TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Institute of Information Systems
Distributed Systems Group
Argentinierstrasse 8/184-1
A-1040 Vienna, Austria

http://www.infosys.tuwien.ac.at

Dissertation

View-based Software Architecture Reconstruction

Claudio Riva

October 2004

Advisor: Prof. Mehdi Jazayeri

Ai miei genitori.

KURZFASSUNG

Erfolgreiche Software Projekte legen besonders Wert auf ihre Software Architektur: die Strukturierung der Bestandteile und die Weise wie diese interagieren. Ohne einem passendem Architektur Design schlägt die Software Entwicklung fehl. Die Architektur Dokumentation vermittelt das gemeinsame Verständnis der wichtigen Design Entscheidungen, die während der Entwicklung getroffen wurden. Meist beschreiben mehrere Viewpoints die essentiellen archikturellen Belange der Stakeholders. Eine effektive Beschreibung der Architektur ist gut vermittelbar und wird von den Team-Mitgliedern gut verstanden.

Jedoch ist die Adaptierung eines Architektur-zentralen Entwicklungsansatzes in der Industrie weiter eine Herausforderung, da gute bewhrte Formalismen und Toolunsterstützung fehlen. So werden Architektur-Beschreibungen nur mittels informellen Notationen skizziert, so dass in der Praxis das Architektur Design kaum mit der aktuellen Implemenetierung übereinstimmt. Dadurch ergibt sich eine Kluft zwischen der *konzeptionellen* Software Architektur, welche in den Köpfen der Entwickler existiert, und der *konkreten* Architektur, welche sich in der Implementierung verbirgt. Desto mehr sich diese Kluft ausweitet, desto weniger wirkungsvoll ist die Software Architektur.

Architecture Reconstruction ist eine Methodik zur Gewinnung der Architektur Beschreibung aus den vorhandenen Hinweisen in der Implementierung. Die dafü am zuverlässigste Informationsquelle ist das System selbst, unter anderem der Source Code oder Execution Traces. Andere Quellen schliessen die vorhandenen Design Dokumentation, Domain Wissen und Interviews mit den System Experten mit ein.

Diese Dissertation präsentiert eine Methode zur Rekonstruktion der Architektur welche auf der Wiederherstellung von archikturell signifikanten Views beruht. Der Rekonstruktionsprozess von Nimeta betont die gewissenhafte Selektion (1) der architekturellen Konzepte als erstklassige Objekte der Wiederherstellung und (2) der signifikanten Views der Zielarchitektur, welche die Belange der Stakeholders reflektieren. Der von NIMETA verwendetet Formalismus basiert auf Relational Algebra zur Definion der Viewpoints, der Abstraktionsregeln und Automatisierung der Checks fr die Architektur-Übereinstimmung.

ABSTRACT

Successful software projects pay a careful attention to their software architecture: the structure of the constituent parts and the ways they interact. Without a properly designed architecture, the software development is likely to stumble. The architecture documents the shared understanding of the important design decisions taken during the development. Multiple viewpoints typically address the essential architectural concerns of the stakeholders. An effective architecture description is well-communicated and well-understood.

However, the industrial adoption of an architecture-centric development approach is still challenged by the lack of a well-established formalism and tool support. Architecture descriptions are sketched with informal notations and, in practice, the architectural designs are unlikely to conform with the actual implementation. This yields to a chasm between the *conceptual* (or intended) software architecture that exists in the minds of the developers and the *concrete* (or as-implemented) architecture that is hidden in the implementation. The more this chasm widens, the less effective the software architecture is.

Architecture reconstruction is the methodology for obtaining a documented architecture description from the available evidence of the implementation. The most reliable source of information is the system itself like the source code or execution traces. Other sources include the existing design documentation, domain knowledge and interviews with system experts. It is a reverse engineering process whose goal is to recover abstract views of the fundamental characteristics of the implementation.

This dissertation proposes a method, called NIMETA , for architecture reconstruction based on the recovery of the architecturally significant views. NIMETA 's reconstruction process emphasizes the scrupulous selection of (1) the architectural concepts as first-class objects of the recovery, and (2) the target architecturally significant views that are reflecting the stakeholders' interests. This results in a well-focused and well-scoped reconstruction method. NIMETA 's formalism is based on the binary relational algebra for defining the viewpoints, the abstraction rules and for automating the architectural conformance checking. We also present the supporting tool environment, NIMETA , and two industrial case studies.

ACKNOWLEDGMENTS

A Tao proverb says: the journey is the reward. Creating this dissertation has been a long journey that started six years ago. The reward comes from all the people I met along the way. I would like to thank all of them for their help, intriguing discussions and the nice moments. Since the list would have been too long, I hope you will accept my anonymous acknowledgements.

First of all, I would like to thank the Nokia Research Center in Helsinki where this work has been carried out. I am especially grateful to Christian del Rosso, Alessandro Maccari, Yaojin Yang, Jianli Xu for their help. Special thanks to Juha Kuusela, Aapo Rautiainen and Heikki Saikkonen for supporting my research activities in Nokia. I also would like to thank all my colleagues for providing an excellent research environment.

I would like to thank Professor Mehdi Jazayeri, my supervisor. Without his support and trust this work would have not been possible. I also would like to thank all the people at the Vienna University of Technology, especially Renate Weiss for helping me to cope with the bureaucracy.

I also would like to thank the board of the Nokia Foundation for the scholarship that helped me to finalize this dissertation.

Thanks to all my friends, especially the ”laiset” group in Helsinki. Without them, this journey would have probably been shorter but definitely less enjoyable.

Finally, I would like to thank my family and my girlfriend for their love and support.

Nessuna umana investigazione si può dimandare vera scienza, se essa non passa per le matematiche dimostrazioni; e se tu dirai che le scienze, che principiano e finiscono nella mente, abbiano verità, questo non si concede, ma si nega per molte ragioni; e prima, che in tali discorsi mentali non accade esperienza, senza la quale nulla dà di sé certezza.

Trattato della Pittura

Leonardo da Vinci, 1498.

TABLE OF CONTENTS

Kurzfassung **iii**

Abstract **v**

Acknowledgments **vii**

Table of Contents **xviii**

List of Figures **xxiii**

List of Tables **xxvi**

Abbreviations **xxvii**

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis 3

1.3 Approach 4

1.4 Contributions 5

1.5 Organization of the dissertation 6

2 Background **9**

2.1 Software artifacts 9
2.2 Software Architecture 10
2.2.1 Informal definition 10
2.2.2 Classical definitions 12
2.2.3 Definition from the IEEE 1471-2000 Standard 15
2.3 Views and Viewpoints 17
2.4 Product Family Development 18
2.4.1 Software Product Family Architecture 18
2.4.2 Product Family Evolution 20
2.4.3 Issues with the Product Family Evolution 23
2.5 Losing a Software Architecture 25
2.6 Reverse Engineering 26

3 Related Work **27**
3.1 Architecture Reconstruction 27
3.2 Formalisms for Architecture Reconstruction 33
3.3 Formalisms for Reverse Engineering 34
3.4 Dynamic Analysis 35

4 Problem Statement **37**
4.1 Intrdocution 37
4.2 Software Architecture Reconstruction 38
4.3 Novelty of the approach 41
4.4 Validation 43

5 Binary Relational Algebra 45

5.1 Introduction 45

5.2 Sets 46

5.3 Relations 46

5.4 Hierarchical Typed Graphs 50

5.5 Views and Viewpoints 53

5.6 Levels of precedence 54

6 The NIMETA Formalism 57

6.1 Introduction 57

6.2 Overview of the Viewpoints 57

6.3 The code viewpoint 60

6.4 The FAMIX design viewpoint 60

6.5 The architecture viewpoints 64

6.5.1 The Module Viewtype 66

6.5.2 The Component & Connector Viewtype 67

6.5.3 The Allocation Viewtype 69

6.5.4 The Feature viewpoints 70

7 The NIMETA Architecture Reconstruction Process 73

7.1 Introduction 73

7.2 Source, Target and Hypothetical Views 75

7.3 The Architecture Reconstruction Process 76

7.3.1 Problem Definition 78

7.3.2 Concept Determination 81

7.3.3 Data Gathering 83
7.3.4 Knowledge Inference 86
7.3.5 Presentation 88
7.3.6 Conformance checking 91
7.3.7 Architecture Assessment 92
7.3.8 Re-documentation 92
7.3.9 Iterations 93
7.4 Maturity levels of architecture reconstruction 94
7.5 Example 95

8 The NIMETA Tool Environment 103
8.1 Introduction 103
8.2 Overview of NIMETA 104
8.3 Extraction tools 107
8.4 Abstraction tools 108
8.4.1 Relational Algebra Engine 108
8.4.2 Prolog 111
8.5 Presentation tools 112
8.5.1 Rigi 112
8.5.2 Hava 114
8.5.3 Rational Rose 115
8.5.4 Web interface 118
8.5.5 SoftVis 119
8.5.6 ART environment 122

9 Case Study 1: NMP product family 125

9.1 Introduction . . . 125

9.2 Overview of the system . . . 126

9.3 Summary of the iterations . . . 128

9.4 The actors . . . 129

9.5 The First Iteration . . . 129

9.5.1 Problem Definition . . . 129

9.5.2 Concept Determination . . . 131

9.5.3 Data gathering . . . 132

9.5.4 Knowledge Inference . . . 132

9.5.5 Presentation . . . 132

9.6 The Second Iteration . . . 132

9.6.1 Problem Definition . . . 133

9.6.2 Concept Determination . . . 134

9.6.3 Data gathering . . . 137

9.6.4 Knowledge Inference . . . 138

9.6.5 Presentation . . . 139

9.7 The Third Iteration . . . 142

9.7.1 Problem Definition . . . 143

9.7.2 Concept Determination . . . 143

9.7.3 Data gathering . . . 149

9.7.4 Knowledge Inference . . . 151

9.7.5 Presentation . . . 155

9.7.6 Architecture Conformance Checking . . . 173

9.8 The Dynamic Analysis 180
9.8.1 Problem Definition 180
9.8.2 Concept Determination 180
9.8.3 Data gathering 181
9.8.4 Knowledge Inference 182
9.8.5 Presentation 182
9.9 Conclusions and Lessons Learned 183

10 Case Study 2: NMP platform 187

10.1 Introduction 187
10.2 Overview of the system 188
10.3 Summary of the iterations 189
10.4 The First Iteration 189
10.4.1 Problem Definition 189
10.4.2 Concept Determination 190
10.4.3 Data gathering 193
10.4.4 Knowledge Inference 194
10.4.5 Presentation 194
10.4.6 Architecture Assessment 198
10.5 The Second Iteration 199
10.5.1 Problem Definition 200
10.5.2 Concept Determination 201
10.5.3 Data gathering 201
10.5.4 Knowledge Inference 202
10.5.5 Presentation 203

10.6 The Third Iteration 203
10.6.1 Problem Definition 205
10.6.2 Concept Determination 207
10.6.3 Data gathering 210
10.6.4 Knowledge Inference 211
10.6.5 Presentation 212
10.7 Conclusions and Lessons Learned 213

11 Conclusions 215

11.1 Summary 215
11.2 Recommendations 216
11.3 Conclusion 217
11.4 Future Work 218

A Nimeta scripts 219

A.1 Extraction scripts 219
A.1.1 snav2nimeta.tcl 219
A.1.2 extract.py 229
A.1.3 strip-comments-file.pl 231
A.2 Abstraction scripts 232
A.2.1 Nimeta module 232
A.2.2 collapse_classes.rcl 236
A.2.3 collapse_dirs.rcl 236

References 239

Curriculum Vitæ et studiorum **251**

LIST OF FIGURES

2.1 Conceptual model of architectural description (source: the IEEE 1471-2000 Standard). 16

5.1 Directed graph representing the relation R. 47

5.2 An example of an hierarchical graph. 51

6.1 The framework of viewpoints. 59

6.2 The simplified FAMIX meta-model. 62

7.1 Views and Viewpoints. 76

7.2 The reconstruction process. 78

7.3 Intended architecture of Venice. 96

7.4 The viewpoints for reconstruction of Venice. 97

7.5 The class diagram of Venice. 99

7.6 Venice's class diagram with the relation *use*. 100

7.7 The top-level diagram of Venice. 100

7.8 The violations of Venice. 102

8.1 Overview of the NIMETA pipeline. 106

8.2 Overview of the NIMETA tool environment. 106

8.3 Example of grouping with RIGI 113

8.4 Overview of RIGI within NIMETA 114

8.5 Vertical and horizontal abstractions in HAVA 115

8.6 Scenarios of usage of HAVA 116

8.7 Overview of the web interface. 119

8.8 Overview of the web interface. 120

8.9 Multiple graph visualization in SoftVis. 121

8.10 Different layouts with SoftVis. 122

8.11 The stereotype definitions for the components (a) and for the dependencies (b). 124

8.12 Example of constraint definition for the architectural profile. 124

9.1 Simplification of the development process. 127

9.2 The summary of the iterations for the case study 1. 128

9.3 An example of the manually reconstructed logical view in UML. 133

9.4 Excerpt from the reference architecture. 135

9.5 The target viewpoint of the second iteration. 136

9.6 The source viewpoint for NPF. 136

9.7 The top-level dependencies in the target view. 139

9.8 The clients of one server. 140

9.9 The context diagram of one component. 141

9.10 The clients of one server. 142

9.11 The meta-model of the NPF case that shows the architectural concepts. . . 144

9.12 The target viewpoints. 146

9.13 The source viewpoint for programming language concepts and the code patterns of NPF. 147

9.14 The source viewpoint for the domain knowledge of NPF. 149

9.15 The overview of the web application. 158

9.16 Overview of the web interface. 159

9.17 The summary of the dependencies. 160

9.18 The detailed dependency table. 161

9.19 The memory management. 162

9.20 The context of memory management. 163

9.21 The top-level diagram for NPF. 164

9.22 The top-level diagram showing the content of the packages. 165

9.23 The content of the *Apps* package. 167

9.24 The content of the *Core* package. 168

9.25 The content of one package of NPF. 169

9.26 The content of one package of NPF. 170

9.27 The development view for NPF. 170

9.28 The geographical view for NPF. 171

9.29 The organizational view for NPF. 172

9.30 The task view for NPF. 173

9.31 The starting order of tasks for NPF. 174

9.32 The viewpoint and the view for the layering conformance checking. 177

9.33 The conformance diagram of one server. 179

9.34 The conformance diagram of one server. 180

9.35 The elements of the feature viewpoint. 181

9.36 The top-level feature view for the ”call release” feature (static diagram and sequence diagram). 183

9.37 The expanded component view. 184

9.38 The expanded sequence diagram. 184

9.39 The details of the call release feature between two applications and one server. 185

10.1 General architecture of the product based on the NNF platform. 191

10.2 MSC drawn during the workshop with the experts. 192

10.3 The viewpoints for the first iteration of NNP. 193

10.4 Top-level dependencies among the modules of the prodcut based on NNP. . 195

10.5 Dependencies between NNP classes and NOP servers in one module. . . . 196

10.6 Dependencies between NNP modules and NOP servers. 197

10.7 Dependencies between NNP modules and NOP servers. 197

10.8 The top-level dependencies between the packages under assessment. . . . 198

10.9 Class-level dependencies from the NNP GUI Library to the STD Graphic Library. 199

10.10 Class-level dependencies from the STD Graphic Library to the NNP GUI Library. 200

10.11 General structure of the NNP platform. 201

10.12 The top-level structure of the NNP platform. 203

10.13 Structure of the NNP platform. 204

10.14 Structure of the NNP platform. 205

10.15 Structure of the NNP platform. 206

10.16 The viewpoints for the third iteration of NNP. 208

10.17 The source viewpoint. 209

10.18 The context diagram of one subsystem. 213

LIST OF TABLES

2.1 The software artifacts and their levels of abstraction. 11

5.1 The levels of precedence for the relational algebra operands (from the most to the less bidding). 55

6.1 The entities and relations of the design viewpoint. 63

6.2 The architectural viewpoints from the catalogue. 65

6.3 The viewpoints of the module viewtype. 67

6.4 The viewpoints of the Component & Connector viewtype. 69

6.5 The viewpoints of the allocation viewtype. 70

8.1 The set operands in Python. 109

8.2 The relational operands in Python. 110

8.3 The mapping between the architectural concepts and UML. 117

9.1 The target viewpoints. 145

9.2 The relations of the source viewpoint. 148

9.3 Summary of the tested presentation tools. 157

10.1 The summary of the iterations for the second case study. 189

ABBREVIATIONS

ADL	Architecture Description Language
API	Application Programming Interface
ASR	Architecturally Significant Requirement
CASE	Computer Aided Software Engineering
CGI	Common Gateway Interface
DLL	Dynamic Link Library
DSP	Digital Signal Processor
GSM	Global System for Mobile communications (Groupe Special Mobile)
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HW	Hardware
IDE	Integrated Development Environment
LOC	Lines of Code
MB	Megabyte
MLOC	Million Lines of Code
MSC	Message Sequence Chart
OS	Operating System
PC	Personal Computer
PDA	Personal Digital Assistant
RAM	Random Access Memory
ROM	Read Only Memory
SDK	Software development kit
SW	Software
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
XML	Extensible Markup Language

CHAPTER 1

INTRODUCTION

A science is any discipline in which
the fool of this generation can go beyond the point
reached by the genius of the last generation.

- Max Gluckman

This dissertation develops a novel methodology called NIMETA for the reconstruction of the software architecture of an existing software system. The methodology includes the formalism for modeling the architectural views and the view-based reconstruction process. NIMETA 's main distinguishing characteristic concerns with the careful selection of the architecturally significant concepts to recover. In this chapter, we motivate its industrial importance for software development and we summarize our approach.

1.1 MOTIVATION

Successful software projects pay a careful attention to their software architecture: what are the constituents of the system and how they interact. Without a proper software architecture the software project is likely to stumble. Many companies have learnt this equation and they have introduced a proper software architecture design activity in their software development processes. The motivation for this dissertation originates from our industrial experience with software architecture modeling within Nokia[1].

Nokia is the world's largest mobile phone manufacturer and leading provider of telecommunication equipments. Nokia's origins are in the network and terminal equipments (e.g.

[1]Nokia Group: www.nokia.com

switching systems and cellular phones). Although network products contain large amount of software, the market is rather mature, consisting of a few big players with well-established development practises. On the contrary, in the recent years the terminal market underwent a radical revolution. Within one decade, the cellular terminals have shifted from simple single-lined devices to full-fledged mobile computing platforms supporting multimedia, gaming and office applications. The improvements in the hardware technology abruptly turned the terminals into *software intensive systems*. Differently from a decade ago when the intelligence was mainly located only in the telecommunication network, in the next generation cellular network (UMTS) the services are likely to be software–intensive and mainly located in the terminals.

This radical shift has revolutionized the product development process for the whole industry with the software playing a more important role than before. Nowadays, the software part is not only responsible for the implementation of the telecom protocols but also for the rich graphical user interfaces, services, games, multimedia and office applications, and third-party applications. All these software-related extensions represent an important added-value for creating compelling products. The quality of the software also influences the overall quality of the product and the end-user can directly perceive it through the user interface. Moreover, Nokia has become a mobile phone platform provider (e.g. the Nokia Series 60,80,90) for other phone manufactures. This places Nokia in competition with pure software companies like Microsoft.

Software architecture plays an important role in Nokia's software-intensive products. The size and the complexity of software systems requires robust software architectures that are well-designed in the first place and maintained heavily during the whole life-cycle. Nokia's products are typically organized in several product families where software assets are shared across similar products. Variance factors are caused by telecom protocols, operator customization, hardware platforms, user interfaces, end-user segment, and so on. Most of the product variations are handled by software variants derived from complex product family architectures. An effective architectural orchestration is required not only for achieving the desired quality levels but also for controlling the software development and evolution (like the introduction of new features).

In the past, not all the systems have been built with a well-documented architecture or the architecture documentation has not been maintained up to date. As a result, the developers often have a scant knowledge of the *as-implemented* architecture of their systems. Especially in the highly dynamic mobile phone domain, products are developed quickly to achieve a fast time-to-market and new features have to be introduced regularly in order to stay competitive. In just a few years, Nokia has established new software platforms that have been partly developed from scratch and partly by reusing existing implementations. To support the architectural control of these activities, software architects are often facing the challenge of not knowing carefully the real software architecture of the system they

architect. While the software architecture is recognized as an important factor of the success of the projects, the practise shows that there is a chasm between the *conceptual* (or intended, as–designed) software architecture in the minds of the architects and the *concrete* (or as–implemented) architecture that is found in the implementation.

The ultimate goal of this dissertation is to define a reconstruction process that can unveil the architectural aspects that are considered important by the developers and bridge the chasm between conceptual and concrete architectures. Over the past years, our team at the Nokia Research center has supported the Nokia business units on the architecture design of large software systems. During this period, we have exploited the NIMETA reconstruction process on several cases from which we have derived the two case studies presented in this dissertation.

1.2 THESIS

In this dissertation we explore the problem of defining a technique for recovering architecturally significant information from the implementation of a software system. Our goal is to enable the software architects to have a precise, consistent and detailed comprehension of the system they architect. We propose a methodology, called NIMETA , for software architecture reconstruction that is based on the recovery of architecturally significant views. It is a reverse engineering process whose goal is to recover a documented software architecture by examining the the available artifacts (such as source code, design documents) and by interviewing the domain experts. The reconstructed architecture is presented through a set of architectural views focused on specific architectural concerns and communicated to the developers. Our method stresses the importance of the careful selection of what architectural concepts to recover.

In this dissertation we address the following research problems:

- The description of the software architecture design is often conducted unsystematically and based on informal notations (textual or graphical). There is no universally accepted architecture modeling language (ADL) and the industrial support for an architecting tool is not adequate. This prevents an effective application of an architecture-centric development process. Although it is not our intention to propose another ADL, we have to face the problem that there is not a well-accepted formalism for architecture description. Nevertheless, we believe that a formal approach is required for architecture reconstruction. Our method is based on the binary relational algebra that has been successfully applied by the reverse engineering community.

- Software architecture documents the design decisions that are considered important by the architects for the development of the system. In this way, the content of the architecture description is dependent on the domain, on the system or by the interests of the architects. The main challenge is how to effectively select *what* useful information to recover for the architects. Our reconstruction process can be tailored to the architecture under reconstruction. We also need to consider how to merge heterogeneous sources of information.

- A well–designed architecture is futile if it is not well–communicated and well–understood by all the developers. One problem that we address is *how* to effectively communicate the reconstructed architectural views.

- The architects need to maintain the architecture description up to date. We must address the problem of automating the reconstruction process in order to deliver regular updates. Typically, the software is regularly built (daily, weekly or biweekly). The reconstruction process has to be able to deliver the reconstructed views at the same intervals and handle unfinished (and often un-compilable) code.

- The architects need to enforce their design decisions in the implementation of the system. Architecture reconstruction can automate of checking the conformance of the implementation against the design decisions. The architects want to avoid to integrate changes that break the integrity of the system.

1.3 APPROACH

We give a brief overview of the approach that we follow in this dissertation. The approach consists of a formalism for modeling architectural views, the reconstruction process, the NIMETA tool environment and two case studies.

First, we define the formalism for defining the viewpoints and the view transformations. We model the architectural views as hierarchical directed typed graphs. The binary relational algebra provides the formalism for the graph operations. From the existing literature we create a list of reference viewpoints that we formalize in terms of relational algebra. In particular, we select the FAMIX model (Demeyer *et al.* , 2001) for the design viewpoint and the architectural viewpoints from the book (Clements *et al.* , 2003).

Then, we define our architecture reconstruction process, called NIMETA . NIMETA follows a traditional reverse engineering process where the facts about the system are extracted from the implementation, transformed into abstract models and then presented in a particular textual or graphical format. NIMETA 's main distinguishing aspects are:

- The architectural concepts play a first-class role in the reconstruction process. The architectural concepts are the types of building blocks of the architecture. They are identified in the *concept determination* activity in conjunction with the architects. Since every software system built with a unique set of concepts, this operation allows us to focus the reconstruction and to tailor the process to the particular architecture of the system.

- NIMETA is based on the reconstruction of well–defined architectural views that have been agreed with the architects. In this way, the reconstruction is well motivated and scoped to the interests of the architects.

- The domain knowledge is modeled in the same way as the facts extracted from the system. This permits a seamless integration during the reconstruction.

NIMETA 's reconstruction process consists of three phases: process design, view recovery and result interpretation. Each phase consists of several activities. The first five activities are essential for recovering the software architecture. The objective of the first activity, *problem definition*, is to define the goals and the expected results of the reconstruction. The second activity, *concept determination*, aims at recovering the architectural concepts and defining the target viewpoints that are needed to solve the identified problems. The third activity, *data gathering*, is focused on extracting the facts from the sources of information and creating the source view. The goal of the fourth activity, *knowledge inference*, is to infer the target architectural views from the source view. The fifth activity, *presentation* is about presenting and communicating the architectural views.

Then, we propose the the NIMETA tool environment. It consists of an integrated collection of tools that can assist the reconstructor during the whole process. It allows the reconstructor to create a pipeline of tools to automate the whole reconstruction process.

Finally, we validate our reconstruction process with two case studies that have been conducted with two different Nokia business units. The first case study demonstrates the development of a complete reconstruction process for a Nokia product family. The reconstruction process is largely automated and it is currently in use by the business units. The second case study demonstrates how architecture reconstruction assisted the development of a new software platform.

1.4 CONTRIBUTIONS

The main contributions of this dissertation are:

1. We provide a general and *expressive formalism* for modeling the architectural views, as they are intended by the software architecture practitioners.

2. We consider the *architectural concepts as first-class elements* of the reconstruction. In this way, we can focus and tailor the reconstruction process to the architecture of the system.

3. We provide a well–scoped reconstruction process that aims at the reconstruction of the target viewpoints that are agreed with the architects.

4. We define a set of reference viewpoints in relational algebra to support architecture reconstruction.

5. We provide the framework for establishing the architecture conformance checking against the intended design and the architectural rules.

6. We involve the stakeholders and the domain experts in the reconstruction process.

7. We illustrate the reconstruction process on two large case studies.

8. We consider the dynamic analysis at the architectural level for the reconstruction of the feature implementation.

1.5 ORGANIZATION OF THE DISSERTATION

Chapter 2 summarizes the background information for understanding this dissertation. Besides the general definition of software architecture in Section 2.2, the definition of views and viewpoints are particularly important for understanding the reconstruction process. Table 2.1 clarifies the software artifacts that are involved in the reconstruction. The Section 2.4, Section 2.5 and Section 2.6 introduce the main concepts about product family, architecture decay and reverse engineering. Chapter 3 reviews the existing approaches for architecture reconstruction (Section 3.1), the formalisms for architecture reconstruction Section 3.2 and reverse engineering Section 3.3 and the approaches for dynamic analysis Section 3.4. Chapter 4 introduces the main problem that is addressed by this dissertation and our approach. Chapter 5 presents the mathematical foundations of the NIMETA approach: binary relational algebra. Chapter 6 formalize the basic design and architectural viewpoints that are necessary for the NIMETA approach. This consists of the NIMETA formalism. Chapter 7 presents in details the NIMETA reconstruction process, including an explanatory example. In Section 7.4 we also define six levels of maturity for the reconstruction process. Chapter 8 gives an overview of the tools that belong to the NIMETA tool environment. The two Nokia case studies are presented in the Chapter 9 and Chapter 10.

Chapter 11 concludes the dissertation and presents the future work. The appendixes contain the implementation of various scripts that we have used.

CHAPTER 2

BACKGROUND

Programs, like people, get old.

- David Lorge Parnas

This chapter provides the background on the research topics that are relevant to this dissertation. We start with an overview of the typical software artifacts and, in particular, of the term software architecture. Then, we introduce the concept of view and viewpoints that are necessary for defining the architecture views. Then, we introduce the software product families focusing on their general architecture and evolution patterns. We also discuss the reasons for losing an architecture description. We define the common terms for reverse engineering.

