Modelling Software Evolution using Algebraic Graph Rewriting *

Selim Ciraci and Pim van den Broek

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}@ewi.utwente.nl

Abstract. We show how evolution requests can be formalized using algebraic graph rewriting. In particular, we present a way to convert the UML class diagrams to colored graphs. Since changes in software may effect the relation between the methods of classes, our colored graph representation also employs the relations in UML interaction diagrams. Then, we provide a set of algebraic graph rewrite rules that formalizes the changes that may be caused by an evolution request, using the pushout construction in the category of marked colored graphs.

Keywords: Software evolvability, Software evolution, Evolution modelling, graph rewriting.

1 Introduction

Studies have shown that maintenance and evolution are the longest and most expensive phases in the software life-cycle [1]. From these phases, evolution has started to receive the greatest attention, due to the marketing demands and fast technological improvements of recent years. For software systems to continue to be effective and to compete with similar systems on the market, they should include new requirements or change the present ones. These requirements modifications have an impact on the overall software system, and for organizations it is important to know this impact without implementing the changes. There are many studies that try to capture the scope of evolution. For example, Bennett and Rajlich [1] state that software evolution occurs only after the initial software system is developed. We consider evolution as a procedure for integrating new requirements to the software system. Thus, evolution does not occur only after

* This work has been carried out as a part of the DARWIN project under the responsibilities of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.

the initial system is developed, since user requirements may also change during the development of the initial system.

The term evolution first appeared in the software engineering literature in a study based on the observations made on the source code of 20 releases of the OS/360 operating system [2]. These observations have shown an increasing trend in the complexity of the overall system. For example, nearly in all releases new modules were added to the system. From then, most of the research on software evolution is focused on analyzing the properties of evolution by conducting empirical analysis on the source code of releases of software systems [3], [4]. Although such an analysis may help in estimation of the growth or the cost of an evolution request, we believe that evolution still lacks a formal background, rules or a model. Such a model may ease realization of evolution requests by depicting the impact of the request by basing the request on certain rules.

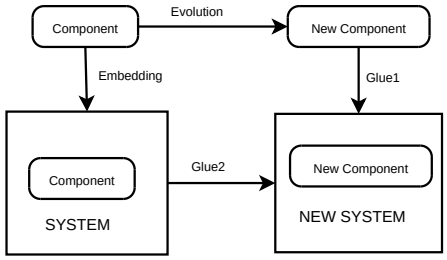

Fig. 1. Evolution model

In this paper, we provide a model for software evolution that is based on algebraic graph rewriting. Our idea is that software architecture can be modelled as a colored graph and the evolution requests can be viewed as morphisms on the components of the software system (e.g. classes), as can be seen in Figure 1. Although UML class and interaction diagrams may also be used; using only one model that combines the information given in these diagrams makes it easier to adopt a model based on algebraic graph re-writting. For software architectures, we follow the Unified Process [5] and use the UML class and interaction diagrams because the process has defined output models and is a widely adopted standard. Medvidovic and et al. [6] also states that UML "as is" is suitable as an architecture description language. A similar approach is taken by Alanen and Porres [7] to combine the changes made on class diagrams of software systems by different developers; similar to a version control system. The idea of using algebraic graph rewriting to model software evolution is also used by Wermelinger and Fiadeiro [8] and Mens, Eetvelde, Demeyer and Janssens [9]. The main difference of our between the studies and our work is that we use marked graphs and we build the evolution model at architecture level from UML class and interaction diagrams.

The paper is organized as follows. In the next section brief information about algebraic graph rewriting can be found. Then in section 3, we present our evolution model; first the model of object oriented software system is presented and then the evolution model build on the software system model is presented. Finally, we conclude our discussion and present the future work in section 4.

2 Background on Graph Rewriting

In this section we present a brief summary on algebraic graph rewriting; detailed information on this topic can be found in [10] and [11]. The main idea of the algebraic approach to graph rewriting is to give an abstract algebraic characterization, using the pushout construction in the category of colored graphs [10]. A colored graph, for example G, is represented by a 6-tuple as $G = \{N_G, A_G, s_G, t_G,$
$m_{G,1}, m_{G,2}\}$. Here, N_G denotes the set of nodes, A_G denotes the set of edges; s_G is a mapping that maps the edges to their sources and t_G maps them to their targets. $m_{G,1}$ and $m_{G,2}$ are mappings that map the nodes and arcs in the graph to the fixed alphabets of node and edge colors. Then a graph morphism f for a given pair of graphs G and G', which is presented as $f : G \to G'$ or f or $G \to G'$, is a pair of maps $f_N : N_G \mapsto N_{G'}$ and $f_A : A_G \mapsto A_{G'}$ that preserves sources, targets and colors. That is:

$$f_N.s_G = s_{G'}.f_A, \quad f_N.t_G = t_{G'}.f_A$$
$$m_{G,1} = m_{G',1}.f_N, m_{G,2} = m_{G',2}.f_N$$

