Architecture Migration Driven by Code Categorization

R. Correia[1,2], C. Matos[1,2], R. Heckel[1], and M. El-Ramly[3]

[1] Department of Computer Science, University of Leicester, University Road, Leicester, LE1 7RH, United Kingdom
{rmc20, cmm22, reiko, mer14}@mcs.le.ac.uk
[2] ATX Software, Rua Saraiva de Carvalho 207C, 1350-300 Lisboa, Portugal
[3] Computer Science Department, Cairo University, Egypt

Abstract. In this paper, we report on the development of a methodology for the evolution of software towards new architectures. In our approach, we represent source code as graphs. This enables the use of graph transformation rules, allowing the automation of the transformation process. Prior to its model representation, the source code is subject to a preparatory step of semi-automatic code annotation according to the contribution of each of its parts in the target architecture. This paper first describes the overall methodology and then focuses on the code annotation and model transformation parts. We also discuss issues of the implementation of the approach based on existing tools.

1 Introduction

As business and technology evolve and software becomes more complex, researchers and vendors of tools in reengineering are constantly challenged to come up with new techniques to effectively support the transition of legacy systems to modern architectures.

In this paper, we introduce a methodology to fill the gap that exists in addressing systematically the complexity of architecture migrations. We argue that, starting from a legacy application, such transitions involve different steps of decomposition. Depending on the target architecture, these are made along one or both technological and functional dimensions. Technological decomposition is used in the layering of software systems and may, for example, lead to a 3-tiered architecture, separating logic, data, and user interface (UI). Functional decomposition, used to move to Service Oriented Architectures (SOAs), separates components which, when removing their UI tier, represent candidate services.

Based on a metamodel for both source and target architecture, the methodology presented in this paper consists in (1) categorizing the source code according to the different elements of the target architecture they shall be mapped to, (2) obtaining a metamodel-based representation of the code, (3) transforming it into the target architecture, and (4) generating the target code.

At this point, we have implemented in an automated way the transformation and partially implemented the semi-automatic code categorization, but only for

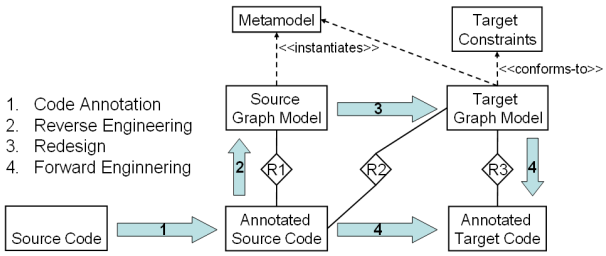

Fig. 1: Methodology for architectural migration

the technological decomposition (or layering). This is where the paper concentrates on. Its main contribution is the automation of the architectural migration using graph transformation rules over a model of the annotated source code. This allows us to: abstract (in large parts of the process) from the specific languages involved and describe transformations in a more intuitive way (compared to code level transformations). Along the paper a small Java client-server application is used to exemplify the implementation of some of the steps. The intended target architecture is 3-tier.

The remainder of this paper is organized as follows: Section 2 presents our methodology for architectural transformation. The code annotation and the model transformation parts of our approach are discussed in Sections 3 and 4 respectively. We review related work in Section 5 and discuss conclusions and further work in Section 6.

2 Architectural Transformation Methodology

In this section we discuss methodological aspects of our approach to architectural redesign. We are following the Horseshoe Model [1], refining it to support automation and traceability. Our methodology consists of the three steps of *reverse engineering, redesign*, and *forward engineering*, preceded by a preparatory step of *code annotation*, as illustrated in Figure 1.

A *metamodel* composed of a type graph that represents the technological paradigm of the system and a list of the code categories needed regarding the target architecture is used. *Target constraints* are also used to ensure that the target model is achieved and complies to the expected one.

1. Code Annotation. The source code is annotated by *code categories*, distinguishing its constituents (classes, methods, or fragments thereof) with respect to their foreseen association to architectural elements of the target system.

The annotation is done by means of comments in the original source code and even though the code is categorized statement by statement, in the end, consecutive statements of the same category are grouped making possible for a whole class or package being annotated with just one code category. This procedure makes the graph model representation of the source code much simpler

than if there was no categorization step, since the level of detail used can be much lower. Simpler graph models make the redesign step easier to scale.