2.1 SOFTWARE ARTIFACTS

Forward engineering is the traditional process of moving from high-level abstractions to a physical implementation of the system. Through this process, an initial requirements are turned into the system specifications that are ultimately implemented in a programming language. It is a process where more and more details are added in order to get a software program that can be executed. Reverse engineering is the inverse process where the program-level details are removed in order to unveil the original abstractions. In this section, we clarify the different levels of abstractions for the typical software artifacts. (Eden and Kazman, 2003) have also defined a similar framework as the one that we present here.

We distinguish between the problem domain and the solution domain. the *problem domain* is focused on the user's perspective and describes what the software system is supposed to

achieve. The problem domain consists of artifacts concerning the domain model and the system requirements. The *solution domain* is focused on the developer's perspective and describes how the system achieves its promises. The artifacts concerning the architecture, the design and the code belong to the solution domain. The features represent the contact point between the problem and the solution domain. We give a detailed description in the following Table 2.1.

2.2 SOFTWARE ARCHITECTURE

There is no universally-accepted definition of the term *software architecture*, even though there is an IEEE standard definition (P1471, 2000), countless definitions from researchers and practitioners [1] and it is a rather overloaded word. Besides the classical definitions, we can argue every development team, software organization or research institute has its own interpretation of what a software architecture represents. In this section we present a brief overview of software architecture from different perspective.

2.2.1 INFORMAL DEFINITION

We start with an informal definition of software architecture based on a short article of M. Fowler (Fowler, 2003). Fowler's main point is that software architecture represents what the expert developers in a team believe to be *important* for the system design. In other words, the architecture is the shared understanding of the important matters of the system design as it is perceived by the expert developers. A particular component is architecturally significant if the expert developers believe so. Based on this interpretation, the architect is the expert developer in the team who is "very aware of what's going on in the project, looking out for important issues and tackling them before become a serious problem" (Fowler, 2003). The architect is able to have a broad view on the design from its code to its requirements and to communicate with different people (programmers, testers, designers, project managers and requirements engineers). The architect often represents a bottleneck where contrasting issues and viewpoints converge. The role of the architect is also to simplify other's people work, for example, by facilitating the communication, minimizing the disorder and controlling the complexity of the design. One interesting point of Fowler is that the architect's role is to minimize architect's need in the development. In the ideal situation there is no need for the architect as changes are easy and they can simply happen. In the

[1] The Software Engineering Instituted (SEI) collected tens of definitions from various authors at their website: http://www.sei.cmu.edu/architecture/definitions.html

	Type of Artifact	Description
Problem Domain	**Domain model**	It models a particular domain of a product family. It typically describes the commonality and variability of the requirements within that particular domain. The domain model gives a overview of the main concepts that are expected by the user and their interrelationships.
	Requirements	The requirements define the users' goals that can be achieved by using the software system. There are two types of requirements: functional and non-functional. For a product family, the general requirements of a product can be based on the domain model.
	Features	There are several interpretations of the term "feature" and it is often domain dependent. We follow the definition of (Turner *et al.* , 1999): a *feature* "is a coherent and identifiable bundle of system functionality". The features represent the contact point between the solution and problem domain. In the problem domain, features represent what the user is willing to pay for, something that can be concretely marketed and is visible in user interface. At least in the telecommunication domain, the term "feature" has a clear connotation. Examples are: make a phone call, send a SMS, add a contact in the phonebook and so on. In the solution domain, features represent what the developers must implement in order to make the software system to fulfill its requirements. Features often represent a common and unifying langauge that is well-understood by a variety of people: managers, UI specifiers, analysts, architects, designers, programmers, testers and so on.
Solution Domain	**Architecture**	The architecture is the collection of design decisions that enable to implement the features (functional requirements) and to satisfy the quality requirements of the system. We have given a detailed definition in Section 2.2. In this context, we want to clarify the distinction between architecture and design. Quoting (Clements *et al.* , 2003): "Architecture is design, but not all the the design is architecture". The architecture establishes certain constraints but does not define the implementation. Many design decisions are left open and are judged by the downstream designers. As an example, the architecture defines the interfaces and constraints of the components but it might not fully define the internal design of those components.
	Design	At this level, the designers provide a clear design of the architectural elements using a particular formalism. For example, in the case of the object oriented design, the architectural elements are described in terms of classes, methods, abstract data types and their interactions.
	Code	At the lowest level of abstraction, there is the source code that is implemented with a particular programming language. The source code is the richest artifact in terms of details.

Table 2.1: The software artifacts and their levels of abstraction.

real situations, making the software easy to change introduce complexity that the architects have to minimize. The work that we present in this dissertation represent an effort to recover parts of the shared understanding that the architects believe is important for the life of the system. (Holt, 2001) present a software architecture as a shared mental model about the important decisions for the development.

2.2.2 CLASSICAL DEFINITIONS

The classical definitions of software architecture include the notions of *components*, their *external properties*, *relationships* among them and *constraints* (Bass *et al.* , 1998; Perry and Wolf, 1992; Booch *et al.* , 1999; Shaw and Garlan, 1996; Garlan and Perry, 1995; Gacek *et al.* , 1995; Hofmeister *et al.* , 2000). Some definitions also include the term *rationale* that describes the justification for the architecture to exist. Through this dissertation we adopt the practical definitions that was elaborated during the European project ARES[2] (Jazayeri *et al.* , 2000).

The description of the software architecture should communicate the essential decisions that have been taken in the design of the software system. The essential decisions of a design are the ones that are expensive to change and, therefore, the most critical for the development and maintenance of a system. (Ran, 2000) gives the following definition:

> **Software architecture** is a set of concepts and design decisions about structure and texture of software that must be made prior to concurrent engineering to enable effective satisfaction of architecturally significant explicit functional and quality requirements, and implicit requirements presented by the the problem and the solution domains (Ran, 2000).

First, architecture is a set of concepts that we use to think of the software system. The concepts are used for the design and the implementation of the system. They represent the bridge between the requirements and the implementation. Second, software systems are built to perform certain operations (functional requirements) with a certain level of quality (quality requirements) in a particular environment. The purpose of the software architecture is to *enable* the satisfaction of the requirements (although it cannot guarantee their satisfaction). Not all the requirements are architecturally significant. As a general rule we can say that the bigger is the impact of a requirement (and related design decisions) on

[2]The project ARES (Architectural Reasoning for Embedded Systems) was an Esprit framework IV project (no. 20477) whose main objective was to enable software developers to explicitly describe, assess, and manage architectures of embedded software families

the system the more architecturally significant it is. As an example, a strict requirement on performance may affect the design of many parts of the system. This helps to define the scope between architecture and design. The detailed design of a component is not as architecturally relevant as component's interaction with its environment. Though such decisions can influence the satisfaction of important requirements, they do not influence the design of other components. Third, some design decisions concern with the software structure and the texture. We discuss the different types of design decisions in the following sections.

Architecturally Significant Requirements

Not all the requirements play an important role in the architecture of the system but only a fraction of them. If they are detected early enough, they can be addressed by a properly designed architecture. Architecturally significant requirements (ASRs) have a major impact on the design of the system and if they are ignored the system will suffer major deficiencies. For example, mobile phones have strict real-time requirements on the communication activities with the base stations. As they represent the major concern in the architecture, the ASRs influence the design of the architectural concepts and their properties. Understanding the ASRs allow us to understand the rationale behind those design decisions.

Architectural Concepts

Before describing the overall structure of the system, we need to define what types of concepts are permitted for the construction of the system. An architectural concept represents the basic type of building block that provides a well-defined behavior and specific properties. The conceptual architecture communicates the rules for using the architectural concepts. The architectural types are not derived by the requirements but they are invented by the architects to simplify the task of bridging the gap between the requirements and the implementation. Of course, the architectural concepts must be properly designed in order to permit the implementation of the requirements. Every system has its own set of architectural types, as systems are built according to different architectural styles. For example, in a distributed software system the architectural concepts may be applications, servers, and software busses, while in an operating system they may be tasks, processes, queues, shared memories, etc. In general, the conceptual architecture is a model the important concepts used for the design of the software, including their properties and relationships. The conceptual architecture also defines how the architectural concepts maps to the implementation platform.

ARCHITECTURAL STRUCTURE

An architectural structure defines how the software is partitioned into components or units for a particular architectural domain. There are different architectural domains (or architectural planes) for the different stages of the software life-cycle (run-time, design-time, implementation-time, build-time, configuration-time). Architectural domains are not different views of the same aspect. Rather, they represent different things. Software can be seen as a set of modules, a set of processes, a set of executable, a set of code files and so on; Hence, a software architecture consists of multiple structures, each describing a particular architectural domain. Partitions allow us to separate the various concerns within a particular structure and good partitions allow us to localize the effects of requirements changes in a limited set of components. Each domain addresses a particular set of ASRs. For example, the run-time domain is concerned about the execution structure, hence about the performance, availability and reliability requirements.

We note that architectural domains are different from architectural views. Architectural domains contain separate entities (a set of modules, a set of processes, a set of executables, a set of source files and so on), even though they are strictly dependent. The architectural views are abstractions of the architectural structures for describing specific concerns (for example, subsystem organization, layering, work division). Architectural views are the means for describing and analyzing various aspects of the underlying architectural structures. Throughout the dissertation we mostly describe the architectures through their architectural views, assuming that they represent a projection and an abstraction of a particular architectural structure.

TEXTURE

The texture represents the recurring and unform microstructure of the software components like design patterns, policies, coordination, styles, standards, component models, programming languages, naming conventions and so on. Texture contain all those local design decisions that have an architectural relevance because they are spread in many parts of the system. Well-designed software has a consistent texture and a mechanism to check its consistency.

2.2.3 DEFINITION FROM THE IEEE 1471-2000 STANDARD

The IEEE 1471 Standard (P1471, 2000) establishes a conceptual framework of concepts and terms of references for the architectural description of software systems. Although there is no reliable consensus on a precise definition of software architecture, the IEEE 1471 is a recommended practice that codifies those elements on which there is a consensus. Throughout this dissertation, we try to conform to the definitions given in the IEEE 1471.

The diagram in Figure 2.1 shows the conceptual model for the architecture description. The IEEE 1471 gives the following definition of software architecture:

> A **software architecture** is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution (P1471, 2000).

The environment consists of stakeholders and concerns based on the following definitions:

> A system **stakeholder** is an individual, team, or organization with interests in, or concerns relative to, a system (P1471, 2000).
>
> A **concern** is an interest which pertains to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders. Concerns include system considerations as performance, reliability, security, distribution, and evolvability (P1471, 2000).

The IEEE 1471 definition covers a variety of uses for the term *architecture*. It clearly highlights that an architecture embodies the *fundamental* things about a system; "fundamental things" must be interpreted in the sense they are fundamental and important for the stakeholders, and with respect to the system's environment. Similarly to the previous definitions, it stresses the importance of focusing architecture description on the design aspects that are perceived significant by the stakeholders. In certain cases, they are the physical components and their relationships; in other cases, they are the logical relationships. Views and viewpoints play an important role in the IEEE 1471 for creating an architectural description. We discuss them separately in the following Section 2.3.

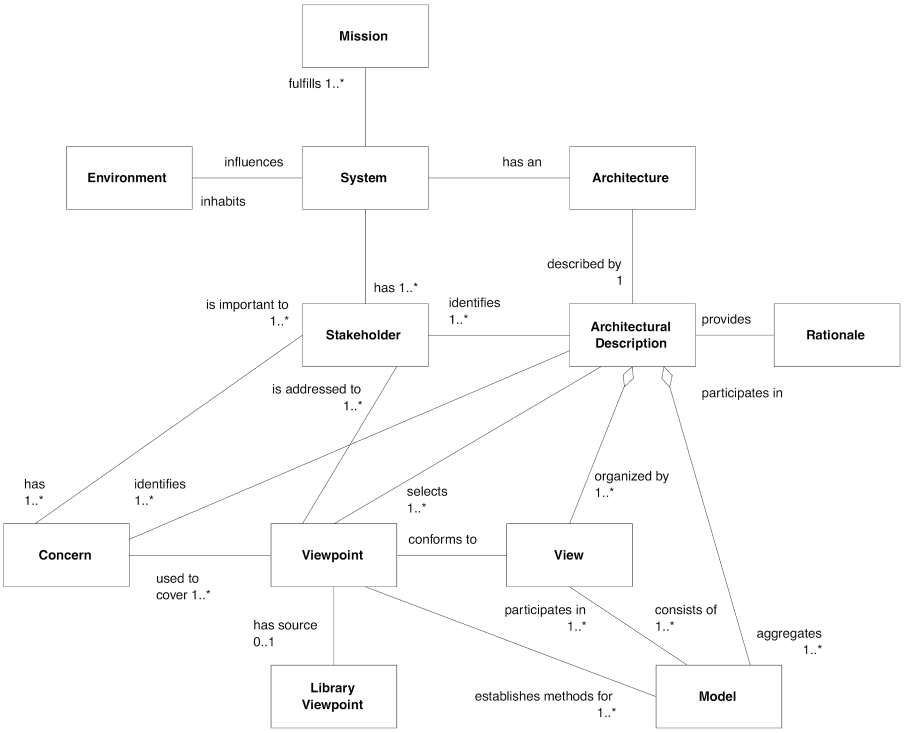


Figure 2.1: Conceptual model of architectural description (source: the IEEE 1471-2000 Standard).

2.3 VIEWS AND VIEWPOINTS

A software system is a complex entity that cannot be described with one single dimension but it requires multiple perspectives (Jazayeri *et al.* , 2000; Clements *et al.* , 2003) . Architectural views address particular concerns of the architecture design and explore different quality attributes of the design. Each view emphasizes certain aspects, while ignoring others in the interest of making understandable a complex system.

We follow the definitions from the IEEE 1471 Standard. A software architecture is documented by a set of **architectural descriptions** that represent concrete artifacts. Each architectural description is organized into one or more views, often called architectural views. A view conforms to a particular viewpoint that describe its content and usage. The standard definitions are:

> A **view** is a representation of a whole system from the perspective of a related set of concerns (P1471, 2000).
>
> A **viewpoint** is a specification of the conventions for constructing and using a views. A pattern or a template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis (P1471, 2000).

In our interpretation a viewpoint defines the semantics of the content and usage of a view. A view is primarily composed of models that provide the specific description of an architecture. The viewpoint defines the language, the modeling technique and the analytical methods for the construction of the views. Viewpoints cover the concerns that are considered fundamental by the stakeholders.

The choice of what views to document depends on the environment and the interests of the stakeholders, therefore we cannot impose a general collection of views that suit all the software systems. The history of architectural views goes back to the concept of *hierarchical structure* found in the work of (Parnas, 1974) on operating systems. Parnas identified several types of structures like program hierarchy, process hierarchy, resource allocation hierarchy, protection hierarchy and module hierarchy. Recently, Perry and Wolf recognized the need for multiple architectural views emphasizing aspects that are important to the stakeholders (Perry and Wolf, 1992). In 1995, P. Kruchten proposed the ”4+1” model for architecture description (Kruchten, 1995). The model distinguishes four different views: logical view, process view, development view, physical view. The fifth view, the use cases, ties together the other four views. (Hofmeister *et al.* , 2000) also propose four different architectural views they have observed in the industrial practice: conceptual architecture,

module interconnection architecture, execution architecture and code architecture. The recently published book of (Clements *et al.* , 2003) about documenting software architecture contains a collection of well–established architectural views for architectural description. We take those views as a reference collection for our formalism as presented in Chapter 6.

2.4 PRODUCT FAMILY DEVELOPMENT

The concept of software product family originates from the hardware industry where hardware product lines[3] enable the production of numerous variants of products and a significative reduction of operational costs by sharing most of the assets. In the recent years the software has become a dominant part in an increasing number of embedded products and it is often affecting the quality and the delivery time of the products. We can estimate that most of the delays in the release of embedded products is due to software rather than hardware faults. To cope with the multitude of software variants required by an industrial product line, the software assets have been organized in software product families and, thus, the paradigm of product line has been transferred to the software embedded in the products. This paradigm shift has happened for most of industries producing embedded products (like cars, consumer electronics and mobile phones) where they need to deliver a customized software system for the various products. The term *software product family* indicates a collection of products that share common requirements, features, architectural concepts, and code, typically in the form of software components. For a detailed definition of product line we refer to the book of (Bosch, 2002). In the following two sections, we present a typical example of a Nokia's product family architecture and we discuss the issues related to its evolution. This section is based on our previous work (Riva and Del Rosso, 2003).

2.4.1 SOFTWARE PRODUCT FAMILY ARCHITECTURE

The main goal of the product family architecture is to describe the commonality and variability of the family in order to make explicit the variation points of the products. We use a conceptual hierarchial framework for describing the elements of the product family architecture. We can identify at least four layers of genericity: the *reference family architecture* layer, the *family architecture* layer, the *lead product architecture* layer and the *copy product* layer.

[3] In this dissertation software product line and software product family can be considered synonymous.

The *reference family architecture* describes the global architectural style that is valid for all the products of the family: architectural significant requirements, architectural rules, patterns, component types, communication infrastructure, runtime issues. The architects can derive the software architecture for the product families from the reference family architecture.

The mobile phones are grouped into product families according to the UI styles, features, telecom standards and hardware generations. This represents a first level of variations where the products are grouped in macro-families (e.g. GSM, TDMA or UMTS). For each product family, the *family architecture* describes the services and features that are available in the platform. They may include the protocol stacks, the OS, the UI kernel, basic applications and hardware drivers.

Each family contains a reference product implementation that we indicate as *lead product architecture*. This product is considered to be the most typical one of the family. It is derived from the family by copying the common elements from the family architecture and by instantiating the abstract elements. The purpose of the lead product architecture is to provide a reference architecture for the other products and to clearly document the variations points available within the family.

At the bottom of the hierarchy, there is the *copy product*. This is typically copied from the lead product and adapted to the specific product requirements. This represents the final product architecture and it is the starting point for the development project (mainly focused on feature configuration, integration and testing).

The reference architecture describes the architectural style of the family and typically contains the information that everyone needs to know about the style, rules, conventions used in the family. Changes in the reference architecture are costly and happen through a careful assessment of their architectural impact.

The description of the reference architecture is a vital asset for a product family. A product family includes products that are built upon the same architectural style. Usually, every product that belongs to a certain family is built according to similar architectural rules. The style includes all such rules that are relevant at the architecture level. Examples of such rules may be ”components must be either clients or servers”, and ”the communication between clients and servers should happen by means of asynchronous messages that conform to a certain format”. The reference architecture document describes the architectural rules that hold for every product that is part of a certain family. It can be thought of as the basis for the architectures of the products. The architecture of every single product of that family is an instantiation (or a specialisation) of the family reference architecture. The reference architecture document should contain at least the parts described in the following sections:

Architecturally significant requirements Architecturally significant requirements include general requirements for the whole family, plus what we call lifetime requirements. These are requirements that must be satisfied at different stages of the software development, and concern different instances of the software. For example, at design time the software is made of logical components, at write time it is made of modules and at run-time it is made of tasks and threads. The requirements for all these phases that concern all the products that are part of a certain family should be described in the document. We use mostly unstructured English text for this, although we are experimenting formal notations for specific issues (as the Petri Nets for describing feature interaction (Lorentsen *et al.* , 2001)). Usually, architecturally significant requirements are rather stable.

Architectural rules This section of the document contains the system-level rules that all products in the family must conform to. Rules may describe what types of components (modules, entities and subsystems) may exist and what kinds of relationships are allowed between the different types of components. When implementing a new feature, developers should create components that conform to the architectural rules specified in this section. An example of our reference architecture can be found in (Kuusela, 1999). Currently, we use UML to describe architectural rules, although we find that this notation has some shortcomings for this purpose (Riva *et al.* , 2001).

Communication infrastructure The communication infrastructure is the means used by the various software components (modules, entities, subsystems) to communicate at runtime. The description of the infrastructure is crucial, as it fixes the structuring of interfaces and the communication paradigm. The effort spent on accurate description of the interfaces seems to pay back during the integration testing phase.

Runtime architecture Runtime architecture concerns with the allocation of processes to threads; the division of tasks at the operating system level; the interaction of different features at system level.

2.4.2 PRODUCT FAMILY EVOLUTION

Software product families are rarely created right away but they emerge when the domain is mature enough to sustain the long-term investments. The typical pattern is to start with a small set of products (often just one). If the business starts to generate profits or looks profitable in the future, new products are introduced in the market. New products are typically copy pasted versions of the existing ones with some additional new features. Most of the differences are achieved at the software level, while the hardware platform remains quite unchanged. On the wave of success, the software embedded in the products becomes a

global asset that becomes in use in several sites worldwide at the same time. Many sites embed the same software in their local products and often make their own local modifications (e.g. customization, updates, patches). As soon as the business becomes more mature, new investments are needed for consolidating the software assets. At this point, the various set of products are migrated towards a product family in order to keep all the software variants under control. The migration process affects the software parts and the organization as well. The organization needs to adjust its operating procedures to support the global management of the products lifecycle (from requirements engineering to testing). The software variants have to turn into a flexible platform where the products of the family can be derived from in a more flexible way. We can identify five different patterns or approaches that appear at different stages of the evolution:

copy/paste: the software variants are created by copying and modifying the existing products. It is the fastest approach for creating a new product and it is typically used when the organization is entering or creating a new business. All the resources are focused in the implementation of new features without a strict control redundancy that the variants create. This approach minimizes the risks of development but it maximize the entropy. This approach gives the highest flexibility for creating new products and entering in a new market. Verhoef et al. describes this process as *software mitosis* (Faust and Verhoef, 2003).

configuration: The variability is embedded in the software with a set of configuration parameters. The parameters allow to enable/disable parts of the code, to select particular algorithms and to configure the modules. Although the method is simple and allows to create numerous variants from a small set of bases, it has several drawbacks concerning the maintainability and evolvability of the code.

component-based: the variable functionality of the software is factored into separate software components and assigned to different development teams with a clear separation of concerns. The variants are achieved by plugging different components into a common software framework. The granularity of the components can range from single classes to entire subsystems. The correct granularity is often a trade-off between the flexibility/maintainability: few large components reuse more software but are harder to compose and maintain, small components might embed too little functionality. The correct size is often reached after some time. The component-based approach has an impact on most of the software engineering activities of the organization, especially for the integration phase of the components in the products. The component-based approach can be considered a key milestone towards a flexible product family.

platform: the concept of software platform emerges when the organization starts to consolidate its experience in a mature domain. The goal is to maximize the reuse of the software components among the products and the throughput of the family. The platform provides a cohesive set of services, libraries, software components and product

frameworks that are used for building the products. The basic services (e.g. telecom protocols, hardware drivers, graphical libraries, common applications) become globally available in a precise and controlled way. But that's not all. The platform becomes a well-defined entity in the organization with its release plan, roadmap for the new features, coding conventions, idioms, testing procedures, architectural documents, training material. The development teams are organized in a matrix structure. On one dimension there are the *component factories* and the *platform management* team responsible for the development of the software components and for the maintenance of the platform. On the other dimension there is the *product development* organization responsible for the development of the products. The crucial phase of the product development is the integration where the different components have to be integrated in the coherent way.

optimized platform: the optimized platform tries to overcome with the integration problems of the platform approach. Most of the resources are spent during the integration of the platform components when architectural mismatches or bugs have to be carefully analyzed and solved by the component owners. The optimized platform solves this problem by enabling the feature-based derivation of the products. The platform offers a rich set of configurable features. The product integrator selects and configures the features to be included in the product and the real integration is automatically achieved by the platform. This has been envisioned in our previous work (Maccari and Riva, 2001).

In real product families the five approaches can coexist at different extent. In a highly dynamic domain, the product family is more directed in the direction of copy/paste approach that offers the fastest time-to-marker. In a stable domain, the platform approach is a better choice because it maximizes the consolidation of the assets. Verhoef et al. describes the same concept using the *grow and prune model* (Faust and Verhoef, 2003). The product family is typically oscillating between grow and prune phases. In the grow phase, the product family is free of exploiting new opportunities without much architectural governance using the copy/paste approach (this leads to the software mitosis phenomena causing a large increase of clones and variants). In the prune phase, the weak branches of the family are removed and the successful products are re-organized in order to be consolidated in the family (re-balancing the robustness and the governance that was lost during the mitosis phase).

(Bosch, 2002) has also proposed a similar structure for the maturity levels of software product lines: independent products, standardized infrastructure, platform, software product line and configurable product base. The different levels represent an evolution of the management of software variability.

2.4.3 Issues with the Product Family Evolution

In this section, we discuss several problems that typically concern the evolution of a product family:

- Increasing Bureaucracy

  The migration towards a product family is a process that introduces bureaucracy in the organization. The software process becomes more complex due to the introduction of new procedures that have to be followed when creating a new product or modifying the platform. Differently from the uncontrolled growing phase, changes have to be well documented and motivated. It also emerges a new hierarchy of managers, architects, feature owners, component designers that are responsible for preserving the integrity of the product family architecture and for approving the changes. Enforcing the architectural governance requires a certain level of bureaucracy but this is also a threat for the flexibility of the family. This tendency towards stiffness is often opposed by practices that increase the flow of communication among different teams, for example by introducing architects that are responsible for heterogenous technology areas.

- Slow process of change

  There are cases when the change requests for new features have to go through a long approval process. If we consider the four-layer architecture of the Section 2.4.1, a typical scenario is the following. A new feature is detect by the product development team at the lowest level. If it is a local feature and it does not have an impact on the family, it is just implemented in the local product. If it has a possible impact on the family, the feature has to be passed over and over to higher levels where its impact is carefully assessed. In the worst case a change request may reach the reference architecture level. This happens when the new feature requires a critical change at the core of the family (for instance, adding a streaming video functionality might require changed in the operating system). At some point the change request may be reject or delayed to avoid the negative impact that its implementation would have on the architecture. A slow process of change is an inevitable drawback for avoiding features that could break the architectural integrity of the family.

- Over-designed platform

  The design of a new software platform is a long-term activity where considerable resources are spent for designing a generic-enough platform to support the long-term evolution of the product family. There is often the risk of designing a platform that is too generic for what is really needed by the products. There is a sort of auto-inducted tendency of searching for the best software design that can handle all the possible

situations. This often leads to the creation of far too complex software frameworks that are very difficult to instantiate. This tendency should be limited and the design activity should investigate the good-enough architectures rather than the best solutions.

- Spaghetti dependency

  A main goal of a product family is to share software among several products. Since the owners of the software components (i.e. the component factories) and the users of these components (i.e. the product development teams) are different, for each product there is an inevitable network of dependencies. Common problems are: the interfaces of the components change without notice, long queues for the change request of widely used components, the clients of the components are unknown (the dependencies are often only visible in the code). Moreover, software dependencies can be easily mapped to human interactions among different development teams and in a multi-site geographically distributed environment managing these interactions is a challenge. Minimizing and controlling the software dependencies of the family is a key activity for the organization.

- Feature reallocation

  In the typical scenario the features of the product family architecture are instantiated in the specific product architecture. However, during the consolidation phase the features in the products can be re-allocated to the platform. In this case, it is necessary to move the implementation of the feature out of the product and integrated it with the platform. This often happens when a feature that has been exploited in one product has been successful and, thus, other products want to use it. In this process, we need to ensure that the feature can be supported in the entire family.

- Cross-family reuse

  There are cases when it is necessary to share software components among different product families for reducing development costs (for example, when migrating one product family to the latest hardware that is already in used by another family). The first problem is that there can be architectural mismatches among the families (e.g. different operating systems) and these differences have to be assessed. The second problem concern the ownership of the common software. In many cases, it is possible that the product family has little influence on the software development somewhere else. This situation often leads to a long integration.

- Introduction of new requirements

  In a dynamic market it is critical to handle the forthcoming requirement in time. Even though the problem of incorporating new requirements is not specific to product family architecture, the process has to accomplish an even more difficult task. The variability of the products must be considered when evolving the architecture and

it must be carefully verified if a requirement for a product can lead to break the product family architecture. In the analysis of the forthcoming requirements must be ascertained how easy is to add them to the current architecture and estimates the work needed for the implementation.