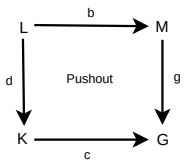

Fig. 2. Pushout of d and b

A pushout is depicted in Figure 2, where the input consists of morphisms $d : L \to K$ and $b : L \to M$ and the output consists of $c : K \to G$, $g : M \to G$ such that the diagram commutes. Here, the morphism b describes which rewriting should be done and d identifies the occurrence of the part of the initial graph (L) that should be rewritten. For definition of categories, commuting diagrams and pushouts the reader is referred to literature [12], [13]. Here, we only present the

construction of the pushout in the category of colored graphs. Given that d and b are morphisms, the pushout is constructed by applying the gluing procedure as follows:

1. Form $M+K$, which is the colored graph that consists of the two components M and K.
2. In $M + K$, identify (glue together) the nodes $b_N x$ and $d_N x$ for all nodes $x \in N_L$, and identify the edge $b_A x$ and $d_A x$ for all edges $x \in A_L$.

The gluing construction given above shows that only items (nodes and edges) can be added to graph G; however, in colored graphs items can be removed. Ehrig [10] provides a solution to the problem of item removal, which uses double pushouts. A simpler approach is provided by Van den Broek [11] that uses single pushouts in the category of marked colored graphs. The idea behind this approach is that the items that are going to be removed are marked. So the graph contains two types of items; marked and not marked. Although, the marked items stay in the graph, they are considered as garbage. A marked graph is an 8-tuple, $G = \{N_G, M_G, A_G, B_G, s_G, t_G, m_{G,1}, m_{G,2}\}$. In a marked graph the sets of nodes and edges are subdivided into sets of marked and non-marked nodes and edges. Thus the only difference between a colored graph representation and marked graph representation is the two sets, M_G and B_G which denote the set of marked nodes and edges respectively. For a given two marked graphs G and G', a morphism, $f : G \to G'$, is a pair of maps $f_N : N_G \bigcup M_G \mapsto N_{G'} \bigcup M_{G'}$ and $f_A : A_G \bigcup B_G \mapsto A_{G'} \bigcup B_{G'}$, which preserves sources, edges and colors like morphisms for colored graphs and, in addition, map marked items onto marked items:

$$f_N(M_G) \subseteq M_{G'},$$
$$f_A(B_G) \subseteq B_{G'}$$

In the category of marked graphs, pushouts are constructed as follows: first the pushout of colored graphs is constructed by using the gluing procedure described above; all markings are ignored. Subsequently, those items are marked onto which marked items are mapped. This pushout always exists [11]. However, at the end the marked items are deleted which may cause dangling arcs and for such cases there is no rewriting.

We revisit the example presented by Alanen and Porres [7] to show an example of what we have described in this section. In this example, the design of a software system that consists of classes A, B (B extends A) and C is modified by the two designers, which is depicted in Figure 3a. The first designer deletes class B and the second one adds class D, which extends class C. The final model is the combination of these two models that tries to capture the changes made by two designers; that is the changes made by both designers are glued on the original design to form the final design. In this figure, class B and the inheritance relation

to class B is marked. So they are treated as garbage and they can be removed from the system. For this example, it is possible to remove marked items since they do not result in dangling arcs.