This semi-automatic code annotation is based on *categorization rules* defined at the level of the Abstract Syntax Tree (AST), taking into account information obtained through dependency analysis and inputs by the developer. The results may have to be revised and the propagation repeated in several iterations, leading to an interleaving of automatic and manual annotations.

2. Reverse Engineering. From the annotated source code, a *graph model* is created, whose level of detail depends on the annotation. For example, a method wholly annotated with the same code category is represented as a single node, but if the method is fragmented into several categories, each of these fragments has to have a separate representation in the model. The relation *R1* between the original (annotated) source code and the graph model is kept to support traceability. This step is a straightforward translation of the relevant part of the AST representation of the code into its graph-based representation.

3. Redesign. The source graph model is restructured in order to comply to the target architecture. In our approach, code categories provide the control required to automate the transformation process, focussing user input on the annotation phase. During this redesign step, the relation with the original source code is kept as *R2* in order to support the code generation.

This *code category-driven transformation* is specified by graph transformation rules, conceptually extending those suggested by Mens *et al* [2] to formalize refactoring by graph transformation.

4. Forward Engineering. The target code is generated from the target graph model and the original source code, using their relation *R2* as an input. The result of this step, the annotated code in relation with a graph model, has the same structure as the input to Step 1. Hence, the process can be iterated. This is particularly relevant if the reengineering is directed towards service-oriented systems, because the transformation has to address first the technological and then the functional decomposition.

3 Code Annotation

The annotated source code is obtained through an iteration of manual definition of the categorization rules and the automatic application of them, based on the categories defined in the *metamodel*. After each iteration, manual input might be needed to refine the categorization rules in order to achieve code fully categorized.

In the example mentioned in section 1, our goal is to reach a three-tier architecture, thus we can use the following code categories: User Interface (UI), Logic, Data, Control: UI to Logic, Control: Logic to UI, Control: Logic to Data. For different target architectures, other categories might be defined as well as more complex ways to represent them.

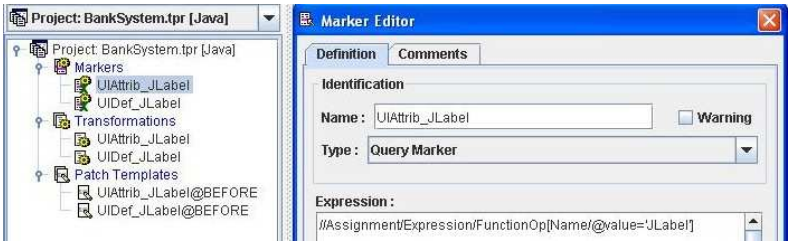

Fig. 2: Categorization rule example

The rules used in the categorization process are applied over the AST. They can be programming language specific or depend on previously categorized code and taking into account information obtained through dependency analysis.

We use L-CARE [3], a tool developed in ATX Software, to design the rules. This tool uses XPath [4] to query the AST and allows us to automatically annotate the source code using comments. L-CARE has been used over the last years in multiple large projects, which ensures the scalability of this step. In Figure 2 we show an example of a rule for Java that categorizes all attributes of type *JLabel* as belonging to the *User Interface* concern, since we know that *JLabel* is of this concern.

The source code annotation allows us to abstract to graph model level only the relevant information. This is extremely important to reduce the size of the source graph model and consequently making the transformation process much more efficient.

4 Redesign

The source graph model of Figure 1 is an abstraction of the code achieved through the categorization process. It keeps traceability to the code in order to facilitate the transformation / generation process and it is an instantiation of the metamodel that holds information about the source and target models.

Graph transformation rules are then applied to the source graph model in a fully automated way, transforming it into the intended target architecture.

4.1 Type Graph

To take advantage of graph transformation rules in the transformation process, we developed a type graph which is part of the *metamodel* present in Figure 1.

The model that we are using has the goal of being flexible enough so it can be instantiated by any OO application regardless of the specific technology. This way there is a better chance that it can be reused for different instantiations of our methodology. The high level view of the model can be seen in Figure 3. The *CodeFragment* package is used to represent code elements and is an extension of the type graph presented by Mens *et al* in [5]. This extension was necessary in

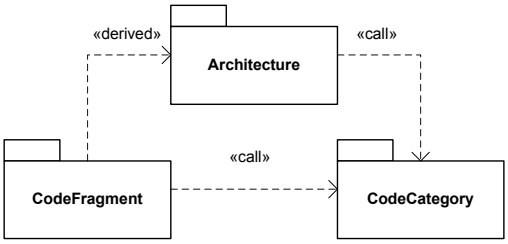

Fig. 3: High level view of the type graph

order to introduce classification attributes and the notion of code blocks, needed because the code categorization requires finer granularity than that of methods. Package *Architecture* includes the concepts of *Component* and *Connector* that allow us to represent the mapping between the programming language elements and the architecture level. The code categories information is represented in package CodeCategory.

