UvA-DARE (Digital Academic Repository)

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.

UvA-DARE is a service provided by the library of the University of Amsterdam (http://dare.uva.nl)
Chapter 4

Described Architecture

In the next three chapters we discuss three levels of the SAR method (respectively described, redefined and managed level). In this thesis, as already mentioned, for each of these levels we focus on the code view and module view of software architecture. Here, we start with the described level of SAR.

4.1 Introduction

Figure 4.1 shows an abstract view on our software architecture reconstruction (SAR) method. The main and most explicit source of information for reconstructing a software architecture is the source code. The source code can be analysed and be reduced to manageable units of information, which we call Info Packs. Information that cannot be extracted from the source code must be supplemented with information from e.g. software architects. Relation Partition Algebra is the model underlying most of the ArchiSpects. The results of Info Packs are expressed in a simple notation, namely the RPA-file formats introduced in Section 3.6.3. Info Packs yield intermediate results that are used to construct ArchiSpects.

The described software architecture of an existing system consists of a set of ArchiSpects, each containing a relevant aspect of the software architecture. The main aim of the resulting described architecture is to support comprehensability of software architecture for (new) software developers and architects. In this chapter we will focus solely on ArchiSpects and their required Info Packs that are related to the module and code view of
architecture. The process of reconstructing the software architecture at a
described level is called reverse architecting [Kri97].

Figure 4.2 shows the Info Packs (rectangles) and ArchiSpects (hexagons)
initially required to describe a software architecture. Solid lines between
Info Packs and ArchiSpects (InfoPack) indicate that the output of the In-
fo Pack is input for the ArchiSpect (InfoPack). Dotted lines mean that there
is a relationship between the two, which is not expressed in terms of input
and output results (the relationship is clarified in the description of the
Infopack or ArchiSpect). In the SAR method, Info Packs and ArchiSpects
are classified per architectural view, which is indicated by the two large
boxes: code view and module view.

In this chapter we will discuss (see also Figure 4.2) the Info Packs Files, Im-
port, Part-Of and Depend of the code view, the ArchiSpects Source Code
Organisation and Build Process of the code view and the ArchiSpects Soft-
ware Concepts Model, Component Dependency and Using and Used Inter-
faces of the module view. The Info Packs and ArchiSpects will be discussed
according to the fixed scheme (context, description, example, method, dis-
cussion) introduced in Section 2.5.2. The Info Packs and ArchiSpects will
be discussed in the order in which they should be applied to a system.

In recent years, we have applied these ArchiSpects to a number of real
systems; the results of some of this work will be used to illustrate these
ArchiSpects:

- in 1994/1995 we analysed the Tele system [KW94], which is a public
Figure 4.2: Overview of Described Architecture
telephony switching system;
- in 1996 we analysed the *Med* system [MK97], which is a medical imaging system;
- in 1997/1998 we analysed the *Switch* system [Med98], which is a private telephony switching system.
- in 1998 we analysed the *Comm* system [Kri98], which is a management system for controlling digital video communication systems.

The modular approach of our SAR method makes it possible to apply those ArchiSpects and InfoPacks which are relevant for your case. We will now repeat how the different sections of each ArchiSpect and InfoPack is organised: In the first section we present the relations with other ArchiSpects and InfoPacks, which is also indicated in Figure 4.2 by means of lines. In the second section, a general description is given including a motivation of why we should apply this ArchiSpect (InfoPack). For the reader’s convenience, we illustrate these ideas with examples from practice. In the fourth section, we describe the steps to be taken to reconstruct the ArchiSpect or InfoPack. The last section discusses related issues or, where appropriate, related work.

### 4.2 ArchiSpect: Software Concepts Model

#### 4.2.1 Context

The *Software Concepts Model* belongs to the module view of software architecture. The ArchiSpect *Source Code Organisation* (Section 4.3) and InfoPack *Part-Of* (Section 4.7) are related to this ArchiSpect.

#### 4.2.2 Description

System documentation includes many domain-specific (or system-specific) terms. For example, in one system, a component will be just a term for a group of smaller programming units, and, with another, it will refer to COM components [Box98] (including its dynamic binding machinery). A good understanding of these concepts is needed for almost any reconstruction activity. The concepts help to classify functionality during discussions, e.g. a subsystem is a far more important architectural notion than a method or function.

The *Software Concepts Model* concentrates on the most important software
concepts and their inter-relationships. Files, functions, types and relations, such as function *accesses* data, function *calls* function and component *contains* function, are often included. Furthermore, the various abstraction levels of software parts (decomposition levels) are described in this ArchiSpect. We will expound this ArchiSpect by providing a number of examples from practice.

The result of this ArchiSpect consists of a UML class diagram [Fow97] containing these concepts and their inter-relations.

The *Part-Of* InfoPack fills out which of the system’s entities belong to the various decomposition levels plus the various *partof* relations between these levels. The software concepts must be mapped onto real items in the software code organisation, e.g., a *subsystem* resides in a directory, which is discussed in the *Software Code Organisation* ArchiSpect.

### 4.2.3 Example

**Tele**

Figure 4.3 shows the concepts and relationships between software concepts of the *Tele* telecommunication system in a UML class diagram.

The system consists of a number of *subsystems*. Each *building block* belongs to exactly one *subsystem*. A *building block* is an aggregation of *files*; each file addresses a single *aspect* (see Section 1.5.3). Furthermore, a *product* in the product family is described as a *parts* list of *building blocks* (and hardware elements). *Building blocks* reside in *layers* which are strictly ordered (<); see also Section 1.5.1. The concepts at the bottom of the class diagram describe programming concepts and relationships between them.

**Med**

The *Software Concepts Model* of the *Med* system is depicted in Figure 4.4. The system consists of a number of *subsystems*. Each *subsystem* is an aggregation of *components* and a *component* is divided into several *packages*. A number of *files* are contained in a *package*. A few programming concepts have been represented at the bottom of the diagram. An *archive* is a set of subsystems; this notion was introduced at a later stage in *Med*’s life-cycle.
Figure 4.3: Software Concepts Model of Tele
The creation of the *Software Concepts Model* is an iterative process. It is important to read relevant system documentation, but often it is also necessary to discuss various non-documented concepts. In particular, relationships between concepts must be made explicit, which will involve discussions with architects. The results of these activities can be presented in a class diagram notation of the UML language [Fow97]. During this iterative process, the class diagrams should serve as input for discussion.

### 4.2.5 Discussion

Software concepts play an important role in the definition of a software architecture. If the software architecture has been properly defined, one can easily obtain the software concepts from the documentation. As most products usually have a long life-cycle, we may assume that new concepts were not initially foreseen and they are consequently not completely and/or properly applied in the software today. An example of a concept that was
introduced only at a later stage is the archive in the Med system.

In particular, concepts that do not have a counterpart in a programming language may become a subject of discussion. Different architects may think differently about the semantics of such a concept. For the overall development it is of importance to make such fuzzy concepts more concrete. This may even result in the introduction of new concepts to achieve a better Software Concepts Model.

The UML associations aggregation and composition represent a special relation; they describe the decomposition tree at a generic level. In the presented examples we do not have recursive definitions in the decomposition tree. In practice, such recursions may exist, e.g. a component may contain either a set of files or a set of components.

4.3 ArchiSpect: Source Code Organisation

4.3.1 Context

The Source Code Organisation belongs to the code view of software architecture. The ArchiSpects Software Concepts Model (Section 4.2), Build Process (Section 4.4), InfoPacks Files (Section 4.5) and Part-Of (Section 4.7) are related to this ArchiSpect.

