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.
Chapter 6

Managed Architecture

The SAR method consists of five levels of reconstruction (initial, described, redefined, managed and optimized). In this chapter we discuss the managed level of SAR. We focus on architecture verification, a means to keep the defined architecture and actual architecture consistent.

6.1 Introduction

The software architecture intended by architects should be well documented. Nevertheless, a number of implicit assumptions relating to the architecture reside in the heads of the architects and developers only. Sources, documents and architects’ minds together in fact embody a system’s intended software architecture.

The actual software architecture, i.e. the architecture implemented by the software developers, will definitely deviate from the intended architecture when no precautions are undertaken to prevent such deviations. Architecture verification is the process of revealing the deviations between the intended architecture and the actual architecture. Preferably, this is performed as early in the development process as possible. The main goal of architecture verification is to achieve architecture conformance. Bass et al. [BCK98] define architecture conformance being the activity that is concerned with keeping developers faithful to the structures and interaction protocols constrained by the architecture. If architecture verification is applied consistently, and is properly integrated in the development process, a managed software architecture is achieved.
Some of the architectural decisions can be formally defined. Given that appropriate information can be extracted, one can automatically verify the formally defined decisions. Other architectural decisions are more intuitive, and therefore it is hard to verify them automatically. For example, a description of the contents of a component like *All functions relating to printing must be contained in the “Printing” component* can currently only be interpreted by humans. In this chapter we will concentrate on the rules that can be automatically verified. The other type of architectural rules could be verified in e.g. review sessions.

If a system’s implementation does not conform to the architecture, this will have to be fixed. Sometimes this may lead to changes in the architecture, but, more often, the design and/or source code will have to be modified. Sometimes both the architecture and the source code will have to be modified. *Architecture violations* describe the parts of the implementation that do not conform to the intended architecture. The ArchiSpects of the managed architecture define the architectural rules and the corresponding architectural violations. Violations are defined in such a way that they support resolving a disconformance (by suggesting a possible solution). An architectural rule is satisfied if there are no architectural violations.

In Chapter 1.5 we discussed a number of good architectural patterns. Here we will discuss some ArchiSpects that correspond to these architectural patterns. For each system, a dedicated set of ArchiSpects must be defined to comprise the definition of the architecture. In this chapter we will discuss the following ArchiSpects:

- *Layering Conformance*;
- *Usage Conformance*;
- *Aspect Conformance*;
- *Generics and Specifics Conformance*.

### 6.2 ArchiSpect: Layering Conformance

#### 6.2.1 Context

The *Layering Conformance* ArchiSpect belongs to the module view. It requires results of the *Import* (Section 4.6) and *PartOf* (Section 4.7) InfoPacks. It is furthermore related to the *Software Concepts Model* (Section 4.2) and *Component Dependency* (Section 4.9) ArchiSpects.
Figure 6.1: Overview of Managed Architecture
6.2.2 Description

A layer is a group of software elements. Layers are strictly ordered. Given the layer ordering, elements of a higher layer may use only elements of lower layer(s). Layering offers the possibility to develop and test the system incrementally: from the bottom layer towards the top layer. The principles of layering have been discussed in Section 1.5.1.

6.2.3 Example

Cons

The architecture of the Cons system describes a transparent layering of the system depicted on the left side of Figure 6.2. The actual implementation (on the right side of the figure) shows a different diagram. For example, some elements in Basic layer use functionality of the Feature layer, which is not specified in the intended architecture.

Tele

The Tele system was developed with the aid of the Building Block method [KW94]. This method requires that each Building Block resides in a single layer. Furthermore, Building Blocks may use only Building Blocks that reside in lower layers. Figure 6.3 shows an example of Building Block (BB)
usage. According to the rule described above the usage relations marked with a cross are not allowed [Kri95, FKO98]. Note that the Tele system achieves Layering Conformance by means of dedicated tools that support the system’s development.

