n_2.s = f(n_1.s)).$$

The definition requires that two nodes are isomorphic if and only if they have the same vertices and node labels. For instance, the node "class" in the production $\langle 1 \rangle$ of the Strategy pattern in Fig. 9 and the node "class" in the node-edge representation of the Strategy pattern in Fig. 8 are isomorphic.

4.3. Design validation

4.3.1. Key structure

We first need to validate if a user-defined pattern instance complies with some fundamental structural properties before it evolves. In general, there exists a key structure [14] in each design pattern that plays an essential role in gluing different design elements into a complete pattern. For instance, in the adapter pattern in Fig. 7, the class 'adapter' with an association and inheritance relationships is the key structure for the adapter

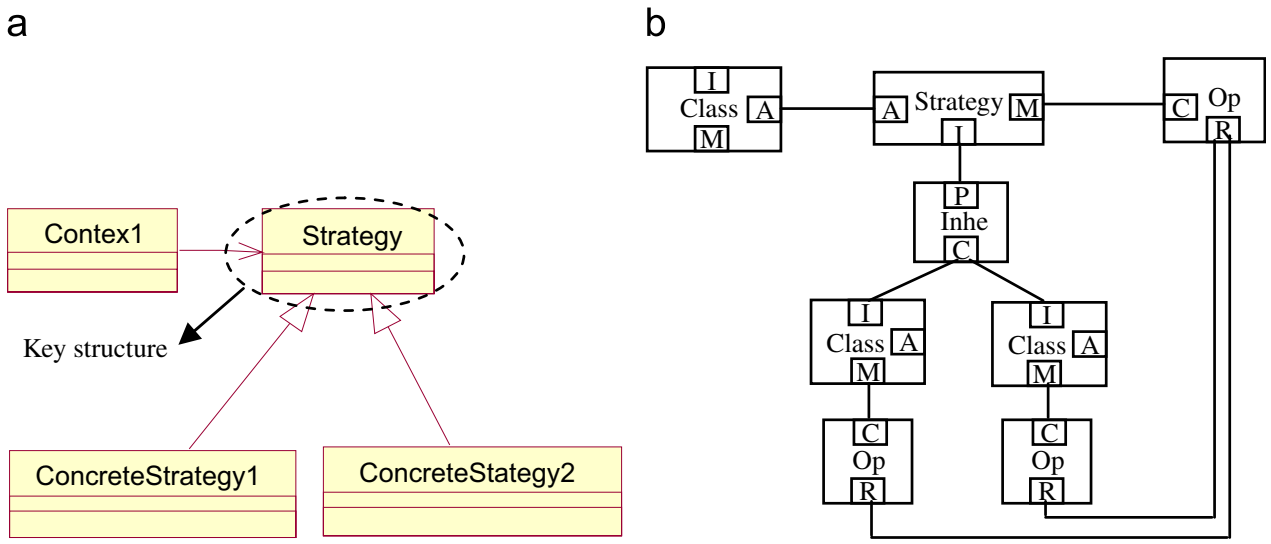

Fig. 8. The Strategy pattern and its node-edge representation.

pattern. For the Strategy pattern, the class "Strategy" is the key structure. The key structure functions as a nexus in a design with the highest number of connections with pattern participants. Moreover, its behavior shows the major semantic meanings that distinguish the pattern from other patterns. A key structure can comprise more than one node.

We develop a linear searching algorithm to locate the key structure in a design pattern. A key structure is characterized by: (1) the number and type of its connections with neighboring elements; (2) the types and names of nodes. For instance, the key structure in the Strategy pattern is a non-abstract class with one aggregation, at lease one generalization and at least one connection to a method call. This information is the criterion to identify the key structure of a design pattern in the linear search.

In the implementation of the linear searching algorithm, the node-edge graph is a 2-tuple G (V, E) and a class is represented in as a 2-tuple (N, R). N and R are one-dimensional arrays that store the class information and its neighboring relationships, respectively. Relationships are distinguished by the weights of edge. As each host graph has a limited number of nodes, the two-dimensional graph can be stored in a one-dimensional array with all the node information recorded in an arbitrary order. The algorithm examines each node and its connections. If a node that satisfies the property of a key structure is identified, we will mark it. The process continues until all nodes are scanned once. The pseudocode of the algorithm is shown as follows:

Algorithm KEYSTRUCTURE_SEARCH
Input: A graph G (V, E)
Output: A sub-graph of G
Create a Queue Q
Enqueue (Q, V)
While Q is not empty do
  for each vertex *v* in Q do
    for all edges e incident on v do
      Add edge e's type to *v*

If v 'edge types satisfy those of a key node in key structure
  Mark v and mark its edges.
Dequeue(Q,v)

4.3.2. Graph grammar parsing

Definition 4. Two graphs G_1 and G_2 are *homomorphic*, denoted as $G_1 \approx G_2$, iff $\exists f$: $G_1 \to G_2$, where f is a mapping such that $\forall n \in G_1.N : n \approx f(n)$ and $\forall e = (v_1, v_2) \in G_1.E : f(e) = (f(v_1), f(v_2)) \in G_2.E$.

