Applying Relation Partition Algebra for Reverse Architecting

André Postma, Marc Stroucken
Philips Research Laboratories
Prof. Holstlaan 4 (WL-01)
NL 5656 AA Eindhoven
The Netherlands
e-mail: {andre.postma, marc.stroucken}@philips.com

Abstract

The ever increasing complexity and ever more rapidly changing requirements for software systems create a need for reuse and adaptation of existing software. An explicit description of the software architecture of a system may help to get a high-level overview of the system, thus facilitating re-use and adaptation of the system. Reverse architecting is a technique applied within Philips to make the module architecture of a software system explicit. In this paper, the reverse architecting technique is briefly described. Furthermore, it is shown how Relation Partition Algebra can be used for reverse architecting purposes as a formalism to describe the module architecture of a system, and provide the architectural information with the right level of detail.

1. Introduction

Royal Philips Electronics N.V. is a world-wide company which develops professional systems (like business communication and medical systems) as well as consumer systems (like television sets and VCRs). The ever increasing complexity of the software, and the ever more rapidly changing requirements for these systems create a need for easily extendible, flexible, configurable, testable and maintainable systems. Because the time between two subsequent generations of a system becomes shorter and shorter, re-use and adaptation of existing software has become a necessity. For this purpose, a good overview of existing software is required. An explicit description of the architecture of a software system may help to get a high-level overview of that system.

In literature, several different definitions of ***software architecture*** exist (see, e.g., [BaCK98, ClNo96]). In this paper, we conform to the definition of Soni et al. (in [SoNH95]) who state that software architecture is concerned with capturing the structures of a system and the relationships among the elements both within and between structures [SoNH95]. Besides this structural information, software architecture also includes constraints on the system or on elements of the system. Depending on the engineering concerns involved, several different structures can be distinguished. Kruchten (in [Kruc95]) and Soni et al. (in [SoNH95]) introduce similar approaches to categorize these structures. Soni et al. (in [SoNH95]) distinguish between several ***categories*** of structures: conceptual architecture, module (interconnection) architecture, code architecture, and execution architecture. Kruchten (in [Kruc95]) defines different ***views*** (logical view, development view, process view, physical view) based around requirements in the form of scenarios. The two approaches are similar, and both provide a possibility for separation of engineering concerns in the description of the software architecture of a system.

Most of our work has been based on the ***module architecture*** of a system. The module architecture captures the functional decomposition of the system (i.e., the decomposition of the system into subsystems, modules, and abstract program units), and the system's subdivision into a number of hierarchical layers. Furthermore, it describes the interrelationships between the various components of the system [SoNH95]. In a number of projects within Philips, ***relation partition algebra*** [FeKO98, FeOm94, FeOm99, Krik97] has been used as a formalism to describe, analyse and manipulate the module architecture of a system. A brief description of relation partition algebra will be given below.

In this paper, we focus on a technique called ***reverse architecting*** [Krik97], which we apply in order to make the module architecture of existing systems explicit. The described reverse architecting

technique consists of three steps: extraction, abstraction, and presentation, each of which will be explained below. See also Figure 1.

