# Framework for Computer-Aided Evolution of Object-Oriented Designs *

Selim Ciraci, Pim van den Broek, Mehmet Aksit
Software Engineering Group
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
PO Box 217
7500 AE Enschede
The Netherlands
{s.ciraci, pimvdb, aksit}@ewi.utwente.nl

## Abstract

*In this paper, we describe a framework for the computer-aided evolution of the designs of object-oriented software systems. Evolution mechanisms are software structures that prepare software for certain type of evolutions. The framework uses a database which holds the evolution mechanisms, modeled as template graph transformations, with the supported evolution types. To evolve the software, the designer enters the type of evolution and provides the names of the software entities that are going to be evolved. The framework fetches the evolution mechanisms, converts the design to a graph model and applies the transformations. As an application of the framework, we implemented a tool for computer-aided evolution that uses object-oriented evolution mechanisms.*

Keywords: Software Evolution, Object-Oriented Evolution, Design patterns, Graph Transformation.

## 1 Introduction

In order to stay on the market and meet users' demands, software systems have to evolve [12]. Anticipated changes (or evolutions) are changes to the software that are foreseen [4]. By using certain software structures (e.g. virtual methods and design patterns [8]) it is possible to prepare the software for anticipated evolutions (by prepare we mean the evolution can be implemented with minor modifications to the existing software entities). We call these software structures *evolution mechanisms*.

*This work has been carried out as a part of the DARWIN project at Philips Healthcare under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

Organizations usually use some evolution mechanisms in order to cope with anticipated evolutions. At the initial design these mechanisms are implemented in software entities where the anticipated evolutions are expected. Thus, when these evolutions happen, they can be implemented easily because the software is already prepared. Implementing the evolution mechanisms at the initial design can require extra effort; however, the assumption is that this extra effort pays off during the implementation of the anticipated changes.

Usually, the software evolution mechanisms are not well documented. Thus, in order to evolve the software, the designer may have to find the used evolution mechanism from the software artifacts (source code or UML models [3]), which requires unnecessary effort and reduces the efficiency of using evolution mechanisms. To address this problem, we have built a tool, that takes the types of evolutions the designer desires and the design of the software system as an input, then recognizes the evolution mechanism implemented and, finally, evolves the system by following the constraints of the evolution mechanism.

Our tool, which we call Computer-Aided Design Evolver (CDE), works in three steps: At the first step, the designer enters the types of evolutions she/he wants with the names of the software entities that are going to be evolved. We call these evolution types *evolution instructions* and they do not contain information about the evolution mechanism. At the second step, template graph transformations of the mechanisms that support these evolution instructions are extracted. These transformations are bound (instantiated) to the software entities by adding the names supplied in the evolution instructions. At the final step, the instantiated transformation rules are fed into a graph transformation tool together with the current design of the system. The graph transformation tool applies the transformation rules

and outputs the evolved version of the design. If the original design of the software system does not use an evolution mechanism that matches the evolution instruction, the design can not be evolved automatically. Thus, our system can also be used to check whether the design contains the evolution mechanisms required by the evolution instruction. In summary, the contribution of this paper is to provide a framework and a tool, establishing:

1. a formal basis for evolving the designs of the object-oriented software systems.

2. guidelines to developers on how to evolve the design with aid of computers.

In the literature, software (design) evolution based on patterns has been investigated ( [5], [11], [2]) and graph transformations are proposed in various studies ( [18], [16]). A limitation of the graph transformation based approaches is that they only provide evolutions of design patterns. Besides this, the designer has to supply the design pattern and the type of evolution on that design pattern she/he desires. In our framework, the designer uses evolution instructions that allow the designer to specify desired evolutions without the knowledge of the underlying evolution mechanism.