Graph homomorphism is an essential concept in graph parsing. When the parser matches a redex G_1 (a subgraph of a host graph) with a graph G_2 of a rule, the redex G_1 and the graph G_2 must be homomorphic. A graph grammar includes a set of productions, which formally summarizes the essential properties of a design pattern. Staring from the key structure located in the first step, we apply the graph grammar to the user-defined pattern. If the pattern can be reduced to an initial graph, i.e. λ, by a sequence of productions, the pattern conforms to the structural properties of a particular design pattern.

Fig. 8(a) shows the Strategy pattern and Fig. 8(b) is its corresponding node-edge representation, in which operations are explicitly added in order to validate the interactions between classes. The syntax productions are defined based on the node-edge representations. Pattern participants such as classes, attributes and operations are represented as non-terminal symbols with respect to their syntactic meanings. Fig. 9 illustrates the productions for the Strategy pattern. In this grammar, production $\langle 1 \rangle$ (for brevity $P\langle 1 \rangle$) reduces the operations in a connected structure composed of a Strategy class and a ConcreteStretegy class. $P\langle 1 \rangle$ further eliminates the ConcreteStrategy node that connects to an inheritance node and the operation node attached to the Strategy node. $P\langle 3 \rangle$ reduces the last ConcreteStrategy node in the structure. $P\langle 4 \rangle$ indicates that a Context-Strategy structure can be generated from the structure produced by $P\langle 3 \rangle$. The last production specifies the context-strategy structure can be reduced to λ. Reaching the state λ implies a successful parsing. Fig. 10 illustrates a parsing process that initiates from the class "strategy" on the Strategy pattern with predefined productions.

4.4. Design evolution

As both the original design pattern and evolved design pattern are represented as graphs, a pattern level design evolution can be naturally interpreted as a graph-transformation: the left graph defines the precondition before an evolution while the right graph specifies the postcondition after evolution. The graph-transformation process proceeds by finding a redex in a host graph and replacing it with the right graph of a transformation rule.

According to the definition and classification of pattern evolutions described in Section 3, each design pattern has certain types of potential evolutions. Each evolution is uniquely determined by two factors: the desired type of evolution and the pattern name.

Definition 5. A pattern level design evolution E is a partial function of type T and Pattern P: $T \times P \to E$, where T = {Independent, Packaged, ClassGroup, CorrelatedClass, CorrelatedAttribute/Operation}, and P = {GoF design patterns}

For instance, according to Table 1 the Strategy pattern may be changed in a *Packaged* evolution. In this type of evolution, developers can add or remove an independent class