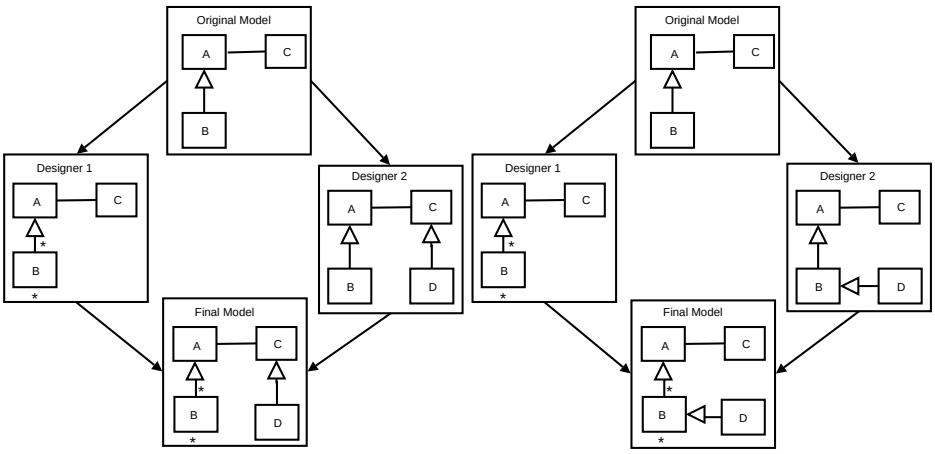

Fig. 3. a) The example presented by Alanen and Porres [7], here marked items can be discarded. b) The same example with a slight modification, here marked items can not be discarded.

Alanen and Porres [7] state that element deletion may result in conflicts. In cases, for example, where the same element is deleted by the two designers, a conflict emerges. Though, such a conflict can be resolved by deleting the equivalent delete operations from one of the designers model during the construction. In our model, this situation does not lead to a conflict. The conflict that occurs when a designer modifies an element and the other designer deletes it, can not be resolved. In Figure 3b, we give an example of such a conflict. Here, again the first designer deletes class B. However, the second designer adds class D, which extends class B. In the final model, it can be seen that if marked items are removed from the system we would end up with a structure that is not a graph; the edge from class D would become dangling. As discussed earlier, for such cases there is no rewriting, due to conflicting changes.

3 Application of Graph Rewriting to Evolution

We present the model of evolution and the model of the object-oriented system that we have built the evolution model on. We continue this section by describing how the evolution requests can be formalized. Although class diagrams can also be used with our model, with these diagrams some of the impacts may

not be captured, because they model only the static structure of a system. For example, assume we have a system whose class diagram is shown in Figure 4. Furthermore, assume that an evolution request requires extensions to the method *getDescription()* in class *FooWorker*. A solution to realize this evolution request may require to add another parameter to this method. However, the exact impact of this change is not clear, since it is not clear from the class diagram which method of class *Driver* calls the method *FooWorker.getDescription()*. Such details may be captured by using UML interaction diagrams. Thus, to capture the impact of changes, one needs to employ both class and interaction diagrams. However, rather then employing two different models, we propose to employ only one model that is similar to class diagrams but also includes the relations from interaction diagrams.

3.1 Object-Oriented System Model

We present a model for object-oriented software that uses marked graph representations in this subsection. This model can be viewed as a class diagram with the relations from collaboration and/or sequence diagrams.

As discussed in section 2, a marked graph has a pair of color alphabets, one to color the edges and one to color the nodes. To model a piece of software as a marked graph, the color alphabets play an important role in describing the relationship between components. In our model of object-oriented software the components, the elements of the node color alphabet (C_E), are the following:

- Class
- Attribute:$\ll Type \gg$
- Method
- Parameter:$\ll Type \gg$
- Return value:$\ll Type \gg$

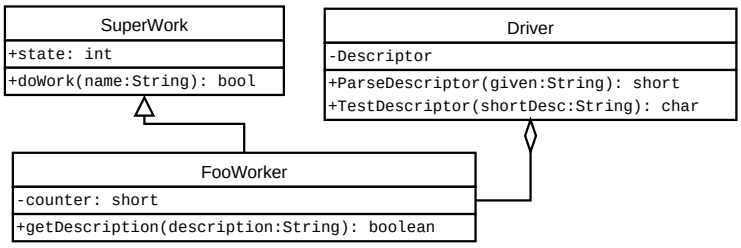

Fig. 4. An example of a class diagram

In the color alphabet, we do not include names of the components because graphs represent the structure of the system. However, without names it would

be hard to identify the components. As a result, in our model we use names to identify components. The edges, in this model, describe the relationship between components, which can be classified into three classes. The first class of relationships depicts which attributes and methods belong to a class. An edge that connects a class to an attribute is colored with *Has Attribute* and an edge that connects it to a method is colored with *Has Method*. To capture encapsulation, we extend these colors to include private and public declarations. So in the color alphabet, we have *Has Public Attribute*, *Has Private Attribute*, *Has Public Method* and *Has Private Method* colors. It is also important for this model to show the parameters and the return values that methods take or return; thus the model also employs the colors *Takes Parameter* and *Returns*.

The second class of relationships shows the connection between classes. For this class, we use the same connections that are used in class diagrams. However, for simplicity, in this paper, we are only including *Association*, *Aggregation*, *Generalization* and *Composition* into the color alphabet of edges. We use the same direction for edges as the directions for relations in UML class diagrams. Changes may also effect the overridden methods, so our model also employs an edge colored as *Overrides* which connects the overridden method to the method that overrides it.