The remaining sections of the paper are organized as follows: Section 2 introduces the framework of the CDE tool. After giving an introduction to the framework, we apply it to object oriented systems. Section 3 describes how we model object oriented systems as graphs. Section 4 describes the evolution instructions, the object oriented evolution mechanisms and the conversion from instructions to mechanisms. An application of the tool in a simple object-oriented design is given in section 5. Section 6 details the software system model to graph-based model conversion. Section 7 presents some alternative approaches and finally section 8 gives the conclusions and the future work.

## 2 Computer-Aided Design Evolution Framework

The framework has two sets of activities; the activities involved in the preparation of the use the CDE tool and the activities executed by the tool. Below we list the activities that are involved in the preparation of the use for the CDE tool; some of these activities are executed manually and some by other tools (for example the software design to graph model converter):

1. Finding evolution mechanisms used by the organization.

2. Making a list that contains the evolution mechanisms, the type of evolutions each mechanism supports, with their constraints. For example, with method overriding it is possible to change the implementation of a method; however, if the class to be inherited from is *final* then we cannot use it.

3. Modeling the evolution mechanisms using a modeling language (e.g. UML).

4. Creating a meta-model describing how the software system can be modeled using graphs.

5. Creating the evolution instructions. The syntax of the evolution instructions is of the form *type_of_evolution (parameter1, parameter2,...)*. Here, the parameters are used to define the changing parts of the software and the type of evolution is used to describe the desired evolution. For example, *changeImplementation(sort, mergeSort)* is the evolution instruction for changing the implementation of *sort* to *mergeSort*. If method overriding is the evolution mechanism to apply this change, then *sort* should be a method.

6. Using the graph meta-model, the evolution mechanisms are modeled as graph transformation rules, which are used by the CDE tool to evolve the design. We call the graph transformation form of an evolution mechanism *evolution operator*. It is important to note here that the evolution operators are stored in a template form. That is, they do not include names of software entities. The reason for this is to increase the reusability of the framework: the same evolution mechanism and type can be used for different products of the organization.

7. Converting the software system design to the graph based model.

The inputs to the CDE tool are the evolution instructions with the names of the software entities that are going to be evolved, and the system design. Below we list the activities executed by the CDE tool:

1. Creating the evolution operators database. When the CDE tool is started with the parameter *-db <evolution instruction> <graph model file>*, it inserts the given instruction and graph model to the evolution operators database.

2. Fetching the evolution operators that match the instructions from the evolution operators database.

3. Instantiating the template graph transformation by adding the names of the software entities.

4. Preparing the graph rewriting system, which contains the evolution operators and the graph form of the design of the software system.

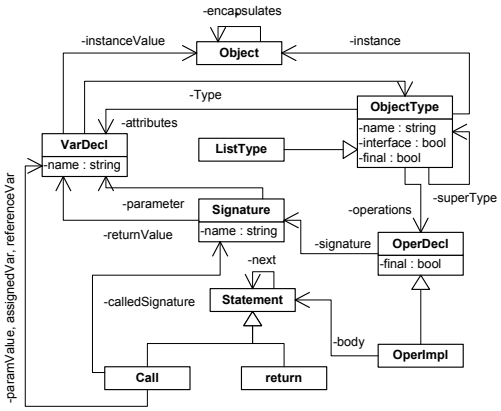


**Figure 1. The meta-model of the graph model used for modeling object oriented systems**

5. Executing the graph transformation system and delivering the evolved graph model and the guidelines to the designer/developer. We use GROOVE [17] as the graph transformation tool.

In this paper, we use the framework and the CDE tool for the computer aided evolution of object-oriented systems. Thus, the meta-model we describe shows how object-oriented systems can be modeled using graphs; this meta-model is detailed in the next section. Object-orientation has mechanisms, such as design patterns and inheritance, that can be used to evolve the software. We use these mechanisms as evolution mechanisms, thus the evolution instructions we present in this paper describe the type of evolutions these mechanisms support; this is detailed in section 4.

