Abstract

In this paper, we propose a framework for reconstructing software architecture from a software system. The framework consists of the steps to reconstruct software architecture for modifying source code. The framework can be used for developers to analyze source code and reconstruct software architecture for modifying the code.

In addition, we perform a case study by analyzing Reflexion Model, an open source code, to verify that the framework can be effectively applied to software architecture reconstruction. For that, we evaluate several tools in order to select those that support our reconstruction. Then we verify the framework and report several issues discovered with the case study. By applying the issues encountered in the case study to the framework, we have made our framework a more practical one that developers can easily understand and adopt.

1. Introduction

Software Architecture Reconstruction has become the main stream of the reverse engineering research area [1, 5]. Architecture reconstruction is a reverse engineering activity that aims at recovering lost design decisions. Robust and clear software architecture is often the key determinant of the success or failure of many software projects [5]. Software architecture reconstruction is especially important in such cases. Some researchers have developed tools for software architecture reconstruction [5, 7]. Other researchers, realizing that software architecture reconstruction is not as simple as traditional reverse engineering of source code, have suggested their frameworks for software architecture reconstruction [4, 7].

However, there is no systematic and concrete guideline for reconstructing software architecture in order to modify source code. Specially, open source systems often have no accurate architecture documentation, and no architects to interview. Also, analysts may have no domain knowledge. Furthermore, the reconstruction of software architecture is not only to generate simple architecture views [2], but also to discover the design decisions, and to provide information for redeveloping the software. In order to understand the software architecture, the developers should know the domain model and the technology of the software, as well as its quality attributes and functional requirements. Hence we propose a software architecture reconstruction framework for modifying source code, and verify the framework by analyzing a software system according to the prescribed steps.

The remainder of the paper is organized as follows: Section 2 reviews the previous studies. Section 3 defines the steps to reconstruct software architecture for modifying open source. Section 4.1 evaluates tools that support the software architecture reconstruction steps. Sections 4.2 and 4.3 conduct a case study of Reflexion Model, an open source program. Section 5 reports the verification result from the case study. Finally, Section 6 summarizes the contributions of this paper and describes future research directions.

2. Related Works

In this section, we consider the previous research on software architecture reconstruction. Section 2.1 surveys the existing frameworks, which define the steps for reconstructing software architecture. Section 2.2 examines several case studies, which offer lessons learned from the analysis of software systems.

2.1 Frameworks for Software Architecture Reconstruction

Two research groups have proposed interesting frameworks for software architecture reconstruction. One group, Nokia, suggested Symphony, the reference framework to compare all activities related to software
architecture reconstruction. The other group, SEI, suggested QADSAR, the framework defining the steps to reconstruct architectural view from source code.

Symphony [4, 5] outlines a view-driven software architecture reconstruction. Symphony consists of two phases. The first phase analyzes the problem for which architecture is needed and defines viewpoints and their mapping from source code. The second step extracts and analyzes information, applies mapping and creates views. Symphony provides a reference framework to find and demarcate research problems in software architecture reconstruction. However, it does not provide a practical guideline to reconstruct software architecture for specific situations [3].

The QADSAR approach [6, 7] has five steps: scope identification, source model extraction, source model abstraction, element and property instantiation, and quality attribute evaluation. QADSAR is valuable because it is a systematic way to introduce a quality attribute driven perspective to software architecture reconstruction. However, QADSAR does not reflect industrial experience in which developers analyze existing source code similar to their target system before developing a new one. Thus, case studies that reconstruct software architecture [8, 9] should reflect a framework that reconstructs software architecture so that the framework could be applied more practically.

2.2 Case Studies of Software Architecture Reconstruction

Two case studies are particularly relevant to our research. The first one is a case study of Linux kernel [8], which distinguishes architecture in the developer’s mind from architecture built-into a system and shows the differences between them. The second one is a case study of Apache [9], which shows how students who have no domain knowledge learned Apache software architecture in a course.

Bowman, Holt and Brewster [8] performed a case study on extracting architectural documentation from Linux kernel. Bowman et al.’s approach is as follows. First, examine the existing documentation. Second, group source files into subsystems, based on directory structure, naming conventions, and code comments. Third, extract relations between the source files. Fourth, determine relations among subsystems. Finally, use the relations to form software architecture of the system. Since these steps are useful to recover design decisions for modifying a software system without an architect to interview, we incorporated these steps into our framework for reconstructing software architecture.

