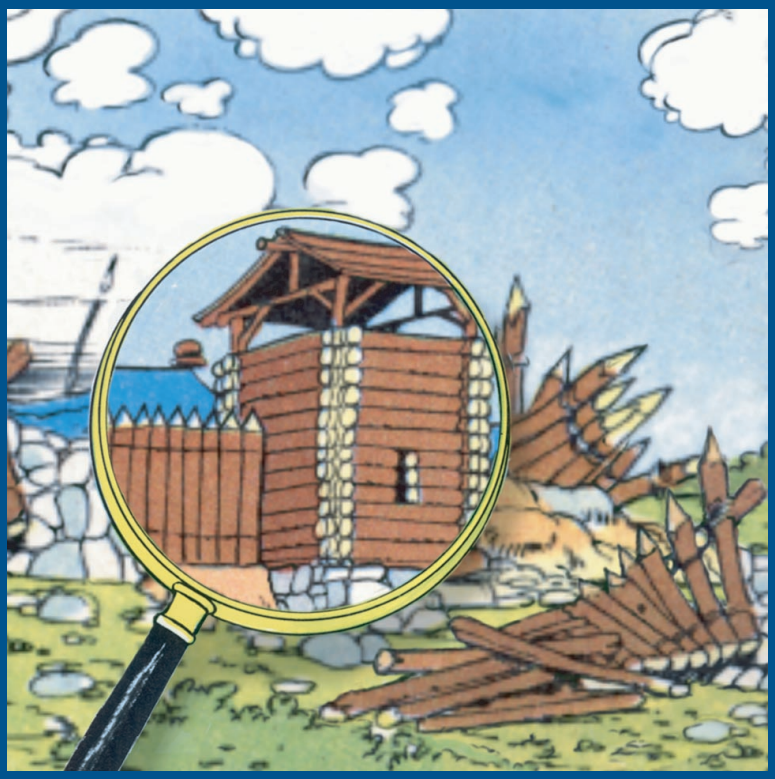
Afhankelijk van rugdikte

Software Architecture Reconstruction

René L. Krikhaar 1999

Software Architecture Reconstruction

René L. Krikhaar



Software Architecture Reconstruction

The work described in this thesis has been carried out at the Philips Research Laboratories in Eindhoven, The Netherlands, as part of the Philips Research programme.

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Krikhaar, René Leo

Software Architecture Reconstruction / René Leo Krikhaar. - Amsterdam: Universiteit van Amsterdam, Faculteit Wiskunde, Informatica, Natuur- en Sterrenkunde, RICS
Proefschrift Universiteit van Amsterdam. - Met samenvatting in het Nederlands.
ISBN 90-74445-44-6
Trefw.: Software Architecture / Reverse Engineering / Software Architecture Reconstruction / Architecture Verification.

Software Architecture Reconstruction

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
Prof. Dr. J.J.M. Franse
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Aula der Universiteit
op dinsdag 29 juni 1999, te 15.00 uur

door

René Leo Krikhaar

geboren te Nijmegen

promotor:	Prof. Dr. J.A. Bergstra
co-promotor:	Dr. C. Verhoef
faculteit:	Wiskunde, Informatica, Natuur- en Sterrenkunde
	RICS

Preface

Motivation

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

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

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

Quality is always of great importance for products. The product's quality must be continuously monitored and where possible improved. Software is becoming a major part of all these products and quality activities are consequently shifting from hardware to software. Short lead times of products and high quality are in fact conflicting requirements which must be carefully managed.

The software of many Philips' products is already undergoing changes as indicated above (increased functionality implemented in software, increased product diversity, decreased lead time and improved quality). At the time of these product's initial development, in some cases decades ago, the software architecture did not play the important role it has today. In those days, the architecture was often not handled explicitly in the engineering phase. At present, the software architecture is of major importance for product development to be able to manage the changes listed above. The spectrum of possible solutions to fill the gap between the absence of an explicit architecture and the need for such an architecture lies between:

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

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

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

Research Contributions

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

InfoPacks. The applicability of this framework is demonstrated by the definition of a number of ArchiSpects and InfoPacks.

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

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

History of the Project

In the early nineties, a main research topic of our department was to investigate, develop and adapt software development methods to build embedded systems (e.g. televisions). During the introduction of a new (formal) method [Jon88a, Jon88b] for developing software for televisions the need for information extraction arose. Small programs were written to retrieve design information from the source code. In those days, visualisation of software information was also needed. This has resulted in the proprietary tool *Teddy-Classic* (discussed in Appendix C). In fact, *Teddy-Classic* was able to display a graph consisting of nodes and edges; nodes represent modules and edges represent module imports. In a later stage, so-called *duppies* (design update proposals) were implemented using shell scripts to check the consistency of the software structure (an early form of architecture verification). But extraction and checking were still performed on an ad-hoc basis. From this work arose the need for a mathematical foundation for making abstractions upon software. This was the embryonic phase of Relation Partition Algebra; see Chapter 3.

In 1993, research was started to analyse a public telephony switching system (*Tele*) developed according to a dedicated method. The analysis resulted in a description of the Building Block method. During this research, again, a need for extraction and abstraction mechanisms for software was recognized. This time, it was needed mainly to identify the concepts behind the design method. Later, the Building Block method, which we would nowadays call an architecture method, was partially applied to another communication system. This meant that we first had to analyse the architectural concepts of this system before we could select and apply the most affecting concepts of the Building Block method.

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

Complexity of Systems

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

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

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

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

[1] It is hard to normalise the data of the different systems; we have handled the figures in a non-scientific fashion. The table is therefore meant mainly to illustrate the complexity of systems.

	Switch	*Med*	*Cons*
Product View			
# customers	10^3	10^3	10^6
# products	10^1	10^1	10^2
Evolutionary View			
system's age/lifetime (years)	15/30	15/30	3/5
release cycle (years)	0.7	0.5	N/A
release footprint (% touched files)	-	60%	N/A
Code View			
# code files	4×10^3	7×10^3	0.6×10^3
# lines of code	1.4×10^6	2.4×10^6	0.4×10^6
# programming languages	3	8	2
Module View			
# subsystems	8	11	5
# file imports	32×10^3	70×10^3	0.8×10^3
# external components	0	5	0
Execution View			
# operating systems	1	4	1
# (main) computing devices	1	5	3
# software processes	> 1000	50	50

Table 1: Characteristics of Large Systems

One can argue about the way how the size of the source code should be measured, but for our purpose the number of lines suffices. The number of programming languages indicates problems that could arise in merging software parts and e.g. the required educational background of developers.

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

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

Related Work and Tools

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

Rigi

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

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

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

Reflexion Models

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

mal model is used to calculate convergences, divergences, and absences of relations in the high-level model or the source model.

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

Relational Algebra

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

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

Software Bookshelf

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

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

Dali

Dali [KC98] is an architecture analysis framework containing e.g. *Rigi* as a presentation tool. It is based on *view extraction*, extraction of static and

dynamic elements from the system, and *view fusion*. *View fusion* consists of combining views in order to achieve new views that are richer and/or more abstract. *Dali* contains an SQL database containing the various views. We consider SQL less accurate for expressing software manipulations. We will therefore introduce Relation Partition Algebra, which has more accuracy (e.g. by means of the operations transitive closure and transitive reduction).

Outline of Thesis

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

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

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

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

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

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

Chapter 7 gives recommendations for the application of software architec-

ture reconstruction.

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

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

Acknowledgements

During my work at the Philips Research Laboratories I had the opportunity to analyse software architectures. This work eventually resulted in this thesis, which could not have been written without the support of my group leader, Jaap van der Heijden, and my cluster leader, Henk Obbink. I also want to thank the director of the Information and Software Technology, Eric van Utteren, who gave me this opportunity.

Loe Feijs encouraged me to write a thesis on my research performed since 1994. Without his work on Relation Partition Algebra my work would never have reached the current level of maturity. He was always willing to discuss subjects and he has always supported my work in many cases.

I would like to thank Jan Bergstra, my promotor at the University of Amsterdam, for his support in discussing many topics and giving advice in writing this thesis. Comments on earlier versions of my thesis were very useful and always utmost to the point. It was a pleasure to work with Jan. The ready knowledge of Chris Verhoef about reverse engineering research in the world was of great help in relating it to software architecture reconstruction research. Discussions with Chris were fruitful and they improved the quality of this thesis.

Throughout the years, I had the opportunity to analyse many systems at Philips. Many Philips' sites (in Europe) were willing to discuss the ins and outs of their systems with me. I appreciate their dedication and the time they have invested despite their often very busy daily work. Although many people were involved, I would like to thank especially Lothar Baumbauer (Germany/Nuremberg), Ad Zephat and Jan Willem Dijkstra (the Netherlands/Best), Paul Schenk (Austria/Vienna), Reinder Bril and Thijs Winter (the Netherlands/Hilversum), and Paul Krabbendam (the Netherlands/Eindhoven). Research of software architectures can indeed only be performed in cooperation with people who actually build large systems.

Therefore, it is of great importance to have access to such systems. Without the support of these people, research into this topic is not practicable.

In various projects in which I participated over the past years I worked together with a number of colleagues for some amount of time. First of all, I would like to thank Jan Gerben Wijnstra with whom I spent about eight months in Nuremberg in Germany. As colleagues we worked together in the Building Block project which aimed to establish an abstraction of the software architecting method used to develop telecommunication systems. During our stay in Nuremberg we were sentenced to spend a lot of spare time together and I still cherish good memories of that time. The initial idea to develop a more structured method for reverse engineering large systems originated in that time.

In 1996 and 1997 Jeroen Medema participated in two reverse architecting projects (*Med* and *Switch*). Jeroen carried out a good deal of practical work and was of great help in pushing our research in the right direction. Jeroen is also responsible for the Java implementation of Relation Partition Algebra described in the appendix. This implementation proved most valuable in the introduction of our research results at various development sites at Philips.

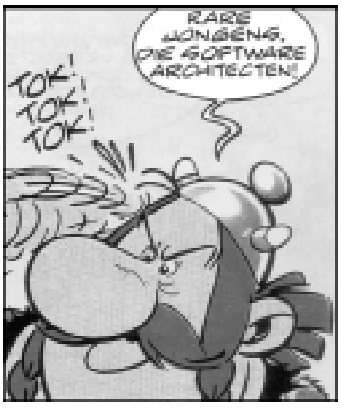
I thank Henk Obbink who often participated in our discussions of software architecture. I also want to thank my colleague Jürgen Müller with whom I discussed a variety of related and unrelated topics. Rob van Ommering supported the work by discussing with me his experiences in creating software for televisions. He was also one of the persons behind the mathematical foundation of Relation Partition Algebra, and participated in the AWK implementation of this theory.

Initial versions of (parts of) the thesis have been reviewed by Reinder Bril, Loe Feijs, Robert Jagt, Jeroen Medema, André Postma and Marc Stroucken. I thank them all for their critical and constructive comments. I also want to thank Maarten Pennings, who helped me with the pecularities of LaTeX [GMS93, Lam85], Noor Krikhaar for correcting the Dutch grammar, Philips Translation Services for correcting the English grammar of an earlier version of this thesis, Frans Willemsen for his text writing advices and Aad Knikman, who helped to create the picture on the cover.

I want to thank the reading committee for reading and for approving my thesis: Peter van Emde Boas (University of Amsterdam), Loe Feijs (Eindhoven University of Technology), Rick Kazman (Carnegie Mellon University and Waterloo University, USA) and Paul Klint (University of Amsterdam).

Finally, I would like to thank all the people I have not mentioned so far, but who have also supported my work and life in a spiritual or technical way.

René L. Krikhaar



Contents

List of Figures

Chapter 1

Software Architecture

In advance of discussing software architecture reconstruction, which is the main topic of this thesis, we briefly present, in this chapter, some issues related to software architecture (amongst others definitions of architecture and architectural view models).

1.1 Introduction

In this chapter we give an overview of definitions of software architecture found in the literature. But we also discuss the importance of having a good software architecture in a software intensive system.

A software architecture must satisfy requirements from a business point of view. These business goals lead to certain objectives for software architecture, to be discussed in Section 1.4. A number of good architectural patterns which may be useful in various software systems will be presented in Section 1.5. Business goals, architectural objectives and architectural patterns are related to each other. In Section 1.6 we derive from specific business goals the related architectural objectives which, in turn, lead to certain architectural patterns. A general view on this model is illustrated in Figure 1.1.

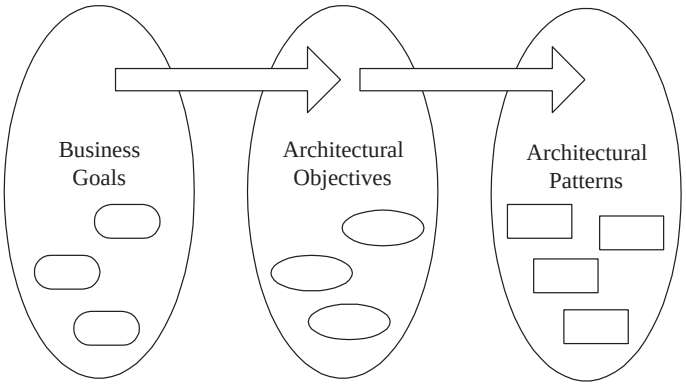


Figure 1.1: Business Goals, Architectural Objectives and Patterns

1.2 Definitions of Software Architecture

In recent years, many definitions of software architecture have appeared in the literature. The need for modular structuring and explicit handling of product families was first discussed in the late sixties [Dij68, Par76, PCW85]. Since then, software in systems has grown tremendously in size and complexity. Although these "old" structuring principles still hold, they have to be transformed into principles for the products of today's sizes. The term software architecture was introduced in the nineties to address, amongst other things, the up-scaling of these structuring principles.

In 1992, Perry and Wolf [PW92] gave a definition of software architecture: a set of architectural elements that have a particular form. The elements may be processing elements, data elements or connecting elements.

According to Shaw and Garlan [SG96], the architecture of a software system defines that system in terms of computational components and interactions between those components. Examples of components include clients, servers, filters and layers of a hierarchical system. Interactions between components may consist of procedure calls, shared variables, asynchronous events or piped streams.

Jacobson, Griss and Johnsons [JGJ97] stated that a software architecture describes the static organization of software in subsystems interconnected through interfaces and defines at a significant level how nodes executing those software subsystems interact with each other.

Bass et al. [BCK98] gave an "often-heard" definition: architecture is com-

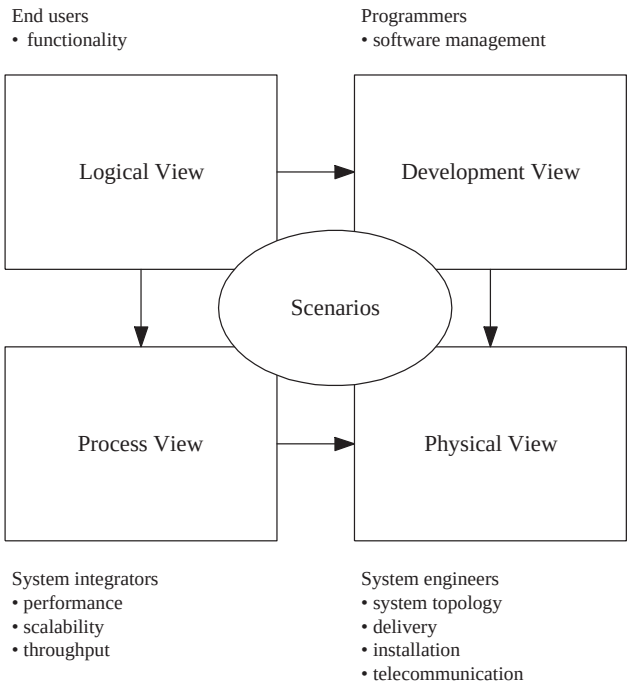


Figure 1.2: The 4+1 View Model

ponents, connectors, and constraints. Connectors are a mechanism for transfering control and data around the system. Constraints are definitions of the behaviour of components.

Many different structures are involved in an architecture of a software system. In order to organize them, models have been defined that give a certain view on software architecture. We found that the models developed by Kruchten [Kru95] and Soni et al. [SNH95] are useful in industry. In the next sections these so-called view models will be discussed, including a model that combines both view models.

1.2.1 The 4 + 1 View Model

Kruchten distinguishes five different views in his *4 + 1 View Model* of architecture [Kru95]. Each view addresses a specific set of concerns which are of interest for different stakeholders. Figure 1.2 (taken from [Kru95]) shows the views, the stakeholders and their concerns.

The *logical view* supports the functional requirements: the services a system should provide to its end users. The designers decompose the system into a set of key abstractions of the domain, which results in a domain model. Kruchten suggests to use an object-oriented style to define the logical view.

The *process view* addresses non-functional requirements, such as performance and availability of resources. It takes into account concurrency and distribution, system integrity and fault tolerance. In this view, the control of execution is described at several levels of abstraction.

The *development view* focuses on the organization of the actual software modules in a software development environment (SDE). It concerns the internal requirements related to ease of development and software management. The development view is represented by module and subsystem diagrams that show the system's export and import relations.

The *physical view* also takes into account the system's non-functional requirements. It maps the various elements identified in the *logical*, *process* and *development* view onto the various hardware elements. This mapping should be highly flexible and should have a minimal impact on the source code itself.

The *scenarios* help to demonstrate that the elements of the four views work together seamlessly. The scenarios are in some respect an abstraction of the most important requirements. A scenario acts as a driver to help designers discover architectural elements, and also helps to illustrate and validate the architecture design.

An example of a scenario is the description of the activation of *follow me* in a telephony switching system (activation of forward direction of incoming calls to another specified extension). Given this scenario, one can discuss how the involved processes communicate with each other via message communication, which components of the system are running, and which hardware devices are involved. The scenario view is in fact redundant with the other views, hence the "+ 1".

The various views are not completely independent of each other. The characterization of the logical elements helps to define the proper process elements. For example, each logical element (object) is either active or passive (autonomy of objects); elements are transient or permanent (persistency of objects). The autonomy and persistency of objects have to do with the process view. The logical view and development view are very close, but address different concerns. The logical elements do not necessarily map

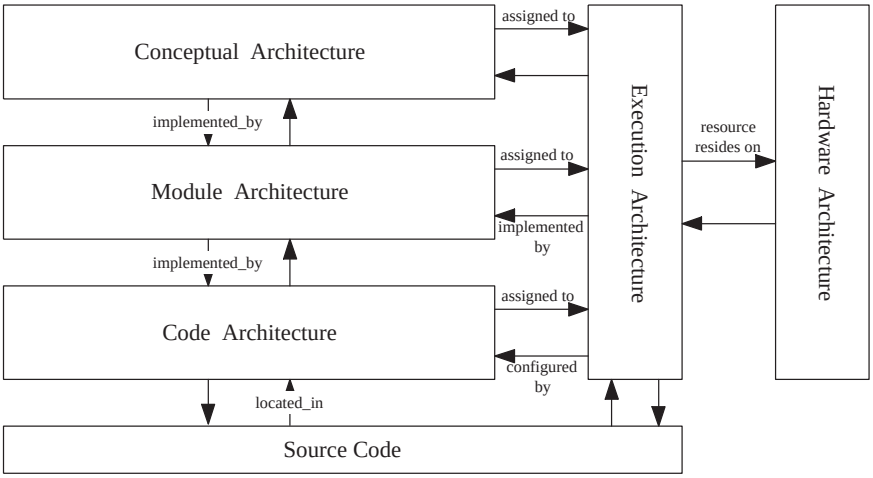


Figure 1.3: Relationships between the Software Architectures

one-to-one on development elements. Similar arguments hold for the relation between process view and physical view.

1.2.2 The SNH Model

Soni et al. [SNH95] have investigated a number of large systems (telecommunication, control systems, image and signal processing systems) to determine the pragmatic and concrete issues related to the role of software architecture. The structures they found in the investigated systems can be divided in several broad categories. Soni et al. distinguished five different views on architecture:

- *conceptual architecture*: describing the system in terms of its major design elements and relationships between them. Typical elements are components and connectors.
- *module (interconnection) architecture*: functional decomposition and layers, which are orthogonal structures. Typical terms are subsystems, modules, layers, imports and exports.
- *execution architecture*: describing the system's dynamic structure. Typical elements are tasks, threads, RPC and events.
- *code architecture*: describing how the source code, binaries and libraries are organised in the development environment. Code resides in files, directories and libraries.

- *hardware architecture*: describing the hardware components and their relations as far as they are relevant for making software design decisions. Processors, memory, networks and disks are typical hardware elements.

The architectural views have relations with each other as depicted in Figure 1.3 (taken from [SNH95]). A conceptual element is *implemented* by one or more elements of the module architecture. Module elements are *assigned to* run-time elements in the execution architecture. In addition, each execution element is *implemented by* some module elements. Module elements are *implemented by* elements of the code architecture. There is also a relationship between run-time elements and executables, resource files (e.g. help texts in different natural languages) and configuration files in the code architecture.

1.2.3 The AV Model

Kruchten described a number of design principles for constructing elements of the various views. The SNH model was defined after an analysis of existing systems, which comprised looking at an architecture from a different angle. Nevertheless, the *4 + 1 View model* and the SNH model are pretty similar. A logical decision (also suggested by [BMR+96]) is to combine the good parts of the two into a new view model (see Figure 1.4). We have taken the *4 + 1 View model* as a basis and integrated it with good parts of the SNH model. The new model has been baptized the *Architectural View model*, abbreviated as the *AV model*.

The *logical view* and *conceptual architecture* are more or less similar. In both cases, the end user is the main stakeholder. The *execution architecture* and *process view* differ only in details. Soni et al. addressed the *hardware architecture* concisely, but Kruchten stressed the *physical view* more explicitly.

The *module architecture* and *code architecture* maps on Kruchten's *development view* of Kruchten. In our new model, we divided Kruchten's *development view* into two parts: *module view* and *code view*. The stakeholders of the *module view* are the programmers. The main stakeholders of the *code view* are people who are responsible for tool support. In the new model, the source code is considered part of the *code view*.

Scenarios in the AV model support forward engineering as well as reverse engineering of software architectures: scenarios play a role in defining ar-

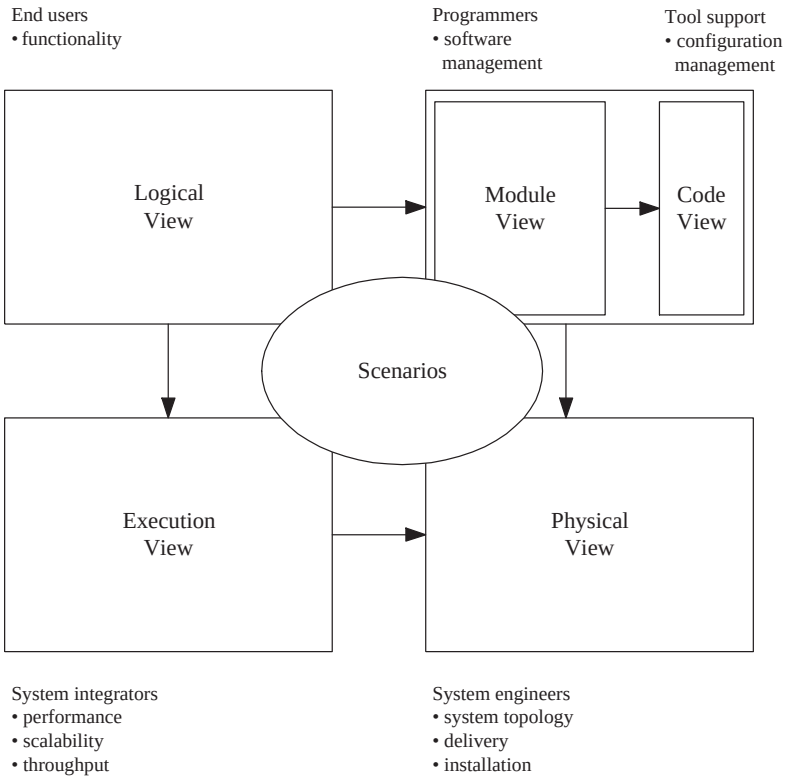


Figure 1.4: Architectural View Model

chitectural elements [Kru95], and they support the analysis of software architecture [KABC96].

The precise contents of all these views have not been described explicitly. In practice, one has to experience which elements are most important. In this thesis we focus on the *module view*, but the *code view* is also required in a supporting role. It is our intention to make the contents of the *module view* and *code view* more explicit and tangible.

1.3 Business Goals

From a business perspective the following goals can be defined for products, having impact on the software architecture within such a product [KW95, JGJ97]:

- short time-to-market;
- low cost of product;
- high productivity of organisation;
- adequate predictability of process;
- high reliability of product;
- high quality of product.

Which goals must be emphasized depends very much on the type of product. The quality of a product is very important especially for medical systems, e.g. a patient must not be exposed to too much X-ray radation. Also important is the quality of consumer products. It is for example impossible to provide every one of the millions of television users an update of the software in their television every six months[1]. In the currently booming market of digital videocommunication systems it is more accepted to deliver several software updates after the first release. In this business, time-to-market has high priority as the aim is to remain ahead of one's competitors. A software architect must be aware of such trade-offs in making proper architectural decisions.

1.4 Architectural Objectives

There are many architectural objectives that justify certain architectural decisions. Bass et al. [BCK98] distinguished different, called quality at-

[1] Although downloading of new software to a television set is foreseen in the near future.

tributes. These quality attributes are discernable at run-time (performance, security, availability, functionality and usability) or they are not discernable at run-time (modifiability, portability, reusability, integrability and testability).

In this thesis we want to discuss architectural objectives in more abstract terms. We distinguish the following architectural objectives, which are not necessarily orthogonal:

- comprehension;
- reuse;
- evolution;
- product family.

1.4.1 Comprehension

Software changes many times during its lifetime. A developer must understand the software well to be able to modify, extend or fix a bug in the system. Approximately half of the time spent on maintenance activities concerns comprehension [PZ93]. Improvement of comprehension therefore increases a developer's productivity.

In many cases software changes are made by developers who did not originally create the part of the software concerned. This is due to the typical lifespans of our systems, which may be decades. An original developer may in the mean time have moved on to another position or may even have left the organisation. Moreover, in view of a system's size and complexity, several developers must often have access to the same part of the software. The nature of today's systems makes it impossible to divide a system from the start into disjunct parts of the software that can be assigned to a single person. Comprehension of software written by other people is therefore necessary.

1.4.2 Reuse

Reuse consists of the further use or repeated use of a software artifact. Typically, reuse means that software artifacts are designed for use outside their original contexts to create new systems [JGJ97]. Proper application of reuse requires a number of precautions. Design for reuse must be explicitly addressed in an organisation to be able to reuse software. Component reuse is currently a hot topic in research and practice. In general, the number

of reusable components greatly influence productivity and quality. Reuse of software is often hard to achieve (particularly due to the *not-invented-here syndrome*); it requires a lot of investment and it must be managed explicitly to be successful. The benefits of a reuse-oriented organisation start at best, two years after introduction [JGJ97]. In a business context return-on-investment times of two years are long, especially compared with the length of time between two releases.

Reusable components can only be developed with a specified architecture in mind. For a functional equivalent component one may request different implementations depending on the architecture and/or satisfying different non-functional requirements. In the world of IC design it has long been accepted that there are different implementations for a component. In the world of software this is less accepted. For example, a component in a pipe-line architecture must behave differently from a component in an event-driven system. In a pipe-line architecture a component continuously reacts on new input data while in an event-driven system a component is triggered before it processes data. It is impossible to combine any arbitrary set of components into a new system. Garlan stressed this point as the *architectural mismatch* [GAO95].

1.4.3 Evolution

From a business perspective, software has come to be the most profitable part of software-intensive systems. Product features of existing systems are often related purely to software extensions. In the past, product requirements were often assumed to be stable. Today they are more dynamic and evolutionary. Requirements rapidly change and product developers must allow for this fact.

The evolution of hardware also has an impact on software. Take for example software that controls image-processing units in a medical system. One must be able to smoothly integrate a new hardware image-processing unit into a new system release or one may even replace such a unit by software. So the thought of possibly having new image-processing units in the future causes this to be explicitly covered in the software architecture. Good intuition of possible market trends helps to define software architectures that are future proof.

1.4.4 Product Family

Product family architectures are architectures especially designed to manage (and enhance) many product variations needed for different markets. For example, in different parts of the world there are different television broadcast standards, which affects the software embedded in a television. A television's user interface is also language-dependend. Furthermore, products may also vary in the number of features they include. A television may be packed with or without an Electronic Programming Guide (EPG). One must be able to switch the EPG feature *on* or *off* in a late stage of the production process. Software architecture must be capable of facilitating all such variations, i.e. it must be flexible.

1.5 Architectural Patterns

Alexander et al. [AIS77] defined a pattern for buildings and towns as follows:

> "A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

These patterns are described in a consistent and uniform style.

The notion of patterns can also be applied in the construction of software. Buschmann et al. [BMR+96] and Gamma et al [GHJV95] used schemes to describe design patterns. Buschmann et al. categorized the patterns into the following groups:

- Architectural Pattern
- Design Pattern
- Idiom (Code Pattern)

Conceptual integrity means that the same concept is always explicitly applied for similar problems. Conceptual integrity supports developers to better understand a system and it leads to programmer independence [Bro82]. In the case of larger systems conceptual integrity is even more important. The size of the development group is larger, which means that developers spend more time communicating with each other. The application of general concepts simplifies internal communication. An architect's task is to document these concepts, but he or she is also responsible for communicating these concepts to the development group.

A concept may also be a typical solution to a certain problem. Concepts must be defined for typical problems in each stage in the development process. Design patterns are examples of typical solutions to design problems.

To illustrate the notion of patterns, we will informally discuss three architectural patterns which are related to the module view: *layering* (Section 1.5.1), *generic and specific components* (Section 1.5.2) and *aspects* (Section 1.5.3). While analysing the *Tele* system we experienced the benefit of applying these patterns. In Chapter 6 we will return to these patterns to discuss architecture verification.

1.5.1 Layering

A layer is a group of software elements. Layers are strictly ordered. Given the ordering, higher layers may use only lower layers. We distinguish two types of layering:

- *opaque layering*: a layer is restricted to use only the layer directly below it. The idea behind opaque layering is that each layer makes an abstraction of all the layers below it and adds some extra functionality. An example of this principle is the *7 layer OSI stack*[2] [Tan76], illustrated in Figure 1.5.
- *transparent layering*; each layer is allowed to access services of all the layers below it see Figure 1.6. A layer abstracts functionalities in lower layers where appropriate, but it does not encapsulate functionality that has already reached a proper level of abstraction in a lower layer.

An advantage of opaque layering is that the user of a layer needs to know only the layer below it. It does not have to have any knowledge of the lowest layers. A disadvantage is that each layer must also provide functionality from the lower layer when required by a higher layer. This often leads to renaming of functions without adding any functionality. Another disadvantage is that the lower layer must have knowledge of higher layers to be able to provide proper functionality (to avoid the risk of all the non-exported functionality of the lower layers being provided again).

A transparent layer provides functionality to the outside world, without paying too much attention to the layers that use the functionality. A disadvantage is that when the interface of a layer changes, it may affect all

[2] In a new edition of his book Tanenbaum defined a hybrid reference model with only five layers.

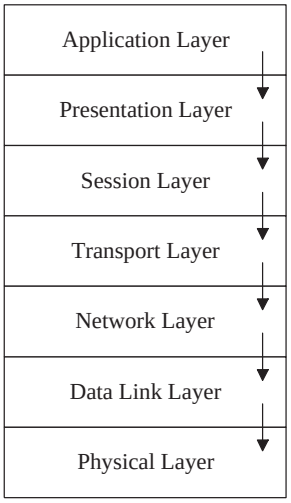


Figure 1.5: Opaque Layering

the higher layers.

Layering generally makes it possible to test a system incrementally. Layers can be tested one by one, starting at the bottom, i.e. when a layer of level n passes the test, one can test layer $n+1$, assuming that layers $1 \ldots n$ are functionally correct. Layers also facilitate the control of the development process and product releases.

Layers can be defined at different levels of abstraction. The following example of layering at the highest level of abstraction, i.e. subsystems, is taken from telecommunication industry [KW94]. It is very common to distinguish in a communication system the following layers, which we call subsystems (from top to bottom):

- *Service Management*; dealing with actual services of the system. In a switching system e.g. it deals with redirecting a telephone call when the *follow me* to another number feature is active.
- *Logical Resource Management*; providing logical resources. These resources are based on resources provided by *Equipment Maintenance*, but they are made hardware-independent. At this level an operator configures a communication system.
- *Equipment Maintenance*; dealing with the maintenance of peripheral hardware. It provides virtually error-free peripheral hardware to the higher subsystems (in a telecommunication system the functions of

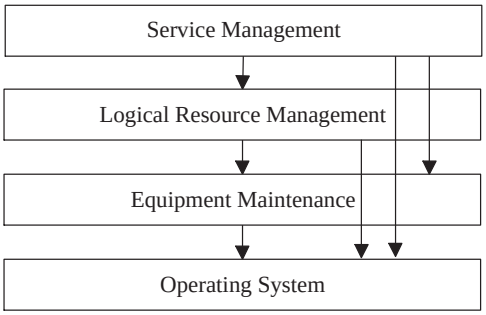


Figure 1.6: *Tele* Subsystems

failing hardware must be taken over by other hardware components). Hardware specifics are hidden. This subsystem provides an abstract representation of physical resources and their usability.

- *Operating System*; containing functionality provided by a normal operating system. It also provides some general functionality to higher subsystems, including e.g. software downloading, recovery and man-machine interface procedures.