The third class of relationships captures the object relations that are extracted from UML interaction diagrams, specifically UML sequence diagrams. This type of relationship is important to include in our model because they specify which methods and parameters are effected by the changes caused by evolution requests. It is possible to extract the relations between the methods of objects by using UML sequence diagrams. As example, for the class diagram given in Figure 4 let's say that the *Driver.TestDescriptor()* method calls the *FooWorker.getDescription()* method. In the sequence diagram this is represented with two arcs; the first one connecting the user of the class *Driver* to an instance of *Driver* class labelled with *TestDescriptor()* and the second one connecting an instance of *Driver* to an instance of *FooWorker* labelled as *getDescriptor(desc)*. Such relations, in our model, are captured with edges that connect a method to another method colored as *Calls*. To show that the calling method is passing parameters to the method that is being called, our model uses edges that connect the calling method to the parameter(s) of the method that is being called, which are colored as *Passes*. If, for example, the method being called does not take any parameters than these edges are not drawn. An evolution request may cause the return value for a method to change, so methods that depend on this method may also need changing. In our model, we make the relationship between a method and a return value explicit with an edge colored with *Gets Return Value* that connects a return value node to a method node.

In Figure 5, we redraw the class diagram shown in Figure 4 using the model presented in this section. In this figure, it is clearly seen that the method *Driver.TestDescriptor* calls the *FooWorker.getDescription* method since there is an edge that connects the node *Method TestDescriptor* to the node *Parameter:String Description*. Thus with this presentation, it is easy to argue that

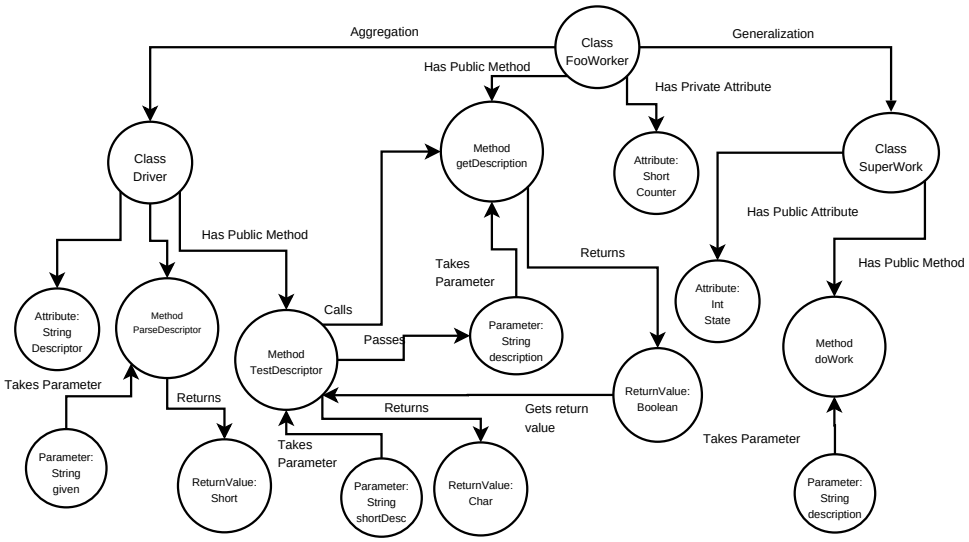

Fig. 5. Class diagram of Figure 4 redrawn with the new model; the method calls are explicit with this model

adding a parameter to the method *FooWorker.getDescription* is going to have an impact on *Driver.TestDescriptor*; some extra-lines are going to be added to handle the new parameter. This figure also shows that the method *Driver.TestDescriptor* makes use of the value returned by the method *FooWorker.getDescription* since there is an edge that connects the return value to the method *TestDescriptor*. In summary, with this model we are able to capture the relations between software components without employing two different models. In the next subsection we build a formal model for evolution using pushouts. The examples shown in the next subsection use the model shown in Figure 5.

3.2 Formalizing Evolution Operations

In this section of the paper, we present how some of the evolution requests can be modelled using pushouts as shown in Figure 1. Our main focus here is on addition and removal operations since we believe that by combining these operators most of the changes in a system can be modelled (e.g., a type change can be modelled by removing the component with old type and adding the component with the new type). Specifically, we focus on evolution requests that cause addition or removal of components in the system. Though, using the model we present here, it also possible to formalize evolution requests that cause the relation between components to change. It is important to note here that to realize removal of components using a single pushout, our model makes use of marked graphs [11].