Gröne, Knöpfel and Kugel [9] summarized the results of one-semester course of source code analysis on the structure of Apache 1.3. The students, who had no domain knowledge about the system, followed the steps below and presented the architecture successfully at the end of the course. The first step is to define the purpose of the analysis and explain key concepts of the system. The second step is to gather domain knowledge, understand the system and model the conceptual architecture of the system. The third step is to understand the function, the configuration and handling of the software product. The final step is to understand the implementation of the software product. We used these steps to establish our own framework for people having no domain knowledge of modifying open source.

3. A Software Architecture Reconstruction Framework (SAROS)

In this section, we outline our framework of software architecture reconstruction for modifying open source (SAROS). We applied the following principles to our framework. First, the base framework is Symphony because Symphony embraces both the preparation and the execution of the reconstruction. Reconstruction Preparation is required because developers understand problems of existing software and a mechanism of the software in this phase. Second, the reconstruction execution of our steps is based on QADSAR because QADSAR has the concrete steps for building architecture views. The third principle is to reflect the activities of case studies, such as Gathering Domain Knowledge, and Understanding the Configuration. These activities define more effective steps for analyzing a software system without having to interview architects.

Figure 1. Our Software Architecture Reconstruction Framework for Modifying Open Source (SAROS)
Figure 1 depicts our Framework, SAROS. Steps for reconstructing software architecture are grouped under Reconstruction Preparation and Reconstruction Execution. Through the process, an analyst reconstructs software architectural views and determines what parts to modify and how to modify them. In the framework, an output of Reconstruction Preparation becomes an input of Reconstruction Execution. A scenario, describing a current status and situation to be improved in the Reconstruction Preparation phase, should be an input to the Reconstruction Execution phase, in order to determine which execution trace should be recorded in the Dynamic Reconstruction View task. Conceptual Architecture, defined in the Reconstruction Preparation phase, should be an input to the Reconstruction Execution phase, in order to customize a module view of an abstract source model in the Static View Reconstruction task. In Reconstruction Execution, Static View Reconstruction analyzes source code and generates a Code View and a Module View. The final result of Static View Reconstruction becomes an input of Dynamic View Reconstruction in order to aggregate detailed modules to abstract modules of Concrete Architecture. Finally, Reconstruction Execution produces the three views of Code View, Module View and Execution View [2, 10].

4. A Case Study: Reflexion Model Program

We performed a case study according to the framework in Section 3. The purpose of the case study is to verify if the framework can be applied to software architecture reconstruction. In addition, the case study provides guidelines for developers to understand and adopt our framework. We selected the Reflexion Model [13, 14, 15] as our target system because it was a moderate-size software and appropriate for a short-term case study. The software was implemented in Java 1.3 and Eclipse 2.1.1 on MS Windows. It consisted of 76 files and the total number of lines was 13,996 and the ratio of comments to code was 0.34.

4.1 Tool Selection for Our Case Study

We selected tools prior to conducting our case study. We chose the candidate tools by considering the environment of our target software. After reviewing candidate tools that could be applied to Java languages for static analysis, we selected Doxygen [16], Together [17], and Reflexion Model [14] for our case study. We excluded other tools because there were no free trial versions, a tool does not provide convenient graphic user interface, or because the selected tools provide more appropriate information.

After reviewing candidate tools that could be applied to Java languages for dynamic analysis, we selected only HPROF [17]. Dynamic analysis tools are very sensitive to their environment and more difficult to obtain than static analysis tools. Almost all advanced tools such as Jinsight [18], Shimba [19] and AVID [18] were unavailable to us. Other tools [20] generally do not support the platform and language of our case study, leaving us no alternative other than HPROF.

4.2 Reconstruction Preparation

The Reconstruction Preparation phase analyzes problems for which architecture is needed and defines the conceptual architecture of the system [4, 5, 8, 9]. Reconstruction Preparation has two tasks: Problem Definition and Concept Determination.

4.2.1 Problem Definition

The purpose of Problem Definition is to analyze and discuss the problem triggering the reconstruction with stakeholders. It consists of two steps, Purpose Description and Scope Identification.

① Purpose Description: In this step, an analyst studies a system to find a compelling reason to start a reconstruction. For that, we collected and read technical papers about Reflexion Model [14, 15] to learn how to operate the system. Then we prepared the environment and installed the software. Next we used the software to analyze a compiler translating TTCN code to C Code. Finally, in the experiment we discovered better ways to use the software [16].