4.3.2 Description

The Source Code Organisation ArchiSpect consists of three parts: description of the way in which source files are stored, the mapping of source code onto software concepts and a description of the process of retrieving files from the configuration management system [BHS80, Bab86].

Many deliverables (source code, design documents, etc.) are produced during system development. These deliverables must be easily accessible to all the developers. Sometimes, previous versions of documents may also be requested. Different people must be able to modify the same documents, but such concurrent access may not result in loss of information. The functionality referred to above is offered by most of the configuration management systems, e.g. Continuus [Con] or ClearCase [Cle]. Most of the configuration management systems are based on a file system (functionality is implemented by locking files, storing versions in different directories,
Source code is one of the deliverables that is stored in the configuration management system; see also the Build Process ArchiSpec. A list of source code files that belong to a certain release is needed to be able to analyse source code (see Files InfoPack). A more general description of the location of files, taking into account different versions, is contained in Source Code Organisation. Also included is the mapping of elements of this ArchiSpec onto concepts of the Software Concepts Model.

During system development, files occur in different development states, e.g. a file is reserved by a developer who extends or modifies it. After the developer has finished, he or she consolidates the file by restoring it in the archive. Describing these development states, their possible transitions and the people who initiated these transitions provides insight into the development process.

### 4.3.3 Example

**Med**

Figure 4.5 shows a model of the Source Code Organisation of the Med system in a UML class diagram [Fow97]. In general, a software archive contains various versions of the system. Each version consists of a number of directories in which files reside. Every time the system is released, a copy of the current version is created.

Table 4.1 shows how some concepts of the Software Concepts Model, (see Figure 4.4) are reflected in the Source Code Organisation (Figure 4.5). A system is reflected onto a version and a component onto a directory. Filenames start with a special prefix of four characters which refers to the package name. The concepts subsystem and archive are not explicitly reflected in the Source Code Organisation.

Figure 4.6 shows the development states of files and possible transitions in a state diagram (UML). The developer (integrator) is in control of the states and transitions with a bold (italic) font style. For example, a developer decides to grab a file from the archive. When he or she has finished modifying the grab-ed file, he or she preptake-s the file which becomes ready for archiving. The integrator take-s the file and tries to build an alpha version of the system. In the event of problems he or she may reject the file; the reject-ed file must be accept-ed by the (latest preptaking) developer who
Figure 4.5: Source Code Organisation of Med

Table 4.1: Software Concepts Model vs. Source Code Organisation of Med
4.3 ArchiSpect: Source Code Organisation

Figure 4.6: Development States and Transitions of Med Files

has to correct the file. If the integrator decides that the file satisfies, he or she consolidate-s the file in the archive.

4.3.4 Method

The Source Code Organisation is often well described in documents. The documentation of the configuration management system (CMS) contains additional information. The first reconstruction activity to be performed consists of reading CMS documentation and filtering the proper information. Additionally, one can interview system integrators and people concerned with configuration management. The Source Code Organisation can be described using class diagrams of the UML language [Fow97]. The next step is to describe the mapping from software concepts onto the entities of this ArchiSpect and identify the gaps in this mapping scheme.

The development states and a development transition diagram can also be extracted by interviewing system integrators and/or reading the appropriate documentation. State diagrams of the UML language can be used to document it.

4.3.5 Discussion

The top-level concepts (e.g., subsystems) in the Software Concepts Model are often not reflected on any tangible item of the Source Code Organisation. There is a risk of the meaning these concepts, which exist only at a more conceptual level, degenerating with time.
Mapping between software concepts, *Software Concepts Model*, and source code items, *Source Code Organisation*, as depicted in e.g. Table 4.1, is relevant for analysing a system. The results of extraction tools must be mapped onto software concepts to be able to ascend in the system’s decomposition hierarchy.

Some steps of the SAR method can be automated, but also integrated in the development process. Knowledge of the *development states* and *transitions* is required for integrating SAR steps in the development process.

### 4.4 ArchiSpect: Build Process

#### 4.4.1 Context

The *Build Process* ArchiSpect belongs to the code view of software architecture. This ArchiSpect is related to the *Source Code Organisation* (Section 4.3) and the *Depend InfoPack* (Section 4.8).

#### 4.4.2 Description

The *Build Process* ArchiSpect includes a description of how various pieces of code (see also *Source Code Organisation* ArchiSpect) must be processed to derive all the executables that comprise the system. This process can be split into a number of smaller build activities, each describing how to create a (intermediate) result from inputs. The different intermediate results are input for new build activities. In Figure 4.7 the cascade of smaller build activities has been divided into four categories: *pre-compile, compile, link* and *post-process.*
The *pre-compile* category consists of code generation activities, e.g., generation of scanners and parsers from higher-level descriptions. The *compile* category consists of source-code-compilation activities. Each input is either a result of a *pre-compile* activity or it is created by hand. The *link* category consists of linking the results of the compile activities into one or more executables. The *post-process* activities consist of gathering all the required files (executables, resource files, bitmaps, help texts, etc.) and loading them on the target system.

If a file changes (as indicated by a file’s modification date), all its derivatives must be rebuilt. To rebuild an entire system, this rebuild process must be recursively applied, so derivatives of modified derivatives must also be rebuilt, and so on. There are several tools for supporting this mechanism, of which the *make* [Fel79] utility is probably the best-known. All these tools work with a build description file that contains the dependencies between the various files and a description of how to create the results by defining commands which must be executed, e.g. `CC -I .../finance ajax.c`. After some modification a *build* tool will update all the (intermediate) derivatives by executing the proper commands.

### 4.4.3 Example

**Med**

The *Build Process* of the *Med* system is depicted in Figure 4.8. A build description has been distributed amongst different files (e.g. `acq.opt`, `acq.od` and `acq.tgt`), each of which is responsible for a certain task. The dotted arrows indicate how these build description files affect the various build activities. With this system the *pre-compile* activities have been merged with the *compile* activities and they are therefore not explicitly depicted. The *imports* relation is in fact part of a *compile* activity (a pre-processor of a compiler). The *post-process* activity consists of *target*-ing various files on the system.

Every night all the files that are in the *archive* or *alpha* development states (see Figure 4.6) are built. The system integrator is responsible for starting the whole build procedure, which is in fact completely automated, every evening. Early in the morning, after a successful build process, the system’s main features are briefly tested. This test lasts approximately half an hour. It is executed just before most of the developers arrive at work. After the test, some files are marked as *reject*-ed while others are stamped *archive*. 


This process of building and testing a system is similar to Daily Build and Smoke Test [CS95].

4.4.4 Method

The build process is well documented for most systems. One should therefore read the appropriate documents including discussions with developers. The main task is to develop an abstract model of this process. In addition, consulting the build description files may also be of help in modelling build process; see also the Depend InfoPack. We experienced that cooperation with integrators during the integration test helps in shaping this ArchiSpect.

4.4.5 Discussion

In one of the systems we investigated the build process is described in a generic way. For most of the files a similar command must be executed to compile the source code. In fact, one can define such a generic command per programming language. Similar statements hold for determining the dependencies between the various files. Per language, a tool can determine
the files on which the compilation of a file depends; see also the *Import* InfoPack. Deviations from the standard way of compilation can be defined per file. The filename, the generic command, the programming language and the deviations from the generic compile command can be stored in a database. A dedicated program can generate a build description file from the records in this database. A *build* tool can then execute it. An advantage of this approach is that new compilers can be introduced simply by changing a single generic compile command.

### 4.5 InfoPack: Files

#### 4.5.1 Context

The *Files* InfoPack is part of the code view of architecture. Knowledge of the *Source Code Organisation* ArchiSpec (Section 4.3) is needed and the *Depend* (Section 4.8) and *Import* (Section 4.6) InfoPacks use the results of this InfoPack.

