Graph Rewrite Systems for Software Design Transformations

Alexander Christoph

christo@fzi.de, Forschungszentrum Informatik (FZI), Germany, http://www.fzi.de

ABSTRACT

Middleware has become an important part in business applications. Different middleware platforms require different software designs. Even version changes of the used platform often lead to complex re-designs of software. Furthermore, complex business and platform-specific knowledge is required, to translate abstract business models to platform specific implementations. This paper presents the software design transformation system GREAT in the context of business applications and the Model Driven Architecture (MDA) proposed by the Object Management Group (OMG). It shows, how graph transformations can be used to transform abstract design models into platform specific ones.

1 Introduction

In the last years, middleware has become more and more important for business applications. Middleware platforms, like OMGs CORBA, SUNs J2EE and Microsofts .NET implement a set of services and communication protocols, that can be used, to connect different applications and to access functionality over remote connections.

The different middleware platforms target at slightly different application domains:

- OMGs CORBA [18] is used to integrate legacy applications together with new software. CORBA offers standardized services, that support remote communication, lookup and other standardized services.
- SUNs J2EE [17] is designed for component-oriented application development and focuses on automatic transaction management and persistence of data.
- Microsofts .NET [16] allows native (local) applications to collaborate over the network and implements communication and security protocols.

Unfortunately, the different platforms make different requirements on software design. So software designs are not portable, as long as they are build specifically for a special middleware platform. Choosing a different platform or even version changes of the used platform requires tremendous porting effort and raises enormous costs.

The same problem exists in different domains of software engineering. Different languages know different concepts of inheritance, implementation, visibility, etc. Software designs, that incorporate a special semantic are thus hard to port to another language.

The Object Management Group (OMG) has realized that problem and developed the *Model-Driven-Architecture* (MDA) [21] approach. The MDA describes a standard for abstract software designs in the domain of business applications, that allows architects to design business software independent from the middleware platform. Standardized transformations allow it to transform the application into a specific version for the chosen platform.

However, these transformations are difficult to implement due to the needed recognition of complex graph structures and a mechanism for consistently modifying software designs.

This paper presents the model transformation system GREAT, that allows to easily specify complex transformations by using a graph-rewriting-system (GRS). Section 2 presents the GREAT transformation system

together with the techniques used. Section 3 shows, how GREAT can be used to implement the model transformations proposed by the MDA. Section 4 gives an idea of the further development of GREAT together with a conclusion before section 5 presents related work.

2 The GREAT transformation system

OMGs first step in proposing abstract designs was the standardization of the Unified modeling language (UML) [11]. UML offers a notation for object-oriented software designs and can be used on all abstraction levels of software development. UML is nowadays widely accepted and used in many software projects. The notation technique can be used to describe the static structure of software (class- and object-diagrams) and its dynamic aspects (interaction-diagrams).

This chapter presents the transformation system GREAT (***Gr****aphgesteuertes* ***E****ntwurfs-****A****nalyse und -****T****ransformationssystem*), that can be used to apply transformations to software designs. GREAT focuses on the static aspects of design, i.e. UML class diagrams.

2.1 Graph Rewrite Systems

Graph rewrite systems (GRS) can be understood as term-replacement systems, that operate on graphs. GRSs are used in domains, where graphs are used to model complex data, like image recognition, compiler optimizations [2] and program transformations. Other application areas can be found in [6].

Using a GRS for graph transformations requires a specification of the graph structure, i.e. the structure of the nodes and edges. A GRS thus consists of a graph description and a set of graph-rewriting rules. The rules contain a graph pattern, that describes the modification points and a set of transformations to be applied to instances of this pattern. These rules are then applied to the host-graph, the input of the GRS. The GRS recognizes pattern instances (occurences of the rule patterns, also *redexes*) and applies the given transformations. A detailed description of GRSs can be found in [5].

2.2 Optimix

There are different implementations of GRSs available. GREAT uses Optimix [3], developed in a PhD thesis [2] for the implementation of compiler optimizers. Optimix is not bound to a specific graph structure, but can, given a graph description, work on every graph. Out of a given graph structure and a set of rewrite-rules, Optimix generates for every rule a Java method, that takes a graph as parameter and implements the search and transformation operations to the given graph. This code generation feature allows to integrate Optimix easily into other applications.