6.2.4 Method

Each Building Block resides in a single layer, which is defined by the \textit{resides \ Blocks, \ Layers} relation\footnote{The PartOf InfoPack can be extended to extract this information.}. Note that the \textit{resides \ Blocks, \ Layers} relation in fact describes a partition of all the Building Blocks over layers (see also Figure 4.3). The layers are strictly ordered, which is reflected in the relation \textit{< \ Layers, \ Layers}. Furthermore, we need the following relations:

- \textit{imports Files, Files} (from the Import InfoPack)
- \textit{partof Files, Blocks} (from the PartOf InfoPack)

We can then define the following rule with its corresponding violations in RPA:

\[
\begin{align*}
\text{imports Blocks, Blocks} &= \text{imports Files, Files} \uparrow \text{partof Files, Blocks} \\
\text{mayuse Blocks, Blocks} &= <^+ \text{Layers, Layers} \downarrow \text{resides Blocks, Layers}
\end{align*}
\]
rule:
imports Blocks,Blocks ⊆ mayuse Blocks,Block

violations:
v_imports Blocks,Blocks = imports Blocks,Blocks \ mayuse Blocks,Blocks
v_Blocks = dom(v_imports Blocks,Blocks)
v_imports Files,Files = imports Files,Files \ (mayuse Blocks,Blocks \ partof Files,Blocks)

explanation

The mayuse relation describes the import dependencies allowed at Blocks level. All the blocks in a layer may use the blocks of all the lower layers, hence the transitive closure upon Layers,Layers. By lowering the allowed usage at Layers level to Blocks level, we get the mayuse Blocks,Blocks relation. The architectural rule defines that the actual import dependencies (imports Blocks,Blocks) is a subset of the allowed import dependencies (mayuse Blocks,Blocks).

The v_imports Blocks,Blocks relation describes the violating imports between Building Blocks. It consists of the actual import dependencies minus the allowed import dependencies. During the process of resolving disconformance one first wants to know which Building Blocks are involved (i.e. domain of v_imports Blocks,Blocks). After that, more precise information (i.e. closer to the source code) is required in terms of source code files. Therefore, we lower the mayuse Blocks,Blocks relation to the Files level and subtract this from the actual import dependencies in order to find the violating file import statements.

Finally, we have to present the violations in a way that will appeal to the person who has to resolve them. The violating import dependencies can be presented in the same way as Component Dependency (i.e. in a diagram or tables). We can also use colours to distinguish allowed usage and forbidden usage relations in a diagram or table.
6.2.5 Discussion

A violation of the layering conformance rule can be resolved in several different ways. First, we can move a complete Building Block to another layer. Secondly, we can remove violating import statements from the source code files (and move the corresponding code to other files). Or, a combination of the two options may resolve the violations.

In this ArchiSpeCt we have used only binary relations. The use of multi-relations offers more dedicated information when it comes to resolving a violation. For example, a large weight in the violating \texttt{v.importsBlocks.Blocks} relation may indicate that the Building Block resides in the wrong layer. Such a modification of the system is relatively simple, especially when compared with removing import statements (and the corresponding movements of functions or other code) from a number of source files.

Multi-relations can also help define some exceptions to the architectural rules. An example is a system that is strictly ordered while a single Building Block (e.g., the \texttt{Loader} Building Block) may use functionality from higher layers. This extra allowed usage can be incorporated in the \texttt{mayuse} multi-relation \((\ldots, \langle \text{Layer}_1, \text{Layer}_2, 1, \ldots \rangle)\).

6.3 ArchiSpeCt: Usage Conformance

6.3.1 Context

The \textit{Usage Conformance} ArchiSpeCt belongs to the module view. It uses the results of the \textit{Import} (Section 4.6) and \textit{PartOf} (Section 4.7) InfoPacks. It is related to \textit{Component Dependency} ArchiSpeCt (Section 4.9).

6.3.2 Description

The documentation of a software architecture often contains a diagram that shows components and relations between those components. Such a diagram in fact defines the allowed usage between components. Component \textit{Usage Conformance} is achieved when the actual implementation conforms to the allowed usage defined in the documentation.

