Architecture Design Recovery of a Family of Embedded Software Systems

An Experience Report

Lars Bratthall and Per Runeson
Dept. of Communication Systems, Lund University, Sweden.
P.O. Box 118, S-221 00 Lund, Sweden.
Phone: +46-462229668., Fax +46-46145823. Email {lars.bratthall|per.runeson}@tts.lth.se.

Key words: Architectural design recovery, experience report, qualitative evaluation

Abstract: Understandability of the current system is a key issue in most reengineering processes. An architecture description of the system may increase its understandability. This paper presents experiences from architectural design recovery in a product family of large distributed, embedded systems. Automated recovery tools were hard to apply due to the nature of the source code. A qualitative evaluation procedure was applied on the performance of the recovery process. The results suggest that producing the necessary architectural documentation during the recovery project costs eight to twelve times as much as producing the same set of documentation during the original development project. By applying a common architectural style for all members of the product family, the component reuse made possible decreased source code volume by 65%.

1. INTRODUCTION

A part of any reengineering project is to create an understanding of the architecture of the current system. This understanding can help determine which pieces are reusable, and to what extent. Also, the current architecture can pose requirements on later developed systems (Abowd et al., 1997). Documentation of the software architecture may also decrease the large proportion of time maintainers spent on developing an understanding of the entity to modify (Holtzblatt et al., 1997). In this paper we present experiences from a project where architectural level design recovery was

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35563-4_35

P. Donohoe (ed.), *Software Architecture*

performed on a product family of five distributed, embedded, software systems.

Design recovery is a phase in reverse engineering where source code and external knowledge are used to create abstractions beyond those obtained directly by examining the system itself (Chikofsky and Cross II, 1990). Biggerstaff (1989) argues that "Design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth." In the project studied, the available source models (Murphy and Notkin, 1995) were the source code for a product family and a few pages of documentation. The access to original system experts was very limited. It was not known what quality attributes the architecture of the software possessed, except that it executed well. It was not known whether the members of the product family shared any common software architecture. The hardware was however well described and identical for all members of the product family. The source code was spread over 90 to 150 files for each member of the product family.

An incremental approach to recovering information from the source code was adopted. To simplify future maintenance the architectural style "Layers" (Shaw and Garlan, 1996) was imposed, due to its known quality properties (maintainability aspects). Imposing an architecture was believed to be feasible as a recovered architecture can be considered an interpretation of a less abstract entity. Different tools for architectural design recovery were investigated, but due to performance constraints only tools that operated on static code could be used. Automated analysis has been discussed by several authors e.g.,, Chase et al. (1998), Harris et al. (1996) and Holtzblatt et al. (1997). Due to certain constructs frequently used in the source code examined, the value of these methods was considered limited.

The software architecture was recovered largely by hand using simple tools like *grep* and *emacs*. SDL (ITU-T, 1996a) was used as architecture description language. Once the architecture of one member of the product family had been recovered, this architecture was reused when attempting architectural recovery on other members of the product family. With some restructuring and minimal reengineering (Chikofsky and Cross II, 1990), both component reuse and architecture reuse (Karlsson, 1995) were used, resulting in a common architecture for all members of the product family as well as a reduction of the total code volume by 65%.

2. CONTEXT

The studied system was contracted to Ericsson Microwave Systems AB who develops complex systems. One of their product areas is

telecommunications. The studied project aimed at designing a family of switches. The switches shared the same set of hardware components, except for different special-purpose printed circuits. One family of subsystems within the switches was studied.

For various reasons the software was not documented according to existing quality standards; the only existing source models available to maintainers were 300 000 lines of C source code, some assembler, and a few pages of documentation, the latter giving little clue regarding the architecture. This rendered any kind of maintenance difficult, as long time had to be allocated just to understand code. Future architectural erosion (Perry and Wolf, 1992) was feared, as there was no known rationale for the architectural design decisions taken.

In order to solve these problems, an architectural design recovery project was launched.

