Scenarios for Mining the Software Architecture Evolution

Yaojin Yang
Nokia Research Center
P.O. Box 407, FIN-00045
+358718008000
yaojin.yang@nokia.com

Claudio Riva
Nokia Research Center
P.O. Box 407, FIN-00045
+358718008000
claudio.riva@nokia.com

ABSTRACT

In this position paper, we introduce our latest activities on architecture evolution analysis through software repository mining. The traditional approaches for software repository mining provide means for analyzing source-level information. However, we believe that software repository mining can also provide valuable results for analyzing the system evolution at the architectural level.

There are two challenges for analyzing the architecture evolution. The first one is to have in place a process for recovering the architectural models of the various releases. Architecture evolution is often visible only in the evolution of the implementation and this complicates the monitoring process. The second one is to have access to the past design models that were created by the architects during the design phase. A practical solutions for versioning the architectural models is not in use yet and this complicates the possibility of accessing the past design decisions.

Analyzing architecture evolution through software repository mining represents the most promising choice. In order to conduct the analysis through software repository mining, we introduce our meta-model covering the design and implementation spaces. Then, we define a set of scenarios that demonstrate the architecturally significant analysis that we can conduct by mining the software repository.

Categories and Subject Descriptors

D.2.11 Software Architectures, D.2.13 Reusable Software

General Terms: Documentation, Experimentation

Keywords: Architecture evolution, Mining software repository, Architecture recovery

1. INTRODUCTION

The software architecture evolution typically happens on two parallel tracks: the design space and the implementation space. While the evolution on the design space concerns the intentions of the designers, the evolution on the implementation space can have deep implications at the architectural level. From our experience, understanding the evolution at the implementation level can help to understand the evolution at the architecture level where it is harder to monitor and trace the changes. For this reason, it is not easy to keep the architecture models up to date. Therefore, providing support to deal with this issue is very important from the perspective of architecture evolution.

We provide an example of architecture evolution triggered by the implementation that is frequently happening in the lifecycle of software platforms like the ones developed in Nokia. We monitor and analyze such evolution through mining software repository.

We consider a *binary component* (like a DLL) that belongs to the implementation space. If the source files associated with the binary component have been modified, we say that the binary component itself is modified and it is evolving. The evolution of the binary component may have implications in the architecture space, i.e. in the architecture design of the system. If the modifications in the binary component affect the way the component interacts with the environment (e.g. using a new interface), we can say that also the *logical component* in the design space has changed and the architecture of the system has also evolved.

We highlight that there is not a direct link between the changes in the implementation with the changes in the architecture. Only some implementation-level changes have an effect on the architecture and we call them *architecturally significant*. The main focus of our work is to study the architecturally significant changes for a software system.

This is our approach for monitoring the architecture evolution by mining software repositories. First, for each release we build an implementation and design combined architecture model according to a defined meta-model by using our reverse architecting environment ([2] and [6]) and import the models into our software repository. Second, we compare the models' implementation spaces between release *2* and its previous release *1* through mining the software repository. Third, if binary component evolution is identified, we trace up to the design spaces of both releases *1* and *2* and identify the parent logical component of evolved binary component. Fourth, we compare the topology of the graphs based on the parent logical component between the models' design spaces of release 1 and release 2. If there are not identical, we claim that there is implementation trigged evolution happening on the parent logical component.

In [3], Koschke and Simon propose an approach to map the design and the implementation based on the same module viewtype. The difference with our work of building an implementation and design combined architecture model is that we do not assume that the viewtypes of design space and implementation space are the same. In fact, our design space is presented in component and connector viewtype and our

MSR'06, May 22–23, 2006, Shanghai, China.

implementation is presented in module viewtype. That is, our model is presented in a combined viewtype.

For supporting architecture evolution analysis through software repository mining, we utilize our existing reverse architecting environment [2] & [6]. The *reverse architecting tool set* offered by the environment provides us a tool chain from source code analysis till model abstraction for recovering architecture models. Columbus [5] tool is deployed in the tool chain as source code analyzer. *MySQL database* is integrated into the environment, which is used as our software repository for storing architecture models of different releases. The environment's *model validation tool* facilitates our comparison between architecture models of different releases. The architecture evolution is identified through the comparison.

2. META-MODEL OF ARCHITECTURE MODELS

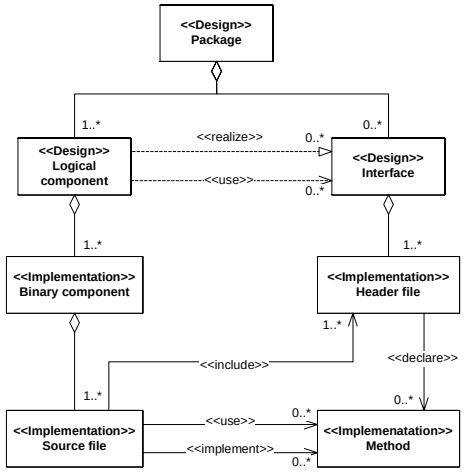

Figure 1. Meta-model for the architecture models

We have developed a simplified meta-model of the architecture that serves the purposes of studying the architectural evolution. The meta-model captures the design and implementation aspects, as shown in Figure 1. The meta-model provides a traceability mechanism between the architecture design and the implementation. This represents a key element for studying the software architecture evolution.