#### 4.5.2 Description

The *Source Code Organisation* ArchiSpec discusses the organisation of files in general terms. The *Files* InfoPack extracts all the source files (all the files created by humans) required to construct a system. The results of this InfoPack serve mainly as input for other InfoPacks. These files can be classified in different categories:

- *Files*, the files of a (version of the) system created by humans;
- *HeaderFiles*, files that specify functions, variables, etc.;
- *BodyFiles*, files that define functions, variables, etc.;
- *ResourceFiles*, files that define help texts, pictures, etc.;
- *BuildFiles*, files that define (part of) the build process;
- *Exts*, extensions of file-names, e.g. *java, cpp*;
- *Cats*, categories of files, e.g. *C-source*;
- *PhFiles*, physical file-names, i.e. the complete name of the file on the file system, e.g. */dev8/ist9/user/krikhaar/med/ver8/ajax.c*.

and the following relations:

- *typed Exts,Cats*, a relation that maps elements of *Exts* onto elements of *Cats*;
• \(\text{typed}_{\text{Files,Exts}}\), a relation that maps elements of \(\text{Files}\) onto elements of \(\text{Exts}\);

• \(\text{located}_{\text{Files,PhFiles}}\), 1-to-1 mapping of \(\text{Files}\) onto their physical locations (\(\text{PhFiles}\)) in a file system (and vice versa).

4.5.3 Example

In view of their sizes, we are unable to give the sets and relations of this InfoPack of existing systems. However, for two systems, we present some related information.

Switch

For the \textit{Switch} system it was easy to determine the system's files involved. All the files created by humans of each version are located in a single directory in the file system. The source files consist of C and C++ files having the extensions .h and .hpp, respectively, for the header files and .c and .cpp, respectively, for the body files.

Med

The \textit{Med} system consists of thousands of files (\(\text{Files}\)). Over 60 different file extensions were found in the system (\(\text{Exts}\)). Some of them exist only because of the system's history (legacy). The whole list of extensions can be grouped into ten types of extensions (\(\text{Cats}\)).

4.5.4 Method

The method for extracting the results of this InfoPack depends very much on the system at hand. The \(\text{Files}\) set consists of all the files in the configuration management system which belong to a single release and which were created by humans. In terms of configuration management, these are all files that can be checked in and checked out. How we can construct such a list will depend on the configuration management system. Other techniques consists of analysing directories and using file-name extensions to determine whether a file was created by human.

The extension of a file name often indicates the type of information contained in the file. One can derive the relation \(\text{typed}_{\text{Files,Exts}}\), which describes
the relation between files and their extensions. The file-exts program (listed in Section A.1) creates this relation from the set of Files.

The Files can be partitioned into a number of sets: HeaderFiles, BodyFiles, ResourceFiles and BuildFiles. The first three sets are the files that eventually appear in some (derived) way in the running system. The BuildFiles are different in the sense that they indirectly belong to the source code: they are the build description files discussed in Section 4.4. One can calculate the various sets on the basis of file-type extensions. For example, given that all HeaderFiles have the extension .h or .hpp, we can calculate as follows in RPA:

\[
\begin{align*}
\text{HeaderExts} & = \{h, hpp\} \\
\text{HeaderFiles} & = \text{dom}(\text{typed}_{\text{Files,Exts}} | \text{ran} \text{ HeaderExts})
\end{align*}
\]

Table 4.2 gives an example of the typed_{Exts,Cats} relation. Each extension is assigned to a single category. This relation must be constructed by hand, in cooperation with an architect.

For the purpose of readability, it is convenient to use short file names instead of full file names (i.e. a device name plus directory name plus base name of a file). It is important to ensure that the resulting file names are unique, but one must also be able to find the file in question in the filesystem at any requested time (i.e. physical file name). The relation located_{Files,PhFiles} is a function from the (unique) file name to the physical file name. This
relation can be derived by removing the file name’s first part of the full file name as much as possible and preserving the file name’s uniqueness.

We can partition all the files according to the categories (Cats). The part-of relation \( (\text{typed}_{\text{Files},\text{Cats}}) \) that describes this relation can be calculated as follows in RPA:

\[
\text{typed}_{\text{Files},\text{Cats}} = \text{typed}_{\text{Exts},\text{Cats}} \circ \text{typed}_{\text{Files},\text{Exts}}
\]

4.5.5 Discussion

It is important to carefully check the extraction results, especially when they are based on heuristics and/or line-oriented Perl [WCS96] scripts (see also discussion in Section 4.6.5). An example of a heuristic is that all files with the extension \(.hlp\) belong to the set of ResourceFiles.

Some checks can be performed at an early stage of analysis already. For example, one can check whether each existing file extension (Exts) belongs to some category (Cats). We can express this in an RPA formula:

\[
\text{ran}(\text{typed}_{\text{Files},\text{Exts}}) \subseteq \text{dom}(\text{typed}_{\text{Exts},\text{Cats}})
\]

4.6 InfoPack: Import

4.6.1 Context

The Import InfoPack belongs to the code view of software architecture. The Files InfoPack (Section 4.5) is used as input and the Component Dependency (Section 4.9) and Using and Used Interfaces (Section 4.10) Architectures use the results of this InfoPack.

4.6.2 Description

To be able to manage large systems, one must divide the software into several separate compilation units. These units use each other by e.g., calling functions, so they require knowledge of each other. A generally accepted concept is to distinguish two parts for each unit: a header part, containing
declarations of e.g. variables and signatures of functions, and a body part, containing the implementation of the names declared in the header. If one unit wants to use another unit, it will import the unit’s header information.

In the programming language C [KR88] (C++ [ES90]), header information and source code are reflected in different files. Historically, the files have the suffixes .h and .c, but, strictly speaking, any file extension may be used. Although not required, it is preferable to ensure a one-to-one correspondence between the header and the body file, so that champion.h contains declarations of names that are implemented in champion.c; nothing less and nothing more than that. For clarity one should define such rules in the coding standards (as this will ensure more conceptual integrity).

Although not absolutely demanded by C/C++-compilers, a header file should contain only the following information:

- macro declarations
- type declarations
- class declarations (C++ specific)
- function (method) declarations, i.e. signatures of functions (methods)
- variable declarations

Such concepts exists for other languages too. A fine example is the Modula-2 programming language [Wir83], which explicitly handles the notions of definition modules and implementation modules. The syntax of this language ensures that the definition module and the implementation module both contain the right type of information.

This InfoPack results in the relations imports Files,Files and partof Files,Units. The imports relation contains tuples (FileX, FileY), where FileX imports FileY. The partof Files,Units relation groups a header file and a body file into a single entity, named unit. The latter relation is of use in reconstructions only if the notions of header and body file have been properly applied (as described above).

4.6.3 Example

We give a fragment of a C/C++ source file (ajax.c) as an example.

```
#include "champion.h"
#include <string.h>
#include "../finance/stock.h"
```
### Table 4.3: \textit{import} files

<table>
<thead>
<tr>
<th>Files</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>ajax.c</td>
<td>champion.h</td>
</tr>
<tr>
<td>ajax.c</td>
<td>string.h</td>
</tr>
<tr>
<td>ajax.c</td>
<td>stock.h</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

In this example there are three \texttt{#include} statements, each with its own specifics. The three header files are literally included before the compiler starts compiling the \texttt{ajax.c} file. The compiler\footnote{In fact, a pre-processor of the compiler searches the files and literally includes them.} searches for the included files in the file system using an \texttt{include-path}. An include-path is an ordered list of directories (in the file system). The compiler searches for an include file by looking for it in the directories (in the given order) as defined in the include-path. The order of the directories in the include-path is relevant when a file occurs more than once in the file system.