2.5 Losing a Software Architecture

There are a number of factors that contribute to losing a software architecture and consequently the need for an architecture reconstruction. (Parnas, 1994) introduced the concept of *software aging* showing how a mathematical product like software can age to the point of becoming usable. Aging is partly due to architectural problems like: architectural erosion, architectural drift and architectural mismatch. (Perry and Wolf, 1992) define the *architectural erosion* as ”violations in the architecture that lead to increased system problems and brittleness”. They also define the term *architectural drift* as ”a lack of coherence and clarity of form which may lead to architectural violation and increased inadaptability of the architecture”. (Garlan *et al.* , 1995) have introduced the term *architectural mismatch* to indicate the gap that exists between the designer’s architectural descriptions and the actual realizations in the code. Most of these problems are due tot the lack of proper formalization and tool support for software architectures. Poor design decisions, lack of the conformance to the architecture, changes with limited architectural understanding can ultimately damage the integrity of the system. (Jaktman *et al.* , 1999) have created a list of factors indicating architectural erosion. The relevant ones are summarized below:

- The complexity of the architecture has increased from a previous release
- The architecture is not documented or its structure is not explicitly known.
- The relationship between the architectural representation and the code is unclear or hard to understand
- Greater resources are required to implement a software change.
- Experience with the software becomes crucial to understanding how to implement a software change.
- The design principles of the architecture are violated when implementing a product variant.

One evident characteristic of architectural erosion is the increasingly reliance on a small set of software experts, ”system gurus”, for making even trivial modifications to the system.

This situation reveals a lack of documentation of domain knowledge that is mainly in the experts' minds. Although this situation is acceptable during the infancy of the system, it represents an utterly dangerous bottleneck for the evolution of the system. When the experts leave, the bring the domain knowledge away and it is intermediately lost.

2.6 REVERSE ENGINEERING

We introduce the definition of reverse engineering and related concept. It is largely based on the taxonomy by Chikofsky and Cross (Chikofsky and Cross II, 1990). We start with the classical definitions of forward and reverse engineering:

> *Reverse engineering* is the process of analysing a subject system to (i) identify the systems components and their relationships and (ii) create representations of the system in another form or at a higher level of abstraction. (Chikofsky and Cross II, 1990)
>
> *Forward engineering* is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system. (Chikofsky and Cross II, 1990)

The combination of reverse engineering and forward engineering is referred as reengineering:

> *Reengineering* is the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. (Chikofsky and Cross II, 1990)

(Casais, 1998) provides a detailed explanation of the concept of reengineering. Reverse engineering is used to create models of an existing software system for program comprehension, re-documentation or problem detection. Conversely, forward engineering is about moving from high-level views of requirements and models towards concrete implementations. Reengineering is a combination of the two. As in forward engineering, reengineering is driven by the need of implementing new requirements. Reengineering is carried out by reverse engineering the models of the existing system, applying the changes due to the new requirements and propagating the changes in the implementation with forward engineering techniques.

CHAPTER 3

RELATED WORK

Science must begin with myths,
and with the criticism of myths.

- Karl Popper

In this chapter we relate our approach with other works in the field of architecture reconstruction process and formalism, software visualization and dynamic analysis.

3.1 ARCHITECTURE RECONSTRUCTION

In this section we relate our work to the existing approaches for software architecture reconstruction. (Stoermer *et al.* , 2002) have also conducted a review of the reconstruction methods and they confirm some of the considerations that we have listed in Section 4.3 .

The reconstruction methods are typically based on an extract-abstract-present cycle, where sources are analyzed in order to populate a repository, which is queried in order to yield abstract system representations, which are then presented in a suitable interactive form to the software engineer. (Tilley *et al.* , 1996) describe the extract-abstract-present approach in more detail, referring to the steps of *data gathering*, *knowledge inference*, and *presentation*.

(Laine, 2001) presents a manual approach performed at Nokia. To reconstruct the architecture, he generated a high-level system overview and assigned the code to the various parts of the model. The identification of components was based on manual examination of the source code using the UNIX utilities EMACS and GREP as well as pen and paper. Data Gathering and Knowledge Inference were combined together.

Fully automatic approaches are based on different kinds of clustering algorithms: coupling, file names, concept analysis, type inference.

RIGI

RIGI is a programmable reverse engineering environment. It contains its own C parser or it can import other relational data in the Rigi Standard Format (RSF). RIGI can visualize the data as hierarchical typed graphs and it provides a Tcl interpreter for manipulating the graph data. The reconstruction process is based on grouping the software elements into cluster by manually selecting the nodes and collapsing them (Müller *et al.* , 1993). The reconstruction operations can also be automated with Tcl scripts. RIGI also offers various capabilities for filtering the nodes, navigating the hierarchical models and making layouts. This features make RIGI the favorite visualization choice for a variety of tools like DALI , BAUHAUS and our NIMETA environment. However, the main limitation of RIGI is the lack of an appropriate formalism for the abstraction operations.

DALI

(Guo *et al.* , 1999) propose an iterative reconstruction process where the historical design decisions are unveiled by empirically formulating/validating architectural hypothesis. The approach is supported by the DALI workbench (Kazman *et al.* , 2001; Kazman, 1996) that consists of the RIGI tool and the POSTGRESQL database. DALI allows the user to create a source code model in a SQL database. The user can then base the abstraction process (mainly a grouping activity) on a set of queries executed in the database. The results are visualized in RIGI as hierarchical graphs. They also point out the importance of modelling not only system information but also a description of the underlying semantics. In our approach, the first phase aims at clarifying the semantics of the concepts involved in the reconstruction. The major limitation of DALI is the SQL that we consider less expressive than the relational algebra.

SAR METHOD

(Krikhaar, 1999) adopts the paradigm extract/abstract/present for architecture reconstruction and base all the reconstruction operations on the Partition Relation Algebra (Feijs *et al.* , 1998). Krikhaar defines a precise process for selecting the sources of information and

creating high-level views of the architecture. However, the work is mainly focused on the module view of the architecture while in our approach we generalize the reconstruction to any architectural style. Krikhaar's and our works have very similar industrial motivations. (Postma, 2003) has also elaborated a module verification method for the Philips systems based on the partition relational algebra. The main difference with the NIMETA approach is that the conformance checking is conducted at the architectural level rather than at the module level.

PBS

(Finnigan *et al.* , 1997) propose the Software Bookshelf (PBS) that is a collection of tools for generating software architectures from program sources and presenting them in a Java-based web user interface. The goal is to keep the architectural documentation up to date. First, automated tools are combined with human effort to extract system documentation and store it in a *Software Bookshelf*. As the systems changes a librarian is responsible for comparing the documentation with the implementation with the tools and to keep the documentation updated. The reconstruction process follows the extract-abstract-present paradigm. The symbolic information is extracted with a C fact extractor and stored in RSF files. Then, the reconstructors define tree-structure decomposition by assigning files to subsystems. Next, the relational calculator GROK determines the high-level relations among the subsystems and they are visualized as graphs. The abstraction operations are based on relational algebra. The tool has been used to extract the software architecture of LINUX file system (Bowman *et al.* , 1999), the APACHE web server (Hassan and Holt, 2000) and the MOZILLA web browser (Godfrey and Lee, 2000). One key feature is the web interface that allows the architects to publish the architectural diagrams on the internet. The concept of publishing the reconstructed models through the web represents a very effective way for distributing the architectural information that we have reused in out NIMETA environment.

SWAGKIT

SWAGKIT [1] is a toolkit developed by the Software Architecture Group at the University of Waterloo, that can be used to extract, abstract and present Software achitectures. Currently Swagkit supports the extraction of C/C++ code, the abstraction to the architectural level and the presentation in a landscape form. Swagkit reuses several components from PBS. Swagkit has been architected as a pipeline, in which new filters can easily be added to support alternative design recover activities. The operations are based on the relational

[1] http://www.swag.uwaterloo.ca/swagkit

algebra manipulator GROK . NIMETA is also based on a very similar pipeline and on the relational calculator. The SWAGKIT is mainly used for the reconstruction of the module view.

REFLEXION MODELS

(Murphy and Notkin, 1997) propose a reconstruction technique based on the reflexion models. The user starts with a structural high-level view model that is iteratively refined to rapidly gain knowledge about the source code. The result is a reflexion model that shows the differences between the developer's high-level model and the recovered model. The technique is based on the definition of the mappings between the source code and the high-level concepts. A formal model is used to calculate the differences. The reflexion model approach is limited to the module view and to source file mappings. Our method is more general as we can address different architectural views we can define arbitrary mappings or transformation of the source code model. (Koschke and Simon, 2003) have extended the original reflexion model to hierarchical arhictecture models. Our approach inherently supports hierarchical models because we use hierarchical graphs for representing the software architectures.

MITRE APPROACH

(Harris *et al.* , 1995) propose a method for architecture reconstruction that combines bottom-up and top-down reconstruction. The bottom-up analysis visualize the overall file structure with a *bird's eye* view and uses a cluster dominance techniques for organizing the files. The top-down analysis uses the architectural style of the system to guide the recovery process. Hypothesized architectural styles are defined and searched in the implementation. Once the style is recognized, the mapping from the style to its realization forms the as-built architecture of the system. The architectural styles to search for are taken from a predefined library of styles. This approach has many similarities to our work as it tries to create high-level models considering the architectural styles of the system. One difference is that in our approach we do explicitly define the architectural style of the system with the stakeholders before starting the reconstruction and we do not need to recognize the mappings as they are clearly defined in the concept determination activity. The approach is also limited to the architectural styles that are defined in the library.

X-RAY

(Mendonça and Kramer, 2001) propose the X-RAY approach for recovering the architecture of distributed systems. X-RAY follows a top-bottom approach where architectural patterns (like pipes) are defined in Prolog and matched with the facts extracted from the source code. The identified instances represent the potential runtime interconnections that are used for creating the logical architecture of the system. Although the expressiveness of Prolog allows them to create precise definitions of the patterns to recover, in our experiments we found that the performance of Prolog is not adeguate to the analysis of large software systems and we had to opt for a relational algebra engine. However, Mendonça's work points out the importance of recovering architectural concepts as it's the main focus of our work (Mendonça, 1999).

ARES METHOD

(Eixelsberger *et al.* , 1998) presents an architecture recovery method for product family as part of the European Commission ESPRIT project ARES (Architecture Reasoning for Embedded Systems) (Jazayeri *et al.* , 2000). The method is based on the identification of the architectural properties and on the recovery of the architectural description for the the proposed properties. They developed a language for describing the properties called ADSL (Architecture Structure Description Language).

REVEALER

Pinzger et al. have proposed a pattern-based approach for architecture recovery (Pinzger *et al.* , 2002; Pinzger and Gall, 2002). The distinguishing aspect is the extraction of code patterns from the code that are specified with XML. This approach permits the reconstruct to define the architecturally relevant dependencies that have to be recovered from the implementation. The REVEALER can be used in the data gathering activity of NIMETA .

SARA METHOD

(Girard, 2003) proposes an architecture reconstruction process that centers around semi-automatic clustering to detect subsystems from which other relevant views are derived. These views are elicited upfront with the stakeholders. The main steps follow the extract–

abstract–present approach.

SYMPHONY

Symphony is a process model for reconstructing software architecture views. It represents the result of a joint effort for defining a unified method for architecture reconstruction. The work in this dissertation has contributed to the definition of the Symphony method (van Deursen *et al.* , 2004).

SOUL

(Mens, 2000) proposes a method for architecture conformance checking based on logic meta programming (Prolog). The implementation artifacts are mapped to an ADL for describing the conceptual architecture. The use of a logic programming is very elegant as it allows to clearly define the mapping rules and the conformance rules. However, the performance makes it unsuitable for industrial usage. The method supports only the Smalltalk language though it could be extended to other languages.

CODECRAWLER

(Lanza and Ducasse, 2003) proposes a lightweight visual approach for reverse engineering that is based on the polymetric view that combines metrics and software visualization. The method is supported by the tool CODECRAWLER that is a language-independent reverse engineering tool based on the FAMIX meta–model. The approach is mainly focused on the comprehension of the structural properties of large object–oriented software, although most of its innovative ideas could benefit the visualization of software architectures as well.

BOX

(Nentwich *et al.* , 2000) presents a portable, distributed and interoperable approach for browsing UML models. BOX is able to transform a UML model in to the VML (Vector Markup Language) that can be displayed by a web browser. This gives the possibility of publishing the UML models on the Internet. NIMETA 's web interface follows a similar

approach for publishing the architectural models in the intranet. Differently from BOX, NIMETA 's approach is based on the target architectural views.

3.2 FORMALISMS FOR ARCHITECTURE RECONSTRUCTION

All the reconstruction methods that we have presented in the previous section use a formalism for elaborating or presenting the facts extracted from the implementation. We can note that the reverse engineering and the software architecture community rely on substantially different formalisms for describing the software models. The gap originates from the rather opposed activities that are required for reverse engineering and forward architecting.

The software architecture community is mainly focused on the forward architecting activities and in the literature different Architecture Description Languages (ADLs) have been proposed. However, the ADLs have not spread in industry. The main reasons are that they are not generic enough, not standardized and poorly supported by CASE tools. Most of the ADLs support descriptions for components, connectors, configurations, and/or other aspects of software architecture. (Medvidovic and Taylor, 2000) have conducted a detailed comparisons of the available ADLs. Most of the ADLs can only describe one particular architectural view and have to be augmented with other modeling mechanisms. In general, ADLs have been successful in demonstrating various research ideas but they have not been widely accepted by practitioners (except in limited domains). Neither the ADLs have been widely used in the architecture reconstruction methods.

The Unified Modeling Language (UML), as a general-purpose design notation, is an alternative to the current ADLs. UML is a standard now and most of our designers are using it. UML descriptions of software architecture not only provide a standard definition of the system structure and system terminology, but also facilitate consistent and broader understanding of the architecture and enable more extensive tool support for architecture design. However, its current semantics fails to meet the needs for architecture description: it is weak at describing interfaces, the abstractions it provides are not univocal and it provides little support for modeling architecturally significant information. UML is also not suitable for modeling reverse engineered architectural models: the notation does not allow to model the source code, and does not offer other support for typification than extension. (Demeyer *et al.* , 1999) describes the problems of using UML for reverse engineering purposes. In conclusion, UML as such is not an ideal language for describing the basic building blocks of software architecture (components, connectors and architectural configurations). In order to produce architectural diagrams in UML format, in our work we mainly stereotype the UML elements (like classifier and package). We can anyway mention attempts to use UML as an architecture description language.

(Robbins *et al.* , 1998) map an ADL to UML with an extended UML meta-model and then use the adapted UML as an ADL. With this approach one can translate the ADL architecture models into UML models. The problem of this approach is the UML architecture model is difficult to understand if the reader is not familiar with that particular ADL, and in addition, the UML architecture model is not as clear and intuitive as the original ADL model and the model created directly with UML. In (Rumpe *et al.* , 1999) ROOM has been integrated with UML and the result is used as an ADL (UML-RT) to model software architecture. This approach has strong case tool support - ROSE RT from Rational/ObjecTime. However it has the limits of the ROOM approach for architecture modelling, such as asynchronous signal communication based interfaces (ports and connectors). (Störrle, 1999) tried to embed architectural concepts and notations into the conceptual framework of UML by introducing new classes of UML's meta-model as stereotypes of existing classes. It is the natural way of extending/adapting UML for architecture modeling. The limitation of this approach is the standardization of embedded architectural concepts as part of UML and the compliance with commercial UML modeling tools. (Hofmeister *et al.* , 1999) present practical guidelines on describing architecture views with UML from the authors' experience. They have explicitly specified the elements of four architecture views (conceptual, module, execution, and code) (Hofmeister *et al.* , 2000) and their corresponding UML Meta-model classes and stereotype names. Their approach can help the architects to achieve clear and consistent architecture descriptions without extension to the current UML standard.

3.3 FORMALISMS FOR REVERSE ENGINEERING

The reverse engineering community has developed its own techniques for modeling the reverse engineered data. The research has been focused on the formalism for the model transformations, the content of the model and the exchange format. Popular formalisms are the relational algebra (Holt, 1998; Feijs *et al.* , 1998; Berghammer *et al.* , 2003), SQL (used in DALI (Guo *et al.* , 1999)) and Prolog (Mens, 2000; Mendonça, 1999). GReQL is also a specific language for query graph structures (Kullbach and Winter, 1999). Prolog offers a the best option for expressiveness while compromising the performance. Relational algebra has been widely used for analyzing large software systems and has support for the transitive closure that is missing in SQL.

Concerning the content, there has been an effort to create a basic set of reference meta-models for various languages. FAMIX[2] is an extensible language independent model for object-oriented programming languages that was originally developed during the FAMOOS project [3] in order to provide a language independent exchange mechanism between the

[2]FAMIX 2.0 specification is available at: http://www.iam.unibe.ch/ famoos/FAMIX

[3]The project FAMOOS (Framework-based Approach for Mastering Objected Oriented Software Evo-

reverse engineering tools (Tichelaar, 2001; Demeyer *et al.* , 2001). The DATRIX schema has been developed for C/C++/Java languages (Holt *et al.* , 2000b). The COLUMBUS/CAN [4] tool is based on its own schema for C/C++ (Ferenc *et al.* , 2002). There has also been an effort to develop a standard reference schema from (Ferenc *et al.* , 2001) and the the Dagstuhl Middle Model (DMM) (Ebert *et al.* , 2001).

Concerning model format, the reverse engineering community tried to define a unique format for storing and exchanging the models like the Graph Exchange Language (GXL) (Holt *et al.* , 2000a).

3.4 DYNAMIC ANALYSIS

The dynamic analysis aims at describing the run time behavior of a software system. Due to the size and complexity of the dynamic traces that we have to analyze, we are mainly interested in the possibility of combining the static and dynamic analysis. Most of the methods and tools are intended to analyst either the static or the dynamic aspects of a system but not both at the same time. SCED (Koskimies *et al.* , 1998) and SCENE (Koskimies and Mössenböck, 1996) are reverse engineering tools that focus on the dynamic analysis tasks. Some attempts have been done to merge the dynamic information into static views. (Systä, 2000) exploited RIGI and SCED to combine static and dynamic analysis. Her work was focused on Java source code level and program comprehension. She used the concept of horizontal and vertical abstractions to reduce the complexity and raise the abstraction level of the subject software. She also showed how the static information can guide the dynamic analysis and how to slice the static view using the dynamic data. ISVIS (Jerding and Rugaber, 1997) is a tool for analyzing C applications, which offers vertical abstractions and restricted horizontal abstractions. It only provides a MSC representation, and the horizontal abstractions are limited without support for static model visualization. Several approaches use a vertical abstraction to cope with message complexity. SCENE uses limited size subscenarios and hyperlinks to source to aid program understanding. SCED uses algorithmic constructs to express repetition like loops and subscenarios. In the filed of tools for static and dynamic reverse engineering there is an attempt based on DALI (Kazman and Carrière, 1998) to support the static and dynamic analysis for object oriented software. (Lange and Nakamura, 1995) have presented the PROGRAM EXPLORER that supports source code, class hierarchy, invocation and object graphs. The various views are synchronized but the tool lacks of abstraction capabilities and is C++ domain dependent. (Richner and Ducasse, 1999) present a method to create views that combine static and dynamic information for

lution) was an Esprit project (no. 21975) that aimed at developing tools and techniques for transforming object-oriented legacy systems into flexible framework-based applications

[4] COLUMBUS from FRONTENDART : http://www.frontendart.com

object-oriented software. It is based on a logic programming language (Prolog) to query the data and on digraphs for the visualization. (Gschwind *et al.* , 2003; Gschwind and Oberleitner, 2003) have proposed the method ARE for dynamic analysis that is based on the tracing the runtime data as parameter and object values. Their approach permits to analyze the logical dependencies between the software elements. The instrumentation is achieved with the the use of aspect oriented programming.

The major shortcomings of the current approaches are (1) architectural level is not the primary focus because most of the tools focus on source-level program understanding, (2) lack of automation in the extraction and analysis process, (3) lack of complete graphical visualization of software models and (4) the tools are often coupled with a specific programming language and can be hardly used to handle generic architectural concepts like applications, servers and subsystems.

CHAPTER 4

PROBLEM STATEMENT

New ideas pass through three periods:
**It can't be done.*
**It probably can be done, but it's not worth doing.*
**I knew it was a good idea all along!*

- Arthur C. Clarke

In this chapter, we introduce the problem that we address in this dissertation and we give an overview of our approach.

4.1 INTRDOCUTION

A software system is the result of the design decisions taken by the developers during its development. Among all the design decisions, the most important are modeled by the software architecture. They are considered important because they create a shared mental model of the essence of the implementation that is useful for all the developers. Based on the classification of design decisions given in Section 2.2, we can conclude that the shared mental model includes:

- a shared terminology that every team member understand when talking about the system. This includes a clear understanding of the architectural concepts.
- a shared understanding of the structure of the system at different levels: functional, physical, at run-time, organizational and so on.

- a shared development practise that enables the developers to make modifications to the software programs without breaking their integrity.
- a shared responsibility of maintaining the system healthy.

This shared mental model is what enables the developers to communicate, to coordinate and, in general, to cope with the complexity of large software systems that are concurrently and distributively developed by a large group of people. Without this model, the development would become confused, slow and ultimately collapse.

However, metal models only exists in the minds of people. Design decisions are often based on intuition, experience, sometimes hype, and, in general, are the result of the human interaction. This makes a software program not only a mathematical construct but also the creative product of a collaborative human effort. In an ideal situation, the mental models could be formalized with an architecture description language and linked with the implementation. But the reality shows that we do not have such a formalism yet. Moreover, to be effective the mental models have to exist in the people's minds. As a result, there is no guarantee that the developers' mental models are somewhat consistent with the implementation. This leads to the main research question that we address in this dissertation: how can we make the mental models of the architecture as consistent as possible with the real implementation ?

We can identify a chasm between the *concrete* (or as-implemented) architecture and the *conceptual* (or as-designed, as-indented) architecture (Bowman *et al.* , 1999). The concrete architecture is embedded in the implementation and mostly unknown, even though it is available in the source code. The conceptual architecture is the developers' understanding of the implementation. Our research problem is to make the concrete architecture explicit and to allow the developers to their conceptual architecture (their mental models) closer to the real one. Architecture reconstruction is the methodology that we use to reveal the concrete architecture. Our goal is to create a model of the software architecture that is conceptually at the same level of the developers' mental models and it rigorously reflects the facts of the implementation.

4.2 Software Architecture Reconstruction

The concept of software architecture has been largely adopted in industry to the point that it is not only an important factor for the quality of the resulting system but also for the control and management of the overall software development process. Many companies are adopting an architecture-centric approach, especially for the development of software

product-lines (Bosch, 2000; Jazayeri *et al.* , 2000; Clements and Northrop, 2001). The architecture description plays a central role in the architecture-centered software development, even though there is no agreement what information it must convey. Anyway it is widely accepted that multiple architectural views are needed to describe the software architecture as presented by (Kruchten, 1995; Hofmeister *et al.* , 2000; Clements *et al.* , 2003). However, industrial practitioners are still facing big challenges in applying an architecture-centric approach effectively. Among the reasons, we can mention the lack of a proper formalism (ADL) suitable for various the domains and the lack of tool support in CASE tools. The architecture description is often maintained with informal notations (being textual or graphical) without a systematic support for analysis, controlled modifications and efficient maintenance. Moreover, even for well-documented architectures there is no guarantee that what there is on papers is really reflected in the implementation. We believe that in most of the cases, the architects have not a systematic control over the most vital, hence architecturally relevant, aspects of a software system.

In this dissertation, we tackle the research problem of developing a methodology for reconstructing a *documented architecture* for a software system from the available evidence (implementation, documentation and people). In particular, we focus on recovering the architecturally relevant views that are considered vital by the software architects for the development and maintenance of their large software systems. We also address the problem of automating the conformance checking of the architecture against specific architectural rules.

Architectural pertinence has been a major concern in our research. We argue that existing reverse engineering techniques do not to satisfy the needs of the software architects in practice. The main reason is that the views they deliver are not at the correct level of abstraction, being too focused on the programming languages aspects rather than on logical or architecturally relevant concepts. On the other side, the software architecture community has not provided a proper formalism for architecture modeling. ADLs are too specific for certain domains and the inaccuracy of UML, a general purpose modeling language, inhibits a rigorous architecture modeling method. Although basic programming concepts like *class*, *function*, or *inheritance* are well-understood, there are countless interpretations for architectural concepts like *component*, *subsystem* or *interface* even within the same development team. Our position is that a fundamental activity of architecture reconstruction is to recover (and define) the shared vocabulary of architectural concepts that build the system. Once this activity is completed, we can focus on the recovery of the architectural views. In this way the reconstruction process can be tailored around the architectural style of the system.

Abstraction is a major concern for architecture reconstruction. In short, architecture recovery is a process of inferring compact conceptual representations from a wide set of facts about a software system. Some of these conceptual representations are not visible in the

implementation of the system (e.g. the concept of dependency is no visible in a for loop). Those are conceptual representations that we need to create in order to simplify complex concepts. Information hiding (as defined by (Parnas, 1972)) plays an important role here. For example, the dependency relationship between two subsystem can be mapped to thousands of function calls. The dependency represents an abstraction because it hides many details that otherwise would overwhelm our comprehension. Humans' mental faculties cannot grasp the whole implementation of a software system at once but they need to use simplified mental models. bstraction is an important point for this task of simplification. This leads to an important point for our thesis. Although abstraction is a fundamental activity for the design of software system, there is another important aspect: the bridges between our mental models and the implementation itself. A mental model (like a sw architecture) loses its effectiveness if it cannot be regularly confronted with the reality. The more the mental models are far away from the reality, the less useable they are (this relates to the concept of architectural drift from (Perry and Wolf, 1992)). Concluding, there are two major concerns in our work:

- to provide the correct abstractions for creating the simplified mental models (i.e. software architecture) of a software implementation.
- to maintain the bridges between the mental models and the real implementation.

Industrial applicability is a another major concern in our work. The customers of our reconstruction projects are involved in multi-million dollars businesses. Their systems represent a huge investment for the company, they took several years of pure development and they will stay in the markets for decades. We need to develop a solution that work in an industrial context and is scalable for software systems containing millions of lines of code. Moreover, we need to guarantee that the reconstruction process is sufficiently simple to understand and easy to operate.

Conformance checking is the fourth concern in our research. Software product families are developed concurrently by different teams. The major preoccupation of the family architects is to preserve the architectural integrity of the system over the time. Especially for a highly dynamic domains like mobile phones, delays in the delivery of required features or quality levels can cause huge losses for the company. The architects need to ensure that the implementation conforms with their intended design. Product families are built around a common architectural style (the family reference architecture) that describes the architectural rules that have to be valid for all the members of the family. The architects need a method to enforce the rules in the implementation and detect those features or products that break the architectural integrity. Our aim is to provide the architects with the convenient information and formalism for conducting the proper conformance checks.

Feature oriented reverse engineering has been a topic of research with the work that we conducted on the dynamic analysis. In the telecommunication domain, system features represent concrete artifacts that are shared across various products of the family. They represent the artifacts at the highest level of abstraction in the solution domain, as we discussed in Section 2.1. From a feature engineering perspective, the goal of reverse engineering is to discover how the features are implemented, what are they interactions, or what are the components are involved by their execution (as initially proposed by (Turner *et al.* , 1999)). In a mobile phone product, reverse engineering could be used (1) to identify how the feature for establishing a phone call is been implemented on various protocols (like GSM or 3G), (2) to discover how the feature for receiving a call interferes with game playing, or (3) to recover the procedure for setting up a WAP connection with GPRS. Understanding the implementation of features is a vital activity for product family as features are typically reused across several products.

4.3 NOVELTY OF THE APPROACH

As discussed in Section 2.2, a software architecture represents a shared understanding of what is perceived vital by the developers for the life of the system. As software systems are built in different ways, what is considered important varies from system to system. For this reason, the architecture reconstruction process must recover what is considered valuable by the stakeholders of the system. The reconstruction process must be tailored to the architectural style of the system under reconstruction. Therefore, we propose a method where the reconstruction is driven by the architectural views and the architectural concepts of the system. This choice ensures that we deliver models at the correct level of abstraction for the architects. This argument represents the novel idea that is exploited in this dissertation. Only few approaches explicitly address the architectural concepts as first-class elements of the reconstruction. Our work bridges the gap by creating a reverse engineering process that addresses the requirements of the architects and aims at delivering valuable information to them. Differently from traditional bottom-up approaches, we select the views to recover based on the architectural style of the system.