1.5.2 Generic and Specific Components

Software components are currently a hot topic in software architecture research and practice. Szyperski used the following definition of software component [Szy97]:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

A component-based system consists of a number of components. One can divide these components into two kinds: *generic* and *specific*. Generic functionality, which resides in *generic components*, exists in almost all the products in the family, and specific functionality, residing in *specific components*, does not exist in all products.

Generic components represent the common part of all the products of a family. A crucial task of an architect is to distinguish generic and specific functionalities. It is not just a matter of factoring out the common functionality, because (yet unknown) future enhancements must also be taken into account.

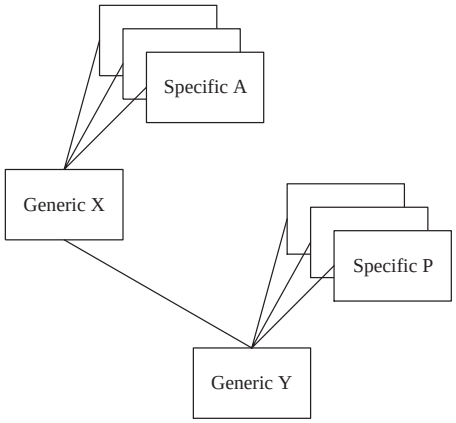


Figure 1.7: Generics and Specifics

Generic components may already be bound at compile and link time without the flexibility of configuration being adversely affected. The set of generic components form the skeleton of all products. Specific components can rely on the availability of this skeleton, but they are not allowed to rely on the availability of specific components.

This also means that only generic components can be responsible for facilitating communication between specific components (see Figure 1.7). During the system's initialisation time, specific components announce themselves to the generics. Via a call-back mechanism the generic component is able to access the specific component at run time. A specific component can call a generic component's functionality, on its turn, this generic component can call (via a call-back function) another specific component's functionality.

Different types of generic components can be distinguished. A refinement of generic component types has been discussed by Wijnstra [Wij96].

Example

A public telephone switching system communicates with several other switches using different protocols and different types of lines. Each customer asks for his or her own set of hardware units and his or her own set of protocols. The system must be configured according to the user's needs. In a late stage of the development trajectory one must still be able to con-

figure a system. It must even be possible to extend such systems (when they are running in the field) with new hardware and/or protocols. Explicit handling of generic and specific functionalities (combined with late binding) satisfies this list of requirements [KW94].

1.5.3 Aspects

In addition to object-oriented system modelling [Boo91], one can also simultaneously address a functional view on the system. In the case of large systems it is even necessary to apply another structuring mechanism for comprehension reasons. We call the means used for this structuring approach *aspects*. Before developing the separate components, one must define aspects which are in principle applicable to each component. Such a set of aspects is fixed for the whole system.

As an example we give the aspects of a typical telecommunication system:

- *normal operation*;
- *man-machine interface*;
- *recovery*;
- *configuration management*;
- *fault handling*;
- *performance observation*;
- *test*.

The notion of aspects is relevant in the various development phases. During system testing the aspects can be used to structure the process and decide on the (functional) completeness of the test. Aspects should explicitly appear in all the software artifacts (design documents, source code). A simple, but effective, implementation of aspects at source code level involves the use of prefixes (according to the aspect name) for functions, variables and files. Aspects must also be handled explicitly in design documents. For example, a reader who is interested in a certain aspect should be guided through the document in a natural fashion. This can e.g. be achieved by prescribing obligatory (sub)sections.

The *System Infrastructure Generics* (SIGs) are special generic components, which usually reside in the lowest subsystem. They deliver some basic functionality of the system. One must define (one or more) system infrastructure generics to implement the basic functionality of an aspect. For example, the man-machine interface uses basic functionality (windows, menus, etc.) which reside in SIGs. Another example is exception handling,

the basic infrastructure for achieving exception handling (e.g. popping as many return addresses from the call stack as required), is implemented in an exception handling SIG.

1.6 Relating Goals–Objectives–Patterns

In the previous sections we have discussed business goals, architectural objectives, and architectural patterns. Although the business goals are very general and hold for (almost) any business, it is obvious that some priority ordering is necessary per system (or market). Given the ordering of business goals, we can derive an ordering of architectural objectives, as illustrated in Figure 1.8. For example, the *cost of product* is related to the amount of *reuse* that can be established. Furthermore, architectural objectives can be mapped on architectural patterns. For example, when a *product family* is concerned it is good to explicitly distinguish *generic and specific components.*

Making an explicit Goals-Objectives-Patterns (GOP) diagram for your system helps to make proper trade-offs during the creation of software architectures. The GOP diagram of Figure 1.8 (simplified version of GOP diagram in [KW95]) should therefore be seen as just an example; extra goals, objectives and patterns and lines could be required for your system. The absence of a line does not necessarily mean that there is not a relationship, but it can be seen as a relative unimportant relation.

1.7 Final Remarks

Most of the discussed issues stem from the Building Block Method used in Nuremberg for the development of telephony switching systems (*Tele*). The Building Block Method and its application to large systems have been discussed in a number of reports [KW94, KL94, Kri94, Kri95, LM95, Wij96].

We have addressed only a few architectural patterns of the module view. Other good architectural patterns for this view exist, but the other views on architecture should also be covered with architectural patterns. In this chapter it has been our intention to give a non-exhaustive overview of the variety of issues relating to software architecture.

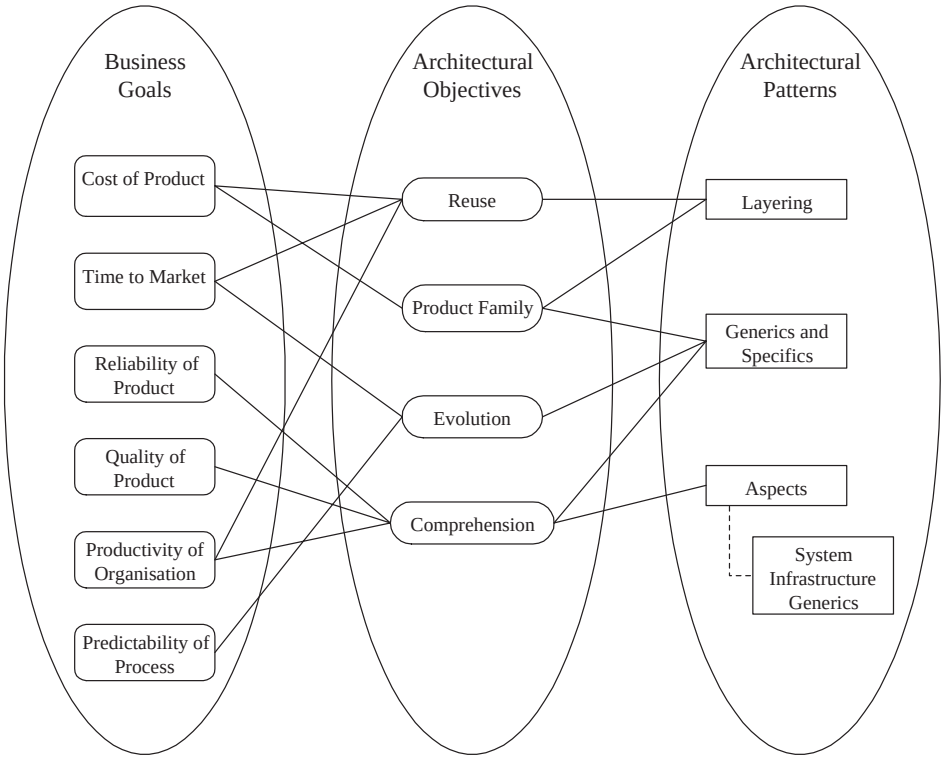


Figure 1.8: Goals, Objectives and Patterns

Chapter 2

Overview of the SAR Method

In the previous chapter we gave an overview of software architecture. In this chapter we present a framework required for a method to reconstruct a software architecture of an existing system.

2.1 Introduction

Here, we introduce a method to reconstruct an existing system's software architecture: the Software Architecture Reconstruction (SAR) method. We discuss a general framework for the SAR method, which is also used to structure this thesis.

In general, all methods consist of four different parts [Kro93]:

- an underlying model;
- a language;
- defined steps and ordering of these steps;
- guidance for applying the method.

In our software architecture reconstruction method, the underlying model consists mainly of Relation Partition Algebra (to be elaborated in Chap-

ter 3). Relation Partition Algebra consists of sets, binary relations, part-of relations and operations on them. Besides a model, RPA is also a language for expressing architectural information: we need graphical and textual notations (graph diagrams, relation tables, lists) to present a reconstructed software architecture.

The reconstruction of software architecture consists of performing the following kinds of steps: extraction, abstraction and presentation (see Section 2.3). Extraction steps will be discussed as parts of InfoPacks (the notion of an InfoPack will be discussed in Section 2.5.2); abstraction and presentation steps are contained in ArchiSpects (the notion of an ArchiSpect will be discussed in Section 2.5.2). A guidance describes the gaps that are not completely covered by the steps or when the steps do not perfectly fit in the situation at hand.

In this chapter we briefly describe the engineering of software architectures (called forward software architecting). Next, we will discuss reverse software architecting, which is the counterpart of forward software architecting. As we will see, improvements in existing software architectures demands both engineering disciplines. We will finish this chapter with a framework into which the software architecture reconstruction method can be fitted.

2.2 Forward Software Architecting

Forward software architecting, or simply software architecting, is the discipline of engineering a software architecture from scratch, or, if an architecture already exists, it consists in engineering the extensions of the architecture. An example of a method dedicated to architecture is the Building Block method [KW94, LM95].

In chapter 1 we have discussed a number of architectural patterns that are related to the module view of software architecture. One can also define architecting as the process of selecting and applying proper patterns for each of the architectural views. It is an engineering discipline that requires a lot of experience, human sense, knowledge of a range of good architectural patterns and the ability to define new appropriate architectural patterns.

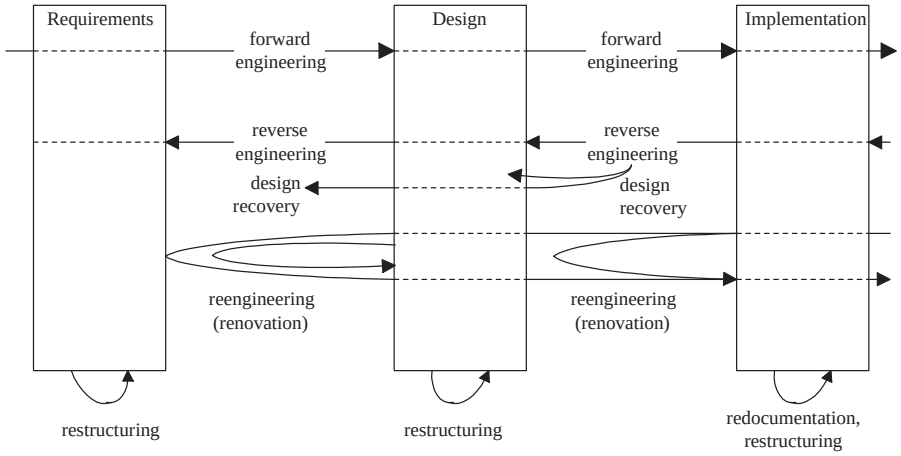


Figure 2.1: Forward and Reverse Engineering

2.3 Reverse Software Architecting

Reverse software architecting is the flavour of reverse engineering that concerns all activities for making existing (software) architectures explicit [Kri97]. Reverse software architecting aims for: recovery of lost architectural information, updating of architecture documentation, supporting of maintenance (comprehension) activities, provision of different (other) views on architecture, preparing for another platform and facilitating impact analysis. *Reverse engineering* was defined as follows by Chikofsky and Cross [CC90]:

> "The process of analysing a subject system to identify the system's components and their relationships and create representations of the system in another form or at a higher level of abstraction."

Figure 2.1 (taken from [CC90]) presents a lot of terminology within a simplified software life-cycle. *Requirements* involves the specification of the problem, *design* is the specification of a solution and *implementation* concerns the creation of a solution which consists of coding, testing and system delivery. *Redocumentation* is the simplest and oldest form of reverse engineering. It concerns the creation or revision of a system's documentation. However, many tools dedicated to redocumentation are only able to generate diagrams, print code in an attractive way, or generate cross-reference

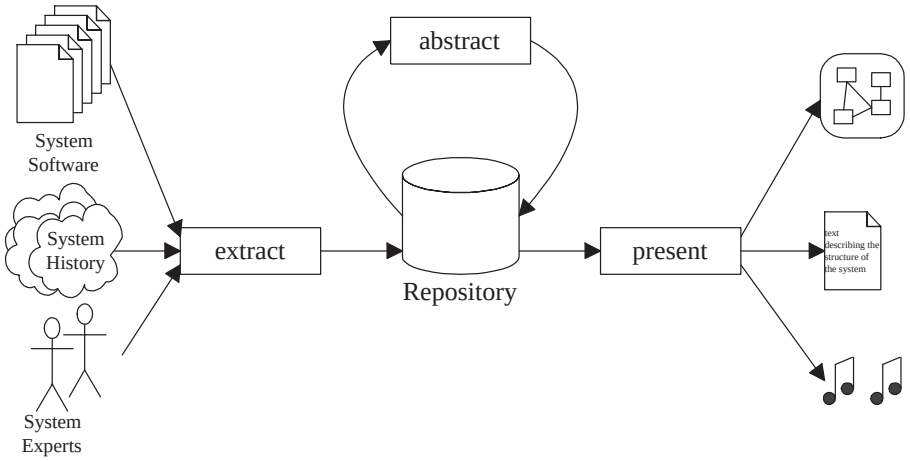


Figure 2.2: Extract, Abstract, and Present

listings. *Restructuring* is the transformation from one representation form into another, preserving the external behaviour. The first experiments in this area concerned the removal of 'goto' statements and their replacement by control structures like 'while' loops, 'if-then-else' clauses and 'for' loops. *Design recovery* means that one identifies meaningful higher levels of abstraction of software. For this activity one requires domain knowledge and designer's knowledge to add the information required to be able to create these abstractions. *Reengineering* is related to the modification of an original system to increase design quality.

Extract – Abstract – Present

The process of reverse engineering (depicted in Figure 2.2) in general consists of three activities:

- *extract*: extracting relevant information from system software, system experts and system history;
- *abstract*: abstracting extracted information to a higher (design) level;
- *present*; presenting abstracted information in a developer-friendly way, taking into account his or her current topic of interest.

Tools can be used to *extract* information from the *system software*, which includes source code, design documentation, etc. The value of the tool's output may depend on the availability of coding standards, and of course

on whether these coding standards are satisfied by the developers. For example, many implementation languages do not explicitly support a module concept (similar to modules in *Modula-2*), but one can force a pseudo-module concept by prescribing certain coding rules. The extraction results are stored in a database, which is called a *repository*. *System experts* can be interviewed to obtain architectural information with the aid of different techniques, e.g. think-aloud sessions, structured interviews and brain-dump sessions. *History information* can be extracted from the software archive or documentation system, providing information about the system's evolution.

Because most of the extracted information is often at programming level, one must *abstract* from this information and bring it to an architecture level. In addition, some filtering of information may be required for certain views on the system. Developers need different views on (parts of) the system in their daily work. The requested view is to a great extent driven by the problem at hand, so good navigation means are needed to retrieve information.

The abstracted information can be *presented* in different ways. Developers may prefer diagrams and pictures, but more fancy media may be applicable such as sound and vision, instead of textual descriptions, e.g. lists of items. Hyperlinks should be added to textual descriptions to achieve good navigation means. All these types of presentation types have already been integrated in various Web browsers, which makes this medium a good candidate for these purposes.

Extracted information may originate from different tools, e.g. if multiple implementation languages are used. Combining this information from different sources may result in incomplete or even conflicting data. One must allow for such situations, noting that incomplete data may appear complete at higher levels of abstraction.

In the appendices we give an overview of the tools that proved to be useful during our research. Extraction tools are discussed in Appendix A. Abstraction tools based on Relation Partition Algebra are presented in Appendix B. In Appendix C we discuss some proprietary presentation tools.

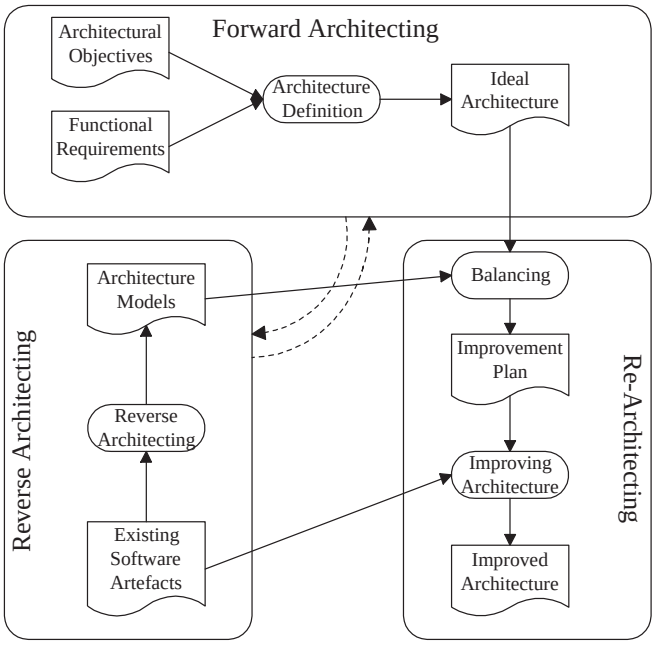


Figure 2.3: Architecture Improvement Process

2.4 Architecture Improvement

One can improve a system by starting from scratch again and rebuilding the complete system. However, this is hardly an option for systems containing software of hundreds of person-years' development. Another approach starts with the existing system as a basis and incrementally improves (parts of) the system.

Figure 2.3 (taken from [Kri97]) shows a process for the latter approach, comprising three typical activities:

- *forward architecting* uses architectural objectives and functional requirements as input for the definition of an *ideal* architecture.
- *reverse architecting* consists of creating explicit architectural models of an existing system, the *as-built* architecture. Traditional reverse engineering techniques can be applied to extract information from software artefacts. Appropriate abstractions must be made to obtain the information at an architectural level.
- *re-architecting* involves balancing an ideal architecture against the existing architecture to prioritize a list of desired improvements. The

next step is to implement a number of these improvements. The size of the improvement steps depends on both business-related issues and technological facts.

Besides software architecting experience (including knowledge of architectural patterns), a lot of domain and system knowledge is required to define an ideal architecture. Reverse architecting can help in extracting domain knowledge from the system, but it can also clarify existing architectural patterns in the system. The recovered patterns may influence choices made during forward architecting. Of course, badly chosen patterns should not be copied by the new ideal architecture, but should be seen as cautions. The current system can give clues for defining a new architecture in a positive sense by recovering existing patterns. But it can also show the design decisions that failed in the past, which must be avoided in the new system.

The whole process is iterative, in the sense that after improvements have been implemented reverse architecting can help to make explicit the created architecture, which may differ from the initially defined architecture. This process is in fact similar to any improvement activity: define the current situation (or check the previously realized improvement), define the desired situation, and define the path to reach the desired situation and execute it.

2.5 The SAR Method

Relation Partition Algebra, architectural views, reconstruction levels, InfoPacks and ArchiSpects are the key elements of our Software Architecture Reconstruction (SAR) method. Architectural views have already been discussed in Section 1.2.3 and RPA will be presented in Chapter 3. Before we can focus on the details of our SAR method, we have to present the notion of reconstruction levels, InfoPacks, ArchiSpects, and a framework in to which these notions can be fitted.

2.5.1 Software Architecture Reconstruction Levels

We introduce different levels[1] of software architecture reconstruction. Each SAR level covers a range of architectural aspects that must be reconstructed.

[1] We have been inspired by the levels in the Capability Maturity Model [Hum89].

Consider a system which is hardly documented and whose software architecture is not known. Such systems are at the *initial* level of reconstruction. By making the software architecture of such a system explicit (i.e. *reverse architecting* the system), we reach the *described* level of SAR[2]. If the gap between the *ideal* software architecture and the described software architecture is too big, one must improve it by redefining parts of it. Then, by *re-architecting* the system, we reach the *redefined* SAR level. After the *architecture improvement*, one must sustain the reached quality level. Without any precautions, the architecture will certainly degenerate after a while. If we can continuously preserve the software architecture in a controlled way, we reach the *managed* level. Now that we have the software architecture completely under control, we can optimise the architecture for all kinds of future extensions, which is called the *optimised* level. So, the following software architecture reconstruction levels exist:

- initial level;
- described level;
- redefined level;
- managed level;
- optimised level.

2.5.2 InfoPack and ArchiSpect

We introduce the terms InfoPack and ArchiSpect as the components of our software architecture reconstruction method. An InfoPack[3] is a package of particular information extracted from the source code, design documents or any other information source. An InfoPack contains a description of the extraction steps to be taken to retrieve certain software information. Alternative extraction techniques may exist for different programming languages, which are discussed as parts of the InfoPack. Sometimes an InfoPack is specific to a certain (programming) language or class of languages, e.g. InfoPacks working with the notion of inheritance are of interest only for object-oriented languages. InfoPacks may also be domain- or application-dependent, which makes them less widely applicable. Examples are the *Import* InfoPack, import dependency extraction, and the *Part-Of* InfoPack, extraction of the decomposition hierarchy. An ArchiSpect[4] is a

[2] We may assume that each system (especially systems that exist for many years) contains some notion of software architecture, although it has not been explicitly documented.

[3] The term InfoPack is an abbreviation of the phrase *information package*

[4] The term ArchiSpect combines the words *architecture* and *aspect*.

view on the system that makes explicit a certain architectural structure. An ArchiSpect is more abstract than an InfoPack and therefore more widely applicable. Most ArchiSpects build upon the results of InfoPacks. A complete set of ArchiSpects in fact describe a system's actual architecture; the InfoPacks serve as supporting units to construct the ArchiSpects. Besides abstraction of information, an ArchiSpect covers possible ways of presenting architectural information. Examples are the *Component Dependency* ArchiSpect, recovery of dependency between the components of a system, and the *Layering Conformance* ArchiSpect, verifying whether a system is correctly layered.

In this thesis we describe InfoPacks and ArchiSpects according to a fixed scheme:

- *Name*: the name of the *InfoPack* or *ArchiSpect*;
- *Context*: the architectural view (see Section 1.2.3) to which it belongs and the related InfoPacks and ArchiSpects (as will be clarified in the *description*);
- *Description*: an introduction to the InfoPack and ArchiSpect;
- *Example*: typical example(s) from Philips' systems (as appeared after the *method* had been applied);
- *Method*: a description of the steps that must be taken to construct the InfoPack or ArchiSpect;
- *Discussion*: discussion of items not addressed in one of the above sections (e.g. discussion of related work).

We can look at InfoPacks and ArchiSpects in different ways. The *method* view focuses on the description of steps and guidance. The *tool* view concerns the tools required to support the application of ArchiSpects and InfoPacks. The *representation* view contains the results of ArchiSpects and InfoPacks. We will use the terms ArchiSpect and InfoPack for each of these views; the context in which the term is used will clarify its actual meaning.

Software Architecture Reconstruction Framework

For each of the SAR levels (besides the initial level[5]), we can fill out a matrix as given in Table 2.1; the rows contain the various SAR levels and the columns contain the architectural views. The cells have been filled with InfoPacks and ArchiSpects. InfoPacks are closely related to dedicated ex-

[5]The initial level refers to the situation that no reconstruction has taken place, so the cells are empty.

	Architectural Views				
SAR levels	Logical View	Module View	Code View	Execution View	Physical View
Optimized	...	...	...	...	...
Managed	...	*ArchiSpect*	*ArchiSpect* *InfoPack*	...	...
Redefined	...	*ArchiSpect*	*ArchiSpect* *InfoPack*	...	...
Described	...	*ArchiSpect*	*ArchiSpect* *InfoPack*	...	...
Initial					

Table 2.1: Software Architecture Reconstruction Framework

traction means and therefore they appear in the code view column. All architectural views are in fact important, but in this thesis we will concentrate on the module view and code view (the non-dotted area in the SAR matrix). In Chapter 7 we will fill out the SAR matrix with the names of discussed InfoPacks and ArchiSpects as we experienced to be useful for the various SAR levels.

Chapter 3

Relation Partition Algebra

In the previous chapter we presented in general terms the SAR method. The underlying model of the SAR method consists of Relation Partition Algebra which is introduced in this chapter. In the succeeding chapters we heavily use Relation Partition Algebra to describe the details of the SAR method.

3.1 Introduction

In this chapter we introduce Relation Partition Algebra (RPA). RPA is based on sets and binary relations. This chapter serves as a brief introduction to RPA, and it is also meant to introduce notations that will be used throughout the thesis.

RPA has been defined in order to be able to formalise descriptions of (parts of) software architectures. Furthermore, in the context of reverse engineering one often wants to query the software structure. RPA offers abilities to express questions in a formal notation, which can be executed on the actual (model of the) software. Throughout this thesis, we will see many applications of RPA for reconstructing software architectures or beautyfing presentations of architectural information.

We will start this chapter by discussing sets and operations on sets. In Section 3.3 binary relations and operations upon them will be presented. The proofs of algebraic laws relating to RPA will not be given here, but we will refer to published work [SM77, FKO98, FK99, FO99]. In Section 3.5 we will extend RPA with multi-sets and multi-relations. RPA formulas can

also be executed; related issues will be discussed in Section 3.6.

3.2 Sets

3.2.1 Primitives of Set Theory

A set is a collection of objects, called elements or members. If x is an element of S, given any object x and set S, we write $x \in S$. The notion of *set* and the relation *is-element-of* are the primitive concepts of set theory. We rely on a common understanding of the meaning of these terms.

A finite set can be specified explicitly by enumerating its elements. The elements are separated by commas, and the enumeration is enclosed within brackets. So, the set which contains elements a, b, and c is denoted by $\{a, b, c\}$. Infinite sets cannot be listed explicitly, so these sets are described implicitly. A set can be described using a predicate with a free variable. The set $\{x \in U | P(x)\}$, for given U (another set playing the role of *universe*), denotes the set S such that $x \in S$ if and only if $x \in U$ and $P(x)$ holds.

We will use the logical operators $\vee$ and $\wedge$ to denote the logical *(inclusive) or* and the logical *and*, respectively. $a \vee b$ holds if and only if a is true or b is true or both a and b are true. $a \wedge b$ holds if and only if a and b are true. Furthermore, $a \Rightarrow b$ holds if a is true then b is true. $a \iff b$ holds if and only if $a \Rightarrow b \wedge b \Rightarrow a$.

At the end of each section we will illustrate the discussed operators with a running example.

example

$$
\begin{aligned}
\mathit{Subsystems} &= \{\mathit{OS}, \mathit{Drivers}, \mathit{DB}, \mathit{App}\} \\
\mathit{Functions} &= \{\mathit{main}, a, b, c, d\} \\
\mathit{InitFunctions} &= \{f | f \in \mathit{Functions} \wedge f \text{ is called at initialisation time}\}
\end{aligned}
$$

3.2.2 Operations on Sets

equal, subset, superset, size

Two sets S_1 and S_2 are *equal*, denoted by $S_1 = S_2$, if for each x it holds that $x \in S_1 \iff x \in S_2$. A set S_1 is contained in S_2, or S_1 is a *subset* of S_2 denoted by $S_1 \subseteq S_2$, if for each x it holds that $x \in S_1 \Rightarrow x \in S_2$. A similar definition holds for a *superset*, $S_1 \supseteq S_2$, which is an alternative notation for $S_2 \subseteq S_1$. A *strict* subset (superset) is a subset (superset) from which equality is excluded. It is denoted by $\subset$ respectively $\supset$. The number of elements in a finite set is called the *size*, denoted by $|S|$.

union, intersection

The *union* of two sets S_1 and S_2, denoted by $S_1 \cup S_2$, is the set $T = \{x | x \in S_1 \vee x \in S_2\}$. The *intersection* of two sets S_1 and S_2, denoted by $S_1 \cap S_2$, is the set $T = \{x | x \in S_1 \wedge x \in S_2\}$.

difference, complement

The *difference* of two sets S_1 and S_2, denoted by $S_1 \setminus S_2$, is the set $T = \{x | x \in S_1 \wedge x \notin S_2\}$. It is also called the *relative complement* of S_2 with respect to S_1. The complement of a set S, denoted by $\overline{S}$, is the set $T = \{x | x \notin S\}$. Given that U is the universe, containing all elements, the complement of a set S can be written as: $\overline{S} = U \setminus S$.

example

$$
\begin{aligned}
\mathit{Subsystems} &= \{\mathit{OS}, \mathit{Drivers}, \mathit{DB}, \mathit{App}\} \\
\mathit{InitFunctions} &\subseteq \mathit{Functions} \\
\mathit{UpperLayers} &= \{\mathit{DB}, \mathit{App}\} \\
\overline{\mathit{UpperLayers}} &= \{\mathit{OS}, \mathit{Drivers}\} \quad \text{(with respect to } \mathit{Subsystems}\text{)}
\end{aligned}
$$

3.3 Binary Relations

3.3.1 Primitives of Binary Relations

Besides the notion of sets, we need more to describe software structures. Relationships between (software) entities play an important role in architecture and design. Binary relations can express such relationships. For example, function-calls within a system can be seen as the binary relation named *calls*.

A *binary relation*, or shortly a *relation*, from X to Y is a subset of the cartesian product $X \times Y$. It is a set of tuples $\langle x, y \rangle$ where $x \in X$ and $y \in Y$. Tuples of a binary relation R can be denoted in different ways. The following notations are used to refer to an element of a binary relation:

- infix notation: xRy
- prefix notation: $R(x, y)$
- tuple notation: $\langle x, y \rangle$

In relational terms $calls(main, a)$ is an abstraction of the following program fragment (written in the programming language C [KR88]):

```
void main () {
    ....
    a(12, i, &ref);
    ....
}
```

Besides a textual representation of relations, one can also represent a relation in a directed graph. A directed graph, or shortly *digraph*, consists of a set of elements, called *vertices*, and a set of ordered pairs of these elements, called *arcs* [WW90]. Assume a digraph G represents the relation $R \subseteq X \times Y$. The arcs of G represent the tuples of R; the vertices represent elements of $X \cup Y$. The vertices with outgoing arcs are elements of X and vertices with incoming arcs are elements of Y. The *calls* relation of a (fictive) program is shown in Figure 3.1.

example

$$calls \quad = \quad \{\langle main, a\rangle, \langle main, b\rangle, \langle a, b\rangle, \langle a, c\rangle, \langle a, d\rangle, \langle b, d\rangle\}$$

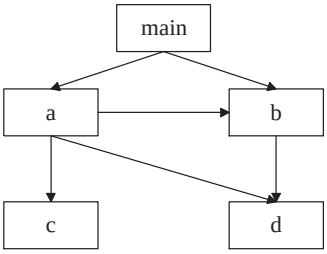


Figure 3.1: Directed Graph Representing *calls* Relation

3.3.2 Operations on Binary Relations

Relations are sets (of tuples), so we inherit the definitions of *equality*, *containment*, *size*, *union*, *intersection*, *difference* and *complement* from the previous section.

converse

The *converse* of relation R, denoted by R^{-1}, is obtained by reversing the tuples of R: $R^{-1} = \{\langle y, x\rangle | \langle x, y\rangle \in R\}$.

product, identity

The cartesian product of two sets X and Y, denoted by $X \times Y$, is the relation $R = \{\langle x, y\rangle | x \in X \wedge y \in Y\}$. A special relation Id_X, or just Id if the set X is obvious, is called the *identity* relation. It is defined as $Id_X = \{\langle x, x\rangle | x \in X\}$.

domain, range, carrier

The *domain* of a relation R, denoted by $dom(R)$, is the set $S = \{x | \langle x, y\rangle \in R\}$. The *range* of relation R, denoted by $ran(R)$, is the set $S = \{y | \langle x, y\rangle \in R\}$. The *carrier* of a relation R, denoted by $car(R)$, is defined as $dom(R) \cup ran(R)$.

restriction

The *domain restrict* of a relation R with respect to a set S, denoted by $R \upharpoonright_{dom} S$, is a relation $T = \{\langle x, y\rangle | \langle x, y\rangle \in R \wedge x \in S\}$. The *range restrict* of a relation R with respect to a set S, denoted by $R \upharpoonright_{ran} S$, is a relation $T = \{\langle x, y\rangle | \langle x, y\rangle \in R \wedge y \in S\}$. The *carrier restrict* of a relation R with respect to a set S, denoted by $R \upharpoonright_{car} S$, is a relation $T = \{\langle x, y\rangle | \langle x, y\rangle \in R \wedge x \in S \wedge y \in S\}$. The carrier restrict can also be defined as: $R \upharpoonright_{car} S = (R \upharpoonright_{dom} S) \upharpoonright_{ran} S$.

exclusion

A variant of restriction is exclusion. The *domain exclude* of a relation R with respect to a set S, denoted by $R \backslash_{dom} S$, is a relation $T = \{\langle x, y\rangle | \langle x, y\rangle \in R \wedge x \notin S\}$. The *range exclude* of a relation, denoted by $R \backslash_{ran} S$, is a relation $T = \{\langle x, y\rangle | \langle x, y\rangle \in R \wedge y \notin S\}$. The *carrier exclude* of a relation R, denoted by $R \backslash_{car} S$, is a relation $T = \{\langle x, y\rangle | \langle x, y\rangle \in R \wedge x \notin S \wedge y \notin S\}$. The carrier exclude can also be defined as $R \backslash_{car} S = (R \backslash_{dom} S) \backslash_{ran} S$.

top, bottom

The *top* of a relation R, denoted by $\top(R)$, is defined as $dom(R) \setminus ran(R)$. Given a directed graph of relation R, the top consists of vertices that are a root. A root is a vertex that has no incoming arcs. Similarly, the *bottom* of a relation R, denoted by $\bot(R)$, is defined as $ran(R) \setminus dom(R)$. They are the leaf vertices of a directed graph, which are the vertices with no outgoing arcs.

projection

The *forward projection* of set S in relation R, denoted by $S \rhd R$, is the set $T = \{y | \langle x, y\rangle \in R \wedge x \in S\}$. The *backward projection* of S in R, denoted by $R \lhd S$, is the set $T = \{x | \langle x, y\rangle \in R \wedge y \in S\}$. Forward projection can also be defined as $S \rhd R = ran(R \upharpoonright_{dom} S)$ and the backward projection can be defined as $R \lhd S = dom(R \upharpoonright_{ran} S)$.