We make a distinction between the design and implementation space. In the design space, a *package* contains one or more *logical components* and a set of *interfaces*. The *logical components* implement the functionality of the system. They can realize or/and depend on any number of *interfaces*.

In the implementation space, the *binary component* is the aggregation of several *source files*. The *source files* include one or more *header files*. The *header files* declare the methods that are implemented in or used by a *source file*.

3. APPROACH FOR MODEL RECOVERY

In our architecture models, the instances of elements and relations presented in the meta-model are mostly captured or abstracted from implementation. However, instances of *aggregation* between *logical component* and *binary component* and *aggregation* between *interface* and *header file* are directly extracted from design.

We use Columbus for capturing model elements and relations in the implementation space. The initial resulting model conforms to the FAMIX meta-model [1] (Table 1). Then, we filter information that is not defined by the meta-model (Figure 1). Since, *binary component* and *aggregation* between *binary component* and *source file* are specified in specific project file, the Columbus has been customized in order to extract such information and import it into the implementation model.

In the design space, model elements of documented *logical component*, *interface* and *package*, *aggregation* between *package* and *logical component*, and *aggregation* between *package* and *interface* are captured by parsing architectural logical view and interface specification document.

In order to merge the implementation model and the design model to form a complete architecture model defined by the meta-model (Figure 1), *aggregation* between *logical component* and *binary component* and *aggregation* between *interface* and *header file* are the key relations to rely on. The *aggregation* between *logical component* and *binary component* can be obtained through parsing design tables and the *aggregation* between *interface* and *header file* can be obtained through parsing interface specification document. If certain *binary component* or *header file* doesn't belong to any *aggregation*, it is usually the case that new *logical component* or *interface* is added into the design but is not documented.

The *dependency* and *realization* between *logical component* and *interface* are considered as key measurements for monitoring implementation trigged architecture evolution. Therefore, they have to be abstracted from implementation.

Entities	
Entity	**Description**
`Class`	The definition of the class
`Method`	The definition of method of a class
`Attribute`	The definition of an attribute of a class
`Function`	The definition of a function or a procedure that has a global visibility
`Macro`	A C++ macro definition with `#define`
`TypeDef`	A C++ type definition with `typedef`
`GlobalVariable`	The definition of a global variable
`File`	A source file
`Directory`	A directory in the file system
`Package`	A Java package
Relations	
Relation	**Description**
has_method	A class declares a method
has_attribute	A class declares an attribute
has_class	A class declares a nested class
inherit	A class inherits from another class
invocation	A method or a function invokes a method or a function
access	A method or a function access an attribute or a global variable
include	A source or header file includes an header file
expansion	A function or a method expands a macro
use_type	A function or a method use a user's defined type
contain	A file contains the definition of a class, a macro, a type definition and a global variable
implement	A file defines the implementation of a method
decl_fn	A file declares a function
def_fn	A file defines a function
contain_file	A directory contains a file
contain_dir	A directory contains another directory
pkg_contain	A Java package contains a class

Table 1. The FAMIX meta-model

4. CHARACTERIZING THE EVOLUTION OF THE SOFTWARE MODELS

During the evolution of the software system, both the design and the implementation spaces are modified. The design space is modified by the software designers according to the requirements of the system. The implementation space is modified by the programmers who are implementing new features or modifying the existing code.

The implementation is driven by the design but not all the changes in the implementation are reflected by the design (as we have discussed in [2]). Moreover, the versioning of the design models is not yet well understood and the practice shows that tracing the modifications from one design to the next one is not an easy task. As a result, the only reliable information about the evolution of the system is mainly visible in the implementation space.

We need to link the evolution in the design space with the evolution in the implementation.

We characterize the evolution of the system by only comparing the topology of the software models.

4.1 The evolution of the implementation space

Changes in the implementation space are identified by changes in the topology of the graphs that we extract with the source code analyzers.

A *build component* is changed when one of the following items has been modified between two different releases:

- The set of *source files* that belong to the *build component*
- The set of *use* relations between a *source file* and a *method*
- The set of *implement* relations between a *source file* and a *method*
- The set of *include* relations between the *source files* in the *build component* and the *header file*

A *header file* is changed when the method declarations have been modified (e.g. when method declarations have been added, removed or changed).

We note that we ignore those modifications that do not modify the topology of the graphs.

4.2 The evolution of the design space

We directly link the evolution of the elements in the design space with the modifications that happen in the implementation space. We define the following rules:

1. One *logical component* is changed when at least one of its *binary components* have been changed, removed or new ones have been added.
2. One *interface* is changed when at least one of its *header files* have been changed, removed or new ones have been added. It is not possible to freeze the interfaces but they can evolve in the same way like components.
3. One *package* is changed when (1) one containing element (either a component or an interface) has changed, (2) a new element has been added or (3) an existing element has been removed.