3. OVERVIEW OF THE ARCHITECTURAL DESIGN RECOVERY PROJECT

Biggerstaff (1989) describes a general design recovery process with maintenance and the population of a reuse library as objectives. In this paper, the focus is on practical experiences gained in applying this process.

Biggerstaff's process has three steps:

1. Supporting program understanding,
2. Supporting population of reuse and recovery libraries, and
3. Applying the outcomes of design recovery for refining the recovery.

These steps are applied iteratively.

3.1 Step 1 — Program understanding for maintenance

An architecture recovery team needs some initial knowledge. It includes:

- Details of the available source models
- Available design recovery tools
- Knowledge of what code to allocate to different components.

These issues were addressed initially.

3.1.1 Details of the available source models

Examining the make files showed that some of the files were never used. Examining the filenames showed similarity in the filenames between different members of the product family, and usually the contents of files with the same filename were similar to some extent. Closer examination

indicated that what had originated as identical files had eroded to slightly different files. The analysis also showed that identical C functions sometimes were allocated to different files, without any obvious rationale.

3.1.2 Investigation of design recovery tools

A number of tools believed to be beneficial in design recovery were investigated. Results indicated that a semi-manual approach was needed.

Making a call graph did not help very much, since the subsystems were based on concurrent software processes, communicating mainly using the real-time operating system built-in signals. The call graph showed intra-process communication fairly well, but inter-process communication was not described well.

Identification of a signal being sent could be automated; simple *grep* commands can look for operating system keywords used to create and send signals. Identification of the receiving software process for signals was difficult; we could not rely on pure lexical analysis, since the receiver of a signal usually was determined at run time. Dynamic analysis by executing the system on the target-system could possibly have provided input to event trace analysis (Jerding and Rugaber, 1997), but we were unable to automatically create event traces due to certain constructs frequently used:

- Other mechanisms than signals were sometimes used, especially direct read/write to memory. This communication could not be traced without impeding the function of the system due to performance violations.
- Communication to other subsystems was handled using signals wrapped into special-purpose packets. The operating system debugger could not symbolically show the contents of these packets.

Further tool support was not investigated. Dynamic analysis conflicted with performance requirements, while automatic static recovery tools would have trouble handling the distributed nature, the special-purpose packets, the usage of direct memory read/ write, and the dynamic determination of receiving software processes. Thus, we in many cases had to identify the receiver of signals by manually walking through scenarios (well defined dynamic sequences).

3.1.3 Code to allocate to components

Some source files belonged to only one software process, while some files needed restructuring as parts of the code in one file belonged to more than one software process. There were also two COTS (Commercial Off-The-Shelf) products involved (the operating system and a TCP/IP stack), each spread across a set of files.

The design artefacts to recover were a static architectural description, interwork descriptions, and different dynamic models.

3.2 Step 2 — Populating reuse and recovery libraries

Based on the input from step 1, a set of hypotheses was decided on.

- Manual work during step 2 and step 3 would be necessary, since a recovered software architecture is an interpretation, not entirely visible in code (Holtzblatt et al., 1997).
- Software processes would be the initial abstraction level of the software components. Thus we used a variant of Harris et al.'s (1996) approach, that equated components with software processes. After looking at code, it was found that trying to divide software processes into smaller components, e.g., concurrent state machines, would be difficult as we could not distinguish the individual state machines in the software processes. Therefore we choose software processes as the initial abstraction level.
- Component connectors were to be represented by inter-process signalling. The contents of inter-subsystem communication packets were to be tracked rather than the special-purpose packet itself. Function calls inside a software process would not be described, since we estimated that recovering this information would be too much work related to the use a maintainer would have.
- Describing the architecture of a member of the product family by showing all software processes and their data/control connectors would show too much detail in some situations. Aggregated as well as non-aggregated components should be provided. The smallest component would consist of code related to a single software process.
- Simple tools like *grep* and *emacs* would be the main tools for analysis. SDL would be used to represent the static architecture description. Message Sequence Charts (ITU-T, 1996b) would be used to represent the control and data flow between components.
- For project reasons, an incremental approach allowing the premature termination and later continuation of the architectural recovery was needed.