The *left image* of a relation R with respect to element y, denoted by $R.y$, is the set $T = \{x | \langle x, y\rangle \in R\}$. The *right image* of a relation R with respect to element x, denoted by $x.R$, is the set $T = \{y | \langle x, y\rangle \in R\}$.

composition

The *composition* of two relations R_1 and R_2, denoted by $R_2 \circ R_1$, is the relation $R = \{\langle a, b\rangle | \exists x \,.\, \langle a, x\rangle \in R_1 \wedge \langle x, b\rangle \in R_2\}$. $R_1; R_2$ is an alternative notation of the composition $R_2 \circ R_1$. One should pronounce $R_2 \circ R_1$ as "apply R_2 after R_1".

Composing a relation n times, $R \circ R \circ \ldots \circ R$ is denoted by R^n. Note that composition is associative (proof is given in [FO99]), so we may omit parentheses around each composition. Furthermore, by definition $R^0 = Id$.

transitive closure

The *transitive closure* of a relation R, denoted by R^+, is the relation $T = \bigcup_{i=1}^{\infty} R^i$, i.e. the union of all R^i. The *reflexive transitive closure* R^* is $R^0 \cup R^+ = Id \cup R^+$.

Special algorithms have been developed to calculate the transitive closure efficiently. In 1962 Warshall [War62] described an $O(n^3)$ algorithm (where n is the size of the carrier of the relation):

```
for i in S do
  for j in S do
    for k in S do
      T[j,k] = T[j,k] + T[j,i] x T[i,k]
```

explanation

The array T represents the existence (boolean value) of tuples $\langle i, j\rangle$ in the given relation. The set S equals the carrier of this relation. Each of the for-loops enumerates the elements in the set. The `+` operation is defined as the logical *or* operation and the `x` operation is defined as the logical *and*. One should read the last statement as follows (having a digraph representation in mind): if there is a path from j to i and there is a path from i to k, then there exists a path from j to k.

As an example we present the transitive closure of the *calls* relation in Figure 3.2.

reduction

A relation R is cycle-free if and only if $Id \cap R^+ = \emptyset$, in other words, in a graph representation of relation R, there is no path from any vertex to

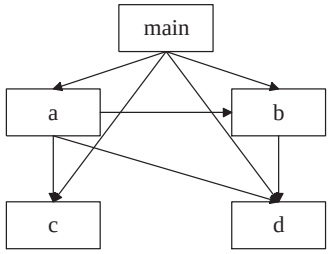


Figure 3.2: Transitive closure of *calls*

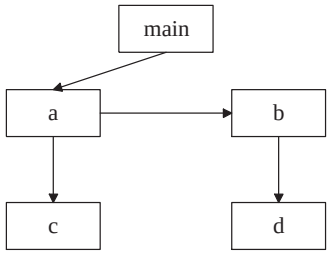


Figure 3.3: Hasse of *calls*

itself that contains an arbitrary number n ($n > 0$) of edges.

The *transitive reduction* of a cycle-free relation R, denoted by R^-, is a relation containing all tuples of R except for *short-cuts*. For example, the tuple $\langle x, z\rangle$ is a short-cut if R contains the tuples $\langle x, y\rangle$ and $\langle y, z\rangle$. The transitive reduction of R is also called the *Hasse* [SM77] of R, or the *poset* of R. The transitive reduction of a cyclic-free relation R can also be defined as $R^- = R \setminus (R \circ R^+)$.

The expression $R \circ R^+$ represents all the pairs of elements in R that can reach each other indirectly (so via another vertex in the digraph). When we substract these tuples from the original relation R we retain the tuples which are not a shortcut. The Hasse of the *calls* relation is illustrated in Figure 3.3.

example

$$\begin{aligned} maincalls &= \{\langle main, a\rangle, \langle main, b\rangle\} \\ maincalls &\subseteq calls \\ calls \setminus maincalls &= \{\langle a, b\rangle, \langle a, c\rangle, \langle a, d\rangle, \langle b, d\rangle\} \end{aligned}$$

$$
\begin{aligned}
\overline{calls} &= \{\langle main, main\rangle, \langle main, c\rangle, \langle main, d\rangle, \langle a, main\rangle,\\
&\quad \langle a, a\rangle, \langle b, main\rangle, \langle b, a\rangle, \langle b, b\rangle, \langle b, c\rangle,\\
&\quad \langle c, main\rangle, \langle c, a\rangle, \langle c, b\rangle, \langle c, c\rangle, \langle c, d\rangle, \langle d, main\rangle,\\
&\quad \langle d, a\rangle, \langle d, b\rangle, \langle d, c\rangle, \langle d, d\rangle\}\\
&\qquad \text{(with respect to } Functions \times Functions\text{)}\\
\top(calls) &= \{main\}\\
\bot(calls) &= \{c, d\}\\
calls \restriction_{dom} \{main\} &= \{\langle main, a\rangle, \langle main, b\rangle\}\\
calls \backslash_{dom} \{main\} &= \{\langle a, b\rangle, \langle a, c\rangle, \langle a, d\rangle, \langle b, d\rangle\}\\
\{main\} \rhd calls &= \{a, b\}\\
calls.b &= \{main, a\}\\
calls^{+} &= \{\langle main, a\rangle, \langle main, b\rangle, \langle main, c\rangle, \langle main, d\rangle,\\
&\quad \langle a, b\rangle, \langle a, c\rangle, \langle a, d\rangle, \langle b, d\rangle\}\\
Hasse(calls) &= \{\langle main, a\rangle, \langle a, b\rangle, \langle a, c\rangle, \langle b, d\rangle\}
\end{aligned}
$$

3.4 Part-Of relations

A *partition* of a non-empty set A is a collection of non-empty sets such that the union of these sets equals A and the intersection of any two distinct subsets is empty. We can see a partition as a division of a pie into different slices.

If we give each of these subsets a name we can construct a so-called *part-of* relation which describes a partition. Assume that these names are defined in a set T, then the part-of relation P is defined as follows: $P = \{\langle x, t\rangle | t \in T \wedge x \text{ is in the subset named } t\}$. In source code, function definitions are contained in a (single) file, so the relation between *Functions* and *Files* is an example of a part-of relation; see Figure 3.4.

Partitions and part-of relations are alternative views on the same concept: decomposition. A third view on decomposition comprises equivalence relations. One can derive an equivalence relation E from a part-of relation P as follows: $E = \{\langle x, y\rangle | \exists t\ \langle x, t\rangle \in P \wedge \langle y, t\rangle \in P\}$. The equivalence relation can also be defined as $E = P^{-1} \circ P$.

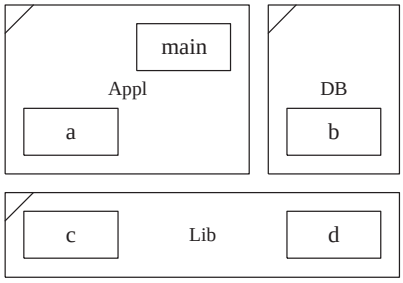


Figure 3.4: Partitioning *Functions*

example

$$\begin{aligned} T &= \{Appl, DB, Lib\} \\ partof &= \{\langle main, Appl\rangle, \langle a, Appl\rangle, \langle b, DB\rangle, \langle c, Lib\rangle, \langle d, Lib\rangle\} \\ eqrel &= partof^{-1} \circ partof \\ &= \{\langle main, main\rangle, \langle main, a\rangle, \langle a, a\rangle, \langle a, main\rangle, \langle b, b\rangle, \\ &\quad \langle c, c\rangle, \langle c, d\rangle, \langle d, d\rangle, \langle d, c\rangle\} \end{aligned}$$

lifting, lowering

Given a relation R and a part-of relation P we can construct a new relation Q by *lifting* R using P, denoted by $R \uparrow P$. The result is the relation $Q = \{\langle x, y\rangle | \exists a, b\ \langle a, b\rangle \in R \wedge \langle a, x\rangle \in P \wedge \langle b, y\rangle \in P\}$. Note that the carrier of relation R must be a subset of the domain of P.

We can also construct a new relation Q by *lowering* R using P, denoted by $R \downarrow P$. The resulting relation is defined as $Q = \{\langle x, y\rangle | \exists\ a, b\ \langle a, b\rangle \in R \wedge \langle x, a\rangle \in P \wedge \langle y, b\rangle \in P\}$. The carrier of relation R must be a subset of the range of P.

The given definition of lifting is in fact an *existential* lifting. An alternative to the above definition of lifting is *universal lifting*, denoted by $R \uparrow_\forall P$. The tuple $\langle c_1, c_2\rangle$ is an element of $R \uparrow_\forall P$ if and only if for all x_1 in c_1 and for all x_2 in c_2 it holds that $\langle x_1, x_2\rangle \in R$. We can now relate lifting and lowering as follows: $R = (R \downarrow P) \uparrow_\forall P)$. Very little use is made of universal lifting, so this alternative definition of lifting will not be used after this section[1].

[1] We will therefore write $\uparrow$ instead of $\uparrow_\exists$.

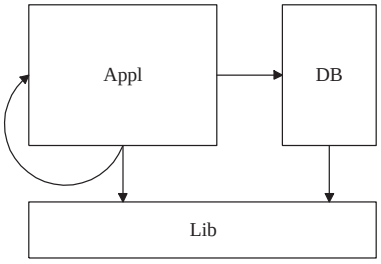


Figure 3.5: Lifting *calls*

example

Figure 3.5 shows the lifted *calls* relation, which is calculated as follows:

$$\begin{aligned} callsLift &= calls \uparrow partof \\ &= \{\langle Appl, Appl\rangle, \langle Appl, DB\rangle, \langle Appl, Lib\rangle, \langle DB, Lib\rangle\} \end{aligned}$$

3.5 Introducing multiplicity in RPA

While reverse architecting the *Med* system, we discovered that it is also useful to attribute a weight to the tuples of a relation. This leads to the introduction of multiplicity into RPA, by means of multi-relations. In the Chapter 5 we will see the importance of applying multi-relations which is also illustrated by examples.

A multi-relation is a collection of tuples in which each tuple may occur more than once. We will represent the tuples and their corresponding weights as a triple $\langle x, y, n\rangle$, where n is the number of occurrences of the tuple $\langle x, y\rangle$. In a running system the number of $calls(a, b)$ may be of interest when looking at e.g. recursion: $\{\ldots, \langle a, b, 7\rangle, \ldots\}$ is the representation of a calls b seven times.

Multi-relations compare to relations as bags (or multi-sets) compare to sets. Multi-sets (or bags) can be represented as sets of tuples, with the second argument being the number of occurrences of the first argument.

3.5.1 Calculating with weights

We must first describe the basics for calculating with weights before we can define multi-sets and multi-relations and their operations. Weights are natural numbers (the set $\{0, 1, 2, \ldots\}$ denoted by $\mathbb{N}$) extended with an explicit value ∞. The arithmetic operations $+$ and $\times$ work as usual if both arguments are elements of $\mathbb{N}$. Their behaviour when applied to ∞ is given by the following rules that hold for all $n \in \mathbb{N}$:

$$\begin{aligned} n + \infty &= \infty + n = \infty \\ \infty + \infty &= \infty \\ 0 \times \infty &= \infty \times 0 = 0 \\ n \neq 0 &\Rightarrow n \times \infty = \infty \times n = \infty \\ \infty \times \infty &= \infty \end{aligned}$$

Substraction of weights is also special. Take for example the rule $(n - m) + m = n$ which only satisfies when $n \geq m$. If $n < m$, then we define $n - m = 0$; we must note that the algebraic law $(n - m) + k = (n + k) - m$ does not hold. Furthermore, the following rules are given, for all $n \in \mathbb{N}$:

$$\begin{aligned} n - \infty &= 0 \\ \infty - \infty &= 0 \\ \infty - n &= \infty \end{aligned}$$

The minimum of two weights is denoted by $min(n_1, n_2)$, to which we add the rules that $min(\infty, n) = min(n, \infty) = n$ and $min(\infty, \infty) = \infty$. Similarly, we define $max(n_1, n_2)$, to which we add the rules $max(\infty, n) = max(n, \infty) = \infty$ and $max(\infty, \infty) = \infty$.

By definition, elements that are not members of a multi-set or multi-relation have a weight of 0. This simplifies the definitions of operations on multi-sets and multi-relations.

3.5.2 Operations on Multi-Sets

mapping

The n-mapping of a set T to a multi-set S, denoted by $\lceil T \rceil_n$, is defined as $S = \{\langle x, n\rangle | x \in T\}$. Furthermore, we define $\lceil T \rceil = \lceil T \rceil_1$. The mapping of a multi-set S to a set T, denoted by $\lfloor S \rfloor$, is defined as $T = \{x | \langle x, n\rangle \in S \wedge n > 0\}$.

equal, subset, superset, size

Two multi-sets are *equal*, denoted by $S_1 = S_2$, if for each e it holds that $\langle e, n\rangle \in S_1 \iff \langle e, n\rangle \in S_2$. A multi-set S_1 is a subset of S_2, denoted by $S_1 \subseteq S_2$ if for each e it holds that $\langle e, n_1\rangle \in S_1 \wedge \langle e, n_2\rangle \in S_2 \Rightarrow n_1 \leq n_2$. A multi-set S_1 is a superset of S_2, denoted by $S_1 \supseteq S_2$ if for each e it holds that $\langle e, n_1\rangle \in S_1 \wedge \langle e, n_2\rangle \in S_2 \Rightarrow n_1 \geq n_2$. The *size* of a multi-set S, denoted by $\|S\|$, is defined as $\sum_{\langle x,n\rangle \in S} n$.

union, addition, intersection, difference

The *intersection* of two multi-sets S_1 and S_2, denoted by $S_1 \cap S_2$, is defined as the multi-set $T = \{\langle e, n\rangle | \langle e, n_1\rangle \in S_1 \wedge \langle e, n_2\rangle \in S_2 \wedge n = min(n_1, n_2)\}$. The *union* of two multi-sets S_1 and S_2, denoted by $S_1 \cup S_2$, is defined as $T = \{\langle e, n\rangle | \langle e, n_1\rangle \in S_1 \vee \langle e, n_2\rangle \wedge n = max(n_1, n_2)\}$.

The *difference* between two multi-sets, denoted by $S_1 \setminus S_2$, is defined as $S = \{\langle e, n\rangle | \langle e, n_1\rangle \in S_1 \wedge \langle e, n_2\rangle \in S_2 \wedge n = n_1 - n_2)\}$. The *addition* of two multi-sets, denoted by $S_1 + S_2$, is defined as $S = \{\langle e, n\rangle | \langle e, n_1\rangle \in S_1 \wedge \langle e, n_2\rangle \in S_2 \wedge n = n_1 + n_2)\}$.

complement

The *complement* of a multi-set S_1 with respect to the set S is defined as: $\overline{S_1} = \lceil S \rceil_\infty \setminus S_1$.

3.5.3 Operations on Multi-Relations

mapping

The n-mapping of a relation R to a multi-relation M, denoted by $\lceil R \rceil_n$, is defined as $M = \{\langle x, y, n\rangle | \langle x, y\rangle \in R\}$. Furthermore, we define $\lceil R \rceil = \lceil R \rceil_1$. The mapping of a multi-relation M to a relation R, denoted by $\lfloor M \rfloor$, is defined as $R = \{\langle x, y\rangle | \langle x, y, n\rangle \in M\}$.

equal, subset, superset, size

Two multi-relations are *equal*, denoted by $M_1 = M_2$, if for each x and y it holds that $\langle x, y, n\rangle \in M_1 \iff \langle x, y, n\rangle \in M_2$. A relation M_1 is *contained* in M_2, denoted by $M_1 \subseteq M_2$, if for each x and y it holds that $\langle x, y, n\rangle \in M_1 \wedge \langle x, y, m\rangle \in M_2 \Rightarrow n \leq m$. Similarly to binary relations $\supseteq$, $\subset$ and $\supset$ are defined for multi-relations. The *size* of a multi-relation M, denoted by $\|M\|$, is defined as $\sum_{\langle x, y, n\rangle \in M} n$.

union, addition, intersection, difference

The *union* of two multi-relations M_1 and M_2, denoted by $M_1 \cup M_2$, is the multi-relation $M = \{\langle x, y, n\rangle | (\langle x, y, n_1\rangle \in M_1 \vee \langle x, y, n_2\rangle \in M_2) \wedge n = max(n_1, n_2)\}$. The *addition* of two multi-relations M_1 and M_2, denoted by $M_1 + M_2$, is the multi-relation $M = \{\langle x, y, n\rangle | (\langle x, y, n_1\rangle \in M_1 \vee \langle x, y, n_2\rangle \in M_2) \wedge n = n_1 + n_2\}$. The *intersection* of two multi-relations M_1 and M_2, denoted by $M_1 \cap M_2$, is the relation $M = \{\langle x, y, n\rangle | (\langle x, y, n_1\rangle \in M_1 \wedge \langle x, y, n_2\rangle \in M_2) \wedge min(n_1, n_2)\}$. The *difference* between two multi-relations M_1 and M_2, denoted by $M_1 \setminus M_2$, is the relation $M = \{\langle x, y, n\rangle | (\langle x, y, n_1\rangle \in M_1 \wedge \langle x, y, n_2\rangle \in M_2) \wedge n = n_1 - n_2\}$.

converse

The *converse* of relation M, denoted by M^{-1}, is obtained by reversing the first two arguments of the triples: $M^{-1} = \{\langle y, x, n\rangle | \langle x, y, n\rangle \in M\}$.

product, identity

The cartesian product of two multi-sets X and Y, denoted by $X \times Y$, is the multi-relation $M = \{\langle x, y, n\rangle | \langle x, n_1\rangle \in X \wedge \langle y, n_2\rangle \in Y \wedge n = n_1 \times n_2\}$. A special multi-relation $Id_{X,n}$, or just Id_n if the set X is obvious, is called the *identity* relation. The identity of a set X is defined as $Id_{X,n} = \lceil Id_X \rceil_n$. When omitted, n must be considered to be 1.

domain, range, carrier

The *domain* of a multi-relation M, denoted by $dom(M)$, is the multi-set $S = \{\langle x, n\rangle | n = \sum_{\langle x, y, m\rangle \in R} m\}$. The *range* of a multi-relation M, denoted by $ran(M)$, is the multi-set $S = \{\langle y, n\rangle | n = \sum_{\langle x, y, m\rangle \in R} m\}$. The *carrier* of a multi-relation M, denoted by $car(M)$, is defined as $dom(M) + ran(M)$.

restriction

The *domain restrict* of a multi-relation M with respect to a set S, denoted by $M \upharpoonright_{dom} S$, is a multi-relation $T = \{\langle x, y, n\rangle | \langle x, y, n\rangle \in M \wedge x \in S\}$. The *range restrict* of a multi-relation M with respect to a set S, denoted by $M \upharpoonright_{ran} S$, is a multi-relation $T = \{\langle x, y, n\rangle | \langle x, y, n\rangle \in M \wedge y \in S\}$. The *carrier restrict* of a multi-relation M with respect to a set S, denoted by $M \upharpoonright_{car} S$, is a relation $T = \{\langle x, y, n\rangle | \langle x, y, n\rangle \in M \wedge x \in S \wedge y \in S\}$. The carrier restrict can also be defined as: $M \upharpoonright_{car} S = (M \upharpoonright_{dom} S) \upharpoonright_{ran} S$.

exclusion

The *domain exclude* of a multi-relation M with respect to a set S, denoted by $M \backslash_{dom} S$, is a relation $T = \{\langle x, y, n\rangle | \langle x, y, n\rangle \in R \wedge x \notin S\}$. The *range exclude* of a multi-relation M with respect to a set S, denoted by $M \backslash_{ran} S$, is a relation $T = \{\langle x, y, n\rangle | \langle x, y, n\rangle \in R \wedge y \notin S\}$. The *carrier exclude* of a multi-relation M, denoted by $M \backslash_{car} S$, is a relation $T = \{\langle x, y, n\rangle | \langle x, y, n\rangle \in M \wedge x \notin S \wedge y \notin S\}$. The carrier exclude can also be defined as $M \backslash_{car} S = (M \backslash_{dom} S) \backslash_{ran} S$.

top, bottom

The *top* of a multi-relation M, denoted by $\top(M)$, is defined as $\top(M) = dom(M \upharpoonright_{dom} \top(\lfloor M \rfloor))$. The *bottom* of a multi-relation M, denoted by $\bot(M)$, is defined as $\bot(M) = ran(M \upharpoonright_{ran} \bot(\lfloor M \rfloor))$.

projection

The *forward projection* of a set S in a multi-relation M, denoted by $S \rhd M$, is the multi-set $T = \{\langle y, n\rangle | n = \sum_{\langle x, y, m\rangle \in M \wedge x \in S} m\}$. The *backward projection* of a set S in a multi-relation M, denoted by $M \lhd S$, is the set $T = \{\langle x, n\rangle | n = \sum_{\langle x, y, m\rangle \in M \wedge y \in S} m\}$. Forward projection can also be defined as $S \rhd M = ran(M \backslash_{dom} S)$ and the backward projection can be defined as $M \lhd S = dom(M \backslash_{ran} S)$.

The *left image* of a multi-relation M of y, denoted by $M.y$, is the multi-set $T = \{\langle x, n\rangle | \langle x, y, n\rangle \in M\}$. The *right image* of a multi-relation M of x, denoted by $x.M$, is the multi-set $T = \{\langle y, n\rangle | \langle x, y, n\rangle \in M\}$.

composition

The composition two multi-relations M_1, denoted by $M_2 \circ M_1$, is defined as $M = \{\langle x, z, n\rangle | n = \sum_{\langle x, y, n_1\rangle \in M_1 \wedge \langle y, z, n_2\rangle \in M_2} n_1 \times n_2\}$.

Given a matrix representation of M_1 and M_2, where the cells contain the weight of a tuple $\langle x, y\rangle$, the composition consists of the multiplication of both matrices [FK99]. Given a representation of a directed graph with weighted edges, the composition consists of the number of all possible paths from x to z, by taking two steps: the first step in M_1 and the second step in M_2.

transitive closure

The *transitive closure* of a multi-relation M, denoted by M^+, is defined as $M^+ = \bigcup_{i=1}^{\infty} M^i$, i.e. the union of all M^i. The *reflexive transitive closure*, denoted by M^*, is defined as $M^0 \cup M^+$.

Warshall's algorithm for calculating transitive closures must be adapted a bit. Here, we give the adapted Warshall algorithm, the proof of correctness is given in [FK99].

```
for i in S do
  for j in S do
    for k in S do
      if T[i,i] == 0
      then T[j,k] = T[j,k] + T[j,i] x T[i,k]
      else T[j,k] = T[j,k] + INFTY x T[j,i] x T[i,k]
```

explanation

T represents the two-dimensional (associative) array which initially contains the multi-relation m. After completion, T contains the multi-relation m^+. The value of $T[i,j]$ represents the weight of tuple $\langle i,j \rangle$. The set S is the carrier of this multi-relation. Addition and multiplication work as defined in Section 3.5.1. Comparing this algorithm with the original one, we see that the factor INFTY is introduced when there is a path $j-i$ and $i-k$ and $T[i,i] \neq 0$. If there is a path from j to i and from i to k and there are paths from i to i (expressed by $T[i,i] \neq 0$), one can reach k ∞ times from j.

reduction

The Hasse of a cycle-free multi-relation M, denoted by M^+, is defined as $M \setminus (M^+ \circ M)$.

lifting, lowering

Given a relation M and a part-of relation P we can construct a new multi-relation Q by *lifting* M using P, denoted by $M \uparrow P$. The result is the multi-relation $Q = \{\langle x,y,n \rangle | n = \sum_{\exists a,b\ \langle a,b,m \rangle \in M \wedge \langle a,x \rangle \in P \wedge \langle b,y \rangle \in P} m\}$.

Given a relation M and a part-of relation P we can construct a new multi-relation Q by *lowering* M using P, denoted by $M \downarrow P$. The resulting multi-relation is defined as $Q = \{\langle x,y,n \rangle | \langle a,b,n \rangle \in M \wedge \langle x,a \rangle \in P \wedge \langle y,b \rangle \in P\}$.

Lifting and lowering (for a relation R as well as for a multi-relation R) can also be defined in terms of composition:

$$\begin{aligned} R \uparrow P &= \lceil P \rceil \circ R \circ \lceil P^{-1} \rceil \\ R \downarrow P &= \lceil P^{-1} \rceil \circ R \circ \lceil P \rceil \end{aligned}$$

$^*, ^+, ^-, ^{-1}$
$\circ, \uparrow, \downarrow$
$\times$
$\cap$
$\cup, +$
$\lhd, \rhd, .$
$dom, ran, car, \top, \bot$
$\restriction_{dom}, \restriction_{ran}, \restriction_{car}, \backslash_{dom}, \backslash_{ran}, \backslash_{car}, \backslash$
$\subseteq, \supseteq, \subset, \supset$
$=$
$\lceil\ \rceil, \lfloor\ \rfloor, \mid\ \mid, \parallel\ \parallel, \overline{}$

Table 3.1: RPA Operator Precedences

3.6 RPA Formulas in Action

In the next chapters we will use RPA formulas to express e.g. abstractions of software information. Before we can define these (composed) formulas, we have to explain how we must interpret these formulas: precedences of operators, and the notations applied for sets, relations and multi-relations. In this section we will also discuss how a given formula can be executed on a computer.

3.6.1 Precedences of Operations

When we combine operators to construct larger expressions, we must say something about the order in which the operators must be applied. Precedence levels of operators indicate the way in which an expression is implicitly grouped into separate parts. In fact, precedence levels automatically place parentheses around parts of the expression to prescribe the order in which the operators are to be applied. In the case of equal precedence level, we apply the left-associative rule, meaning that e.g. $a+b+c = (a+b)+c$. The mapping, size, and complement operators already group expressions by their notations. The precedence levels of the RPA operators are given in Table 3.1 (at the top of the table are the operators with highest precedence). In this thesis we will often use parentheses in formulas for reasons of readability.

3.6.2 Notational Aspects

We will use a special notation to distinguish various sets, multi-sets, relations and multi-relations. For sets and multi-sets we will use the same notation e.g. a set of functions will be denoted by *Functions*. A relation representing function calls in a system, $calls \subseteq Functions \times Functions$, will be denoted by $calls_{Functions,Functions}$. For multi-relations, we will use a similar notation, except that we will emphasize multiplicity as follows $calls_{\mathbf{Functions,Functions}}$. Using this notation, we can immediately qualify the relation's domain and range. Relations with the same base names, but operating on different domains and in different ranges can be easily identified.

3.6.3 Execution of RPA formulas

We use many RPA formulas to describe the software architecture reconstruction method. Each RPA formula can be easily transformed into an executable code. Sets, multi-sets, relations and multi-relations are expressed in special formatted files on the file system. For example, the file named `calls.Functions.Functions` contains the $calls_{Functions,Functions}$ relation. The application of an operator to one or more operands consists in calling the appropriate program or function given the proper input files. A discussion of some RPA implementations is given in Appendix B.

example

Consider the following formulas (copied from Section 4.10):

$$\begin{aligned} imports_{Files,Comps} &= partof_{Files,Comps} \circ imports_{Files,Files} \\ importsExt_{Files,Comps} &= imports_{Files,Comps} \setminus partof_{Files,Comps} \\ UsingExts &= dom(importsExt_{Files,Comps}) \\ using_{Files,Comps} &= partof_{Files,Comps} \restriction_{dom} UsingExts \end{aligned}$$

Initially, we have the following files (representing relations), which are results of extraction tools:

- `imports.Files.Files` representing $imports_{Files,Files}$;
- `partof.Files.Comps` representing $partof_{Files,Comps}$.

We can translate the above formulas straightforwardly into executable code (e.g. executed in a *Unix* shell). We do not need any knowledge of the semantics of the formulas to make this translation[2].

```
rk_csh: rel_comp partof.Files.Comps imports.Files.Files   \
          > imports.Files.Comps
rk_csh: rel_diff imports.Files.Comps partof.Files.Comps   \
          > importsExt.Files.Comps
rk_csh: rel_dom importsExt.Files.Comps > UsingExts
rk_csh: rel_domR partof.Files.Comps UsingExts \
          > using.Files.Comps
```

After we have performed the calculations we have the following files:

- `imports.Files.Files` ($imports_{Files,Files}$);
- `partof.Files.Comps` ($partof_{Files,Comps}$);
- `imports.Files.Comps` ($imports_{Files,Comps}$);
- `importsExt.Files.Comps` ($importsExt_{Files,Comps}$);
- `UsingExts` ($UsingExts$);
- `using.Files.Comps` ($using_{Files,Comps}$).

3.7 Discussion

Work relating to Relation Partition Algebra has already been discussed in [FO94, FKO98]. From [FKO98] we pick out the work of Holt [Hol96, Hol98], as it shows a remarkable correspondence to our work. Though RPA has been developed independently, both approaches use binary relational algebra (Tarski Relational Algebra [Tar41]) to describe rules in software architecture and re-engineering applications. There are differences between both algebra's. For example, Holt [Hol98] defines an induction operator; it is defined as $C \circ R \circ P$; C is a containment relation and P is a parent relation ($P = C^{-1}$). Holt does *not* define containment relation as a representation of a (hierarchical) partitioning. This means that a module A may be contained in component X as well as component Y. In RPA, the *lift* operator is more carefully defined in this respect.

Furthermore, Holt treats a system's hierarchical decomposition as a single containment relation, so he does not distinguish different levels of contain-

[2] In this *Unix* shell the prompt is named `rk_csh:`, the backslash informs the shell that the command continues on the next line and the operator `>` means that the resulting output is written into the named file.

ment. We distinguish different *partof* relations to represent various levels in a system's hierarchical decomposition. Holt does not define transitive reduction, which is a useful operator for improving the presentation of graphs with many (directed) edges.

The need for executing relational formulas (see Appendix B) is also recognized by Holt. He calls his relational calculator *grok*.

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 *InfoPacks*. 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 *InfoPacks* are expressed in a simple notation, namely the RPA-file formats introduced in Section 3.6.3. *InfoPacks* 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 InfoPacks that are related to the module and code view of

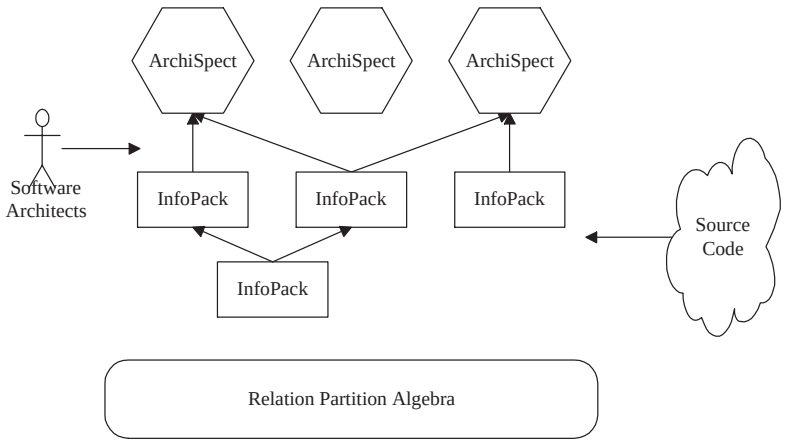


Figure 4.1: Software Architecture Reconstruction Method

architecture. The process of reconstructing the software architecture at a described level is called *reverse architecting* [Kri97].

Figure 4.2 shows the InfoPacks (rectangles) and ArchiSpects (hexagons) initially required to describe a software architecture. Solid lines between InfoPacks and ArchiSpects (InfoPack) indicate that the output of the InfoPack 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, InfoPacks 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 InfoPacks *Files*, *Import*, *Part-Of* and *Depend* of the code view, the ArchiSpects *Source Code Organisation* and *Build Process* of the code view and the ArchiSpects *Software Concepts Model*, *Component Dependency* and *Using and Used Interfaces* of the module view. The InfoPacks and ArchiSpects will be discussed according to the fixed scheme (context, description, example, method, discussion) introduced in Section 2.5.2. The InfoPacks 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

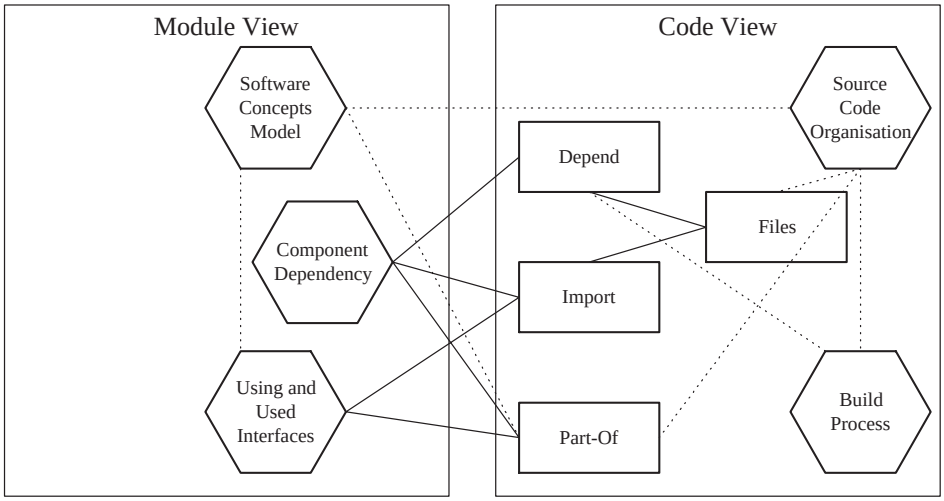


Figure 4.2: Overview of Described Architecture

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.