We categorize addition and removal of software components into three levels, which are parameter and return value level, method and attribute level and class level. For each level, we describe the pushouts that formalize the changes and we conclude our discussion in this subsection, by showing how the pushouts at different levels can be combined to model an evolution request; since an evolution request may require changes at these three levels. For example, an evolution that requires removal of a method, also requires the parameters and the return values of that method to be removed. In our model, such an evolution can be modelled by using the pushout that removes a parameter from a method for each parameter the method has, and then the pushout that removes the method.

In Figure 1 we show the abstract model of software evolution. In the next sections, we are going to generalize this diagram to show evolution on different software components. For each pushout described in the next sections, we have four graphs; which are:

- Component: contains the software components which are going to evolve.
- New Component: contains the components after the evolution.
- System: shows the system that contains the original components
- New System: shows the system containing the evolved components

For each pushout, we also have four morphisms labelled as *Embedding*, *Evolution*, *Glue1* and *Glue2*.

Evolution of Parameters Evolution requests may cause changes in parameters in various ways; however, as discussed earlier we are going to model addition and removal of a parameter as pushouts. These pushouts then can be combined to realize other changes in parameters. The pushouts presented in this section, depict the changes between two methods; in other words the *Component* of Figure 1 is made up of two methods, in which one calls the other one and the one being called is going to evolve. In many cases more than one method call the method that is going to evolve; for such cases these methods can also be included in the *Component* graph.

First, we describe how addition of a new parameter can be modelled with a pushout as shown Figure 6.The system graph shows that, with *Embedding* morphism, the methods are a part of the system; in the figure it is clearly seen that the methods and relation between methods are preserved. This figure also shows the importance using names to identify the components, since without the names it would be impossible recognize which methods in the *Component* graph are mapped to which methods in the *System* graph. Then, it can be seen that the evolution morphism occurs which adds a parameter to the method that is being called. Adding this parameter also causes two edges to be added to the component; the *takes parameter* edge and the *passes* edge. *Evolution* is also a morphism since again the methods and the relation between methods are preserved. The pushout construction glues the new parameter to the system and the cost of such a glue operation is proportional to adding the new parameter to all methods that call this method.

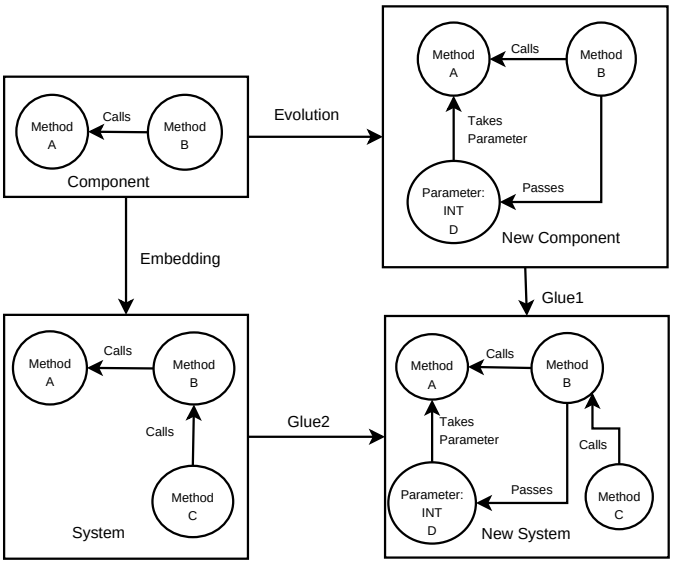

Fig. 6. Addition of a new parameter to an existing method.

In Figure 7, we show the pushout that formalizes the removal of parameter from a method. In *New Component*, the edge that connects the calling method to the parameter of the method that is being called is marked, which means the calling method should not pass this parameter. As it can be seen, the only difference between *Component* and *New Component* is that the parameter and the edges colored with *Passes* and *Takes Parameter* are marked.

As presented in section 2, the last step in a marked graph construction consists of removing the marked nodes and edges from the construction, if removing them does not cause any dangling arcs. From Figure 7, it can be easily seen that removing marked items does not cause any dangling items, so these items can be removed; thus the operation has succeeded.

Evolution requests may also require the type of parameters to be changed. Though we do not have a special pushout that formalizes such a change. This is because a type change can be handled by first applying the parameter removal pushout to remove the parameter with the old type and then with the parameter addition pushout the new parameter can be glued to the system.