This led to the workflow described in table 1. On the horizontal axis, activities performed are shown. On the vertical axis, levels of increased value of the recovered artefacts are shown. Components are created at increasing abstraction levels, named C_1, C_2 and C_3. Level C_n components are aggregated from level C_{n-1} components.

Table 1. Goals versus performed activities

	A. Identify software processes	B. Allocate source files to software processes, abstraction level C_1	C. Restructure source files	D. Create special components	E. Cluster level C_1 components into level C_2 components	F. For every software process: Identify sent signals and receivers	G. Describe signals	H. Clarify service provider/requester	I. Cluster level C_2 components into level C_3 components	J. Represent using an ADL
Baseline established	A									
Source code allocated to level C_1 components		B	C							
COTS components handled				D						
Level C_2 components defined					E	F	G			
Level C_3 components defined								H	I	
Architecture graphically described										J

3.2.1 Creation of first order components (level C_1) - activities A-D

All source files belonging to a software process were assigned to one C_1 component. All assembler files were allocated to one C_1 component. Each set of COTS files was allocated to one C_1 component each.

Some files could not be associated with a single software process despite restructuring. These functions were assigned to a library component. The types of level C_1 components created were Single Software Process components, Library components, Assembler components and COTS components.

Level C_1 components were fairly easy to identify; simple tools allowed partly automated analysis. As the source code was not very interleaved (Rugaber et al., 1995) only little restructuring was needed.

3.2.2 Creation of second order components (level C_2) - activities E-G

In order to raise the component abstraction level from each component containing only one software process, to components containing several such components an iterative approach was used. A graphical representation of inter-C_1 control and data communication was drawn using SDL. The inter-process communication constructs prior identified in the source were represented by SDL signals or SDL remote procedure calls. By analysing the communication routes, the type and amount of communication, level C_2 components were decided on. If a set of level C_1 components solved one easily delimited task, they were to be clustered into a level C_2 component.

Identifying level C_2 components was more difficult than identifying level C_1 components. Exact rules for clustering could never be devised, since some level C_1 components participated in solving more than one task.

3.2.3 Creation of third order components (level C_3) - activities H-J

The source code indicated that there were similarities between the members of the product family. We attempted to impose a layered architectural style (Shaw and Garlan, 1996), by clarifying service provider/requester relationships between components. Some restructuring of the original C_2 components was required.

Grouping of level C_2 components into C_3 layer components was done by looking at 'distance from hardware'. All hardware-close level C_2 components were assigned to a level C_3 layer component 'Hardware Abstraction Layer'. Other level C_3 layer components, with decreasing knowledge of hardware specifics, were 'Subsystem Controller', 'Main Controller' and 'Supervision and Test'.

There were several reasons for attempting the layered architectural style:

- The layered architectural style is well known for its good maintainability properties.
- By dividing hardware-close functionality from control, we expected greater chances of component reuse in other members of the product family.

We expected to be able to decrease the difference between different members of the product family by using a common architectural style for all of them.

This multiple-level component architecture was represented in SDL. SDL was chosen, as it allows the direct representation of architectural features (Harris et al., 1996) such as software processes, components consisting of one or more processes, aggregated components, components without any software process, inter-process signalling and remote procedure calls. Thus,

many issues related to the representation problem (Rugaber and Clayton 1993) were avoided. However, there was some semantic distance between C and SDL that had to be mapped: Direct memory reads/writes, interrupts, and the special-purpose packet used to convey signals between different subsystems. These constructs were mapped to SDL signals and SDL remote procedure calls. We used naming conventions to distinguish these constructs from the direct mapping between C source signals and these other communication constructs.

3.3 Step 3 — Applying the outcomes of the design recovery

The above steps were applied for one member of the product family. In doing design recovery for the other members of the product family, the already defined components and the architectural style were reused. By restructuring components by merging files if possible, the number of new components was held down.

Design recovery for the other products in the family was much quicker than for the first member. A large part of the improvement came from having to document few new components. Also, knowing the expected architectural style, less work had to be done in choosing how to restructure the software to fit the architecture.