Since it is necessary to keep traceability to the code in order to facilitate the transformation / generation process, a method to associate it to the type graph had to be considered. Given that we want to be as language-independent as possible we did not link the type graph directly to the source code but used instead an attribute (ASTNodeID) to associate its elements to the AST of the program.

4.2 Transformation Specification

Our use of graph transformation rules to describe model transformations is similar to what is being used in refactoring research [6]. However, refactoring rules are not enough for all reengineering purposes because sometimes it is necessary to perform transformations that are not completely behavior-preserving. An example of this is when we want to transform a legacy client-server system into a web-based application. The UI has to be changed because of the differences in the user communication paradigm between these different architectures. Another major difference is that our transformations are code category oriented, thus allowing architectural modifications.

An example of transformation rule specification is the *Move Method UI* rule. This rule searches for occurrences of methods classified as UI that are contained in classes that are classified as non UI (for example: having multiple concerns). The result is that those methods are moved to the appropriate UI classes. A small example of this rule application is presented in section 4.3. To specify the rules we are using the Tiger EMF Transformation tool [7], an Eclipse plugin.

4.3 Transformation Execution

For our instantiation of the reengineering methodology we are using code generated by the transformation specification in Tiger.

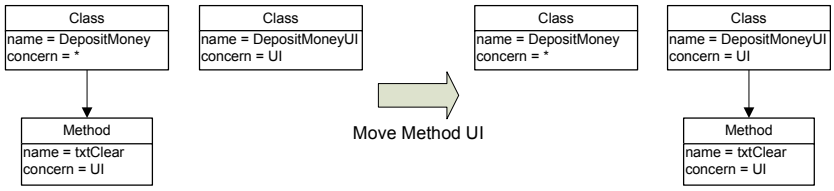

Fig. 4: Graph models before (left) and after (right) the application of rule Move Method UI. The graphs were simplified for readability. (The architectural elements are not shown and, in reality, the attribute "concern" does not exist directly in the code elements but is used as an association to code category elements.)

The transformation execution applies the rules defined in the transformation specification to the source graph model to obtain the target one.

Part of an example of the source graph model can be seen in the left side of Figure 4. The value "*" for the attribute "concern" means that the element contains more than one concern. For example, class "DepositMoney", even though it is not visible in the simplified figure, contains methods that belong to different concerns. This occurrence constitutes a potential candidate for the application of the transformation rule *Move Method UI* previously described. When we apply it, the method "txtClear" is moved from the class "DepositMoney" to "DepositMoneyUI", a class belonging to the UI concern as shown in the right side of Figure 4. This transformation is an example of a rule that contributes to the layering of the application.

4.4 Constraints

The global constraints, in our approach, are imposed by the type graph. This way we ensure that source, intermediate and target models are compliant to the general requirements.

In order to assure that the target model complies to the desired architecture, we define target constraints over the metamodel that correctly reflect the architectural paradigm. For instance, in 3-tier applications, there should be no UI and Logic layer methods in the same class and no direct links from UI to Data These constraints can be defined as graph transformation rules.

5 Related Work

Program transformation can occur in different levels of abstraction. The source-to-source level of transformation is the most established one, both in research and in industrial implementations. There are several research ideas that led to successful industrial tools. Examples from research include TXL [8] and ASF+SDF [9]. DMS from Semantic Designs [10] and Forms2Net from ATX Software [11] are program transformation tools being successfully applied in the industry. Transformations at the detailed design level, due to its applications as maintenance

techniques, have an increasing interest that is following the same path. Practices such as Refactoring [12] are driving the implementation of functionalities that automate detailed design level transformations. These are mainly integrated in development environments as is the case of Eclipse [13] and IntelliJ [14]. However, there is still a lot of ongoing research in this area, for instance, the work of Mens *et al* in the determination of dependencies between refactorings [5]. At the architectural level of program transformation there is some important research, e.g. the work in the Software Engineering Institute of CMU [1], but the industrial cases have been limited to specific source and target architectures and programming languages.