The first \texttt{#include} statement refers to \texttt{champion.h}. The compiler searches for this file, at first in the current directory, and secondly via the include-path. In the second \texttt{#include} statement, the file-name is enclosed by angles (< >). The compiler consequently searches for it only via the include-path (ignoring files in the current directory). In the third \texttt{#include} statement, the file-name is preceded by a relative path (\texttt{./finance/}). This is in fact similar to the first statement, except for the relative path which is taken into account during searching.

The results of the import extraction of this example are given in Table 4.3. Table 4.4 shows the \textit{partof} \texttt{Files,Units} relation.

### 4.6.4 Method

The method in constructing this InfoPack comprises the following steps:

- starting with a list of files belonging to the system; this list is a result of the \texttt{Files} InfoPack;
- determining the include-path per file; the \textit{Build Process ArchiSpec}t describes where one can find this information;
- extracting the include statements per file and determining the included file (using the include-path);
<table>
<thead>
<tr>
<th>Files</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>champion.c</td>
<td>champion</td>
</tr>
<tr>
<td>champion.h</td>
<td>champion</td>
</tr>
<tr>
<td>ajax.c</td>
<td>ajax</td>
</tr>
<tr>
<td>ajax.h</td>
<td>ajax</td>
</tr>
<tr>
<td>stock.c</td>
<td>stock</td>
</tr>
<tr>
<td>stock.h</td>
<td>stock</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 4A: \textit{part of} \(\text{Files}, \text{Units}\)

- reflecting the information in the \(\text{imports Files,Files}\) relation file;
- determining the units and constructing \(\text{part of} \text{Files,Units}\).

In practice, many peculiarities make extraction slightly more difficult than described above. For example, different operating systems have different file systems with their own filename conventions (e.g., \textit{Windows NT}, does not distinguish cases in file names, \textit{Unix}, however, is case-sensitive with respect to file names). When different file systems are used, case sensitivity problems must be solved first (e.g., by converting cases).

In the following sections we will discuss the extraction of an \(\text{imports}\) relation from source code written in different programming languages: C/C++, Java, Objective-C and CHILL. Many parts of the systems we investigated have been implemented in these programming languages. The discussion of the extraction will also serve to illustrate how an \(\text{imports}\) relation can be extracted from source code written in other languages.

\textbf{C and C++}

The C++ language [ES90] is an object-oriented version of the C language [KR88]. The import mechanisms of the two languages are the same. First, we strip comments from the source files (the \textit{comment-strip} program is presented in Section A.3). Secondly, we extract the inclusion of files (the \textit{C-imports} program is presented in Section A.4). The \#include statement contains the file name of the included file. This file name is enclosed between a pair of quotes (""") or between angles (< and >). The extraction program uses these facts to filter the proper information, which results in an \(\text{imports} \text{Files,Files}\) relation.
One can also construct the partofFiles,Units relation on the basis of filenames. The relation is based on a naming convention: the names of header and body files are the same except for their suffixes, .h and .c, respectively (the program is given in Section A.2).

Java

The Java language [Jav, Web96] supports an import statement enabling the use of other classes. The Java compiler searches for the imported classes via the CLASSPATH environment variable. The mechanism is similar to the include-path mechanism of C/C++.

We give an example (ajax.java) to illustrate various import statements of Java:

```java
import Player;
import traffic.transport;
import car.*;
import java.awt.Button;

class ajax
```

The first import statement asks for the Player class, which means that the compiler searches for a Player.class file. This file must reside in one of the directories defined in the CLASSPATH.

The second import statement defines that the transport class from the traffic package is imported. The compiler searches for transport.class in a traffic/sub-directory of one of the directories in the CLASSPATH.

In the third import statement, all the classes of the car package are imported (they are present in CLASSPATH's car/sub-directory) by using a wildcard (*). The fourth import statement imports from the java.awt package the Button class. The compiler searches for a Button class in a java/awt/sub-directory of CLASSPATH.

The Java language also contains a class-grouping mechanism called packages. The first statement of a Java source file may be a package declaration, which means that all the classes defined in the file belong to that defined package.

---

2In Java a class file is the compiled version of the accompanying java source file.
An extraction program is much simpler when comments have already been removed from the source. Java comments are equivalent to C++ comments, so they can be stripped with the `strip-comment` program (presented in Section A.3). An $\text{imports}_{\text{Classes,Classes}}$ relation (an example is given in Table 4.5) is the result of the $J$-$\text{imports}$ extraction program (presented in Section A.5). The $\text{Classes}^+$ refers to an extended set of classes; the wildcard notation has not yet been resolved. The $J$-$\text{package}$ extraction program (presented in Section A.6) generates a $\text{defines}_{\text{Classes}^+,\text{Classes}}$ relation (an example is given in Table 4.6). So, given the class definitions per package, we are able to resolve the wildcard notation resulting in an $\text{imports}^3$ relation:

$$\text{imports}_{\text{Classes,Classes}} = \text{imports}_{\text{Classes,Classes}} \circ \text{defines}_{\text{Classes}^+,\text{Classes}}$$

---

$^3$Public classes and file names are in fact interchangeable in Java, so we can see it as an $\text{imports}_{\text{Files,Files}}$ relation.
Objective-C

Objective-C [CN91] is an object-oriented version of the programming language C. It is possible to translate Objective-C code into C code with the aid of a relatively simple translator; the resulting C file can then be compiled by a C-compiler. Various Objective-C compilers use this strategy\(^4\).

Objective-C units can use functionalities of other units by importing the corresponding header file (as in C). We give an example:

```c
#import <ReportGenerator.h>
#import <ReportDefinitions.h>
```

The *ObjC-imports* extraction program (presented in Section A.7) is an adapted version of *C-imports*. It filters `#import` instead of `#include` statements.

CHILL

The programming language CHILL [ITU93, SMB83], CCITT High Level Language, is used particularly to build telecommunication systems. The language contains state-oriented constructs. There are several dialects of this language, but, fortunately, these derivatives do not differ in their import mechanisms. We give a CHILL example:

```chill
Subscriber: MODULE

GRANT
    counting, connect_me;

SEIZE
    make_connection;

SEIZE
    PortHandler ALL;

END Subscriber;
```

A module can export names (such as functions, types and variables) with a `GRANT` statement (`counting, connect_me`). A module can only use names

\(^4\)The first C++ compilers also used this strategy to compile C++ files.
(make_connection) of another module by importing them with a SEIZE statement. It is also possible to import ALL the (exported) names of a module (PortHandler).

If strict coding standards are applied, one may be able to extract information with a Perl [WCS96] program, otherwise a dedicated parser is required.

4.6.5 Discussion

We have given extraction programs for import statements in C/C++, Java and Objective-C programs. For systems that mix a number of programming languages one can concatenate the results of the various programs. While analysing different systems, we found that the extraction programs sometimes had to be changed a bit. For example, when operating system environment variables are used within C include statements, one must interpret these variables.

#include "BAS_ENV:string.h"

In the above example, the environment variable BAS_ENV must be resolved first to determine the physical location of the file that is imported. In this particular case it was fairly easy because this environment variable name leads directly to the directory involved (i.e. BAS/).

The given extraction programs are implemented in Perl [WCS96]. A Perl program is interpreted (by a Perl interpreter); one can easily modify it. Therefore, it is very handy during the process of analysing software. On the other hand, these programs are based on many assumptions concerning the layout of the input, which makes them error-prone. One should therefore carefully check the results of these programs.