This ArchiSpeCt is in fact an extension of \textit{Layering Conformance}. We describe precisely which components may use each other, while \textit{Layering Con-
<table>
<thead>
<tr>
<th>Comp</th>
<th>Comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acq</td>
<td>Abb</td>
</tr>
<tr>
<td>Acq</td>
<td>Rec</td>
</tr>
<tr>
<td>Acq</td>
<td>Log</td>
</tr>
<tr>
<td>Rec</td>
<td>Sen</td>
</tr>
<tr>
<td>Sen</td>
<td>Log</td>
</tr>
<tr>
<td>Man</td>
<td>Str</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 6.1: Component Usage Table of Med

formance is based on a more general concept of allowed usage. When both ArchiSpects are applied to a system, one can also check whether the Usage Conformance and Layering Conformance are compatible, even before any code has been written.

6.3.3 Example

Med

The Med system formally describes the usage between components, which is defined in a simple table (which is similar to a RPA-formatted file). Part of the component usage table is presented in Table 6.1 (files of the left component may import files from the right component).

6.3.4 Method

The allowed component usage can be defined manually by an architect or it can be extracted from the architecture documentation. In the case of the Med system, we can simply translate the component usage table into the mayuse_{Comp,Comp} relation. The architectural rule that must hold is defined as follows in RPA (also discussed in [FKO98]):

\[
\text{rule:} \\
\text{imports}_{\text{Comp,Comp}} \subseteq \text{mayuse}_{\text{Comp,Comp}}
\]
violations:

\[
\begin{align*}
\nu_{\text{imports}} & = \text{imports}_{\text{Comps}, \text{Comps}} \setminus \text{mayuse}_{\text{Comps}, \text{Comps}} \\
\nu_{\text{Comps}} & = \text{dom}(\nu_{\text{imports}}_{\text{Comps}, \text{Comps}}) \\
\nu_{\text{imports}}_{\text{Files}, \text{Files}} & = \text{imports}_{\text{Files}, \text{Files}} \setminus (\text{mayuse}_{\text{Comps}, \text{Comps}} \downarrow \text{partof}_{\text{Files}, \text{Comps}})
\end{align*}
\]

The explanation of these formulas is similar to that of the formulas of Layering Conformance. We can also use the same presentation techniques to reveal violations.

### 6.3.5 Discussion

The build process of the Mad system incorporated controlled component usage. An architect maintains a component usage table that describes the allowed usage between components. All the source files of a component reside in separate directories in the file system. Before a file is compiled, the `include-path` for the compiler is automatically set by the build environment according to the entries in the component usage table. Illegal inclusion of files consequently results in a failure of the compiler: file `olise.h` not found.

This approach has a great advantage over the method described above and that is that a developer gets feedback on illegal usage at a very early stage. On the other hand, it imposes a certain organisation of the source code files which may not hold for every system.

The coupling measure (see Section 5.3) and Usage Conformance are related. They both say something about the coupling between a system’s components. Coupling describes a general metric for measuring usage between components that can be applied to any system. In general, the aim is to minimise coupling. In contrast to coupling, Usage Conformance defines exactly which components of a specific system may use each other.

### Reflexion Models

Murphy et al. [MNS95, MN97] introduced reflexion models to discuss differences between a high-level model, as defined by an engineer, and a source model, as extracted from source code. The high-level model describes the
mental model of the engineer, whereas the source model describes the actual implementation of the system. By comparing the relations in the two models one can partition them into three categories: convergences, divergences and absences. Convergences occur in both models, divergences occur only in the source model and absences occur only in the high-level model. The authors used reflexion models to compare a design (high-level model\(^2\)) with an implementation (source model); they concluded that divergences do not adhere to design principles (in order to achieve design conformance).

