Model-Based Architecture Restructuring Using Graph Clustering

Niels Streekmann
OFFIS - Institute for Information Technology
Technology Cluster Enterprise Application Integration
Escherweg 2, 26121 Oldenburg, Germany
niels.streekmann@offis.de

Wilhelm Hasselbring
University of Kiel
Software Engineering Group
Olshausenstr. 40, 24098 Kiel, Germany
wha@informatik@uni-kiel.de

Abstract

Implementations of existing systems often do not follow the originally intended architecture. Continous extension disregarding the intended architecture leads to a decline of the maintainability of these systems. To recover maintainability architectural restructuring becomes necessary. We present a model-based architecture restructuring approach that is based on business requirements. The goal of the approach is to semi-automatise the architecture restructuring process in order to support reengineers. We use graph clustering to implement the automatisation.

1. Introduction

In many existing business information systems the maintainability has worsened due to years of maintenance that did not respect the originally intended architecture of these systems. This lead to architecture erosion. Given reasons are often the pressure for quick implementations of business requirements and the difficulty to convince customers of the necessity to invest in architecture maintenance. These investments are commonly not made until they become inevitable. In consequence the adaptability of business systems to quickly changing business requirements decreases.

One possibility to recover maintainability and to enable the efficient implementation of new business requirements is architecture restructuring. We assume that maintainability is higher when the architecture includes clearly defined interfaces that are based upon business functions. This assumption complies with the concept of service-oriented architectures, which are at the centre of interest today, because they allow a stronger relationship between business requirements and software systems and therefore support adaptability to business change.

This paper proposes a model-based approach to architecture restructuring. The approach is aimed to support reengineers in the planning phase of a restructuring project. It supports the decision of how the existing implementation can be transfered to a new architecture without changing the functionality of the code or the execution environment. The approach focuses on realising a target architecture instead of on understanding the existing architecture. It is assumed that the target architecture is built in regard to current and future business requirements. The goal is to find dependencies in the current implementation that do not conform to the target architecture. These dependencies have to be removed during the actual restructuring of the implementation, e.g. using refactoring, in order to apply the target architecture. Therefore understanding the existing architecture and its dependencies in every detail is not required in our approach.

The contribution of this paper is an approach that provides automatised support for architecture restructuring of existing software systems. The approach is based on the combination of forward and reverse engineering to connect reengineering tasks to business requirements. The automisation is implemented by graph clustering using weighted dependencies in existing systems as core artefacts.

The paper is structured as follows. Section 2 describes our approach to support architecture restructuring. Section 3 describes how the approach can be used to improve the maintainability of existing systems while Section 4 introduces further application scenarios. Section 5 sketches the planned evaluation of the approach. Section 6 lists related work before Section 7 concludes the paper.

2. Model-Based Architecture Restructuring

This section describes our model-based approach to architecture restructuring. It is supposed to support reengineers in assigning elements of the implemented architecture (called source elements in our approach) to components of the target architecture (target components). Figure 1 shows an overview of the approach. We assume that a model of the existing system can be created using reverse engineering methods. Another assumption is that there is a target architecture model, which is typically created by a software architect based on business requirements. The forward engineering process that leads to the target architecture can comprise more steps than the simplifying depiction in Figure 1. A description of the mentioned models and the requirements we have identified regarding their content is given in Section 2.1.