The IEEE (P1471, 2000) also emphasize that the architectural viewpoints are dependent on what the stakeholders perceives as important to document. This makes architecture reconstruction a peculiar reverse engineering activity where the the process and the goals are dependent on the architectural style of the system. While the recovery of a class diagram is a general techniques for any object-oriented programming language, we can only generalize the architecture recovery for certain architectural styles. For this reason, in our approach the selection of the architectural viewpoints play an important role and they are selected by the stakeholders.

Based on the related work presented in Chapter 3, we can make the following considerations about the state of the art on architecture reconstruction:

- There is no approach or tool that permits an explicit selection of the architectural views that can address the specific needs of the stakeholders. Most of the approaches are focused only on a particular type of view.
- Existing approaches do not select the architectural concepts as first-class entities of the reconstruction.
- Only few methods ((Mendonça, 1999; Harris *et al.* , 1995) follow a top-bottom reconstruction approach and explicitly take into consideration the architectural style of the system.
- Most of the approaches address the case of understanding an unknown software system. Although this is an important program understanding activity, in our experience the reconstruction projects are initiated and driven by the software architects who have clear expectations on the outcome. In our experience, only few cases of reconstruction were initiated by the acquisition of an external company[1].
- The approaches do not consider how to integrate domain knowledge with information extracted from heterogeneous sources (such as different language extractor, CASE tools and documentation).
- Some approaches are not scalable to industrial cases.
- Among the various formalisms the relational algebra seems to be the most powerful while preserving expressiveness.
- Existing commercial modeling tools are incapable of recovering the architectural information from the implementation.

We compiled a list of criteria that our reconstruction process and formalism should satisfy:

Adaptability We should be able to tailor the reconstruction method to a wide range of software systems. It should pose no a priori restriction about the architectural styles, programming languages, domains, design methods and development tools.

[1] Even in these cases the developers of the external company were legally bound to to facilitate the software intake for a certain period of time.

Extensibility The formalism should be flexible and extensible enough so that it is possible to iteratively define the content of the reconstruction and modify it over the time. The tool environment should also be extensible in order to integrate the development tools that are already in place in the organization.

Simplicity The process and the formalism should be simple and intuitive so that it can be understood by architects and developers, and it can be operated by unexperienced reconstructors.

Efficiency The overall approach should be scalable for analyzing systems made of million lines of code and capable of producing results in a reasonable amount of time.

Generality The formalism should be enough general to describe different architectural views (class diagram, logical view, component and connector).

Expressiveness The formalism should be sufficiently expressive in order to handle complex architectures, inconsistent data, multiple views. It should provide a clear notation, which is semantically well defined and easy to understand by all stakeholders.

Repeatability It should be possible to automate the reconstruction process in order to run it at every build of the system with minimal human effort.

4.4 VALIDATION

To validate our thesis, we proceed in the following way:

1. We develop NIMETA 's formalism by formalizing the reference viewpoints (design and architecture) with the binary relational algebra.

2. We define the architecture reconstruction process in terms of five basic activities: problem definition, concept determination, data gathering, knowledge inference, presentation. We also define additional activities for conformance checking, architecture assessment and re-documentation. The process consists of three phases: *process design*, *view recovery* and *result interpretation*.

3. We develop an integrated tool environment, called NIMETA , that supports our reconstruction method.

4. We perform two industrial case study with the Nokia business units. The case studies demonstrate real-life architecture reconstruction experiences in practise. Both the case studies were carried out by our team at the Nokia Research Center in close

cooperation with the Nokia business units. The reconstruction projects have been completed and the the technology transfer is still ongoing.

Based on several experiences with the existing approaches (as also documented in (O'Brien, 2002), we made the following decisions for the development of the NIMETA reconstruction method:

- NIMETA 's formalism is based on the binary relational algebra. Although a logic programming language (like Prolog) proved to be well suited for representing architecture knowledge (as shown in our work (Riva, 2002)), we opted for the binary relational algebra for several reasons. First, we have an efficient implementation of the algebra engine (especially the expensive transitive closure), while the Prolog implementation could not scale up to large datasets. Second, the algebra is sufficiently expressive to elegantly formulate the transformations required by our reconstruction tasks (as confirmed by (Fahmy *et al.* , 2001).

- We adopt a view-centered reconstruction method. The architects propose the viewpoints to recover (either selected or combined from a viewpoint catalogue). The view recovery phase is guided by the selected viewpoints.

- The architectural views are represented as hierarchical typed directed graphs. The types of nodes and arcs conform to the viewpoint. The hierarchical structure is used for defining abstractions and reducing the complexity of the graphs.

- We take the well-accepted viewpoints from the literature ad a reference. For the design viewpoint we rely on the FAMIX model as it is language independent, it captures the relevant aspects of object-oriented systems and we contributed to its development during the FAMOOS project. For the architecture viewpoints, we rely on the recently published book on documenting software architectures (Clements *et al.* , 2003)) as it contains a rich collection of architectural views.

- We base the tool environment on a configurable pipeline of tools and Python scripts. The data exchange is mainly based on the RSF format from where the other formats are converted. We plan to rely mainly on commercial or academic tools when possible (especially for the extraction and the visualization), otherwise we develop our own tools.

CHAPTER 5

BINARY RELATIONAL ALGEBRA

If people do not believe that mathematics is simple,
it is only because they do not realize how complicated life is.

- John von Neunman

5.1 INTRODUCTION

In this chapter we introduce the calculus of binary relations. The first investigations on the theory of relations can be found in the article published by (Morgan, 1860) in 1860. Realizing the limitations of the traditional logic for describing even simple arguments, De Morgan recognized the full importance of relations and he vaguely attempted to formalize it. The fundamental laws of the theory of relations were established by C. S. Peirce who clearly formulated the theory as a calculus with the basic operands (Peirce, 1933). Schröder extended the calculus with the operations of transitive closure, reflexive transitive closure and De Morgan duals in 1895 (Schroöder, 1895). For the following half century the subject has been neglected till 1941, when A. Tarski revitalized the field and create the modern approach to the calculus of relations (Tarski, 1941). (Schmidt and Ströhlein, 1993) refined Tarski's calculus and formulated the theory of the relations that has been applied in many domains . We take their work as a reference for our formalism. A more detailed history of the calculus of binary relations can be found in the paper of (Pratt, 1992).

Application of the binary relational algebra to the description of software architecture can be found in the work of (Holt, 1998), (Feijs *et al.* , 1998; Feijs and Krikhaar, 1999; Feijs and van Ommering, 1999) and (Berghammer *et al.* , 2003). In Section 3.2, we gave an overview of their approaches. The formalism that we use throughout this thesis is based

on the binary relational algebra and is centered around three elements: *sets*, *relations* and *graphs*. We introduce them and their operations in the following sections. We take the book of (Schmidt and Ströhlein, 1993) as a basic reference.

5.2 SETS

A set S is a collection of well-defined distinct objects called *elements*(or members). If x is an element of the set S we write $x \in S$. The symbol $\emptyset$ denotes the empty set. We can define a set by enumerating its members. For example, the set containing the elements a, b, c is denoted by $\{a, b, c\}$. The set of all the elements which have the property $P(x)$ can be written as $S = \{x|P(x)\}$. For combining various properties, we use the Boolean operators $\vee$ (disjunction, or) and $\wedge$ (conjunction, and). $P \vee Q$ is true if and only if (denoted by iff) P is true or Q is true or both P and Q are true. $P \wedge Q$ is true iif P and Q are both true. We also define the implication operators $\Rightarrow$ and $\Leftrightarrow$. $P \Rightarrow Q$ holds if P is true then Q is true. $P \Leftrightarrow Q$ holds iif $P \Rightarrow Q \wedge Q \Rightarrow P$.

Two sets P and Q are *equal*, denoted by $P = Q$, iff $\forall x[x \in P \Leftrightarrow x \in Q]$. A set P is a *subset* of the set Q, denoted by $P \subseteq Q$, iff $\forall x[x \in P \Rightarrow x \in Q]$. If $P \subseteq Q$ and $P \neq Q$ then P is a *proper subset* (or *strict subset*) of Q, denoted by $P \subset Q$. A set P is a *superset* of Q, denoted by $P \supseteq Q$, iff $\forall x[x \in P \Leftarrow x \in Q]$. The *proper superset* is denoted by $Q \supset P$. The cardinality of a finite set P, denoted by $|P|$, is the number of distinct elements in the set P.

The *union* of two sets M and N, denoted by $M \cup N$ or simply $M + N$, is the set $T = \{x|x \in M \vee x \in N\}$. The *intersection* of two sets M, N is denoted by $M \cap N$ and is the set $T = \{x|x \in M \wedge x \in N\}$. The *difference* of two sets M and N, denoted by $M \setminus N$ or simply $M - N$, and is the set $T = \{x|x \in M \wedge x \notin N\}$. We do not introduce the complement operator.

The Cartesian product of two sets M, N is denoted by $M \times N$ and is the set $R = \{(x, y)|x \in X \wedge y \in Y\}$.

5.3 RELATIONS

A *binary relation* R, or simply a *relation*, from X to Y is a subset of the Cartesian product $X \times Y$. The elements $x \in X, y \in Y$ are in relation R if $(x, y) \in R$. We use several

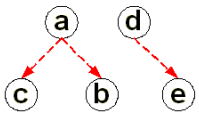


Figure 5.1: Directed graph representing the relation R.

notations for indicating a binary relation:

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

We can define a relation by enumerating its elements. For example, we can define a relation like: $R = \{(1, 2), (5, 6), (7, 8)\}$.

A relation can be represented in a directed graph. Let V be a set of elements, called *vertices*, and $R \subseteq V \times V$ be a relation. We define $G = (V, R)$ to be the *directed graph* of R. The pairs of elements in R are called *arcs* (or *edges*) and for the element (x, y) are directed from the x to y. The Figure 5.1 shows the directed graph for the relation $R = \{(a, b), (a, c), (d, e)\}$.

The operations of *union*, *intersection*, *difference* and the *equality* for the binary relations are inherited from the operations for the sets:

$$\begin{aligned} P \cup Q = P + Q &= \{(x, y) | (x, y) \in P \vee (x, y) \in Q\} \\ P \cap Q &= \{(x, y) | (x, y) \in P \wedge (x, y) \in Q\} \\ P \setminus Q = P - Q &= \{(x, y) | (x, y) \in P \vee (x, y) \notin Q\} \\ P = Q &\Leftrightarrow \forall (x, y)[(x, y) \in P \Leftrightarrow (x, y) \in Q] \end{aligned}$$

The *inverse* (or transposition) of a relation R, denoted by R^{-1}, is obtained by reversing the tuples of R:

$$R^{-1} = \{(x, y) | (y, x) \in R\}$$

We define the operations for selecting the *domain*, *range* and *carrier* of a relation R:

domain	$dom(R) = \{x \mid \exists (x, y) \in R\}$
range	$ran(R) = \{y \mid \exists (x, y) \in R\}$
carrier	$car(R) = dom(R) \cup car(R)$

Considering the directed graph $G = (V, R)$, we can note that the $car(R)$ is a subset of the set of vertices V ($car(R) \supseteq V$).

We define the operations for restricting the domain and range of a relation R to a set S:

domain restrict	$R\|_{dom}S = \{(x, y)\|(x, y) \in R \wedge x \in S\}$
range restrict	$R\|_{ran}S = \{(x, y)\|(x, y) \in R \wedge y \in S\}$
carrier restrict	$R\|_{car}S = (R\|_{dom}S)\|_{ran}S$

We define the operations for excluding from the domain and range of a relation R a set S:

domain exclude	$R \setminus_{dom} S = \{(x, y)\|(x, y) \in R \wedge x \notin S\}$
range exclude	$R \setminus_{ran} S = \{(x, y)\|(x, y) \in R \wedge y \notin S\}$
carrier exclude	$R \setminus_{car} S = (R \setminus_{dom} S) \setminus_{ran} S$

We define the operators *left projection* and *right projection* for selecting elements of the relation R with respect to the set S and the operators *left image* and *right image* for selecting elements of R with respect to y and x respectively:

left projection	$R.S = \{x\|(x, y) \in R \wedge y \in S\}$
right projection	$S.R = \{y\|(x, y) \in R \wedge x \in S\}$
left image	$R.y = \{x\|(x, y) \in R\}$
right image	$x.R = \{y\|(x, y) \in R\}$

We define the *top* and the *bottom* of a relation R. Given the directed graph of R, the top consists of the set of vertices that have no incoming arcs (root vertices). The bottom consists of the set of vertices that have no outgoing arcs (leaf vertices). The definitions are:

top	$\top(R) = dom(R) - ran(R)$
bottom	$\bot(R) = ran(R) - dom(R)$

The *relational composition* of P, Q, denoted by $P \circ Q$, allows us to compose different relations. Given the directed graph of $P \cup Q$, the relation $P \circ Q$ is the set of all the edges that can be drawn by following an edge of P and an edge of Q. The definition is:

relational composition	$P \circ Q = \{(x, y)\|\exists z(x, z) \in P \wedge (z, y) \in Q\}$

The composition operator is associative and it is possible to compose a relation n times: $R \circ R \circ R... \circ R$ (denoted by R^n). By definition, we have that $R^0 = Id$.

The *identity* relation for the set X, denoted by Id_X, is the relation:

identity relation $$Id_X = \{(x, x) | x \in X\}$$

Considering the directed graph $G = (V, R)$, we assume that the identity relation is calculated on the set of vertices V and we simply write Id. The identity relation of a graph represents the set of all the edges that loop directly back to the nodes. The identify relation has the following properties:

$$xIdy \Leftrightarrow x = y$$

$$R \circ Id = Id \circ R = R$$

The *transitive closure* and the *reflexive transitive closure* of a relation R, denoted by R^+ and R^*, are defined as:

transitive closure $$R^+ = \bigcup_{i=1}^{\infty} R^i$$

reflexive transitive closure $$R^* = R^0 + R^+ = Id + R^+$$

Given a directed graph $G = (V, R)$, the transitive closure is the set of all the arcs that connect the vertices that are reachable by following the arcs of R.

Given the directed graph of R, a *cycle* is a path that starts and ends at the same vertex and has at least one arc. A relation R is defined *acyclic* iff $Id \cap R^+ = \emptyset$. In an acyclic graph there is no path from any vertex to itself.

The operation of *transitive reduction*, denoted by R^-, removes all the *short-cuts* from the relation R. An edge (x,y) is a short-cut iif $\exists z | (x, z) \in R \wedge (z, y) \in R$. The transitive reduction is defined as:

transitive reduction $$R^- = R - (R \circ R^+)$$

The transitive reduction of R is also called the *Hasse diagram* of R. The transitive reduction also removes all the cycles of R.

5.4 HIERARCHICAL TYPED GRAPHS

In the previous section, we have given a definition of a directed graph of a relation R. In this section, we give other definitions related to graph that will help us for describing the various views of a software system.

From the previous section, we recall that graph is pair of sets (V, R). The elements of the set V are the vertices of the graph. If the elements of the relation R are ordered pairs (x, y) then the graph is *directed*. If the pairs of R are unordered then the graph is *undirected*. Throughout the thesis, we only consider directed graphs based on the ordered pairs of a relation R. The edges are directed from the element x to the element y.

A *path* is list of vertices of a graph where each vertex has an edge from it to the next vertex. An undirected graph is a *connected* graph if there is a path between every pair of vertexes. A directed graph is a *strongly connected graph* if there is a path from each vertex to every other vertex.

A *tree* is a connected, directed and acyclic graph. The top of the graph is the *root* of the tree. A *forest* is a collection of one or more trees.

A *typed graph* is a graph where we can distinguish different types of vertices and edges. Given a set of vertices V, a type relation T for the vertices and the relations $R_1, R_2, ...R_n$, we can define a typed graph as the set $G = (V, R_1, R_2, ...R_n)$. The different types of nodes and edges are often visualized with different colors or notations (hence the term *colored graphs*). For a typed graph we have the following properties:

$$dom(T) = V$$

$$\forall i[R_i \subseteq V \times V]$$

For simplicity, we often indicate a typed graph as $G = (V, R)$ and we assume that the can distinguish the various relation R_i.

A *hierarchical typed graph* is a typed graph with a hierarchy. Given a base graph $G = (V_G, R_G)$ and a tree (i.e. a directed acyclic graph) $C = (V_C, E_C)$, we define the hierarchical typed graph as the set $H = (C, G)$ with the the requirement that $\perp E_C = V_G$. The tree C is called *containment tree* or *containment relation*. The type relation T for H is defined as $T = T_G + T_C$. We can indicate the hierarchical typed graph as $H = (C, R_1, R_2, ...R_n)$, or simply $H = (C, R)$, and we assume that:

$$V = \bigcup_{i=1}^{n} car(R_i)$$

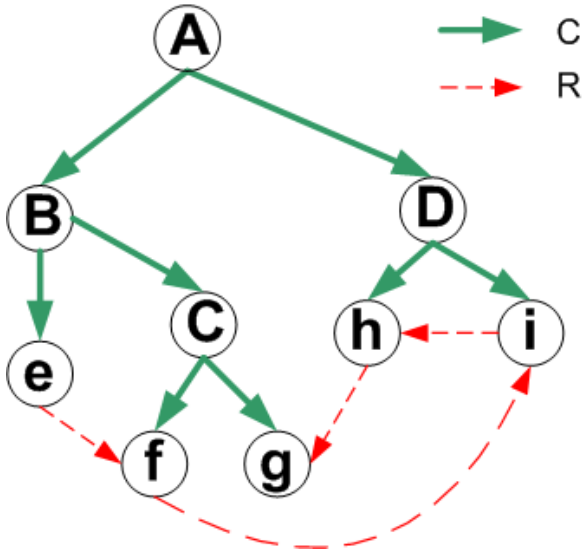


Figure 5.2: An example of an hierarchical graph.

Figure 5.2 shows the hierarchical graph $G = (C, R)$:

$$C = \{(A, B), (A, D), (B, C), (B, e), (C, f), (C, g), (D, h), (D, i)\}$$
$$R = (e, f), (h, g), (i, h), (f, i)$$

Given a containment tree C, we define several operators for the working on the trees:

parent (part-of)	$P(C) = R^{-1}$
sibling	$S(C) = P(C) \circ C - Id$
descendant	$D(C) = C^{+}$
reflexive descendant	$Do(C) = C^{*}$
ancestor	$A(C) = P(C)^{+}$
reflexive ancestor	$Ao(C) = P(C)^{*}$

Given a hierarchical graph $H = (C, R)$, we define the operation of *lifting*, denoted by $R \uparrow C$, in the following way:

lifting $\quad R \uparrow C = \{(x, y) | \exists a, b (a, b) \in R \wedge (x, a) \in C^{+} \wedge (y, b) \in C^{+} \wedge x \neq y\}$

The operator of lifting calculates the set of edges between the nodes of C that are induced by the relations of R. We also introduced the operations of *left lifting*, denoted by $R \uparrow C$, and *right lifting*, denoted by $R \uparrow C$:

$$\begin{array}{ll} \text{left lifting} & R \uparrow C = \{(x,y) | \exists a (a,y) \in R \wedge (x,a) \in C^+ \wedge x \neq y\} \\ \text{right lifting} & R \uparrow C = \{(x,y) | \exists a (x,b) \in R \wedge (y,b) \in C^+ \wedge x \neq y\} \end{array}$$

We can define the *full lifting*, denoted by $R \Uparrow C$:

$$\text{full lifting} \quad R \Uparrow C = \{(x,y) | \exists a,b (a,b) \in R \wedge ((x,a) \in C^+ \vee (y,b) \in C^+) \wedge x \neq y\}$$

The full lifting is equivalent to the union of the normal lifting, left lifting and right lifting. We use the full lifting mainly in the implementation of the tools for calculating all the possible lifted edges.

We can also define the lifting operators in relational algebra using the operators on trees that we have previously defined:

$$\begin{array}{ll} \text{lifting} & R \uparrow C = D(C) \circ R \circ A(C) - Id - D(C) - A(C) \\ \text{left lifting} & R \uparrow C = D(C) \circ R - Id - D(C) \\ \text{right lifting} & R \uparrow C = R \circ A(C) - Id - A(C) \\ \text{full lifting} & R \Uparrow C = R \uparrow C + R \uparrow C + R \uparrow C \\ & \quad = Do(C) \circ R \circ Ao(C) - Id - Do(C) - Ao(C) - R \\ & \quad = D(C) \circ R \circ A(C) + D(C) \circ R + R \circ A(C) - Id - D(C) - A(C) \end{array}$$

The opposite of the lifting is called *lowering*, denoted by $R \downarrow C$:

$$\text{lowering} \quad R \downarrow C = \{(x,y) | \exists a,b (a,b) \in R \wedge (a,x) \in C^+ \wedge (b,y) \in C^+ \wedge x \neq y\}$$

In relational algebra, we have:

$$\begin{array}{ll} \text{lowering} & R \downarrow C = A(C) \circ R \circ D(C) - Id - D(C) - A(C) \\ \text{reflexive lowering} & R \downarrow C = Ao(C) \circ R \circ Do(C) \end{array}$$

The lowering operator allow us to calculate the edges of R that induce a lifted edge. Let $P = R \uparrow C$ be the set of the lifted edges and let (x,y) be an edge in P, the set $Q = ((x,y) \downarrow C) \cup R$ contains the edges of R that induce the lifted edge (x,y).

We provide an example of the various operations on the graph $G = (C, R)$ of Figure 5.2:

$$
\begin{aligned}
C &= \{(A,B),(A,D),(B,C),(B,e),(C,f),(C,g),(D,h),(D,i)\} \\
R &= \{(e,f),(h,g),(i,h),(f,i)\} \\
dom(R) &= \{e,h,i,f\} \\
car(R) &= \{e,f,h,g,i\} \\
R|_{dom}\{e,i\} &= \{(e,f),(i,h)\} \\
R.g,h &= \{(h,i)\} \\
i.R &= \{h\} \\
\top(C) &= \{A\} \\
\bot(C) &= \{e,f,g,h,i\} \\
Id_R &= \{(e,e),(f,f),(g,g),(h,h),(i,i)\} \\
P(C) &= \{(B,A),(D,A),(C,B),(e,B),(f,C),(g,C),(h,D),(i,D)\} \\
S(C) &= \{(B,D),(D,B),(C,e),(e,C),(f,g),(g,f),(h,i),(i,h)\} \\
D(C) &= \{R,(A,e),(A,C),(A,f),(A,g),(A,h),(A,i),(B,f),(B,g)\} \\
R \uparrow C &= \{(B,D),(D,B),(C,D),(D,C)\} \\
R \restriction C &= \{(B,i),(C,i),(D,g)\} \\
R \upharpoonright C &= \{(h,C),(h,B),(f,D),(e,C)\} \\
R \Uparrow C &= \{(B,D),(B,i),(h,C),(h,B),(C,D),(e,C),(D,C), \\
&\quad (D,B),(D,g),(C,i),(f,D)\} \\
\{(C,D)\} \downarrow C &= \{(f,i),(f,h),(g,h),(g,i)\} \\
\{(C,D)\} \downarrow C) \cup R &= \{(f,i)\}
\end{aligned}
$$

5.5 VIEWS AND VIEWPOINTS

The concepts of views and viewpoint are necessary for describing the software architecture of a system, as we have discussed in Section 2.3. We give their definitions in terms of binary relational algebra.

We define a *viewpoint* $V_P = (T_E, T_R, T_C, P)$ as the union of all the types of entities, T_E and relations T_R that are allowed in the viewpoint, the type of containment relation and the

set P of properties (or rules) that are satisfied for the viewpoint:

viewpoint
$$V_P = (T_E, T_R, T_C, P)$$
$$T_E = \{\text{entity types}\}$$
$$T_R = \{\text{relation types}\}$$
$$T_C = \{\text{containment relation type}\}$$
$$P = \{\text{properties}\}$$

We define a *view* $V = (C, R, T)$ as an hierarchical typed graph where C is the containment relationship, R is the of all the relations and T is the type relation for the vertices. A view conforms to its viewpoint and the properties P must hold. The definition is:

view
$$V = (C, R, T)$$
$$C = \{\text{containment relation}\}$$
$$R = \{\text{relations}\}$$
$$T = \{\text{type relation}\}$$

5.6 LEVELS OF PRECEDENCE

In Table 5.1 we list below the levels of precedence from the most bidding to the lowest bidding. Parentheses can be used to specify the order of operations.

Operand	**Name**
=	equal
$\subseteq$	subset
$\supseteq$	superset
$\subset$	proper subset
$\supset$	proper superset
$+, \cup$	union
$-, \cap$	intersection
$\circ$	relational composition
$\uparrow$	lifting
$\restriction$	left lifting
$\upharpoonright$	right lifting
$\Uparrow$	full lifting
$\downarrow$	lowering
$\times$	cartesian product
Id_X	identity relation
$dom(X)$	domain
$ran(X)$	range
$car(X)$	carrier
$R\|_{dom}X$	domain restriction
$R\|_{ran}X$	range restriction
$R\|_{car}X$	carrier restriction
$R \setminus_{dom} X$	domain exclusion
$R \setminus_{ran} X$	range exclusion
$R \setminus_{car} X$	carrier exclusion
$\top(X)$	top
$\bot(X)$	bottom
$R.S$	left projection
$S.R$	right projection
$R.y$	left image
$x.R$	right image
$^+$	transitive closure
*	reflexive transitive closure
$^-$	transitive reduction
$^{-1}$	inverse
^{n}	power

Table 5.1: The levels of precedence for the relational algebra operands (from the most to the less bidding).

CHAPTER 6

THE NIMETA FORMALISM

As far as the laws of mathematics refer to reality, they are not certain;
as far as they are certain, they do not refer to reality.

- Albert Einstein

6.1 INTRODUCTION

Architecture reconstruction is essentially an activity of reconstructing architectural views. The method method requires a solid formalism for describing the software models at various levels of abstraction: code, design and architecture. We seek a formalism that is sufficiently expressive, reasonably succinct, fairly natural to comprehend, easily scalable to describe the software architectures of industrial systems and suitable to fully automate the reconstruction process. After reviewing and experimenting the existing approaches, we have decided to base our formalism on the the binary relational algebra that we have introduced in Chapter 5. In Section 5.5 we have also formalized the concept of views and viewpoints that was introduced in Section 2.3. In this chapter we describe and formalize the well-accepted viewpoints from the literature. They represent a catalogue of reference viewpoints for architecture reconstruction.

6.2 OVERVIEW OF THE VIEWPOINTS

Although in the literature a number of architectural views has been proposed, there is no consensus on what views are commonly needed for forward architecting. Some views from

different authors are actually very similar while other views are focused on very specific concerns. In our experience, every system is a new architecture with its own architectural styles, rules, codes and policies. As (Perry and Wolf, 1992) noted, even if the system was designed according to standard architectural styles, the ubiquitous customization of architectural elements turn the system into an unique creation. Software organizations and even single teams tend to develop their own set of views that work in their own limited domain.

An important contribution of our work is that the architecture reconstruction process is independent of the viewpoints. We are not imposing our own viewpoints but we can tailor the reconstruction process to the existing ones. Nevertheless, we believe that it is necessary to define a minimal framework of viewpoints to provide a solid foundation to the method. The reference viewpoints can be used when no other viewpoints are available or they can complement the viewpoints that are already in use.

We have organized the viewpoints in three distinct layers: code, design and architecture. Viewpoints in the higher layers are less detailed and architecturally relevant. The viewpoints in the lower layers are rich in details and reflecting the implementation. This division reflects the levels described in Section 2.1 and the division propose by (Eden and Kazman, 2003). During the Dagstuhl seminar on the interoperability of reverse engineering tools we also proposed a similar three-layer framework for the reverse engineering meta models(Ebert *et al.* , 2001). The Figure 6.1 shows the three layers of viewpoints and their mappings. The mappings between the various elements of the layers allow us to to establish the traceability of the models. We briefly introduce the layers below:

Architecture At the top of the framework, there are the *architectural viewpoints* that describes the architecturally relevant aspects of the system. Even if the field is not mature enough, the recent book of (Clements *et al.* , 2003) tries to provide a unified set of architectural viewpoints that we take as a reference set for the architecture layer.