2.3 GREAT

The following sections describe the GREAT transformation system and its implementation. The design goals for GREAT where the interoperability with other software design tools and an easy way of specifying, understanding and documenting model transformations. The following sections describe the GREAT graph model, the GREAT rule translator, responsible for generating transformation modules and the GREAT transformation framework, that provides methods for model modifications.

The GREAT graph model The graph structure used by GREAT is derived from the UML meta-model for class diagrams. These diagrams form a graph, where the entities of the object-oriented paradigm (i.e., classes and interfaces) are the nodes and relations between these entities (e.g., inheritance and implementation) are the edges of the graph. However, nodes and edges can have properties, like name, visibility, etc. Thus the GREAT graph meta-model represents entities and relations as graph nodes. A Relation between entities in the class diagram is represented as instance of the respective relation node, which is connected to the participating entity-nodes. Figure 1 shows an example graph.

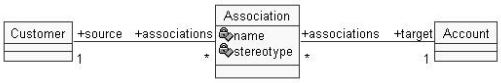


Fig. 1. Example GREAT graph

The GREAT rule translator A GREAT design transformation is expressed by means of a set of Optimix rules and a set of actions that are to be applied to the identified redexes. Optimix rules are mainly used to implement the pattern-search in the class diagram, whereas the modifications itself are implemented by calls to the appropriate framework methods.

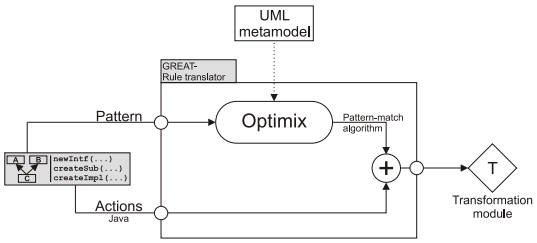


Fig. 2. GREAT rule translator

Figure 2 shows the GREAT rule translator. The tool generates a model transformation module out of a set of Optimix rules and a set of transformation actions. It calls Optimix on the rules together with the GREAT graph model description and attaches the generated Java method to the transformation actions. Then it calls the Java compiler, to create an executable of the module. These transformation modules are executed by the GREAT transformation framework, described in the following section.

The GREAT transformation framework The GREAT transformation framework offers an open transformation platform for UML class diagrams. Figure 3 shows its structure. The framework can inter-operate with other software design tools (CASE tools) via the XMI (XML-based Model Interchange) [12] format. Further, the framework provides a set of transformation methods, that can be used to apply changes to an imported model. Transformations can be implemented by supplying transformation modules. These modules are then executed by the framework, after a model has been imported. After all transformations have taken place, the model is exported.

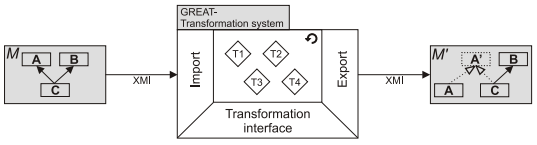


Fig. 3. GREAT transformation framework

Model import and export GREAT reads and writes UML models using the standardized XMI format. XMI was developed by the OMG to allow tools of different vendors to inter-operate. By using XMI, GREAT is independent from the CASE tool used by the designer. Before a model can be transformed, it has to be imported by the XMI importer. Importing a model generates a graph according to the GREAT

graph model. This graph can then be manipulated by the transformation rules. After transformations have finished, the model is exported to an XMI file.

Model manipulating API One goal for GREAT was the ability to write easy and understandable transformation modules. For this reason a transformation interface has been designed, that provides model manipulation functionality. Its methods provide entity creation, retrieving and deletion, the manipulation of relations and entity properties. The methods are responsible for keeping the graph structure correct, when creating or deleting an entity.
A transformation module uses the methods of the transformation interface when applying changes to the model. This can be done either directly in the graph rewrite rule or
separately by an algorithm, working on the set of identified graph nodes.

Processing transformation modules At the moment, GREAT uses programmed graph rewriting. This technique allows the rule developer to specify, in which order transformation modules are executed. Every transformation module is executed once. The transformation has finished, after all modules have been executed. A single module can be designed in two ways: it can use the graph rewriting techniques provided by Optimix to identify and manipulate the model graph, or, it can use the graph rewrite technique merely to identify and collect its redex nodes and implement the redex manipulation in a second step. The first approach is feasible, when applying small changes to the model, the latter, when additional actions like conflict resolution are required.