We can easily translate these ideas into Relation Partition Algebra. Consider a source model consisting of a relation \(U_{SM}\) and a high-level model containing a relation \(U_{HLM}\). We can then define the three categories as follows in RPA:

\[
\begin{align*}
\text{convergences} &= U_{SM} \cap U_{HLM} \\
\text{divergences} &= U_{SM} \setminus U_{HLM} \\
\text{absences} &= U_{HLM} \setminus U_{SM}
\end{align*}
\]

6.4 ArchiSpect: Aspect Conformance

6.4.1 Context

The Aspect Conformance ArchiSpect belongs to the module view. It requires results of the Import (Section 4.6), PartOf (Section 4.7) and Aspect Assignment (Section 5.4) InfoPacks. It is related to the Aspect Coupling ArchiSpect (Section 5.5).

6.4.2 Description

The notion of aspects has already been discussed in Section 1.5.3. We would like to enforce certain dependencies between parts of the system that belong to aspects (see also Section 5.5). The Usage Conformance, discussed in Section 6.3, restricts the usage between components. Aspect Conformance can be seen as an additional means in controlling a system’s

\(^2\)They extracted a high-level model from design documentation (object diagrams).
complexity, by restricting usage between software parts belonging to certain aspects.

6.4.3 Example

Tele

The recovery aspect belongs to all the functions involved in initialising a system or recovering a system from some erroneous state (e.g., due to a hardware failure). After recovery, the system is in a defined state, but during recovery, one cannot rely on a defined state in other Building Blocks. During recovery, a Building Block may therefore only access data and functionality of its own, as illustrated in Figure 6.4.

6.4.4 Method

An architect must define the $\text{maydepend}_{\text{Asps}, \text{Asps}}$ relation, which defines the allowed usage between aspects. Furthermore, we require the relations
imports \textit{files}, \textit{files} (Import InfoPack), \textit{partof} \textit{files}, \textit{comps} (PartOf InfoPack) and \textit{addresses} \textit{files}, \textit{asps} (Aspect Assignment InfoPack). We define the following architectural rule (also discussed in [FKO98]):

\[
\text{depends}_{\text{asps}, \text{asps}} = \text{addresses}_{\text{files}, \text{asps}} \circ \text{imports}_{\text{files}, \text{files}} \circ \text{addresses}^{-1}_{\text{files}, \text{asps}}
\]

\textbf{rule:}

\[
\text{depends}_{\text{asps}, \text{asps}} \subseteq \text{maydepend}_{\text{asps}, \text{asps}}
\]

\textbf{violations:}

\[
\begin{align*}
\text{v}\_\text{depends}_{\text{asps}, \text{asps}} &= \text{depends}_{\text{asps}, \text{asps}} \setminus \text{maydepend}_{\text{asps}, \text{asps}} \\
\text{v}\_\text{imports}_{\text{files}, \text{files}} &= \text{imports}_{\text{files}, \text{files}} \setminus \\
&(\text{addresses}^{-1}_{\text{files}, \text{asps}} \circ \\
&\text{maydepend}_{\text{asps}, \text{asps}} \circ \text{addresses}_{\text{files}, \text{asps}})
\end{align*}
\]

\textbf{explanation}

The \textit{depends}_{\text{asps}, \text{asps}} relation is created by lifting the domain and range of the \textit{imports} relation (as already discussed in Section 5.5). The actual dependencies between aspects (\textit{depends}_{\text{asps}, \text{asps}}) must be a subset of the allowed dependencies (\textit{maydepend}_{\text{asps}, \text{asps}}) to ensure compliance with the rule. The violating dependencies between aspects are all actual dependencies minus the allowed ones. When solving the problem, one wants to know how the violations occur in source code, i.e. violating imports at \textit{files} level.

\subsection*{6.4.5 Discussion}

The following formulas are defined for the \textit{Tele} system, concerning the \textit{recovery} aspect (given a \textit{calls} relation and \textit{addresses} relation at \textit{functions} level):

\[
\text{Funcs}_{\text{recovery}} = \text{addresses}_{\text{func}, \text{asps} \cdot \text{recovery}}
\]
\[
\begin{align*}
\text{recoverts } \text{funcs,funcs} & = \text{calls } \text{funcs,funcs} \upharpoonright \text{dom } \text{funcs,recovery} \\
\text{recoverts } \text{blocks,blocks} & = \text{recoverts } \text{funcs,funcs} \\
& \quad \quad \text{partof } \text{funcs,files} \uparrow \text{partof } \text{files,blocks}
\end{align*}
\]