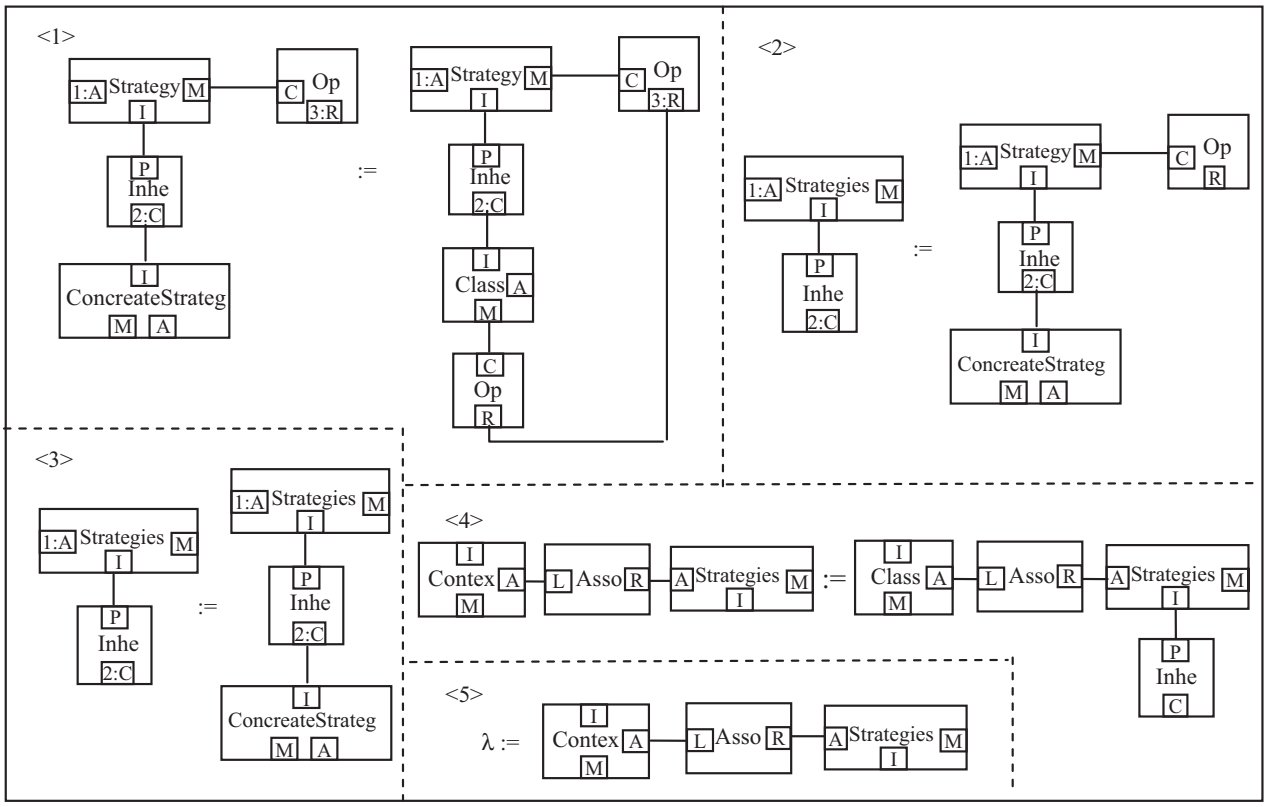

Fig. 9. Graph grammars for the Strategy pattern.

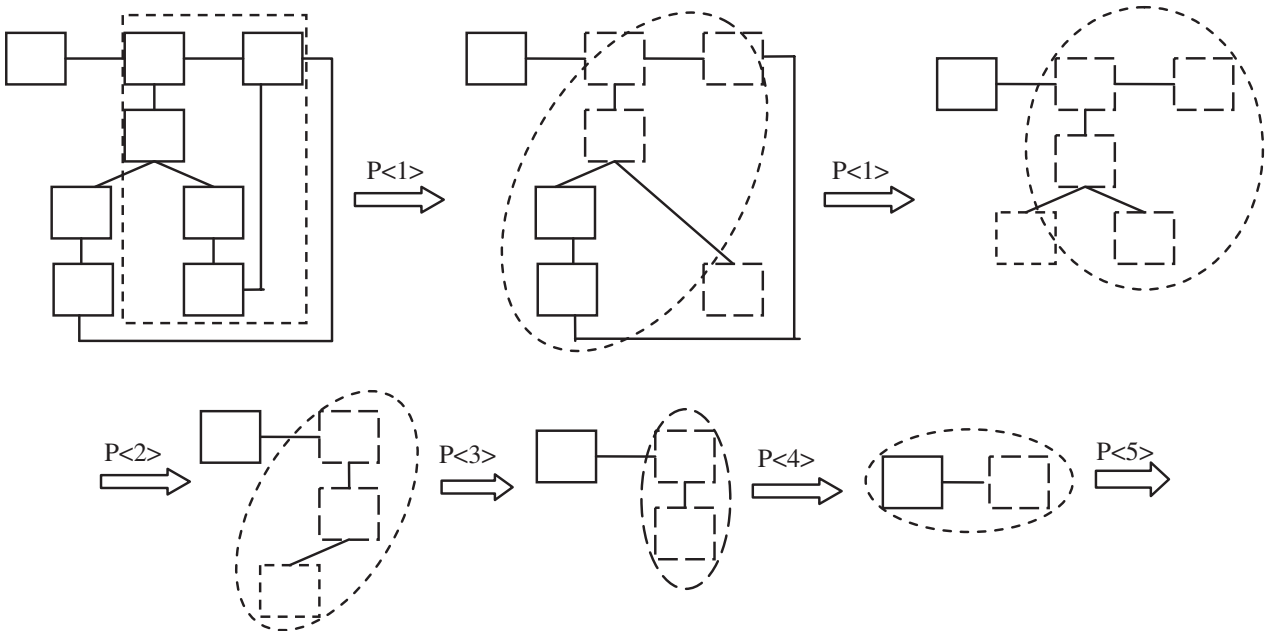

Fig. 10. A paring process of the Strategy pattern.

(e.g. the ConcretateStrategy2 class in Fig. 11) with its corresponding method or attributes (e.g. the AlgorithmInterface method) and relationships with other classes (e.g. the inheritance relationship with strategy) as well. Fig. 12 gives the production to realize a

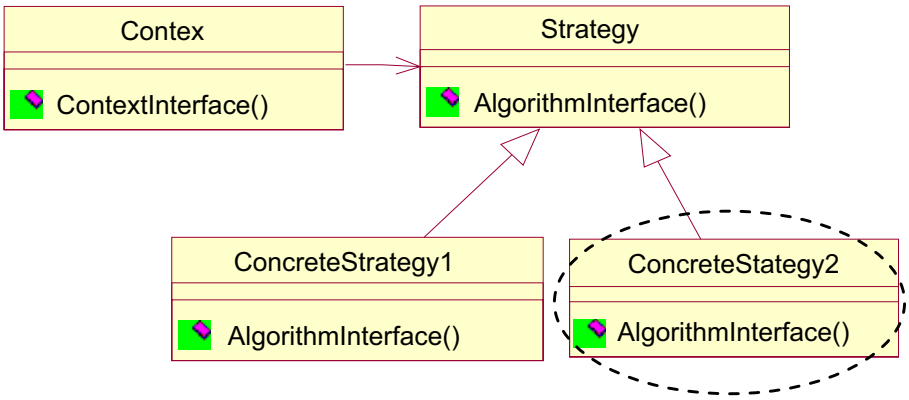

Fig. 11. Packaged evolution on the Strategy pattern.