5. SCENARIOS OF ARCHITECTURE EVOLUTION ANALYSIS

We present common scenarios of architecture evolution analysis. The scenarios are listed according to their impact on the overall architecture (from high to low impact). In Table 2, we analyze the relations between the types of architecture evolution and scenarios.

Adding one new feature

One typical scenario is to add a new feature in the system. This activity typically involves creating new interfaces, modifying existing interfaces and introducing new logical or/and binary components. Adding a new feature can have a large impact on the overall architecture.

The modification of the existing interfaces may impact the functionality of existing binary components. However, not all the cases can be predicted at design time and only during the implementation problems may arise. After the implantation, it is important to monitor what are the effects of these changes on the overall design.

The new logical or/and binary components may also create unexpected dependencies that are discovered only during the implementation. It is important to detect these architectural changes.

Restructuring the design

Improving the overall design is a preventive maintenance activity that can have large impacts on the system. The designers should be able to monitor how these activities can affect the various logical or/and binary components, how often they are happening and if certain logical or/and binary components are often modified.

Modifying one binary component

When one binary component is modified to extend its functionality the changes may impact other binary components or/and even logical components. It is important to control the effects of the changes.

Studying the evolution of one logical component

Studies on the evolution of particular logical components are typically conducted to assess their quality, stability and to identify the weaknesses. By studying the evolution at the architectural level we may be able to reveal unfavorable patterns of evolutions like too frequent changes or changes with too big effects on the rest of the system. The studies may lead to restructure the logical component.

Monitoring the evolution of the interfaces

Interfaces cannot be frozen but they are evolving. In most cases, new interfaces are added to provide access to new functions. Ideally interface should be kept stable, but modifying interfaces is sometimes required by the modification of implementation. However, removing interface is not a common case.

Fixing a bug in the system

We expect that bug fixing is not causing big impacts at the architectural level. Bug fixing should only be limited by modifying the internal implementation of the binary components but not their external dependencies. If there are architectural modifications happening because of bug fixing, then there may be fundamental problems in the implementation and the current design should be revisited.

Correlating software metrics with the architectural evolution

Several software metrics are calculated by the Columbus tool. We need to correlate the trends of the software metrics with the changes in the architecture. This will enable us to monitor the effect of certain architectural changes on the quality of the software.

	Design trigged evolution	Implementation trigged evolution
Adding one new feature	**Happen**	**Maybe happen**
Restructuring the design	**Happen**	**Maybe happen**
Modifying one binary component	**Not happen**	**Happen**
Studying the evolution of one logical component	**Happen**	**Maybe happen**
Monitoring the evolution of the interfaces	**Happen**	**Not happen**
Fixing a bug in the system	**Not happen**	**Maybe happen**
Correlating software metrics with the architectural evolution	**Happen**	**Happen**

Table 2. Types of architecture evolution in the scenarios

6. CONCLUSIONS

In this paper, we presented our work of analyzing software architecture evolution through mining software repository. Our position is that the software repositories contain valuable information for monitoring the architecture evolution but this information is not ready available. In order to make it more explicit, we need to isolate the architecturally significant changes that happen in the implementation and have an impact on the design space.

In the future, we will focus on providing concrete examples of analysis of the evolution of very large software systems (containing tens of millions of lines of code). One of our goal is to define architecture evolution patterns, especially implementation trigged evolution, to provide tool support for monitoring and analyzing the evolution, and ultimately to refine our architecture design so that the implementation evolution will have minimum impact on the architecture level.

7. REFERENCES

[1] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 – the FAMOOS information exchange model. Technical report, University of Bern, 2001.

[2] C. Riva, P. Selonen, T. Systäa, and J. Xu. UML-based reverse engineering and model analysis approaches for software architecture maintenance. *In Proc. of the The 20th IEEE International Conference on Software Maintenance*, Chicago, Illinois, USA, September 11th - 17th 2004. IEEE Computer Society, 2004.

[3] R. Koschke R. and D. Simon, Hierarchical Reflexion Models, *In Proc. of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, 13-16 November 2003, Victoria, Canada, IEEE Computer Society Press, 2003, 36-45.

[4] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. *In Proc. of 4th Working IEEE / IFIP Conference on Software Architecture (WICSA 2004)*, 12-15 June 2004, Oslo, Norway, pages 122–132. IEEE Computer Society, 2004.

[5] Ferenc, R.; Beszédes A.: Gyimóthy T., Extracting facts with Columbus from C++ code, *In Proc. of Proceedings. 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, Tampere, Finland, March 24-26, 2004.

[6] Riva, C.; Selonen, P.; Systa, T.; Tuovinen, A.-P.; Xu, J.; Yang, Y., Establishing a software architecting environment. *In Proc. of the 4th Working IEEE / IFIP Conference on Software Architecture*, Oslo, Norway, July 12-15 2004.