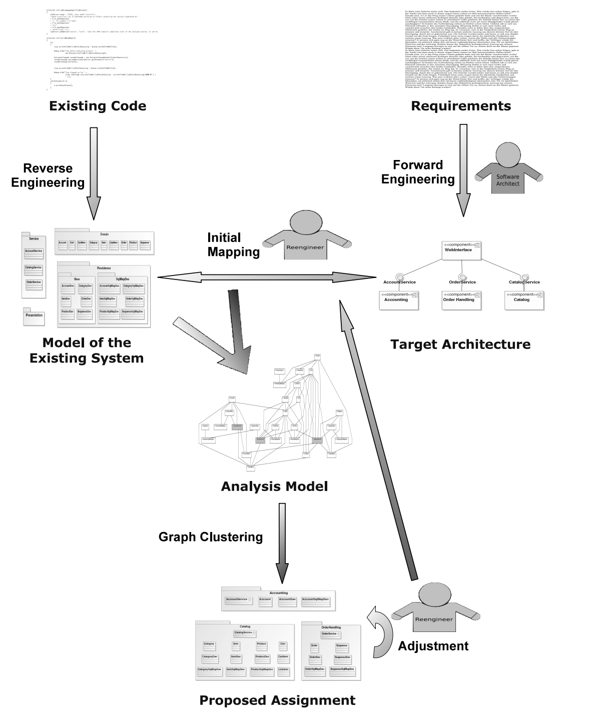

Figure 1. Overview of the proposed approach

The first step of the approach itself is an initial mapping between elements of the target architecture and the model of the existing system made by a reengineer. The mapping is detailed in Section 2.2. Together with the model of the existing system the initial mapping is taken as input for the creation of our analysis model. The analysis model is a graph containing source elements as nodes and dependencies between them as edges. A description and an example are given in Section 2.3.

The analysis model is used as input for a graph clustering algorithm that assigns all source elements to target components. This proposed assignment is the output for the reengineer. A simple algorithm that incorporates the initial mapping and dependency weights is given in Section 2.4. The reengineer is supposed to check the assignment incorporating quality metrics and knowledge about the system. If the result is not satisfying, adjustments can be made directly to the assignment or to the analysis input, which makes the analysis an iterative process that can be influenced by the reengineer. More details about possible adjustments are described in Section 2.5. The adjusted proposed assignment can then be taken as a basis for the restructuring of the actual implementation. It shows which source elements belong to which target components and indicates the dependencies that do not conform to the target architecture.

2.1 Forward and Reverse Engineering

The basis of the proposed approach is the combination of forward and reverse engineering in order to restructure an existing system. In a forward engineering process the target architecture of the future system is defined by a software architect according to the business requirements. In our approach we only focus on the structural architecture of the system. Behavioural and deployment aspects are not considered. We assume that the architecture describes the components of the future system and its interfaces including operations. We also assume that the information objects that are managed by the components are part of the modelled architecture.

The model of the existing system is created from the source code using reverse engineering. The model has to provide information about all source elements and their dependencies. Source elements are e.g. classes, interfaces and methods in object-oriented systems. The consideration of higher-order elements like packages is only reasonable in scenarios where implemented architecture and target architecture are very similar. In other cases these should be ignored. The dependencies between source elements have to be described with their types. The following types of dependencies are currently considered for systems written in Java:

- Method Calls
- Inheritance
- Classes as types of return parameters of methods
- Classes as types of method parameters
- Classes as types of constructor parameters
- Classes as types of variables
- Castings to a class

These dependencies are weighted according to the indication they give about the cohesion of two source elements regarding the semantics of these elements in the business domain. E.g. we assume that inheritance and the type of a return parameter indicate a higher cohesion than method calls. These weights are used by the graph clustering algorithm to compute the assignment of source elements to target components.

2.2 Initial Mapping

The initial mapping has to be done by reengineers in order to get a first relation between the target architecture and the existing system. The initial mapping serves as a starting point for the clustering algorithm described in Section 2.4. The goal is to keep the initial mapping as small as possible to minimise the effort of the reengineers in the restructuring process. In order to be able to apply the following analysis, the initial mapping has to map at least one source element to each target component. Opposed to a complete manual mapping of source elements to target components we try to define the minimal initial mapping that results in a proposed assignment (cf. Figure 1) of high quality. The quality of the proposed assignment can be measured by metrics defined on the architecure model or judged by a reengineer based on his knowledge about the business domain and technical details of the system.

The initial mapping consists of two kinds of mappings: the mapping of interfaces and operations in the target architecture to source elements and the mapping of information objects in the target architecture to source elements. For object-oriented systems, interfaces of the target architecture are mapped to interfaces in the model of the existing system or the operations of the target architecture interfaces are mapped to methods in the model of the existing system. Information objects can be mapped to classes in the model of the existing system.

Figure 2 shows a simplified example taken from a first case study using the JPetStore reference implementation[1]. The upper half depicts parts of the target architecture. The component *OrderHandling* provides an interface *OrderService* to retrieve and insert order objects. The component also manages *Order* information objects. The lower half shows the model of the existing system with an interface *OrderDao* and its methods which uses an *Order*. As a simple example of an initial mapping the *Order* in the target architecture can be mapped to *Order* in the existing system and the operations of the interface *OrderService* can be mapped to the methods of the interface *OrderDao*.