\textbf{rule:}

\[
\text{recoverts } \text{blocks,blocks} \setminus \text{Id } \text{blocks} \subseteq \emptyset
\]

There is a single exception to the \textit{recovery} rule. The \textit{SysRecovery} Building Block controls the whole \textit{recovery} process. Therefore, it may access all the \textit{recovery} functions of the other Building Blocks. We adapt the above rule as follows:

\textbf{rule:}

\[
(\text{recoverts } \text{blocks,blocks} \setminus \text{dom } \{ \text{SysRecovery} \}) \setminus \text{Id } \text{blocks} \subseteq \emptyset
\]

In Section 5.5 we discussed \textit{Aspect Cohesion} and \textit{Aspect Coupling}. \textit{Aspect Conformance} and these metrics are closely related. They are the metrics which should be maximised or minimised in any system. \textit{Aspect Conformance} defines the exact relations between the aspects of a specific system. One may assume that the architect has considered the \textit{Aspect} metrics in defining \textit{Aspect Conformance}.

\section*{6.5 ArchiSpect: Generics and Specifics Conformance}

\subsection*{6.5.1 Context}

The \textit{Generics and Specifics Conformance} ArchiSpect belongs to the module view. It requires the results of \textit{Import} (Section 4.6) and \textit{PartOf} (Section 4.7) InfoPacks.
6.5.2 Description

The notions of generic and specific components have been discussed in Section 1.5.2. There is a special relationship between these two types of components. A generic component and a corresponding set of specific components belong together semantically. The common functionality resides in the generic component, while the specific functionality resides in various specific components.

Each product in the family comprises (most of) the generic components, while the specific components vary per product. Hence, in a system, a component can only count on the availability of generic components. Therefore, specific components can only be accessed via their corresponding generic component (via a call-back mechanism).

6.5.3 Example

Tele

A switching system (e.g., Tele) must be able to handle different kinds of physical lines to communicate with other switching systems. A dedicated hardware unit (peripheral processing unit, PPU) handles the physical communication with other systems. The central unit of a system contains software to control proper usage of the PPUs. During the development of Tele one does not know which products will ultimately be configured, so one cannot rely on the availability of software that controls a certain PPU.

At a very abstract level, each communication line performs the same functionality, namely communication with other systems. The generic component addresses this abstraction: hiding all the specific characteristics of various communication lines. During initialisation time each specific component subscribes itself to the generic component. The allowed usage relation between generic and specific components is illustrated in Figure 6.5.

Note that the Tele system has completely achieved Generics and Specifics Conformance.
6.5.4 Method

The input\(^3\) for this ArchiSpec consists of the set of generic components (\textit{Generics}) and the set of specific components (\textit{Specifics}). Furthermore, we require the relations: \textit{imports} \textit{Files,Files} and \textit{partof} \textit{Files,Comps}. Note that the \textit{Generics} and \textit{Specifics} describe a partition of \textit{Comps}.

Components are prohibited to import functionality from specific components. One can define this as follows:

\[
\begin{align*}
\text{imports}\ Comps,\Comps & = \text{imports}\ Files,\Files \uparrow \text{partof}\ Files,\Comps \\
\text{importsSpec}\ Comps,\Comps & = (\text{imports}\ Comps,\Comps \mid \text{ran} \ Specifics) \setminus \\
& \quad \text{Id}\ Specifics
\end{align*}
\]

\textbf{rule:}

\[
\text{importsSpec}\ Comps,\Comps \subseteq \emptyset
\]

\textbf{violations:}

\[
\begin{align*}
\text{v.imports}\ Comps,\Comps & = \text{importsSpec}\ Comps,\Comps \\
\text{v.}\ Comps & = \text{dom}(\text{v.imports}\ Comps,\Comps) \\
\text{v.imports}\ Files,\Files & = (\text{importsSpec}\ Comps,\Comps \downarrow \text{partof}\ Files,\Comps) \cap
\end{align*}
\]