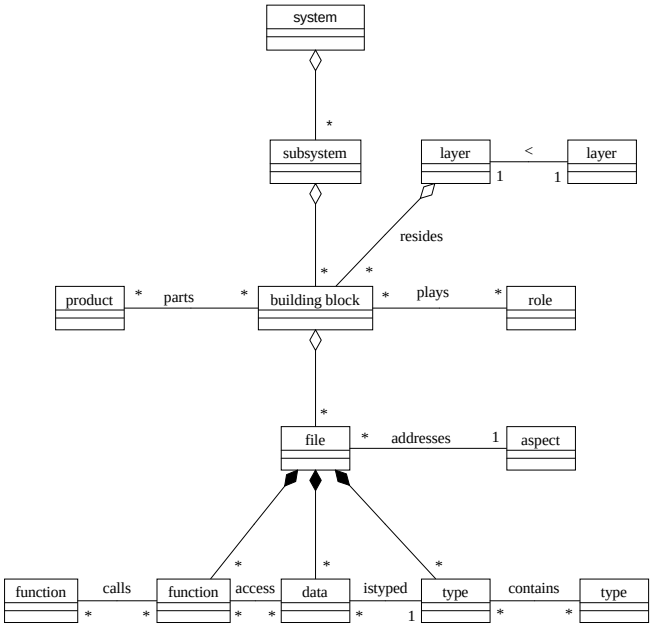


Figure 4.3: Software Concepts Model of *Tele*

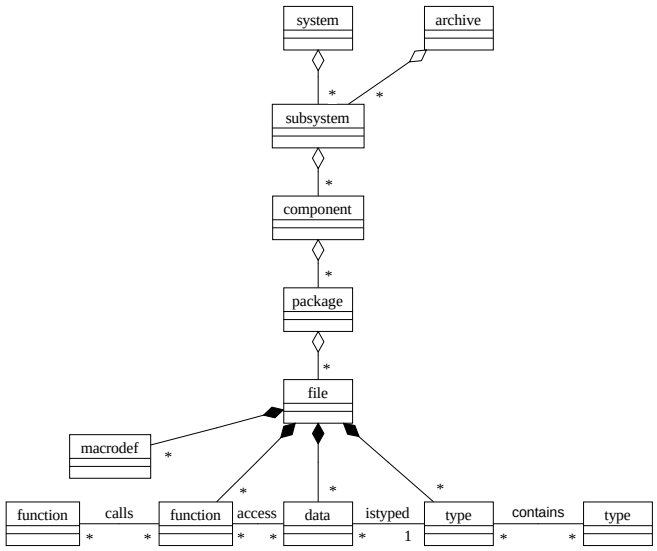


Figure 4.4: Software Concepts Model of *Med*

4.2.4 Method

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,

etc.).

Source code is one of the deliverables that is stored in the configuration management system; see also the *Build Process* ArchiSpect. 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 ArchiSpect 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

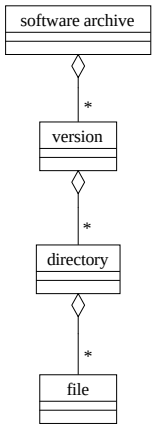


Figure 4.5: Source Code Organisation of *Med*

Software Concepts Model	Source Code Organisation
system	version
component	directory
package + file	file

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

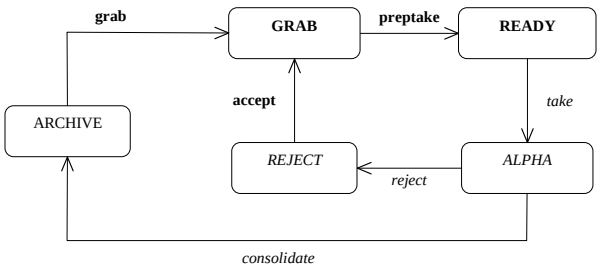


Figure 4.6: Development States and Transitions of *Med* Files

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.

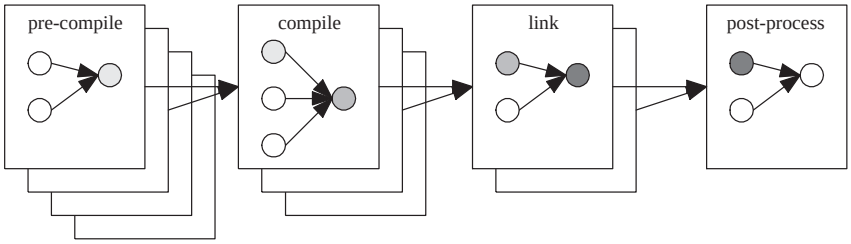


Figure 4.7: Build Activities

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*.

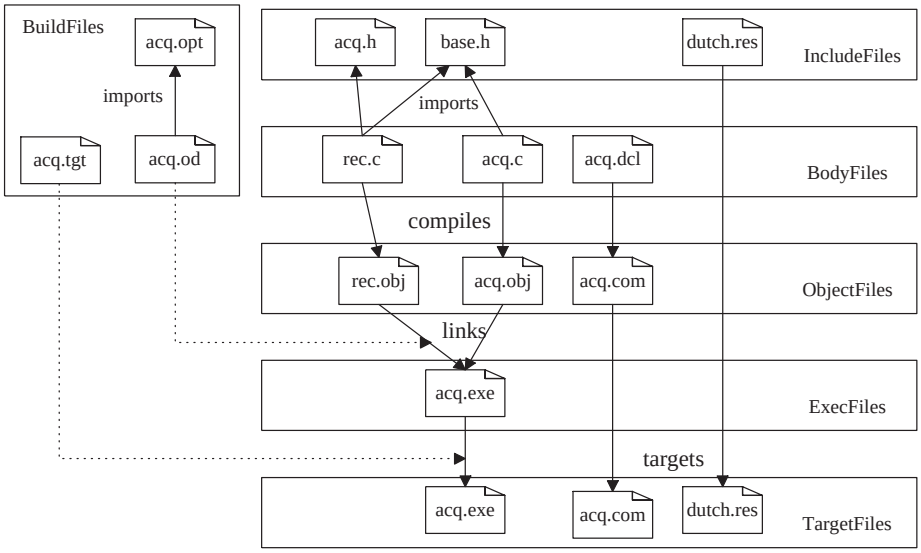


Figure 4.8: Build Process *Med*

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* ArchiSpect (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* ArchiSpect 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*;

- $typed_{Files,Exts}$, a relation that maps elements of *Files* onto elements of *Exts*;
- $located_{Files,PhFiles}$, 1-to-1 mapping of *Files* onto their physical locations (*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 *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 *Med* system consists of thousands of files (*Files*). Over 60 different file extensions were found in the system (*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 (*Cats*).

4.5.4 Method

The method for extracting the results of this InfoPack depends very much on the system at hand. The *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 $typed_{Files,Exts}$, which describes

Exts	*Cats*
c	C-source
h	C-source
java	Java-source
txt	Help-text
hlp	Help-text
gif	Picture
jpeg	Picture
bmp	Picture

Table 4.2: $typed_{Exts,Cats}$

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-name extensions. For example, given that all *HeaderFiles* have the extension `.h` or `.hpp`, we can calculate as follows in RPA:

$$\begin{aligned} HeaderExts &= \{h, hpp\} \\ HeaderFiles &= dom(typed_{Files,Exts} \upharpoonright_{ran} HeaderExts) \end{aligned}$$

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 ($typed_{Files,Cats}$) that describes this relation can be calculated as follows in RPA:

$$typed_{Files,Cats} \quad = \quad typed_{Exts,Cats} \circ typed_{Files,Exts}$$

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:

$$ran(typed_{Files,Exts}) \quad \subseteq \quad dom(typed_{Exts,Cats})$$

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) ArchiSpects 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 $\langle FileX, FileY \rangle$, 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"
```

Files	*Files*
ajax.c	champion.h
ajax.c	string.h
ajax.c	stock.h
...	...

Table 4.3: $imports_{Files,Files}$

In this example there are three `#include` statements, each with its own specifics. The three header files are literally included before the compiler starts compiling the `ajax.c` file. The compiler[1] searches for the included files in the file system using an *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 `#include` statement refers to `champion.h`. The compiler searches for this file, at first in the current directory, and secondly via the include-path. In the second `#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 `#include` statement, the file-name is preceded by a relative path (`../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 $partof_{Files,Units}$ 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 *Files* InfoPack;
- determining the include-path per file; the *Build Process* ArchiSpect describes where one can find this information;
- extracting the include statements per file and determining the included file (using the include-path);

[1] In fact, a pre-processor of the compiler searches the files and literally includes them.

Files	*Units*
champion.c	champion
champion.h	champion
ajax.c	ajax
ajax.h	ajax
stock.c	stock
stock.h	stock
...	...

Table 4.4: $partof_{Files,Units}$

- reflecting the information in the $imports_{Files,Files}$ relation file;
- determining the units and constructing $partof_{Files,Units}$.

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. *Windows NT*, does not distinguish cases in file names, *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 *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 *imports* relation can be extracted from source code written in other languages.

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 *comment-strip* program is presented in Section A.3). Secondly, we extract the inclusion of files (the *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 $imports_{Files,Files}$ relation.

One can also construct the $partof_{Files,Units}$ relation on the basis of file names. 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:

```
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`[2]. 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.

[2] In Java a `class` file is the compiled version of the accompanying *java* source file.

Classes+	*Classes*
MyPackage.ajax	Player
MyPackage.ajax	traffic.transport
MyPackage.ajax	car.*
MyPackage.ajax	java.awt.Button
...	...

Table 4.5: $imports_{Classes,Classes}$

Classes+	*Classes*
MyPackage.*	MyPackage.ajax
MyPackage.ajax	MyPackage.ajax
car.*	car.door
car.*	car.wheel
car.*	car.gear
...	...

Table 4.6: $defines_{Classes+,Classes}$

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 $imports_{Classes,Classes+}$ relation (an example is given in Table 4.5) is the result of the *J-imports* extraction program (presented in Section A.5). The *Classes+* refers to an extended set of classes; the wildcard notation has not yet been resolved. The *J-package* extraction program (presented in Section A.6) generates a $defines_{Classes+,Classes}$ 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 $imports$[3] relation:

$$imports_{Classes,Classes} = imports_{Classes,Classes+} \circ defines_{Classes+,Classes}$$

[3]Public classes and file names are in fact interchangeable in Java, so we can see it as an $imports_{Files,Files}$ 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:

```
#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:

```
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[5]. 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.

[5]Some languages offer more grouping possibilities Java e.g. includes the concept of packages.

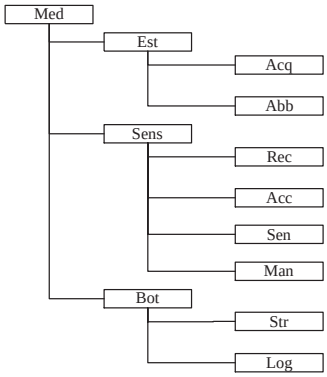


Figure 4.9: Implementation of Decomposition Levels *Med*

```
/*
 *  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* ArchiSpect. 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* ArchiSpect (Section 4.4). It uses the results of the *Files* InfoPack (Section 4.5). The result is used by the *Component Dependency* ArchiSpect (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: $depends_{Files,Files}$.

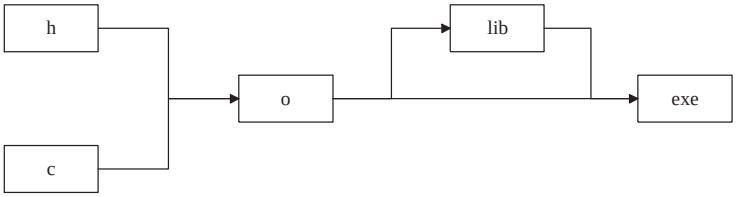


Figure 4.10: $depends_{Exts,Exts}$

This InfoPack also results in the relation $depends_{Exts,Exts}$. 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:

```
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: $depends_{Files,Files}$.

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

The relation $depends_{Exts,Exts}$ can be calculated as follows in RPA:

$$depends_{Exts,Exts} = depends_{Files,Files} \uparrow typed_{Files,Exts}$$

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 $imports_{Files,Files}$ 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 $depends_{Files,Files}$ 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*

(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[6]. 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.

[6] Some software development environments reduce the parsing time by saving precompiled 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 InfoPacks 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: $partof_{Files,L_N}$, $partof_{L_N,L_{N-1}}$, ..., $partof_{L_2,L_1}$

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.

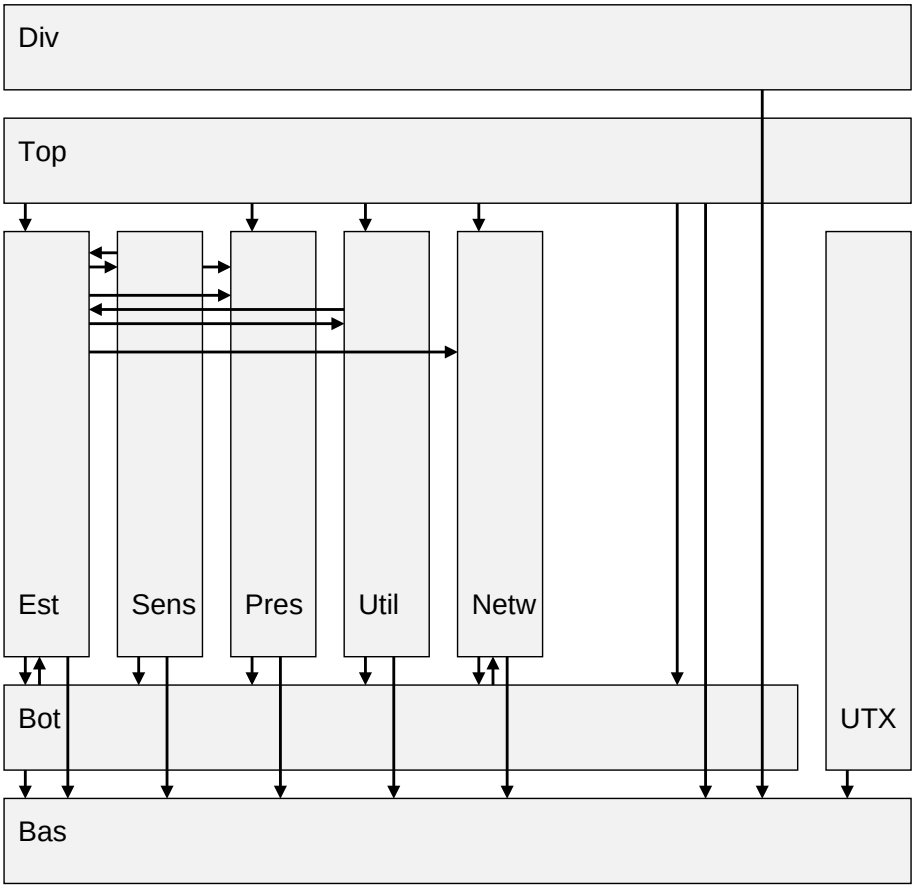


Figure 4.11: Component Dependency *Med*

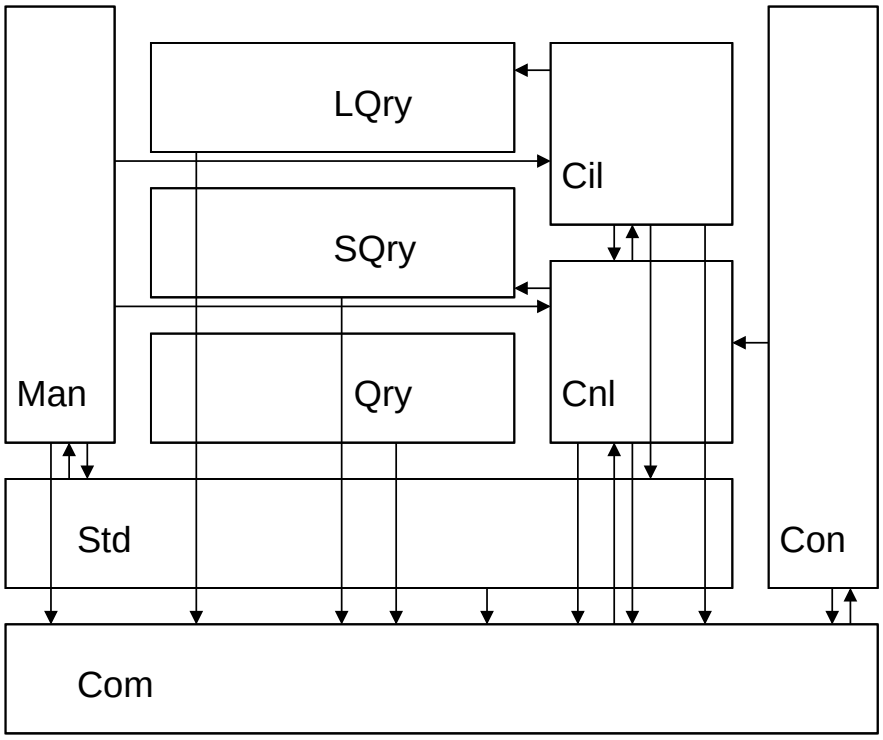


Figure 4.12: Import Dependency *Comm*

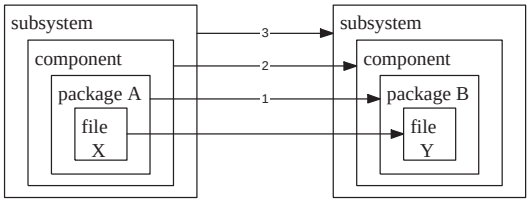


Figure 4.13: Lifting $imports_{Files,Files}$

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-Packs-Comps-Subs*:

$$
\begin{aligned}
imports_{Packs,Packs} &= imports_{Files,Files} \uparrow partof_{Files,Packs} \\
imports_{Comps,Comps} &= imports_{Packs,Packs} \uparrow partof_{Packs,Comps} \\
imports_{Subs,Subs} &= imports_{Comps,Comps} \uparrow partof_{Comps,Subs}
\end{aligned}
$$

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 $partof_{Files,Packs}$. We have lifted the *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 visualisors 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 *using* subsystems are listed in the first column and the *used* subsystems in the first row. A mark appears in the cells where a *using* subsystem imports a *used* subsystem.

	Div	Top	Est	Sens	Pres	Util	Netw	UTX	Bot	Bas
Div	▶									▶
Top		▶	▶		▶	▶	▶		▶	▶
Est			▶	▶	▶	▶	▶		▶	▶
Sens			▶	▶					▶	▶
Pres					▶				▶	▶
Util		▶				▶			▶	▶
Netw							▶		▶	▶
UTX								▶		▶
Bot			▶				▶		▶	▶
Bas										▶

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:

$$
\begin{aligned}
Comps_A &= partof_{\,Comps,Subs}.A \\
Comps_B &= partof_{\,Comps,Subs}.B \\
suspectsC_{A,B} &= (imports_{\,Comps,Comps} \restriction_{dom}\ Comps_A) \restriction_{ran}\ Comps_B
\end{aligned}
$$

explanation

The $Comps_A$ ($Comps_B$) set contains all the components that belong to subsystem A (B), i.e. the left image of $partof_{Comps,Subs}$ with respect to A (B). Taking the relation $imports_{Comps,Comps}$, we must look at the tuples that start (i.e. *domain*) from components of A and end (i.e. *range*) with components of B.

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

$$\begin{aligned} Packs_A &= partof_{Packs,Comps} \upharpoonright_{ran} Comps_A \\ Packs_B &= partof_{Packs,Comps} \upharpoonright_{ran} Comps_B \\ suspectsP_{A,B} &= (imports_{Packs,Packs} \upharpoonright_{dom} Packs_A) \upharpoonright_{ran} Packs_B \end{aligned}$$

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

Reflexion Models

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

```
[ file=*string*\.[ch]   mapTo=StringHandling ]
[ file=tcpip/ip*\.[ch]  mapTo=IPServices ]
```

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

$$mappedSourceModel = \{ \langle\langle h_1, h_2\rangle, \langle s_1, s_2\rangle\rangle |$$

$$\langle s_1, s_2 \rangle \in SourceModel \wedge mapping(h_1, s_1) \wedge mapping(h_2, s_2) \}$$

The domain of the *mappedSourceModel* describes the use relation at a high level. This is similar to the result obtained on lifting an $imports_{Files,Files}$ relation to a subsystem level.

4.10 ArchiSpect: Using and Used Interfaces

4.10.1 Context

The *Using and Used Interfaces* ArchiSpect belongs to the module view. The results of the *Import* (Section 4.6) and *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 *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 *Files* interface of component K. We must note that

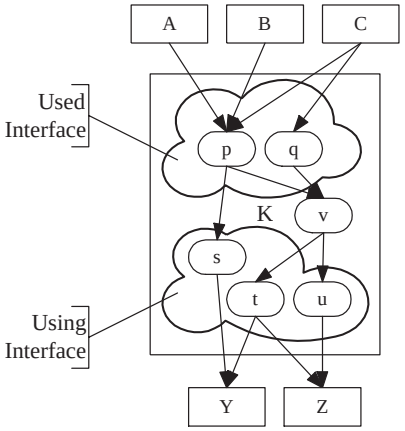


Figure 4.14: Using and Used interfaces

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

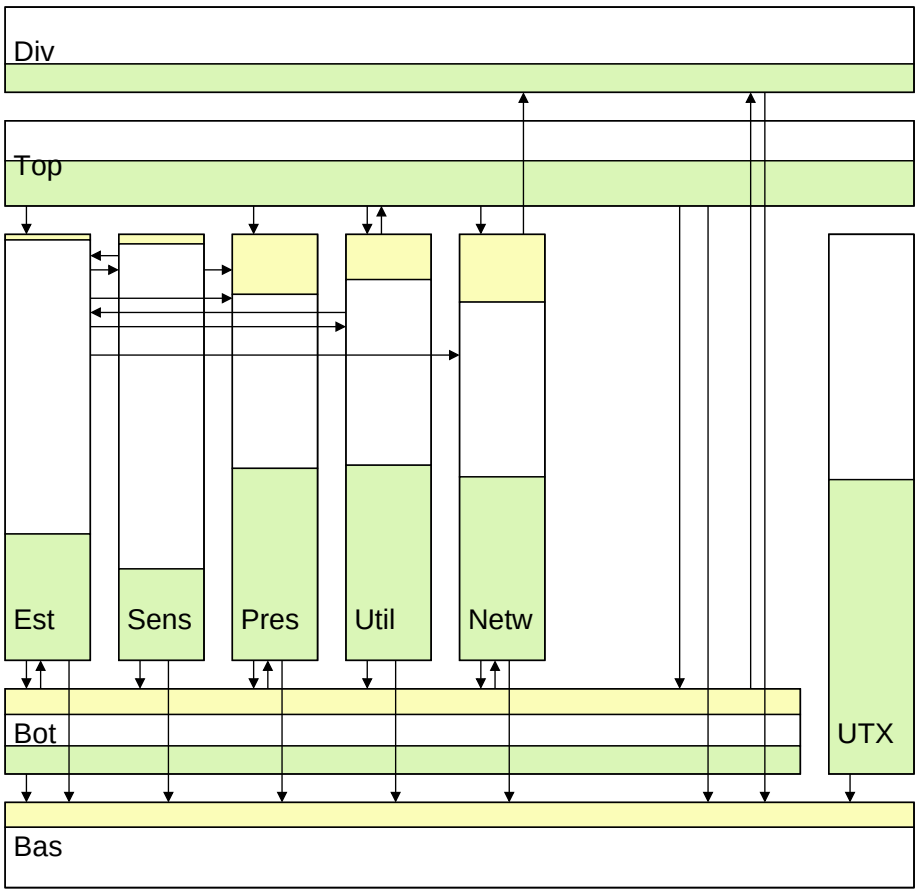


Figure 4.15: Using and Used Interfaces of *Med*

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-

Subsystem	# Files	# using Files	ratio using	# used Files	ratio used
Div	13	4	0.308	0	0.000
Top	151	80	0.530	0	0.000
Est	4015	1191	0.297	52	0.013
Sens	489	105	0.215	11	0.022
Pres	326	147	0.451	46	0.141
Util	411	188	0.457	43	0.105
Netw	448	193	0.431	71	0.158
UTX	50	27	0.540	0	0.000
Bot	586	193	0.329	177	0.302
Bas	802	0	0.000	234	0.292

Table 4.8: Using and Used Interfaces Ratios of *Med*

lowing relations: $imports_{Files,Files}$ and $partof_{Files,Comps}$. In general, to be able to calculate the L_X interface of L_Y we need the relations: $partof_{L_X,L_Y}$ and $imports_{L_X,L_X}$. For clarity, we will adhere to the *Files* interface of components. We will define, in RPA, the using interface step-by-step.

$$\begin{aligned} imports_{Files,Comps} &= partof_{Files,Comps} \circ imports_{Files,Files} \\ importsExt_{Files,Comps} &= imports_{Files,Comps} \setminus partof_{Files,Comps} \end{aligned}$$

explanation

We construct a relation $importsExt_{Files,Comps}$, 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: $imports_{Files,Comps}$. 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 $partof_{Files,Comps}$ relation.

The second step consists of the following RPA formulas:

$$FilesUsingExt = dom(importsExt_{Files,Comps})$$

$$using_{Files,Comps} = partof_{Files,Comps} \upharpoonright_{dom} FilesUsingExt$$

explanation

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

The next step consists of the following RPA formulas:

$$\begin{aligned} imports_{Comps,Files} &= imports_{Files,Files} \circ partof^{-1}{}_{Files,Comps} \\ importsExt_{Comps,Files} &= imports_{Comps,Files} \setminus partof^{-1}{}_{Files,Comps} \\ FilesUsedExt &= ran(importsExt_{Comps,Files}) \\ used_{Files,Comps} &= partof_{Files,Comps} \upharpoonright_{dom} FilesUsedExt \end{aligned}$$

explanation

The $importsExt_{Comps,Files}$ 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 $imports_{Files,Files}$, which is performed by composing it with the conversed *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 $C \in Comps$:

$$\begin{aligned} Using_C &= \frac{|using_{Files,Comps}.C|}{|partof_{Files,Comps}.C|} \\ Used_C &= \frac{|used_{Files,Comps}.C|}{|partof_{Files,Comps}.C|} \end{aligned}$$

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 *imports*. So we arrive at the following RPA formulas:

$$
\begin{aligned}
importsPlus_{Comps,Files} &= imports^{+}{}_{Files,Files} \circ partof^{-1}{}_{Files,Comps} \\
importsPlusExt_{Comps,Files} &= importsPlus_{Comps,Files} \setminus partof^{-1}{}_{Files,Comps} \\
UsedPlusExts &= ran(importsPlusExt_{Comps,Files}) \\
usedPlus_{Files,Comps} &= partof_{Files,Comps} \restriction_{dom}\ UsedPlusExts
\end{aligned}
$$

explanation

The direct or indirect use of a file is expressed by the transitive closure of the *imports* relation: $imports^{+}{}_{Files,Files}$. 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 (*Files* and *Components*) one can also select two non-consecutive levels (*Functions* and *Subsystems*). For two non-consecutive levels L_X and L_Y we have to compose a *partof* relation from existing ones:

$$
partof_{L_X,L_Y} = partof_{L_{Y-1},L_Y} \circ partof_{L_{Y-2},L_{Y-1}} \circ \ldots \circ partof_{L_X,L_{X+1}}
$$

4.10.5 Discussion

The significance of addressing ratios of using and used interfaces is also recognized in Laguë 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:

- $F(i)$: all files
- $IF(i)$: all header files
- $IM(i)$: all body files
- $D(j,i)$: all files that use files from layer j
- $S(j,i)$: all files that are used by layer j

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:

$$UR(i) = \frac{|\bigcup_{j \neq i} D(j,i)|}{|IF(i)|}$$

They calculated the using interface and the used$^+$ interface ratios in a similar manner.

Laguë 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 Laguë 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 comprehensable 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 up-date) 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

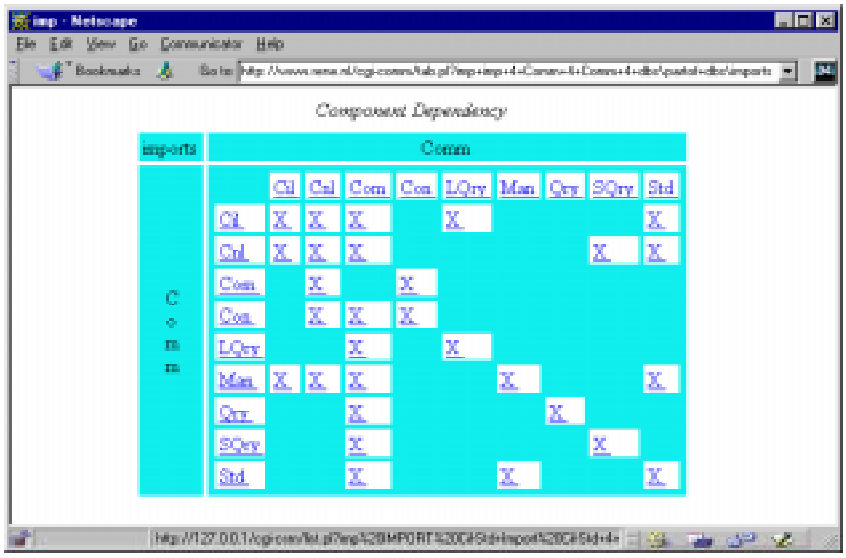


Figure 4.16: *Comm* Table Viewer

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* [FHK+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.

Chapter 5

Redefined Architecture

In the previous chapter we have discussed the described level of software architecture reconstruction. The next level of the SAR method concerns the improvement of existing software architectures: the redefined level.

5.1 Introduction

A described architecture consists of the explicit description of the software architecture of an existing system. This is for example useful for helping developers comprehend that system. The improvement of an existing software architecture is logically the next subject in our discussion. Improving an existing software architecture may be of great help in simplifying a system's maintenance and its extensions. The realization of improvements results in a redefined architecture which will be discussed in this chapter [KPS+99]. The process of improving a software architecture is called (*software*) *re-architecting*.

Changing a software architecture may affect many parts of the software. Before a change can be introduced, an architect must know exactly which parts will be affected but also the cost of implementing the change. Impact analysis is a technique for calculating the consequences of a change in advance, without realizing it in the actual source code. An architect can quietly consider all the pros and cons of a change before he or she decides to implement it.

Figure 5.1 shows the various activities involved in architecture improvement. A *software model* containing the ArchiSpects as discussed in e.g.

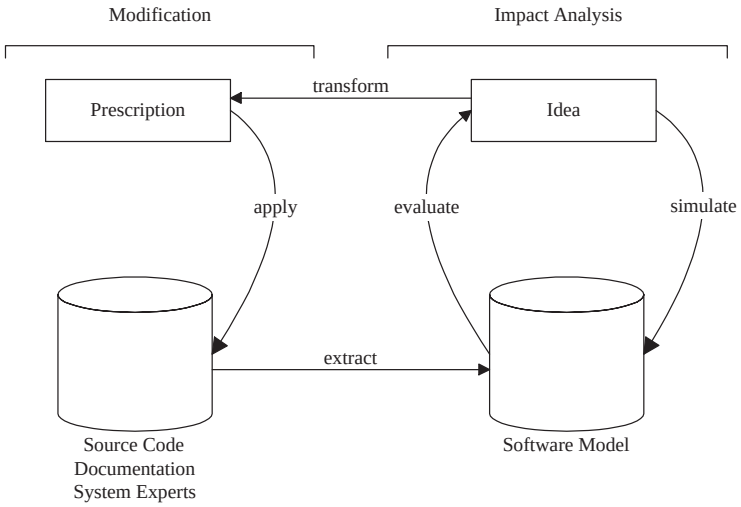


Figure 5.1: Architecture Improvement

Chapter 4 is derived from the *source code*, *documentation* and information obtained from *system experts*. This *software model* is subjected to impact analysis: an architect has an *idea* (e.g. moving a function to another module by means of changing the *partof* relation) that can be *simulated* (e.g. by means of recalculating certain ArchiSpects). This results in a modified *software model*. This new model must be *evaluated* by the architect, which may influence the original *idea*, resulting in an adapted or new *idea*. After some iterations, the refined *idea* may be accepted or discarded. Accepted *ideas* must be *transformed* into a *prescription* of modification for the implementation. This prescription can be *applied* to the *source code* and *documentation*, resulting in a new system. Note that when we extract information from the modified system, we obtain the same *software model* as we had constructed after simulating the accepted *idea*.