The degree of reusability of components was proportional to the distance from hardware. The closer to hardware, the more easily could components be reused. By having several component abstraction levels (C_3, C_2, C_1), we could reuse parts or whole of components:

- Most level C_1 hardware-close components could be reused at least once.
- Some members of the product family could share level C2 components.
- Some members of the product family could share level C3 components. A few new level C3 components had to be created, usually by replacing only a few level C_1 components inside a level C_3 component.

The layered architectural style could be reused for all members of the product family.

4. RESULTS AND EXPERIENCES GAINED

The project resulted in a common architectural style for all members of the product family. This enabled component reuse, that decreased the total code volume (lines of source code) by 65%. The volume of architectural descriptions and component descriptions were reduced by approximately 30%, relatively what would have been needed if no reuse had been applied.

A number of faults were discovered in the process of comparing components from different members of the product family.

The set of hypothesis described in section 3.2 remained unmodified through-out the project. However, they would probably have changed if the first step in Biggerstaff's process had not been. This first step helped in deciding on the set of work hypotheses.

It is a daunting task to do architectural recovery when tools can provide only limited aid. Subjective estimation indicates that the effort of our recovery/reuse project amounted to eight to twelve times the effort to accomplish the same results (architectural description, common architectural style, component-based architecture) during the original development project. The estimation is based on accurate figures for the recovery/restructure project and subjective estimations regarding how to handle the problem during original development. Future maintenance is expected to be much simpler and faster than would be possible without the architectural descriptions and component design. Without the design recovery project any maintenance would be extremely difficult.

Experiences have been collected by conducting interviews with the designers involved in the architecture recovery project, as well as future maintainers and some involved managers. Experiences reported are related to tools, people and the recovery process used.

4.1 Tool support

Tools have been used for recovery as well as representation of the recovered architecture. During recovery, UNIX *grep* and the colour marking functions of *emacs* were helpful, especially combined into small scripts. *Grep* allowed the searching of common features across several members of the product family. *Emacs* helped in performing manual slicing, as well as it helped in comparing several versions of files automatically.

SDL has shown to be suitable for describing the component architecture down to the software process level. It was possible to unambiguously describe the constructs believed usable for our purposes. Some semantic distance demanded mapping rules between C and SDL. We believe SDL to be a possible architecture description language for systems, where components are mainly based on software processes and connectors are mainly inter-process communication.

A future challenge to solve is that there is no automatic correspondence between the source code and the architectural abstractions. For example, lack of intense communication between two components is not necessarily a sign of that the two components should not be aggregated into a larger

component. In the system studied, this was apparent when we decided to group hardware-close components into an aggregated component.

4.2 People

Experiences related to people concern previous knowledge and other intellectual instruments for design recovery. As the rationale for architectural decisions is not seen in C, having even limited access to original designers have been extremely beneficial. They have been able to provide information that has not been available in other source models.

Having knowledge of architectural styles helped in choosing to use a layered architecture, as well as trying to establish the service provider/requester divisions, which is a client/server architectural style (Shaw and Garlan, 1996). It is believed that any recovery team can benefit from having access to original design knowledge, domain architecture knowledge and knowledge of architectural styles. Manual design recovery is error prone. This emphasises the need for automated design recovery, or better yet, do it right during the original development.

4.3 The recovery process

Dynamic analysis was difficult due to performance issues. For the purpose of maintenance, dynamic models are considered necessary. Better original descriptions would have been preferred, or, an elaborate debugging component should have been available. For example, being able to run the software on the target platform with relaxed timing requirements would have aided in analysing the software dynamically.

The interleaving problem was rarely encountered, as we never split software processes into more than one component. Content coupling, in terms of several processes sharing a library of functions, was handled by either restructuring those files (by splitting them and allocating them to separate components) or allocating the library functions to a separate library component. By dividing the recovery process into discrete steps, management gained visibility into the project and could decide on project alterations and resource allocation. The incremental approach was thus perceived as beneficial.