Reverse engineering tools are able to extract a fair amount of source-code-related information (e.g. function calls, variable access, but they do not analyse function pointers). We can process the output of these tools to obtain, e.g. an import relation. For example, the (intermediate) output of QAC [Pro96] (a commercial tool that checks the quality of C programs) can easily be translated into RPA-formatted files. The QAC-imports program (presented in Section A.8) filters appropriate statements from QAC output and generates an imports <Files,Files> relation.

There is a major difference between the Perl approach and the QAC (or any other reverse engineering tool) approach. QAC parses the complete
source code, taking into account all kinds of pre-processor settings (like \#define). This means that for certain compiler settings parts of the code are not parsed (e.g., code between \#ifdef and \#endif statements. This may also result in skipping of the inclusion of header files. Note that the various products in a product family are often distinguished by including product-dependent header files (using the \#ifdef construct). For a complete analysis of the different products we have to extract all the included files.

4.7 InfoPack: Part-Of

4.7.1 Context

The Part-Of InfoPack belongs to the code view. It is related to the Software Concepts Model (Section 4.2) and Source Code Organisation (Section 4.3) ArchiSpects. The results are input for the Component Dependency (Section 4.9) and Using and Used Interfaces (Section 4.10) ArchiSpects.

4.7.2 Description

Each large system is decomposed into various parts; these parts can in turn be decomposed into smaller parts; see also the Software Concepts Model ArchiSpect. This form of decomposition is applied a number of times until the level of statements is reached. Programming languages offer only a few levels of decomposition: statements are grouped in functions and functions reside in classes or files. We could imagine a counterpart for each software concept in a programming language concept. For example, layers (see Section 1.5.1) are often applied for large systems, but programming languages do not support this concept.

All decomposition levels should be reflected in the system's code view in some way, as already indicated in Section 4.3. For example, one can use directories in the file system to reflect the decomposition hierarchy. Special comments in the headers of source files can also be used to reflect the decomposition level(s). This may easily lead to the introduction of errors because developers may forget to maintain headers.

\footnote{Some languages offer more grouping possibilities Java e.g. includes the concept of packages.}
Figure 4.9: Implementation of Decomposition Levels Med

/*
* Subsystem: Operating System
* Component: Event Handling
*/

4.7.3 Example

Med

The subsystem and component decomposition levels of the Med system (e.g., Est and Acq, respectively) map onto directories in the source code organisation; see Figure 4.9. The Package level is reflected in the applied file name convention of source code files (encoded in the prefix of the file name).

Tele

The source code files appear in a single directory. The decomposition hierarchy is reflected in the documentation structure (which can be automatically extracted by analysing directories on the file system).

4.7.4 Method

The result of this InfoPack consists of a number of partof relations, according to the part-of levels generally described in the Software Concepts
Model. If the Source Code Organisation reflects parts of the decomposition hierarchy, one can derive the partof relations by inspecting the directories (a program is presented in Section A.9). Comments in the headers can moreover be analysed by simple Perl [WCS96] programs.

The other partof relations must be created by hand with the help of software architects. After that, we have all the partof relations already discussed at an abstract level in the Software Concepts Model ArchiSpec. The set of partof relations embodies a system's decomposition hierarchy.

4.7.5 Discussion

For reverse engineering purposes one should be able to reconstruct the decomposition hierarchy from source code and/or from source code organisation. In fact one should take provisions, already during the initial creation of a system in order to make it easier to extract information from the system's source code. Otherwise, extra information will have to be obtained by interviewing architects and/or dedicated heuristics will have to be applied during reconstruction.

4.8 InfoPack: Depend

4.8.1 Context

The Depend InfoPack belongs to the code view. It is related to the Build Process ArchiSpec (Section 4.4). It uses the results of the Files InfoPack (Section 4.5). The result is used by the Component Dependency ArchiSpec (Section 4.9).

4.8.2 Description

The build description file contains knowledge relating to how to construct the entire system is to be created. How files depend on each other is described in the build description files. With large systems, the whole build process often consists of executing a number of build description files. Figure 4.7 shows a general view on the Build Process. For this InfoPack we make explicit the various activities in this build process in terms of files and a depends relation between files: \( \texttt{depends}_{\text{Files,Files}} \).
This InfoPack also results in the relation $\text{depends}_{E^{ts},E^{ts}}$. This relation describes dependencies at a more abstract level. It is based on file name extensions, e.g., a .o file depends on .c and .h files. It is interesting to analyse this relation because some curious tuples may be found in the case of legacy systems. These curiosities often originated in the past when different (or no) coding standards were used. In Figure 4.10 a .exe depends on .lib and .o files, a .lib file depends on .o files, and a .o file depends on .c and .h files.

### 4.8.3 Example

**Med**

With the Med system, MMS [VAX96] is used as the language for describing the dependencies between the various files. It contains the commands to be executed to build the system. We give a fragment of an MMS file:

```plaintext
COMP:main.exe : ,-  
   SUBCOMP:string.obj,-  
   SUBCOMP:finance.obj,-  
   COMP:main.obj,-  

link COMP:main.obj SUBCOMP:string.obj SUBCOMP:finance.obj -  
   -out COMP:main.exe
```

The terms COMP and SUBCOMP are VMS environment variables that refer to a certain directory belonging to components COMP and SUBCOMP, respectively. main.exe is created by executing the specified `link` command. It depends on two object files of SUBCOMP and one object file of COMP.
4.8.4 Method

We will start with the BuildFiles created by the Files InfoPack. The next step is to parse the build description files and extract the dependencies between the files. The last step is to combine the extracted information in a single relation: \( \text{depends}_{\text{Files,Files}} \).

For MMS we built a simple parser in Lex [LS86] and Yacc [Joh75]. The dependencies were written to a relation: \( \text{depends}_{\text{Files,Files}} \). A similar strategy can be used for make [Fel79] files.

The relation \( \text{depends}_{\text{Exts,Exts}} \) can be calculated as follows in RPA:

\[
\text{depends}_{\text{Exts,Exts}} = \text{depends}_{\text{Files,Files}} \uparrow \text{typed}_{\text{Files,Exts}}
\]

4.8.5 Discussion

In practice, the build description files are often (partially) generated. Information about the import relation between files is required for the generation of these files. Additional information is required to determine the proper link commands to construct the executables. To extract the \( \text{imports}_{\text{Files,Files}} \) relation (outcome of the Import InfoPack (Section 4.6)) one may tap information from this process. However, this will be the relation for a single product in the family (see also Section 4.6.5).

With legacy systems there may be a discrepancy between the files as discovered in a \( \text{depends}_{\text{Files,Files}} \) relation and the files in a system (see the Files InfoPack). The results of this InfoPack may also help to discover references to files that are a user’s proprietary, i.e., a file in a user directory which may suddenly disappear when he or she leaves the organisation. The results of the Depends InfoPack should be carefully compared (e.g., by executing RPA expressions) with the results of the Files InfoPack in order to check the correctness of the extraction results.

4.9 ArchiSpect: Component Dependency

4.9.1 Context

The Component Dependency ArchiSpect is part of the module view. It uses the results of the Import (Section 4.6), Part-Of (Section 4.7) and Depend
4.9 ArchiSpect: Component Dependency

(Section 4.8) InfoPacks.

4.9.2 Description

A system’s build time consists of the time consumed by the various activities of the Build Process (pre-compiling, compiling, linking and post-processing). The compilation of a single-source file consists of parsing all the imported header files and the source file itself. Typically, each header file is parsed as many times as it is imported in source files. Usually, after a modification, the system is rebuilt by compiling the directly or indirectly changed source files. When a header file is changed, all the source files that include this header file must be recompiled. Recompilation should be minimised by minimising the import system’s dependency. This ArchiSpect can help to estimate the average time to recompile the system after a change. This may affect procedures of building a system (e.g. the nightly built should better start at 05.00 PM).