The advantages of this approach are clear. An architect can apply his or her ideas to a software model and figure out the consequences without involving many people and without affecting the actual system. Note that we have described a purely architecture-driven analysis, which does not take into account business, organisation or process-related issues. For example, the implementation of an improvement may be beneficial from a software-engineering point of view, but it may have disastrous consequences for the product's time-to-market. Besides the impact analysis described above, considerations of the latter kind must also be taken into account in a com-

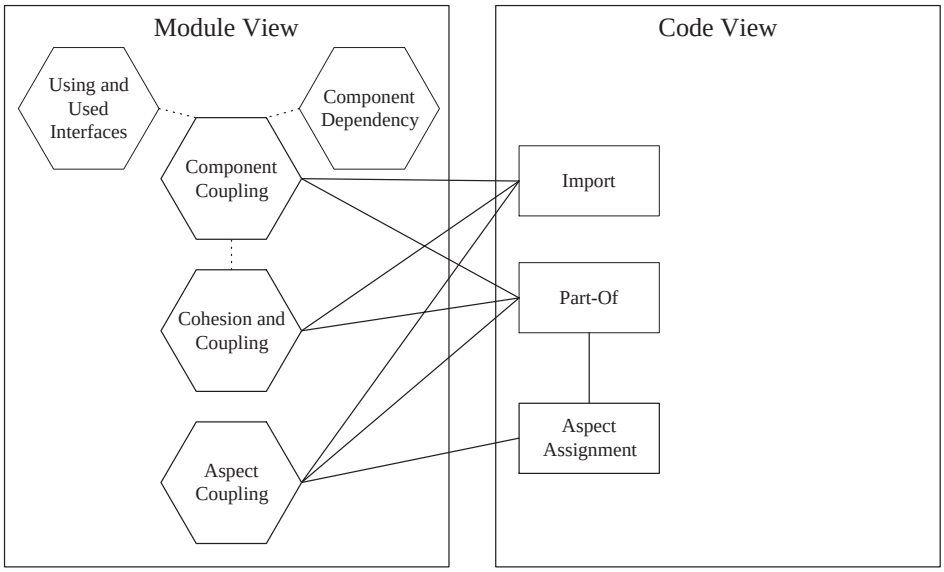


Figure 5.2: Overview of Redefined Architecture

mercial setting.

In this chapter we will discuss some ArchiSpects that are helpful in performing impact analysis aimed at improving the software architecture of an existing system; see Figure 5.2. The following ArchiSpects will be discussed in an arbitrary order:

- *Component Coupling*, the quantified dependencies between the software parts of a system;
- *Cohesion and Coupling*, metrics that describes connectivity between various software parts;
- *Aspect Coupling* (using the *Aspect Assignment* InfoPack), quantified dependencies between various parts, taking into account a certain aspect.

5.2 ArchiSpect: Component Coupling

5.2.1 Context

The *Component Coupling* ArchiSpect belongs to the module view. The results of the *Import* (Section 4.6) and *PartOf* (Section 4.7) InfoPacks are required. The *Component Dependency* (Section 4.9) and *Using and Used Interface* (Section 4.10) ArchiSpects are closely related.

5.2.2 Description

The *Component Coupling* ArchiSpect quantifies dependencies as depicted in the resulting diagram of the *Component Dependency* ArchiSpect, e.g. see Figure 4.11. Quantification is useful for example when we want to remove a dependency between X and Y. The size of a relation can be of help in estimating the amount of effort that will be involved in removing that dependency. Assume that components consist of files that import each other. Then, the number of import statements is a measure (or *weight*) of the intensity of dependency between the components. But, a relation can be quantified in different ways. We define the following weights:

- *size-oriented* weight, meaning that the number of relations at the lower level is reflected in the weight of the lifted relation at the higher level;
- *fan-in-oriented* weight, meaning that the number of entities (e.g. files) *used* by a component is reflected in the weight;
- *fan-out-oriented* weight, meaning that the relation is quantified by the number of a component's *using* entities.

To illustrate this, consider the *use* relation depicted in Figure 5.3. Component *CompHigher* uses component *CompLower* because *High1* uses *Low1*, amongst other relations. According to the above definition, we obtain the following weights:

- size-oriented: 4 (the number of dashed arrows)
- fan-in-oriented: 3 (the number of files a dashed arrow points to)
- fan-out-oriented: 2 (the number of files at which one or more dashed arrows start)

A combination of the three weights helps to re-architect a system. Assume we want to remove a dependency between two components. The size-oriented weight indicates how many import statements must be removed to

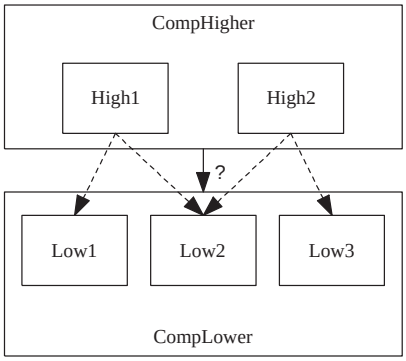


Figure 5.3: Lifting with Multiplicity: 2–3–4–case

achieve this. On the other hand, the fan-out-oriented weight indicates how many files of the including component will be affected. So, it defines the number of files that have to be changed to remove the dependency between the components.

If we want to replace a component, then the component's interface with other components must be known. The fan-in-oriented weight defines the number of files that will be used by the including component.

5.2.3 Example

The reconstruction of *Component Dependency* of two systems has been discussed in Section 4.9. In this section we present the *Component Coupling* of these systems. The resulting diagram of this ArchiSpect, created by *Teddy-PS*, contains the same arrows as the diagram of *Component Dependency*, but the arrows are of different thicknesses. The thickness of an arrow[1] is a measure of the weight of the relation. For the corresponding relation, a tuple with a large (small) weight is represented by a thick (thin) arrow.

The cells in the table representation of *Component Coupling* contain the three different weights of the various tuples, i.e. the fan-in-oriented, size-oriented and fan-out-oriented weights.

[1] We have used a logarithmic function to map the weight onto an arrow width of a few points.

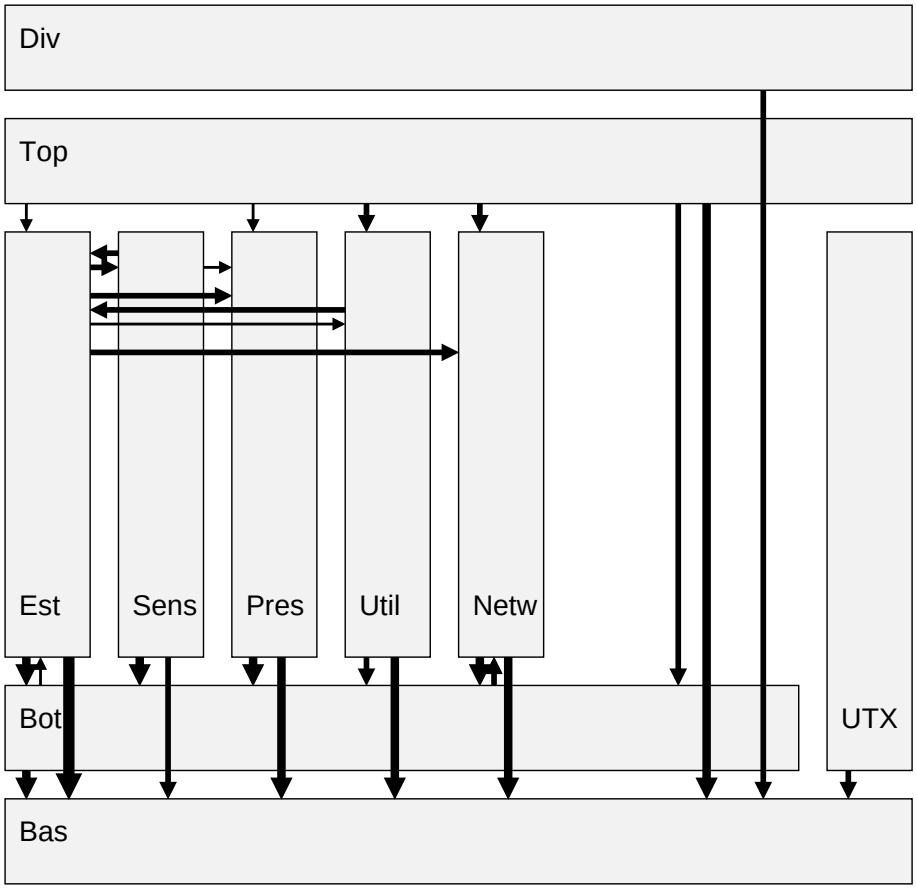


Figure 5.4: Component Coupling *Med*

Med

The (size-oriented) *Component Coupling* of *Med* is presented in Figure 5.4. The corresponding diagram of *Component Dependency* is depicted in Figure 4.11.

A table representation of *Med* is given in Table 5.1 (with its related Table 4.7). Each affected cell contains three figures, i.e. the fan-in-oriented, size-oriented and fan-out-oriented weights. Note that the table representation explicitly shows the identity relation (if applicable) in contrast with the diagram representation. Furthermore, all the variants of quantification

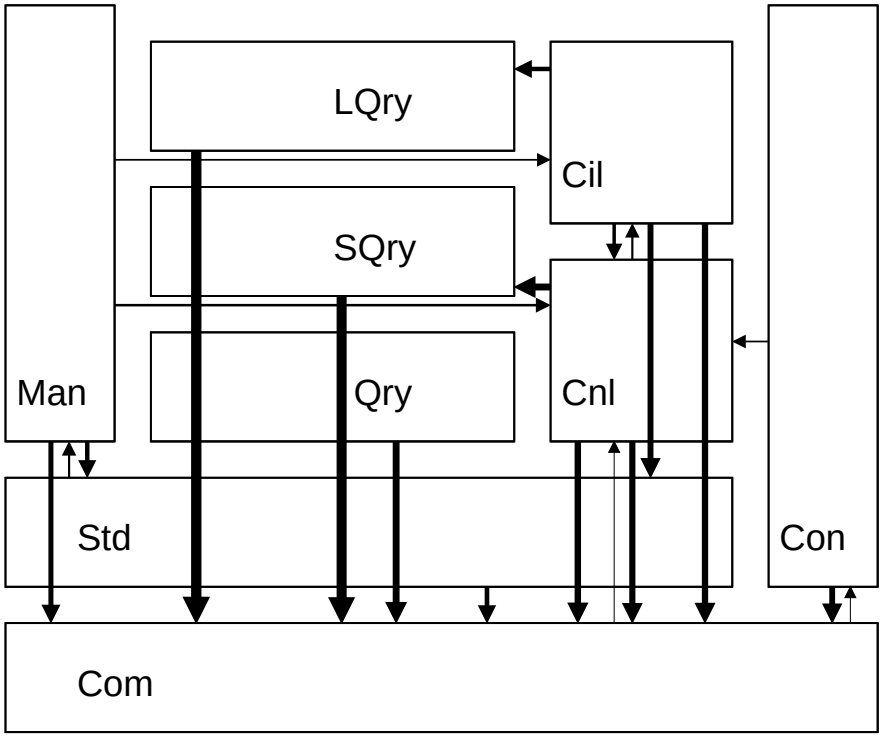


Figure 5.5: Component Coupling *Comm*

are presented in a single table.

Comm

The *Component Dependency* of the *Comm* system is depicted in Figure 4.12. Figure 5.5 shows the (size-oriented) *Component Coupling* of this system.

5.2.4 Method

The steps to be executed are similar to those of the *Component Dependency* method (described in Section 4.9). The required input consists of[2]:

[2]For clarity, we will still use the *Files* and *Comps* decomposition levels in the description, but any other pair of levels could be used.

fan-in size fan-out	Div	Top	Est	Sens	Pres	Util	Netw	UTX	Bot	Bas
Div	9 9 2									34 39 4
Top		66 151 75	11 11 9		2 2 2	41 52 16	18 25 16		59 204 52	119 719 80
Est			1301 8220 1170	11 41 34	39 75 43	4 6 6	23 55 29		117 1623 522	186 9324 1191
Sens			32 144 27	117 333 98	4 4 2				11 52 23	124 968 105
Pres					151 611 147				57 257 96	119 1483 147
Util			20 75 41			193 899 188			43 105 41	144 1773 188
Netw							236 1028 193		108 901 156	149 1957 193
UTX								23 55 27		32 197 27
Bot			2 4 4				46 98 27		227 860 193	147 1626 193
Bas										393 3725 338

Table 5.1: Component Coupling (fan-in, size, fan-out) of *Med*

- $imports_{\mathbf{Files},\mathbf{Files}}$, a multi-relation that can be constructed by mapping the *imports* relation: $\lceil imports_{Files,Files} \rceil$;
- $partof_{Files,Comps}$, a part-of relation which may be the result of a composition of a chain of *partof* relations, for example (*Med*):

$$partof_{Files,Comps} = partof_{Packs,Comps} \circ partof_{Files,Packs}$$

The three variations of weight are calculated as follows (note that we introduce two new lift operator notations $\uparrow_{\triangleleft}$ and $\uparrow_{\triangleright}$):

size-oriented:

$$imports_{\mathbf{Comps},\mathbf{Comps}} = imports_{\mathbf{Files},\mathbf{Files}} \uparrow partof_{Files,Comps}$$

fan-in-oriented:

$$\begin{aligned} importsFI_{\mathbf{Comps},\mathbf{Comps}} &= imports_{\mathbf{Files},\mathbf{Files}} \uparrow_{\triangleleft} partof_{Files,Comps} \\ &= \lceil partof_{Files,Comps} \rceil \circ \\ &\quad \lceil \lfloor imports_{\mathbf{Files},\mathbf{Files}} \circ \lceil partof^{-1}_{Files,Comps} \rceil \rfloor \rceil \end{aligned}$$

fan-out-oriented:

$$\begin{aligned} importsFO_{\mathbf{Comps},\mathbf{Comps}} &= imports_{\mathbf{Files},\mathbf{Files}} \uparrow_{\triangleright} partof_{Files,Comps} \\ &= \lceil \lfloor \lceil partof_{Files,Comps} \rceil \circ imports_{\mathbf{Files},\mathbf{Files}} \rfloor \rceil \circ \\ &\quad \lceil partof^{-1}_{Files,Comps} \rceil \end{aligned}$$

explanation

In the first definition, the $imports_{\mathbf{Files},\mathbf{Files}}$ multi-relation is lifted to component level. The lift operator for multi-relations takes into account the number of dependencies at file level in constructing the dependency at component level (as defined in Section 3.5).

To explain fan-in-oriented and fan-out-oriented lifting, we must consider an alternative formula for the lifting of relations: $U \uparrow P \equiv P \circ U \circ P^{-1}$. This alternative formula also exists for multi-relations: $U \uparrow P \equiv \lceil P \rceil \circ U \circ \lceil P^{-1} \rceil$. Note that we must first map the *partof* relation (which is always a binary relation) onto a multi-relation. Furthermore, $\lceil P^{-1} \rceil \equiv \lceil P \rceil^{-1}$.

Figure 5.6 shows the various steps of fan-in-oriented lifting (this figure is an alternative to the view presented in Figure 5.3). In the last part

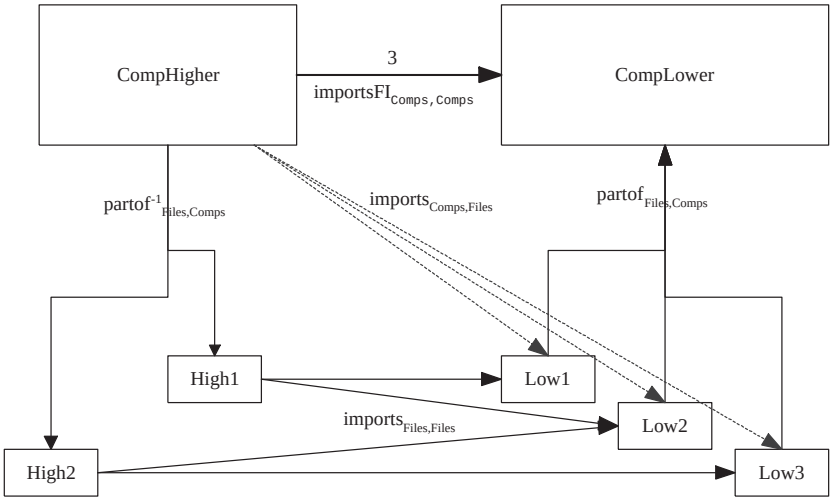


Figure 5.6: Fan-in-oriented lifting

of the given formula (i.e. the second composition), part of the lift operation is performed ("lifting in its domain"). This intermediate result describes a relation from *Comps* to *Files* (represented in Figure 5.6 as $imports_{Comps,Files}$). We are interested in the files imported by a component; we are not interested in the number of times these files are imported. So the intermediate result is normalised, i.e. setting its weight is set to 1 by mapping it first to a normal relation and then back to a multi-relation. The first part of the fan-in-oriented formula performs the rest of the lift operation ("lifting in its range"). The number of file dependencies is thus taken into account in obtaining a fan-in-oriented weight.

The fan-out-oriented lift is defined in a similar manner. We start with the first part of the formula: the intermediate result contains the normalised multi-relation $imports_{\mathbf{Files,Comps}}$. The last step is to lift the domain of this intermediate result to the component level, to obtain the relation $importsFO_{\mathbf{Comps,Comps}}$ with a fan-out-oriented weight.

5.2.5 Discussion

We have discussed an ArchiSpect which is useful in the context of re-architecting. This ArchiSpect should be part of the software model (as depicted in Figure 5.1).

Undesired component dependencies are intuitively considered less harmful when they have a small weight and more harmful when they have a large weight. An architect may use this information to detect weak spots and to analyse these spots. A large size-oriented weight between two components and a small fan-in-oriented weight may indicate that a file is located in the wrong component. To check whether this is true for the system at hand we will modify the *partof* relation (idea) and will recalculate the *Component Coupling* ArchiSpect (simulate).

There exists a relation between the various weights described above. Let's call the three kinds of weights (size-oriented, fan-in-oriented and fan-out-oriented) s, fi and fo, respectively. We will explain that the following inequation holds: $max(fi, fo) \leq s \leq fi \times fo$. Consider all the tuples in the $imports_{Files,Files}$ relation that are responsible for the dependency between two components. The number of tuples in this restricted *imports* relation (let's call it *imp*) corresponds to the size-oriented weight s. The size of the range of the *imp* relation describes the fan-in-oriented weight fi. The size of the domain of *imp* describes the fan-out-oriented weight fo. The largest possible *imp* relation consists of the cartesian product of the domain and the range, which has a size of $fi \times fo$. The smallest possible *imp* relation contains at least the set of tuples that span the domain (which is sized fi) and the range (which is sized fo). Therefore, the *imp* relation has a minimum size being the maximum of fi and fo.

5.3 ArchiSpect: Cohesion and Coupling

5.3.1 Context

The *Cohesion and Coupling* ArchiSpect belongs to the module view. It requires the results of the *Import* (Section 4.6) and *PartOf* (Section 4.7) InfoPacks. It is related to the *Component Coupling* ArchiSpect (Section 5.2).

5.3.2 Description

The complexity of a system highly affects the system's comprehensibility, maintainability and testability. Cohesion and coupling play important roles in expressing complexity. Topics relating to module cohesion and module coupling were discussed by Yourdon and Constantine in the seventies [YC79, SMC74]. These measures have also been used to develop tools

that automatically cluster parts of software, e.g. [MMR+98].

A system's *cohesion* describes the connectivity of entities within the part comprising them. It is defined as the ratio of the number of actual dependencies between these entities and the number of all possible dependencies (this agrees with the definition of *intra-connectivity* given in [MMR+98]). *Coupling* describes the connectivity between two different entities in terms of their comprising parts (in [MMR+98] this is called *inter-connectivity*). It is defined as the ratio between the number of actual dependencies of these entities and the number of all possible dependencies. A general rule of thumb (and not more than that) for achieving, amongst other things, a high degree of comprehensibility, is that a system should minimize coupling in favour of maximising cohesion.

Consider the system depicted on the left side of Figure 5.7. The components *CompLeft*, *CompRight* and *CompLow* contain some files that import each other (dotted arrows). The components *CompLeft* and *CompRight* import each other (solid arrow) by means of file Y. By moving this file Y from *CompRight* to *CompLeft*, we obtain the situation depicted on the right side of this figure. As we can see in the diagram, the degree of coupling between the components has decreased and the degree of cohesion between the files vof *CompLeft* has increased. Without having any knowledge of the system's semantics we may conclude that the structure has been improved and that the new system is easier to understand. We must note that a good software architecture cannot be created simply by optimizing the cohesion and coupling quality measures; many other aspects also play a role (e.g. decomposing a system into parts that semantically belong together).

5.3.3 Example

Comm

The *Cohesion and Coupling* of *Comm* at subsystem level are presented in Table 5.2 (– means that there is no cohesion/coupling). In this example we have chosen files as the constituents of a subsystem (we can take other entity levels, too, e.g. modules). We conclude that for all subsystems the degree of cohesion is low. Files are small units with respect to subsystems and therefore we may expect a low degree of cohesion. We may even state that a high degree of cohesion would be suspect in the case of this system. For the same reason the coupling figures are also low. This discussion shows that a proper understanding of the system is required to be able to draw

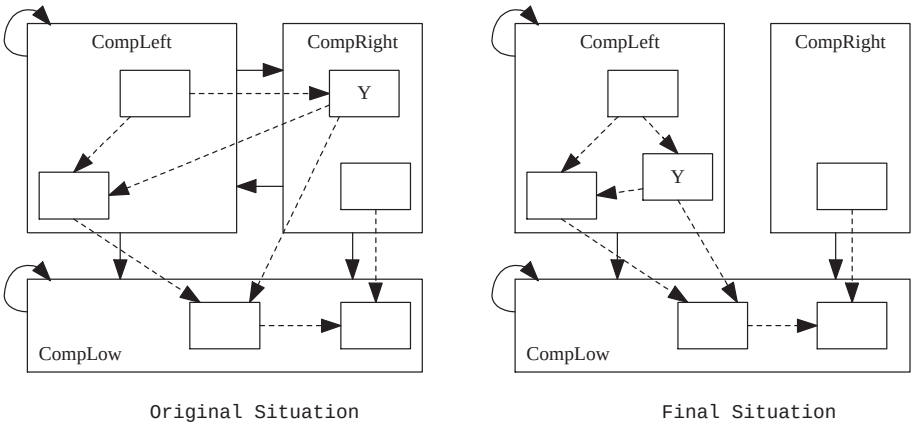


Figure 5.7: Re-clustering

	Coupling								
	Man	LQry	SQry	Qry	Cil	Cnl	Con	Std	Com
Man		-	-	-	0.058	0.205	-	10.0	36.9
LQry	-		-	-	0.045	-	-	-	147.6
SQry	-	-		-	-	0.509	-	-	138.0
Qry	-	-	-		-	-	-	-	145.2
Cil	0.058	0.045	-	-		0.087	-	3.28	13.5
Cnl	0.205	-	0.509	-	0.087		0.082	9.8	26.5
Con	-	-	-	-	-	0.082		-	42.2
Std	10.0	-	-	-	3.28	9.80	-		118.7
Com	36.9	147.6	138.0	145.2	13.5	26.5	42.2	118.7	
Coh	30.7	2.04	2.15	8.90	5.06	11.6	29.4	66.4	0.000

Table 5.2: *Cohesion and Coupling* ($\times 10^{-3}$) of Components of *Comm*

proper conclusions from metrics.

5.3.4 Method

In this section we will discuss the calculation of *Cohesion and Coupling*, given the *imports* and *partof* relations. For clarity in the discussion we will use only two decomposition levels, namely *Files* and *Components*.

The *Dominating Ratio* (DR) between two components X and Y relates the actual file imports to all possible file imports between these two components. For example, in Figure 5.8, component X imports file y_1 twice,

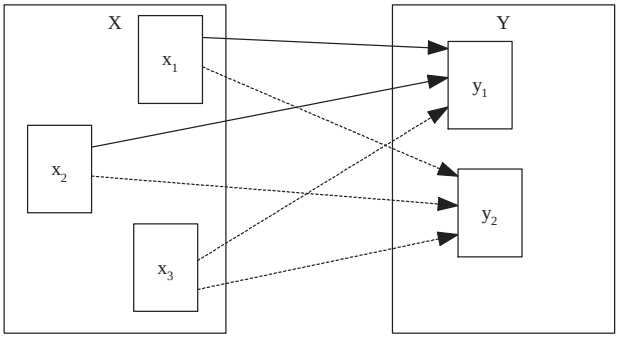


Figure 5.8: Dominating

indicated by the solid arrows. The dashed and solid arrows (6 in total) indicate all the possible imports between the files of component X and those of component Y. Therefore, in this example, the dominating ratio is $DR_{X,Y} = \frac{2}{6} = 0.333$.

We define the *Dominating Ratio* of component X with respect to component Y, denoted as $DR_{X,Y}$, as follows:

$$
\begin{aligned}
Files &= dom(partof_{Files,Comps}) \\
imports_{\mathbf{Comps},\mathbf{Comps}} &= \lceil imports_{Files,Files} \rceil \uparrow partof_{Files,Comps} \\
impAll_{\mathbf{Comps},\mathbf{Comps}} &= \lceil Files \times Files \rceil \uparrow partof_{Files,Comps} \\
DR_{X,Y} &= \frac{\| imports_{\mathbf{Comps},\mathbf{Comps}} \upharpoonright_{dom} \{X\} \upharpoonright_{ran} \{Y\} \|}{\| impAll_{\mathbf{Comps},\mathbf{Comps}} \upharpoonright_{dom} \{X\} \upharpoonright_{ran} \{Y\} \|}
\end{aligned}
$$

explanation

The size-oriented weight of the multi-relation $imports_{\mathbf{Comps},\mathbf{Comps}}$ refers to the number of file import statements in the code. The multi-relation *impAll* describes all the possible imports between components. The *imports* multi-relation is restricted in its domain with X and it is restricted in its range with Y, resulting in a multi-relation $\{\langle X, Y, w\rangle\}$. The weight w refers to the number of file imports between components X and Y. The size of this singleton multi-relation is equal to w. Analogously, the number of possible imports is calculated by starting with the multi-relation *impAll*. We obtain the dominating ratio $DR_{X,Y}$ by dividing both sizes of doubly restricted relations.

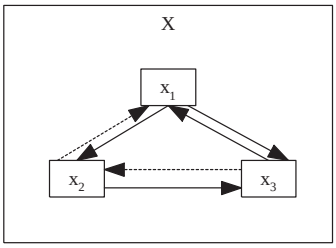


Figure 5.9: Cohesion

cohesion

The cohesion of a component indicates the degree of connectivity between its comprising files. One way of interpreting connectivity is to look at the *imports* relation. Cohesion is defined as follows:

$$\begin{aligned} Cohesion_X &= \frac{\|imports_{\mathbf{Comps},\mathbf{Comps}} \upharpoonright_{car} \{X\}\|}{\|impAll_{\mathbf{Comps},\mathbf{Comps}} \upharpoonright_{car} \{X\}\|} \\ &= \frac{\|imports_{\mathbf{Comps},\mathbf{Comps}} \upharpoonright_{dom} \{X\} \upharpoonright_{ran} \{X\}\|}{\|impAll_{\mathbf{Comps},\mathbf{Comps}} \upharpoonright_{dom} \{X\} \upharpoonright_{ran} \{X\}\|} \\ &= DR_{X,X} \end{aligned}$$

explanation

The numerator defines the number of imports between files within component X (solid arrows in Figure 5.9). In fact, the restriction results in a singleton relation $\{\langle X, X, w\rangle\}$, where w indicates the number of actually imported files. The denominator contains the number of possible imports inside X, as we have seen above (solid and dashed arrows in Figure 5.9). This corresponds to the dominating ratio of X with respect to X.

The cohesion of component X, as illustrated in Figure 5.9, is $\frac{4}{6} = 0.667$.

coupling

Coupling is a measure of the degree of connectivity between two components. Given the *imports* relation as a connectivity artefact we define coupling as:

$$
\begin{aligned}
imp_{\mathbf{X},\mathbf{Y}} &= imports_{\mathbf{Comps},\mathbf{Comps}} \upharpoonright_{dom} \{X\} \upharpoonright_{ran} \{Y\} \\
imp_{\mathbf{Y},\mathbf{X}} &= imports_{\mathbf{Comps},\mathbf{Comps}} \upharpoonright_{dom} \{Y\} \upharpoonright_{ran} \{X\} \\
impAll_{\mathbf{X},\mathbf{Y}} &= impAll_{\mathbf{Comps},\mathbf{Comps}} \upharpoonright_{dom} \{X\} \upharpoonright_{ran} \{Y\} \\
impAll_{\mathbf{Y},\mathbf{X}} &= impAll_{\mathbf{Comps},\mathbf{Comps}} \upharpoonright_{dom} \{Y\} \upharpoonright_{ran} \{X\} \\
Coupling_{X,Y} &= \frac{\|imp_{\mathbf{X},\mathbf{Y}} \cup imp_{\mathbf{Y},\mathbf{X}}\|}{\|impAll_{\mathbf{X},\mathbf{Y}} \cup impAll_{\mathbf{Y},\mathbf{X}}\|} \\
&= \frac{\|imp_{\mathbf{X},\mathbf{Y}}\| + \|imp_{\mathbf{Y},\mathbf{X}}\|}{\|impAll_{\mathbf{X},\mathbf{Y}}\| + \|impAll_{\mathbf{Y},\mathbf{X}}\|} \\
&= \frac{\|imp_{\mathbf{X},\mathbf{Y}}\| + \|imp_{\mathbf{Y},\mathbf{X}}\|}{2 \times \|impAll_{\mathbf{X},\mathbf{Y}}\|} \\
&= \frac{1}{2} \times \left(\frac{\|imp_{\mathbf{X},\mathbf{Y}}\|}{\|impAll_{\mathbf{X},\mathbf{Y}}\|} + \frac{\|imp_{\mathbf{Y},\mathbf{X}}\|}{\|impAll_{\mathbf{Y},\mathbf{X}}\|}\right) \\
&= \frac{DR_{X,Y} + DR_{Y,X}}{2}
\end{aligned}
$$

explanation

The numerator of $Coupling_{X,Y}$ counts the number of file import statements of component X importing files from component Y and vice versa (solid arrows in Figure 5.10). As in the cohesion definition, the denominator contains the number of all possible imports (solid and dashed arrows in Figure 5.10). The ratio is a measure of the degree of coupling, which can be rewritten in terms of dominating ratios. Because $X \neq Y$, and therefore $imp_{\mathbf{X},\mathbf{Y}} \cap imp_{\mathbf{Y},\mathbf{X}} = \emptyset$, we can rewrite the numerator by adding the sizes of the two multi-relations. Analogously, the denominator can be written as the sum of two multi-relations. Furthermore, these latter two multi-relations are both of the same size (due to the construction of $impAll$ it holds that: $impAll \equiv impAll^{-1}$).

We note that the degree of coupling between X and Y is equal to the degree of coupling between Y and X (due to the associative $+$ operator).

The degree of coupling between components X and Y, illustrated in Figure 5.10, is $\frac{1}{2} \times (\frac{2}{6} + \frac{3}{6}) = 0.417$.

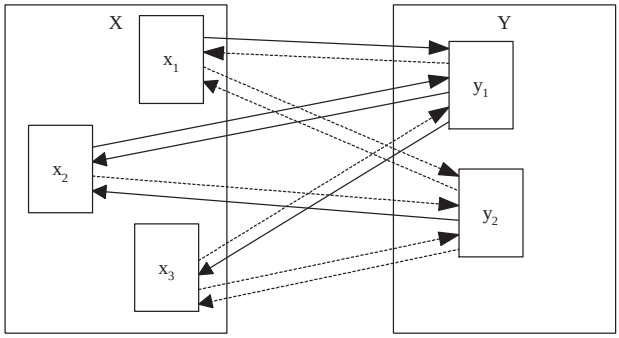


Figure 5.10: Coupling

5.3.5 Discussion

Given a set of entities and relations between them (e.g. the *imports* relation between *Files*), one can cluster these entities by applying the heuristic "maximise cohesion and minimise coupling". Note that it makes no sense to increase cohesion simply by artifically creating extra relations between entities within a cluster. Therefore we should consider the above heuristic more carefully. By creating various clusters one in fact divides the set of existing tuples of the relation, e.g. *imports*, into two parts: a set of tuples that do not cross a cluster's border and a set of tuples that do cross a cluster's border. So it is better to define the heuristic as: "maximise the number of tuples in the cohesion part and minimise it in the coupling part".

Clustering of software parts at different levels in the decomposition hierarchy is a task which can indeed not be performed automatically [Wig97, MMR+98]. Cohesion and coupling metrics can help an architect to make the right decisions about clustering (a tool for software reclustering, based upon these metrics, has been implemented by Brook [Bro99]). Note that clustering can never be driven by optimising the value of two metrics. Many factors play a role in the clustering process, but only a few can be expressed in metrics.

5.4 InfoPack: Aspect Assignment

5.4.1 Context

The *Aspect Assignment* InfoPack belongs to the code view. We may need the results of the *PartOf* InfoPack (Section 4.7). The results are used by *Aspect Coupling* (Section 5.5).

5.4.2 Description

The notion of aspects has already been discussed in Section 1.5.3. Aspects are a design concept, but they should also be reflected in the implementation in some way. This can be realized in various ways. For example, a file addresses only a single aspect of the system: the aspect to which the file belongs can be encoded in the file name.

The main result of this InfoPack consists of the $addresses_{Files,Asps}$ relation and the *Aspects* set.

5.4.3 Example

The *Tele* system explicitly engineers the notion of aspects during system development in all its phases (i.e. in the forward-architecting process). A file, having an aspect-related name, addresses exactly one aspect.

We will illustrate this with an example of a system that did not initially consider aspects.

Med

During our re-architecting activities we introduced aspects into the *Med* system. Although it was not possible to apply it precisely in its full meaning, it helped us to construct a new view on the system. We were able to identify the following aspects:

- *Clinical*: all the software that is sent to a hospital along with the medical system;
- *Test*: the software needed to test the system during its development;
- *Development*: the software that comprises all the dedicated tools that are required to develop the system (e.g. for code generation).

We refine the first aspect into:

- *Operational*: software activated by an operator in the hospital;
- *Research*: software prepared for academic hospitals for clinical research purposes;
- *Diagnostic*: software relating to all the service operations performed by a service mechanic;
- *Installation*: software required only to install the system at a (new) site.