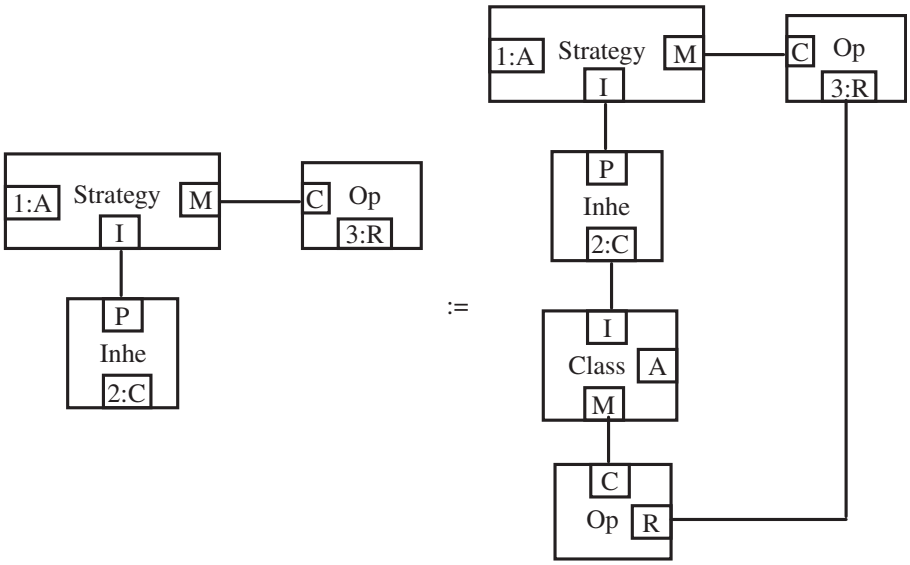

Fig. 12. A production for packaged evolution on the Strategy pattern.

Packaged evolution for the Strategy pattern. Before applying the production to a host graph representing the original design pattern, the transformation controller will first find a match of the left graph of the production, i.e. a redex, in the host graph. In this case, the redex consists of an inheritance node, a strategy node and an operation node. Once the redex is found, it will be replaced by the right graph of the production. As the productions encapsulate all necessary modifications, all the elements can be changed together to keep the fundamental properties invariant. Otherwise, manually performing the modification will cause some potential problems. Moreover, a manual modification requires a deep understanding of the evolved pattern since missing modification on any affected pattern participant will violate the structural integrity.

5. A prototype system

This section shows a prototype system for an automated design pattern evolution based on SGG. At the system level, the graph-transformation approach described above is realized by four modules as shown in Fig. 13: node editor, production editor, graph-transformation engine and graphical editing tool in the node-edge format.

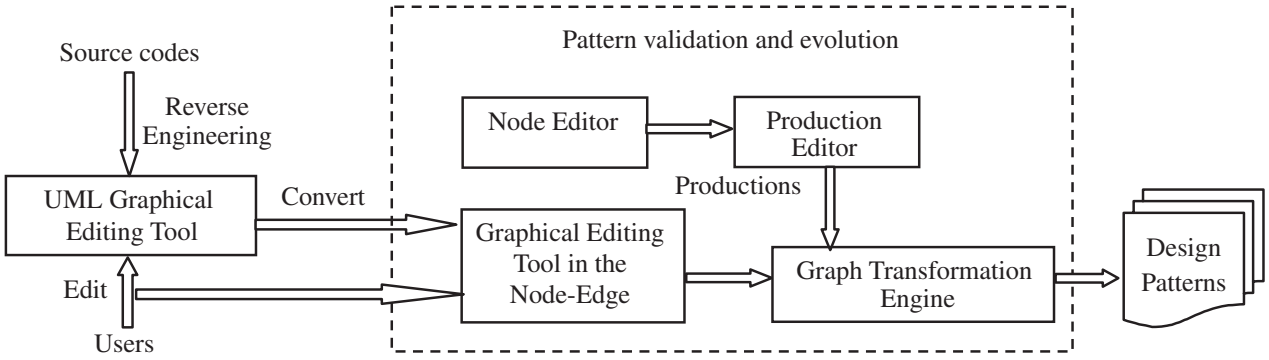

Fig. 13. System architecture.