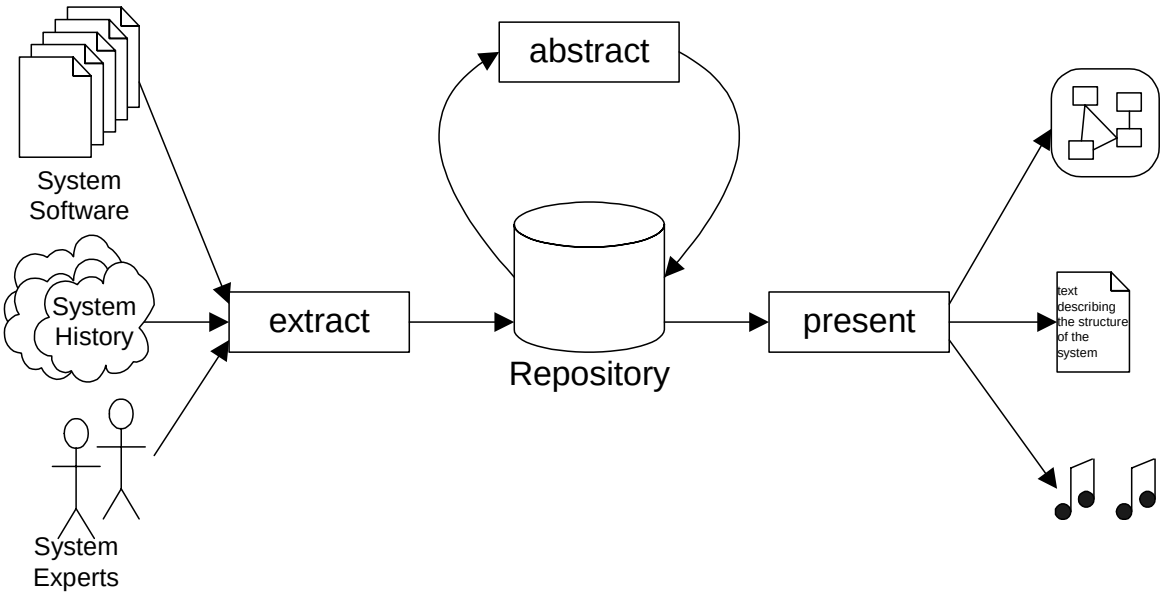

Figure 1 Extract - abstract - present paradigm

The remainder of this paper is structured as follows. First, an overview is given of the reverse architecting technique used to make the module architecture of existing software systems explicit. Then, a brief description of relation partition algebra is given and its application in reverse architecting is demonstrated.

2. Reverse architecting

In [ChCr90], reverse engineering is defined as the process of analysing a subject system to identify the system's components and their relationships and create representations of the system in another form or at a higher level of abstraction. The main goal of reverse engineering is to increase comprehensibility of the system for maintenance and new development. Reverse architecting [Krik97] is the flavour of reverse engineering that concerns all activities for making existing software architectures explicit.

Just like reverse engineering, reverse architecting can be subdivided into three subsequent steps: extraction, abstraction, and presentation (Figure 1).

The ***extraction phase*** is concerned with extraction of architectural information from the source code and storing the architectural information in a repository. The architectural information is stored as a collection of sets and relations between the elements of the sets. Although the architectural information is extracted solely from the system software (i.e., source code, directory structure, configuration management), consulting the system documentation and the system architects will generally be required to obtain information about how the architectural information is mapped onto the source code (e.g., by means of naming conventions, directory structures, and so on). However, in some cases, the configuration management of a system is done in such a way, that the required architectural information can be extracted without the help of system documentation or system architects (see, e.g., [BFGK99, KJMF99]). Since the systems we consider usually contain millions of lines of code, automation of the extraction process is required. For this process, extraction tools like scripts, source browsing tools, etcetera are needed. Generally, the extraction tools are specific for the system from which the information is extracted.

In the description of the module architecture of a system, we can distinguish between different levels of abstraction. I.e., the system architecture can be described in terms of large course-grained elements (e.g., units, modules) at a high level of abstraction, but also in terms of smaller fine-grained elements

(e.g., methods, classes) at a lower level of abstraction. Usually, the architectural information extracted from the system software yields a description of the system in terms of fine-grained elements (i.e. at a low abstraction level). In general, the elements at a higher level of abstraction are conceptual entities which can not be directly extracted from the system software. The ***abstraction phase*** consists of grouping and filtering extracted information, so as to obtain a manageable set of information at the desired level of abstraction. In this step, the so-called ***lift-operator*** plays an important role, which is used to 'lift' relations to a higher level of abstraction. See also Section 3.