The steps involved in modeling the evolution mechanisms using graphs and converting the software design to the graph model can be automated. Because we apply the framework to object-oriented software systems, we have modified the open source UML editor ArgoUML [1] to export UML class and sequence diagrams to the graph-based model. Furthermore, the evolved design, output of the graph transformation tool, can be imported to ArgoUML to display the evolved UML diagrams. Thus, with these modifications to ArgoUML, we automate the preparation activity *7*. We describe the details of our modifications to ArgoUML in section 6.

## 3 Graph Model for Object Oriented Systems

The meta-model of our graph model for object-oriented systems is presented in Figure 1. Kastenberg et al. [10] provide a detailed graph model that captures all execution semantics of object-oriented systems. Our model is based on their model; however, we make certain abstractions. For example, our model does not include every statement in a method's implementation; we only include *Call* and *return* statements. The reasons for these abstractions are: 1) the model can be created from the design of the software and at design time not every detail about a software system is known 2) to model evolution and evolution mechanisms it is important to capture the communication between software entities (e.g. methods).

**Table 1. Evolution Mechanisms of Object Oriented Systems**

| Evolution Mechanism | Type of Evolution |
|---|---|
| Strategy Pattern | implementation change |
| Method Overriding | implementation change |
| Bridge Pattern | implementation change |
| Decorator Pattern | extending implementation |
| Method Overriding with super call | extend implementation |
| Visitor Pattern | extending interface |
| Adapter Pattern | changes in interface |

In our model, we use the same abbreviations used by Kastenberg et al. [10]. Thus, *Decl* stands for declaration, *Oper* stands for operation and *Var* stands for variable. The model has two parts; a static and a dynamic part. The static part represents the typing relations between object types (i.e. the classes and interfaces) and the attributes and the methods (operations) they have. The object-types are modeled with nodes labeled with *ObjectType*. The sub-typing relation between object-types are shown with edges labeled with *superType*. Abstract methods are represented by operation declaration nodes (nodes with label *OperDecl)*. The label *OperImpl* is used to model that an operation is implemented. A signature node is drawn for each distinct method signature in the system. Thus, to model a method overriding, both the overridden and overriding operation implementation nodes are connected to the same signature node.

The dynamic parts of the model show the call relations with instances of object-types. In order to find which method is being called, the model employs two edges. The first one, labeled *calledSignature*, identifies the called signature and is drawn from a call node to a signature node. The second edge, labeled *referenceVar*, is drawn from the call node to a variable declaration node to identify the type of receiver of the call. For example, for the call *a.foo()*, the called declaration edge is drawn to a signature node whose name is *foo* and has no parameters and the reference variable edge is drawn to a variable declaration node with name *a*. In case the call has to pass parameters, a *paramValue* edge is drawn from the call node to the variable declara-

tion node whose value is going to be passed. The object nodes represent the objects in the system. The connection from an object-type to an object (edges with label *instance*) describes whose instance the object is and the connection from a variable declaration node to an object (edges with label *instanceValue*), describes which instance the variable holds.

# 4 Evolution instructions and mechanisms for object oriented systems

## 4.1 Evolution mechanisms and types of evolutions they support

A prerequisite for the computer-aided design evolution framework is finding the evolution mechanisms and defining the types of evolutions they support. In this subsection, we provide an example for this step using the evolution mechanisms of object-oriented systems; we provide a sample list of mechanisms with the type of evolutions they support in Table 1.

## 4.2 Evolution Instructions

Once the evolution mechanisms and the types of evolutions they support are found, these types are used to formulate the evolution instructions. An important aspect of these instructions is parameterizing the names of the software entities.For example, the strategy pattern allows an implementation of a method to be changed. Thus, it works on two software entities, the name of the method whose implementation is going to change and the name of the implementation and these names should be passed to the mechanism.