② Scope Identification: In this step, the analyst identifies quality attribute scenarios which are grounds for finding the part to be reconstructed. For that, we prioritized the five improvement items and selected the first one: supporting hierarchical high-level modeling. Then we specified the use-case for supporting hierarchical high-level modeling in Figure 2.

When defining high-level model, the user chooses a rectangle icon from the menu of the high level model. Then, the user draws a rectangle representing the upper modules embracing lower modules shaped in ellipses. When defining map, the user adds the mapping relationships between the lower modules and the upper modules to the map. When computing map, the tool computes hierarchical mappings between several layers, and then presents the hierarchical model.

Figure 2. Scenarios
4.2.2 Concept Determination

The purpose of Concept Determination is to identify the architecture concept and a recovery strategy that are relevant to the problem. It consists of Conceptual Architecture Identification and Understanding the Configuration.

① Conceptual Architecture Identification: In this step, the analyst draws the conceptual architecture by studying technical papers. We drew the conceptual architecture (Figure 2) that represents the main functions of the Reflexion Program. The Extraction part parses the source code and extracts detailed source information from the code. The Modeling part supports user drawing the high level model, then converts the diagram into a text file. The Mapping part parses the text file that holds the information of the high level model and mapping rules. The Calculation part maps the relationships among source elements to the relationships among modules of high level model. The Presentation part shows the result of the calculation in the graphic form.

![Figure 3. Conceptual Architecture of Reflexion Model](image)

② Understanding the Configuration: In this step, we learned how to configure and administer the product and how to utilize the API. We had to study and experiment with the mechanism and graphic packages for Eclipse plug-in software before we started configuring. After that, we imported the source code as an Eclipse plug-in project into the Eclipse tool, and created the executable file of the Reflexion Model. For studying the API, we analyzed the main file and grasped the four classes connected to the user interface of the RM program. We caught the four classes as starting points of the analysis of the RM source code.

4.3 Reconstruction Execution

The Reconstruction Execution phase extracts and analyzes information, defines and applies mapping between source and conceptual model, and creates views. Reconstruction Execution has two tasks: Static View Reconstruction and Dynamic View Reconstruction.

4.3.1 Static View Reconstruction

The purpose of Static View Reconstruction is to analyze code and build the concrete architecture in a module view [2, 8, 9, 13, 14] from the source code. It consists of three steps: Static Model Extraction, Static Model Abstraction, and Implementation Analysis.

① Source Model Extraction: In this step, an analyst extracts source elements from available source code files. For that, we executed Doxygen that showed a directory structure, a list of files, classes, etc. Then, we edited the result and made Table 1, which includes the list of files. Next, we executed Together to grasp class diagrams. However, those tools did not extract call relationships among source elements, except for inheritance relationships. We noticed some tools extracting and abstracting the relationships at once. Thus we passed the extraction of concrete relationships from this extraction step to the next abstraction step and expected abstraction tools to help.

<table>
<thead>
<tr>
<th>Table 1. Code Structure of Reflexion Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Doxygen/RM/jntool/sec/ca/ubcs</code></td>
</tr>
<tr>
<td><code>- compute </code>java (9)<code> </code>spline <code>java (3)</code></td>
</tr>
<tr>
<td><code>- eclipse </code>graph <code>plugin </code>java (15)`</td>
</tr>
<tr>
<td><code>- gui </code>plugina <code>java (1)</code></td>
</tr>
<tr>
<td><code>- map </code>java (1)<code> </code>model <code>java (3)</code></td>
</tr>
<tr>
<td><code>- struct </code>java (5)`</td>
</tr>
<tr>
<td><code>- util </code>java (10)`</td>
</tr>
<tr>
<td><code>- sugiyama </code>algorithm <code>java (12)</code></td>
</tr>
<tr>
<td><code>- model </code>java (4)<code> </code>java (4)`</td>
</tr>
</tbody>
</table>

② Source Model Abstraction: In this step, the analyst should identify and apply aggregation strategies. We utilized the RM for showing the tentative concrete architecture that exhibits actual relationships among modules. We began with the directory structure, and modified mapping rules several times until we had captured the proper mapping between the conceptual architecture and source elements. Finally, we were able to create concrete architecture (Figure 4). A number on an arc between two modules represents the number of call relationships between those two modules. A difference between our conceptual architecture (Figure 3) and the concrete architecture (Figure 4) is that there are no definite call relationships between the Calculation module and the Extraction, Modeling, and Mapping modules in the concrete architecture; The symbols, XYZ, denote no call relationships among the modules in Figure 4. Since our target source has graphic user interface and event handling mechanisms, the relationships between events are implicit.
1. Implementation Analysis: In this step, the analyst elicits detailed information about the software product that is required for reconstruction. Our purpose for the modification was to enable the Reflexion Model to support hierarchical high-level modeling. For that, we had to identify parts of source code to be modified. Then we examined the source code and made a table of classes and functions which may contain important information.