In practice mappings will be more complex than in the given example. Naming and structure can be different in the target architecture and the existing system and the implementation of interfaces described in the target architecture may be distributed over different interfaces and classes in the existing system. We assume that the initial mapping is done manually by a reengineer. Related work on the automisation of such mappings exists (e.g. [15]), but automisation of the initial mapping is not in the scope of our work, because we assume the initial mapping to be realisable with acceptable effort. We also assume the manual mapping to be more reliable than current automatised approaches.

[1]`http://ibatis.apache.org/javadownloads.html`, last visit: February 9, 2009

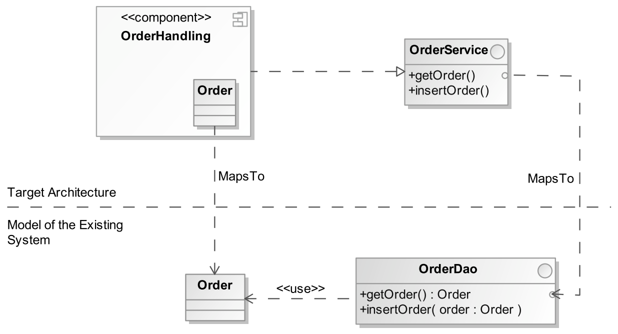

Figure 2. Simple example of an initial mapping

2.3 Analysis Model

The analysis model is a graph model which is structured as follows: the nodes of the graph represent source elements while directed edges represent dependencies between source elements. The nodes are typed according to the type of the source elements. The edges are typed according to the types of dependencies described in Section 2.1. The dependency types are weighted in order to gain architectural reasonable assignments of source elements to target components in the analysis. The implementation of this graph-based analysis is described in 2.4. The initial mapping is taken as a pre-assignment of source elements to target components with which the nodes are annotated.

Figure 3 shows an example of dependencies between interfaces and classes using the elements from the model of the existing system from figure 2 and the class *OrderSqlMapDao*. As can be seen in figure 3, *OrderDao* defines two methods that have *Order* as the type of an input and a return parameter. These dependencies are also depicted in figure 3. Since *OrderSqlMapDao* implements *OrderDao* – which is another dependency– it also inherits the dependencies to *Order*. In the implementation of these methods two more dependencies occur since *Order* is used as the type of a variable and the type of a cast.

Figure 4 depicts the example in a graph representation with typed nodes and edges. For reasons of clearness, we only show the dependencies on the level of classes and inter-

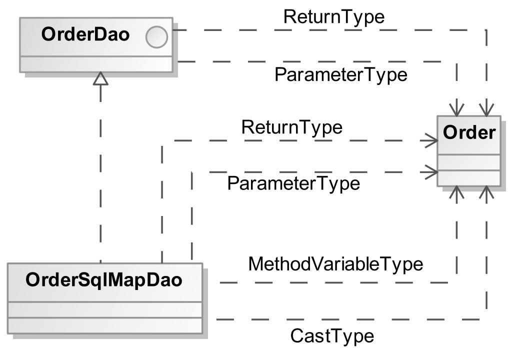

Figure 3. Example of dependencies between classes

faces, in the example. The lowest structural level we consider in the approach are methods, which covers the case that classes are split up or combined during the restructuring. The structure of methods, e.g. loops and branches, and hence the split-up of methods during restructuring are not considered.