In Table 2, we present the evolution instructions for the evolution types described in the previous subsection. These instructions follow a syntax similar to the function calls of C or similar languages. The major reason for this is to provide a familiar syntax to designers, so the language can be adopted easily. The parameters of the instructions do not enforce which software entities the evolution mechanisms are going to work with (e.g. methods or classes). With this notation the designer does not need to know the underlying evolution mechanism. It is the job of the CDE tool to find the implemented evolution mechanism and evolve the design accordingly. This allows us to overcome the problem of undocumented evolution mechanisms.

Table 2, is in fact the evolution operators database formed and used the by CDE tool (the real database contains a pointer to file or directory containing the graph transformation or transformations of the evolution mechanism). As can be seen from the table, some evolution instructions can be achieved with more than one evolution mechanism.

**Table 2. Evolution Operators Database**

| Instruction | Evolution Mechanism |
|---|---|
| changeImplementation (ofEntity, toNewImplementation) | Strategy Pattern. |
| changeImplementation (ofEntity, toNewImplementation) | Method Overriding. |
| changeImplementation (ofEntity, toNewImplementation) | Bridge Pattern. |
| extendImplementation (ofEntity, withImplementation) | Method Overriding with super call. |
| extendImplementation (ofEntity, withImplementation) | Decorator Pattern. |
| changeInterface (ofEntity, toNewInterface) | Adapter Pattern. |
| addInterface (ofEntity, newInterface) | Visitor Pattern. |

For such instructions, the CDE tool forms the graph production system with the graph transformations (i.e. the graph model) of each mechanism. The transformations only match when the initial design includes the evolution mechanism. Thus, if there is no match, then the designer can conclude that none of the evolution mechanisms are in the design. As a result, our framework can also be used at the initial design by the designer to evaluate whether the evolution mechanisms for the anticipated changes are in the design or not.

## 4.3 Evolution Mechanisms as template graph transformations

The evolution operators (the graph models of the evolution mechanisms) should not include the names of the software entities they work on for them to be general (i.e. applicable on more then one place). To achieve this, the evolution operators are modeled by marking the names of the entities with a special string that signals the CDE tool to replace this string with the name of the software entity. We call these models with special strings *template graph transformations*. In Figure 2-(a), we present the template graph transformation for the strategy pattern. A graph transformation rule in GROOVE is represented as follows: The thick (green) nodes and edges represent the graph elements that are going be added to the graph. The dashed (blue) nodes and edges represent the graph elements that are going to be deleted from the graph. All other edges and nodes are required to occur in the graph to say that this rule has an occurrence in the source graph (i.e. the rule matches). When a rule matches to a graph, the graph elements that are marked to be deleted in the transformation rule are removed and the graph elements that are marked to be added are added to

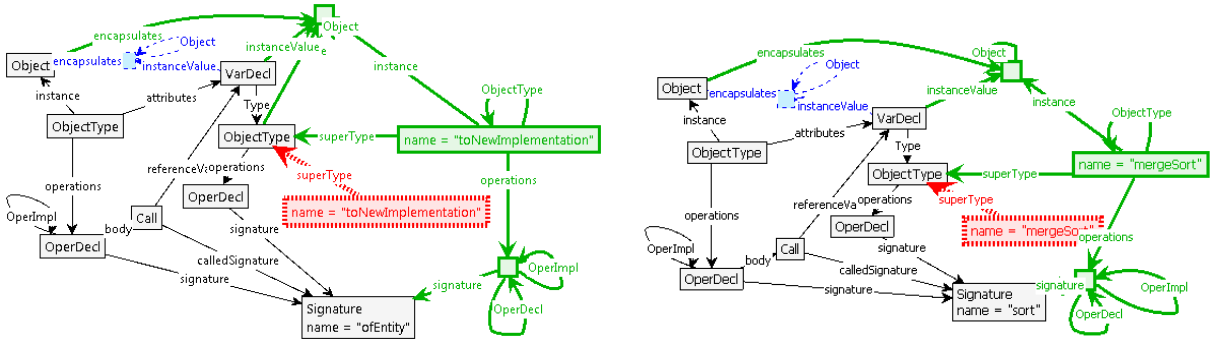


**Figure 2. (a) Template graph model of the strategy pattern; the names of software entities are parameterized. (b) Initialized graph model for the strategy pattern for the evolution instruction** *changeImplementation(sort, mergeSort)***.**

the graph. The thick (red) dashed nodes are the negative application conditions [9]; their presence in the graph prevent the rule from matching. Here, we only describe how a graph transformation rule works in GROOVE, interested readers on the subject of graph transformations are referred to the literature [7].

