A Framework for Software Architecture Recovery

Wolfgang Eixelsberger
Lasse Warholm
ABB Corporate Research
Bergerveien 12
N-1361 Billingstad, Norway
+47 6684 3060
{wolfgang,lasse}@nocrc.abb.no

Rene Klösch
Harald Gall
Berndt Bellay
Technical University of Vienna
Distributed Systems Department
Argentinierstrasse 8/184-1
A-1040 Vienna, Austria
{gall,kloesch,bellay}@infosys.tuwien.ac.at

1. ABSTRACT

The recovery of „higher-level“ representations from given source code of an existing software system is important for the development of program families. Therefore, we evaluated current reverse engineering technology to which extent and how architectural elements can be identified in a software system. The architecture recovery framework we discuss in this paper is ongoing research work within the ESPRIT project ARES (Architectural Reasoning for Embedded Systems). [1]

2. INTRODUCTION TO ARCHITECTURE RECOVERY

2.1 Definition of software architecture and architecture recovery

Software architecture is a relatively new research area and, hence, no widely accepted definition exists. Software architecture definitions in general contain three elements: components, connectors and rationals, such as e.g. ‘The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.’ [1]

Since software architectures determine the gross structure of a system, architectural representations enable software developers to explicitly describe, assess, and manage architectures of embedded software systems.

We define architecture recovery as a process of identifying and extracting higher level abstractions from existing software systems [2].

2.2 Architecture representation/properties

An architectural representation consists of structural and non-structural information about a software architecture. Structural information are components and connectors describing the configuration of a system and non-structural information are architectural properties. Architectural properties are for example, safety patterns, communications patterns, or application domain guidelines and constraints.

2.3 Related Approaches

In [3] an architecture modeling language is used to codify specific architectural styles and a recognizer tools tries to identify instances of such styles. The basic mechanism used is program slicing as defined in [4]. In contrast to these approaches we use commercially available reverse engineering tools together with application domain knowledge introduced by a human engineer to complete the automatically recovered software views by manually generated ones.

3. FRAMEWORK FOR ARCHITECTURE RECOVERY

3.1 Introduction to the framework

In many cases, architectural information is available as block-line diagrams. However, most architectural information is inherent and hidden in different views as source code or design documentation. The extraction of architectural information from these different views, therefore, is required in a structured manner.

Figure 1 sketches an overview of the proposed framework for an architecture recovery methodology. The input of the recovery process is the source code, the design documentation, and domain knowledge.

Information from the source code can be extracted with the help of reverse engineering tools and by manual recovery. Reverse engineering tools perform static analysis on the code and extract information like call graphs, cross reference tables, and data flow diagrams. Human interaction is not possible while the tools are analyzing the source code.

[1] ARES is supported by the European Commission under ESPRIT framework IV contract no. 20477 and is pursued by Nokia RC Finland, ABB Norway, Philips RC Holland, Imperial College, Technical University Madrid, and Technical University of Vienna.

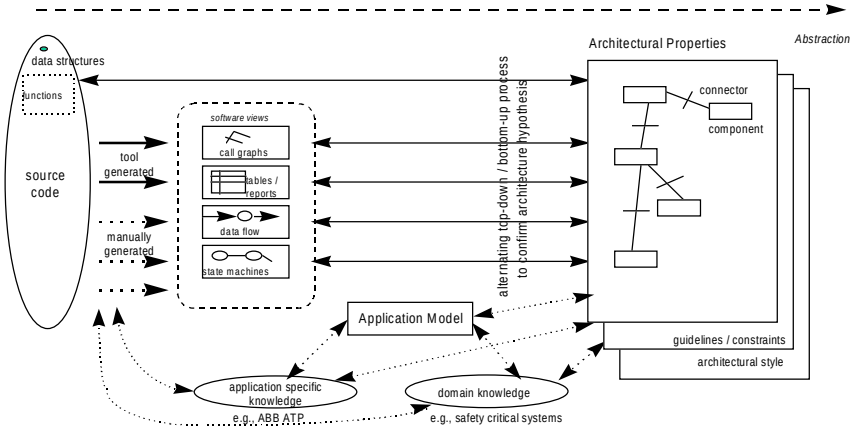

Figure 1: Framework for Architecture Recovery

Reverse engineering tools provide a higher level of abstraction since information that is not of interest for the specific view is excluded. The results of reverse engineering tools have to be analyzed by human experts and do not represent architectural representations per se.

Manual recovery is performed on the source code by human experts. Human experts, especially domain experts, can analyze the source code using knowledge that is not available for reverse engineering tools. Such knowledge includes information about domain knowledge, high-level design decisions, coding standards and system requirements.

The recovery of design documentation and domain knowledge delivers additional information into already existing abstractions such as data flow diagrams and supports the generation of additional software views, for example, state transition diagrams. The extracted information still is not considered to be an architectural representation of the system under study.

Based on our experience in object-oriented re-architecturing, we determined the following strategy towards architecture recovery:

1. Studying the application and domain-specific knowledge (such as standards of the domain and domain-specific rules and constraints)
2. Forming an architectural hypothesis regarding the system and its structure
3. Verifying and refining the architectural hypothesis against the software system under study.
4. Generating different software views of the examined system.
5. Use of reverse engineering tools.

3.2 Evaluation of reverse engineering tools

In order to identify which software views can be automatically generated and what tasks for the architecture recovery can be done by tools several reverse engineering tools were evaluated. The evaluation is based on the reverse generated software views of a case study from ABB and their usability for architecture recovery. The capabilities of the reverse engineering tools were assessed in providing means both for recovering architectural elements and for generating higher-level abstractions from given source code.