Evolution of Methods Here, we show how addition or removal of a method can be modelled using pushouts. For addition, presented in Figure 8, the *Component* graph includes a class, to which the new method is going to be added to, and a method, which is going to call the new method. When there is going to be more than one method calling the new method, these methods can also be added in the *Component* graph to show that they are also going to be effected by the addition

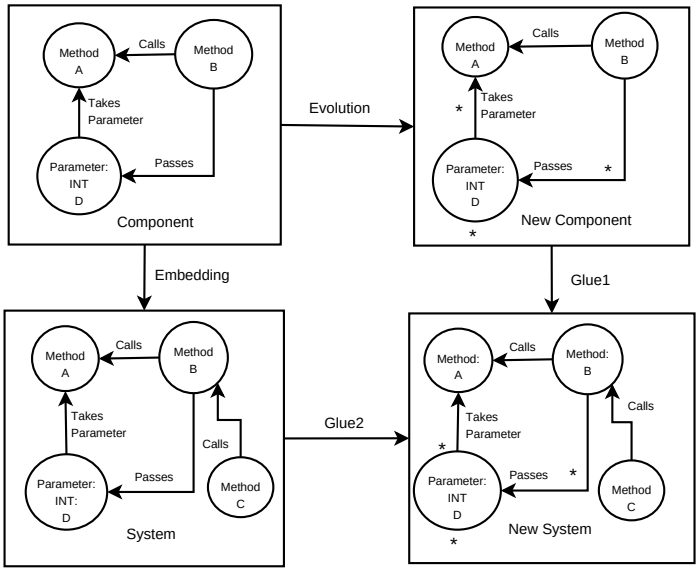

Fig. 7. Removal of a parameter from a method.

or the same pushout can be repeated until all these methods are covered. The figure shows that with the *Embedding* morphism the items in *Component* are a part of the *System*. With *Evolution* a new method is added to the *Component* and with gluing constructions this method is included in the *System* so the it evolves to *New System*. Here, we show the addition of public method (hence *New Component* includes an edge colored as *Has Public Method*; however, the same pushout can also be used to add a private method.

Adding methods are formalized using pushouts that are similar to the pushouts presented for addition of parameters. From this pushout, it can be said that the cost of this operation is proportional to adding the code that calls the new method. If this new method has a return value and/or parameters, then the pushouts presented for adding these can be used after using the pushout that adds the method. Furthermore, since the cost of adding a parameter or a return value is proportional to adding it to all methods we can say that the cost of adding the method is proportional to adding the code that calls this method and handles its parameters and return values. Although without pushouts it is possible to calculate the cost of adding a method to the system with intuition, pushouts have helped in formalizing the addition.

In Figure 9, we show how method removal can be formalized. The figure shows that method B calls the method A of class F. The evolution request requires removal of method A and, in turn, this requires removal of all the edges that connects this method to other methods and its class. So with the *Evolution* morphism these edges and the method node are marked. Then with

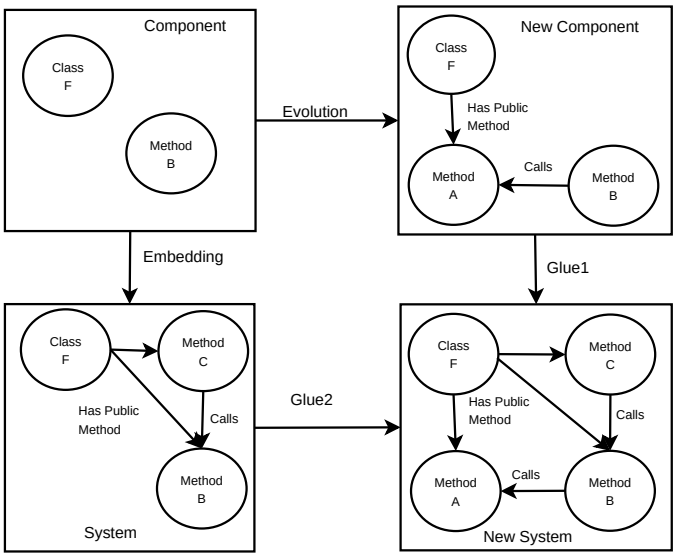

Fig. 8. The pushout that shows addition of a method. In order not to complicate the diagram, the *has public method* color for the edges between the class F and methods C and D is not shown