The strategy pattern is used to implement the evolution instruction *changeImplementation* (Table 2). In Figure 2-(a), the name of the method whose implementation is going to change is labeled with *"name=ofEntity"*. This string signals the CDE tool to replace it with the value of the parameter *ofEntity* of the evolution instruction *changeImplementation*. We call the process of replacing the signal string with the values of the evolution instructions *instantiation*. In Figure 2-(b), we present the instantiated graph model of the strategy pattern for the evolution instruction *changeImplementation(sort, mergesort)*. When this rule is applied to the design that uses the strategy pattern, an object-type named *mergeSort* that implements the method *sort* is added.

Some evolution mechanisms may not be modeled with one graph transformation. In such cases, all the graph transformation rules modeling the evolution mechanism is put into a directory and this directory is passed to the CDE tool. For example, modeling method override mechanisms requires an iteration over the sub-types of the object type declaring the method that is going to be overridden. We modeled this evolution mechanism with 3 graph transformation rules: the first one identifies the object type declaring the method, the second one iterates over the sub-types and the third one adds the new object-type and the overriding method.

## 5 Example

Assume we have the design of the software system whose class diagram is given in Figure 3-(a) and whose sequence diagram is given in Figure 3-(b). Also assume that we want to change the implementation of *sort* to *mergeSort*. We first convert this design to a graph model using ArgoUML; we present the graph model of the class *quickSort* in Figure 3-(b). Then, the evolution instruction *changeImplementation(sort,mergeSort)* is passed to the CDE tool (in a text file). The CDE tool fetches the evolution operators (the template graph transformations in GXL format) for all evolution mechanisms that support this evolution from the evolution operators database (Table 2). The CDE tool, then, adds the names *sort* and *mergeSort* to the appropriate places in the template rules and forms the graph rewriting system. Lastly, the CDE tool launches GROOVE to execute the graph rewriting system.

It can be seen from the evolution database presented in Table 2 that 3 evolution operators support the evolution instruction *changeImplementation*. The graph rewriting system created by the CDE tool contains 5 graph transformation rules; 3 for method override, 1 for strategy and 1 for bridge. Figure 4 shows GROOVE with the graph-based model of the example system design, the graph rewriting system with the 5 graph transformation rules and the matches of each transformation rule (shown with text *Match* under the transformation rule). Since the example design does not contain the bridge pattern, the transformation rule for the bridge pattern does not have a match. However, the transformation rules *methodOverrideMark* and *strategy* have matches. Clicking on each match applies the transformation rule.

For this example, the transformation rule method override matches to 2 different locations in the design be-