Modifying or extending part of the system requires knowledge of its context (e.g. ‘neighbouring’ modules). A developer must understand all the implications of a change and he or she should therefore consider the consequences outside the affected source, too. With fewer import dependencies a developer need understand less a modification’s context.

During development and maintenance a developer spends a lot of time learning to comprehend the system, sometimes up to 50% of the time spent on maintenance [PZ93]. It is hard to forecast the time needed for such comprehension activities, and therefore it is hard to plan modification activities. Locality of a change is inversely proportional to the unpredictability of the required modification time. A modification in one part may result in other modifications in other parts, which is also known as the ripple-effect.

When software is modified, the system should always be tested again. One can focus on the modified source, but one must always carefully consider any software that uses a modified functionality. Component Dependency can help in determining the parts that have to be tested again.

---

Some software development environments reduce the parsing time by saving pre-compiled header files in a binary format.
4.9.3 Example

Subsystems use each other’s functionality to operate properly. This usage can be presented with box-arrow diagrams (here created by the Teddy-PS tool; see Section C.3, but there also exist commercial and non-commercial tools, e.g. Rigi [SWM97]), but we must be explicit about the semantics of the boxes and arrows. A box represents a piece of software, e.g., a subsystem; an arrow from one box (importing entity) to another box (imported entity) represents an import dependency. The table diagram (created by the TabView tool; see Section C.5), shows marks (▸) in those cells for which an import dependency exists. If there is a mark, the entity given in the leftmost column imports the entity given in the top row.

Med

The component dependency of Med is given in Figure 4.11 at subsystem level (note that arrows from a box to that same box, i.e., the identity relation, have not been depicted). The same information is also given in another form in Table 4.7 (here, the identity relations appear in the diagonal of the table).

Comm

The component dependency of the Comm system is depicted in Figure 4.12.

4.9.4 Method

We will start the reconstruction of Component Dependency with the results of the following Info Packs and relations:

- Import; imports Files, Files
- Part-Of; a chain of partof relations starting at Files level and finishing at Subs level. More generally, we need partof relations at all the levels in the decomposition hierarchy: partof Files, LN; partof LN, LN-1; ..., partof L2, L1

Knowledge of the results of the Software Concepts Model ArchiSpect is needed to be able to correctly interpret the various partof relations, namely the “chain” of decomposition levels versus the “chain” of partof relations.
Figure 4.11: Component Dependency Mod
Figure 4.12: Import Dependency Comm
The next step is to make proper abstractions of the given import information. A chain of lift operations must be executed to derive the component dependency at the requested level. One can derive this, in RPA, as follows, given the decomposition hierarchy \( Files\rightarrow Packs\rightarrow Comps\rightarrow Subs \):

\[
\begin{align*}
\text{imports}_{Packs, Packs} & = \text{imports}_{Files, Files} \uparrow \text{partof}_{Files, Packs} \\
\text{imports}_{Comps, Comps} & = \text{imports}_{Packs, Packs} \uparrow \text{partof}_{Packs, Comps} \\
\text{imports}_{Subs, Subs} & = \text{imports}_{Comps, Comps} \uparrow \text{partof}_{Comps, Subs}
\end{align*}
\]

**explanation**

As illustrated in Figure 4.13: if a file \( X \) in package \( A \) imports a file \( Y \) from package \( B \) then package \( A \) imports package \( B \) (arrow 1). This principle has been applied in the above RPA expression by means of the \( \uparrow \) (lift) operator. Information that file \( X \) is part of package \( A \) and file \( Y \) is part of package \( B \) is described in the relation \( \text{partof}_{Files, Packs} \). We have lifted the \( \text{imports} \) relation at \( Packs \) level to the \( Comps \) level by lifting again (arrow 2). By lifting a third time we reach the \( Subs \) level (arrow 3).

The last step is to present the information; a typical form of representation is a graph. A number of graph visualisers are discussed in Appendix C. We prefer to use a layout and format that will be familiar to the developers. So we have applied the notation used in the system’s documentation. For example, the layout of boxes drawn in Figure 4.11 is the same as in pictures in the Med documentation. The Teddy-PS visualisation tool (see Section C.3) can be used to create this diagram.

Another form of presentation is a table or matrix; the \textit{using} subsystems are listed in the first column and the \textit{used} subsystems in the first row. A mark appears in the cells where a \textit{using} subsystem imports a \textit{used} subsystem.
Table 4.7: Component Dependency of Med

The TabView visualisation tool (see Section C.5) can be used to create a similar table in a Web browser.

4.9.5 Discussion

The order in which the subsystems appear in Table 4.7 has been carefully chosen. The subsystems at the top of the system (see Figure 4.11), Div and Top, are listed first, and those at the bottom, Bot and Bas, appear in the last row and the last column, respectively. In the case of a system with a layered structure as discussed in Section 1.5.1, one may expect marks in the upperright corner of the table (i.e. above the diagonal). An opaque layered system may have marks only in the diagonal cells and in exactly one cell above these cells.

Assume that a dependency between subsystem A and subsystem B is curious. We will then want to investigate the reasons for its existence. For example, we might want to find out which imports at component level are responsible for it. The following RPA formulas can be used for this purpose:

\[
\text{Comps}_A = \text{partof}_{\text{Comps,Subs}} A
\]
\[
\text{Comps}_B = \text{partof}_{\text{Comps,Subs}} B
\]
\[
\text{suspects}_{A,B} = (\text{imports}_{\text{Comps,Comps}} | \text{dom} \text{Comps}_A) | \text{ran} \text{Comps}_B
\]
The \( \textit{Comps}_A \) (\( \textit{Comps}_B \) set contains all the components that belong to subsystem \( A \) (\( B \)), i.e. the left image of \( \textit{partof}_{\textit{Comps}, \textit{Subs}} \) with respect to \( A \) (\( B \)). Taking the relation \( \textit{imports}_{\textit{Comps}, \textit{Comps}} \), we must look at the tuples that start (i.e. \textit{domain}) from components of \( A \) and end (i.e. \textit{range}) with components of \( B \).

Analogously, we can descend the decomposition hierarchy to obtain more specific information:

\[
\begin{align*}
\textit{Packs}_A &= \textit{partof}_{\textit{Packs}, \textit{Comps}} \mid \textit{ran} \ \textit{Comps}_A \\
\textit{Packs}_B &= \textit{partof}_{\textit{Packs}, \textit{Comps}} \mid \textit{ran} \ \textit{Comps}_B \\
\textit{suspectsP}_{A,B} &= (\textit{imports}_{\textit{Packs}, \textit{Packs}} \mid \textit{dom} \ \textit{Packs}_A) \mid \textit{ran} \ \textit{Packs}_B
\end{align*}
\]

We have discussed this ArchiSpect by taking the \textit{imports} relation as a starting point. It is however also possible to start with the \textit{depends} relation, which will result in a slightly different interpretation of the diagrams.