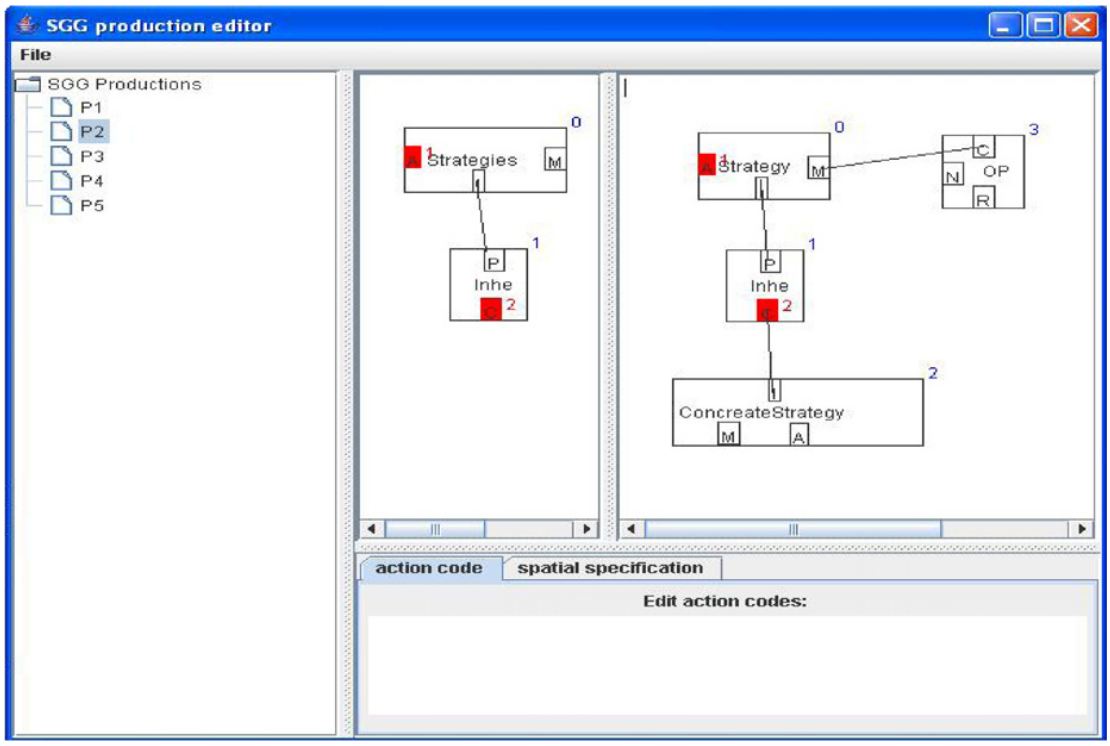

Fig. 14. SGG production editor.

The node editor and production editor provide an easy-to-use user interface to define graph grammar and graph-transformation rules. A graph grammar formalizes the essential properties of a design pattern while graph-transformation rules specify design evolutions. Designers can use a graphical editing tool to edit design patterns in the node-edge representation. Alternatively, designers can also specify design patterns in the form of UML, which are automatically converted to a node-edge representation. The graph-transformation engine is the kernel of the system, which validates the structure of a design pattern and evolves a design upon users' requests. The graphical design system allows users to define nodes, edit host graphs, customize production rules and automatically parse host graphs. If a host graph can be parsed successfully, the system will report a valid result and show the parsing path.

In the case study, we use the Strategy pattern as a host graph and illustrate its parsing process. Fig. 14 is the screenshot of the production editor. All productions of the Strategy