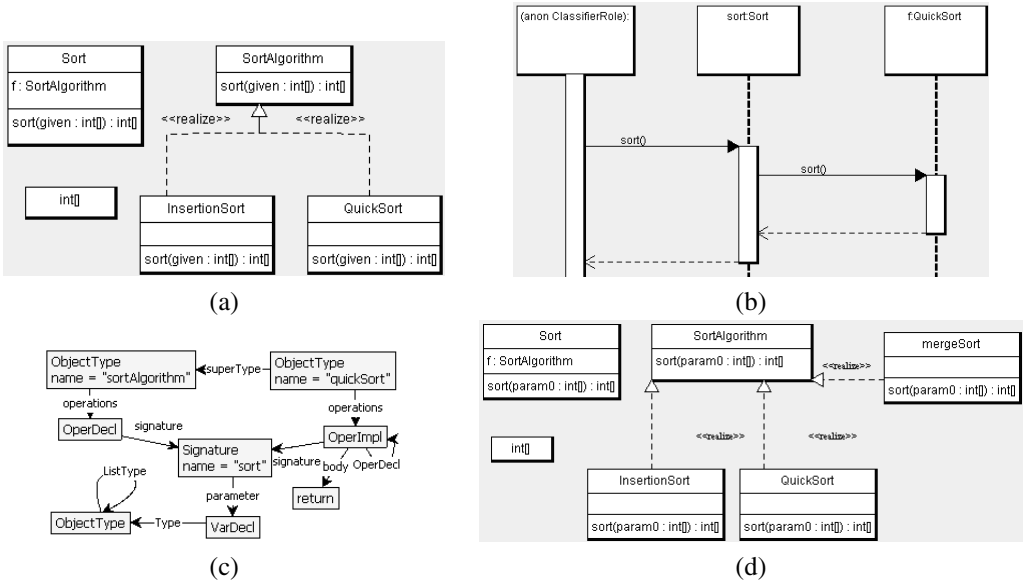


**Figure 3. a) A sample system design that uses strategy pattern b) The sequence diagram explaining the call relations c) the graph model for the class *quickSort*, d) the class diagram resulting from the evolution instruction *changeImplementation(sort,mergeSort)***

cause there are two object-types that declared the method *sort*. Clicking on one of these matches marks one of these types as candidates for sub-typing. After this rule, the rule *methodOverrideSubType* matches which adds a object-type with name *mergeSort*. The transformation rule *methodOverrideIterate* matches when the object-type *SortAlgorithm* is marked as a candidate for sub-typing and this rule marks one of the sub-types of this type as a candidate (which can then be sub-typed with the rule *methodOverrideSubType*). The transformation rule strategy matches to 1 location and when clicked this rule simply adds the object-type *mergeSort* as a sub-type of object-type *SortAlgorithm*. Using GROOVE, the evolved graph-model can be saved, which can then be imported to ArgoUML to display the evolved class diagram. Figure 3-(d), shows the class diagram resulting from the application the transformation rule modeling strategy pattern.

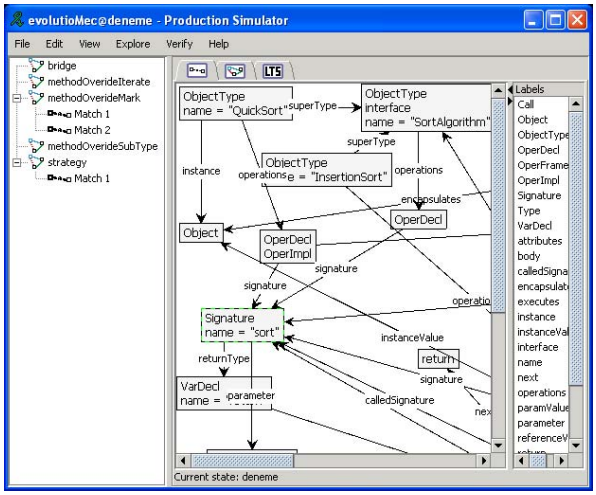


**Figure 4. GROOVE tool displaying the transformation rules for evolution instruction *changeImplementation(sort,mergeSort)*.**

## 6 Software design to graph model conversion: modifications to ArgoUML

As discussed before, the graph model consists of two parts: the static and the dynamic part. The information required in the static part of the model is located in UML class diagrams and the information for the dynamic parts is located in UML sequence or collaboration diagrams. Thus, class diagrams and sequence or collaborations diagrams are required to generate the graph model of the software system (or an execution scenario of the system). In this section, we detail how the conversion from UML class and sequence diagrams can be made.

