Software architecture reconstruction
Krikhaar, R.

Citation for published version (APA):
Krikhaar, R. (1999). Software architecture reconstruction

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Preface

Motivation

Over the past few years, software architecture has become a major topic in embedded systems development. It is commonly agreed that a good software architecture is indispensable for the development of product families [BCK98] of software-intensive systems. Our company, the Royal Philips Electronics, develops a large range of software intensive systems from medical systems to television sets.

Originally, medical systems were hardware systems with a small amount of software, but in recent years the software has acquired a much more important place in the system, e.g. in the reconstruction of medical images obtained with an X-ray camera. Similarly, at first, televisions did not contain any software, but nowadays these systems are controlled mainly by software, providing e.g. automatic tuning of TV channels.

From an industrial point of view, products containing similar functionalities will have to be introduced on the market ever more rapidly (short lead time). A high level of hardware and software reuse is hence a prerequisite for survival in the competitive market. Different customers want different products, each with their own characteristics, which may even be expressed in non-functional product means, for example the different natural languages as applied in the user interface.

Quality is always of great importance for products. The product’s quality must be continuously monitored and where possible improved. Software is becoming a major part of all these products and quality activities are consequently shifting from hardware to software. Short lead times of products and high quality are in fact conflicting requirements which must be carefully managed.
The software of many Philips’ products is already undergoing changes as indicated above (increased functionality implemented in software, increased product diversity, decreased lead time and improved quality). At the time of these product’s initial development, in some cases decades ago, the software architecture did not play the important role it has today. In those days, the architecture was often not handled explicitly in the engineering phase. At present, the software architecture is of major importance for product development to be able to manage the changes listed above. The spectrum of possible solutions to fill the gap between the absence of an explicit architecture and the need for such an architecture lies between:

- rebuilding the system from scratch and taking care of software architecture explicitly;
- reconstructing an architecture from the implicit architecture and improving this architecture by re-architecting the system.

In this thesis, we focus on the latter side of this spectrum. Our ultimate objective is to define a method for reconstructing software architecture of existing systems. Reconstruction of software architectures requires synergy between tools and domain experts [Cor89, Kri97, SWM97, KC98]. Therefore, we may conclude that there cannot be such a thing as a full-fledged architecture reconstruction tool, though tools that support reconstruction are indispensable.

We propose to make a clear separation between extraction of information from a system and the presentation of the extracted results by means of a separate abstraction activity. In current research an abstraction activity is often not recognised as a separate activity. During software architecture analysis one often wants to query an existing system, e.g., which components use the functionality of component DB? It is not possible to capture all such queries in advance. A flexible set of abstraction operators helps to formulate such queries in an expressive way. In this thesis we will use Relation Partition Algebra to express, amongst other things, such queries.

**Research Contributions**

In this thesis we present a framework for our software architecture reconstruction (SAR) method. This framework consists of two dimensions: SAR levels and views upon software architecture. We define ArchiSpects and InfoPacks as the components that fit in this framework. For all SAR levels, each of the architectural views contain a number of ArchiSpects and
InfoPacks. The applicability of this framework is demonstrated by the definition of a number of ArchiSpects and InfoPacks.

By many people in the software community, formal methods are often considered as inapplicable especially for large real-world systems. Nevertheless, we believed in a formal approach to reconstruct software architectures. Therefore, we developed Relation Partition Algebra (RPA) which is an extension of relational algebra. We showed the applicability of RPA in different industrial settings, which resulted in several ArchiSpects that are defined in terms of RPA.

Currently, a lot of research is performed on software architectures. This research contributes to define better software architectures in the different industrial settings. Besides defining a good architecture, one must, in the various steps of software development, also take care of the proper application of this architecture. A formal definition of a software architecture makes it possible to automatically verify the results of software development (e.g. design and source code) against the defined software architecture. Although, currently, we are not able to define the complete architecture in a formal fashion, we recommend to introduce architecture verification, as much as possible, in any development process.

History of the Project

In the early nineties, a main research topic of our department was to investigate, develop and adapt software development methods to build embedded systems (e.g. televisions). During the introduction of a new (formal) method [Jon88a, Jon88b] for developing software for televisions the need for information extraction arose. Small programs were written to retrieve design information from the source code. In those days, visualisation of software information was also needed. This has resulted in the proprietary tool Teddy-Classic (discussed in Appendix C). In fact, Teddy-Classic was able to display a graph consisting of nodes and edges; nodes represent modules and edges represent module imports. In a later stage, so-called duppies (design update proposals) were implemented using shell scripts to check the consistency of the software structure (an early form of architecture verification). But extraction and checking were still performed on an ad-hoc basis. From this work arose the need for a mathematical foundation for making abstractions upon software. This was the embryonic phase of Relation Partition Algebra; see Chapter 3.
In 1993, research was started to analyse a public telephony switching system (Tele) developed according to a dedicated method. The analysis resulted in a description of the Building Block method. During this research, again, a need for extraction and abstraction mechanisms for software was recognized. This time, it was needed mainly to identify the concepts behind the design method. Later, the Building Block method, which we would nowadays call an architecture method, was partially applied to another communication system. This meant that we first had to analyse the architectural concepts of this system before we could select and apply the most affecting concepts of the Building Block method.