\(^3\)An InfoPack can be defined to provide this information.
import \( Files, Files \)

**explanation**

The \( importsSpec \) relation represents all the imports of specific components (excluding imports of itself). The architectural rule is satisfied if and only if this relation is empty. The violating components consist of the components that import a specific component, i.e., \( v_Comp \). The violating imports at \( Comps \) level can be lowered to \( Files \) level to obtain all the possible violating imports at this level. The intersection of this intermediate result with the actual imports \( (imports_{Files, Files}) \) leads to the actual violating imports \( (v_imports_{Files, Files}) \).

### 6.5.5 Discussion

An alternative definition of the above rule is defined as follows:

\[
\begin{align*}
ExpSpecifics &= \text{dom} (\text{imports}_{Comps, Comps} \setminus \text{Id}_{Comps}) \\
ExpGenerics &= Comps \setminus ExpSpecifics
\end{align*}
\]

**rule:**

\[
\begin{align*}
ExpGenerics &\subseteq Generics \\
ExpSpecifics &\subseteq Specifics
\end{align*}
\]

**explanation**

In fact, we define a pattern that recognizes specific components: all the specific components import functionality only from generic components or from themselves. Given this characteristic, the specific components should be defined in the \( ExpSpecifics \) set; the other components are therefore \( ExpGenerics \). The rule consists of verifying whether all the recognized generic (specific) components are indeed generic (specific).

### 6.6 Architecture Verification in Action

In the previous sections we discussed a number of ArchiSpects relating to the managed architecture. Architecture conformance can be achieved
when the application of these ArchiSpects to an existing system results in the satisfaction of the architectural rules. The ArchiSpects discussed in this chapter should be seen as examples of how ArchiSpects of the managed architecture can be defined. Although the presented ArchiSpects can be useful for many systems, each system may require its own ArchiSpects to enforce architecture conformance. Below we will briefly discuss the introduction of an architecture verification process in real environments.

The first step toward achieving architecture conformance consists of formalising the architecture decisions (as described in the documentation and/or stored in the heads of architects). As shown in this chapter, relation partition algebra can be applied to formalise a number of these decisions. In addition to these rules, violations must also be defined, in such a way that they support a developer in resolving possible disconformances.

The second step concerns the incorporation of architecture verification in the development process. Automation of the verification process is required to be able to successfully introduce it. The execution of InfoPacks and ArchiSpects should be incorporated in the Build Process. We can distinguish three general points in time at which architecture conformance can be introduced:

- **early**: as soon as a developer has written some code the applicable architectural rules are checked (in parallel to e.g. a compile job).
- **mediate**: at the time the rules are checked a module is “checked in” in the source code management system. If errors are detected, the module involved is not accepted by the source code management system.
- **late**: an architect initiates architecture verification at certain times during system development (e.g. by starting a rule checking program). If disconformance is established, the architect must submit a Problem Report.

It will depend on the situation which of the three alternatives will have to be applied. In general, a system should be verified as early as possible. If possible, a developer should be given feedback immediately after he or she has broken an architectural rule. If one is not familiar with architecture verification, one may prefer to check architecture conformance at a late stage. That way, the introduction of many changes in the development

---

4 Rules that cannot be formalised (an example is given in Section 6.1) should be validated in e.g. review sessions.

5 For example, we can create a special “thread” in the Build Process activities to execute InfoPacks and ArchiSpects.
process (and the related tools) is avoided, so the continuity of development is guaranteed. After a while, one can shift to an early stage (which will affect the development environment more).

The first time the architectural rules of an existing system are checked, many violations may be expected. It will be practically impossible to solve all the violations immediately. Therefore, one should first only verify the architecture and identify the violations. In a next verification session, the newly detected violations can then be compared with the previously detected violations. But this time it must be ensured that no new violations are introduced (in other words, that the number of violations does not increase). The violations can then be resolved to improve the actual architecture at a convenient time. This way, architecture conformance can be ensured without affecting the schedules of product deliveries.