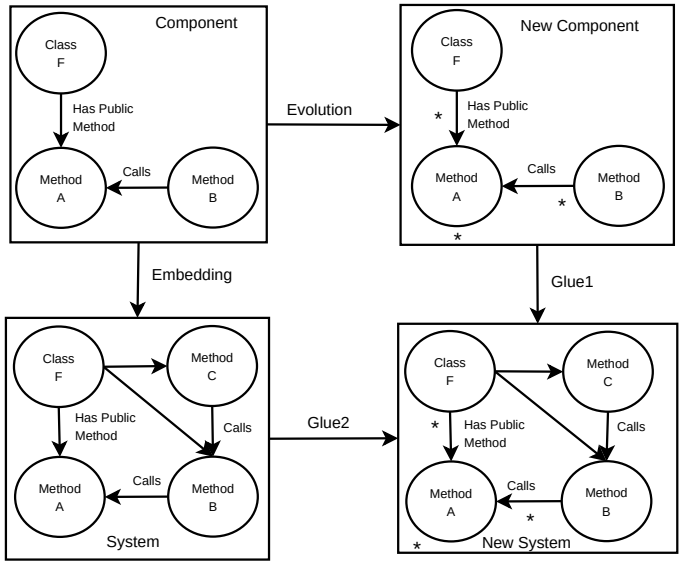

Fig. 9. The pushout that shows the removal of a method. In order not to complicate the diagram, the *has public method* color for the edges between the class F and methods C and D is not shown

gluing construction these marked items are glued to the system. It can clearly be seen from the figure that all morphisms preserve target, source and color mappings and the resulting structure is a graph since there are no dangling arcs. So the marked items can be removed from the system. If the method that is going to be removed has parameters or a return value, then applying only this pushout causes dangling edges that connect the method to its parameters and return value. Thus for such methods, the pushout that removes the parameter or return value should be applied first.

Attribute addition and removal have very similar pushouts as the pushouts for method addition and removal, so we do not show them explicitly here. In the pushout that adds an attribute to a class, the *Component* includes a class. Then the *Evolution* morphism adds the new attribute; thus *New Component* includes the new attribute that is connected to its class with an edge colored with *has public attribute* (or *has private attribute* if the attribute is private). For removal, *Component* contains the attribute that is going to be removed with the edges that connects it to its class and to the methods using it. With *Evolution* morphism these edges and the attribute is marked; the gluing constructions glue the marked items to the *System* thus *New System* is formed. Here, removing the attribute and edges that connect the attribute does not cause any dangling arcs, so they can be removed and the operation succeeds.

Evolution of Classes Evolution in classes can be formalized similar to the pushouts that add or remove a method. In this section, we show the pushouts that describe addition and removal of a class that generalizes a class and aggregates another one. So in the addition pushout diagram, the *Component* graph includes two classes and the *New Component* graph includes three classes with two new edges; one that shows the generalization and the other one shows the aggregation as depicted in Figure 10. For different class relations the same pushout can be used, though the edges that connect the new class to other classes should be colored accordingly. As can be seen from Figure 10, the target, source and color mappings are preserved for each graph; thus the changes are morphisms. Attributes and methods to the newly added class can be added by using the pushouts for these class components. For example, if a class with several methods is going to be added to system then the pushout that adds a method is used for each method after applying the pushout that adds the class. Furthermore, the attributes to these are added with an attribute addition pushout.

In Figure 11, we show the pushout that formalizes the removal of a class. In this figure, class E is removed from system. This causes the edges that connect class E to its base class F and to class D to removed as well. Here, removing the marked items again does not cause any dangling arcs; thus the items can be removed from the system. In general, classes to be removed have several methods that may have some parameters and attributes. For such classes, first the pushout that removes parameters from methods is used; then the pushout that removes the method is applied. Lastly, the class is removed from the system.

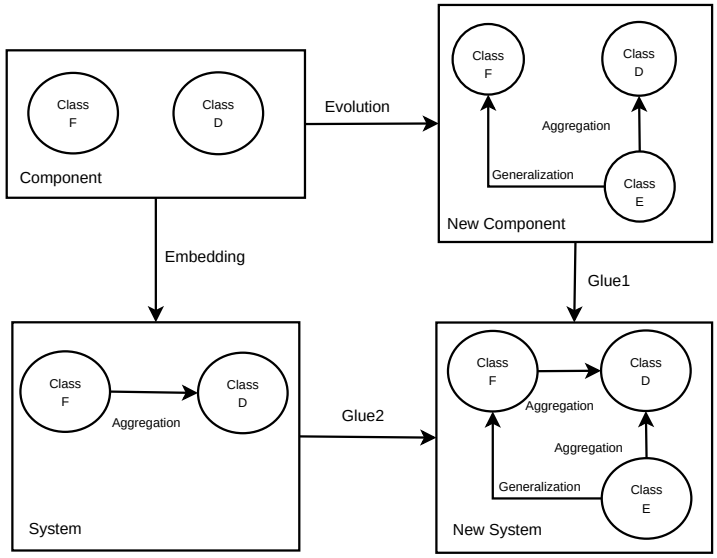

Fig. 10. The pushout that shows addition of a class together with relations with existing classes.