Design Zooming on details, we have the *design viewpoints* that captures the essence of the important design aspects without the overwhelming details of the code viewpoint. Typically only limited information is available like classes, attributes and method bodies, method invocations, variable accesses and inheritances. This information is sufficient for dependency analysis. The approaches in the literature are mainly focused on supporting particular reverse engineering tools, as we have discussed in Section 3.3. (Tichelaar, 2001) has elaborated a language independent meta-model for modeling object-oriented software called FAMIX. We take FAMIX as a reference viewpoint for the design layer.

Code The *code viewpoints* give a very detailed representation of the syntactic structure

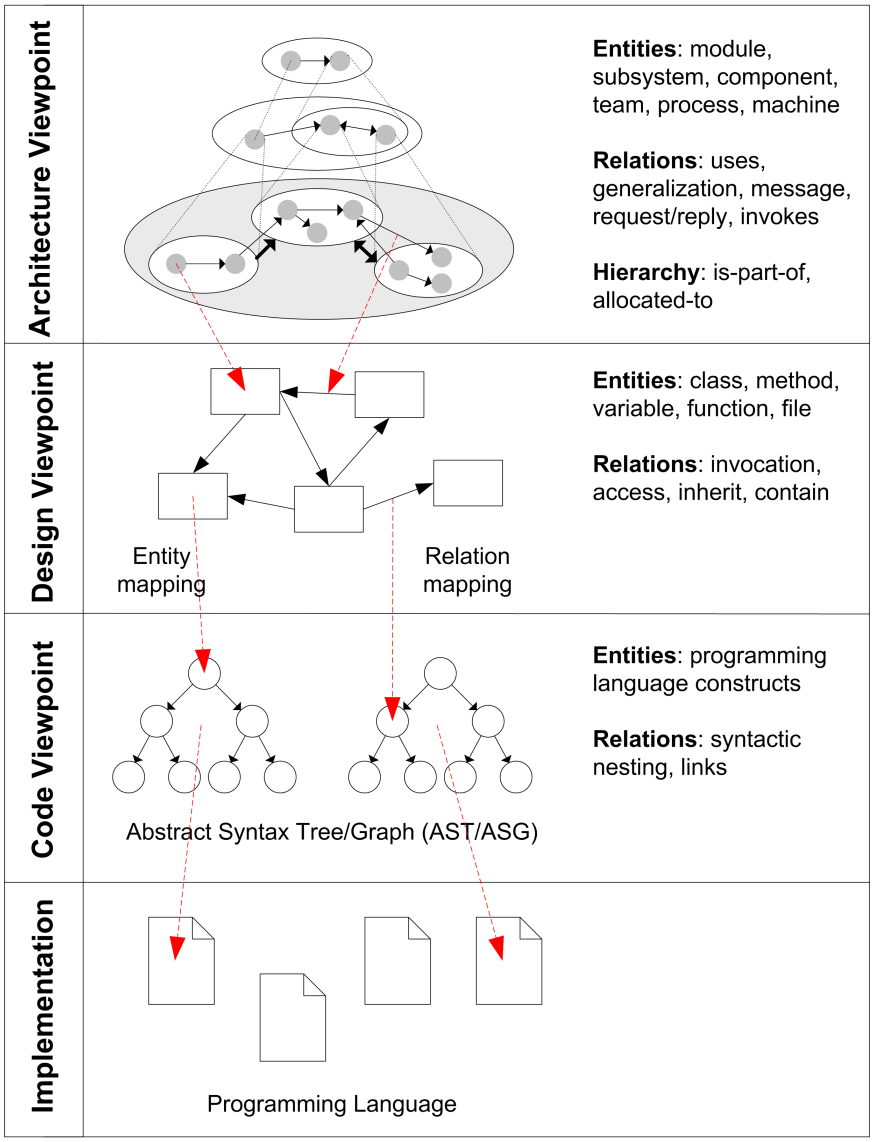


Figure 6.1: The framework of viewpoints.

of the source files. We should interpret the code viewpoint as a precise representation from where the source files can be regenerated. This is typically modeled with the abstract syntax tree (AST) level information. It is normally detailed enough to regenerate the source code and sufficient for control-flow analysis.

Implementation At the bottom level, we have the *implementation* of the system that consists of a set of documents like source files, configuration files, build commands, log files and so on.

6.3 THE CODE VIEWPOINT

The code viewpoint provides a very detailed representation of the source code. We can represent it with an abstract syntax tree (AST) or an abstract syntax graph (ASG). The purpose of the AST is to describe the syntactic decomposition of a software program with a tree-nesting structure. The level of details of the AST is variable and it depends on the intended use. For example, compilers ignore certain information like comments and brackets that is required if we are interested in reproducing the original source code. For our purposes, the AST of the code viewpoint should provide a one-to-one mapping with the source code.

We do not provide an explicit description of the source viewpoint because it is not in the scope of our dissertation. The reason is that there are extractors for generating the design views that we can use to create the architectural views. However, we believe that for completeness and traceability the viewpoints in the higher layers should be based on the code viewpoint. As a reference viewpoints we take the schemas that we have discussed in Section 3.3.

6.4 THE FAMIX DESIGN VIEWPOINT

The design viewpoint describes the essence of a software program at an abstract level. We are interested in a representation that is close to the source code but less detailed. It must be abstract but authentic in a wait that we can establish a one-to-one mapping between elements in the code and the elements in the design view.

We take the FAMIX[1] meta-model as a reference viewpoint for the design layer. The FAMIX meta-model models the design aspects of object oriented software in a language-

[1]FAMIX 2.1 specification is available at: http://www.iam.unibe.ch/ famoos/FAMIX

independent way (Demeyer *et al.* , 2001). It consists of a *language-independent core* that captures the common aspects of objected-oriented languages and various *language extensions* that deal with the specific aspects of the languages. There are extensions for Java, C++, Smalltalk and Ada. The FAMIX meta-model allows us to capture the design-level aspects of the implementation.

In our cases we mainly analyze C/C++ and Java code, we have condensed some aspects of the core and the extensions in a unique meta-model or schema. We have also simplified certain aspects of the original specification. The FAMIX meta-model is intended for achieving tool interoperability while it does not need to correspond internal implementation of a reverse engineering environment. The differences between our customized version and the original FAMIX meta-model are minimal and do not break the tool interoperability goal of FAMIX. Throughout the thesis, we take the FAMIX meta-model as a reference meta-model for several case studies written with C/C++/Java languages.

We refer to the FAMIX specification for a detailed explanation of the FAMIX meta-model. In this section, we formalize the FAMIX meta-model in terms of the binary relational algebra. As through this dissertation we only use our relational algebra variant defined below, we will simply refer to it as FAMIX without any ambiguities. The diagrams in Figure 6.2 show the basic entities and relationships of FAMIX and the Table 6.1 gives a detailed explanation of the entities and relations of the design viewpoint.

For the naming conventions, we use the FAMIX rules:

- Naming of attributes: the name is obtained by concatenating the containing class with attribute name using a '.' as a separator: e.g. `Foo.var`.

- Naming of functions: the signature of a function contains the name of the function followed by the list of parameter types: e.g. `foo(int, char*)`.

- Naming of methods: the unique name is obtained by concatenating the name of the class with the signature of the method: e.g. `Foo.print(char*)`.

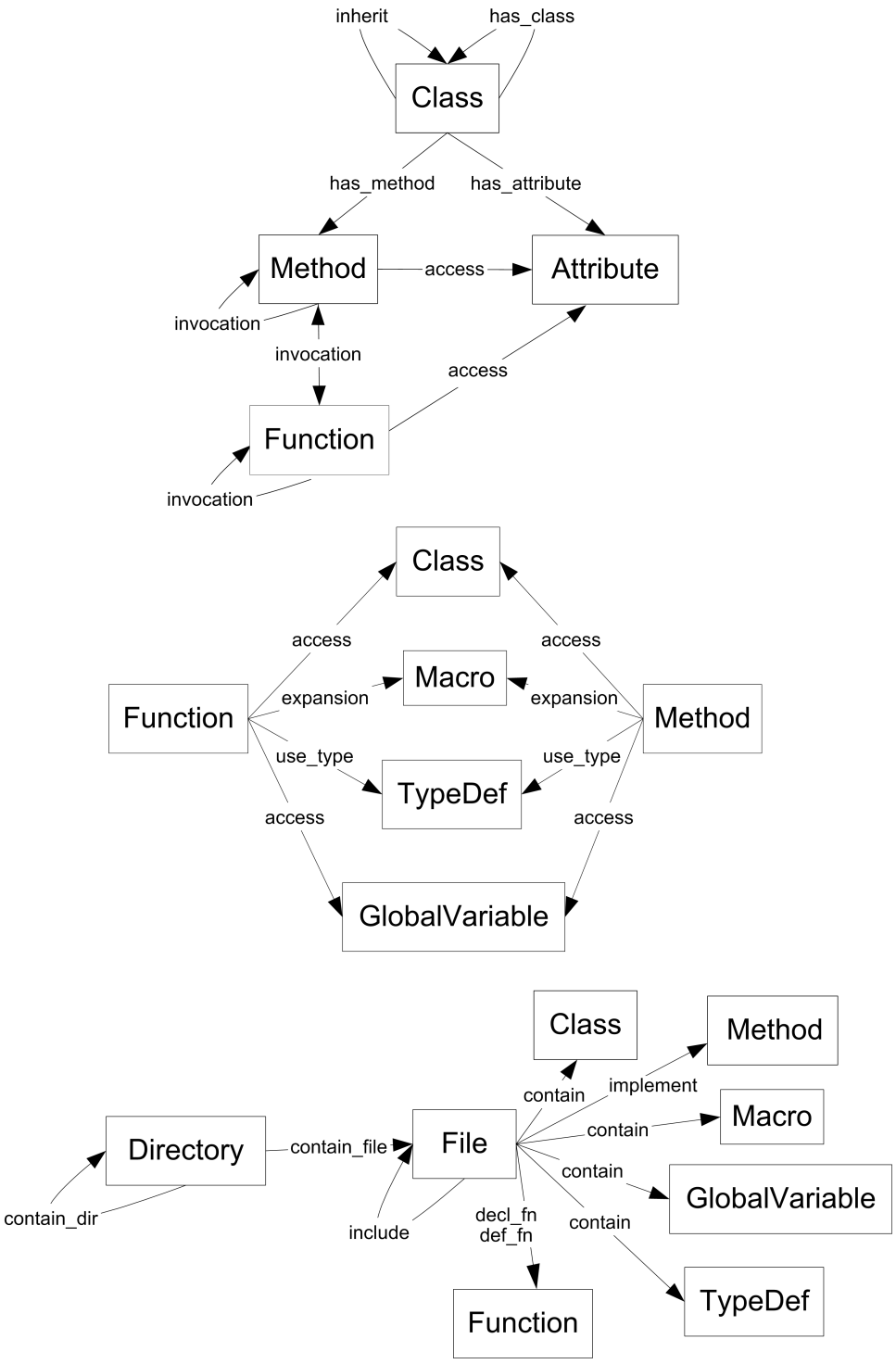


Figure 6.2: The simplified FAMIX meta-model.

Entities		
Entity	**Description**	**FAMIX meta-model**
`Class`	The definition of the class	Class
`Method`	The definition of method of a class	Method
`Attribute`	The definition of an attribute of a class	Attribute
`Function`	The definition of a function or a procedure that has a global visibility	Function
`Macro`	A C++ macro definition with `#define`	
`TypeDef`	A C++ type definition with `typedef`	TypeDef (C++ plugin)
`GlobalVariable`	The definition of a global variable	GlobalVariable
`File`	A source file	
`Directory`	A directory in the file system	

Relations		
Relation	**Description**	**FAMIX meta-model**
has_method	A class declares a method	inverse of the property belongsToClass
has_attribute	A class declares an attribute	inverse of the property belongsToClass
has_class	A class declares a nested class	
inherit	A class inherits from another class	InheritanceDefinition
invocation	A method or a function invokes a method or a function	Invocation
access	A method or a function access an attribute or a global variable	Access
include	A source or header file includes an header file	Include (only C++ plugin)
expansion	A function or a method expands a macro	
use_type	A function or a method use a user's defined type	
contain	A file contains the definition of a class, a macro, a type definition and a global variable	the sourceAnchor property
implement	A file defines the implementation of a method	
decl_fn	A file declares a function	
def_fn	A file defines a function	
contain_file	A directory contains a file	
contain_dir	A directory contains another directory	

Table 6.1: The entities and relations of the design viewpoint.

We provide an example with the following fragment of Java code:

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet
{

        public void paint(Graphics g) {
                g.drawString("Hello world!", 50, 25);
        }
}
```

The design view for the Java code is the the following set of relations:

$$
\begin{aligned}
type &= \{ \ (\text{'Applet'},\text{' Class'}), \\
&\quad (\text{'HelloWorld'},\text{' Class'}), \\
&\quad (\text{'Graphics'},\text{' Class'}), \\
&\quad (\text{'HelloWorld.paint(Graphics)'},\text{' Method'}), \\
&\quad (\text{'Graphics.drawString(String, int, int)'},\text{' Method'})\} \\
has\_method &= \{ \ (\text{'HelloWorld'},\text{' HelloWorld.paint(Graphics)'}), \\
&\quad (\text{'Graphics'},\text{' Graphics.drawString(String, int, int)'})\} \\
inherit &= \{ \ (\text{'HelloWorld'},\text{' Applet'})\} \\
invocation &= \{ \ (\text{'Graphics.drawString(String, int, int)'},\text{' Graphics.drawString(String, int, int)'})\}
\end{aligned}
$$

6.5 THE ARCHITECTURE VIEWPOINTS

In this section, we formalize the commonly-accepted architectural views with the binary relational algebra. As a reference catalogue, we take the architectural views described in the book of (Clements *et al.* , 2003). In the practical cases, we often need to customize the views or to create new views. The goal of this section is to provide a reference material to start with. We point out that the reference viewpoints are rather generic and open to various interpretation. The interpretation is often dependent on the team or the organization. Table 6.2 summarizes the architecture viewpoints that are presented in the following sections.

Viewtype	Viewpoint	Entity	Relationship
Module	Decomposition	module, subsystem	is-part-of
	Uses	module	uses
	Generalization	module	is-a
	Layered	layer	allowed-to-use
Component & Connector	Pipe & Filter	filter, pipe	bindings
	Shared-Data	shared-data store, data-accessor	r/w connector
	Publish-Subscribe	component (publisher, subscriber)	publish-subscribe, event bus
	Client-Server	client, server	request/reply message
	Peer-to-Peer	peer	invokes-procedure
	Communicating-Processes	unit, task, process, thread	data exchange, message passing, synchronization, control
Allocation	Deployment	component physical element	allocated-to
	Implementation	module, configuration item, (file, directory)	containment allocated-to
	Work Assignment	module, organizational unit (a person, a team, a department, a subcontractor)	allocated-to

Table 6.2: The architectural viewpoints from the catalogue.

6.5.1 THE MODULE VIEWTYPE

The category *Module Viewtype* groups the architectural views that document the modular structure of system's software. The key concept is the *module* that represents a software unit, with a well-defined interface that provides a coherent unit of functionality. The interpretation of a module can vary from system to system and often depends on the developers' decisions how to modularize the system. Any non-trivial software system is partitioned into separated software parts that typically consists of functions, files, classes or other computational elements. An aggregate of modules is often called a *subsystem*. Beside the general understanding of the concept of module and subsystem, we assume that its precise definition is system-dependent and it is discovered during the reconstruction activity.

The **Decomposition** viewpoint describes the partitioning of modules and subsystems in their constituents. It shows the hierarchical decomposition of the system in modules and their responsibilities within the system. The decomposition viewpoint conveys a general view of the system and is one of the favorite mechanism for communicating the structural organization of a software system to a variety of shareholders. Especially for large systems, the decomposition viewpoint provides a condensed view of the main parts of the software. There are no strict rules for the construction of this viewpoint and often the interpretation of its semantic can only be found in the design practises of the organization. This viewpoint is often overlapping with the allocation viewpoints.

The **Uses** viewpoint shows the interdependencies among the modules and tells developers what other modules must exist in order for a certain part of the system to work correctly. A module A *uses* another module B if the correctness of A depends on the correct implementation of B. This implies that changes in module B will affect the behavior of A. The *uses* relation is a specialization of the *depends-on* relation and should not be confused with other *depends-on* relations (e.g. *calls, inherit-from, include-from*). A module A can include a module B but not necessarily use it at runtime. The *uses* dependencies create constraints for the whole development process from design to testing (e.g. module A cannot be tested without module B). The uses viewpoint is an important tool for the planning of the development process and assessing the effects of changes in the system. In the next sessions, we will show that his viewpoint plays a key role in the architecture reconstruction method.

The **Generalization** viewpoint describes the commonalities and variations among the modules by defining a generalization relationship among them. A parent module is a more general version of the children modules when the parent owns certain aspects that are common to all the children. The variations (or specializations) are manifest only in the children modules. The generalization relationship implies the inheritance of the interfaces and, at some extent, of the implementation. While the generalization viewpoint is derived from the object-oriented designs where generalization is a widely used mechanism, there are numer-

Viewpoint	T_E	T_R	T_C	P
Decomposition	module, subsystem	$\emptyset$	is-part-of	is-part-of *is a tree*
Uses	module, subsystem	uses	$\emptyset$	$\emptyset$
Generalization	module	generalization, is-a	$\emptyset$	generalization *is acyclic*
Layered	layer	allowed-to-use	$\emptyset$	allowed-to-use *is acyclic*

Table 6.3: The viewpoints of the module viewtype.

ous applications for describing the architectures of product families (i.e. the commonalities and variations of the various components).

The **Layered** viewpoint describes the organization of the software into units where each unit belongs to a layer. A layer represents a virtual machine in the sense that it provides a cohesive set of services through a set of public interfaces. The layered organization of the units enforce a strict ordering of usage: if the layer A is above the layer B, the units in the layer A are allowed to use the services provided by the units in the layer B. In practise we can find several layering schemes that can allow more or less freedom for the *allow-to* relation, however one rule must always hold for a layered architecture: lower layers cannot use without restrictions the facilities of higher layers. The layered viewpoint is commonly used for partitioning a software and, although it is often not directly documented, it is implicitly used in other views like the decomposition viewpoint.

Table 6.3 summarizes the various definitions of the module viewpoints, according to the viewpoint definition in :

viewpoint $V_P = (T_E, T_R, T_C, P)$

6.5.2 THE COMPONENT & CONNECTOR VIEWTYPE

The viewpoints in the category *Component & Connector viewtype* concern with the modeling of elements that have a runtime presence, such as clients, servers, processes and databases. The key concepts of the viewpoints are the *components* and *connectors*. Components represent runtime visible processing units or data storage elements. Connectors are the communication mechanisms that allow the components to interact. There are various

types of components and connectors with different properties that are defined in the specific viewpoints. Table 6.4 summarizes the various definitions of the component & connector viewpoints.

The **Pipe-and-Filter** viewpoint models a system whose main logic is characterized by a sequence of transformations of streams of data, e.g. signal processing applications. The initial filter receive a large amount of data that is processed by a cascade of filters. The component type is the *filter* that transforms the data that it receives from the input ports and deliver it to the output ports. The connector type is the *pipe* that provides the unidirectional connection between the pipes.

The **Shared-Data** viewpoint models a system that is characterized by the exchange of persistent data. One or more shared databases store the data that other components can access. The main components are the *shared-data store* and the *data accessor* that is a computational unit. The connectors are the *data reading* and *data writing* connectors. For this viewpoint, the connectors are formalized as a binary relation.

The **Publish-Subscribe** viewpoint models a system where the components can register to a communication bus where they can publish an event and subscribe for receiving events. The communication bus is responsible for delivering the events to all the subscribers. The component types are the *publisher* and the *subscriber*. The connector type is the publish-subscribe relationship, formalized as a binary relation.

The **Client-Server** viewpoint models a system where the components can request services of other components. The servers provide a set of services that the clients can request through their interfaces. The communication is always initiated by the clients and can be synchronous or asynchronous. The component types are the *client* and the *server*. The connectors are the *request* and *reply* messages and are modeled as a binary relation.

The **Peer-to-Peer** viewpoint models a system where the components, called peers, can directly interact to exchange services. Any peer can request another peer for services. The communication mechanism is similar to the request/reply interaction of the Client-Server viewpoint but the interaction may be initiated by any peer. Systems buit with object distributed infrastructure like CORBA, COM+ and Java RMI are examples of peer-to-peer systems. The component type is the *peer* such as distributed objects and the connector type is the *invokes-procedure*.

The **Communicating-Processes** viewpoint models the concurrency aspects of a system where the components interact through various mechanisms of communications such as synchronization, message passing, data exchange and so on. The component type is a *concurrent unit* such as tasks, processes and threads. The connector types are the communication mechanisms and are formalized as binary relations.

Viewpoint	T_E	T_R	T_C	P
Pipe-and-Filter	filter, pipe	bindings	$\emptyset$	$\emptyset$
Shared-Data	shared-data store, data accessor	read, write	$\emptyset$	$\emptyset$
Publish-Subscribe	publisher, subscriber	publish-subscribe	$\emptyset$	$\emptyset$
Client-Server	client, server	request, reply	$\emptyset$	$\emptyset$
Peer-to-Peer	peer	invokes-procedure	$\emptyset$	$\emptyset$
Communicating-Processes	unit, task, process, thread	data exchange, message passing, synchronization, control	$\emptyset$	$\emptyset$

Table 6.4: The viewpoints of the Component & Connector viewtype.

6.5.3 The Allocation Viewtype

The category *Allocation viewtype* concern with the organizational aspects of a software system like hardware allocation, development team structure and file system structure. The viewpoints in this category document how the various modules and components are mapped to elements in the development environment. The key concepts are the software elements (e.g. modules and components) and the *allocated-to* relationship. Table 6.5 summarizes the various definitions of the allocation viewpoints

The **Deployment** viewpoint describes the allocation of the runtime elements to the execution platforms. The performance requirements are met by the particular constraints expressed with this viewpoint. There are two categories of elements in this viewpoint: software elements and environmental elements. The software elements are the runtime units typically from Component & Connector viewpoints. The environmental elements correspond to the physical units that store, transmit or compute data such as processor, memory, disk, network, communication channels and so on. The typical relation for this viewpoint is the *allocated-to* that shows how the software elements are allocated to the physical units. This relation is dynamic because it can the allocation of software elements can change at runtime. There are also variations of the basic relation: *migrates-to*, *copy-migrates-to* and *execution-migrates-to*. The topology of the relation is not fixed but in practise we can interpret the *allocated-to* relation as a containment relation.

The **Implementation** viewpoint maps the modules of the module viewpoints to the development infrastructure. Modules are implemented with many separate files that are typically

Viewpoint	T_E	T_R	T_C	P
Deployment	component, physical element	$\emptyset$	allocated-to	allocated-to *is a forest*
Implementation	module, configuration element	$\emptyset$	containment allocated-to,	containment, allocated-to, *is a forest*
Work Assignment	module, organizational element	$\emptyset$	allocated-to	allocated-to *is acyclic*

Table 6.5: The viewpoints of the allocation viewtype.

organized in a hierarchy of directories and stored in a configuration management system. The files contain the source code, definitions, configuration instructions, build commands and all the data that are necessary for creating the final executables. The elements of this viewpoints are the module and the configuration elements like a file or a directory. There are two types of relations: *containment* and *allocated-to*. The *containment* relation specifies that one configuration element contains another element. The *allocated-to* relation describes the allocation of modules to the configuration elements. In relational algebra, the two containment relations are considered to be forests.

The **Work assignment** viewpoint describes the allocation of the development work across the development teams and the organizational structure. The viewpoint documents the mapping between the software elements and the groups of humans who are responsible for the design, development, maintenance and testing activities. The elements of the viewpoint are the the *module* and the *organizational unit* such as a person, a team, a department a subcontractor and so on. The *allocated-to* relation maps the software elements to the organizational units.

6.5.4 THE FEATURE VIEWPOINTS

The previous viewpoints are mainly concerned with the structural aspects of the architecture as they describe the potential interactions among the components. We refer them as *static viewpoints*. If we want to describe the behavior of the system at runtime, we have to introduce a time line. For each viewpoint we can define its related *dynamic viewpoint* by associating a time line. We use the dynamic viewpoints to document the behavior of the architecture. In a dynamic viewpoint only a subset of the potential interactions found in the static viewpoint are active at a given point in time. Some behavioral characteristics of the system can only be documented with the run-time descriptions: concurrency, deadlocks, synchronization, timing, real-time constraints, performance and so on. Documenting the

behavior is as important as documenting the static aspects. (Stewart, 1999) puts the lack of measurement of execution time in his lists of 30 pitfalls for real-time software developers.

Architects document the behavior to show how a particular element behaves under certain circumstances or how it interacts with other elements. There are various notations for the documentation of the behavior: use cases, sequence diagrams, collaboration diagrams, message sequence charts, statecharts, Petri nets and so on. From our experience with Nokia's systems, we have found important to recover the *feature viewpoints* from the system execution. The feature viewpoint describes the implementation of one or a set of features at a particular level of abstraction. The purpose of the feature view is to describe how the elements of a static viewpoints interact in order to accomplish the goals of the feature. For the representation of the feature view, we use the sequence diagrams. The participants are the elements of the viewpoint and the messages represent the run-time interactions. We also support the possibility of grouping participants in abstracted elements as in the static viewpoints. This allow us to calculate the run-time dependencies. The types of abstractions are described in the Section 8.5.2.

CHAPTER 7

THE NIMETA ARCHITECTURE RECONSTRUCTION PROCESS

Design and programming are human activities;
forget that and all is lost.

- Bjarne Stroustrup

Similarly to archeology, architecture reconstruction is the process of analyzing the evidence of past design decisions for inferring an architectural model of the system. In NIMETA 's approach the architectural concepts play a key role in the reconstruction. The outcome is a set of reconstructed architectural views. This chapter presents the NIMETA reconstruction process. After an introduction, we define the types of views and present the reconstruction process. We conclude with the description of the maturity levels and an example.

7.1 INTRODUCTION

Software architecture reconstruction is the process of obtaining a documented architecture for an existing system by inferring the architectural information from the available evidence. The most reliable source of information is the system itself, either the source code or traces obtained from its execution. Other sources of information are the design documentation, the description of supported features, interviews with the system experts. In general, architecture reconstruction brings together the reliable data from the system implementation with less-formalized domain knowledge.

We can define the following goals for architecture reconstruction:

- To create an architectural model that makes the concrete architecture explicit from the implementation. By confronting the reconstructed architectural model with the conceptual models, the developers can increase their understanding of the system.
- To enable the architects to analyze the various structural dependencies among the software components.
- To enable the architects to check the architectural conformance of the implementation.
- To enforce the architectural rules as they are formulated by the architects.

As every system as its own architectural style, we tailor the reconstruction process around the architectural concepts. Together with the architects, we carefully select the architectural concepts that we need to recover and present in the final views. We also define how the architectural concepts map to the implementation (following the framework of viewpoints defined in Chapter 6). This guarantees that we can deliver meaningful abstract models to the architects.

The outcome of the reconstruction is a set of architectural views. With the stakeholders, we define what are the significant viewpoints to present and the presentation format. The content of each viewpoint is carefully defined according to the interests of the stakeholders. We can either rely on the viewpoints already in use or the standard viewpoints defined in Section 6.5.

Reconstructing a software architecture requires an active involvement of the experts and stakeholders such as developers, architects, management, designers and testers. The most experienced people are often difficult to reach but they are the most valuable especially at the beginning for setting the goals of the reconstruction. For this reason, there must be a compelling reason for starting a reconstruction and this reason has to be well-understood and accepted by the management. The reconstruction of an architecture is an activity that can last from months to years and it has to be properly financed. The maximum benefits are obtained when the reconstruction is well-integrated with the forward engineering practise and there is a continuous exchange of information. For this other reason, it is very important that the reconstruction is defined and monitored by the most experienced people in the organization.

We need a flexible approach that can be well customized to the subject system. Differently from forward engineering where there is a large degree of freedom in the choice of the most appropriate views to design a system, we have to deal with design practises that are already in place in the organization and we need to reconstruct them as similar as possible. Our method for architecture reconstruction consists of the following activities:

1. Problem definition
2. Concept determination
3. Data gathering
4. Knowledge inference
5. Presentation
6. Conformance checking
7. Architecture assessment
8. Re-documentation

The activities are discussed in details in the following sections. In (Riva, 2000) we have presented an initial study of the reconstruction process that has been formalized in (Riva, 2002). Our initial approaches have been unified in Symphony reconstruction process (van Deursen *et al.* , 2004).

7.2 Source, Target and Hypothetical Views

In the Chapter 6 we have defined different categories of viewpoints: code, design and architecture. The views based on those viewpoints represent the final and intermediate material of the reconstruction process. We define three types of views that are involved in the reconstruction process.

The *source view* is a view of the system that can be extracted from its artifacts such as source code, build files, configuration information, documentation or traces. The source view is based on a code viewpoint or a design viewpoint (such as abstract syntax trees or flow graphs). The source views represent the basic information for creating the architectural viewpoints.

The *target view* represents one goal of the reconstruction process that has been agreed with the stakeholders. The target view describes the as-implemented aspects of a software system and it is needed to solve the problems or to perform the tasks for which the reconstruction process was carried out. The target views are typically architectural views.

The *hypothetical view* is a view of the system that is not so accurate but represents the current understanding or desires of the developers of the system. The hypothetical view

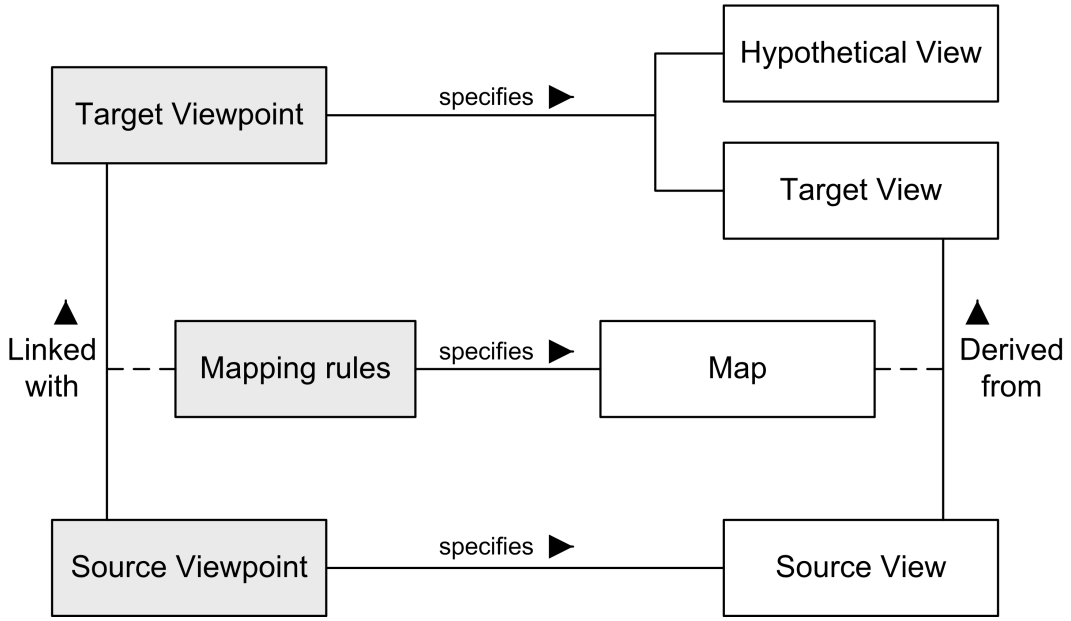


Figure 7.1: Views and Viewpoints.

describes the as-designed or as-intended aspects of the architecture. This view is typically an architectural view and is created by interviewing the experts or from the existing documentation. The hypothetical view can be serve two purposes: (1) it can serve as a reference design against which to check the implementation or (2) it can represent the postulated architecture and the current understanding of the system. The hypothetical view is used to guide the reconstruction process.

The *mapping rules* define the rules for creating the target view from the source view. The rules can be either formalized with in relational algebra or defined empirically.

The *map* is the actual mapping between a source view and a target view.

7.3 The Architecture Reconstruction Process

The goal of the reconstruction process is to recover the architecturally significant views that satisfy the needs of the stakeholders of the system. The process has been designed to fulfill this main requirement and the other requirements described in Chapter 4.

The actors of the reconstruction are responsible for conducting the reconstruction process,

interested in the result or represent a source of information. The main actors are:

stakeholders who hold a particular interest in the system and in the result of the reconstruction.

process designer who is responsible for defining the goals of the reconstruction and the viewpoints to recover. For generality, we assume that the process designer is competent with the reconstruction process but only familiar with the architecture under analysis.

reconstructor who is responsible of recovering the target views from the implementation. The reconstructor follows the instructions defined by the process designer.

system experts who represent the main experts and source of information for the various parts of the system, such as as architects, chief designers and programmers.

The reconstruction process is an iterative incremental process that is divided in three phases: *process design*, *view recovery* and *result interpretation*. We describe the three phases below. Each phase consists of several activities. The diagram in Figure 7.2 shows the dataflow and the three phases of the reconstruction process.

The first phase is the **Process design**. During this phase, the process designer elicits the architectural problems from the stakeholders and defines the source and target viewpoints for the reconstruction. The process designer is also responsible for defining the architecturally relevant concepts and their mappings to the implementation. The phase consists of two activities: *problem definition* and *concept determination*. This phase is a conceptual activity that is mainly conducted through workshops with the stakeholders and interviewing the system experts.

During the second phase, **View recovery**, the reconstructor is responsible for recovering the target views that have been defined in the process design phase. The reconstructor has to carefully analyze the available artifacts in order to gather the required data for creating the target views. This phase can be partly automated with tools and require the help of the system's experts in order to be conducted efficiently.

The third phase is the **Result Interpretation**. During this phase, the reconstructor interprets the results of the reconstruction with the stakeholders. One typical activity is to check the conformance of the reconstructed views against the hypothetical views or against certain architectural rules that have to be satisfied in the implementation (architecture conformance checking). The results of the reconstruction are a reliable source of information for an

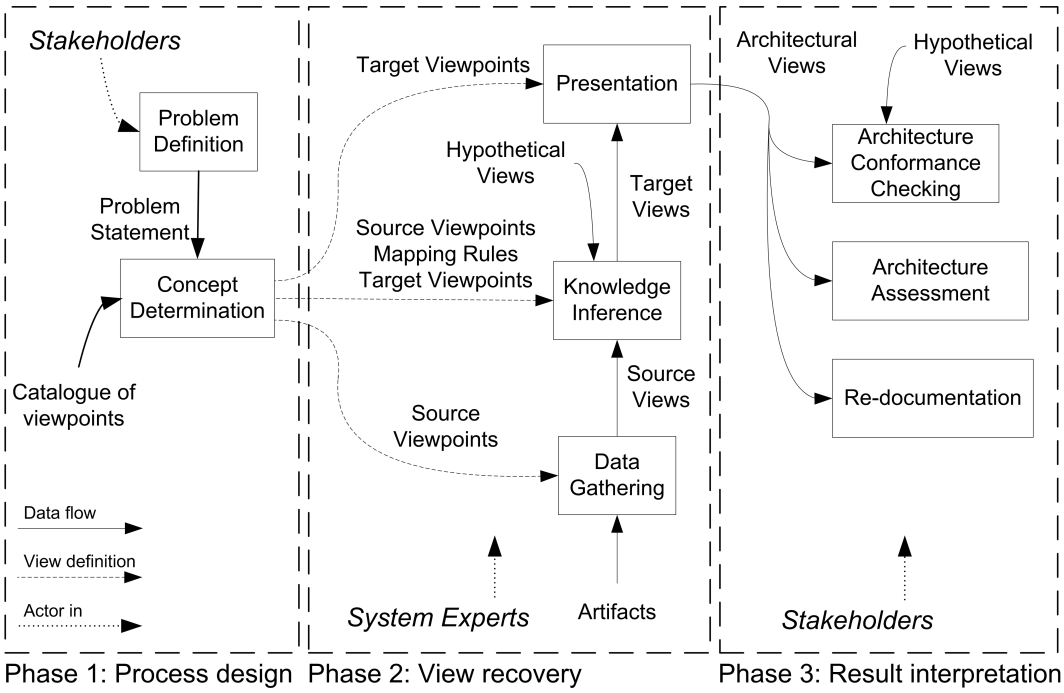


Figure 7.2: The reconstruction process.

assessment of the architecture against future requirements or quality attributes. The architectural views also used for re-documenting a system that had an outdated or not-existed architectural documentation.

7.3.1 PROBLEM DEFINITION

Reconstructing an architecture requires the active involvement of the stakeholders and the system experts from the very beginning. The first activity, problem definition, concerns with bringing up the motivations for the architecture recovery. The process designer, playing the role of a detective, has to investigate what problems in the current development practice could be solved by a documented architecture description. Typically, the main motivation is to support an efficient evolution of the software system.

The investigation is conducted by interviewing the stakeholders and by organizing workshops with the experts. It is often the case that contrasting perspectives emerge from people with different interests on the system: (1) the short-term view of the management that is worried about how quickly new feature can be introduced, how many new products added to the family, costs and resources, (2) the long-term view of the architects who are concerned

about the quality aspects of the architecture (robustness, performance, reliability, availability) and (3) the practical view of the developers who are facing concrete problems with the architecture (e.g. unclear component dependencies, unclear interfaces, disorder in the code disorder, database administration). During these discussions, the architects should play an active role in solving the contrasting views between the programmers and management.

During the discussions, the process designer has to collect the requirements for the reconstruction activity, paying attention the reliability of the different perspectives and validating them through multiple sources. The process designer should also pay attention to distinguish between real facts about the system and the suggestions for future improvements. People are often reluctant to discuss the shortcomings of their own software, especially when confronted with other colleagues. Other people often try to move the discussion towards what they believe to be the hot topics, and sometimes they see it as a chance to propose their own solutions. Although proposal for future changes are important, they should be avoided at this stage. The goal of the reconstruction is to discover the actual implementation rather than proposing how it should be. The process designer should act objectively and abstain from judging the design (he might not even have the competence for that). The assessment of the architecture is conducted in the third phase when the concrete architectural model is available.

The process designer also needs to document the architecturally significant requirements (ASRs) of the system. Understanding the ASRs allow us to understand the motivations behind certain design decisions and the critical aspects of the system.

Several activities can benefit from an up to date architecture: architecture design, dependency checking, assets management, component reuse, software management, platform maintenance and test organization. The process designer needs to identify what activities require an improvement and they can be achieved. For this purpose, it is common to define a set of key use cases that demonstrate how the reconstructed architecture will be used. In the next activity, the architectural views will be derived also from the use cases defined at this point.

There are several techniques that can be used during the problem definition: structured workshops, checklists, role playing and scenario analysis. At least, one iteration is necessary for defining the requirements or the reconstruction and getting them approved by the the stakeholders. The most experienced people of the team and from different ares should be invited to the events.

There are also several organization issues that should be considered. The process designer needs to know who are the contacts for the technical questions, how they can be reached and who are the final users of the reconstructed architecture (e.g. only the architects or all the developers). The process designer and the stakeholders need to define how the people

will be trained to use the models, how the technology transfer will happen, who will be in charge for the future maintenance of the models, the priorities and the schedule for the reconstruction. All these aspects must be agreed and approved by the management.

The outcome of this step includes an approved reconstruction plan; a memorandum of the problem statement with the list of problems and expected results (preferably expressed in a terminology familiar to all the stakeholders); summaries of interviews, workshop sessions, relevant discussions and use cases; summaries of high-level documentation if available; an elaboration of the problem statement based on the summaries; the list of documentation, resources, contacts that are available during the reconstruction.

The duration of this activity depends on the maturity level of the organization. A well-defined architectural team is the ideal solution because they architects have a broad view on the system and should know remedies are needed. If the architects are not available, they should be nominated. The team should at least be at the maturity level 1 (see Section 7.4).

The process designer should preferably be external to the development team in order to be able to take an objective and independent view on the architectural problems. He or she should have good communication skills, be an analytic thinker and experienced with architecture reconstruction.

Below there is a summary of the problem definition activity.

Objective: is to define the problems that a reconstructed architecture description should solve (the goals of the reconstruction).

Input: discussions with the stakeholders and system experts.

Actors: the process designer who is responsible for conducting the discussions and summarizing the problem statement; the stakeholders who can motivate the needs for the reconstruction and what are the expected results; the system experts who can be interviews for the technical aspects.

Techniques: workshops, checklists and scenario analysis.

Output: memorandum of the problem statement, approved reconstruction plan, summaries of workshops and interviews.

7.3.2 Concept Determination

Software systems are built according to a particular architectural style that is the unique result of the design decisions taken over a long period of time. The architectural style comprises design decisions about the architectural concepts that are used to build the system: the types of building blocks that can be used to compose the system (e.g. components, classes, applications) and the communication infrastructure that enables the components to interact at runtime (e.g. software busses, remote procedure calls, function calls). The style can vary as the architecture needs to change and adapt to new scenarios. At a certain point of time, the architectural style of a system represents the best effort the developers made for satisfying the architecturally significant requirements. Changes to the concepts are often difficult because they may impact various parts of the the architecture. The architectural concepts represent the way developers think of a system. Since our intention is to deliver architectural views that are as close as possible to the mental models of the developers, the architectural concepts are the first-class entities of the reconstruction and they represent the terminology of the reconstruction.

The goal of this activity is to recover the architectural concepts and to define the terminology of the reconstruction. The main actor is the process designer who is responsible for determining the source, target and hypothetical viewpoints and the mappings between them. The architectural concepts are the elements that populate the viewpoints. We break this activity in three steps that are described below.

Identification of the architectural concepts. The first step is to recover and clarify the architectural concepts and to document them. The architectural concepts vary from one system to another: in a distributed software system the architectural concepts may be applications, servers, software busses while in an operating system they may be tasks, processes, queues, shared memories, etc.

We need to recover three aspects: the architectural types, their relationships, and their mappings to the implementation (e.g. classes, functions, variables, files). This step is conducted in collaboration with the system experts and by reading the available documentation. One technique is to organize a series of workshops and ask the experts to explain the implementation of a set of key features. By describing those scenarios, we can identify the run-time elements, hence, the architectural concepts. The process designer should pay a particular attention to the vocabulary used by the developers i.e. design patterns, design conventions, micro-architectures).

Define the target viewpoints. The next step is to define the target viewpoints whose instances, the views, have to be reconstructed. The target views are used to solve the problems stated in the problem definition activity and they should be agreed with the stakeholders.

The viewpoints are taken from the catalogue of well-known viewpoints presented in Section 6.5 or, as often happens, ad-hoc viewpoints are created for a specific reconstruction.

The technique that we follow for creating the target viewpoints is based on the Stakeholder/Views table described in (Clements *et al.* , 2003) and refined in (van Deursen *et al.* , 2004). The first step is to produce a list of candidate views with an indication of the stakeholders that can benefit and to what extent it can solve a particular problem. The stakeholders know which viewpoints will be useful or at least they have an initial idea. Typically, the module viewpoints (especially the Decomposition viewpoint) are valuable to the architects, project managers, members of the development team, testers, integrators and maintainers. The Component & Connector viewpoint addresses the needs of architects, maintainers and development teams. The Allocation viewpoints addresses the organization problems and are more important to project managers and architects. The second step is to combine views in order to create a manageable set of consistent views. In our cases, we often merge the Uses view with a containment view such as the Decomposition viewpoint or an Allocation viewpoint. The Component & Connector views can also be combined with an Allocation view. Once the minimal set of viewpoints is defined, the last step is to prioritize them depending on the stakeholders' interests and extraction dependencies (one view might need information from another view before being created).

Define the source viewpoint. The source viewpoint specifies the source view that contains the basic information for creating the target views. With the help of the system experts, we can break down the architectural concepts of the target viewpoints into the source elements (that belong to the design or code viewpoints of Figure 6.1). In old legacy systems, there are often many inconsistencies and duplicates due to various architectural styles overlayed. We can ask the experts to explain how the architectural concepts are implemented, configured or deployed and we can bring together all the various elements needed to identify the concepts. Once this step is completed for all the architectural concepts, the source viewpoint contains the basic facts that we need to extract from the implementation.

Define the mapping rules. The mapping rules represent the traceability links between the target and source viewpoints. They show how the architectural concepts in the target viewpoint are mapped to the elements in the source viewpoint (hence in the implementation). Ideally, it is a formal description how to infer the target view from the source view. Realistically, the description consists of a set of heuristics and guidelines that will be used during the data gathering activity. The *map* is an instance of the mapping rules and it maps the source to the target views.

Define the Hypothetical Views. The last step is to define the hypothetical views that describe the actual stakeholders' understanding of the target views. The hypothetical views capture the as-designed or as-intended aspects of the selected target viewpoints. They can serve the purpose of guiding the reconstruction activity and of being a baseline to compare

with the target views. In the former case, the hypothetical views represent an imperfect or incomplete instantiation of the target viewpoints. The reconstructor's goal is to create the complete versions of those viewpoints. In the latter case, the hypothetical views represent the as-intended design that the stakeholders want to enforce in the implementation. Once the target views have been recovered they can be checked against the intended design for conformance and the differences can be reported to the stakeholders.

Depending on the level of maturity of the organization, the information the required information is easily available in the documentation or it must be recovered. In the case of product families, the reference architecture should document exactly this aspects and it should be the primary source of information for this activity.

Below there is a summary of the concept determination activity.

Objective: is to recover the architectural concepts and to define the viewpoints.

Input: existing documentation, interviews of the stakeholders and system experts, catalogue of reference viewpoints and reference architecture for product families.

Actors: the process designer and the stakeholders.

Techniques: workshops and scenario analysis of key system features.

Output: definition of architectural concepts; source and target viewpoints and their mappings; hypothetical views.

7.3.3 Data Gathering

Data gathering is the first activity of the view reconstruction phase. The goal is to collect the data contained in the source viewpoint. The result is the source view that represents the base knowledge about the system that will be use in the rest of the reconstruction. We can use any reliable source of information. The source code is the primary source, being the most reliable. However, there are other artifacts that we have to consider: buildfiles/makefiles, unit tests, configuration files. In general, at this point we should extract facts that we believe are trustable and available in a formalized form (e.g. in the source code, in a database, in a log file). Uncertain or inconsistent knowledge should be analyzed in next activity (e.g. unclear ownership of the modules). The final data is stored in a repository and processed in the next activity.

We identify three categories of facts that we extract from the implementation:

Programming language concepts This category contains the concepts from the design viewpoint (described in Section 6.4). They are implemented in the source files with a particular programming language. For instance in C/C++ programs, we commonly need to extract the function definitions, the classes (including methods and attributes), global variables, macros and their dependencies (variable accesses, function calls, include dependencies). This information can be formalized using one of the reference schema described in Section 6.4. For our cases, the FAMIX model provides the required abstraction level. The extraction of the programming language concepts relies on well-established techniques. Plenty of academic and commercial tools are available: SNIFF+ [1], SOURCE NAVIGATOR [2], the front-ends from the Edison Design Group [3], COLUMBUS/CAN [4] and CPPX parser [5]. Below we discuss the differences between the various extraction techniques.

Code patterns This category contains those concepts that are implemented with a particular code pattern and are not adequately identified with the tools of the previous category. For example, a remote procedure call might consist of several instructions for setting the various parameters of the call. Their extraction require ad-hoc analysis based on lexical analysis, manual inspections or island grammars.

Domain concepts Concepts that are required by the knowledge inference activity. They are not easy to map and require domain knowledge. In this group we have those concepts that are not directly detectable in the source code but they are required by the reconstruction process. They represent conceptual information that is required to increase to level of abstraction in the model. This requires to analyze the existing documentation (manually or with text analysis tools), the design models stored in CASE tools or information from databases.

Techniques for data gathering can be divided in static and dynamic analysis. Static analysis techniques examine the system's artifacts and obtain information that is valid for all possible executions.

Dynamic techniques collect information from the system's execution. This is typically done by instrumenting the system, tracing the execution path and analyzing the traces. Other techniques for extracting the data are debugging, profiling or simulation in a runtime environment. The interpretation of the traces is limited to that particular execution and it cannot be generalized to other executions. However, there are techniques for guaranteeing the coverage of the system.

[1] SNIFF+ from WindRiver: http://www.windriver.com
[2] SOURCE NAVIGATOR from Red Hat: http://sources.redhat.com
[3] Edison Design Group: http://www.edg.com
[4] COLUMBUS/CAN from FrontEndART: http://www.frontendart.com
[5] CPPX from University of Waterloo: http://www.swag.uwaterloo.ca/cppx

There are various techniques for statically analyzing the textual artifacts. We briefly review them:

Manual Inspection the reconstructor can manually inspect the source code and gather the relevant information. For example, exploring the directory structure and filling a module-directory table. If the data is not well-structured, this is often the only solution.

Lexical Analysis aims at searching the textual files for particular text patterns. The most well-know tool in the UNIX environment is GREP that searches text for strings matching a regular expression. GREP is a popular but primitive reconstruction tools for quickly finding lexical dependencies. More advanced text processing tools (like AWK , PERL , `lex`) allow the user to specify a specific action when the pattern is matched. Lexical analysis techniques are typically used for extracting the concepts of the code pattern category. Pinzger et al. have proposed a framework for structuring the patterns to extract with XML (Pinzger and Gall, 2002; Pinzger *et al.* , 2002).

Syntactic Analysis is based on parsers and they can deliver very accurate and detailed information. Parser-based tools typically build an abstract syntax tree of the input and allow the user to traverse, query or match the tree for certain patterns. Although they provide very accurate results, they often require that the source code is compilable. This technique is used for extracting the concepts in the programming language category. Tools like COLUMBUS/CAN , CPPX and the Edison Design Group are based on this technology.

Fuzzy parsing Fuzzy parsers are an hybrid between lexical and syntactical analysis (Koppler, 1997). They are able to recognize only certain parts of the programs and discard the un-relevant tokens. They are typically hand-crafted to perform a specific task e.g. program browsing. Fuzzy parsers are robust against the grammatical errors in the text files and they can also parse incomplete files. Tools like SNIFF+ and SOURCE NAVIGATOR are based on fuzzy parsing.

Island grammars are used for generating robust parsers from grammar definitions by combining the detailed specification of grammars with the liberal behavior of lexical approaches (Moonen, 2001). Island grammars combine the accuracy of syntactic parsing with the flexibility and tolerance of lexical analysis.

The output is a relational data set that can be stored in a repository. There are different options for storing the facts. The simplest format is the RSF format (RIGI [6] Standard Format) where the data are stored as triples (`verb subject object`) in plain ASCII files

[6] RIGI : http://www.rigi.csc.uvic.ca/

where `verb` specifies the type of relationship, `subject` the source entity and `object` the destination entity. Models in RSF formats can be directly visualized as graphs in several tools like RIGI and SOFTVIS (Telea *et al.* , 2002). For exchanging the model among different tools, there is an emerging standard called GXL (Graph Exchange Language) (Holt *et al.* , 2000c). The models can also be stored in a relational database.

The data in the source view can be validated with the stakeholders and with random checks against the implementation.

The summary of the data gathering activity is reported below:

Objective: is to gather the data from the implementation and create the source view.

Input: source code, builfiles/makefiles, configuration files, existing documentation and databases.

Actors: the reconstructor and system experts.

Techniques: instrumentation and tracing for dynamic analysis; manual inspection, lexical analysis, syntactic analysis and fuzzy parsing for static analysis.

Output: the populated repository containing the source view.

7.3.4 KNOWLEDGE INFERENCE

The goal of this activity is to infer the target views from the source view. The source view is typically a large relational data set that is not structured-enough for expressing the high-level architectural information of the target viewpoints. The reconstructor is responsible for creating the target views by condensing the low-level details of the source view and by abstracting them into architectural concepts. The *mapping rules* formalize (or at least describe) how the elements in the source viewpoint are mapped to the elements in the target viewpoint. The *map* is the actual mapping between a particular source view and a target view. For instance, if the mapping rules define a rule about using the naming conventions to combine the classes into modules, the map is the actual list of classes mapped to the modules. The reconstructor is responsible for creating the map and eventually for refining the mapping rules. This activity requires to interview the system experts for formalizing details that have not been considered in the design phase. New information may be required to complete the map and this may lead to several iterations of the process design phase or the data gathering activity. The final outcome are the target views.

Depending how well-formalized the mapping rules are, this activity can be fully or partly automated (hence requiring the manual input of the reconstructor). For instance, the organizational information how the components are allocated to projects may reside in a database or only informally in project plans. In the first case, the information can be easily extracted from the database during the data gathering phase; in the second case, we need to reason on the information in the documents and we need to manually create the map. We can distinguish between the information that is easily extractable in an almost fully automated way and the information that is based on domain knowledge. The first type of information is extracted during data gathering and the second type is extracted in the knowledge inference. As the organization and the reconstruction process mature, the domain knowledge becomes increasingly more formalized. In the ideal situation, all required information is formalized in the source view and knowledge inference is an operation of model transformation. In practise, the ideal case requires the domain knowledge to be explicit and this is achieved after several iterations.

The domain knowledge is the means for increasing the level of abstraction. We can extract it from the interviews with the system experts, from requirements database, feature list and reference implementation. In general we should expect that the domain knowledge is available in an informal way. If the hypothetical view has been recovered, this can also be a useful source of information. The hypothetical view is not a trustable source because it describes the the intended design of the system and there is no guarantee that it conforms to the reality. Therefore, the hypothetical view can guide the reconstructor but its facts should be carefully investigated and validated. Technological, organizational, and often historical background knowledge is also necessary during this activity. For this reason, in the ideal situation the reconstructor is one experienced developer of the system.

Two separate steps are part of this activity: (1) the definition of the map and (2) the creation of the target view. The map can be a relation in the source view, a relation obtained by composing different relations from the source view or a new relation defined during this activity. The creation of the target view can be done manually or automatically depending on the abstraction technique.

Existing techniques can be categorized as manual, automatic or semi-automatic:

Manual approaches are based on the abstraction capability of a reconstruction tool. For example, RIGI allows to group elements in clusters manually. The map is created while operating with the tool and it is not explicitly defined.

Semi-automatic approaches help the reconstructor to create architectural views in an interactive or formal way. They typically rely on the manual definition of the map. Differences among the approaches concern the expressiveness of the language used

for defining the transformations, support for calculating transitive closures of relations, degree of repeatability of the process, amount of interaction required by the user, and the types of architectural views that can be generated.

The relational algebra, defined in Chapter 5, allows the reconstructor to define a precise set of transformations for creating the target view. In Section 3.3 we have shown that relational algebra is also for the reconstruction of industrial software. Other formal approaches are based on meta logic programming like Prolog (Mens, 2000).

Light-weight approaches are the reflexion models (Murphy *et al.* , 2001), Tcl scripts for defining graph transformations in RIGI , SQL queries for defining grouping rules like in DALI , or the ad-hoc graph query language (GReQL) of GUPRO (Kullbach and Winter, 1999).

Fully automatic approaches are based on different kinds of clustering algorithms: automatic clustering (Mancoridis *et al.* , 1998), file names (Anquetil and Lethbridge, 1999) and concept analysis (van Deursen and Kuipers, 1999; Eisenbarth and Koschke, 2003).

The mapping is often difficult because of hidden dependencies. (van Deursen *et al.* , 2004) report about one interesting experience about the identification of ”logical” or ”hidden” interfaces. These were not explicitly visible in the source code and were discovered only by studying the control flow of the application and data sharing between classes that had no explicit dependencies. The quality of the knowledge inference is dependent on the data gathering activity.

The summary for this activity:

Objective: is to derive the target views from the source view.

Input: source view, domain knowledge, hypothetical view.

Actors: the reconstructor and system experts.

Techniques: manual abstraction with the help of general-purpose tools; semi-automatic approaches based on relational algebra or other formalisms; automatic clustering.

Output: the target views.

7.3.5 PRESENTATION

An effective visualization format is necessary for making the architectural views available to the stakeholders. The goal of this activity is to make the target views physically avail-

able. The target views provide the information for solving the stakeholders' problems but they have to be properly presented to be effective. Ideally, the target viewpoints have been carefully designed to address the problems; however, even in the best cases, the stakeholders need to browse the the target views, make queries and understand the implications of the architectural dependencies in the implementation. With the term *presentation*, we indicate any means of communicating information to a viewer in a textual or graphical format. There are several key requirements that should be considered when selecting the presentation format: readability, traceability, availability and interactivity. Readability concerns with the ease the users can answer particular questions about the architectural views. Traceability concerns with the ability of relating the high-level architectural concepts back to the implementation. The availability concerns with the variety of users that can be reached with a particular visualization format. Interactivity concerts with the ability of the end-user to modify the presentations. The various visualization techniques address different requirements.

In the NIMETA environment (see Chapter 8) we rely on three categories of presentation formats: *textual reports*, *hierarchical relational graphs*, *hyper-linked web documents* and *CASE (Computer Aided Software Engineering) tools*:

Textual reports are a basic presentation format where the relations of the architectural views are listed in a textual document. Due to the large amount of the data, it is necessary to structure the document accordingly to the needs of the end-user by filtering out irrelevant data. Textual reports are simple to search and they are useful a quick inspection of the data.

Graphs are a natural choice for visualizing architecture elements and their (often binary) relations as confirmed by two independent surveys (Bassil and Keller, 2001) and (Koschke, 2003). We are especially interested in the hierarchical graphs because they can render the containment relationship of the architectural views in a very natural and intuitive way. The graph-based visualization typically allow the user to interact like querying, zooming, navigation, selection, hiding and calculating the transitive closure. In Chapter 8, we have presented three graph based visualization tools based on hierarchical graphs that we typically use in our reconstruction projects: RIGI , SOFTVIS , DOT and Rational Rose for UML graphs. They provide the ability to navigate the whole architectural views from the top elements to the details. They do not limit the end-user to a particular view on the data but they allow him/her to build his/her own views from the data set. One drawback is that graph-based visualization often lack usability and they require a certain amount of training to become an efficient user. However, their high flexibility makes them the favorite choice for reconstructors and advanced users. Graph-based visualization are often used during the discovery process when the user needs to investigate the recovered information.

Hyper-linked web documents are an efficient way for publishing the architectural views to a wide range of users. This visualization format favors the usability and accessability of the architectural views while limiting the flexibility of the visualizations. The textual or graphical visualizations are typically pre-defined and the end-user has a limited influence on the presentation (e.g. selecting parameters or zoom in/out). The user can select the content of the documents but the visualization format is fixed. An architecture-driven development process requires the architecture documentation to be available to a wide range of stakeholders (architects, programmers, designers, testers, managers). Web documents are an efficient media for distributing the architectural knowledge and can maximize the benefits of the reconstruction work. The documents can be published in the company's intranet and they do not require special training. Simplicity and ease of use are the key requirements for the design of the web documents, following the traditional web design guidelines (Krug, 2000). Web documents also offer the advantage of combining text information (like tables and detailed text) with graphical diagrams. Hyperlinks allow the user to browse the architectural views by following the dependencies paths or other model dependencies.

CASE tools are often used by large organizations for creating and maintaining the design documentation and other artifacts. CASE tools provide a robust environment for designing the software, generating code fragments, checking the consistency and so on. UML (Unified Modeling Language) (Booch *et al.* , 1999) has become the defacto standard modeling language for most of the commercial tools. Although its semantic is not considered precise and its use for architecture modeling is not commonly understood, often it is necessary to present the recovered views in the visualization format of the CASE tool that is used for forward engineering. The goal is to present the models in a format that is already familiar and well-understood by the development team. This often requires to convert the architectural views to the UML format or to the formalism used by the CASE tool.

Objective: is to present the target views in a textual or graphical format.

Input: target views.

Actors: the reconstructor and stakeholders.

Techniques: textual reports, hierarchical relational graphs, hyperlinked web documents, UML format and CASE tools.

Output: the target views rendered with a particular format.

7.3.6 CONFORMANCE CHECKING

The goal of this activity is to check the conformance of the recovered architectural views against the architectural rules. Enforcing the architectural rules is a method for guaranteed the integrity of the architecture during its evolution. The enforcement can be loose or strong. In the case of loose enforcement, the output of the conformance checking is a list of violations that is reported to the stakeholders who are responsible to take the proper actions. In the case of a strong enforcement, changes in the architecture are rejected if the architectural rules are not satisfied. Similarly to compiling where errors make the compiler to fail, a strong conformance enforcement rejects changes that violates the architectural integrity.

There are different architectural rules depending on the views. Certain views can be recovered especially for checking the conformance of particular rules:

- We can check the conformance of the target view against the hypothetical view. The hypothetical view represents the intended design that the architects or designers want to achieve in the implementation. In this way, we can check that the design has been followed correctly during the implementation. Violations can lead to rethink the implementation or, sometimes, the design. We can check either structural (for example, the decomposition view against the intended decomposition) or behavioral aspects (for example, a particular trace against a use case or against its specification).
- We can also check the conformance of the implementation against the architectural style of the system. The architectural style defines the interaction patterns and communication protocols that the various elements of the system must follow. We can enforce that the style is respected in the implementation. For instance, In a layered architecture we may want to forbid layer bridging (a layer can only use the layer below). In the case of product family, we need to check the conformance of the implementation against the reference architecture that has to be valid for all the products of the family.

The automation of the reconstruction process allows us to regularly check the architectural conformance at every build or major release of the system. The check can be conducted manually by the stakeholders or it can be completely automated by specifying the architectural rules with the binary relational algebra (or other formalisms like UML as presented by (Selonen and Xu, 2003)).

This activity is summarized below:

Objective: is to check the conformance of the target views against the architectural rules.

Input: target views.

Actors: the reconstructor and stakeholders.

Techniques: manual conformance check, automatic checking with the rules specified in binary relational algebra or other formalisms (like UML).

Output: list of violations.

7.3.7 ARCHITECTURE ASSESSMENT

Architecture assessment methods (like SAAM (Kazman *et al.* , 1994) and ATAM (Kazman *et al.* , 1998)) can benefit of a reconstructed architectural model. The goal of the assessment is to check that certain quality aspects are satisfied in the software architecture. The reconstructed views provide up-to-date information for the assessment, for checking the quality aspects and for calculating the architecture quality metrics.

We summarize this activity below:

Objective: is to check that certain quality attributed are satisfied in the software architecture.

Input: target views.

Actors: the stakeholders.

Techniques: techniques for architecture assessment like SAAM and ATAM.

Output: assessment report.

7.3.8 RE-DOCUMENTATION

The re-documentation activity requires to update the existing documentation of the system with the reconstructed views. Depending on the maturity of the organization, we can envision three typical situations that are described below:

- The architects manually update the existing documentation with the new information extracted from the reconstructed views. The views are examined, compared with

the existing ones and the documentation is up to date. It is the responsibility of the architects to decide how the information is presented in the final architectural documentation.

- The architects integrate the existing documentation with the reconstructed views. The views are already in an acceptable format that is familiar to all the stakeholders. The views are integrated by simply copying and pasting the diagrams in the architecture documentation.
- The reconstructed views are the official documentation of the architecture of the system. There is no other design documentation for the structural aspects of the architecture except the views that are recovered with the reconstruction process. This is an ideal case where the forward and reverse engineering activities are harmonized in the development process and are based on the same meta-data and format. Since the reconstructed views are in the same format of the design views, the designers can carry out their design activities on views that reflect very closely the implementation.

We summarize the re-documentation activity below:

Objective: is to update the existing architecture documentation.

Input: target views.

Actors: the architects and the stakeholders.

Techniques: manual update, integration and official documentation.

Output: up to date architecture documentation.

7.3.9 ITERATIONS

To produce an adequate architectural model, reconstruction process has to be re-iterated several times. The initial abstraction rules are based on the conceptual model of the developers and it can take several iterations to align it with the concrete model. New architectural concepts become significant while the reconstruction is progressing and have to be introduced in the model. The data-gathering phase can also be refined by increasing the quality of the extracted information with more powerful analyzers (often the extraction is a trade-off between the speed/size and the quality of the analysis). Moreover, as the reconstruction process matures the domain knowledge becomes formalized and the whole reconstruction pipeline can be automatized.

7.4 MATURITY LEVELS OF ARCHITECTURE RECONSTRUCTION

We define different levels of maturity for architecture reconstruction. (Krikhaar, 1999) also defined five levels of maturity for the SAR reconstruction process: initial (the architecture not known and the system is hardly documented), described (software architecture is explicit), redefined (the ideal and concrete architectures are similar), managed (the quality of the architecture is sustainable) and optimized (the architecture can be extended). We agree with most of Krikhaar's levels as they can be applied to our reconstruction process as well. Anyway, we revisit them and extend Krikhaar's list:

0. No architecture There is no control of the software architecture. The architecture is either unknown or only in the minds of the of the programmers. The development relies heavily on humans' interaction. This is the typical situation of software systems that are immature or un-maintained or quickly developed (as it could be the case of a start-up company that follow an extreme programming practice). A reconstruction process cannot be started till when the architects have been nominated, that is in the next level.

1. Responsibilities The organization is aware of the importance of a software architecture. Architects are nominated and their main job is to preserve the integrity of the system. A documented architecture does not exist yet but the architects understand the importance of having one. Although the architects are likely to be selected among the most experiences developers, with time they increase their power against the management. Empowered architects are able to acept/reject architectural decisions and features that might break the integrity of the system. In this level a reconstruction activity can start and the customers are the architects.

2. Described There is a process in place for recovering an architecture description from the implementation. The description is reliable and used in the team for communication. However, the architecture of the system does not conform to the intended architecture of the architects.

3. Refined Based on the described architecture, the architects took the necessary actions to align the implementation with their intended design. At this point, the reconstructed architecture conforms with the desires of the architects. There is no need for the architects to keep a separate design documents, as the architecture can be automatically recovered.

4. Managed An automatic mechanism is in place for checking the conformance of the implementation against the architectural rules. The architects can enforce that only a

build that passes all the rules can be released. In this way, the architects can guarantee that all the releases have a desired quality level.

5. **Optimized** At this level, the architecture can be extended while preserving certain quality levels. The architecture is able to self-maintain its integrity while it is extended to support new features.

6. **Self-organizing** We envision that in the future an expert system will assist the architects to cope with the complexity of the design. This is described in the future work.

7.5 EXAMPLE

In this section, we demonstrate the reconstruction process on a small Java application called Venice. Venice is a graphical applications for visualizing software architectures in UML. It was developed by four students from the University of Helsinki in a period of 3 months (approximately 7 person months) for a software engineering project. The Venice project strictly followed the typical waterfall process (requirements definition, architecture design, implementation and testing). The total source code consists of approximately 28 KLOC. One third-party graphic library has been used.

PROBLEM DEFINITION

After collecting the requirements, the students proposed an initial architecture for Venice. The diagram in Figure 7.3 is taken from the architecture description document and shows the *intended* design of Venice. Venice consists of four main subsystems:

Application that contains the implementation of the main program as a Java application and as a Java applet.

ModelStorage that is responsible for reading/writing the graphic files in the GXL format.

Commands that contains the implementation of the various graphical commands for zooming, panning, filtering, collapsing nodes and so on.

GUI that contains the graphic visualization engine and the user interface.

The diagram also show the two external libraries: Jazz and JFrame.

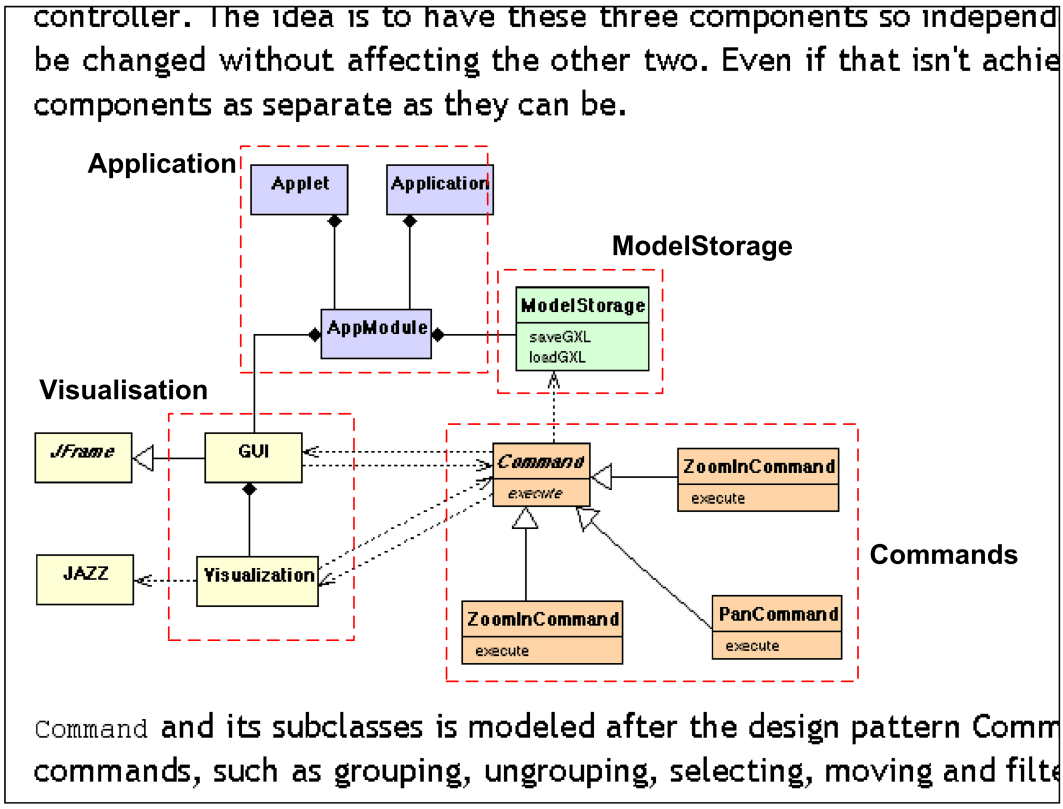
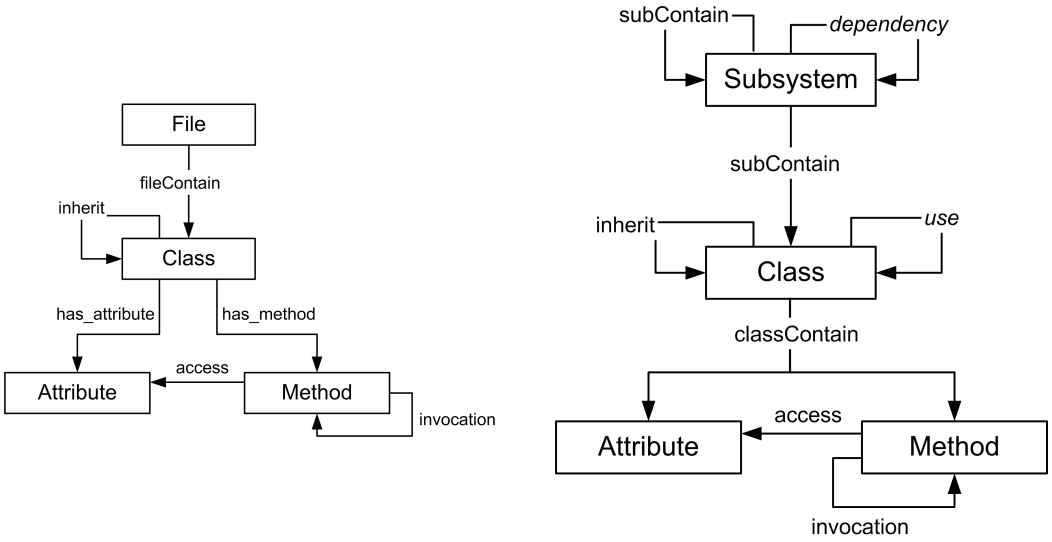


Figure 7.3: Intended architecture of Venice.

The goal of the reconstruction is to validate the intended architecture design.

CONCEPT DETERMINATION

The target viewpoint is the decomposition viewpoint that describes the partitioning in subsystems. Subsystems are functional clusters of Java classes. Methods and attributes are logically grouped within the classes where they are defined. We are interested in analyzing the dependencies among the subsystems. The source view is based on the FAMIX design viewpoint. The diagrams in Figure 7.4 represent the source and target viewpoints.



The source viewpoint. The target viewpoint.

Figure 7.4: The viewpoints for reconstruction of Venice.

We define the following mapping rules:

- The relation *classContain* groups attributes and methods in their classes. It is based on the *has_method* and *has_attribute*.
- The relation *subContain* is a containment relaiton that groups classes in subsystems and subsystems in top-level subsystems. It is defined by interviewing the developers.
- The relations *use* and *dependency* are calculated by lifting the class-level relationships (*inheritance*, *access* and *invocation*).

DATA GATHERING

Since the source viewpoint is based on the FAMIX model, we extract the source view with SOURCENAVIGATOR and the SNAV2NIMETA script (as described in Chapter 8). The source view contains 80 classes, 520 methods, 346 attributes, 76 files, 55 inheritances, 2,000 accesses and 1,194 invocations.

KNOWLEDGE INFERENCE

We defined the relation *subContain* by interviewing the developers. We asked the students to cluster the files they were responsible for in subsystems according to their functionality. The subsystem have been ulteriorly clustered in the top-level subsystems.

We calculate the relation *usage* that represents the low-level dependencies

$$usage = access + invocation$$

We lift the relation *usage* with the relation *classContain*:

$$\begin{aligned} classContain &= has\_method + has\_attribute \\ use &= usage \uparrow classContain \end{aligned}$$

The relation *use* contains the class-level dependencies induced by the accesses and the method invocations. The inheritance dependency is already at the class-level. We can now lift the class-level dependencies to the subsystem level:

$$dependency = (use + inherit) \uparrow subContain$$

The relation *dependency* contains the top-level dependencies among the subsystems induced by accesses, method invocations and inheritances.

PRESENTATION

We present the target view with graphs in RIGI and DOT . The Figure 7.5 shows the Venice's class diagram that has been calculated with the script discussed in Section 8.4.1.

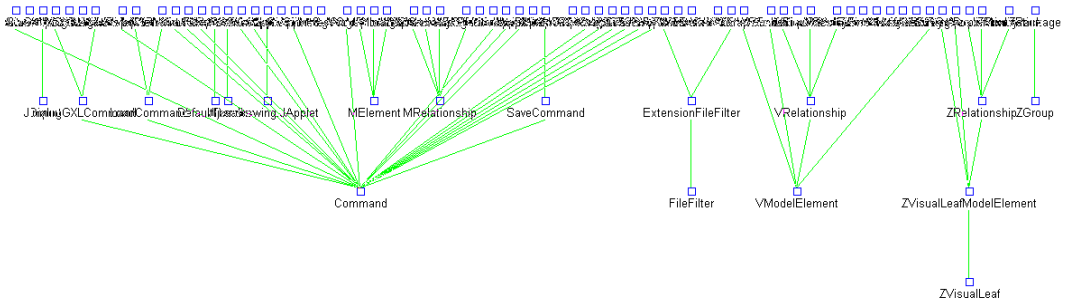


Figure 7.5: The class diagram of Venice.

The Figure 7.6 shows the classes of Venice and the dependencies caused by variable accesses, method invocations and inheritances. The dependencies are shown at the class-level.

The two graphs in Figure 7.7 show the top-level diagram of Venice with and without the subsystem `LIBRARY`. We hide the subsystem `LIBRARY` becomes we are mainly interested in the internal dependencies of Venice rather than with the external environment.

CONFORMANCE CHECKING

We check the conformance of the implementation against its design. Based on the documentation shown in Figure 7.3, we can define the following relation *permit* that contains the permitted dependencies among the subsystems:

$$
\begin{aligned}
permit \quad = \{ \quad & (\text{'Commands'},\text{'ModelStorage'}),\\
& (\text{'Commands'},\text{'GUI'}),\\
& (\text{'Commands'},\text{'Visualisation'}),\\
& (\text{'Application'},\text{'GUI'}),\\
& (\text{'Application'},\text{'ModelStorage'}),\\
& (\text{'GUI'},\text{'Commands'}),\\
& (\text{'GUI'},\text{'Visualisation'})\}
\end{aligned}
$$

The we calculate the top-level dependencies and we remove the dependencies with the

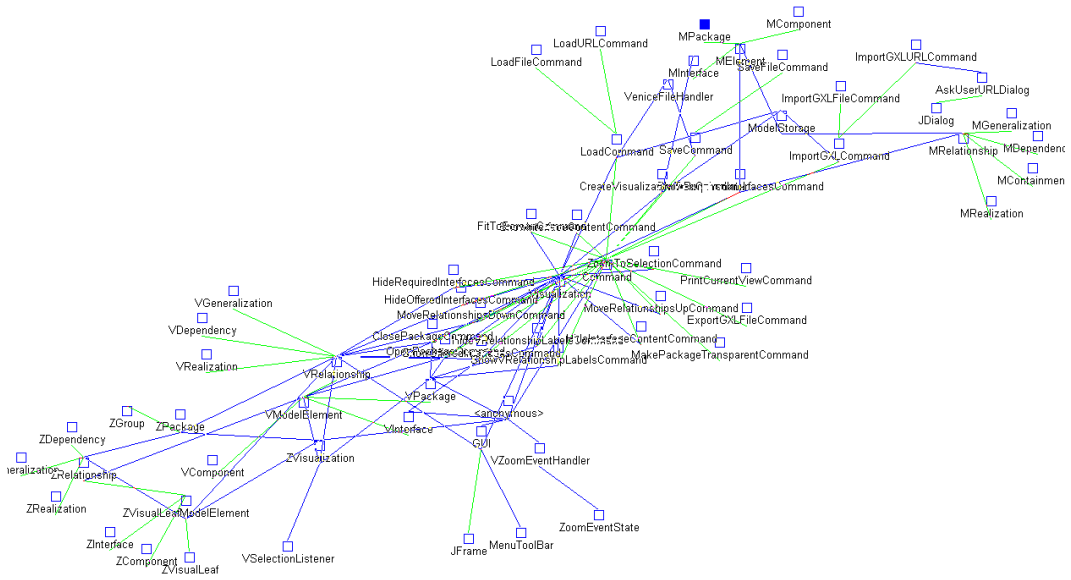
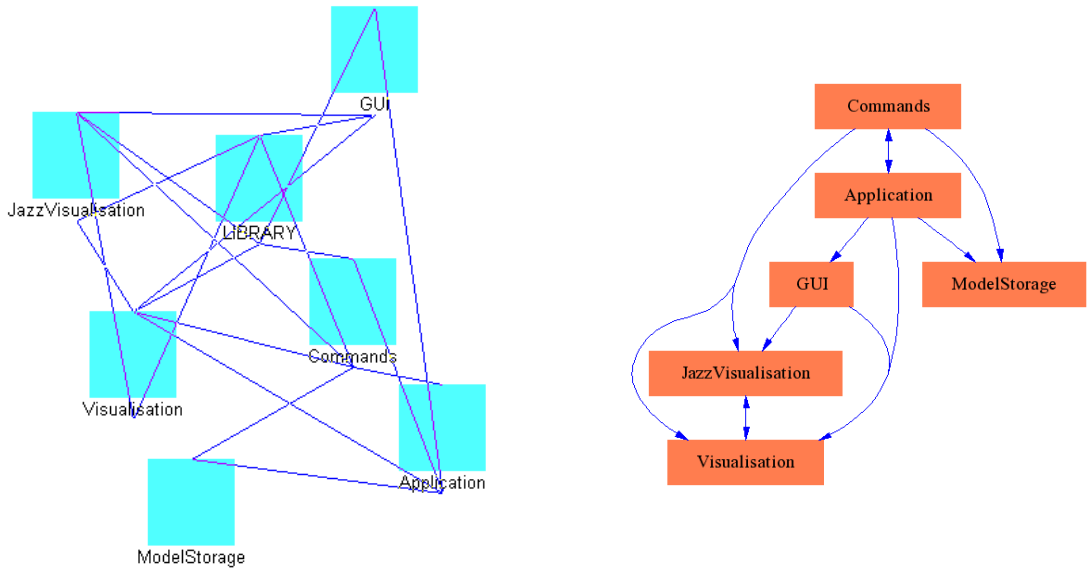


Figure 7.6: Venice's class diagram with the relation *use*.



The top-level diagram with the `LIBRARY`. The top-level diagram without the `LIBRARY`

Figure 7.7: The top-level diagram of Venice.

LIBRARY.

$$\begin{aligned} topDeps &= depdendency|_{car} \top subContain \\ topDepsNoLib &= topDeps \setminus_{car} \{\text{'LIBRARY'}\} \end{aligned}$$

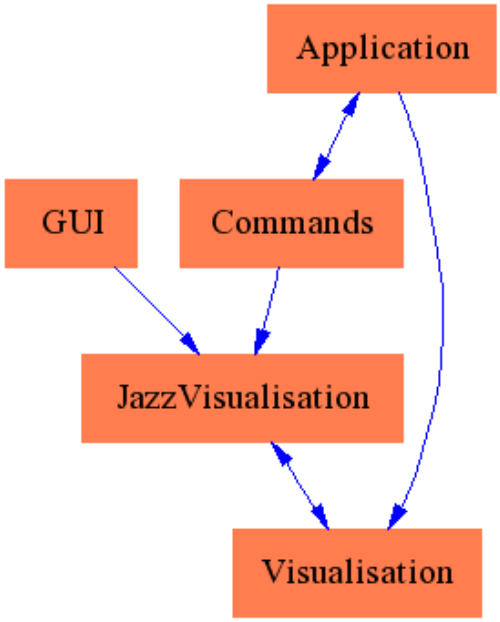
We calculate the violations with the different between the top-level dependencies and the permitted dependencies:

$$violations = topDepsNoLib - permit$$

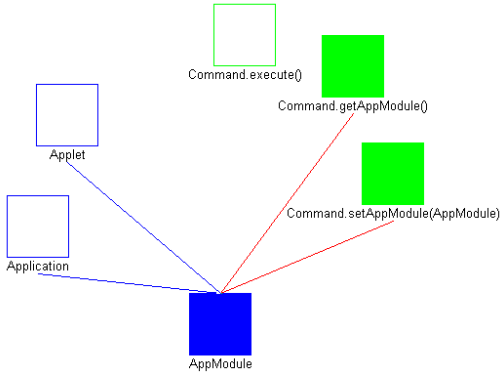
The result is shown in the graph Figure 7.8 that shows the forbidden dependencies. We can make the following considerations:

- The subsystem `JazzVisualisation` is not present in the design documentation. It represents a collection of interfaces for using the external Jazz library. The students confirmed that it should be added to the documentation.
- The subsystem `Application` depends on `Visualisation` and `Commands`. The students knew about this dependency that represents a workaround for a bug in the web browser. The dependency is caused by variable accesses.
- The subsystem `Commands` depends on `Application`. This is an unplanned dependency due to an architectural mismatch of the original design with the Java sandbox model. The students considered a bad design but they could not find a better solution than this one. With the help of RIGI we have navigated the graph and we have discovered the reason for this dependency. The graph in Figure 7.8 shows the violations: two methods from `Commands` are accessing the variables of the class `AppModule` contained in `Application`.

Even in this is a simple example based only on 28KLOC of Java code, we have found several violations that were not documented in the original design. Although the students were aware of them, they thought they could solve them before the end of the project. Since this did not happen, after the end of the project the documentation was left in an inconsistent form compared to the implementation.



Venice's forbidden dependencies.



The access violations from the methods of `Commands` to the variables of the class `AppModule`.

Figure 7.8: The violations of Venice.

CHAPTER 8

THE NIMETA TOOL ENVIRONMENT

Basic research is what I'm doing when I don't know what I'm doing.

- John von Neuman

This chapter presents the NIMETA tool environment that supports our reconstruction process. The tool environment is designed as a pipeline of tools for extraction, abstraction and presentation. We present the tools that are part of the pipeline.

8.1 INTRODUCTION

Throughout this thesis, we assume that the organization is adopting an architecture-centric development process where the maintenance of a robust and well-documented software architecture plays an important role. Architecture reconstruction is the means for maintaining the software architecture well-documented and to support and efficient evolution of the system. However, practitioners are still facing big challenging in applying the architecture-centric development process. One important reason is that there is no universally accepted architecture description/modelling language (ADL) to support the design activity. Usually ADLs are either too specific to cover the whole architecture or too generic to model the details of complex software systems. ADLs may be used for specific purposes but they are not able to capture the design of the whole system. On the other hand, the practitioners lack a proper architecting tool. The architecture description is scattered around with informal notations and heterogenous tools (from spreadsheets to graphical drawing tool). In this situation it is very difficult to analyze, modify or maintain the architecture descriptions.

In applying the NIMETA process we realized that the existing commercial modeling tools are incapable of reverse engineering the architectural information from the implementation

in an efficient way. The lack of a proper reverse engineering tool is a critical deficiency for our approach since we are dealing with very large systems containing millions LOC. The recovery of the reconstruction results in models with hundreds of thousands relationships. This information has to be handled efficiently and in a short time. Our customers are involved in highly dynamic business where the time-to-market is a critical factor. Based on these considerations, we concluded to develop our own reconstruction tool environment, called NIMETA , that is practical, effective, simple to operate, scalable and adaptable to the existing tool infrastructure. We summarize the main requirements below:

- Clear separation of the three activities: extraction, abstraction, presentation.
- Support for multiple languages, different architectural styles, relational data extracted from various sources.
- configurable according to the architectural concepts and views.
- Extensible with new plug-ins and external tools to support the three main activities.
- Simple and human readable format for the representation of the data.
- Scalable to multi-million LOC systems.
- support for the generation of UML diagrams although UML is not the internal formalism of the environment.

After experimenting the existing reverse engineering tools, we decided base NIMETA on existing commercial and academic tools. Our intention is to create a rich environment with a basic set of tools and the possibility to plug-in new ones with minor effort. We also need the possibility to connect the various tools in a pipeline in order to automated the reconstruction process.

8.2 OVERVIEW OF NIMETA

In NIMETA , we represent the data in relational format and we store them in the RIGI Standard Format (RSF). RSF is a rather simple format for storing relational data in plain ASCII files. The format consists of triples in the format: $< verbsubjectobject >$ where $verb$ is the relation, $subject$ is the source entity and $object$ is the destination entity. For example, the relation $call = \{(a, b), (c, d)\}$ can be expressed by:

```
call a b
call c d
```

It is possible to specify the type of the entities with the built-in relation *type*. RSF is suitable for storing relational graphs as it is main use in RIGI . From RIGI we have also borrowed the concept of domain. Every RSF file is associated with a particular domain that defines the relations and entities that are allowed in the RSF file. The choice of RSF is based on the fact that the format is easy to generate, read, search and concatenate. NIMETA supports also the GXL format (Graph eXchange Language) mainly for exchanging the graph information with other reverse engineering tools.

In NIMETA , we make the assumptions that all the names of the entities are unique. This simplifies the process and avoids the need of unique identifies. This assumption is not an obstacle when clear naming conventions have been defined. For the object-oriented entities, we follow the naming convention of the signatures as it is specified by FAMIX (Demeyer *et al.* , 2001). We show few examples below:

Entity type	**signature**
Class	CHelloWorld
Method	CHelloWorld.paint(int, char*)
Attribute	CHelloWorld.text
Function	foo(int, CHelloWorld*)
Global Variable	foo

The various tools are operated in a pipeline where the intermediate files are typically in RSF. The diagram in Figure 8.1 shows the typical pipeline consisting of tools for extraction, abstraction, view selection, presentation/conformance checking. The input data is the source code and other architectural artifacts (like spreadsheet, mapping tables, other models). The source view is union of the relational datasets generated from the extraction tools and is the input for the abstraction tool. The RE model is an intermediate model that typically contains all the precalculated relations that are necessary for creating the target views. The view selection is typically a script that selects a subset of the relations in the RE model and builds the target view. At the end of the pipeline, the view is presented with a presentation tool. In NIMETA , the tools are typically operated through Python scripts, allowing us to completely automated the reconstruction process.

The Figure 8.2 shows the set of tools that are part of the NIMETA environment. NIMETA contains a collection of scripts for extracting the data with Source Navigator, creating the relational algebra transformations and for converting the models to various formats. The goal of NIMETA is to allow the reconstructor to select the best tool for a certain task and

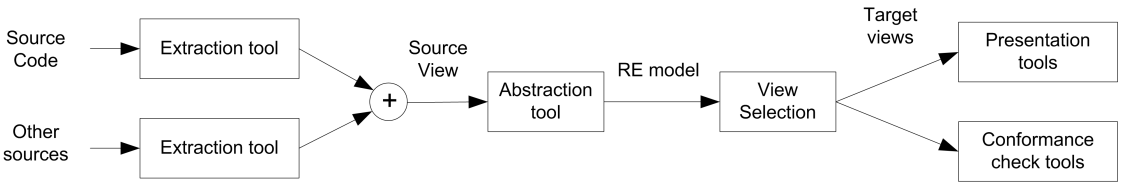


Figure 8.1: Overview of the NIMETA pipeline.

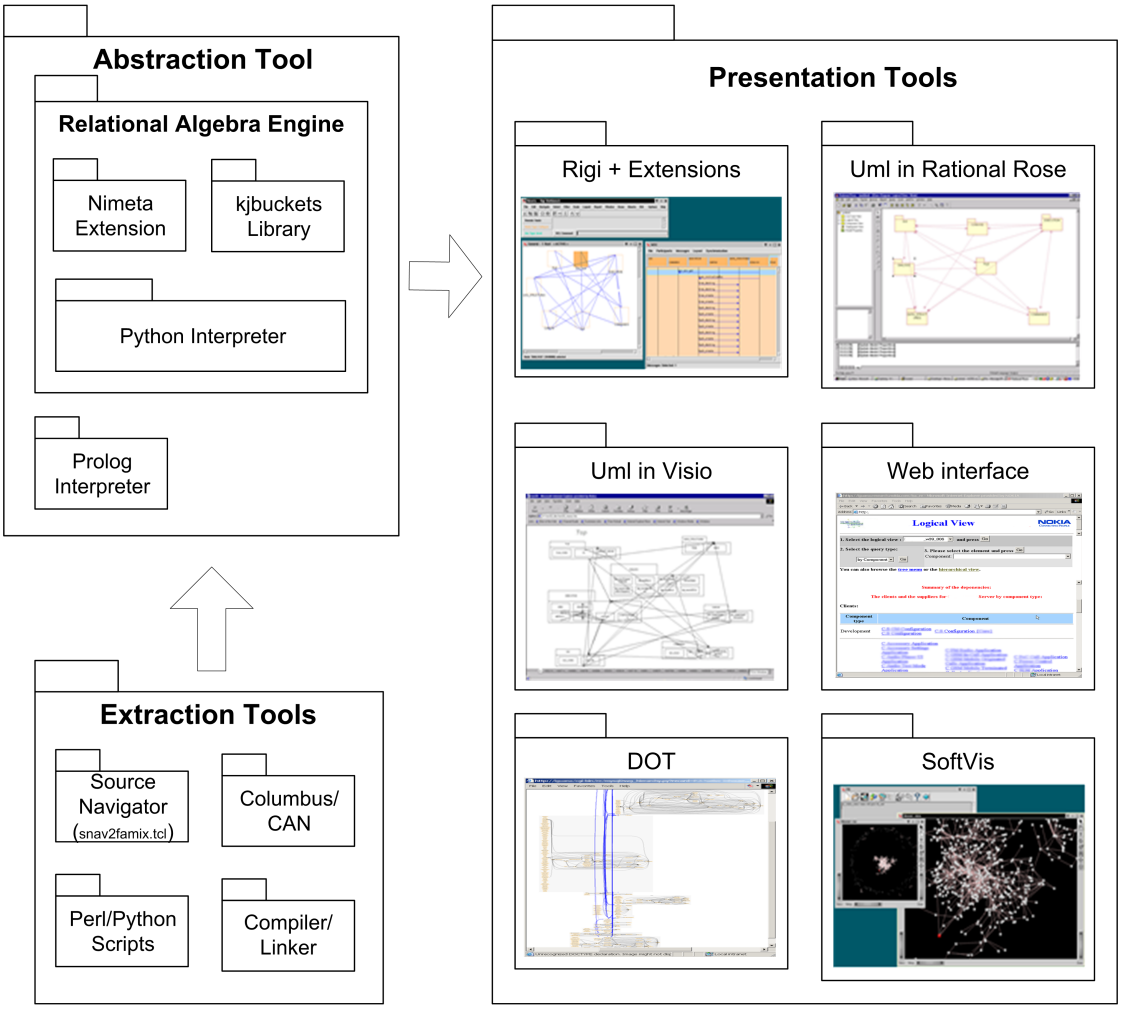


Figure 8.2: Overview of the NIMETA tool environment.

eventually to integrate new ones in a simple way. NIMETA relies on several formats to store the data: RSF , GXL and the MySQL[1] database. The visualization formats are: hierarchical graphs (in Rigi and in a proprietary visualization tool (Telea *et al.* , 2002)), UML diagrams (in Visio[2] and Rational Rose [3]) and web pages.

The user interface of NIMETA is based on an extension of RIGI and it has been programmed in Tcl/Tk. The user interface allows to execute the various scripts, visualize intermediate results and convert the RSF files to other formats. The diagram in Figure 8.2 shows the various tools that are part of the NIMETA environment. We discuss them in details in the following sections.

8.3 EXTRACTION TOOLS

The extraction of the data is supported by four different techniques that are selected according to the type of information source. All the extraction tools produce one or more RSF files that can eventually be concatenated.

SOURCE NAVIGATOR [4] is an IDE for a large variety of programming language. It offers an API for accessing the symbol table generated by a fuzzy parser. We have written a TCL script, called `snav2nimeta.tcl`, for extracting the symbolic information and storing in RSF or GXL. The output conforms to the FAMIX specification. The source code is listed in the Appendix A.1.1. Although based on a fuzzy parser, the tool can deliver reasonable information about the static dependencies, at least for an initial analysis. The parser ignores multiple compilation paths, dynamic bindings and configuration flags that are often important for C/C++ programs.

COLUMBUS/CAN is a robust parser front end and it can provide very detailed information about the static dependencies. Acting as a compiler and linker, it can overcome the limitations of SOURCE NAVIGATOR . The tool can generate the RSF and GXL format.

For the extraction of specific architectural concepts, we mainly rely on regular expressions implemented in Perl or Python scripts. The script `extract.py` (listed in Appendix A.1.2) show an example of an extraction script that was used on the case study 1. The original script contained about 50 regular expressions. The script `strip-comments-file.pl` is a Perl script used for removing the comments from the source files.

[1] MySQL: http://www.mysql.org
[2] Microsoft Visio: http://www.microsoft.com
[3] Rational Rose: http://www.rational.com
[4] SOURCE NAVIGATOR from `Red Hat`: http://sources.redhat.com

The compiler and linker often produce information about the static dependencies. We have often used the output of the compilers for embedded software (i.e. ARM). This is one alternative available in the NIMETA environment.

8.4 ABSTRACTION TOOLS

NIMETA contains its own relational algebra engine and at the moment it is the most efficient supported technique for the abstraction operations. Over the years we have experimented other techniques (like Prolog and SQL) but they all performed poorly on the transitive closure. GROK [5] appeared to be the best performing implementation (Holt, 1998). NIMETA 's engine, although less-performing than GROK , is accessible from the Python interpreter and we believe this represents an advantage in terms of extensibility and ease of use.

8.4.1 RELATIONAL ALGEBRA ENGINE

NIMETA 's relational algebra engine is based on two components: the kjbuckets library and the NIMETA extension. The kjbuckets[6] library is a freely available library which defines graph and set datatypes for the the Python programming language: `kjSet` and `kjGraph`. The library provides an optimized implementation of various relational algebra operations (like the transitive closure). More relational operators have been supported with the NIMETA extension and the possibility of reading/writing RSF file. Table 8.1 and the Table 8.2 shows the Python syntax for the set and relational operands respectively. The kjbuckets objects also support the method `.items()` for converting the sets/graphs to a Python list. The code listed in Appendix A.2.1 shows the implementation of the NIMETA extension based on the kjbuckets module.

[5] GROK is part of the SWAG Toolkit: http://www.swag.uwaterloo.ca/swagkit
[6] kjbuckets library: http://gadfly.sourceforge.net/kjbuckets.html

Operand	Name	Python syntax
$S = a, b, c$	assignment	`S=kjSet(['a','b','c'])`
$a \in S$	member	`S.member('a')`
$\|S\|$	cardinality	`len(S)`
X=Y	equal	`X==Y`
$X \subseteq Y$	subset	`X.subset(Y)`
$X \supseteq Y$	superset	`Y.subset(X)`
$X + Y, \cup$	union	`X + Y`
$X - Y, \cap$	intersection	`X-Y`
$X \times Y$	cartesian product	`cproduct(X,Y)`

Table 8.1: The set operands in Python.

The NIMETA relational algebra engine allows us to clearly define the transformations for creating the target views from the source view. We demonstrate the scripts that implement the transformation for creating the examples shown in Section 7.5. The following script implements the operations for creating the class diagram of Venice shown in Figure 7.5. After selecting all the `Class` elements and the relation *inherit*, the script generates and RSF file that can be directly loaded in RIGI for visualization.