3.2.1 Difficulties with the case study

The difficulties with the case study resulted mainly from three aspects which are typical for real-world applications:

First, in real-world applications usually more than one programming language is used. In the case study reported in this paper C and Assembler were used. The reverse engineering tools evaluated are not able to parse Assembler code so only the C parts could be parsed and studied. This results in incomplete views of the application.

Furthermore, the C code contained externally defined functions and variables which are defined and instanciated in the assembler part. As a result some reverse engineering tools did not include these variables and functions in the generated reports. These variables are a serious problem because they were used for the main data flow in the application.

The case study was created for different development and target environments and was reverse engineered on another environment. This created some problems parsing the C source code: To parse the C code with the

reverse engineering tools platform dependent parts of the source code had to be supplied (e.g. header files) and platform properties had to be taken into account (e.g. file naming conventions, interrupt usage). Furthermore, application specific knowledge was required to parse the application in a useful way (e.g. versions of the application). Macro definitions and additional files were used to generate different versions of the application.

The generation of useful software views with the reverse engineering tools also requires application specific and domain knowledge. This mainly results from the size of the application and different aspects of interest (e.g. error routines that resulted in clustered graphical views).

3.2.2 Overview of the reverse engineering tools

The reverse engineering tools were chosen to represent a wide spectrum of tools. The evaluated tools used on the case study were:

- Refine/C (Reasoning Systems Inc.) is a reverse engineering tool that is integrated in the programmable and, hence, extensible reverse engineering tool development environment Software Refinery and provides only base capabilities.
- Imagix 4D (Imagix Corp.) is a reverse engineering tool that can not be extended but provides a large set of built-in facilities.
- Rigi (Rigi Research Project at the University of Victoria) is a public domain reverse engineering tool that provides standard reverse engineering capabilities. It is also an open tool with respect to the provided programming interface to the internal representations of the source code parsing.
- Sniff+ (TakeFive Software) is a software development environment for forward engineering which also provides reverse engineering capabilities.

3.2.3 Evaluation Results

The results of the evaluation show that the reverse engineering tools evaluated in this case study do not provide the same functionality. In fact they provide very different capabilities. The best example are the software views that differ widely from tool to tool not only in their representation but also in depth and type of information. Although it is said that applying reverse engineering tools on an application is an easy task it requires application specific knowledge (e.g. to parse the source code in a useful way) and domain knowledge (e.g. to handle parsing problems) for many tasks during reverse engineering making it not as easy and simple as it may look.

Some views that are supported by the reverse engineering tools cannot be generated in a useful way due to application specific problems (e.g. different source languages used, client-server software). The tool-generated views can only show a part of the application tracing a specific problem (the view of the complete application is not usable in most cases).

It is time consuming to generate useful software views. This results from the incompleteness of some views which have then to be manually refined or completed and from the fact that, although the generation of the views is automatic, most of the time additional effort is needed to relate the view to the specific problem traced (or a part of the application).

The views only represent static information that can be found automatically in the source code. Additional application knowledge, therefore, is not included.

3.3 Manual analysis of source code

3.3.1 Motivation

Reverse engineering tools provide the reverse engineer with lots of information. The process has therefore to include a filtering mechanism to filter out information that is useful for architectural recovery. After the filtering process the software views are still incomplete because of missing domain specific information.

The reverse engineer has therefore to manually reengineer important software views.

3.3.2 Analysis Results

Manual analysis has been performed on a test case of the ARES project. Because of the data orientation of the test case software (real-time software) two views have been manually recovered: interface view and data-flow view.

Since the interface to other subsystems of the test case are mainly based on shared memory and interrupt functions written in assembler language the reverse engineering tools failed to provide interface information. Manual code analysis provided all necessary interface information that was also a precondition for the data flow analysis.

Data flow analysis has been performed for important data flow through the test case system. The manual recovered data flow has already been filtered and is therefore easy to use and evaluate.

Manual recovered views are important views for the first level of abstraction in the architecture recovery process. However, these views do not contain architectural information per se but are an important input in the architecture recovery process.

3.4 Architectural Hypothesis

The identification of architectural elements requires additional input from the application domain: a coarse description of the main components and their interrelationships is used as an architectural hypothesis. This architectural hypothesis (e.g. „the system consists of

an antenna that receives signals and passes it on to a signal decoder to identify telegrams...“) provides a good starting point for architecture recovery: following a process of alternating top-down and bottom-up recovery, architectural elements (such as „types of telegrams“) can be identified much more purposeful.

The architectural hypothesis provides high-level application domain information whereas specific „instances“ of architectural elements in the source code of an application can be found by using several software views generated by reverse engineering tools.

4. CONCLUSION

Our framework for architecture recovery forms an important attempt to combine application domain knowledge and the capabilities of reverse engineering tools in order to strive for the requirements of an architecture recovery tool.

The architectural recovery framework is still under development and will be evaluated in the case study from ABB.

5. REFERENCES

[1] Garlan D., Perry D.E., Introduction to the Special Issue on Software Architecture, *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, April 1995

[2] Gall H., Jazayeri M., Klösch R., Lugmayr W., Trausmuth G., Architecture Recovery in ARES, *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, November 1996.

[3] Harris D.R., Reubenstein H.B., Yeh A.S., Reverse Engineering to the Architectural Level, in *Proceeding of ICSE-17*, IEEE Computer Society Press, pp. 186-195, April 1995.

[4] Weiser M., Program Slicing, *IEEE Transactions on Software Engineering*, SE-10(4):352-357, July 1984.