Our model only supports communication between objects when one of the objects has a pointer to the other object. Thus, the *Association*, *Aggregation* and *Composite* relations between the classes in the class diagrams are

converted to attributes. For example, an association relation from class *A* to class *B* is converted to an attribute of class *A* with type *B*; we use name of the relation (i.e. the name of the association arrow or the name of the end with the arrow head) as the name of the attribute. To be able to export a call action, the sequence diagram should supply the name of the reference variable. This can be either supplied as the name of the message or the name of the call action; the exporter supports both notations. We create an instance for each new object encountered in the sequence diagram (this also applies for the parameters of methods). Object representations (i.e. the UML classifiers) with the same name and type in the sequence diagram refer to the same object; we follow this rule also in the exportation process and create one object node for each such object.

We have extended the ArgoUML [1] UML modeling tool, to save the class and sequence diagrams as graph-based models and to create UML class diagrams from the graph-based models. The main motivation in choosing ArgoUML is that it supports sequence diagram editing and loading *XMI* files with sequence diagrams. To create the graph model, the user has to input the names of the class diagram and the sequence diagram that are going to be used. Once these names are entered, the exporter first exports the class diagram and then exports the sequence diagram. During the exporting process, errors in the model are also checked (for example, if the number of parameters passed to a method in a call action does not match, the export is not completed). To create the UML class diagram showing the evolved design (i.e. after the evolution mechanisms are applied), the designer selects the file containing the evolved design in graph form and enters the name of the class diagram where the model will be drawn. Currently, sequence diagram creation from the graph-based model is not implemented,this is left as a future work.

## 7 Related Work

In our previous study [6], we have shown how evolution operations can be formalized using graph rewriting; however, in that study we have not shown how these formalism can be used. In this paper, we use graph transformations to evolve the designs of object-oriented systems using evolution mechanisms.

Kobayasahi et al. [11] define steps and languages for instantiating and evolving a template design pattern. These steps are similar to ours; however, our evolution mechanisms are not limited to design patterns and our evolution operations are formalized using graph transformations.

Zhao et al. [18] present how design patterns can be evolved correctly using graph transformations. They have a similar approach to ours, in which the designer enters the desired evolution and the tools evolve the design. Here, the desired evolutions are expressed only in terms of the design patterns. Our evolution instructions describe the change type without the underlying design pattern; that is, for us the design pattern is a way to achieve the evolution. Also, our framework is not limited to design patterns. Moreover, we use graph transformations for finding the evolution mechanisms rather then using a linear search algorithm.

Mens et al. [13] model refactorings as graph transformation rules and use critical pair analysis to detect the dependencies between refactorings. Refactorings are behavior preserving changes that are made to improve the structure of the software. The changes that we consider in this paper are usually changes made to add functionality to the software system and we formalize these changes by means of evolution mechanisms such as design patterns. The CDE tool at its current state rejects the evolution when evolution instructions contain different types of evolution applied to the same software entity. This is because such an evolution can lead to an inconsistent design.

During evolution, the models of the software system can become inconsistent. Mens et al. [14] use graph transformations to detect and to resolve such inconsistencies in UML diagrams. They define a graph meta-model for object-oriented systems and model inconsistencies as graph transformations. The graph transformation rules are used to search for the occurrence of the inconsistencies in the design. Their inconsistency search methodology is similar to our evolution mechanism search. However, our aim is finding the evolution mechanism and evolving the design.

Niere et al. [15] use graph transformation rules to detect and recover the design patterns from source-code semi-automatically. Their transformation rules look for the design pattern and add graph elements that mark the recovered design pattern. Because design patterns can have variations, they define sub-patterns for variations and design patterns are described in terms of these sub-patterns. To handle variations in evolution mechanisms, our framework consists of steps in which the evolution mechanisms are extracted. This way, the variations can be found and modeled. Also, the approach presented by Niere et al. [15] can be combined with our approach to handle variations.