6 Conclusion and Future Work

Most of the ongoing research in the context of automated software transformation, as well as existing industrial tools, focus on textual and structural transformation techniques that intend to solve very specific problems within well defined domains (e.g. program restructuring, program renovation, language-platform migration). Our experience indicates that such techniques fall short of addressing in a systematic way the complexity of the architecture-based transformation problem. In practice, when such a problem arises, these approaches have to be combined in a trial and error fashion, the success of which often depends on the experience of the reengineering team and on the specific problem at hand. On the other hand, there exist techniques and tools that work well at an architectural level, but with the main goal of documenting and visualizing the architecture of applications rather than supporting increased levels of automation in architecture-based transformations. Although such tools can provide a very good starting point and facilitate the subsequent effort, in industry projects a reengineering approach that starts with redocumenting architectures is often too limited given the time and budget constraints.

In this work we have presented a systematic approach in order to explicitly address this issue. This paper focused on the code annotation and model transformation techniques to obtain the target architecture. The use of code category-driven graph transformation rules provides us several benefits, including: abstraction from programming language specifics (languages of the same paradigm share most of the same implementation), description of the transformations in a more intuitive way than that of code level transformations, and possibility of using existing tools for the tranformation execution and for analysis over the models (e. g. constraint checking).

Presently we are in the process of completing the tools in order to apply them to a large real-world scenario. This way, it will be possible to test a good set of categorization and transformation rules and see if more need to be developed.

Acknowledgments

R. Correia and C. Matos are Marie-Curie Fellows seconded to the University of Leicester as part of the Transfer of Knowledge, Industry Academia Partnership Leg2Net (MTK1-CT-2004-003169). This work has also been supported by the IST-FET IP SENSORIA (IST-2005-16004).

We would also like to thank L. Andrade and G. Koutsoukos (ATX Software) for their contribution in the development of the overall methodology. M. El-Ramly contributed to this work while lecturer at the University of Leicester.

References

1. Kazman, R., Woods, S., Carrière, J.: Requirements for integrating software architecture and reengineering models: CORUM II. In: WCRE '98: Proceedings of the Fifth Working Conference on Reverse Engineering, Washington, DC, USA, IEEE Computer Society (1998) 154–163
2. Mens, T., Demeyer, S., Janssens, D.: Formalizing behaviour preserving program transformations. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, Grzegorz Rozenberg, eds.: Graph Transformation. Volume 2505 of LNCS., Barcelona, Spain, Springer (2002) 286–301
3. ATX Software: L-CARE. http://www.atxsoftware.com/?sec=products&it=818
4. W3C: XPath. http://www.w3.org/TR/xpath
5. Mens, T., Taentzer, G., Runge, O.: Analyzing refactoring dependencies using graph transformation. Software and Systems Modeling (2007) To appear.
6. Mens, T., Eetvelde, N.V., Demeyer, S., Janssens, D.: Formalizing refactorings with graph transformations. Journal of Software Maintenance and Evolution: Research and Practice **17**(4) (2005) 247–276
7. Tiger EMF Transformation Project: Tiger EMF Transformation. http://tfs.cs.tu-berlin.de/emftrans
8. Cordy, J., Dean, T., Malton, A., Schneider, K.: Source transformation in software engineering using the TXL transformation system. Journal of Information and Software Technology **44**(13) (2002) 827–837
9. van den Brand, M., Heering, J., Klint, P., Olivier, P.: Compiling language definitions: the ASF+SDF compiler. ACM Transactions on Programming Languages and Systems **24**(4) (July 2002) 334–368
10. Baxter, I., Pidgeon, C., Mehlich, M.: DMS®: Program transformations for practical scalable software evolution. In: ICSE '04: Proceedings of the Twenty Sixth International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2004) 625–634
11. Andrade, L., Gouveia, J., Antunes, M., El-Ramly, M., Koutsoukos, G.: Forms2Net - Migrating Oracle Forms to Microsoft .NET. In Lämmel, R., Saraiva, J., Visser, J., eds.: Generative and Transformational Techniques in Software Engineering. Volume 4143 of LNCS., Springer (2006)
12. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Boston, MA, USA (1999)
13. The Eclipse Foundation: Eclipse. http://www.eclipse.org/
14. Jetbrains: IntelliJ IDEA. http://www.jetbrains.com/idea/