Finally, the ***presentation phase*** consists of presenting the architectural information in a way that appeals to the architects and system developers. Both graphical and textual (web-based, browsable) representations are being used. Usually, the graphical representations are used for obtaining a global insight in the system architecture, whereas the textual representations are used to get more detailed information.

3. The use of Relation Partition Algebra for reverse architecting

Relation partition algebra (RPA) is an algebra defined on ***sets*** and ***relations***. All kinds of operators on sets and/or relations are defined within RPA: set / relation inclusion ($\subseteq$), composition (;), intersection ($\cap$), union ($\cup$), converse ($^{(-1)}$), transitive closure ($^{+}$), to only mention a few. For a detailed description of RPA, we refer to [FeKO98, FeOm94, FeOm99, Krik99]. In order to increase the expressiveness of RPA, it has also been extended with so-called ***multi-sets*** and ***multi-relations***, in which for each element in the multi-set or multi-relation, the number of occurrences of this element has also been indicated. For more information on multi-sets and multi-relations in RPA, we refer to [FeKr99, Krik99]. In this paper, we will restrict the use of RPA to sets and relations. We introduce RPA by means of an example in which we show how RPA can be used for reverse architecting.

When applying RPA for reverse architecting, our extraction step consists of expressing the module architecture of a system in terms of sets and relations. In architecture descriptions, ***sets*** are used to define important architectural entity types, like, e.g., units, modules, and functions. In the examples below, the sets of units, functions, and modules will be denoted by *U*, *M*, and *F*, respectively. In RPA, a (basic) set is defined simply by enumerating the elements that it contains, e.g.:

$$U = \{unit_1 , unit_2 , unit_3 \}$$
$$M = \{m_1 , m_2 , m_3 , m_4 \}$$
$$F = \{f_1 , f_2 , f_3 , f_4 , f_5 , f_6 \}$$

More sets can be derived by using set expressions in RPA (E.g., the set of all relevant architectural entities of the above example could be defined by $U \cup M \cup F$).

We use ***relations*** to express the dependencies that exist between elements of the architectural entity types. In our work we have restricted ourselves to binary relations. A binary relation is a set of tuples. In architecture descriptions, binary relations consist of tuples of architectural entities. A binary relation is determined by its name and a specification of its domain and range. We give some examples to make things clear. In these examples, the usage relation between functions is denoted as $U_{F,F}$, and the part-of relation between functions and modules as $P_{F,M}$.

$$U_{F,F} = \{<f_1 , f_2>,<f_1 , f_3>,<f_1 , f_4>,<f_3 , f_5>,<f_4 , f_6>\}$$
$$P_{F,M} = \{<f_1 , m_1 >,<f_2 , m_2 >,<f_3 , m_3 >,<f_4 , m_4 >,<f_5 , m_1 >,<f_6 , m_2 >\}$$

In Figure 2, we have depicted the various relations that play a role in this example. The modules m_1 through m_4, and the functions f_1 through f_6 have been indicated with boxes. The part-of relation has been implicitly depicted by positioning each function within the module that it is part of. The usage relation between functions is indicated by thin arrows, whereas the usage relation between modules is indicated with thick arrows.

Part-of relations play an important role in RPA. They describe the partitioning of the system into subsystems and components. Characteristic of such a part-of relation is that it is acyclic and functional. It describes a partition, i.e., each domain element occurs at most once in the relation.

For reverse architecting purposes, several basic relations can be directly extracted from the system (during the extraction phase). Other relations (especially those with respect to high-level conceptual entities in the system) can be calculated during the abstraction phase from the extracted sets and/or relations and stored in the repository. For the purpose of abstraction, the ***lift operator*** ($\uparrow$) has been defined. The lift operator lifts relations to a higher level of abstraction, which may be helpful in obtaining the right level of abstraction. The lift operator can be expressed in terms of the composition and converse operators as follows:

$$U \uparrow P = P^{(-1)} ; U ; P$$

Here, U represents a low-level usage relation (i.e. a usage relation between fine-grained elements of the system). P represents a part-of relation which describes a partitioning (i.e., it describes which fine-grained elements are part of which coarse-grained entities). $U \uparrow P$ describes a usage relation at a higher level of abstraction.