3 Automated design transformations with GREAT

OMGs model driven architecture (MDA) aims at the development of business applications. The MDA can be seen as further step towards abstract software design after the standardization of the UML. The details of MDA are described in [19]. The MDA defines two sorts of design models: The platform-independent model (PIM) describes the concepts of the application, while the platform-specific model (PSM) specifies the implementation of these concepts in the context of the used middleware platform.

The MDA proposes four kinds of design transformations:

- PIM to PSM: transforms an abstract model into an implementation model for a given platform.
- PSM to PIM: analyses a given implementation and restores its concepts into an abstract model.
- PSM to PSM: refines and deploys implementation specific models.
- PIM to PIM: refines, enhances or analyses an abstract model.

To enable correct transformations, the PIMs use UML profiles, to specify relevant concepts in the model. UML profiles define a set of stereotypes and tagged values that can be used, to label model entities. Different UML profiles exist for different application areas.

Transformation algorithms use the information supplied by the PIM to implement a feasible mapping for the target platform.

This chapter shows, how GREAT can be used to specify the model transformations proposed by the MDA. Besides the information supplied by UML stereotypes and tagged values, GREAT uses pattern matching of subgraphs to identify parts of the model to be manipulated. This allows powerful transformations, since also complex structures in software models can be recognized.

The following examples are provided, to show how a rule developer can use GREAT for both, PIM to PSM and PSM to PIM transformations.

3.1 PIM to PSM - modeling relations

Figure 4 shows a situation, where a local class (Customer) is connected via an association relationship to a business entity, i.e., a class, whose instances usually will be located on a different machine.

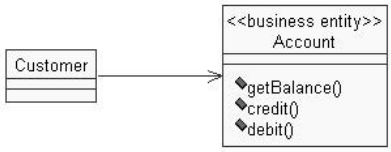


Fig. 4. PIM: remote association

This kind of relation cannot be implemented in the usual way, i.e., through a simple pointer in the Customer class, because additional characteristics of the association, like the address of the remote machine must be modeled.

The implementation could be done with the help of the *Business Facade* design pattern [8]. This pattern splits the relation up into two parts, connected by an interface and an implementation. Figure 5 shows the pattern applied to the example. The purpose of the pattern is it, to hide the fact, that the supplier of the association is not on the same machine as the client. The Account_Impl class provides the required lookup and communication algorithms.

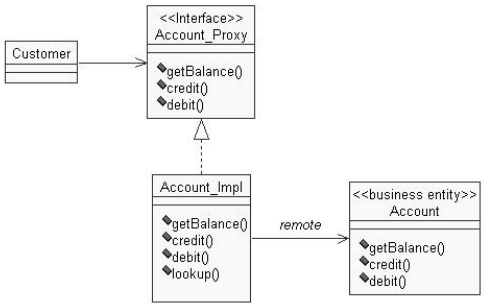


Fig. 5. PSM: Business Facade pattern

In order to transform the example according to the design pattern, a GREAT rule could look as follows:
GREAT uses a GRS - Optimix - that allows subgraphs to specified in a declarative way. The GRS generates a method, that searches a given graph for instances of this subgraph. The code presented in figure 6 represents subgraphs according to figure 4.

After matching instances have been found, transformations can be applied. Transformations can be implemented, using the model transformation interface, provided by GREAT. The actions that modify the example according to the design pattern, are given below.
Both, Optimix graph rules and transformation actions are given to the GREAT rule translator. The translator triggers Optimix to generate the subgraph search method and combines it with the given

```
EARS remoteRelations(
       graph:Root,
       resultSet:EntitySet) {

  if c1 in graph.entities,
     c1 matches Clazz,
     a in c1.associations,
     c2 in a.target,
     c2 matches Clazz(stereotype => ST),
     ST == "business entity"
  then {*
    resultSet.add(new Entity[] {c1, a, c2});
  *}
}
```

Fig. 6. Structure of subgraph to be matched

transformation actions into a transformation module. This module is executed by the GREAT framework after a model is loaded.

3.2 PSM to PIM - analyzing an implementation

Maintaining, and extending given implementations are the most expensive activities in the software life-cycle. This is due to the fact, that in most cases no or outdated abstract software designs are available. The MDA approach proposes PSM to PIM transformations to analyze and understand implementation specific designs. The following example shows, how the PSM from the previous section can be reversed with the help of GREAT.