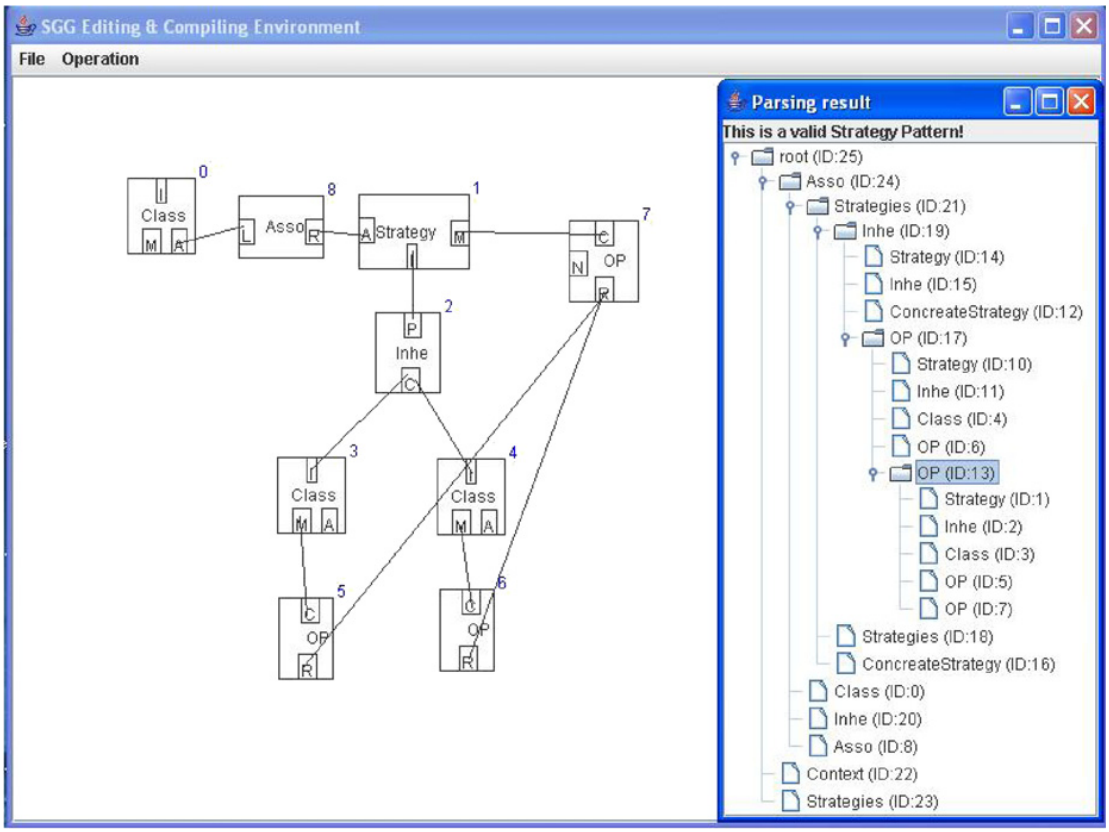

Fig. 15. A valid Strategy pattern.

pattern are listed in the leftmost column. The two columns on the right show the left graph and right graph of production $\langle 2 \rangle$, respectively, as defined in Section 4.3 (Fig. 9). Shaded vertices are marked. We can also import predefined productions. Fig. 15 illustrates the host graph of the Strategy pattern and a valid parsing result. The popup shows a valid parsing result on the host graph and a tree structure that indicates how the host graph is constructed by the grammar. The host graph in Fig. 16 is not valid since the connection between nodes 7 and 9 is missing, which violates the structural properties of the Strategy pattern.

Similarly, design pattern evolutions can also be conducted by this system after minor modifications to the parser and applying the rules differently. This case study shows the feasibility of using SGG formalism to represent and validate design patterns.

6. Related work

Design pattern becomes popular in OO software research communities since Gamma et al. [1] proposed the catalog of design patterns. Research on design patterns mainly focuses on the techniques for pattern application, pattern recovery and pattern evolution. With the development of visual language applications [15–17], many research work have taken advantage of the formal foundation of visual languages, i.e. graph grammars and transformations, to address issues in the area of design patterns.

Radermacher [18] used graph queries and graph rewriting rules to specify design patterns. In his work, applications were reconstructed to meet certain prerequisites of a middleware for distribution purpose. Systems were represented as graphs using

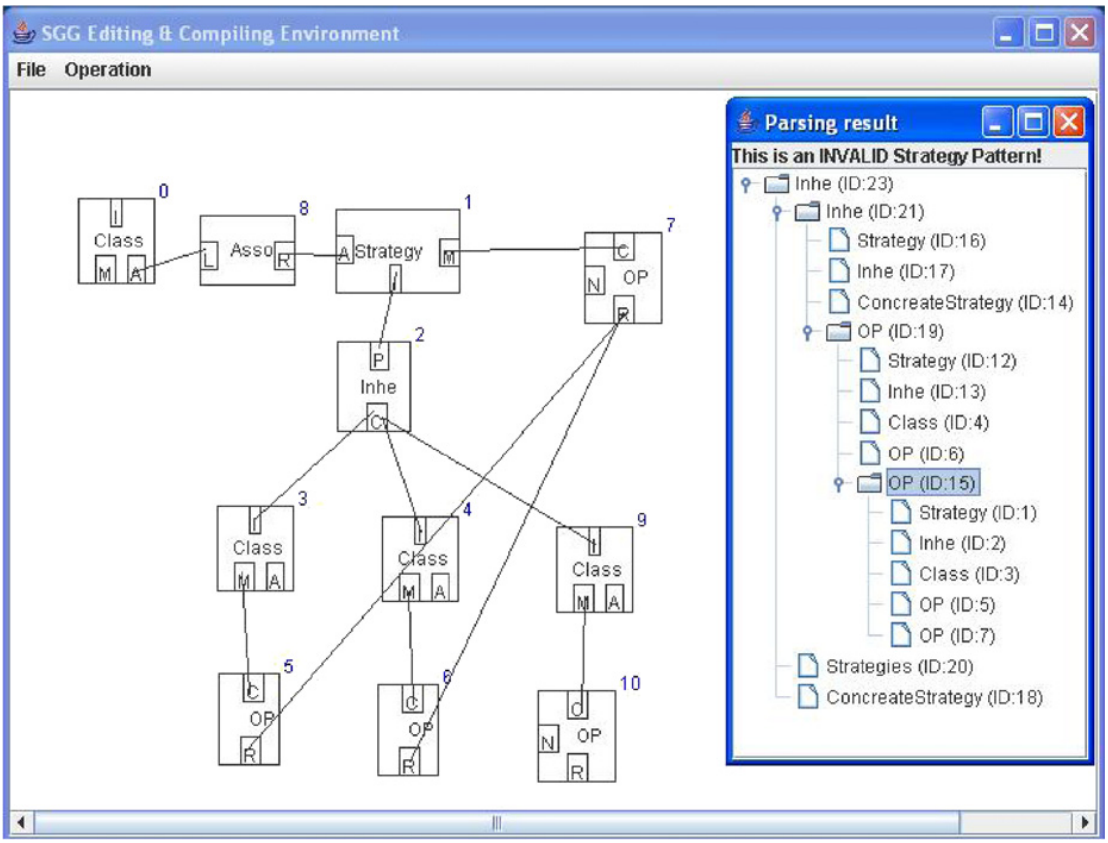

Fig. 16. An invalid Strategy pattern.

