Software architecture reconstruction

Krikhaar, R.

Citation for published version (APA):

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: https://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.
Chapter 7

Concluding Remarks

We finish this thesis with recommendations for application of the SAR method in a real world system. Furthermore, the application of RPA in several contexts is elaborated once more.

7.1 Recommendations for Application

In this thesis we have discussed a framework for the Software Architecture Reconstruction method. InfoPacks and ArchiSpects fit in various architectural views at different levels in this framework. This modular structure of the SAR method simplifies discussing software architecture reconstruction. For the module view and code view of software architecture, we presented a number of InfoPacks and ArchiSpects, summarized (») in Table 7.1. The SAR method can be enhanced with new ArchiSpects, which fit in the framework.

When applying SAR to an existing system, one should first consider which ArchiSpects are most valuable to reconstruct. Most of the discussed ArchiSpects are based on the imports relation. One imports a header file to be able to use a function, a type definition, a macro and/or a global variable from another file. The imports relation is in fact a mixture of a number of relations: calls, accesses and typed, and for object-oriented languages also the inherits relation. When reconstructing a system in more detail, one requires a refinement of the imports relation. For example, for each of these relations the Component Dependency ArchiSpect can be refined, e.g. Component Dependency for function calls.
<table>
<thead>
<tr>
<th>SAR levels</th>
<th>Logical View</th>
<th>Module View</th>
<th>Code View</th>
<th>Execution View</th>
<th>Hardware View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Managed</td>
<td>▶ Generics and Specifics Conformance</td>
<td>▶ Aspect Conformance</td>
<td>▶ Usage Conformance</td>
<td>▶ Layering Conformance</td>
<td>▶ Aspect Assignment</td>
</tr>
<tr>
<td>Redefined</td>
<td>▶ Aspect Coupling</td>
<td>▶ Cohesion and Coupling</td>
<td>▶ Component Coupling</td>
<td>▶ Depend</td>
<td>▶ Resource Usage</td>
</tr>
<tr>
<td>Described</td>
<td>▶ Using and Used Interfaces</td>
<td>▶ Component Dependency</td>
<td>▶ Software Concepts Model</td>
<td>▶ Source Code Organisation</td>
<td>▶ Build Process</td>
</tr>
<tr>
<td>Initial</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Software Architecture Reconstruction
For the reconstruction of the execution view of software architecture, we suggest the following ArchiSpects: Process Communication, Process Topology and Resource Usage. The Process Communication ArchiSpect describes how processes (extracted by a Processes InfoPack) communicate with each other (e.g. via TCP/IP, a database, shared memory or shared files). The Process Topology ArchiSpect describes in terms of a running system how and when processes are created and killed. The Resource Usage ArchiSpect describes the usage of different resources, e.g. RAM memory, disk memory and cpu. A first experiment in reconstructing Resource Usage of Mod is presented in [KPZ99]. The suggested InfoPack and ArchiSpects of the execution view are presented (>) in Table 7.1.

Although separately discussed, the module view, code view and execution view are related. In [KFJM99, BFG99] we discussed how scenarios, applied to the Switch system, can help developers comprehend these three views. The importance of combining static analysis with dynamic analysis has also been discussed by Kazman and Carrière [KC98].

For the described and managed level of software architecture reconstruction, one should integrate reconstruction activities in the development process. An up-to-date described architecture supports developers in their activities by means of given opportunities to comprehend the software architecture better. Web technology is the most appropriate mechanism for presenting requested information to developers due to its accuracy and multi-media nature. Also, for the managed architecture, one should integrate reconstruction activities tightly in the development process. In this way, feedback relating to architecture conformance can be given as soon as possible. In the case of new systems, special attention must be given to architecture verification, because it is easier to introduce architecture conformance in an early stage of the product’s life-cycle than it is to introduce it in existing systems. In that stage, it is easier to take special measurements and define extra coding standards to increase the possibilities of extracting architectural information and verifying architectural decisions.

### 7.2 Relation Partition Algebra

Since 1994, when Relation Partition Algebra (RPA) was defined [FO94], in 1994, we have applied it in various areas of software architecture analysis. We experienced that RPA is suitable for making software abstractions, embellishing the presentation of information, expressing software metrics,
performing dedicated analyses, navigating through information, verifying architectural decisions and recognising patterns in software. We will briefly discuss these different areas.

RPA offers filtering operators (e.g. $|_{\text{dom}}$, $|_{\text{ran}}$, $\setminus_{\text{dom}}$) and grouping operators (e.g. $\uparrow$) for abstracting information from software. These operators make it possible to focus on specific data (i.e. to answer a question a software analyst has in mind), and to eliminate irrelevant data. Furthermore, information can be combined (e.g. through composition: $\circ$) to obtain more dedicated information.

The presentation of a function call graph of a large system probably results in a diagram that contains a large black area (i.e. all the edges of the graph). Abstractions can help to reduce the amount of information to embellish a graph presentation. For example, lifting a large imports relation reduces the amount of information in a smart way. Transitive reduction also improves the presentation of information. The transitive reduction removes shortcuts from a (cycle-free) relation, resulting in a more convenient graph.

We can also express software-related metrics in RPA (e.g. cohesion and coupling). The notion of partof relations gives such metrics an extra dimension; one can consider cohesion and coupling at different levels in the decomposition hierarchy.

RPA is also useful for performing dedicated analyses, e.g. detecting cyclic dependencies in a system, recognising local functions and calculating components to be tested:

- To detect whether a relation ($R$) contains cycles, it suffices to calculate $R^+ \cap Id$. If this equals the empty relation, then $R$ contains no cycles [FKO98].
- A function is local to another function if it is used by this function only. Some programming languages offer concepts for defining local functions (e.g. Pascal). To minimise a system’s complexity, one should define local functions close to their caller (preferably by limiting the scope of the local function).
- Given a dependency relation ($D$) between components and a list of modified components ($M$), it is calculated which components must be tested again (because of the changes): $\text{dom}(D^* |_{\text{ran}} M)$, i.e. all the components which, directly or indirectly, depend on a changed component must be tested again.

\[ A \text{ tuple } (x, z) \text{ is a short-cut if the relation contains the tuples } (x, y_1), (y_1, y_2), \ldots, (y_n, z) \text{ for some } n \geq 1. \]
Presentation can be made more dynamic by offering some navigation mechanisms. For example, a user may want to zoom-in or zoom-out on certain information. *TabView* is a presentation tool that provides navigation abilities by executing RPA formulas and re-calculating a new table.

For the *managed* architecture, we have defined a number of ArchiSpects that incorporate architectural rules. RPA is suitable for formalising architectural decisions, making it possible to automatically verify an implementation.

In the discussion of Section 6.5 we described how the generic and specific components can be recognised in software. We formulated the *pattern* to which generic and specific components adhere in RPA.

As indicated above, a number of different areas of software analysis can be covered by RPA. It is a great advantage to have a single formalism for different applications (consider e.g. the learning curve of a new formalism). RPA offers a formal notation, but RPA formulas can also be executed on a computer. In this way, one can easily explore different aspects and parts of the system, using an interactive RPA calculator, see Figure B.2. After performing various calculations, one can consolidate these calculations by defining a new ArchiSpect which reconstructs a certain interesting aspect of architecture. This approach in fact roughly describes the way in which we analysed a number of systems at Philips and the way in which we deduced various InfoPacks and ArchiSpects.

The diversity of applying RPA summarized above strengthens our thoughts of using RPA as a foundation for an Architecture Description Language (ADL). The semantics of notations and operations of an ADL can be expressed in terms of RPA. Further research into this topic is required to validate these thoughts.