So the *Aspects* set and the *ClinicalAspects* subset are defined as follows:

$$\begin{aligned} Aspects &= \{Operational, Research, Diagnostic, \\ &\quad Installation, Test, Development\} \\ ClinicalAspects &= \{Operational, Research, Diagnostic, Installation\} \end{aligned}$$

5.4.4 Method

The first step is to determine the various aspects of the system. This task is easy when aspects have been used already during architecture design. If not, we will have to discuss the notion of aspects with architects and designers. The second step consists of identifying these aspects in the software. Although the notion of aspects may not have been explicitly identified so far, it may be possible to determine aspects in code. For example, an aspect like *Logging* may express itself in the naming of functions (`WriteLogMessage`) and/or the naming of files (`LogUtilities`). Given such naming conventions, one can assign an aspect to most of the functions (or files). Because of the heuristic nature of extraction, the results of this extraction should be carefully checked. The functions (or files) that cannot be assigned to an aspect in this way must be assigned by hand.

With the *Med* system, we used the names of the packages to assign aspects. So we were able to extract the $addresses_{Packs,Asps}$ relation by analysing the package name. We can lower this relation to the *Files* decomposition level:

$$addresses_{Files,Asps} = addresses_{Packs,Asps} \downarrow partof_{Files,Packs}$$

5.4.5 Discussion

This InfoPack may be hard to construct in the case of systems in which aspects are not handled explicitly. To reconstruct *Aspect Assignment*, one should take the decomposition level that fits such an assignment best. The *partof* relation can be used to bring the assignment to any requested level. For example, if it is possible to reconstruct aspect assignment at *Functions* level ($addresses_{Funcs,Asps}$), one can bring it to a *Files* level (using composition). In that case, the resulting $addresses_{Files,Asps}$ relation is not necessarily functional.

If aspects appear only at statement level, it is practically impossible to obtain a useful *addresses* relation. Sometimes the notion of aspects must be relaxed somewhat to realize aspect assignment. Although this particular result does not comply precisely with the definition of aspects, it may be helpful in re-architecting a system.

5.5 ArchiSpect: Aspect Coupling

5.5.1 Context

The *Aspect Coupling* ArchiSpect belongs to the module view. The results of the *Import* (Section 4.6), *PartOf* (Section 4.7) and *Aspect Assignment* (Section 5.4) InfoPacks are required as input. This ArchiSpect is related to the *Component Coupling* ArchiSpect (Section 5.2).

5.5.2 Description

Consider a programmer who is working on message logging. He is not interested in all the code, but only in the parts concerning statements about logging. If we can offer the programmer a reduced *logging* view on the system, it will be easier for him to perform his *logging* task.

Aspects structure a system in addition to e.g. functional structuring. Both structuring means are more or less orthogonal, which helps to create two completely different views on the system. Aspect structuring plays the most important role during certain development activities, e.g. when dealing with message logging while functional structuring is important e.g. when adding a new feature to a system (e.g. *Follow Me* into a telecommunication switching system).

Design decisions relating to aspects should also be reflected in source code. For example, an aspect can be reflected as a set of files, which means that a single file belongs to exactly one aspect.

A structuring mechanism is effective only when properly applied. This will be apparent from e.g. low degree of connectivity between the parts that result from structuring. A rule of thumb for aspect connectivity is: an aspect A may only use functionality that belongs to aspect A. Other dependencies between aspects could be defined by an architect, but, for clarity, they should be minimal.

5.5.3 Example

Med

The *Aspect Coupling* of the *Test* aspect is presented in Figure 5.11 (created by the *Teddy-PS* tool). This diagram is of the same type shown for *Component Coupling*. In fact, it is a subset[3] of the diagram depicted in Figure 5.4 on page 104. The dependency between aspects is represented in Figure 5.12.

5.5.4 Method

We will start with the *imports* multi-relation at the proper decomposition level, which corresponds to the decomposition level of the domain of the *addresses* relation. For each aspect we construct a diagram or table similar to the diagram shown for *Component Coupling*. Assume we want to construct the diagram for an *Asp* aspect, then we have to restrict the *imports* multi-relation for this aspect:

$$\begin{aligned} FilesAsp &= addresses_{Files,Asps}.Asp \\ importsAsp_{\mathbf{Files},\mathbf{Files}} &= imports_{\mathbf{Files},\mathbf{Files}} \restriction_{dom} FilesAsp \end{aligned}$$

explanation

The *FilesAsp* set contains all the files assigned to the *Asp* aspect. We reduce the imports relation by looking only at the files that belong to

[3] An arrow can occur in the aspect diagram only if it occurs in the component diagram, with the same or greater weight.

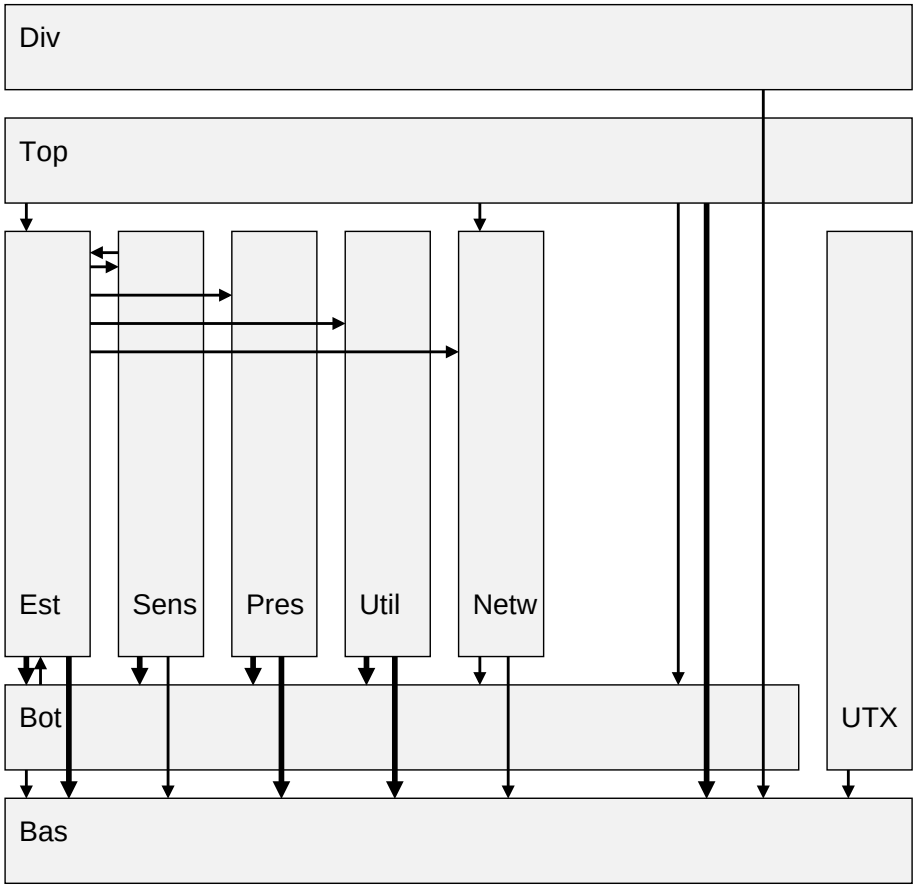


Figure 5.11: *Test* Aspect Coupling for *Med*

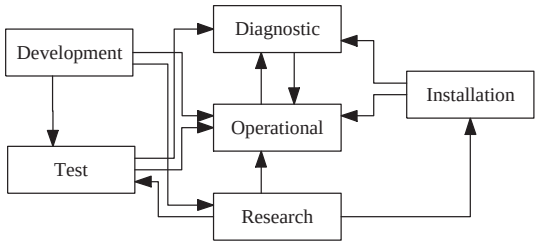


Figure 5.12: Dependencies between Aspects of *Med*

this set. This means that we have to restrict the *imports* relation in its domain using *FilesAsp*.

The dependencies between aspects yield an alternative abstract view on the system. Given the *imports* relation, we can derive the dependency between the aspects as follows:

$$\begin{aligned} depends_{\mathbf{Asps},\mathbf{Asps}} &= \lceil addresses_{Files,Asps} \rceil \circ \\ &\quad imports_{\mathbf{Files},\mathbf{Files}} \circ \\ &\quad \lceil addresses^{-1}{}_{Files,Asps} \rceil \end{aligned}$$

explanation

If the *addresses* relation defines a partition, we can lift the *imports* relation to the level of *Aspects*. But we may not assume this, so we must bring both the domain and the range of the *imports* relation to the aspect level through composition.

The last step of the method consists of presenting this information. We use the same presentation techniques as used for *Component Dependency*, e.g. by means of the *Teddy-PS* tool.

5.5.5 Discussion

The proper application of aspects results in a clear division of a system into slices (each belonging to a single aspect) that make virtually no use of each other. The *Aspect Cohesion* and *Aspect Coupling* metrics can be defined in a similar manner to the description in Section 5.3. The containment relation is defined by the $addresses_{Files,Asps}$ relation (although it is not necessarily a partition). Furthermore, the $imports_{Files,Files}$ relation represents the dependencies between files. We define the dominating ratio ($DR_{A,B}$) between aspect A and B as follows:

$$\begin{aligned} imports_{\mathbf{Asps},\mathbf{Asps}} &= \lceil addresses_{Files,Asps} \rceil \circ imports_{\mathbf{Files},\mathbf{Files}} \circ \\ &\quad \lceil addresses^{-1}{}_{Files,Asps} \rceil \\ impAll_{\mathbf{Asps},\mathbf{Asps}} &= \lceil addresses_{Files,Asps} \rceil \circ \lceil Files \times Files \rceil \circ \end{aligned}$$

$$DR_{A,B} = \frac{\lceil addresses^{-1}{}_{Files,Asps} \rceil}{\| imports_{\mathbf{Asps},\mathbf{Asps}} \restriction_{dom} \{A\} \restriction_{ran} \{B\} \|} \frac{\lceil addresses^{-1}{}_{Files,Asps} \rceil}{\| impAll_{\mathbf{Asps},\mathbf{Asps}} \restriction_{dom} \{A\} \restriction_{ran} \{B\} \|}$$

So, as with cohesion and coupling, we define *aspect cohesion* and *aspect coupling* as follows:

$$\begin{aligned} Cohesion_A &= DR_{A,A} \\ Coupling_{A,B} &= \frac{DR_{A,B} + DR_{B,A}}{2} \end{aligned}$$

5.6 Concluding Remarks

Improving a software architecture often involves a lot of questions and deducing more precise questions from the answers or defining improvements. Relation Partition Algebra offers a flexible way of asking these questions in a formal manner; the answers are obtained by executing the RPA formulas.

Impact analysis (see Figure 5.1 on page 100), or *what-if* analysis, consists of an iterative process of defining an *idea*, *simulating* it on a *software model* and *evaluating* the results. For example, an *idea* could be to move a function from one file to another. The *simulation* of this *idea* consists of changing the appropriate sets, relations and multi-relations of the *software model* (including re-calculating the derived relations to retain a consistent model). We have also presented a number of quality metrics (cohesion, coupling, aspect cohesion, aspect coupling) that can support the *evaluation* of a *software model*. But the intuition of architects also plays an important role [Cor89]. An ArchiSpect like *Component Coupling* helps an architect shape his or her intuition.

We could use our experience to develop a dedicated tool that supports the impact analysis process (Computer-Aided Impact Analysis). An *idea* can be transformed into actions that must be executed during simulation (i.e. a script for modifying the software model). The tool could be designed to keep the *software model* consistent. To support *evaluation*, such a tool should be able to present various metrics and diagrams and tables as results of various ArchiSpects.

There exist clustering algorithms that e.g. try to cluster functions into cohesive groups that are minimally coupled. In large systems, functionalities are grouped at various levels, which makes these algorithms hard to apply. In addition, factors that are hard to measure play a role with respect to deciding whether to cluster functionalities, e.g. the semantics of functions.

Accepted *ideas* must be implemented in the actual software. An *idea*, e.g. move function f from file x to file y, must be translated into a *prescription* that can be applied to the source code. Dedicated transformation tools (often based on compiler technology to a great extent) can automatically apply simple *prescriptions* to the source code. The rest must be specified in terms of change requests and must be performed manually by developers.

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.

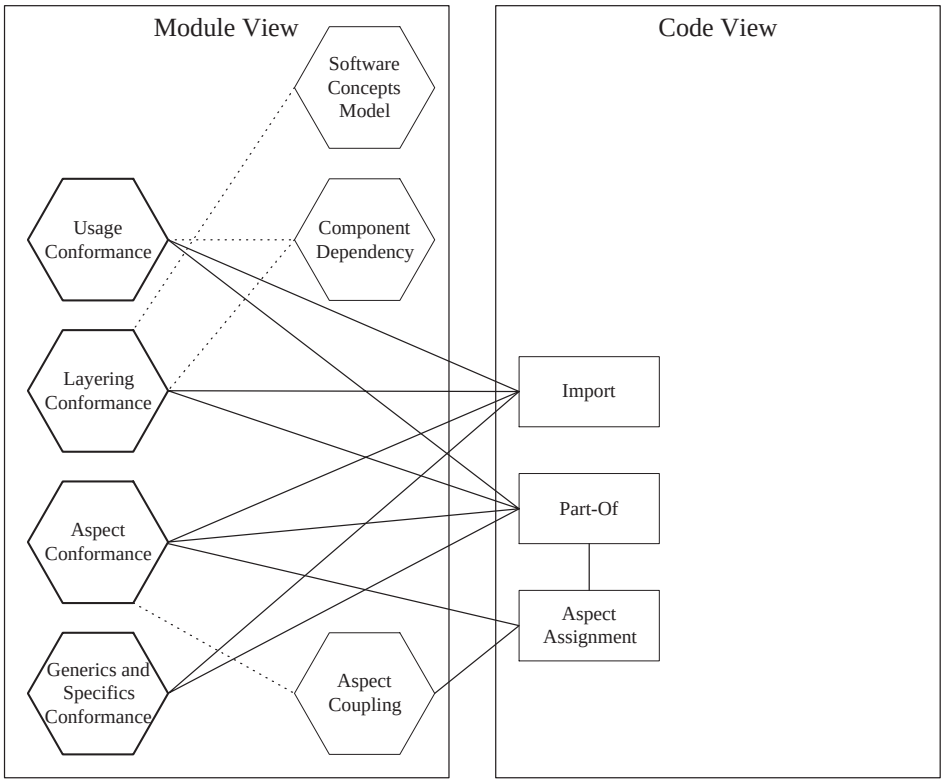


Figure 6.1: Overview of Managed Architecture

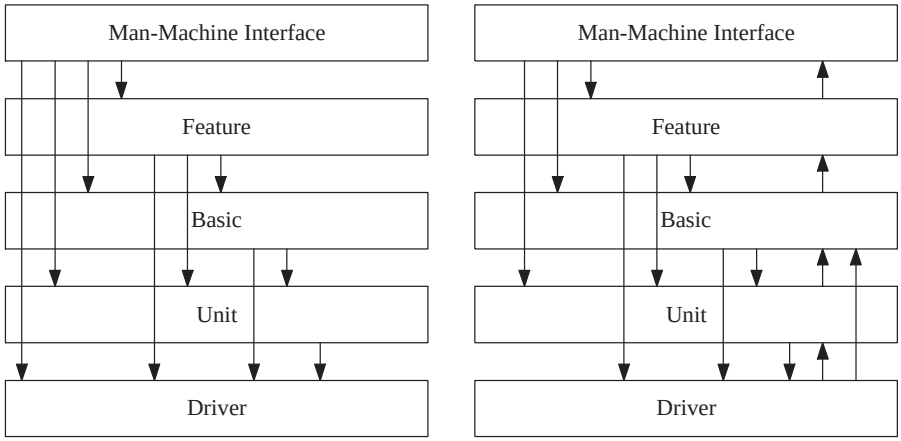


Figure 6.2: Layering Conformance of *Cons*

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)

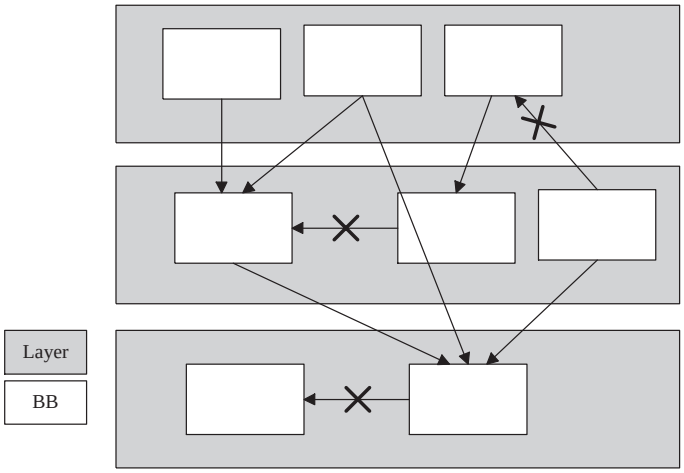


Figure 6.3: Layering Conformance of *Tele*

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 $resides_{Blocks,Layers}$ relation[1]. Note that the $resides_{Blocks,Layers}$ 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 $<_{Layers,Layers}$.

Furthermore, we need the following relations:

- $imports_{Files,Files}$ (from the *Import* InfoPack)
- $partof_{Files,Blocks}$ (from the *PartOf* InfoPack)

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

$$imports_{Blocks,Blocks} = imports_{Files,Files} \uparrow partof_{Files,Blocks}$$
$$mayuse_{Blocks,Blocks} = <^{+}_{Layers,Layers} \downarrow resides_{Blocks,Layers}$$

[1] The *PartOf* InfoPack can be extended to extract this information.

rule:

$$imports_{Blocks,Blocks} \subseteq mayuse_{Blocks,Block}$$

violations:

$$\begin{aligned} v\_imports_{Blocks,Blocks} &= imports_{Blocks,Blocks} \setminus mayuse_{Blocks,Blocks} \\ v\_Blocks &= dom(v\_imports_{Blocks,Blocks}) \\ v\_imports_{Files,Files} &= imports_{Files,Files} \setminus \\ &\quad (mayuse_{Blocks,Blocks} \downarrow partof_{Files,Blocks}) \end{aligned}$$

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 $v\_imports_{\mathbf{Blocks},\mathbf{Blocks}}$ 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 *Loader* Building Block) may use functionality from higher layers. This extra allowed usage can be incorporated in the *mayuse* multi-relation ($\{\ldots, \langle Layer_1, Layer_2, 1\rangle, \ldots\}$).

6.3 ArchiSpect: Usage Conformance

6.3.1 Context

The *Usage Conformance* ArchiSpect belongs to the module view. It uses the results of the *Import* (Section 4.6) and *PartOf* (Section 4.7) InfoPacks. It is related to *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 *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 *Layering Conformance*. We describe precisely which components may use each other, while *Layering Con-*

Comp	*Comp*
Acq	Abb
Acq	Rec
Acq	Log
Rec	Sen
Sen	Log
Man	Str
...	...

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 $mayuse_{Comps,Comps}$ relation. The architectural rule that must hold is defined as follows in RPA (also discussed in [FKO98]):

rule:

$$imports_{Comps,Comps} \subseteq mayuse_{Comps,Comps}$$

violations:

$$
\begin{aligned}
v\_imports_{\,Comps,Comps} &= imports_{\,Comps,Comps} \setminus mayuse_{\,Comps,Comps} \\
v\_Comps &= dom(v\_imports_{\,Comps,Comps}) \\
v\_imports_{\,Files,Files} &= imports_{\,Files,Files} \setminus \\
&\quad (mayuse_{\,Comps,Comps} \downarrow partof_{\,Files,Comps})
\end{aligned}
$$

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 *Med* 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 aboveand 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:

$$\begin{aligned} convergences &= U_{SM} \cap U_{HLM} \\ divergences &= U_{SM} \setminus U_{HLM} \\ absences &= U_{HLM} \setminus U_{SM} \end{aligned}$$

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).

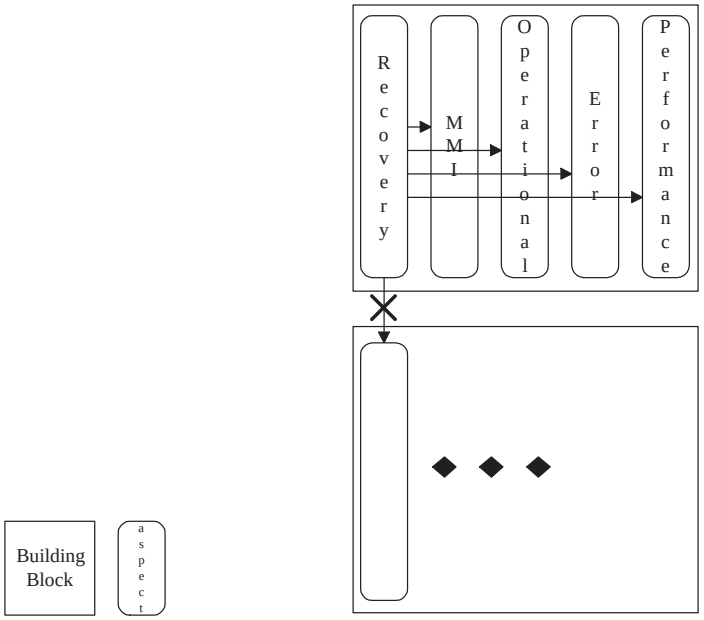


Figure 6.4: Aspect Conformance

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 therfore only access data and functionality of its own, as illustrated in Figure 6.4.

6.4.4 Method

An architect must define the $maydepend_{Asps,Asps}$ relation, which defines the allowed usage between aspects. Furthermore, we require the relations

$imports_{Files,Files}$ (*Import* InfoPack), $partof_{Files,Comps}$ (*PartOf* InfoPack) and $addresses_{Files,Asps}$ (*Aspect Assignment* InfoPack). We define the following architectural rule (also discussed in [FKO98]):

$$depends_{Asps,Asps} = addresses_{Files,Asps} \circ imports_{Files,Files} \circ addresses^{-1}{}_{Files,Asps}$$

rule:

$$depends_{Asps,Asps} \subseteq maydepend_{Asps,Asps}$$

violations:

$$v\_depends_{Asps,Asps} = depends_{Asps,Asps} \setminus maydepend_{Asps,Asps}$$

$$v\_imports_{Files,Files} = imports_{Files,Files} \setminus (addresses^{-1}{}_{Files,Asps} \circ maydepend_{Asps,Asps} \circ addresses_{Files,Asps})$$

explanation

The $depends_{Asps,Asps}$ relation is created by lifting the domain and range of the *imports* relation (as already discussed in Section 5.5). The actual dependencies between aspects ($depends_{Asps,Asps}$) must be a subset of the allowed dependencies ($maydepend_{Asps,Asps}$) 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 *Files* level.

6.4.5 Discussion

The following formulas are defined for the *Tele* system, concerning the *recovery* aspect (given a *calls* relation and *addresses* relation at *Functions* level):

$$Funcs_{recovery} = addresses_{Funcs,Asps}.recovery$$

$$
\begin{aligned}
recovers_{Funcs,Funcs} &= calls_{Funcs,Funcs} \restriction_{dom} Funcs_{recovery} \\
recovers_{Blocks,Blocks} &= recovers_{Funcs,Funcs} \uparrow \\
&\quad partof_{Funcs,Files} \uparrow partof_{Files,Blocks}
\end{aligned}
$$

rule:

$$
recovers_{Blocks,Blocks} \setminus Id_{Blocks} \subseteq \emptyset
$$

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

rule:

$$
(recovers_{Blocks,Blocks} \setminus_{dom} \{SysRecovery\}) \setminus Id_{Blocks} \subseteq \emptyset
$$

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

6.5 ArchiSpect: Generics and Specifics Conformance

6.5.1 Context

The *Generics and Specifics Conformance* ArchiSpect belongs to the module view. It requires the results of *Import* (Section 4.6) and *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*.

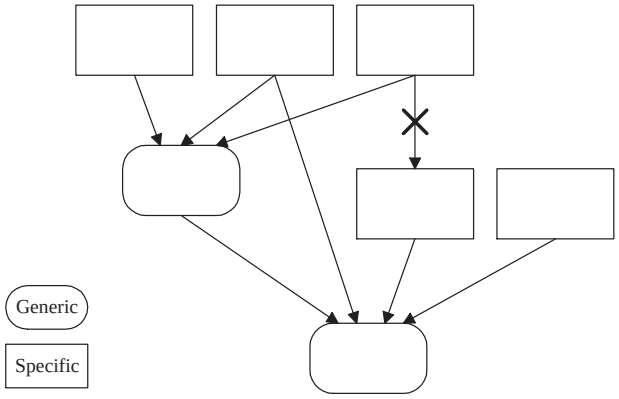


Figure 6.5: Generic and Specific Components

6.5.4 Method

The input[3] for this ArchiSpect consists of the set of generic components (*Generics*) and the set of specific components (*Specifics*). Furthermore, we require the relations: $imports_{Files,Files}$ and $partof_{Files,Comps}$. Note that the *Generics* and *Specifics* describe a partition of *Comps*.

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

$$
\begin{aligned}
imports_{Comps,Comps} &= imports_{Files,Files} \uparrow partof_{Files,Comps} \\
importsSpec_{Comps,Comps} &= (imports_{Comps,Comps} \upharpoonright_{ran} Specifics) \setminus \\
&\quad Id_{Specifics}
\end{aligned}
$$

rule:

$$
importsSpec_{Comps,Comps} \subseteq \emptyset
$$

violations:

$$
\begin{aligned}
v\_imports_{Comps,Comps} &= importsSpec_{Comps,Comps} \\
v\_Comps &= dom(v\_imports_{Comps,Comps}) \\
v\_imports_{Files,Files} &= (importsSpec_{Comps,Comps} \downarrow \\
&\quad partof_{Files,Comps}) \cap
\end{aligned}
$$

[3] An InfoPack can be defined to provide this information.

$imports_{Files,Files}$

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_Comps. 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:

$$\begin{aligned} ExpSpecifics &= dom(imports_{Comps,Comps} \setminus Id_{Comps}) \\ ExpGenerics &= Comps \setminus ExpSpecifics \end{aligned}$$

rule:

$$\begin{aligned} ExpGenerics &\subseteq Generics \\ ExpSpecifics &\subseteq Specifics \end{aligned}$$

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[4]. 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*[5]. 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.

Chapter 7

Concluding Remarks

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

7.1 Recommendations for Application

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

When applying SAR to an existing system, one should first consider which ArchiSpects are most valuable to reconstruct. Most of the discussed ArchiSpects are based on the *imports* relation. One imports a header file to be able to use a function, a type definition, a macro and/or a global variable from another file. The *imports* relation is in fact a mixture of a number of relations: *calls*, *accesses* and *typed*, and for object-oriented languages also the *inherits* relation. When reconstructing a system in more detail, one requires a refinement of the *imports* relation. For example, for each of these relations the *Component Dependency* ArchiSpect can be refined, e.g. *Component Dependency* for function calls.

	Architectural Views				
SAR levels	Logical View	Module View	Code View	Execution View	Hardware View
Optimized					
Managed		▶Generics and Specifics Conformance ▶Aspect Conformance ▶Usage Conformance ▶Layering Conformance	▶Aspect Assignment ▶Part-Of ▶Depend		
Redefined		▶Aspect Coupling ▶Cohesion and Coupling ▶Component Coupling	▶Aspect Assignment ▶Part-Of ▶Depend	▷Resource Usage	
Described		▶Using and Used Interfaces ▶Component Dependency ▶Software Concepts Model	▶Source Code Organisation ▶Build Process ▶Part-Of ▶Depend ▶Files	▷Process Communication ▷Process Topology ▷Processes	
Initial					

Table 7.1: Software Architecture Reconstruction

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

Although separately discussed, the module view, code view and execution view are related. In [KFJM99, BFG+99] we discussed how scenarios, applied to the *Switch* system, can help developers comprehend these three views. The importance of combining static analysis with dynamic analysis has also been discussed by Kazman and Carrière [KC98].

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

7.2 Relation Partition Algebra

Since 1994, when Relation Partition Algebra (RPA) was defined [FO94], in 1994, we have applied it in various areas of software architecture analysis. We experienced that RPA is suitable for making software abstractions, embellishing the presentation of information, expressing software metrics,

performing dedicated analyses, navigating through information, verifying architectural decisions and recognising patterns in software. We will briefly discuss these different areas.

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

The presentation of a function call graph of a large system probably results in a diagram that contains a large black area (i.e. all the edges of the graph). Abstractions can help to reduce the amount of information to embellish a graph presentation. For example, lifting a large *imports* relation reduces the amount of information in a smart way. Transitive reduction also improves the presentation of information. The transitive reduction removes *short-cuts*[1] from a (cycle-free) relation, resulting in a more convenient graph.

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

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

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

[1] A tuple $\langle x, z \rangle$ is a *short-cut* if the relation contains the tuples $\langle x, y_1 \rangle, \langle y_1, y_2 \rangle, \ldots, \langle y_n, z \rangle$ for some $n \geq 1$.

Presentation can be made more dynamic by offering some navigation mechanisms. For example, a user may want to *zoom-in* or *zoom-out* on certain information. *TabVieW* is a presentation tool that provides navigation abilities by executing RPA formulas and re-calculating a new table.

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

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

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

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

Appendix A

Extraction Tools

This appendix lists the source codes of a number of Perl [WCS96] programs. The programs are related to the *Files*, *Imports* and *Part-Of* InfoPacks discussed in Chapter 4. We used these tools in various reconstruction activities, but sometimes we changed these programs slightly in order to satisfy the system at hand.

A.1 file-exts.pl

```
#!/home/krikhaar/cadbin/perl
#   input: Files
#   standard output: typed.Files.Exts
#
while (<>) {
  chop;
  if (/.*\.([^.]+)/) {
    print "$_ $1\n";
  }
  else {
    print "! ERROR: unmatched filename $_\n";
  }
}
```

A.2 units.pl

```
#!/home/krikhaar/cadbin/perl
```

```
#   input: Files
#   standard output: partof.Files.Units
#
while (<>) {
  chop;
  if (/(.*)\.([^\.]+)/) {
    print "$1 $2\n";
  }
  else {
    print "! ERROR: unmatched filename $_\n";
  }
}
```

A.3 comment-strip.pl

```
#!/home/krikhaar/cadbin/perl
#   input: <file>
#   standard output: <stripped file>
#

# read whole file at once
undef $/;
$_ = <>;

# remove comment /* ... */ with minimal ... match
s{/\*.*?\*/}{}gsx;

# remove comment // ...
s{//.*?\n}[]gsx;

# pre-process line extension (\\)
s/(^[  ]*#.*?)\\\\\n/$1/g;   # at first line of file
s/(\n[  ]*#.*?)\\\\\n/$1/g;   # at all other lines

# print the stripped file
print;
```

A.4 C-imports.pl

```
#!/home/krikhaar/cadbin/perl
#   input: Files
#   standard output: imports.Files.Files
```

```
#
while (<>) {
  if (/[ ]*(\S*)/) {
    $INP=$1;
    open INP or die "! Unable to open file $INP\n";
    while (<INP>) {
      if (/^[ ]*#[ ]*include[ ]*(\S*)/) {
        $impfile = $1;
        $impfile =~ s/"//g;
        $impfile =~ s/<//;
        $impfile =~ s/>//;
        print "$INP $impfile\n";
      }
    }
  }
}
```

A.5 J-imports.pl

```
#!/home/krikhaar/cadbin/perl
#   input: Files
#   standard output: imports.Classes.Classes+
#
foreach $java (@ARGV) {
  print "Java: $java\n";
  $java =~ /.*[\/\\]([^\/\\]+).[Jj][Aa][Vv][Aa]/;
  $class = $1;

  print "Class: $class\n";

  $package = "";
  open JAVA, $java or die "! Unable to open file $java\n";
  while ( <JAVA> ) {
    if ( /package\s+([^;]+)\s*;/ ) {
      $package = $1.".";
    }

    if ( /import\s+([^;]+)\s*;/) {
      $import = $1;
      print "$package$java $import\n";
    }
  }
}
```

A.6 J-package.pl

```
#!/home/krikhaar/cadbin/perl
#   input: Files
#   standard output: defines.Classes+.Classes
#
foreach $java (@ARGV) {
  $java =~ /.*[\/\\]([^\/\\]+).[Jj][Aa][Vv][Aa]/;
  $class = $1;

  open JAVA, $java or die "! Unable to open file $java\n";
  while ( <JAVA> ) {
    if ( /package\s+([^;]+)\s*;/ ) {
      $package = $1;
      print "$package.* $package.$class\n";
      print "$package.$class $package.$class\n";
    }
  }
}
```

A.7 ObjC-imports.pl

```
#!/home/krikhaar/cadbin/perl
#   input: Files
#   standard output: imports.Files.Files
#
while (<>) {
  if (/[ ]*(\S*)/) {
    $INP=$1;
    open INP or die "! Unable to open file $INP\n";
    while (<INP>) {
      if (/^[ ]*#[ ]*import[ ]*(\S*)/) {
        $impfile = $1;
        $impfile =~ s/"//g;
        $impfile =~ s/<//;
        $impfile =~ s/>//;
        print "$INP $impfile\n";
      }
    }
  }
}
```

A.8 QAC-imports.pl

```
#!/home/krikhaar/cadbin/perl
#   input: <QAC intermediate file(s)>
#   standard output: imports.Files.Files
#
while (<>) {
  chop;
  @fields = split/[ \t]+/;
  if ($fields[0] =~ /<INCL>/) {
    print "$fields[1] $fields[2]\n";
  }
}
```

A.9 directory.pl

```
#!/home/krikhaar/cadbin/perl
#   input: <list of to be inspected directories>
#   standard output: partof.Files.Directories
#
while (<>) {
  if (/[  ]*(\S*)/) {
    $dir=$1;
    opendir DIR,$dir or die "! Unable to open file $INP\n";
    @allfiles = readdir $DIR;
    foreach $f (@allfiles) {
      print "$f $dir\n"
    }
    closedir;
  }
}
```

Appendix B

Abstraction Tools

B.1 Introduction

In this appendix we discuss some implementations of Relation Partition Algebra. A number of implementations have been created in a broad range of programming languages:

- functional language: Clean and *Prolog*;
- scripting language: Perl and *AWK*;
- database language: *SQL*.
- imperative language: Pascal, C, C++, Basic, and *Java*;

For each kind of language we will briefly discuss some issues of particular language.