In later projects, the focus shifted to the development of a uniform approach or method (based on the extract-abstract-present paradigm) for analysing the software architecture of existing systems.

Complexity of Systems

In this section we list a number of system's characteristics to give the reader an impression of the variety of concerns a software architecture has to deal with. Therefore, in Table 1, we summarize some of the characteristics of three typical systems of Philips: two professional systems Telecommunication (Switch) and Medical Systems (Med) and a consumer electronics system (Cons)¹. All of these characteristics play their own role in almost any architectural decision, or they are an outcome of such a decision (e.g. the number of subsystems).

The number of customers and the number of different products are given in the Product View part of the table. In case of professional systems, each customer gets his or her own dedicated system. But we have used the definition that two products are different only when they differ significantly either in hardware or in software.

The Evolutionary View part shows figures relating to the system's current age and its expected total lifetime. The release cycle describes the average time between two major releases. The release footprint indicates the percentage of files that have been touched since the last release.

The number of code files is given in the Code View part of the table.

¹It is hard to normalise the data of the different systems; we have handled the figures in a non-scientific fashion. The table is therefore meant mainly to illustrate the complexity of systems.
One can argue about the way how the size of the source code should be measured, but for our purpose the number of lines suffices. The number of programming languages indicates problems that could arise in merging software parts and e.g. the required educational background of developers.

The Module View part of the table shows the number of subsystems into which the system is decomposed. The number of file includes gives an indication of the interconnectivity between the various software parts. In some cases parts of the software are built by external parties (external components) with their typical integration difficulties.

The number of software processes listed in the Execution View deserves some extra attention. For the Med system, these are software processes with their own address space, but for the Cons system these are activities that can be compared with threads (i.e. sharing an address space). The Switch system has its own operating system supporting light-weight processes. In the last two systems, the processes are in fact created at initialisation time, whereas in the Switch system processes are dynamically created during system operation. The number of computing devices describes the processing units in the system in which software runs.

<table>
<thead>
<tr>
<th>Product View</th>
<th>Switch</th>
<th>Med</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td># customers</td>
<td>$10^3$</td>
<td>$10^3$</td>
<td>$10^6$</td>
</tr>
<tr>
<td># products</td>
<td>$10^1$</td>
<td>$10^1$</td>
<td>$10^2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Evolutionary View</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>system's age/lifetime (years)</td>
<td>$15/30$</td>
<td>$15/30$</td>
<td>$3/5$</td>
</tr>
<tr>
<td>release cycle (years)</td>
<td>0.7</td>
<td>0.5</td>
<td>N/A</td>
</tr>
<tr>
<td>release footprint (% touched files)</td>
<td>-</td>
<td>60%</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code View</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># code files</td>
<td>$4 \times 10^3$</td>
<td>$7 \times 10^3$</td>
<td>$0.6 \times 10^3$</td>
</tr>
<tr>
<td># lines of code</td>
<td>$1.4 \times 10^6$</td>
<td>$2.4 \times 10^6$</td>
<td>$0.4 \times 10^6$</td>
</tr>
<tr>
<td># programming languages</td>
<td>3</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module View</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># subsystems</td>
<td>8</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td># file imports</td>
<td>$32 \times 10^3$</td>
<td>$70 \times 10^3$</td>
<td>$0.8 \times 10^3$</td>
</tr>
<tr>
<td># external components</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution View</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># operating systems</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td># (main) computing devices</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td># software processes</td>
<td>$&gt; 1000$</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 1: Characteristics of Large Systems
Related Work and Tools

In this thesis in appropriate sections, we will relate our work to work of others. For the reader’s convenience, in advance, we will briefly discuss some closely related work. This short introduction is also meant for readers who are familiar with that work, to put our work into perspective.