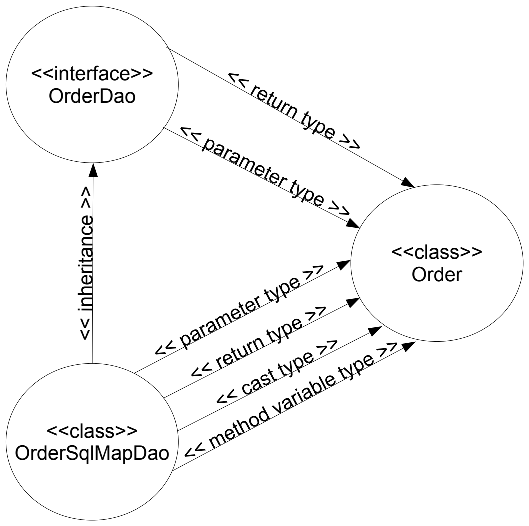

Figure 4. Graph representation of the example

2.4 Graph Clustering Analysis

To assign all source elements to target architectures we use graph clustering methods. We will describe a simple hierarchical clustering algorithm that makes use of the information given in the initial mapping and the dependency weights in the following. The pre-assignments to target components are used to define initial clusters of source elements – whereby one cluster corresponds to one target component. To be able to define one initial cluster for each target component at least one source element has to be assigned to each target component in the initial mapping. An initial cluster comprises all source elements that are mapped to the interfaces or information objects of its corresponding target component. In the example in Figure 2 and Figure 3 *OrderDao* and *Order* would be assigned to an initial cluster that represents the target component *OrderHandling* while *OrderSqlMapDao* would not be assigned to an initial cluster.

The following enumeration sketches the hierarchical clustering algorithm. It works like a typical hierarchical clustering algorithm as e.g. described in [12] with the exception that it will not combine two initial clusters. Thus the algorithm will not execute until only one cluster is left, but will stop when all source elements are assigned to one of the initial clusters. The cohesion between clusters is defined using the dependency weights. To compute the cohesion the weights of all dependencies between the source elements in both clusters are added. A high sum of dependency weights indicates a high cohesion between the clusters.

```
1. Create one cluster for each source
   element

2. Define initial clusters

3. Search for the two clusters with the
   highest cohesion that are not both
   initial clusters

4. Combine the two clusters with the
   highest cohesion to a new cluster

5. If one of the two clusters is marked
   initial, also mark the new cluster

6. Goto 3 as long as there are clusters
   that are not initial clusters
```

Figure 5 shows an example of the clustering algorithm. The clusters for each source element are depicted at the bottom. Cluster 2 and 3 and 6 to 8 are combined to initial clusters (IC1 and IC2) due to their pre-assignment to target components. Shaded clusters are marked as initial cluster which means that they are not allowed to be combined. Above the second dotted line the iterative hierarchical clustering process is shown. Unmarked clusters can be combined with unmarked or marked cluster until only marked clusters are left. Relating to the example used before, 2 and 3 could e.g. be substituted with *Order* and *OrderDao* and 1 with *OrderSqlMapDao*.

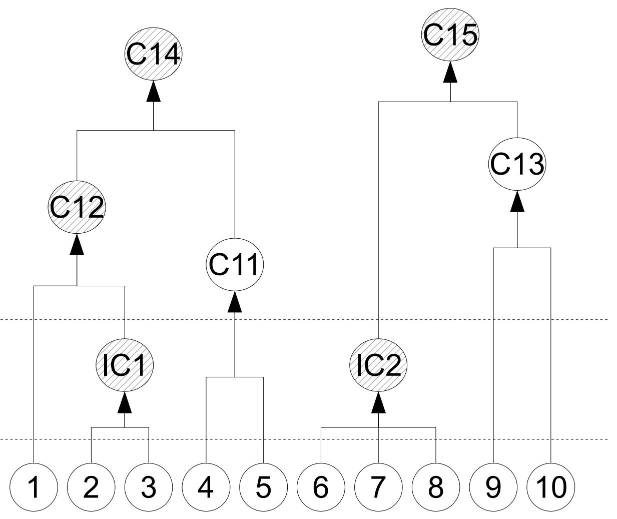

Figure 5. Simple clustering example

1 The sketched algorithm is only a first implementation of a clustering algorithm that considers the initial mapping and dependency weights. Further work on the algorithm will follow, including the evaluation of other published algorithms towards their adaptation to our requirements. Another open question is whether the algorithm should provide a total clustering as shown here or just a partial clustering that leaves relevant clustering decisions for the reengineer. Besides the assignment to one target component, these decisions could e.g. also include to redundantly add a source element to two target components, if this is reasonable in the domain or simplifies the further reengineering process.

2.5 Adjustments

There are two kinds of adjustments a reengineer may perform after the analysis. One is to adjust the assignments manually in order to apply minor changes, if the analysis provides an adequate result. The second adjustment is to change the initial mapping in order to gain better analysis results. The examination of the relationship between changes in the initial mapping and the proposed adjustments is part of our future work.