2. Dynamic View Reconstruction

   The purposes of Dynamic View Reconstruction are to analyze execution records of software and construct an abstract execution model in a C&C view [2, 11, 12, 13]. It consists of the three steps: Execution Model Extraction, Execution Model Abstraction, and Execution Analysis.

   1. Execution Model Extraction: In this step, the analyst extracts the real interaction of software elements from software execution. For that, we selected the third scenario described in Figure 2. Then we extracted a trace file from the execution of the function, “compute Reflexion Model,” by using HPROF. As a result, we created a trace file with 150,796 lines. The trace data was not accurate in the call relationships between functions because HPROF just recorded execution order and did not mark when a function starts and finishes.

   2. Execution Model Abstraction: In this step, the analyst abstracts details of real interaction of software elements. We introduced the mapping rules of the “Source Model Abstraction.” According to the mapping rules, we abstracted the trace data of the “Execution Model Extraction” to the upper level, as in Figure 6. A number on an arc between two modules represents the number of execution orders between two modules. Then we discovered that the modules that had showed no relations in the static analysis showed some relations in the dynamic analysis. We analyzed an execution order of Reflexion Model by importing and manipulating the trace data in MS Excel. As a result, we discovered that the implicit execution order of modules in a Static View became explicit, and we were able to see the flow from extraction to presentation.

   3. Execution Analysis: In this step, the analyst elicits detailed information about the software system. First, we identified what we need to know in the source code, referring to the concrete scenario from the Problem Definition task, “Computing Reflexion Model.” Then, we checked which parts are affected by the scenarios when we modify the calculating algorithm to support hierarchical mappings among several layers. Finally, we examined the source code and found the important parts related to the modification requirements.

5. Verification Results

   We verified our framework with a case study in order to check that the framework is reasonable and feasible. Two phases, brought from Symphony, helped us arrange the tasks of software architecture reconstruction, without omitting additional tasks such as defining the purpose and studying the software system. A difference between our framework and Symphony is that our framework arranges the static analysis and the dynamic analysis sequentially while Symphony performs the two tasks as one. The case study assured us that the two tasks complement each other, but are not selectable for an analytical purpose. In some minor aspects, though the analysis tasks define an extraction step before an abstraction step, similarly with QADSAR, the tool supporting the steps has integrated the two steps. Thus, one of a user’s main tasks is to define the mapping between source code and conceptual architecture. In addition, finding appropriate mapping between the model in our mind and the model in source code was done iteratively. With respect to tool application, we noticed that current tools were not mature enough to support analysis and abstraction of an execution flow of a software system. Therefore, we need to build such a tool in order to apply our framework to practical situations.
6. Conclusions

In this paper, we have proposed our framework for reconstructing software architecture from open source (SAROS). We utilized Symphony as the basic framework, and then incorporated QADSAR into it. We also integrated the procedures used in several case studies of software systems. SAROS can be a guideline for developers to analyze source code and reconstruct software architecture. Since some tasks of software architecture reconstruction are tedious and repetitive, we examined tools in our case study. Then we verified our framework (SAROS) through a case study of Reflexion Model, an open source software system. During the case study, we found and repaired several defects in the early versions of SAROS. In addition, the case study added a practical guideline that helps developers understand and utilize our framework by showing a concrete example of a software system. Developers can find a conceptual model, a concrete model and several views embodied in the case study.

We propose the following future research directions. First, we should survey tools more thoroughly and find a way to acquire several advanced tools. There are many reverse engineering tools, but the tools we used were limited to Java language. Many research papers report the result of reverse engineering in Java. However, the tools discussed in those research papers have not been published yet. This has limited our research. Second, we should perform more case studies in order to consolidate our framework. Our case study of Reflexion Model targeted a specific environment and a specific subject. The program was an Eclipse plug-in, analyzing source code and presenting a high level model. We need to consider other subject areas. Performing several case studies for a variety of subjects would prove that our framework is indeed widely applicable. In the long term, we want to develop our own toolset to reconstruct software architecture for modifying source code that is based on the SAROS framework.

7. References