5. CONCLUSIONS AND THE FUTURE

From the studied architecture recovery project, we conclude that the design recovery process described by Biggerstaff (1989) works, but

undertaking a design recovery project with limited access to system experts and other source models than the source code, is a daunting task. Especially, understanding hardware-close software is difficult, as it requires detailed hardware understanding. Knowledge of architectural styles and their properties help in choosing a suitable architecture to represent the code, as one knows what quality attributes a particular architecture possesses. An incremental approach to recovering the software architecture is beneficial since it increases visibility into the recovery process.

The recovery project would have benefited from a larger set of well-defined component connectors. Full semantics for the mapping between source code and an architecture description language would allow the automatic creation and simultaneous maintenance of code and architectural views.

Tool support for architectural recovery is important. In industrial projects like this, where the product is supposed to have a life-span of at least 15 years, any description of the architecture should be represented using commercially available tools. We agree with researchers, e.g., Kazman and Carrière (1998), claiming that several methods are necessary in a design recovery project, thus concluding that a workbench with open interfaces is a suitable architecture for design recovery tools.

ACKNOWLEDGEMENTS

This work was partly funded by The Swedish National Board for Industrial and Technical Development (NUTEK), grant 1K1P-97-09690. The project was conducted while employed at the Q-Labs Group. Employees at Ericsson Microwave Systems AB and members of the Software Engineering Research Group at the Department of Communication Systems, Lund University, have provided insightful input.

REFERENCES

Abowd, G., Goel, A., Jerding, D.F., McCracken, M., Moore, M., Murdock, J.W., Potts, C., Rugaber, S., Wills, L. (1997) MORALE. Mission ORiented Architectural Legacy Evolution, in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society, Los Alamitos, USA, 150–9.

Biggerstaff, T.J. (1989) Design Recovery for Maintenance and Reuse. *IEEE Computer*, 22(7), 36–49.

Chase, M.P., Christey, S.M., Harris, D.R., Yeh, A. S. (1998) Recovering Software Architecture from Multiple Source Code Analyses, in *Proceedings of the ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*.

Chikofsky, E.J., Cross II, J.H. (1990) Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1), 13–7.

Harris, D.R., Yeh, A.S., Reubenstein, H.B. (1996) Extracting Architectural Features from Source Code. *Automated Software Engineering*, 3(1/2), 109–38.

Holtzblatt, L.J., Piazza, R.L., Reubenstein, H.B., Roberts, S.N., Harris, D.R. (1997) Design Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 23(7), 461–72.

ITU-T (1996a) *Recommendation Z.100. Specification and Description Language, SDL*, International Telecommunication Union.

ITU-T (1996b) *Recommendation Z.120. Message Sequence Charts*, International Telecommunication Union.

Jerding, D., Rugaber, S. (1997) Using Visualization for Architectural Localization and Extraction, in *Proceedings of the Fourth Working Conference on Reverse Engineering*, IEEE Computer Society, Los Alamitos, USA, 56–65.

Karlsson, E.A. (1995) *Software Reuse - A Holistic Approach*. John Wiley, Chichester, Great Britain.

Kazman, R., Carrière, S.J. (1998) View Extraction and View Fusion in Architectural Understanding, in *Proceedings of the Fifth International Conference on Software Reuse*, IEEE Computer Society, Los Alamitos, USA, 290–9.

Murphy, G.C., Notkin, D. (1995) Lightweight Source Model Extraction. *SIGSOFT Software Engineering Notes*, 20(4), 116–27.

Perry, D.E., Wolf, A.L. (1992) Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), 40–52.

Rugaber, S., Clayton, R. (1993) The Representation Problem in Reverse Engineering, in *Proceedings of Working Conference on Reverse Engineering*, IEEE Computer Society, Los Alamitos, USA, 8–16.

Rugaber, S., Stirewalt, K., Wills, L.M. (1995) The Interleaving Problem in Program Understanding, in *Proceedings of the Second Working Conference on Reverse Engineering*, IEEE Computer Society, Los Alamitos, USA, 166–75.

Shaw, M. and Garlan, D. (1996) *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, New Jersey, USA.