The main reason for proposing this iterative process is the reduction of the effort for the reengineer. It makes it possible to start a first analysis with a small initial mapping, e.g. only of information objects. In many cases these will imply many other assignments like persistence or computation on the information objects. If the results turn out to be inadequate, the mapping can be refined and the algorithm can be run again until a satisfying result is reached. Inadequacy of results refers to assignments that are not reasonable from the viewpoint of the business domain or that lead to many dependencies between target components that do not conform to the architecture. The latter corresponds to high coupling and results in high refactoring effort to dissolve the dependencies in order to implement the target architecture.

3. Maintainability Improvement

Lack of maintainability is a frequent problem in the development of enterprise information systems. Especially systems that have been developed for years suffer from architecture erosion and inadequate documentation. Often the knowledge about details of the system is only existing in the heads of a few developers. New developers will then have problems to acquaint themselves with the system. Also architecture erosion leads to unexpected behaviour of the system when changes are made, because not all dependencies in the system are known or documented.

We assume that a system that conforms to a target architecture will exhibit a better maintainability. To enable this the target architecture should have a clear structure that complies to the business needs and the implemented system should not have dependencies that do not conform to this architecture. It will also allow a better task sharing between developers and a better separation of concerns regarding the functional aspects of the system. Furthermore, an such a target architecture can serve as a documentation for the system as long as it is maintained together with it.

It is assumed that the advantages of our approach for maintainability improvement will show best, when the implemented architecture and the target architecture have a certain descrepancy. That descrepancy may be due to architecture erosion or a planned restructuring of the system according to the adjustments to business requirements. The approach can tap its full potential when the effort of manual assignment of source elements to target components is infeasible due to the complexity of the system or unknown dependencies. In practice this will appear e.g when packages can not be mapped to a target component as a whole, but will have to be split up.

4. Other Application Scenarios

The proposed approach is applicable in several scenarios. Besides the improvement of maintainability in a reengineering project other scenarios are the extraction of services for a service-oriented architecture, the smooth migration or the extension of a product to a product line.

4.1 Extraction of Services

The extraction of services from monolithic applications is a possible task during the introduction of a service-oriented architecture. This task is needed to offer parts of the functionality of a complex system as an independent service in several areas of an application landscape. Another reason is simply to reduce the dependencies, which may be an even more important reason that the reusability of a service. Work in the area of the migration to service-oriented architectures can be found in [16, 17].

The target architecture in this scenario has to include at least two components. One modelling the service that shall be extracted and one modelling the system it shall be extracted from. It also has to model the complete interface of the service and all required interfaces it will still use from the existing system. The provided interfaces together with the information objects it manages will define which source elements belong to the service implementation while the required interfaces define the borders of service implementation. Using this input information the analysis algorithm will assign source elements to the service component and dependencies that do not correspond to the target architecture can be recognised.

4.2 Smooth Migration

Smooth migration describes a stepwise migration of an existing system from a source environment to a target environment [7]. An example is the migration from COBOL to Java. In these cases the functionality of the existing system remains the same in the new system and new requirements are only implemented in the target environment. Systems that are candidates for a smooth migration are often grown over years and suffer from architecture erosion. To migrate these systems stepwise it is necessary to first restructure them in order to be able to clearly define the source elements that are migrated in one migration step. The application of graph clustering to identify the migration steps in a smooth migration scenario is described in [13].

If the proposed approach is applied to this scenario, the target architecture will in most cases be very similar to the architecture of the current implementation. The reason therefor is to keep the refactoring effort in the source environment as small as possible. Further improvements of the architecture may then be made in the target environment. Thus the main task of the analysis algorithm is to find dependencies that do not conform to the target architecture. In this case it may be useful to incorporate packages in the initial mapping to be able to map groups of source elements that shall not be changed in the target architecture directly to one target component.

4.3 Product Extension

Individual software products are sometimes extended to software product lines. Reasons for this are the improvement of the adaptability for different customers or the consolidation of existing products. To accomplish the extension the components of the individual system have to be uncoupled in order to enable configuration for different customers. Therefore the dependencies between components have to be reduced and need to conform to the target product line architecture.