Figure 8 shows the description of the subgraph, according to figure 5, that has to be identified in the implementation model.

```
...
// prepare result set
resultSet = new ResultSet();

// call generated search method
remoteRelations(hostGraph, resultSet);

// get resulting nodes
for (Enumeration e = resultSet.elements(); e.hasMoreElements(); ) {
  Entity[] entities = (Entity[])e.nextElement();

  Clazz c1 = (Clazz)entities[0];
  Association a = (Association)entities[1];
  Clazz c2 = (Clazz)entities[2];

  // create an interface for the remote class
  Interface intf = trafo.newInterface(c2.getName()+"Intf");
  Clazz impl = trafo.newClazz(c2.getName()+"Impl");

  // create an implementation relationship
  trafo.newImplementation(impl, intf);

  // establish new associations
  trafo.newAssociation(c1, intf);
  trafo.newAssociation(impl, c2);

  // remove old association
  trafo.removeAssociation(a);

  // copy method declarations
  trafo.copyMethods(c2, intf);
  trafo.copyMethods(c2, impl);

  // add an additional lookup method to
  // the implementation class
  trafo.newMethod(impl, "lookup");
}
...
```

Fig. 7. Algorithm to transform the subgraph

```
EARS findFacade(
       graph:Root,
       resultSet:EntitySet) {

if c1 in graph.entities,
   c1 matches Clazz,
   a1 in c1.associations,
   i1 in a1.target,
   e1 in i1.implementations,
   c2 in e1.source,
   a2 in c2.associations,
   c3 in a2.target
then {*
  resultSet.add(
    new Entity[] c1, a1, i1, e1, c2, a2, c3);
*}
```

Fig. 8. Subgraph structure for PSM to PIM transformation

```
...
// prepare result set
resultSet = new ResultSet();

// call generated search method
findFacade(hostGraph, resultSet);

// get resulting nodes
for (Enumeration e = resultSet.elements();
     e.hasMoreElements(); ) {
  Entity[] entities = (Entity[])e.nextElement();

  Clazz c1 = (Clazz)entities[0];
  Association a1 = (Association)entities[1];
  Interface i1 = (Interface)entities[2];
  Implementation e1 =
    (Implementation)entities[3];
  Clazz c2 = (Clazz)entities[4];
  Association a2 = (Association)entities[5];
  Clazz c3 = (Clazz)entities[6];

  // create a direct association btw c1 and c3
  trafo.newAssociation(c1, c3);

  // remove pattern elements
  trafo.removeAssociation(a1);
  trafo.removeAssociation(a2);
  trafo.removeImplementation(e1);
  trafo.removeClazz(c2);
  trafo.removeInterface(i1);
}
...
```

Fig. 9. Transformation algorithm to restore the abstract model

The transformations presented in figure 9 restores the original abstraction by removing the design pattern and creating the direct association between the classes.

4 Conclusion and further work

GREAT implements a graph oriented transformation system for software designs. It brings the power of a Graph Rewriting System to UML models and helps to implement model transformations. These transformations can be used in many situations, one of them is the MDA approach. The paper presented the GREAT system, its structure and tools. First results were presented, that showed, how GREAT can be used to describe complex transformations on software designs.

One major goal for the further development of GREAT will be the simplification of transformation rule specification. Up to now, the rule developer must specify the graph-rewrite part of transformation rules in the Optimix language. In the future, this could be replaced by specifying the subgraph itself with the UML. Another issue is the execution of transformation modules: at the moment it is investigated, if a complete graph rewriting could be used, i.e., the transformation system executes all rules continuously, until no redex is found anymore. For this to be implemented, the system had to know in advance, if the transformation will terminate. Assmann showed in [3] certain termination criteria for GRSs. To make use of these, the system had to analyze the modifications applied to the redexes.

5 Related work

In the field of Graph rewrite systems for software engineering, lot of work has been done by Assmann. In [1] he showed, how GRS can be used for program analysis and defined a subset of GRS, EARS, which is guaranteed to terminate. For another subset, XGRS, he formulated a termination criteria. In [2] he used EARS and XGRS for program optimizations and in [4] he proposed the usage of GRSs in the field of

Aspect Oriented Programming (AOP) [15].