One should not only think about the implementation of RPA operators, but also about the interface of these operators. We distinguish the following types of interfaces: API (application programmers interface) consisting of a set of functions (e.g. Java implementation), a command line (e.g. AWK implementation), a graphical calculator (e.g. SQL implementation) and a sophisticated graphical interface [Pet97]. These interfaces will not be discussed any further here.

B.2 RPA-Prolog

The first implementation of RPA was written in Prolog [SS86] using the SWI-Prolog interpreter [Wie96]. The key design decision for almost any

RPA implementation is the internal representation of sets and relations. For *RPA-Prolog* [Kri95] we have chosen the following data structure:

- A set is implemented as a list containing the elements.
- A relation is a compound term *obj(SetX, SetY, Rel)*; with an invariant $SetX = dom(Rel)$ and $SetY = ran(Rel)$. *Rel* is a list of compound terms *rel(Elem, ElemRel)*; where $Elem \in SetX$ and *ElemRel* is a list of elements with which *Elem* has a relation. Each *Elem* occurs only once in the *Rel* list.
- The elements in the list *Rel* and the elements in the list *ElemRel* are ordered according to the order in *SetX* and *SetY*, respectively.

The Prolog code of the *rel_dom* and *rel_comp* operators are listed below.

```
/* Example of facts that represent a set and relation */

set(functions, [main,a,b,c,d]).
relation(calls, obj(functions, functions,
                    [rel(main, [a,b]), rel(a, [b,c,d]),
                     rel(b,[d])])).

/*
   rel_dom(Relation, Set) <-
      Set is the domain of Relation
*/
rel_dom(obj(SetX, _, Rel), obj(SetX, Domain)) :-
  domain(Rel, Domain).
domain([rel(E,_)|R], [E|Dom]) :-
  domain(R, Dom).
domain([], []).

/*
  rel_comp(Relation1, Relation2, Relation) <-
     Relation is the composition Relation2 ; Relation1
*/
rel_comp(obj(NameX, NameY, Rel1), obj(NameY, NameZ, Rel2),
         obj(NameX, NameZ, Result)) :-
  set(NameY, SetY),
  set(NameZ, SetZ),
  comp(Rel1, SetY, SetZ, Rel2, Result).

comp([], _, _, _, []).
comp([rel(X,XList)|Rel1], SetY, SetZ,
     Rel2, [rel(X,CList)|Result]) :-
```

```
  compose(XList, SetY, SetZ, Rel2, CList),
  CList \== [],
  !,
  comp(Rel1, SetY, SetZ, Rel2, Result).
comp([_|Rel1], SetY, SetZ, Rel2, Result) :-
  comp(Rel1, SetY, SetZ, Rel2, Result).

addlist([], [], List, List).
addlist([Z|SetZ], [Z|YList], [Z|RestList], [Z|List]) :-
  !,
  addlist(SetZ, YList, RestList, List).
addlist([Z|SetZ], YList, [Z|RestList], [Z|List]) :-
  !,
  addlist(SetZ, YList, RestList, List).
addlist([Z|SetZ], [Z|YList], RestList, [Z|List]) :-
  !,
  addlist(SetZ, YList, RestList, List).
addlist([_|SetZ], YList, RestList, List) :-
  addlist(SetZ, YList, RestList, List).

compose([], _, _, []).
compose([Y|XList], [Y|SetY], SetZ, [rel(Y,YList)|Rel], List) :-
  !,
  compose(XList, SetY, SetZ, Rel, RestList),
  addlist(SetZ, YList, RestList, List).
compose(XList, [Y|SetY], SetZ, [rel(Y,_)|Rel], List) :-
  !,
  compose(XList, SetY, SetZ, Rel, List).
compose([Y|XList], [Y|SetY], SetZ, Rel, List) :-
  !,
  compose(XList, SetY, SetZ, Rel, List).
compose(XList, [_|SetY], SetZ, Rel, List) :-
  !,
  compose(XList, SetY, SetZ, Rel, List).
```

B.3 RPA-AWK

The input files for the AWK [AKW88] implementation consist of so-called RPA files. A `set` file contains a single element of the set at each line; in a `relation` file each line contains a tuple of two elements separated by white space. The lines in a `multi-set` file and a `multi-relation` file contain an

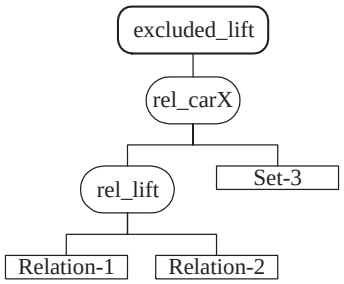


Figure B.1: High Level Operations

extra field to represent the weight.

We have implemented each RPA operator in a separate AWK script. The input of these scripts consists of files (including standard input) and the output is given on standard output. The *unix* pipe mechanism can be used to concatenate a number of operators, implementing for example an *excluded-lift* as depicted in Figure B.1:

```
rel_lift Relation1 Relation2 | rel_carX - Set3
```

A standard wrapper is used to parse the various arguments before the actual AWK script is called. This wrapper is responsible for checking the command line arguments, creating temporary files if needed (the Unix way of referring to standard input '-' has been used to read from standard input), and some exception handling code. For clarity the wrapper code has been removed from the code below. Again, we give the source code for the *rel_dom* and *rel_comp* operators.

```
#! /bin/sh
# Calculates the domain of a relation
Usage="Call: rel_dom <rel>"
#
<wrapper code> the variable $IN1 gets a value
#
awk 'seen[$1]==0 { print $1; seen[$1]=1; }
    ' $IN1
```

```
#! /bin/sh
```

```
# Composes two relations

Usage="Call: rel_comp <rel1> <rel2>"
#
<wrapper code> the variables $IN1 and $IN2 get a value
#
awk 'flag==0 { right[$1" "nr[$1]++]=$2; }
     flag==1 { n=nr[$2]; for (i=0; i<n; i++) {
                 s=$1" "right[$2" "i];
                 if (seen[s]==0) { print s ; seen[s]=1; }
               } }
    ' flag=0 $IN2 flag=1 $IN1
```

B.4 RPA-SQL

Another implementation is built on top of a database program. Any database that supports SQL would suffice, but we have used MS-Access [Boe96]. The various sets, relations, multi-sets and multi-relations are stored in separate tables. For relations, the columns in the table are named *dom* and *ran*, respectively. The table name refers to the relation's name.

```
Query: rel_dom(<rel>):

  SELECT DISTINCT <rel>.dom
  FROM <rel>;

Query: rel_comp(<rel1>, <rel2>):

  SELECT DISTINCT <rel1>.dom, <rel2>.ran
  FROM <rel1> AS rel1 INNER JOIN <rel2> AS rel2
    ON rel1.ran = rel2.dom
  GROUP BY rel1.dom, rel2.ran;
```

Note that the *rel_dom* query refers to the relation name `<rel>`. In SQL it is however not possible to use such a construct. Therefore we have developed a Visual Basic program to instantiate such free variables in our SQL description. A new SQL statement is generated in which the actual values of these variables are filled out. After that, the generated SQL statement is applied to the data in the database.

```
Query: rel_dom(calls):

  SELECT DISTINCT calls.dom
  FROM calls;

Query: rel_comp(calls, calls):

  SELECT DISTINCT calls.dom, calls.ran
  FROM calls AS rel1 INNER JOIN calls AS rel2
    ON rel1.ran = rel2.dom
  GROUP BY rel1.dom, rel2.ran;
```

The stack-oriented RPA calculator shown in Figure B.2 has been built on top of this program. Sets and relations can be pushed on the stack, operations (represented by different buttons) are applied to the element(s) on the top of the stack.

B.5 RPA-Java

In our Java [Web96] implementation of RPA we made much use of classes of standard packages. Various container classes of the *java.util* package were used; e.g. relations were represented in *HashTable*s. We constructed an RPA package that contains the classes *Set*, *Relation*, *MultiSet*, and *MultiRelation*. Each class defines its own methods that perform related RPA operations. For example, the *rel_dom* method of *Relation* calculates this object's domain; the *rel_comp* method of *Relation* calculates its composition with another *Relation* object. As already discussed in Section B.3, RPA files can be read and written by calling the provided IO methods.

The command line interface of the AWK implementation proved to be very handy in software architecture analysis. We have therefore implemented a similar interface on top of this Java implementation, consisting of a number of small programs, each calling a single method of the RPA package.

```
public class Relation {

  // storage of tuples of Relation
  protected Hashtable tuples;

/** dom( ) returns the domain of this Relation */
```

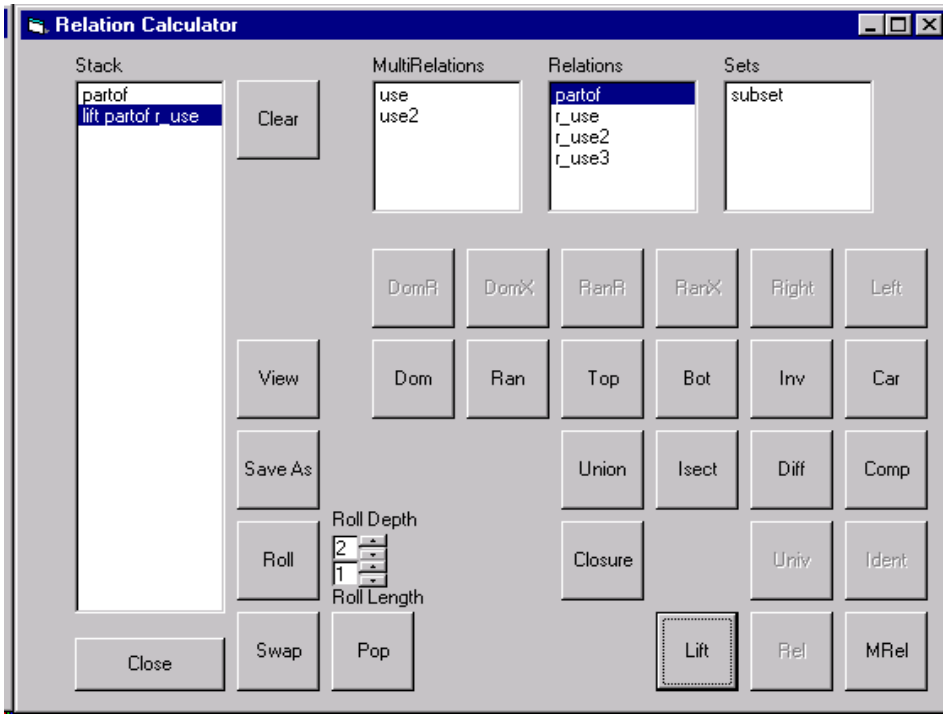


Figure B.2: RPA Calculator

```
  public Set dom( ){
    Set         s = new Set( );
    Enumeration e = tuples.keys( );
    while( e.hasMoreElements( ) ){
      s.insert( (String)( e.nextElement( ) ) );
    }
    return s;
  }

/** comp( Relation r ) returns a Relation defined as r o 'this' */
  public Relation comp( Relation r ){
    Relation res = new Relation();
    Enumeration e1 = r.tuples.keys( );

    while( e1.hasMoreElements( ) ){
      String      s1 = (String)( e1.nextElement( ) );
      Hashtable   h1 = (Hashtable)( r.tuples.get( s1 ) );
      Enumeration e2 = h1.keys( );
      while( e2.hasMoreElements( ) ){
        String    s2 = (String)( e2.nextElement( ) );
        Hashtable h2 = (Hashtable)( s.tuples.get( s2 ) );
        if( h2 != null ){

          Enumeration e3 = h2.keys( );
          int          i1 = get( h1, s2 );
          while( e3.hasMoreElements( ) ){
            String s3 = (String)( e3.nextElement( ) );
            int    i2 = get( h2, s3 );
            res.insert( s1, s3, Integer( i1 * i2 ) );
          }
        }
      }
    }
    return res;
  }
}
```

B.6 A Brief Comparison of RPA tools

The Prolog implementation should just be seen as a first experiment with the aim of becoming familiar with RPA.

The AWK implementation proved to be very suitable in daily practice. It

is easy to use (certainly for persons familiar with *unix* concepts). The AWK scripts are interpreted and can be easily combined using other shell scripts. It is easy to incorporate these scripts, e.g. in a *make* facility. A disadvantage may be the performance, the scripts may take some time in the case of large relations.

The SQL implementation performs poorly in the case of large relations. This holds especially for calculating a relation's transitive closure. The transitive closure is implemented as an extra program (Visual Basic) which iterates over a number of RPA operations (composition and union).

The advantage of Java is that it is platform-independent. It is also easy to integrate RPA in e.g. Java applets in a Web browser.

Appendix C

Presentation Tools

In this appendix we briefly discuss, in a chronological order, a number of proprietary presentation tools we have developed over the years.

C.1 Teddy-Classic

Teddy-Classic [Omm93, Roo94] displays components and relations between components. Components are represented by boxes and relations are represented by lines. *Teddy-Classic* requires as input a `relation` file and optionally a `view` file and a `component` file. The user can layout the components on the screen (using a mouse). The layout can be saved in a so-called `view` file. Later on, the `view` file can be used again, even in combination with another `relation` file (having the same components as carrier). So, in *Teddy-Classic*, the `relation` and `view` are separate concepts.

Components are clickable, meaning that the user can 'click' on a box, which results in a text viewer with component information. This 'click' information is described in a `component` file that describes the relation between components and information files. *Teddy-Classic* has various types of boxes, which represent different types of components. The lines also have different representations, dictated by the types of components to which they are connected.

Figure C.1 shows an example of the output of *Teddy-Classic*. It shows the same information as presented in the diagram of Figure 4.11 (page 83). A thick line between two components means that there is a relation between the graphically lower component and the higher component. A thin line

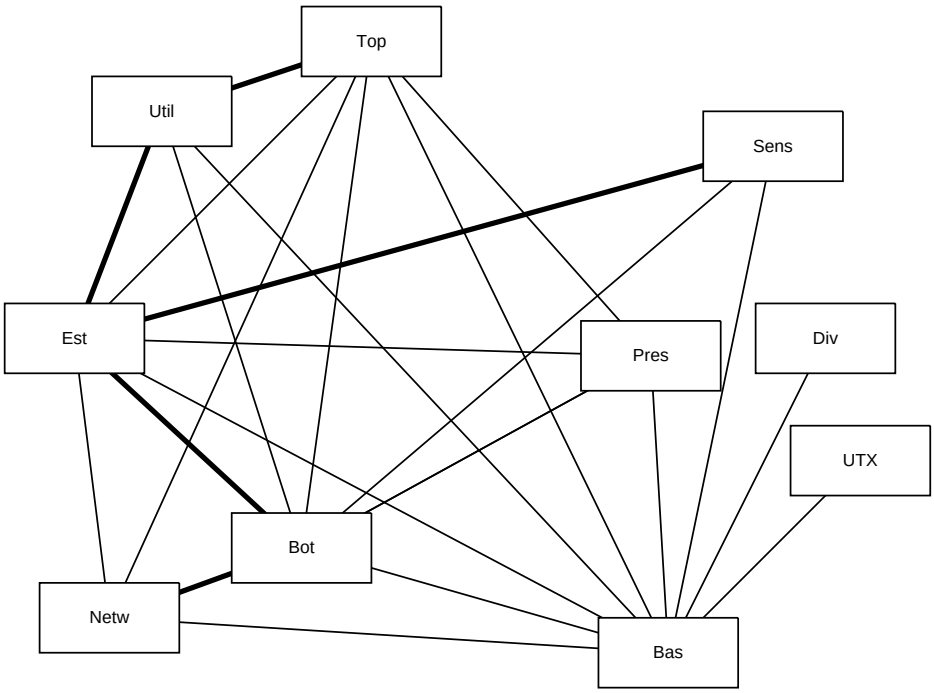


Figure C.1: Teddy-Classic

represents a relation between the higher component and the lower component. From these rules we infer that bi-directional relations are always represented as thick lines.

Teddy-Classic was written, in 1992, in the programming language C using X windows. The separation of relations, click information and views is a powerful concept: various relations can be displayed with the same view information. A drawback of *Teddy-Classic* is its graphical appearance. Bi-directional relations are not explicitly handled and all the boxes are of the same size; lines start (end) in the middle of a box, which reduces the possibilities of creating an appealing layout. There are now more tools available that provide similar functionalities, e.g. *Rigi* [SWM97], developed at the University of Victoria, Canada.

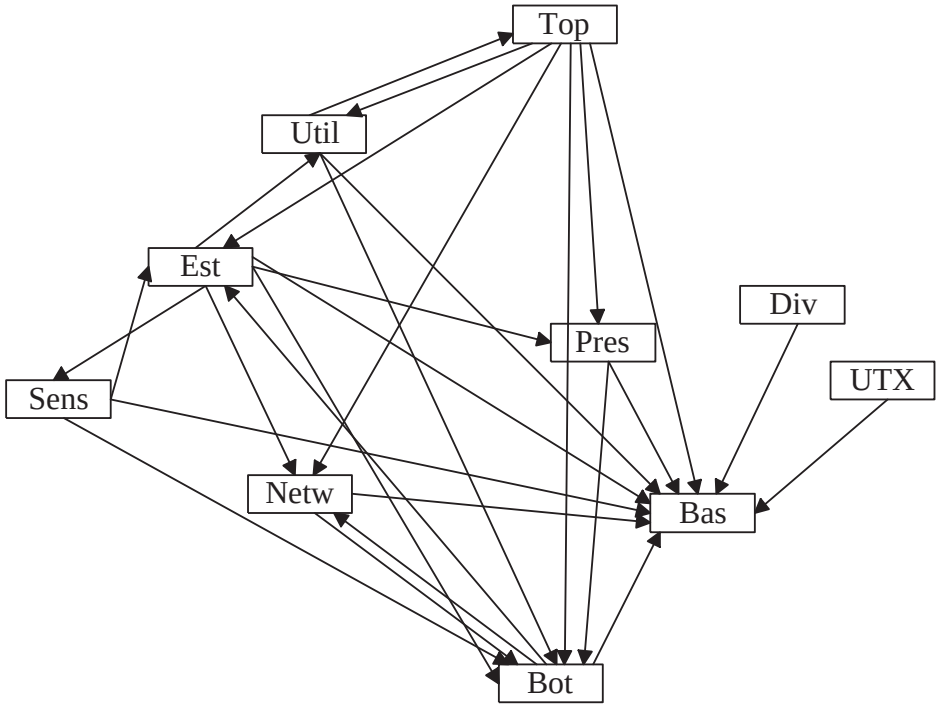


Figure C.2: Teddy-Visio

C.2 Teddy-Visio

In 1996, the functionality of *Teddy-Classic* was also implemented in Visual Basic in combination with *Visio*^{TM} [Vis]. The *Visio* tool displays 1-D and 2-D objects, say for clarity arrows and boxes. Each box has various connection points to which an arrow can be connected. The connected arrows automatically re-size when a box is moved (using a mouse). *Visio* calculates the best connection points for arrows (they call it 'dynamic glueing') and it automatically layouts boxes. Furthermore, arrows may be straight, but the tool can also bend lines to beautify the layout. An example of the output of *Teddy-Visio* is given in Figure C.2.

C.3 Teddy-PS

The aim of *Teddy-PS* was to present architectural information in forms resembling as closely as possible the diagrams already used in the architecture documentation concerned. So, in 1996, *Teddy-PS* was developed to resolve drawbacks of *Teddy-Classic*: boxes of the same sizes and arrows that start (end) at predefined positions at the border of these boxes.

Teddy-PS requires as input a `view` file and one or more `relation` files. The `view` file is a prepared postscript file that contains the layout of all the components (boxes) and possible relations between components (arrows). The `view` file is manually created (by adapting a copy of a template `view` file). Per `relation`, *Teddy-PS* filters, from the `view` file, the corresponding arrows and gives them a colour. *Teddy-PS* also calculates the sizes of arrows in the case of multi-relations. An example of the output of *Teddy-PS* is given in Figure C.3.

C.4 Teddy-ArchView

All of the *Teddy* tools discussed above use two dimensions to present information. *Teddy-ArchView*, developed in 1997, presents architectural information in a three-dimensional picture [FJ98]. The tool's input consists of various relations and part-of relations. From this information, *Teddy-ArchView* generates a VRML [VRM] description. The result is presented in a standard Web browser (using a plug-in, a VRML viewer) or any other VRML viewer. The viewer gives the user the opportunity to walk through the information, in a virtual-reality world. An example of such a world is given in Figure C.4.

C.5 TabVieW

TabVieW (developed in 1998) presents relations and multi-relations in a tabular (or matrix) form in a Web browser. The input for this tool is a *use* relation, e.g. $imports_{Files,Files}$, and a number of *partof* relations. For the sake of discussion, we will call the *use* relation $U_{1,1}$ and the chain of *partof* relations $P_{1,2}, P_{2,3}, \ldots$. For example, decomposition level 1 refers to *Files*, level 2 to *Comps*, level 3 to *Subs* and level 4 to *Systems*, so $partof_{1,2}$ describes which *Files* belong to which *Comps*.

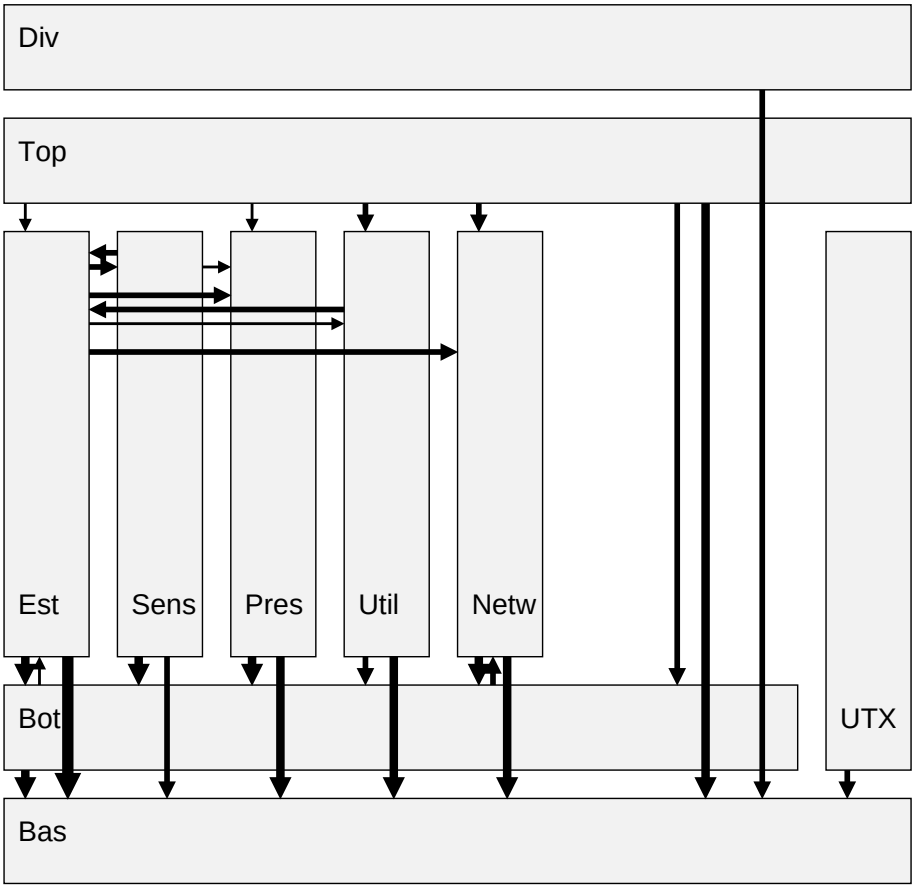


Figure C.3: Teddy-PS

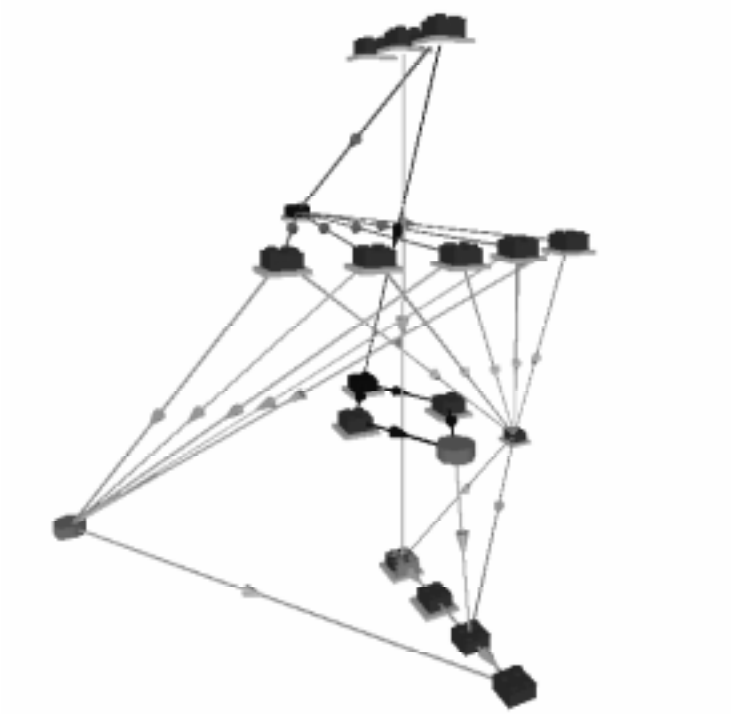


Figure C.4: Teddy-ArchView

TabVieW shows, in a Web browser, a matrix that belongs to a tuple of focus points: a focus point in the domain x at level d plus a focus point in the range y at decomposition level r. In the first column the domain's focus point x is presented, in the second column the constituents of x are listed. In Figure C.5 the domain's focus point is *Comm*. The constituents of x can be calculated with RPA: $partof_{d-1,d}.x$. In the given example the constituents of x are, amongst others, *Cil*, *Cnl*, and *Std*. Analogously, in the first row the range's focus point y is presented, and in the second row its constituents ($partof_{r-1,r}.y$) are given. The cells in the matrix show whether a tuple exists in the relation $U_{d-1,r-1}$ or, in the case of a multi-relation[1], they also show the corresponding weight in RPA:

$$U_{d-1,r-1} \quad = \quad P_{r-2,r-1} \circ \ldots \circ P_{1,2} \circ U_{1,1} \circ P^{-1}_{d-2,d-1} \circ \ldots \circ P^{-1}_{1,2}$$

The user can navigate through the information by clicking on hyperlinks. Zooming-in can be achieved by clicking on elements in the second column or row. A new matrix is then calculated and presented with the clicked element as a new focus point (preserving the other focus point). Zooming-out can be performed by clicking on the element in the first column (row). The parent of this element becomes the new focus point (again preserving the other focus point). When the user clicks on the cells, the corresponding tuples (*lowered* to decomposition level 1) are presented in a table. All the calculations are performed on request (i.e. after a user's click); a Perl script accessed via *cgi* [SQ96] calculates a new matrix or table.

Since the development of this prototype, a more elaborate version of *TabVieW* has been implemented by the *Switch* development team [Gla98, BGKW99]. It contains a more dedicated user interface containing, amongst other things, more zooming and hiding actions.

[1] In case of multi-relations, the *part-of* relation P of the formula should be interpreted as a multi-relation.

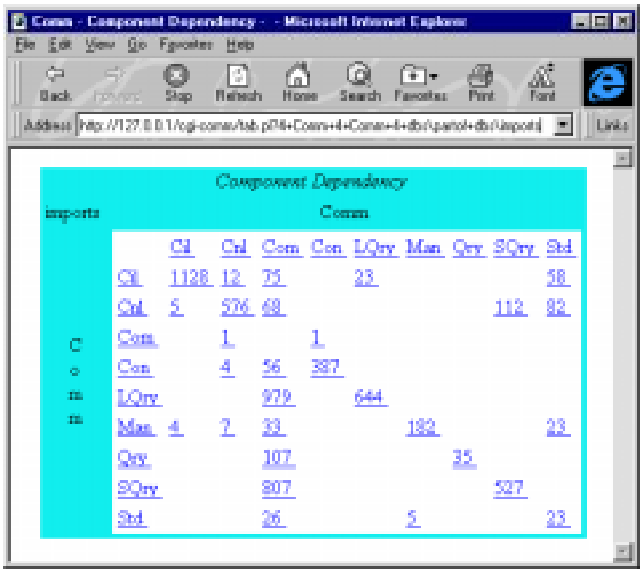


Figure C.5: TabVieW

Appendix D

RPA Operators in a Nutshell

In this appendix we give an overview (quick reference guide) of all the operators on sets, multi-sets, relations and multi-relations. The first column of these tables contains the operators and the types of operands (all given in the same order as discussed in Chapter 3). The second column contains the mathematical names, and the third column contains the *mnenomic*, which includes only ASCII characters (recommended as function/method name in implementations).

$=: Set \times Set \to Bool$	equal	set_eq
$\subseteq: Set \times Set \to Bool$	subset	set_sub
$\supseteq: Set \times Set \to Bool$	superset	set_sup
$\subset: Set \times Set \to Bool$	strict subset	set_ssub
$\supset: Set \times Set \to Bool$	strict superset	set_ssup
$\cup: Set \times Set \to Set$	union	set_union
$\cap: Set \times Set \to Set$	intersection	set_isect
$\setminus: Set \times Set \to Set$	difference	set_diff
$\overline{}: Set \to Set$	complement	set_compl
$\lvert\ \rvert: Set \to Int$	size	set_size

Table D.1: Operations on Sets

$=$: $Rel \times Rel \to Bool$	equal	rel_eq
$\subseteq$: $Rel \times Rel \to Bool$	subset	rel_sub
$\supseteq$: $Rel \times Rel \to Bool$	superset	rel_sup
$\subset$: $Rel \times Rel \to Bool$	strict subset	rel_ssub
$\supset$: $Rel \times Rel \to Bool$	strict superset	rel_ssup
$\cup : Rel \times Rel \to Rel$	union	rel_union
$\cap : Rel \times Rel \to Rel$	intersection	rel_isect
$\setminus : Rel \times Rel \to Rel$	difference	rel_diff
$\circ : Rel \times Rel \to Rel$	composite	rel_comp
$\times : Set \times Set \to Rel$	cartesian product	rel_times
$Id_X : Set \to Rel$	identity	rel_ident
$dom : Rel \to Set$	domain	rel_dom
$ran : Rel \to Set$	range	rel_ran
$car : Rel \to Set$	carrier	rel_car
$\restriction_{dom}$: $Rel \times Set \to Rel$	domain restriction	rel_domR
$\restriction_{ran}$: $Rel \times Set \to Rel$	range restriction	rel_ranR
$\restriction_{car}$: $Rel \times Set \to Rel$	carrier restriction	rel_carR
$\setminus dom : Rel \times Set \to Rel$	domain exclusion	rel_domX
$\setminus ran : Rel \times Set \to Rel$	range exclusion	rel_ranX
$\setminus car : Rel \times Set \to Rel$	carrier exclusion	rel_carX
$\top : Rel \to Set$	top	rel_top
$\bot : Rel \to Set$	bottom	rel_bot
$\rhd : Set \times Rel \to Set$	forward projection	rel_fproj
$\lhd : Rel \times Set \to Set$	backward projection	rel_bproj
$. : Rel \times Elem \to Set$	left image	rel_left
$. : Elem \times Rel \to Set$	right image	rel_right
$\mid \mid : Rel \to Int$	size	rel_size
$^{-1} : Rel \to Rel$	converse	rel_conv
$\overline{} : Rel \to Rel$	complement	rel_compl
$^{+} : Rel \to Rel$	transitive closure	rel_clos
$^{*} : Rel \to Rel$	reflexive transitive closure	rel_rclos
$^{-} : Rel \to Rel$	transitive reduction	rel_hasse

Table D.2: Operations on Binary Relations

$\uparrow$: $Rel \times Par \rightarrow Rel$	lifting	rel_lift
$\downarrow$: $Rel \times Par \rightarrow Rel$	lowering	rel_low

Table D.3: Operations on Part-Of Relations

$\lceil\ \rceil : Set \rightarrow mSet$	mapping	set_2mset
$\lceil\ \rceil_n : Set \times Int \rightarrow mSet$	n-mapping	set_n2mset
$\lfloor\ \rfloor : mSet \rightarrow Set$	mapping	mset_2set
$=$: $mSet \times mSet \rightarrow Bool$	equal	mset_eq
$\subseteq$: $mSet \times mSet \rightarrow Bool$	subset	mset_sub
$\supseteq$: $mSet \times mSet \rightarrow Bool$	superset	mset_sup
$\subset$: $mSet \times mSet \rightarrow Bool$	strict subset	mset_ssub
$\supset$: $mSet \times mSet \rightarrow Bool$	strict superset	mset_ssup
$\cup : mSet \times mSet \rightarrow mSet$	union	mset_union
$+ : mSet \times mSet \rightarrow mSet$	addition	mset_sum
$\cap : mSet \times mSet \rightarrow mSet$	intersection	mset_isect
$\setminus : mSet \times mSet \rightarrow Set$	difference	mset_diff
$\overline{\ \ } : mSet \rightarrow mSet$	complement	mset_compl
$\|\ \| : mSet \rightarrow Int$	size	mset_size

Table D.4: Operations on Multi-Sets

$\lceil\ \rceil : Rel \to mRel$	mapping	rel_2mrel
$\lceil\ \rceil_n : Rel \times Int \to mRel$	n-mapping	rel_n2mrel
$\lfloor\ \rfloor : mRel \to Rel$	mapping	mrel_2rel
$=: mRel \times mRel \to Bool$	equal	mmrel_eq
$\subseteq: mRel \times mRel \to Bool$	subset	mrel_sub
$\supseteq: mRel \times mRel \to Bool$	superset	mrel_sup
$\subset: mRel \times mRel \to Bool$	strict subset	mrel_ssub
$\supset: mRel \times mRel \to Bool$	strict superset	mrel_ssup
$\cup : mRel \times mRel \to mRel$	union	mrel_union
$+ : mRel \times mRel \to mRel$	addition	mrel_sum
$\cap : mRel \times mRel \to mRel$	intersection	mrel_isect
$\setminus : mRel \times mRel \to mRel$	difference	mrel_diff
$\circ : mRel \times mRel \to mRel$	composite	mrel_comp
$\times : mSet \times mSet \to mRel$	cartesian product	mrel_times
$Id_X : Set \to mRel$	identity	mrel_ident
$Id_{X,n} : Set \times Int \to mRel$	identity	mrel_ident
$dom : mRel \to mSet$	domain	mrel_dom
$ran : mRel \to mSet$	range	mrel_ran
$car : mRel \to mSet$	carrier	mrel_car
$\restriction_{dom}: mRel \times Set \to mRel$	domain restriction	mrel_domR
$\restriction_{ran}: mRel \times Set \to mRel$	range restriction	mrel_ranR
$\restriction_{car}: mRel \times Set \to mRel$	carrier restriction	mrel_carR
$\setminus dom : mRel \times Set \to mRel$	domain exclusion	mrel_domX
$\setminus ran : mRel \times Set \to mRel$	range exclusion	mrel_ranX
$\setminus car : mRel \times Set \to mRel$	carrier exclusion	mrel_carX
$\top : mRel \to mSet$	top	mrel_top
$\bot : mRel \to mSet$	bottom	mrel_bot
$\rhd : Set \times mRel \to mSet$	forward projection	mrel_fproj
$\lhd : mRel \times Set \to mSet$	backward projection	mrel_bproj
$. : mRel \times Elem \to mSet$	left image	mrel_left
$. : Elem \times mRel \to mSet$	right image	mrel_right
$\|\ \| : mRel \to Int$	size	mrel_size
$^{-1} : mRel \to mRel$	converse	mrel_conv
$\overline{\ \ } : mRel \to mRel$	complement	mrel_compl
$^{+} : mRel \to mRel$	transitive closure	mrel_clos
$^{*} : mRel \to mRel$	reflexive transitive closure	mrel_rclos
$^{-} : mRel \to mRel$	transitive reduction	mrel_hasse
$\uparrow: mRel \times Par \to mRel$	lifting	mrel_lift
$\downarrow: mRel \times Par \to mRel$	lowering	mrel_low

Table D.5: Operations on Multi-Relations

Bibliography

[AIS77] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language – Towns Buildings Construction –*. Oxford University Press, 1977.