Relating to the proposed approach the target architecture will be build according to similar business requirements as the existing system. The main difference will be the facilitation to configure a product according to varying customer needs. Thus the architecture of the implemented system and the architecture of the target system can in some cases also be very similar, especially in relatively new systems. Nonetheless it is a goal in such projects to reduce the intended dependencies between components, which legitimates the refactoring effort.

5. Evaluation Aspects

An evaluation of the approach is planned to answer the following research questions:

- How can the quality of the restructuring solution be improved?
- How does the analysis algorithm react to changing input parameters?
- How does the approach behave compared to alternative approaches?

The evaluation is based on an GQM plan as described in [1]. The goals in the plan correspond to the afore mentioned research questions. Some of the question and metrics of the GQM plan are exemplified in the following. The metrics include metrics for the internal and external evaluation according to [9]. Both kinds of metrics are needed our context since internal metrics help to ensure the software engineering quality while external metrics ensure domain and business orientation.

An important question regarding the quality improvement of the restructuring solution is the consideration of the quality of the analysis results. To evaluate the quality several metrics are defined in our GQM plan. A simple metric is to count the source elements per target component. For a high quality usually a uniform allocation is assumed, but depending on the difference between implemented and target architecture the results may vary. More significant results can be expected from the consideration of coupling and cohesion metrics. We will consider coupling and cohesion metrics as e.g. proposed by [8] and [4] for object-oriented systems.

Other metrics that influence the quality of the proposed assignment are metrics regarding the dependencies that do not conform to the target architecture. Relevant metrics are the number and the weight of these dependencies. Both influence the refactoring effort since they have to be dissolved in the architecture restructuring project. It is assumed that a small number of these dependencies and low weights indicate a high quality. To improve the quality also the weights themselves have to be in the scope of the evaluation.

To improve the usability of the approach the reaction of the graph clustering algorithm to changing input parameters will be evaluated. The relevant input parameters are the initial mapping and the dependency weights. For both it has to be examined how the proposed assignment changes for typical changes of the input parameters. Typical changes for the initial mapping are e.g. the addition of the mapping of an information object or the operation of an interface.

First results in a small evaluation scenario showed that the proposed assignment does not change continually for changing dependency weights, but exhibits bigger changes at certain values. It has to be examined how these values are formalised in order to predict this behaviour.

Since most related approaches are aimed at the implemented architecture as a starting point for program understanding or architecture refactoring (cf. 6), the comparison with these approaches will only be possible for architecture restructurings where the target architecture is similar to the implemented architecture. For these cases the assignments of source elements to target components can be compared. Metrics are the number of identically assigned source elements or the number and weight of dependencies between components that do not conform to the target architecture. Other metrics are the rating of the assignments by experts regarding the reengineering effort and the domain-specific belonging of source elements to target components.

A first case study with a small web application showed that the approach was applicable in that case. It also indicated that clustering algorithms utilising dependency weights and an initial mapping can produce better results than simple hierarchical clustering algorithms without weighted dependencies and initial clusters. It also turned out that the algorithm described in Section 2.4 leads to groups of source elements that are clustered differently for varying dependency weights. It indicated that there are certain thresholds for dependency weights at which the clustering result changes. This will have to be examined more deeply in further case studies with larger systems.

6. Related Work

The combination of forward and reverse engineering which is the basis of the proposed approach also plays a major role in the integration of software systems. [6] describes an approach that combines domain models with concrete models of existing systems for data integration in e-commerce. It also describes the influence of models on each other in an iterative approach. This can be assigned to our approach by changing the target architecture according to the knowledge gained from the analysis of the existing system.

The combination of forward and reverse engineering for the integration of software systems is also described in [14], which describes a model-driven integration process. It proposes a linking on the platform-independent model level as defined in [11] that corresponds to the initial mapping described in Section 2.2.

[5] describes how the reflexion method for program understanding [10] can be extended by automatic clustering. Clustering methods are used to automatise the manual assignment of source elements to a hypothesised architecture. The main difference to our approach is that the hypothesised architecture shall approximate the implemented architecture as good as possible. So the focus of the approach lies in finding unknown dependencies that do not conform to the hypothesised architecture. By contrast, our approach is supposed to create an assignment of source elements to a newly designed architecture motivated by new business requirements.