PROGRES schema. Graph queries were used to detect violations to the distribution prerequisites. A supporting tool was built to transform a system in a way such that it conforms to a specific pattern. His work defined the system transformation as patterns based on the prerequisites of distribution, which is an application of design pattern and different from out work that uses graph-transformation to realize pattern evolution.

Another pattern application is refactoring. One aim of refactoring is to apply a suitable design pattern to enhance software flexibility. Mens et al. [19] adopted typed graph to represent source codes and formalized two refactorings: encapsulated field and pull up method. The result showed that graphs rewriting rules could specify the source-code transformation implied by a refactoring and the formalism guarantees the behavior preservation. The two refactoring reconstructed the system by implicitly applying a design pattern. Their work was the transformation from a design without a design pattern to a design with design patterns.

Costagliola et al. [13] proposed a design pattern recovery approach using visual languages. Patterns were expressed in terms of visual grammars and retrieved by a pattern recognition parser. This parser used an attributed-based representation of XPG grammar, which is not as expressive as the SGG. The SGG keeps structural information by representing classes and relationships as nodes and linking them via edges. Moreover, to parse a graph they exhaustively examined every node as a start point. On the contrary, we used a linear searching algorithm as a preprocessing to locate a key structure as the start symbol, which greatly improves the parsing efficiency step.

There are also many design pattern recovery techniques that do not employ visual language concepts. Shull et al. [20] proposed an inductive method to manually discover customized, domain specific patterns from OO software systems. They tried to find out recurring patterns that solved some specific problems by mining common structures from students' assignments. Antonial et al. [21] presented a multi-stage reduction strategy using software metrics and structural properties to extract design patterns from OO designs or codes. In their work, codes and designs were mapped into an intermediate representation, called Abstract Object Language. Ferenc et al. [22] presented the Columbus framework to recognize design patterns from C++ source code. They build an abstract semantic graph that contains all the information about the source code. Columbus uses a XML-based DPML (Design Pattern Makeup Language) file to depict a design pattern and matches the graph of DPML to the ASG to find patterns represented by DPML. Tsantails et al. [23] developed a pattern detection methodology based on a similarity scoring algorithm between graph representations of the pattern to be detected and the system under study. This approach allows the detection of customized patterns. These work mine patterns from systems mainly based on pattern structures. The evolution of patterns was, however, not addressed.

Dong et al. [11] defined the classification of pattern evolutions and proposed an XMI-based approach to design pattern evolutions. Different from our work, both the original and evolved UML models were presented in XMI format. The evolution was performed as XSLT transformation with a set of user predefined rules. Java Theorem Prover (JTP) was deployed to verify the system. The XMI files had to be converted to an RDF/RDFS format before validation. In contrast, our approach specifies all the necessary manipulations in graph grammar rules, which enhances the expressiveness and accuracy of the evolution. Furthermore, design representation conversions between validation and evolution stages are avoided in our system since we use the same graphical format to denote a system design.

Ciraci et al. [24] formalized software evolution using algebraic graph rewriting. They modeled software architecture as a colored graph that combined the information from UML class diagrams and interaction diagrams. The evolution requests were viewed as morphisms on the components of the software system. They use transformation rules for evolution requests and argue that these rules can be combined to realize various evolution requests. Indeed, graph rewriting techniques can formalize software evolution; Ciraci et al., however, did not apply them to the design pattern level software evolution, nor to the validation any structure in the system.

Kobayashi et al. [25] considered pattern evolutions from the perspective of software development. They evolved the analysis patterns that represent system requirements to the design patterns at the design level. The evolved patterns do not necessarily maintain the structural properties of the original pattern because the analysis patterns and the design patterns serve for different purposes. In their work, the evolution of patterns was to transform the artifacts in previous stages into new ones during a software development cycle. The evolution target was different from ours and the consistency issue of this approach had not been addressed.

7. Conclusion and future work

Evolution of design patterns represented in UML diagrams is an important issue in software development. This paper has presented a graph-transformation approach to

pattern-based design validation and evolution, which helps developers to validate their pattern level designs and allows further consistent modifications. We defined a set of productions for each design pattern based on its fundamental structure and behavior properties. We examined the characteristics of design pattern and summarized the potential evolutions for each pattern. Based on the classification of design pattern evolutions, we specified graph-transformation rules to manipulate the pattern participants while maintaining the underlying pattern properties of the design. As the transformation rules for each evolution include all the influenced pattern participants, the evolved design is consistent with the original one.

Based on the SGG formalism, we implemented a syntax-oriented graphical design environment facilitated by an automatic design evolution tool. A design is validated by SGG parser in the graphical environment and the syntax-correctness of the evolved pattern is guaranteed by the evolution tool. We investigated the applicability of our approach by running an example of the Strategy pattern.