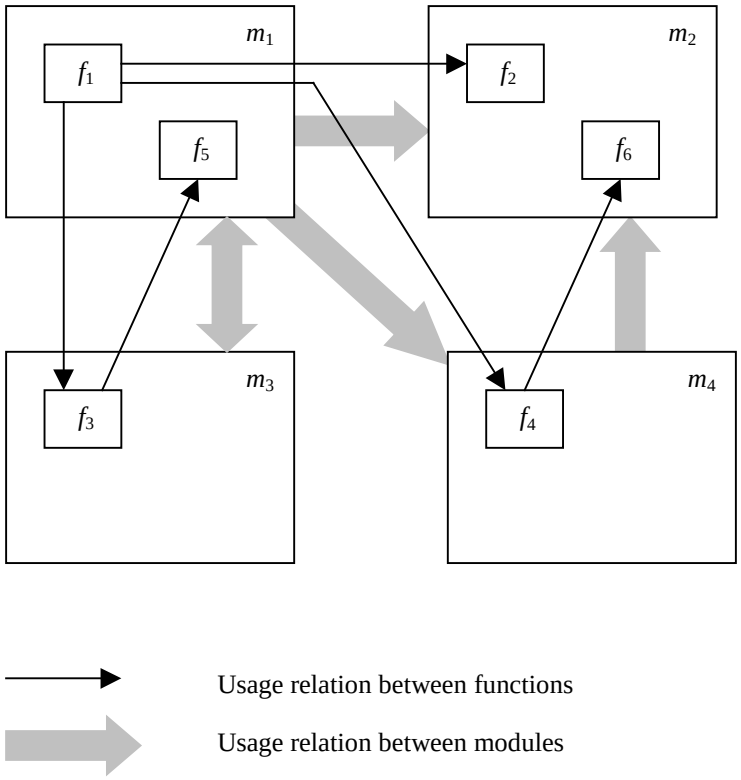

Figure 2 Example of part-of and usage relations

To clarify the meaning of the operators just mentioned, we will now give examples of the converse, composition and lift operator for the above relations:

$$P_{F,M}{}^{(-1)} = \{<m_1 , f_1>, <m_2 , f_2>, <m_3 , f_3>, <m_4 , f_4>, <m_1 , f_5>, <m_2 , f_6>\}$$

I.e., the converse operator alternates the domain and the range of a relation.

$$U_{F,F} ; P_{F,M} =$$
$$U_{F,M} =$$
$$\{<f_1 , m_2>,<f_1 , m_3>,<f_1 , m_4>, <f_3 , m_1>,<f_4 , m_2>\}$$

I.e., the composition of two relations R_1 and R_2 yields all tuples with a domain element of a tuple of the R_1 and the range element of a tuple of R_2, for which it holds that the range element of the tuple of R_1 is identical to the domain element of the tuple of R_2. I.e., $<f_1 , m_2>$ is part of $U_{F,F}$; $P_{F,M}$, since $U_{F,F}$ contains a tuple $<f_1 , m_1>$ and $P_{F,M}$ contains a tuple $<m_1 , m_2>$. The composition operator is associative, i.e., for any relations *P, Q, R,* it holds that $P ; Q ; R = (P ; Q) ; R = P ; (Q ; R)$.

$$U_{F,F} \uparrow P_{F,M} =$$
$$P_{F,M}{}^{(-1)} ; U_{F,F} ; P_{F,M} =$$
$$P_{F,M}{}^{(-1)} ; (U_{F,F} ; P_{F,M}) =$$
$$P_{F,M}{}^{(-1)} ; U_{F,M} =$$
$$U_{M,M} =$$
$$\{<m_1 , m_2>, < m_1 , m_3>, < m_1 , m_4>, < m_3 , m_1>, < m_4 , m_2>\}$$

From the last example, it is clear that by using the lift-operator, it is possible to calculate the uses relation between modules from the uses relation between functions and the part-of relation between modules and functions.