\section*{Reflexion Models}

We will finish this discussion by relating \textit{Component Dependency} to the \textit{reflexion models} introduced by Murphy et al. \cite{MNS95}. A so-called source model is extracted from the source code. This model contains a use relation between source model entities (\textit{smodel\ entities}), e.g. files. Additionally, there is a mapping which describes how source model entities are to be mapped onto high level model entities (\textit{hlmodel\ entities}). We give an example of a mapping table:

\[
\begin{align*}
[ \textit{file}=\ast \textit{string}\backslash . [\textit{ch}] & \quad \text{mapTo=StringHandling} \\
[ \textit{file}=\textit{tcpip}/\textit{ip}\backslash . [\textit{ch}] & \quad \text{mapTo=IPS\ services} ]
\end{align*}
\]

All source model entities must be mapped onto high level model entities. In fact, this mapping defines a \( \textit{partof}_{\textit{SMEs}, \textit{HLEs}} \) making use of regular expressions to reduce the number of entries in the mapping table. From these pieces of information a \textit{mappedSourceModel} can be constructed which is a set of tuples of tuples:

\[
\textit{mappedSourceModel} = \{ \langle \langle h_1, h_2 \rangle, \langle s_1, s_2 \rangle \rangle \}
\]
\[ \langle s_1, s_2 \rangle \in SourceModel \land mapping(h_1, s_1) \land \\
mapping(h_2, s_2) \} \]

The domain of the \textit{mappedSourceModel} describes the use relation at a high level. This is similar to the result obtained on lifting an \textit{imports} \texttt{Files,Files} relation to a subsystem level.

### 4.10 ArchiSpect: Using and Used Interfaces

#### 4.10.1 Context

The \textit{Using and Used Interfaces} ArchiSpect belongs to the module view. The results of the \textit{Import} (Section 4.6) and \textit{Part-Of} (Section 4.7) InfoPacks are used as input.

#### 4.10.2 Description

The interface provided by a class consists of the methods and data which are not private. A class can be (re-)used in a proper way when one is aware of at least its interface (both syntax and semantics). This principle can be applied at each decomposition level, so to reuse components one must be aware of the component’s interface.

The \textit{Component Dependency} ArchiSpect shows the interconnectivity of components. It is relevant to know the constituents of a used component that are used by other components. When a component must be replaced by another component one should at least know the connection points of that component with the outside world.

Good software architectures explicitly describe the interfaces of all the components. It is also possible to reconstruct the interfaces of components. The using interface of a component consists of the component’s elements that are using (elements of) other components. The used interface of a component consists of the component’s elements which are used by (elements of) other components.

Figure 4.14 shows the using interface and used interface. The rounded boxes represent files; the square boxes represent components. In this example we will show the \textit{Files} interface of component \textit{K}. We must note that
Interfaces can be calculated at various decomposition levels, e.g., the *Functions* interface of subsystems. The *Files* interface of a component consists of the set of files that are using (being used-by) other components.

A poor design creates a single *global* include file per subsystem, which includes all the header files of the subsystem. This minimises the used interface of a subsystem, but it is not considered a good design decision, because a modification of this *global* include file will necessitate recompilation of all the source files that use this subsystem. We introduce the notion of the used+ interface to overcome this problem. The used+ interface, at *Files* level, consists of all the directly or indirectly included files. Note that the used+ interface is most relevant when we are investigating the *Files* interface. At higher abstraction levels we will often obtain all the entities of the system when we consider indirect usage. At file level the (in)direct inclusion of files always “starts” and “stops” at header files.

### 4.10.3 Example

**Med**

We reconstructed the *Files* interface of subsystems for the *Med* system. In Figure 4.15 the using and used interfaces are expressed as ratios of a subsystem’s full set of files. Boxes represent the subsystems; the upper (shaded) part of the box represents the ratio of the files that belong to the used interface, the lower (shaded) part of the box represents the ratio of
Figure 4.15: Using and Used Interfaces of Med

the files that belong to the using interface. Note that some files may form part of the using interface as well as the used interface. The using and used interfaces can also be presented in a tabular format as given in Table 4.8.

4.10.4 Method

The results of the Import and Part-Of InfoPacks are required for this ArchiSpect. Knowledge of the Software Concepts Model ArchiSpect is needed to understand the way partof relations are organised.

To be able to calculate the Files interface of components we need the fol-
Table 4.8: Using and Used Interfaces Ratios of Med

following relations: $\text{imports}_{\text{Files, Files}}$ and $\text{partof}_{\text{Files, Comps}}$. In general, to be able to calculate the $L_X$ interface of $L_Y$ we need the relations: $\text{partof}_{L_X, L_Y}$ and $\text{imports}_{L_X, L_X}$. For clarity, we will adhere to the $\text{Files}$ interface of components. We will define, in RPA, the using interface step-by-step.

\[
\begin{align*}
\text{imports}_{\text{Files, Comps}} &= \text{partof}_{\text{Files, Comps}} \circ \text{imports}_{\text{Files, Files}} \\
\text{importsExt}_{\text{Files, Comps}} &= \text{imports}_{\text{Files, Comps}} \setminus \text{partof}_{\text{Files, Comps}}
\end{align*}
\]

explanation

We construct a relation $\text{importsExt}_{\text{Files, Comps}}$, that defines an import relation containing tuples as indicated by the arrows from files (rounded boxes) to components (square boxes) in Figure 4.14. For each file we calculate which components it uses: $\text{imports}_{\text{Files, Comps}}$.

We are only interested in relations that pass the boundaries of components, so from this relation we subtract the relation that contains all the internal imports expressed by the $\text{partof}_{\text{Files, Comps}}$ relation.

The second step consists of the following RPA formulas:

\[
\text{FilesUsingExt} = \text{dom}(\text{importsExt}_{\text{Files, Comps}})
\]
\[ \text{using}_{\text{Files,Comps}} = \text{partof}_{\text{Files,Comps}} \mid \text{dom FilesUsingExt} \]

**explanation**

The using interface of component \( K \) consists of the files of component \( K \) residing in the \( \text{importsExt}_{\text{Files,Comps}} \) relation. The \( \text{FilesUsingExt} \) set contains all the files that are used by components other than their containers. The \( \text{using}_{\text{Files,Comps}} \) relation assigns the \( \text{FilesUsingExt} \) set to the components to which they belong. This is a subset of the \( \text{partof} \) relation, so we restrict this relation to its domain with respect to \( \text{FilesUsingExt} \).

The next step consists of the following RPA formulas:

\[
\begin{align*}
\text{imports}_{\text{Comps,Files}} &= \text{imports}_{\text{Files,Files}} \circ \text{partof}^{-1}_{\text{Files,Comps}} \\
\text{importsExt}_{\text{Comps,Files}} &= \text{imports}_{\text{Comps,Files}} \setminus \text{partof}^{-1}_{\text{Files,Comps}} \\
\text{FilesUsedExt} &= \text{run}(\text{importsExt}_{\text{Comps,Files}}) \\
\text{used}_{\text{Files,Comps}} &= \text{partof}_{\text{Files,Comps}} \mid \text{dom FilesUsedExt}
\end{align*}
\]

**explanation**

The \( \text{importsExt}_{\text{Comps,Files}} \) relation represents the arrows from files (rounded boxes) to components (square boxes) indicated in Figure 4.14. In this case we have to lift the domain part of the \( \text{imports}_{\text{Files,Files}} \), which is performed by composing it with the reversed \( \text{partof} \) relation. We calculate the rest in a similar manner as for the using interface.

The ratio of the using and the used interfaces of components, i.e., the number of files in the interface related to the total number of files in a component, can be calculated in RPA for each component \( C \in \text{Comps} \):

\[
\begin{align*}
\text{Using}_C &= \frac{|\text{using}_{\text{Files,Comps}}(C)|}{|\text{partof}_{\text{Files,Comps}}(C)|} \\
\text{Used}_C &= \frac{|\text{used}_{\text{Files,Comps}}(C)|}{|\text{partof}_{\text{Files,Comps}}(C)|}
\end{align*}
\]
explanation

The number of files of component $C$ that use “something” from outside that component is given in the numerator part. The total number of files of component $C$ is given in the denominator part. The used interface ratio is calculated in a similar manner.