As the future work, we will incorporate the approach to UML modeling tools by automatically converting UML diagrams to node-edge diagrams. Productions for each design pattern will be semi-inducted automatically from given software systems [26]. The concept of pattern evolution defined in this paper can be extended if we can derive productions for customized patterns. The behavior aspect of a design pattern will also be formalized.

Acknowledgments

The authors would like to thank the Guest Editors and the anonymous reviewers for their insightful and constructive comments that have helped us to significantly improve the paper.

References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: Elements of Reusable Object-oriented software, Addison-Wesley, Reading, MA, 1995.
[2] M. Ó Cinnéide, P. Nixon, Automated software evolution towards design patterns, in: Proceedings of the fourth International Workshop on Principles of Software Evolution, 2001, pp. 162–165.
[3] L.J. Arthur, Improving Software Quality: An Insider's Guide to TQM, Wiley, New York, 1993.
[4] J. Zhang, Y. Lin, J. Gray, Generic and domain-specific model refactoring using a model transformation engine, Model-driven Software Development-Research and Practice in Software Engineering, Springer, Berlin, 2004.
[5] J. Dong, S. Yang, Y. Sun, E. Wong, QVT based model transformation for design pattern evolutions, in: Proceedings of the 10th IASTED international conference on Internet and multimedia systems and applications (IMSA), 2006, pp.16–22.
[6] D. Varró, A formal semantics of UML statecharts by model transition systems, in: Proceedings of ICGT 2002, Lecture Notes in Computer Science, vol. 2505, 2002, pp. 378–392.
[7] G. Rozenberg (Ed.), Handbook of graph Grammars and Computing by Graph-Transformation: Foundations, vol. 1 (1), World Scientific, Singapore, 1997.
[8] J. Kong, K. Zhang, X. Zeng, Spatial graph grammars for graphical user interfaces, ACM Transactions on Computer–Human Interaction (TOCHI) 13 (2) (2006) 268–307.
[9] J. Rekers, A. Schürr, Defining and parsing visual languages with layered graph grammars, Journal of Visual Languages and Computing 8 (1) (1997) 27–55.
[10] D.Q. Zhang, K. Zhang, J. Cao, A context-sensitive graph grammar formalism for the specification of visual languages, Computer Journal 44 (3) (2001) 187–200.

[11] J. Dong, S. Yang, K. Zhang, A model transformation approach for design pattern evolutions, in: Proceedings of the 13th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems, 2006, pp. 80–92.
[12] C. Zhao, J. Kong, K. Zhang, Design pattern evolution and verification using graph-transformation, in: Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS'07), 2007, pp. 290a.
[13] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino, M. Risi, Design pattern recovery by visual language parsing, in: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering, 2005, pp. 102–111.
[14] I. Philippow, D. Streitferdt, M. Riebisch, S. Naumann, An approach for reverse engineering of design patterns, Software System Model 4 (2005) 55–70.
[15] J. Kong, G. Song, J. Dong, Specifying behavioral semantics through graph-transformation, in: Proceedings of Workshop on Visual Modeling for Software Intensive Systems, 2005, pp. 51–58.
[16] D. Le Métayer, Describing software architecture styles using graph grammars, IEEE Transaction on Software Engineering 24 (5) (1998) 521–533.
[17] R. Bardohl, G. Taentzer, M. Minas, 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, Singapore, (1999) 105–180.
[18] A. Radermacher, Support for design patterns through graph-transformation Tools, in: Proceedings of International Workshop and Symposium on Applications of Graph-Transformation with Industrial Relevance (AGTIVE), 1999, pp. 111–126.
[19] T. Mens, S. Demeyer, D. Janssens, Formalizing behavior preserving program transformations, in: Proceedings of International Conference on Graph-transformation, 2002, pp. 286–301.
[20] F. Shull, W.L. Melo, V.R. Basili, An inductive method for discovering design patterns from object-oriented software systems, Technical Report, Computer Science Department, University of Maryland, 1996.
[21] G. Antoniol, G. Casazza, M.D. Penta, R. Fiutem, Object-oriented design pattern recovery, Journal of Systems and Software 59 (2) (2001) 181–196.
[22] R. Ferenc, A. Beszedes, L. Fulop, J. Lele, Design pattern mining enhanced by machine learning, in: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 295–304.
[23] N. Tsantails, A. Chatzigeorgiou, Design pattern detection using similarity scoring, IEEE Transaction on Software Engineering 32 (11) (2006) 896–909.
[24] S. Ciraci, P. Brook, Modeling software evolution using algebraic graph rewriting, in: Proceedings of Workshop on Architecture-Centric Evolution (ACE 2006), 2006.
[25] T. Kobayashi, M. Saaki, Software development based on software pattern evolution, in: Proceedings of the Sixth Asia–Pacific Software Engineering Conference, 1999, pp. 18–25.
[26] K. Ates, J.P. Kukluk, L.B. Holder, D. Cook, K. Zhang, Graph grammar induction on structural data for visual programming, in: Proceedings of IEEE International Conference on Tools with Artificial Intelligence (ICTAI), 2006, pp. 232–242.