After the abstraction phase, the repository contains both the extracted sets and relations and the sets and relations calculated during the abstraction phase. In the presentation phase, the system architect or developer can select the sets and/or relations that he/she wants to be presented from the repository. In this way, the architect or developer can choose the right level of abstraction to gain more insight in the system's architecture. In the presentation tools used within Philips, the user may interactively change the level of abstraction at which the architectural data is presented.

4. Conclusion

In this paper, we have indicated that reverse architecting, as it is currently being applied within Philips, is a useful technique for gaining insight in the structure of existing software systems. We have briefly described the reverse architecting technique, which consists of three phases: extraction, abstraction, and presentation. Finally, we have shown that relation partition algebra can be used as an underlying formalism for abstraction.

5. Acknowledgements

The authors would like to thank Reinder Bril, Wim Christis, and Jaap van der Heijden for carefully reading this manuscript and for providing useful suggestions to improve this paper.

6. References

[BaCK98] Bass, L., Clements, P., and Kazman, R., *Software Architecture in Practice*, SEI Series in Software Engineering, Addison Wesley, Reading, Massachusetts, 1998, ISBN 0-201-19930-0.

[BFGK99] Bril, R.J., Feijs, L.M.G., Glas, A., Krikhaar, R.L., and Winter, T., *Maintaining a legacy: towards support at the architectural level*, Manuscript NL-MS-20209, submitted for publication, 1999.

[ChCr90] Chikofsky, E.J., Cross II, J.H., Reverse Engineering and Design Recovery: A Taxonomy, in: *IEEE Software*, January 1990, pp. 13-17.

[ClNo96] Clements, P.C., and Northrop, L.M., *Software Architecture: An Executive Overview*, SEI, Technical Report, CMU/SEI-96-TR-003, ESC-TR-96-003, Pittsburgh, February 1996.

[FeKO98] Feijs, L., Krikhaar, R., and van Ommering, R., A Relational Approach to Software Architecture Analysis, in: *Software Practice and Experience*, Vol. 28, No. 4, April 1998, pp. 371-400.

[FeKr99] Feijs, L.M.G., and Krikhaar, R.L., Relation Algebra with Multi-Relations, in: *International Journal Computer Mathematics*, Vol. 70, 1999, pp. 57-74.

[FeOm94] Feijs, L.M.G., and van Ommering, R.C., *Theory of Relations and its Applications to Software Structuring*, Philips internal report, Philips Research, 1994.

[FeOm99] Feijs, L.M.G., and van Ommering, R.C., Relation Partition Algebra - mathematical aspects of uses and part-of relations -, in: *Science of Computer Programming*, Vol. 33., 1999, pp. 163-212.

[KJMF99] Krikhaar, R.L., Jong, R.P. de, Medema, J.P., and Feijs, L.M.G., Architecture Comprehension Tools for a PBX System, in: *Proceedings of the 3rd European Conference on Software*

Maintenance and Reengineering, March 3-5, 1999, Amsterdam, IEEE Computer Society, ISBN 0-7695-0090-0, 1999, pp. 31-39.

[Krik97] Krikhaar, R.L., Reverse architecting approach for Complex Systems, in: *Proceedings of the International Conference on Software Maintenance (ICSM'97),* IEEE Computer Society, 1997, pp. 4-11.

[Krik99] Krikhaar, R.L., *Software Architecture Reconstruction*, Ph.D. thesis, University of Amsterdam (UvA), ISBN 90-74445-44-6, 1999.

[SoNH95] Soni, D., Nord, R.L, and Hofmeister, C., Software Architecture in Industrial Applications, in: *Proceedings of the ICSE'95*, ACM Press, 1995, ISBN 0-89791-708-1, pp. 196-207.