The formulas of the used$^+$ interface are the same as those of the used interface except that we start with the transitive closure of $\textit{imports}$. So we arrive at the following RPA formulas:

\[
\begin{align*}
\textit{importsPlus}_{\text{Comps},\text{Files}} &= \textit{imports}^+_{\text{Files},\text{Files}} \circ \text{partof}^{-1}_{\text{Files},\text{Comps}} \\
\textit{importsPlusExt}_{\text{Comps},\text{Files}} &= \textit{importsPlus}_{\text{Comps},\text{Files}} \setminus \text{partof}^{-1}_{\text{Files},\text{Comps}} \\
\textit{UsedPlusExts} &= \text{ran}(\textit{importsPlusExt}_{\text{Comps},\text{Files}}) \\
\textit{usedPlus}_{\text{Files},\text{Comps}} &= \text{partof}_{\text{Files},\text{Comps}} \setminus \text{dom} \textit{UsedPlusExts}
\end{align*}
\]

explanation

The direct or indirect use of a file is expressed by the transitive closure of the $\textit{imports}$ relation: $\textit{imports}^+_{\text{Files},\text{Files}}$. The rest of the calculations are carried out in a similar manner as those for the used interface.

As already mentioned, the above formulas can be applied at various decomposition levels. Instead of two consecutive levels ($\text{Files}$ and $\text{Components}$) one can also select two non-consecutive levels ($\text{Functions}$ and $\text{Subsystems}$). For two non-consecutive levels $L_X$ and $L_Y$ we have to compose a $\text{partof}$ relation from existing ones:

\[
\text{partof}_{L_X, L_Y} = \text{partof}_{L_Y-1, L_Y} \circ \text{partof}_{L_Y-2, L_Y-1} \circ \ldots \circ \text{partof}_{L_X, L_X+1}
\]

4.10.5 Discussion

The significance of addressing ratios of using and used interfaces is also recognized in Lagné et al. [LLMD97, LLB+98]. In this work information is extracted from the source code by filtering the include statements from the C(++) source code (probably similar to the Import InfoPack as discussed in Section 4.6). From this information the authors constructed different sets (not further discussed in their paper). Each set belongs to a layer $i$; it contains certain types of files:
They calculated the used interface ratio of layer \( i \) with respect to layer \( j \) as follows:

\[
UR(j, i) = \frac{|D(j, i)|}{|IF(i)|}
\]

And the used interface ratio of layer \( i \) as follows:

\[
UR(i) = \frac{\left| \bigcup_{j \neq i} D(j, i) \right|}{|IF(i)|}
\]

They calculated the using interface and the used\(^+\) interface ratios in a similar manner.

Lagnë et al. used the number of header files as the denominator in their formulas. We used the total number of files, i.e. the header and body files. In practice the number of header files will correspond to the number of body files. So our ratios will be half the ratio of Lagnë et al. It is however possible to rewrite our using and used formulas to obtain the same ratios:

\[
Using_{Comp} = \frac{|using\ Files, Comps, Comp|}{|partof\ Files, Comps, Comp\cap\ HeaderFiles|}
\]

\[
Used_{Comp} = \frac{|used\ Files, Comps, Comp|}{|partof\ Files, Comps, Comp\cap\ HeaderFiles|}
\]

Large ratios for using and used interfaces means that hardly any information is hidden. One should therefore strive to create small interfaces. In fact, this holds for each level of abstraction. It is however more important to have small interfaces at the higher levels of abstractions than lower ones. In general, the system is more comprehensible when all the interface ratios are small.
4.11 Concluding Remarks

In this chapter InfoPacks and ArchiSpects of the module view and code view at the described level have been discussed. These InfoPacks and ArchiSpects and their relations are represented in Figure 4.2. Reconstructing ArchiSpects from existing systems is a very useful way of learning to comprehend the system. The results of these ArchiSpects can be used to enhance (or update) the software architecture documentation.

Over the years we have reverse architected a number of module views of systems [Kri97]. For many of these systems we were able to reconstruct the Component Dependency ArchiSpect in a few days with the help of an architect. The results of this ArchiSpect helped to feed discussions about the system’s software architecture. We experienced that it is best to have a first step of defining improvement activities relating to the module view.

Software changes with time, so the software architecture may change, too. Indeed, ArchiSpects have to be reconstructed every time a system is modified. Most InfoPacks extract information from the source code which are therefore most accurate. After an initial reconstruction, one can think about automating this process. The various reconstruction activities must then be integrated somewhere in the Build Process. In this way, one can every day obtain an up-to-date set of ArchiSpects, which may serve as part of the system documentation. We have applied this strategy to three development sites at Philips (Med, Switch, and Comm). Every night, the Component Dependency is reconstructed and, the next day, it is presented to developers (on request).

Results of ArchiSpects can be presented in a Web browser. The information to be presented must be stored on the Web server. The developers will then have easy access to this information by means of a browser tool with which they are already familiar. Besides just presenting ArchiSpects in a Web browser, one can also provide user interaction. Consider the Component Dependency ArchiSpect which can be reconstructed at different levels in the decomposition hierarchy. A developer may want to zoom-in and zoom-out on information by clicking on boxes and arrows in diagrams (as shown in e.g. Figure 4.12) or on entries in tables (as shown in e.g. Table 4.7).

This can be easily achieved using standard Web technology [SQ96]: hot spots to click on boxes or arrows in diagrams and cgi scripts to calculate more detailed (or more abstract) information on request. TabView (see Figure 4.16 and Section C.5) is a presentation tool that uses cgi scripts to
calculate new tables\(^7\) on request of an architect or developer (by clicking on a hyperlink).

We will finish this chapter with a brief comparison of our approach with the *Software Bookshelf* and *Dali*.

**Software Bookshelf**

The *Software Bookshelf* [FKH+97] is a Web-based approach for presenting software-related information. A kind of bookshelf captures, organizes, manages and delivers comprehensive information of a software system. It is an integrated suite of code analysers and visualisation tools. The authors distinguish three roles: builders, librarians and patrons. The builder develops the bookshelf’s framework and all kinds of tools to support a librarian. A librarian populates the bookshelf with meaningful information of the software system. A patron is the system’s end user (developer, manager, architect). Web technology is very suitable for presenting architecture information due to its multi-media nature. Therefore, our approach could be

\(^7\)One can store any table a developer may ask for on the server, but in the case of large systems this will be many tables. Besides consuming a lot of cpu time for generation, these tables will take a lot of disk space on the Web server.
combined with the *Software Bookshelf*, especially in view of this system's open architecture. For example, the *Software Bookshelf* uses *Rigi* as its presentation tool, but that could be replaced by our presentation tools. It would also be possible to extend *Rigi* [SWM97] with our RPA-based abstraction techniques to improve presentations, e.g. by applying transitive reductions to remove edges from graphs.

**Dali**

In their *Dali* system, Kazman and Carrière [KC98] distinguish *view extraction* and *view fusion* to support software architecture understanding. By combining different extraction views, one can, through fusion, arrive at more appropriate views. For example, a profiler extracts the *actual calls* of a system; a static analyser extracts the *potential calls* of a system relating to the modules containing these calls, which can be fused into a view that shows actual relations between modules. Unlike our tools, *Dali* uses a SQL database to store information and SQL operators to fuse information. The authors found the expressive power of SQL operations sufficient; but *transitive closure*, for example, cannot be expressed in a single SQL query or a fixed set of queries. Note that, it is possible to map most RPA operators on standard SQL queries; this work is discussed in Appendix B. The need for an operator like *transitive closure* is indispensable in the field of reconstructing software architectures (as shown in different parts of this thesis). Therefore, we prefer the RPA approach to reconstruct software architectures.