In [10] Fischer et al. propose the use of Graph rewriting to specify the behavior of systems. They use a GRS to specify changes in the object graph of an object-oriented program at runtime. The GRS is then transformed into source code in the appropriate language.

Pertiu and Shen [20] use a GRS to derive a performance graph out of software designs. They analyze static and dynamic diagrams to build a Graph, representing the different roles of entities in design, allowing to analyze the system performance.

UMLAUT [14] is a design transformation framework, that provides searching, retrieving and modification functionality. UMLAUT allows the developer to write transformation code, that is executed on an imported software model. Transformations are specified, using a language similar to LISP. In contrast to GREAT, the developer has to specify the search algorithm along with the transformation. This leads to situations where difficult pattern matching algorithms must be implemented by the rule developer.

Recoder [13] implements a transformation framework for programs. Recoder can import source code in Java and apply modifications to it. An example is the 'obfuscation' of programs, i.e. the renaming of classes and variables to make the code harder to understand, when reverse-engineered.

References

1. Uwe Assmann. On Edge Addition Rewrite Systems and their Relevance to Program Analysis. In Cuny J., Ehrig H., Engels G., and Rozenberg G., editors, *5th International Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*. Springer, 1994.
2. Uwe Assmann. *Generierung von Programmoptimierungen mit Graphersetzungssystemen*. PhD thesis, Universität Karlsruhe, Fakultät für Informatik, 1996.
3. Uwe Assmann. Graph Rewrite Systems for Program Optimization. In *ACM Transactions on programming Languages and Systems (TOPLAS)*, volume 22, 4. ACM Press, New York, NY, USA, 2000.
4. Uwe Assmann and A. Ludwig. Aspect Weaving by Graph Rewriting. In U. Eisenecker and K. Czarnecki, editors, *Generative Component-based Software Engineering*. Springer, 2000.
5. D. Blostein and A. Schuerr. Computing with Graphs and Graph Rewriting. *Software Practice and Experience*, 29(3):1–21, 1999.
6. Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Practical Use of Graph Rewriting. Technical Report 95-373, Department of Computing and Information Science, Queen's University, Kinsgton, Ontario, Canada, 1995.
7. Alexander Christoph. About UML transformations based on Graph Rewriting Systems. In *Workshop on Transformations in UML (WTUML), held during the ETAPS'01 conference*. Genova, It., April 2001.
8. John Crupi. Core J2EE Patterns. *Java Developer's Journal*, 06(08), August 2001.
9. Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In *Proceedings of UML 2000*, number 1939 in LNCS, page 323ff. Springer Verlag, October 2000.
10. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation TAGT*, number 1764 in Lecture Notes in Computer Science. Springer, 1998.
11. Object Management Group. UML version 1.1, 1997.
12. Object Management Group. OMG XML Metadata Interchange (XMI) Specification, 2000.
13. D. Heuzeroth, W. Löwe, A. Ludwig, and Uwe Assmann. Aspect-Oriented Configuration and Adaptation of Component Communication. In Jan Bosch, editor, *Generative and Component-Based Software-Engineering, Third International Conference, GCSE 2001, Erfurt Germany*, number 2186 in Lecture Notes in Computer Science, pages 58–69. Springer, Sep 2001.
14. Wai-Ming Ho, J.-M. Jezequel, A. Le Guennec, and F. Pennaneac'h. UMLAUT: an extendible UML transformation framework. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering, Cocoa Beach, Florida, USA*, pages 275–278. Institute of Electrical and Electronics Engineers, 1999.
15. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 -*

Object-Oriented Programming 11th European Conference, Jyväskylä, Finland, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.
16. Microsoft. Defining the basic elements of .NET. http://www.microsoft.com/net/defined/whatis.asp, April 2002.
17. Sun microsystems. Java 2 Enterprise Edition (J2EE) platform specification, 1.3. http://java.sun.com/j2ee/, 2001.
18. Robert Orfali, Dan Harkey, and Jeri Edwards. *Instant CORBA*. Addison-Wesley, 1998.
19. Architecture Board ORMSC. Model Driven Architecture (MDA), July 2001.
20. Dorina C. Petriu and Hui Shen. Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications. In *Proceedings of the Performance TOOLS'2002*.
21. Richard Soley. Model Driven Architecture, November 2000.