## 8 Conclusion and Future Work

In this paper, we described a framework for computer-aided evolution of software designs. The main aim of this framework is to overcome the limited documentation problem for evolution mechanisms. Evolution mechanisms are software structures that make certain evolutions easier. However, these mechanisms should be carefully documented; otherwise the designers need to recover the used evolution mechanism from the design model or the source code, which can be time consuming. This reduces the effect

of using the evolution mechanism.

The proposed framework has both manual and automatic steps. The manual steps involve finding the evolution mechanisms, the type of evolution they support and modeling them using the graph model we present in this paper. We have created a list of evolution mechanisms with the type of evolution they support and presented in Table 2. Using the type of evolutions the mechanisms support, we created the evolution instruction language. In this language, the designers need to specify the desired evolution type and the names of the software entities that are going to be evolved. Here, the designer only needs to know that the names she/he inputs are in the design; however, the designer is not bothered with the details of the used evolution mechanism or to which software entities these names are mapped to. The graph models of the evolution mechanisms are modeled in template forms. These graph models are put in to a database with the type of evolution they support. We have developed the Computer-Aided Design Evolver (CDE) tool that gets the evolution instructions in the form we present in this paper, parses the instructions and according to the names in the instructions it instantiates the template graph transformation rules and executes the graph transformation tool GROOVE, which then applies the graph transformations and prints out guidelines for the designers.

The evolution mechanisms and the language can easily be extended or adapted to the needs of organizations. For certain evolutions, more then one evolution mechanism can be applicable or more then one possible match can be found. Currently, we only display the guidelines for each possible evolution path. However, we can use some heuristics to eliminate the possible evolution paths. This is another point that we plan to work out. We are also going to extend the UML to graph model converter so that evolution mechanisms can be modeled in UML and then converted to template graph transformation rules.

## References


[1] Argouml. http://argouml.tigris.org.

[2] M. Aoyama. Evolutionary patterns of design and design patterns. *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 110–116, 2000.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Professional, September 1998.

[4] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17(5):309–332, 2005.

[5] M. O. Cinnéide and P. Nixon. Automated software evolution towards design patterns. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 162–165, New York, NY, USA, 2001. ACM.

[6] S. Ciraci and P. van den Broek. Modeling software evolution using algebraic graph rewriting. In *Workshop on Architecture-Centric Evolution (ACE 2006)*, ACE, Nantes, France, Jul 2006. ECCOP.

[7] H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Spinger-Verlag, 1979.

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

[9] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundam. Inf.*, 26(3-4):287–313, 1996.

[10] H. Kastenberg, A. G. Kleppe, and A. Rensink. Defining object-oriented execution semantics using graph transformations. In *Proceedings of the 8th IFIP, Bologna, Italy*, volume 4037 of *LNCS*, pages 186–201, London, 2006. Springer-Verlag.

[11] T. Kobayashi and M. Saeki. Software development based on software pattern evolution. *Software Engineering Conference, 1999. (APSEC '99) Proceedings. Sixth Asia Pacific*, pages 18–25, 1999.

[12] M. M. Lehman, D. E. Perry, J. C. F. Ramil, W. M. Turski, and P. Wernick. Metrics and laws of software evolution. In *Fourth International Symposium on Software Metrics*, pages 20–32, November 1997.

[13] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, Sept. 2007.

[14] T. Mens, R. van der Straeten, and M. DŠHondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In *Model Driven Engineering Languages and Systems*, volume 4199/2006, pages 200–214, Washington, DC, USA, 2006. Springer-Berlin.

[15] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 338–348, New York, NY, USA, 2002. ACM.

[16] A. Radermacher. Support for design patterns through graph transformation tools. In *AGTIVE '99: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 111–126, London, UK, 2000. Springer-Verlag.

[17] A. Rensink. The groove simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance*, pages 479–485. Springer Verlag, 2004.

[18] C. Zhao, J. Kong, J. Dong, and K. Zhang. Pattern-based design evolution using graph transformation. *J. Vis. Lang. Comput.*, 18(4):378–398, 2007.