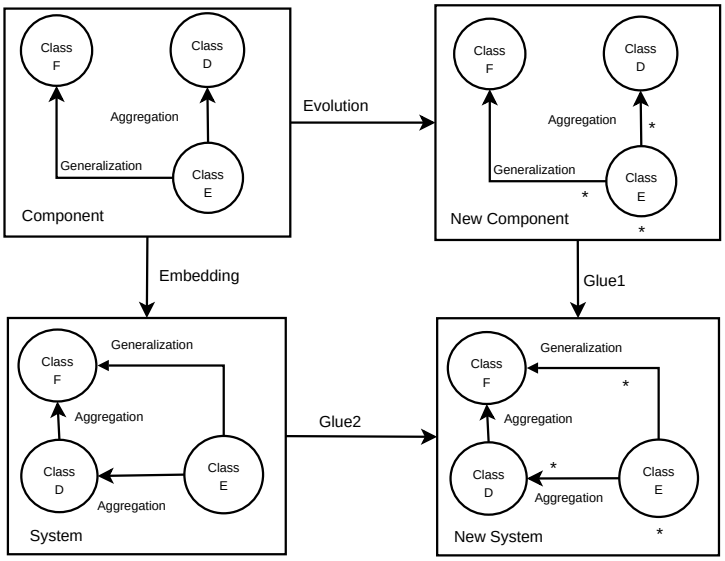

Fig. 11. The pushout that shows the removal of a class and its relations with other classes.

4 Conclusions and Future Work

In this paper, we have presented a way to model evolution requests using algebraic graph rewriting. Our idea is that evolution requests can be formalized with the rules we have presented here. Then these rules can be combined to realize various evolution requests in a top-down manner just like an algorithm. For example, an evolution request that requires addition of a method can be formulated using the method addition rule and the parameter addition rules.

The rules presented here depict the type of changes an evolution request may cause on the system. Thus our next step in modelling software evolution would be using the changes presented in this paper as a taxonomy and finding the cost of each class of changes. For example, the cost of a parameter removal operation is proportional to removing the parameter from all the methods that call the method that has evolved.

References

1. Bennett, K.H., Rajlich, V.: Software maintenance and evolution: a roadmap. ICSE - Future of SE Track (2000) 73–87
2. Belady, L., Lehman, M.: A model of large program development. IBM Sys. J. **15**(1) (1976) 225–252
3. Kemerer, C.F., Slaughter, S.: An empirical approach to studying software evolution. IEEE Transactions on Software Engineering **25** (1999) 493–509
4. Lehman, M.M., Perry, D.E., Ramil, J.C.F., Turski, W.M., Wernick, P.: Metrics and laws of software evolution. Fourth International Symposium on Software Metrics (1997) 20–32
5. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley Professional (1999)
6. Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E.: Modeling software architectures in the unified modeling language. ACM Trans. Softw. Eng. Methodol. **11**(1) (2002) 2–57
7. Alanen, M., Porres, I.: Difference and union of models. In Stevens, P., Whittle, J., Booch, G., eds.: UML 2003 - The Unified Modeling Language. Volume 2863 of Lecture Notes in Computer Science., Spinger-Verlag (2003) 2–17
8. Wermelinger, M., Fiadeiro, J.L.: A graph transformation approach to software architecture reconfiguration. Sci. Comput. Program. **44**(2) (2002) 133–155
9. Mens, T., Van Eetvelde, N., Demeyer, S., Janssens, D.: Formalizing refactorings with graph transformations. Journal on Software Maintenance and Evolution: Research and Practice (2005) 2–31
10. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In Claus, V., Ehrig, H., Rozenberg, G., eds.: Graph Grammars and Their Application to Computer Science and Biology. Volume 73 of Lecture Notes in Computer Science., Spinger-Verlag (1979) 1–69
11. van den Broek, P.M.: Algebraic graph rewriting using a single pushout. In Abramski, S., Maibaum, T., eds.: TAPSOFT'91. Volume 493 of Lecture Notes in Computer Science., Springer-Verlag (1991) 90–102
12. Barr, M., Wells, C.: Category Theory for Computing Science. Prentice Hall (1988)
13. Fiadeiro, J.L.: Categories for Software Engineering. Springer, Leicester, United Kingdom (2005)