Rigi

*Rigi* [SWM97] is a tool that supports the extraction of information (e.g. *rigiparse* extracts information from C source code) and the presentation of extracted information (e.g. showing coloured line-box diagrams). After the initial information has been presented, one can perform some simple abstractions, e.g. the creation of composites and calculation of complexity quality measures. *Rigi* is an open tool, which means that new functionality can be easily added using the *Rigi Command Language*. The repository of *Rigi* consists of a resource-flow graph, containing different types of vertices and edges, representing software entities and relations between software entities.

*Rigi* can be very useful in the analysis of a system. The standard way of presenting graphs (equally sized nodes and fixed points for connecting edges to boxes) could, however, be a drawback. We think reconstructed architectural information should be presented in a layout which is similar to the software architecture information as initially documented in the development group concerned.

The extraction and presentation functionalities of *Rigi* can be easily combined with our ideas of a separate abstraction activity. For example, the user interface of *Rigi* can be extended with a menu and abstraction machinery to query software.

Reflexion Models

Murphy et al. [MNS95, MN97] have described *reflexion models*. A *reflexion model* shows the differences and agreements between the engineer’s high-level model and the model of the source code. An engineer defines a high-level model and specifies how this model maps to source code. A tool computes a *reflexion model* that shows where the engineer’s high-level model agrees with, and where it differs from the source model. A for-
mal model is used to calculate convergences, divergences, and absences of relations in the high-level model or the source model.

Reflexion models show differences between the engineer’s mental model of the system and the as-built model of the system. We will use architecture verification, which is a process that makes explicit distinctions between the as-built architecture of the system and the intended architecture of the system (as defined in advance by architects). A similar approach, called design conformance, is discussed in [MNS95].

Relational Algebra

Holt [Hol96, Hol98] suggests Tarski’s Relation Algebra as a theoretical basis for software manipulations (or in fact he considers manipulation of visualisation).

There is a remarkable correspondence between Holt’s work and our own work on Relation Partition Algebra [FO94, FKO98].

Software Bookshelf

The software bookshelf [FHK+97] is a framework for capturing, organizing, and managing information on the system’s software. The bookshelf framework is an open architecture, which allows a variety of tools to be integrated, e.g. the Rigi presentation tool. Reverse engineering tools can populate the bookshelf repository from which information can be retrieved by other tools. All information transport within this framework is performed via Web protocols.

The open architecture makes this framework interesting for integration with other approaches, e.g. with our approach as described in this thesis. Web technology incorporates many presentation and navigation techniques that are useful for software reconstruction. The software bookshelf distinguishes three different roles: builder, patron and librarian. We experienced that these three roles are useful in introducing reconstruction technology in an organisation.

Dali

Dali [KC98] is an architecture analysis framework containing e.g. Rigi as a presentation tool. It is based on view extraction, extraction of static and
dynamic elements from the system, and *view fusion*. *View fusion* consists of combining views in order to achieve new views that are richer and/or more abstract. *Dali* contains an SQL database containing the various views. We consider SQL less accurate for expressing software manipulations. We will therefore introduce Relation Partition Algebra, which has more accuracy (e.g., by means of the operations transitive closure and transitive reduction).

**Outline of Thesis**

In Chapter 1 we discuss the term software architecture. An overview of some keynote papers is given, including models describing various views on software architecture. Business goals, objectives and patterns for software architecture are presented. The relations between these items are illustrated in a so-called GOP (Goals, Objectives, Patterns) diagram.

In Chapter 2 we focus on the engineering aspects of software architecture. We discuss aspects of reverse engineering in general and the aspects of reverse engineering software architectures in particular. The global design of our software architecture reconstruction (SAR) method is discussed, including an introduction to the notions of Info Packs and ArchiSpects (modular pieces of our method).

In Chapter 3 we discuss the mathematical foundation of our method: Relation Partition Algebra (RPA). RPA is an extension of relation algebra fine-tuned for, but certainly not restricted to, software.

In Chapter 4 we focus on the comprehension of existing software architectures. The baseline is a system, typically evolved over fifteen years, which is not completely known by all of its current developers. A number of Info Packs and ArchiSpects are presented.

Chapter 5 addresses re-defining the software architecture of an existing system. Before one can improve, one must clarify the current architecture and one must define the required architecture. Our reverse architecting method supports the development of an improvement plan by analysing the impact of certain changes.

In Chapter 6 a way of managing software architectures is presented: by verifying whether the design/implementation satisfies the software architecture one achieves architecture conformance.

Chapter 7 gives recommendations for the application of software arche-
ture reconstruction.

The appendices present extraction tools, abstraction tools and presentation tools as referred to throughout the thesis. The last appendix presents all the RPA operators in a nutshell.

The thesis contains many examples, which we have slightly modified to retain Philips’ competitive edge. In my opinion, this does not affect the illustrative value of these examples.