Further approaches that are aimed at the understanding of the existing architecture and changing it according to new business requirements are described in [2, 3]. These differ from our approach since they restructure the architecture incrementally based on knowledge about the existing system. Our approach on the other hand takes an architecture as a starting point that is independent of the existing system and aims at reducing the knowledge needed about the existing system during the reengineering process.

7. Conclusions and Future Work

In this paper we presented an approach to support the architecture restructuring. The approach is based on models of an existing system and a target architecture. From these an graph-based analysis model is created from which assignments of source elements to target components can be computed via graph clustering. We introduced possible application scenarios and sketched an GQM plan for the evaluation of the approach. The approach can enable the recovery of the maintainability of existing systems by supporting architecture restructuring.

Our future work will focus on the evaluation of the proposed approach according to the evaluation aspects described in Section 5. The evaluation will take place in a case study with an industrial partner. We will further improve the clustering algorithm and examine the possibility to insert other published algorithms. We will also improve the tooling for the creation of the analysis model and the initial mapping.

References

[1] V. R. Basili, G. Caldiera, and H. D. Rombach. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*, pages 528–532. Wiley, 1994.

[2] M. Bauer and M. Trifu. Architecture-Aware Adaptive Clustering of OO Systems. In *8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 3–14, 2004.

[3] W. R. Bischofberger, J. Kühl, and S. Löffler. Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking. In *EWSA*, volume 3047 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2004.

[4] S. Chidamber and C. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[5] A. Christl, R. Koschke, and M.-A. Storey. Equipping the Reflexion Method with Automated Clustering. In *Proc. of 12th Working Conference on Reverse Engineering*, pages 89–98, Pittsburgh, PA, USA, November 2005. IEEE Computer Society.

[6] W. Hasselbring. Web Data Integration for E-Commerce Applications. *IEEE Multimedia*, 9(1):16–25, March 2002.

[7] W. Hasselbring, R. Reussner, H. Jaekel, J. Schlegelmilch, T. Teschke, and S. Krieghoff. The Dublo Architecture Pattern for Smooth Migration of Business Information Systems: An Experience Report. In *Proceedings of the 26th International Conference on Software Engeneering (ICSE 2004)*, pages 117–126. IEEE Computer Society Press, May 2004.

[8] M. Hitz and B. Montazeri. Measuring Coupling and Cohesion In Object-Oriented Systems. In *Proceedings of the 3rd International Symposium on Applied Corporate Computing (ISACC1995)*, Oct. 1995.

[9] O. Maqbool and H. A. Babri. Hierarchical Clustering for Software Architecture Recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.

[10] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[11] Object Management Group (OMG). MDA Guide Version 1.0.1, June 2003.

[12] S. E. Schaeffer. Graph Clustering. *Computer Science Review*, 1(1):27–64, 2007.

[13] N. Streekmann and W. Hasselbring. Towards Identification of Migration Increments to Enable Smooth Migration. In R. Kutsche and N. Milanovic, editors, *First International Workshop on Model-Based Software and Data Integration - MBSDI 2008*, number 8 in Communications in Computer and Information Science, pages 79–90. Springer Verlag, April 2008.

[14] W.-J. van den Heuvel. Matching and Adaptation: Core Techniques for MDA-(ADM)-driven Integration of new Business Applications with Wrapped Legacy Systems. In *Proceedings of MELS Workshop (EDOC)*. IEEE Press, October 2004.

[15] W.-J. van den Heuvel. *Aligning Modern Business Processes and Legacy Systems - A Component-Based Perspective*. Cooperative Information Systems. MIT Press, 2007.

[16] A. Winter and J. Ziemann. Model-based Migration to Service-oriented Architectures - A Project Outline. In H. Sneed, editor, *CSMR 2007, 11th European Conference on Software Maintenance and Reengineering, Workshops*, pages 107–110. Vrije Universiteit Amsterdam, Mar. 2007.

[17] J. Ziemann, K. Leyking, T. Kahl, and D. Werth. Enterprise Model driven Migration from Legacy to SOA. In R. Gimnich and A. Winter, editors, *Workshop Software-Reengineering und Services*, Fachberichte Informatik, pages 18–27, Koblenz, Germany, 2006. University of Koblenz-Landau.