[AKW88] Alfred V. Aho, Brian W. Kernigham, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley Publishing Company, 1988.

[Bab86] Wayne A. Babich. *Software Configuration Management — Coordination for Team Productivity*. Addison-Wesley Publishing Company, 1986.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Publishing Company, 1998.

[BFG+99] Reinder J. Bril, Loe M.G. Feijs, André Glas, René L. Krikhaar, and Thijs Winter. Maintaining a Legacy: towards support at the architectural level. *Journal of Software Maintenance*, 1999. Submitted on invitation.

[BGKW99] Reinder J. Bril, André Glas, René L. Krikhaar, and Thijs Winter. "Hiding" expressed using Relation Algebra with Multi-Relations. 1999. Submitted for publication.

[BHS80] Edward H. Bersoff, Vilas D. Henderson, and Stanley G. Siegel. *Software Configuration Management — An investment in Product Integrity*. Prentice Hall, 1980.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A systems of patterns – pattern-oriented software architecture –*. John Wiley, 1996.

[Boe96] Koos Boertjens. *Access 7 voor Windows 95 – voor gevorderden –*. Academic Service, 1996.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, 1991.

[Box98] Don Box. *Essential COM*. Addison-Wesley Publishing Company, 1998.

[Bro82] Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley Publishing Company, 1982.

[Bro99] Jacques Brook. Design and Implementation of a tool for reclustering. Master's thesis, Eindhoven University of Technology, 1999.

[CC90] E. Chikofsky and J. Cross. Reverse Engineering and Design Recovery: A taxanomy. *IEEE Software*, pages 13–17, January 1990.

[Cle] http://www.rational.com/products/ccmbu/clearcase.

[CN91] Brad J. Cox and Andrew J. Novobiliski. *Object-Oriented Programming – an evolutionary approach*. Addison-Wesley Publishing Company, second edition, 1991.

[Con] http://www.continuus.com.

[Cor89] T.A. Corbi. Program Understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[CS95] M. Cusumano and R. Selby. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. New York: Free Press, 1995.

[Dij68] E.W. Dijkstra. The structure of the THE-multiprogramming system. *Communications on the ACM*, 11(5):341–346, 1968.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1990.

[Fel79] Stuart I. Feldman. Make - A Program for Maintaining Computer Programs. *Software - Practice and Experience*, 1979.

[FHK+97] P.J. Finnigan, R.C. Holt, I. Kalas, S. Kerr, and K. Kontogiannis. The software Bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[FJ98] L. Feijs and R.P. de Jong. 3D visualization of software architectures. *Communications on the ACM*, 41(12):73–78, December 1998.

[FK99] L.M.G. Feijs and R.L. Krikhaar. Relation Algebra with Multi-Relations. *International Journal Computer Mathematics*, 70:57–74, 1999.

[FKO98] L. Feijs, R. Krikhaar, and R. van Ommering. A relational approach to Software Architecture Analysis. *Software Practice and Experience*, 28(4):371–400, April 1998.

[FO94] L.M.G. Feijs and R.C. van Ommering. Theory of Relations and its Applications to Software Structuring. Philips internal report, Philips Research, 1994.

[FO99] L.M.G. Feijs and R.C. van Ommering. Relation Partition Algebra – mathematical aspects of uses and part-of relations –. *Science of Computer Programming*, 33:163–212, 1999.

[Fow97] Martin Fowler. *UML Distilled - applying the standard object modeling language.* Addison-Wesley Publishing Company, 1997.

[GAO95] D. Garlan, R. Allen, and J. Ocklerbloom. Architectural Mismatch: Why reuse is so hard. *IEEE Software*, pages 179–184, November 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.

[Gla98] André Glas. Module Architecture Browser. Technical report, Philips Business Communications, 1998.

[GMS93] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion.* Addison-Wesley Publishing Company, 1993.

[Hol96] Richard C. Holt. Binary Relation Algebra Applied to Software Architecture. CSRI Technical Report 345, Computer Systems Research Institute, 1996.

[Hol98] Richard C. Holt. Structural Manipulations of Software Architecture using Tarski Relational Algebra. In *Proceedings of Fifth Working Conference on Reverse Engineering*. IEEE Computer Society, 1998.

[Hum89] Watts S. Humphrey. *Managing the Software Process*. Addison-Wesley Publishing Company, 1989.

[ITU93] ITU. CCITT Z.200 CCITT High Level Language (CHILL) – Recommendations Z200, 1993.

[Jav] http://java.sun.com/doc/language_specification.html.

[JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse – Architecture, Process and Organization for Business Success*. ACM Press, 1997.

[Joh75] S.C. Johnson. YACC – Yet Another Compiler-Compiler. Technical report, Bell Laboratories, 1975.

[Jon88a] H.B.M. Jonkers. Introduction to COLD-K. Deliverable of esprit project meteor, Philips Research, 1988.

[Jon88b] H.B.M. Jonkers. The SPRINT Method. Philips internal report RWB-113-hj-90071, Philips Research, 1988.

[KABC96] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario Based Analysis of software architecture. *IEEE Software*, pages 47–55, November 1996.

[KC98] Rick Kazman and S. Jeromy Carrière. View Extraction and View Fusion in Architectural Understanding. In *Proceedings of the Fifth International Conference on Software Reuse*, 1998.

[KFJM99] R.L. Krikhaar, L.M.G. Feijs, R.P. de Jong, and J.P. Medema. Architecture Comprehension Tools for a PBX System. In *Proceedings of Third European Conference on Software Maintenance and Reengineering*, pages 31–39. IEEE Computer Society, 1999.

[KL94] René Krikhaar and Frank van der Linden. Comparison of the Building Block Method with other methods. Philips internal report RWB-508-re-94073, Philips Research, 1994.

[KPS+99] René Krikhaar, André Postma, Alex Sellink, Marc Stroucken, and Chris Verhoef. A Two-phase Process for Software Architecture Improvement. *submitted for publication*, 1999.

[KPZ99] R.L. Krikhaar, M. Pennings, and J. Zonneveld. Employing Use-cases and Domain Knowledge for Comprehending Resource Usage. In *Proceedings of Third European Conference on Software Maintenance and Reengineering*, pages 14–21. IEEE Computer Society, 1999.

[KR88] B. Kernighan and D.M. Ritchie. *The C programming Language*. Prentice Hall, second edition, 1988.

[Kri94] René Krikhaar. Dynamic Aspects of the Building Block Method. Philips internal report RWB-508-re-94071, Philips Research, 1994.

[Kri95] R.L. Krikhaar. A Formal View on the Building Block Method. Philips internal report RWB-506-re-95014, Philips Research, 1995.

[Kri97] R.L. Krikhaar. Reverse Architecting Approach for Complex Systems. In *Proceedings International Conference on Software Maintenance*, pages 4–11. IEEE Computer Society, 1997.

[Kri98] René Krikhaar. Reverse Architecting *Comm* – User Manual –. Philips internal report, Philips Research, 1998.

[Kro93] Klaus Kronlof. *Method Integration – Concepts and Case Studies*. John Wiley, 1993.

[Kru95] P. Kruchten. The 4 + 1 View Model of Architecture. *IEEE Software*, pages 42–50, November 1995.

[KW94] René Krikhaar and Jan Gerben Wijnstra. Product development with the Building Block Method – a process perspective –. Philips internal report RWB-508-re-94070, Philips Research, 1994.

[KW95] René Krikhaar and Jan Gerben Wijnstra. Architectural Concepts for the Single Product Line. Philips internal report RWB-508-re-95047, Philips Research, 1995.

[Lam85] Leslie Lamport. *LaTeX- A Document Preparation System.* Addison-Wesley Publishing Company, 1985.

[LLB+98] Bruno Laguë, Charles Leduc, André Le Bon, Ettore Merlo, and Michel Dagenais. An Analysis Framework for Understanding Layered Software Architectures. In *Proceedings 6th International Workshop on Program Comprehension*, pages 37–44. IEEE Computer Society, 1998.

[LLMD97] Bruno Laguë, Charles Leduc, Ettore Merlo, and Michel Dagenais. A Framework for the Analysis of Layered Software Architectures. In *Proceedings of the 2nd International Workshop on Empirical Studies of Software Maintenance*, pages 75–78. IEEE Computer Society, 1997.

[LM95] F. van der Linden and J. Müller. Creating Architectures with Building Blocks. *IEEE Software*, pages 51–60, November 1995.

[LS86] M.E. Lesk and E. Schmidt. LEX – A lexical analyzer generator. Technical report, Bell Laboratories, 1986.

[Med98] Jeroen Medema. Manual on Module Architecting *Switch.* Philips Internal Report SR 2290-98.0214, Philips Business Communications, 1998.

[MK97] J.P. Medema and R.L. Krikhaar. Reverse Module Architecting for *Med* - handbook for analysing the module interconnection architecture of the *Med* software -. Philips internal report Nat. Lab. Technical Note 80/97, Philips Research, 1997.

[MMR+98] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In *Proceedings 6th International Workshop on Program Comprehension*, pages 45–52. IEEE Computer Society, 1998.

[MN97] Gail Murphy and David Notkin. Reengineering with Reflexion Models: A Case Study. *Computer*, pages 29–36, August 1997.

[MNS95] G. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *Proceedings Third ACM Sigsoft Symposium on Foundations of Software Engineering*, pages 18–28. ACM New York, 1995.

[Omm93] R.C. van Ommering. TEDDY user's manual. Technical report 12NC-4322-2730176-1, Philips Research, 1993.

[Par76] D. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, 1976.

[PCW85] D.L. Parnas, P. Clements, and D. Weiss. The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, 1985.

[Pet97] Marcel Peters van Ton. Visual Logic: an experiment in graphical assertion language design. Master's thesis, Eindhoven University of Technology, 1997.

[Pro96] Programming Research Ltd. *QAC Version 3.1 User's Guide*, 1996.

[PW92] D. Perry and A. Wolf. Foundations for the Study of Software Architecture. *ACM Software Engineering Notes*, 17(7):40–52, 1992.

[PZ93] G. Parikh and N. Zvegintzov. *Tutorial on Software Maintenance*. Los Alamitos, CA: IEEE Computer Society Press, 1993.

[Roo94] M. Roosen. Design Visualization definition and concepts. Philips internal report RWB-508-re-94040, Philips Research, 1994.

[SG96] Mary Shaw and David Garlan. *Software Architecture – perspectives on an emerging discipline –*. Prentice Hall, 1996.

[SM77] Donald F. Stanat and David F. McAllister. *Discrete Mathematics in Computer Science*. Prentice Hall, 1977.

[SMB83] C.H. Smedema, P. Medema, and M. Boasson. *The Programming Languages - Pascal, Modula, CHILL, Ada*. Prentice Hall, 1983.

[SMC74] W.P. Stevens, G.J. Myers, and L.L. Constantine. Structured Design. *IBM Systems Journal*, 13(2):115–139, 1974.

[SNH95] D. Soni, R. Nord, and C. Hofmeister. Software Architecture in Industrial Application. In *Proceedings International Conference on Software Engineering*, pages 196–207, 1995.

[SQ96] Stephen Spainbour and Valerie Quercia. *Webmaster in a Nutshell.* O'Reilly, 1996.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog – Advanced Programming Techniques –.* The MIT Press, 1986.

[SWM97] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Mueller. Rigi: A Visualisation Environment for Reverse Engineering. In *Proceedings of International Conference on Software Engineering*, pages 606–607, 1997.

[Szy97] Clemens Szyperski. *Component Software – beyond Object-Oriented Programming.* Addison-Wesley Publishing Company, 1997.

[Tan76] Andrew S. Tanenbaum. *Structured Computer Organization.* Prentice Hall, 1976.

[Tar41] A. Tarski. On the calculus of relations. *Journal of Symbolic Computing*, 6(3):73–89, 1941.

[VAX96] Digital Equipment Corporation. *Using VAXset – user manual*, 1996.

[Vis] http://www.visio.com/.

[VRM] http://www.vrml.org/.

[War62] S. Warshall. A Theorem on Boolean Matrices. *Journal on the ACM*, pages 11–12, 1962.

[WCS96] L. Wall, T. Christiansen, and R.L. Schwartz. *Programming Perl.* O'Reilly, second edition, 1996.

[Web96] Joe Weber. *Using JAVA.* QUE, second edition, 1996.

[Wie96] Jan Wielemaker. *SWI-Prolog Reference Manual.* University of Amsterdam, 1996.

[Wig97] T.A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. *Proceedings of Fourth Working Conference on Reverse Engineering*, 1997.

[Wij96] Jan Gerben Wijnstra. Supporting System Families with Generics and Aspects. Philips internal report RWB-506-re-96004, Philips Research, 1996.

[Wir83] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, second corrected edition, 1983.

[WW90] Robin J. Wilson and John J. Watkins. *Graphs – an introductory approach.* John Wiley, 1990.

[YC79] Edward Yourdon and Larry L. Constantine. *Structured Design – Fundamentals of a Discipline of Computer Program and Systems Design.* Yourdon Press, 1979.

Glossary

4 + 1 View Model:
: A specific architecture view model that partitions an architecture into 4 views (logical view, development view, process view, physical view) plus an additional view (scenarios) that combines the four views (see Section 1.2.1).

abstraction:
: An activity that raises extracted information to a higher level of abstraction, e.g. to an architectural level (see Section 2.3).

ArchiSpect:
: A concept of the SAR method that describes how an aspect of architecture of an existing system can be reconstructed (see Section 2.5.2).

architectural pattern:
: A recurring solution to a problem relating to architecture (see Section 1.5).

architecture:
: The main structures of a system, also used as a shorthand for software architecture (see Chapter 1).

architecture conformance:
: The situation in which the implementation of a system conforms to the architecture (see Section 6.1).

architecture improvement:
: The process of improving the architecture of an existing system (see Section 2.4).

architecture verification:
: The process of verifying an implementation by comparing it with its architecture to assess architecture conformance (see Section 6.1).

AV model:
: An architectural view model that partitions an architecture into five different views (logical view, module view, code view, execution view, physical view) plus an extra view (scenarios) that combines the five views (see Section 1.2.3).

binary relation:
: A set of tuples ($\{\ldots, \langle x, y\rangle, \ldots\}$) representing a certain relation, e.g. *calls* for function calls within a system (see Section 3.3).

Building Block method:
: A dedicated software architecture method that stems from telecommunication system development (see Section 2.2).

component:
: A generic name for a piece of software; this term is sometimes used to refer to a piece of software at a certain level of decomposition.

decomposition hierarchy:
: The hierarchy of software entities of a system including their containment relationship (see Section 4.7).

decomposition level:
: A certain level in the decomposition hierarchy (see Section 4.7).

extraction:
: An activity involving the retrieval of information from source code, design documentation and/or domain experts (see Section 2.3).

Files:
: An example of the notation used in this thesis for sets (see Section 3.6.2).

forward architecting:
: The discipline of creating new architectures for software systems (see Section 2.2).

impact analysis:
: The process of simulating a possible idea for modification in a software model to analyse, in advance, the possible consequences of its application to actual source code (see Section 5.1).

$imports_{\mathbf{Files},\mathbf{Files}}$:
: An example of the notation used in this thesis for multi-relations: $imports_{\mathbf{Files},\mathbf{Files}}$ represents the multi-relation: $imports \subseteq Files \times Files$ (see Section 3.6.2).

$imports_{Files,Files}$:
: An example of the notation used in this thesis for binary relations: $imports_{Files,Files}$ represents the binary relation: $imports \subseteq Files \times Files$ (see Section 3.6.2).

InfoPack:
: A concept of the SAR method that describes how to extract (architecture-relevant) information from existing software (see Section 2.5.2).

lifting:
: An RPA operation involving a relation and a part-of relation resulting in a relation at a higher level of abstraction (see Section 3.4).

lowering:
: An RPA operation involving a relation and a part-of relation resulting in a relation having a finer coarse of granularity (see Section 3.4).

multi-relation:
: A bag of tuples $\langle a, b \rangle$, represented as a set of triples $\langle a, b, n \rangle$, where n represents the number of occurrences, called the weight (see Section 3.5).

multi-set:
: A bag of entities, represented as a set of tuples $\langle a, n \rangle$, where n represents the number of occurrences, called the weight (see Section 3.5).

part-of relation:
: A relation that describes a partition, i.e. a division of a set of entities into various non-overlapping (named) parts (see Section 3.4).

presentation:
: The activity of showing (architectural) information to developers and architects in an appropriate way e.g. by means of diagrams, tables and/or text (see Section 2.3).

re-architecting:
: The process of modifying the software architecture of an existing system (see Section 2.3).

repository:
: A data-store containing software-related information (see Section 2.3).

reverse architecting:
: Reverse engineering of software architectures (see Section 2.3).

reverse engineering:
: The process of analysing a subject system to identify the system's components and their relationships and create representations of the system in another form or at a higher level of abstraction (see Section 2.3).

RPA:
: Relation Partition Algebra, an algebra based upon sets, relations and partitions (see Chapter 3).

SAR:
: Software Architecture Reconstruction, a method for reconstructing software architectures (see Chapter 2).

SAR level:
: A level of the Software Architecture Reconstruction method (see Section 2.5.1).

set:
: A collection of objects, called elements or members (see Section 3.2).

SNH model:
: An architectural view model that partitions an architecture into five different views: conceptual architecture, module interconnection architecture, execution architecture, code architecture and hardware architecture (see Section 1.2.2).

software architecture:
: A heavily overloaded term, which covers at least the main structures of a software system (see Section 1.2).

software architecture reconstruction:
: The process of recovering an existing software architecture, improving an existing software architecture and/or verifying the architecture of an existing system (see Section 2.5).

software architecting:
: The process of creating software architectures (see Section 1.2).

Summary

This thesis concerns *Software Architecture Reconstruction* of large embedded systems of the kind developed at Philips (MRI scanners, telephony switching systems, etc.). These systems typically consist of millions of lines of code[1], from which the first lines of code may have been developed more than fifteen years ago. A complete crew of software developers, typically sixty persons, continuously maintains and extends the system with new functionality.

Chapter 1 discusses the term software architecture. It concerns the design of simple and clear structures to be able to share software code amongs different products. A well designed architecture can be reused in dfifferent products. We must note that the term software architecture is somewhat ambigious. Therefore, we discuss a number of definitions and views on software architecture. Finally, a number of product's aims from a business perspective are discussed. For example, a product should be available in the market as soon as possible. The business goals determine the objectives of an architecture, for example possibilities to reuse parts of the software in other systems. On its turn these architectural objectives can be translated into architectural patterns.

In Chapter 2 we give an overview of the SAR (Software Architecture Reconstruction) method. First, we introduce terminology like software architecting (the construction of a new software architecture), reverse architecting (the process of making explicit the software architecture of an existing system) and architecture improvement. Next, the basic notions of the SAR method are discussed: InfoPacks, ArchiSpects and software architecture reconstruction levels. InfoPacks describe information extraction from software and ArchiSpects describe aspects of software architecture. A number

[1] To print a system of two million lines of code, we need a pile of 300 books of the size of this thesis.

	Architectural Views				
SAR levels	Logical View	Module View	Code View	Execution View	Hardware View
Optimized					
Managed		▶Generics and Specifics Conformance ▶Aspect Conformance ▶Usage Conformance ▶Layering Conformance	▶Aspect Assignment ▶Part-Of ▶Depend		
Redefined		▶Aspect Coupling ▶Cohesion and Coupling ▶Component Coupling	▶Aspect Assignment ▶Part-Of ▶Depend		
Described		▶Using and Used Interfaces ▶Component Dependency ▶Software Concepts Model	▶Source Code Organisation ▶Build Process ▶Part-Of ▶Depend ▶Files		
Initial					

Table S.1: Software Architecture Reconstruction

of InfoPacks and ArchiSpects can be defined for each architectural view. The following software architecture reconstruction levels are identified: initial, described, redefined, managed and optimized. A framework (with a focus on the module view and code view) of the SAR method is given in Table S.1. The cells of the table contain various InfoPacks and ArchiSpects discussed in this thesis.

In Chapter 3 Relation Partition Algebra (RPA) is discussed. Relation Partition Algebra is an algebra based on sets, binary relations and operations on them. Partitions play a special role in the algebra, which can be expressed in so-called part-of relations. In particular, dedicated operations

upon relations and part-of relations make the algebra very useful for software architecture. Multi-relations are an extension of binary relations, which were found to be very useful for architecture analysis. In the SAR method, we consequently apply RPA to describe the InfoPacks and ArchiSpects.

In Chapter 4 we show how the software architecture of an existing system can be described by defining a number of InfoPacks and ArchiSpects. The ArchiSpects *Source Code Organisation* and *Build Process* belong to the code view and the ArchiSpects *Software Concepts Model*, *Component Dependency* and *Using and Used Interface* belong to the module view of software architecture. The InfoPacks that are required to reconstruct these ArchiSpects are also discussed.

In Chapter 5 we describe a number of ArchiSpects that support the improvement (or redefinition) of an existing software architecture. The ArchiSpects *Component Coupling*, *Cohesion and Coupling* and *Aspect Coupling* are discussed. These ArchiSpects can be used by an architect to analyse the impact of the introduction of certain architectural changes.

In Chapter 6 we discuss architecture verification, i.e. the process of checking whether the implementation agrees with the defined software architecture. Therefore, we discuss the ArchiSpects (*Layering Conformance*, *Usage Control*, *Aspect Conformance*, and *Generics and Specifics*) that can help to manage a software architecture.

Chapter 7 contains some concluding remarks and recommendations.

The appendices contain the extraction, abstraction and presentation tools used to reconstruct and present ArchiSpects of a number of Philips systems. The last appendix contains an overview of the RPA operators.

Samenvatting

Niet alle lezers van dit proefschrift zullen weten wat software-architecturen zijn. We zullen dit begrip proberen uit te leggen door een vergelijking te maken met de architectuur van huizen.

> Het zal duidelijk zijn dat de architectuur van een wolkenkrabber (letterlijk) hemelhoog verschilt van die van een zomerhuisje. Een wolkenkrabber heeft natuurlijk een heel ander soort fundering nodig, maar ook grotere aan- en afvoerpijpen voor het water, er zal met andere bezoekersaantallen rekening moeten worden gehouden, enzovoorts.
>
> Toch wordt van softwaresystemen soms wel verwacht dat ze in toepassingen worden gebruikt van een geheel andere schaalgrootte. Dergelijke verwachtingen kunnen niet altijd worden waargemaakt. Een televisietoestel bijvoorbeeld heeft een gesloten architectuur, wat betekent dat nieuwe functionaliteiten, geschreven in software, via de kabel kunnen worden binnengehaald en als die met elkaar zijn verbonden kunnen gebruikers met elkaar communiceren.
>
> Als dergelijke productveranderingen niet zijn voorzien en de software-architect daarmee met zijn ontwerp geen rekening heeft gehouden, zal het systeem niet zijn voorbereid op toekomstige uitbreidingen. Een software-architect heeft dus ook tot taak om zoveel mogelijk rekening te houden met toekomstige uitbreidingen.
>
> Het is natuurlijk onpraktisch om de ontwerper van een zomerhuisje te laten werken met allerlei specificaties die gelden voor veel grotere gebouwen, en voor wolkenkrabber en zomerhuisje zal niet snel een gezamenlijke architectuur te ontwerpen zijn. Maar het zou al een hele verbetering zijn als de architectuur

> van de wolkenkrabber ook toepasbaar zou zijn voor een te ontwerpen flatgebouw, en andersom. Ook voor software geldt dat het om allerlei redenen, waarover hieronder meer, van belang is om bij de onderliggende architectuur rekening te houden met toekomstige uitbreidingen en andere toepassingen.

Dit proefschrift gaat over de reconstructie van software-architecturen van grote systemen zoals die door Philips worden ontwikkeld zoals MRI-scanners en telefooncentrales. De software van dit soort systemen bevat miljoenen regels code[1] waarvan de eerste code vaak al vijftien jaar oud is. Er is een groep van zo'n zestig software-ontwikkelaars nodig om het systeem te onderhouden en uit te breiden met nieuwe functies. Die inspanning kan aanzienlijk worden gereduceerd als software-architecturen zodanig worden ontworpen dat ze toepassing in vele systemen mogelijk maken.

Hoofdstuk 1 gaat over het begrip software-architectuur. Daarbij gaat het om het ontwerpen van duidelijke en eenvoudige algemene structuren die het mogelijk maken om softwarecode te delen met andere producten. Op die manier kan een productfamilie ontstaan. Een goed ontworpen architectuur kan voor vele producten worden gebruikt. Het begrip software-architectuur is overigens niet eenduidig. Er worden verschillende definities gehanteerd. Een aantal definities worden besproken, samen met een aantal gezichtspunten op software-architecture. Tenslotte worden een aantal doelen voor producten besproken vanuit bedrijfsperspectief. Zo'n doel is bijvoorbeeld dat het product snel voor de markt beschikbaar moet zijn. De bedrijfsdoelen bepalen uiteindelijk de eisen die aan de architectuur worden gesteld, zoals de mogelijkheid van hergebruik. Deze eisen zijn op hun beurt weer richtlijnen voor de architectuurpatronen.

In hoofdstuk 2 wordt een overzicht gegeven van de Software Architectuur Reconstructie (SAR) methode. Nadat termen worden uitgelegd zoals *software architecting* (de constructie van een nieuwe software-architectuur), *reverse architecting* (het proces van het expliciet maken van de software-architectuur van een bestaand systeem), en architectuurverbetering, worden de basisonderdelen van de SAR-methode besproken: InfoPacks, ArchiSpects en niveaus van software-architectuur reconstructie (*SAR levels*). InfoPacks beschrijven extractie van informatie uit software. ArchiSpects beschrijven aspecten van software-architectuur. Voor elk architectuurgezichtspunt kunnen andere InfoPacks en ArchiSpects worden gedefinieerd. De

[1] Het afdrukken van twee miljoen regels code levert een stapel op van 300 boeken met de omvang van van dit proefschrift.

	Architectural Views				
SAR levels	Logical View	Module View	Code View	Execution View	Hardware View
Optimized					
Managed		▶Generics and Specifics Conformance ▶Aspect Conformance ▶Usage Conformance ▶Layering Conformance	▶Aspect Assignment ▶Part-Of ▶Depend		
Redefined		▶Aspect Coupling ▶Cohesion and Coupling ▶Component Coupling	▶Aspect Assignment ▶Part-Of ▶Depend		
Described		▶Using and Used Interfaces ▶Component Dependency ▶Software Concepts Model	▶Source Code Organisation ▶Build Process ▶Part-Of ▶Depend ▶Files		
Initial					

Table T.1: Software-Architectuur Reconstructie

software-architectuur reconstruction *levels* die zijn gedefinieerd, zijn : initieel (*initial*), beschreven (*described*), opnieuw gedefinieerd (*redefined*), gecontroleerd (*managed*) en geoptimaliseerd (*optimized*). In tabel T.1 wordt het raamwerk van de SAR-methode (met een nadruk op de *module view* en de *code view*) getoond. De cellen in de tabel bevatten de InfoPacks en ArchiSpects zoals verwoord in dit proefschrift.

In hoofdstuk 3 wordt Relatie Partitie Algebra (RPA) behandeld. RPA is een algebra, gebaseerd op verzamelingen, binaire relaties en operaties die hierop plaats vinden. Partities, die een aparte rol spelen in deze algebra, kunnen worden uitgedrukt in "delen-van" relaties (*partof*-relations). Het

zijn vooral de speciale operaties op relaties en de *partof*-relaties die deze algebra geschikt maken voor software-architectuur. Multi-relaties zijn een uitbreiding op binaire relaties die uitstekend geschikt bleken voor architectuur analyse. In de SAR-methode wordt RPA systematisch toegepast om InfoPacks en ArchiSpects te beschrijven.

In hoofdstuk 4 wordt uitgelegd hoe een software-architectuur van een bestaand systeem kan worden beschreven door een aantal InfoPacks en ArchiSpects te definiëren. De ArchiSpects *Source Code Organisation* en *Build Process* behoren tot de *code view* en de ArchiSpects *Software Concepts Model*, *Component Dependency* en *Using and Used Interfaces* behoren tot de *module view* van software-architectuur. De InfoPacks die noodzakelijk zijn om ArchiSpects te reconstrueren worden ook besproken.

In hoofdstuk 5 worden een aantal ArchiSpects beschreven die het verbeteren (of opnieuw definiëren) van een bestaande software-architectuur ondersteunen. De ArchiSpects *Component Coupling*, *Cohesion and Coupling* and *Aspect Coupling* worden besproken. Deze ArchiSpects kunnen door een architect worden gebruikt om het effect van een bepaalde verandering in de architectuur te analyseren.

Hoofdstuk 6 gaat over architectuurverificatie, het proces om de implementatie te verifieren met de gedefinieerde software-architectuur. Er worden enkele ArchiSpects (*Layering Conformance*, *Usage Control*, *Aspect Conformance*, en *Generics and Specifics Conformance*) besproken die bijdragen aan het beheersen van de software-architectuur.

De dissertatie wordt afgesloten met enkele concluderende opmerkingen en aanbevelingen voor gebruik.

De appendices bevatten extractie-, abstractie- en presentatie- programma's die zijn gebruikt om ArchiSpects van een aantal Philips-systemen te reconstrueren en presenteren. De laatste appendix laat een overzicht van alle RPA-operatoren zien.

Curriculum Vitae

René Krikhaar was born in Nijmegen, the Netherlands, on January 30, 1963. In 1981, he finished secondary school (Voorbereidend Wetenschappelijk Onderwijs; pre-university education) at the *Canisius College* in Nijmegen. That same year he started to study Computer Science at the *Catholic University of Nijmegen*. He obtained his university degree ("doctorandus") in 1986. After fulfilling his military obligation in Bussum and Schaarsbergen, he started his career at *Philips Telecommunication and Data Systems* in Hilversum in 1987, in a group developing methods and tools for designing Application Specific Integrated Circuits (ASICs). In 1991, in his spare time, he started a second study at the *Centre of Knowledge Technology* in Utrecht in cooperation with the *Middlesex University* in Great Britain. This resulted in a degree of Master of Science with distinction in Knowledge Engineering in 1994. That same year, he joined the Information and Software Technology department of the *Philips Research Laboratories* in Eindhoven. In 1994 and 1995 he spent a few months working with a group of system designers who develop public *telephony switching systems* in Nuremberg, Germany. In 1999 he continued his career at *Philips Medical Systems* in Best, the Netherlands. He is currently a software architect of the Magnetic Resonance Imaging system.