```
from nimeta import *

in = readGraphRSF('venice.rsf')
out = {}

out['type'] = ranRest(in['type'], ['Class'])
out['inherit']= in['inherit']

writeGraphRSF(out)
```

Operand	Name	Python syntax
$R = (a,b),(c,d)$	assignment	`R=kjGraph([('a','b'),('c','d')])`
$(a,b) \in R$	member	`R.member('a','b')`
$\|S\|$	cardinality	`len(S)`
$X + Y, \cup$	union	`X + Y`
$X - Y, \cap$	intersection	`X-Y`
$X \circ Y$	relational composition	`X * Y`
$R \uparrow C$	lifting	`lift(R,C)`
$R \upharpoonleft C$	left lifting	`lLift(R,C)`
$R \upharpoonright C$	right lifting	`rLift(R,C)`
$R \Uparrow C$	full lifting	`fullLift(R,C)`
$R \downarrow C$	lowering	`lowering(R,C)`
Id_S	identity relation	`ident(S)`
$dom(R)$	domain	`dom(X)`
$ran(R)$	range	`ran(X)`
$car(R)$	carrier	`car(X)`
$R\|_{dom}S$	domain restriction	`domRest(R,S)`
$R\|_{ran}S$	range restriction	`ranRest(R,S)`
$R\|_{car}S$	carrier restriction	`carRest(R,S)`
$R \setminus_{dom} S$	domain exclusion	`domExcl(R,S)`
$R \setminus_{ran} S$	range exclusion	`ranExcl(R,S)`
$R \setminus_{car} S$	carrier exclusion	`carExcl(R,S)`
$\top(C)$	top	`top(C`
$\bot(C)$	bottom	`bottom(C`
$R.S$	left projection	`lProj(R,S)`
$S.R$	right projection	`rProj(R,S)`
$R.y$	left image	`lImage(R,y)`
$x.R$	right image	`rImage(R,x)`
R^+	transitive closure	`R.closure()`
R^*	reflexive transitive closure	`rtclosure(R)`
R^-	transitive reduction	`treduction(R)`
R^{-1}	inverse	`~R`
R^n	power	`power(R,n)`

Table 8.2: The relational operands in Python.

The transformation for creating the class-level dependencies of Venice (shown in Figure 7.6) is implemented by the following script:

```
from nimeta import *

#Read in the graph
in = readGraphRSF('venice.rsf')

out = {}

#Calculate the class-level dependencies
contain = in['has_method'] + in['has_attribute']
usage = in['invocation'] + in['access']
highUsage = fullLift(usage, contain)

#Output relations
out['type'] = ranRest(in['type'], ['Class','Method','Attribute'])
out['level'] = contain
out['composite'] = highUsage
out['inherit'] = in['inherit']
out['invocation'] = in['invocation']
out['invocation'] = in['access']

writeGraphRSF(out)
```

8.4.2 PROLOG

Although the relational algebra engine is our favorite abstraction tool, we provide also a Prolog interpreter for specifying the abstraction transformations. NIMETA includes the SWI PROLOG [7] interpreter and it is integrated in the NIMETA user interface. The user can easily define a set of Prolog propositions for manipulating a relational graph. Below we provide an explicative example.

We can transform the relational data from the RSF formta into Prolog facts. Each relation in the form '`rel src dst`' is converted to the Prolog fact '`rel(src,dst).`'. For example, the following list of facts represents the the source view for one system:

```
type('HelloWorld', 'Class').
type('HelloWorld.paint()','Method').
```

[7] SWI PROLOG : http://www.swi-prolog.org

```
has_method('HelloWorld', 'HelloWorld.paint()').
invocation('HelloWorld.paint()', 'Graphics.paint()').
```

We can use the Prolog rules to infer new relations. For example, we can implement the transformation for creating the class-level dependencies of Venice (shown in Figure 7.6) in Prolog. We define the following containment rule:

```
contain(X,Y) :- has_method(X,Y).
```

The, we define the usage relation:

```
usage(X,Y) :- invocation(X,Y).
```

We can define the transitive closure in the following way:

```
tclos(Rel, X, Y) :- P=..[Rel,X,Y], call(P).
tclos(Rel, X, Y) :- P=..[Rel,X,Link], call(P), tclos(Rel,Link,Y).
```

We calculate the class-level dependencies by computing the transitive closure of the relation *contain* and lifting the relation *usage*:

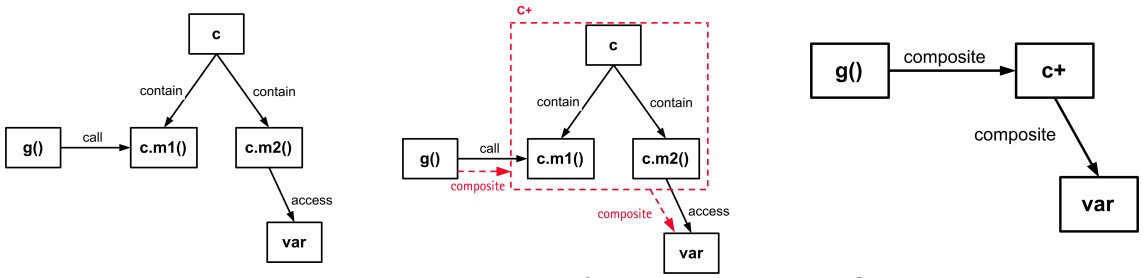
```
highUsage(X,Y) :- tclos(grouping,X,T1),tclos(grouping,Y,T2),usage(T1,T2).
```

8.5 PRESENTATION TOOLS

NIMETA supports various formats for presenting the target views. The purpose has been to integrate the textual and graphical interfaces that provides the most effective communication formats.

8.5.1 RIGI

The user interface of NIMETA is based on RIGI and several extensions. Although RIGI is a proper reverse engineering environment that includes its own C extractor, we can consider it a general-purpose graph visualization tool for visualizing hierarchical directed typed graphs. Each graph is drawn according to a particular *domain* that defines the permitted types (and colors) of nodes and edges. The graphs are drawn in a way that the edges are directed from the bottom of the source node to the top of the destination node. One of the key features of RIGI is the possibility to group (to *collapse* in RIGI 's terminology) nodes



The original graph The selection of the grouping The final graph

Figure 8.3: Example of grouping with RIGI .

in hierarchies in order to perform abstractions on the basic graph. RIGI takes care of calculating the *composite* arcs that are induced by the arcs among the grouped nodes. The reverse operation of *expanding* is also supported. The possibility of making abstractions is an important feature for architecture reconstruction when the user typically starts with a large unstructured data sets that needs to be organized. The three graphs in Figure 8.3 demonstrates RIGI 's grouping feature. RIGI also offers various features for navigating the graphs, selecting and filtering the elements according to various criteria. For these reasons, we decided to rely on RIGI as the main user interface for NIMETA . In this way, the reconstruct can visualize the intermediate results of the reconstruction as RIGI 's graphs. RIGI is extensible with the TCL scripting language that has been used to create the NIMETA 's specific extensions. The NIMETA extensions assists the end-user to access the various external tools for extraction, abstraction and presentation. We have also defined separate plug-ins for supporting specific reconstruction functions for certain systems (like the case study 1). The Figure 8.4 shows an overview of the NIMETA 's user interfaces based on RIGI and HAVA .

We summarize the main points about the integration of RIGI in NIMETA :

- The domain of the graphs is defined according to the system under analysis and to the particular viewpoints that have been defined for the reconstruction.
- As input format, we use the partly structured RSF[8] format that is a variant of RSF used for defining the type of the nodes and the hierarchical structure of the graph. The partly structured RSF contains the built-in relations *type* (the type relation), *level* (the containment relation) and the built-in node `Root` (the top node of the hierarchy). The partly structured RSF that is generated by NIMETA 's scripts precalculate also the composite arcs and can be directly loaded in RIGI .

[8]RSF File format: http://www.rigi.csc.uvic.ca/list-archives/rigi-developer-archive/2000-02-10-15.26.16-19165

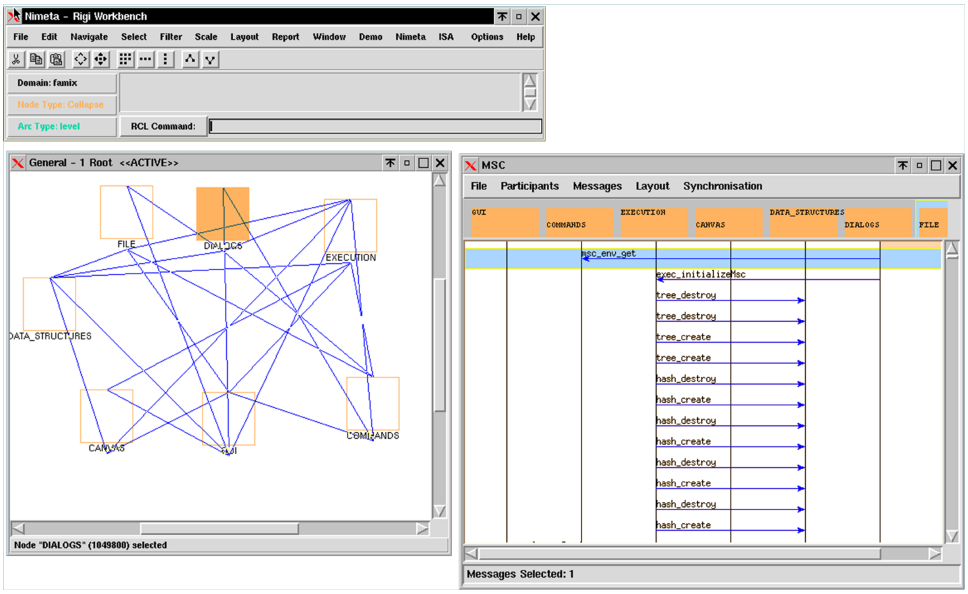


Figure 8.4: Overview of RIGI within NIMETA .

- The exporters from RIGI to other formats are implemented in TCL. They traverse the whole graphs and produce the information in the required format.

8.5.2 HAVA

We have extended RIGI with HAVA to support the combined analysis of static and dynamic data (Riva and Rodriguez, 2002). Dynamic data is extracted by collecting the traces of an instrumented software system and is typically drawn in a Message Sequence Chart (MSC). To cope with the complexity of the MSCs, HAVA provides two mechanisms of abstraction: horizontal and vertical abstractions. Horizontal abstractions allow us to group the participants of a MSC in order to reduce their number. Vertical abstractions consist with grouping a set of contiguous messages into one high-level message or scenario. The key feature of the integration is the possibility of synchronizing the static and the dynamic visualizations. As the participants are architectural entities in the static view, HAVA links the horizontal grouping in the MSC with the hierarchical grouping from the static view of RIGI . The Figure 8.5 shows the vertical and horizontal abstractions. HAVA takes care of synchronizing the groupings between static and dynamic views. This approach allows us to compress considerably the MSCs and to reveal the high-level dynamic dependencies between the subsystems. The Figure 8.6 shows an example of usage of Hava.

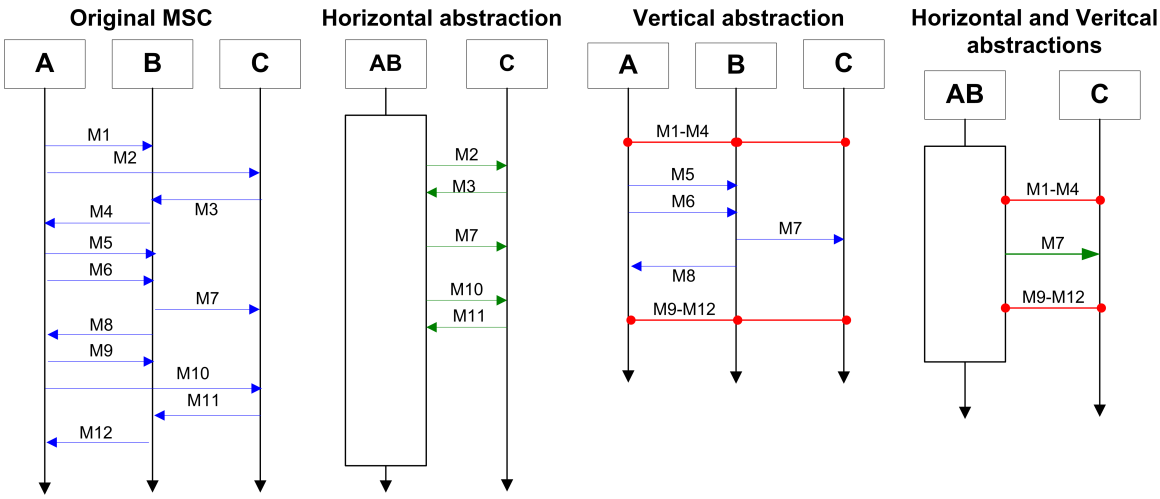


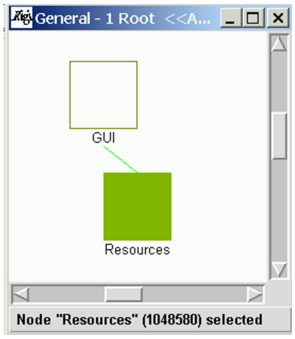
Figure 8.5: Vertical and horizontal abstractions in HAVA .

8.5.3 RATIONAL ROSE

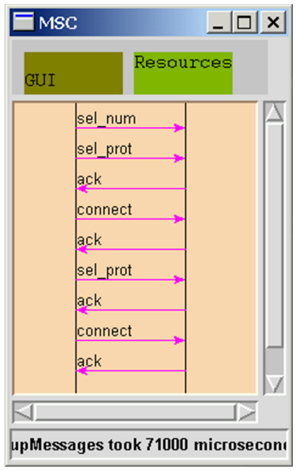
Rational Rose is a modeling tool based on UML. It is rather popular among the architects, although it cannot be considered an architecting tool. There are several reasons for being part of NIMETA . First, it is familiar to the architects that are the end-users of the reconstructed views. Second, it often contains architectural information required by the reconstruction process. Third, it allow us to create a proper architecting/rearchitecting environment. In the section Section 8.5.6 we discuss about the UML-based architecting environment, called ART, we have proposed.

The main point about Rational Rose's integration is the conversion of the reconstructed models to UML. The conversion is carried by establishing a mapping between the architectural concepts and the UML concepts, as explained by (Yang and Xu, 2003). We make an extensive use of stereotypes in order to express the architectural concepts properly. Table 8.3 shows the mapping rules that we typically use for the conversion to UML.

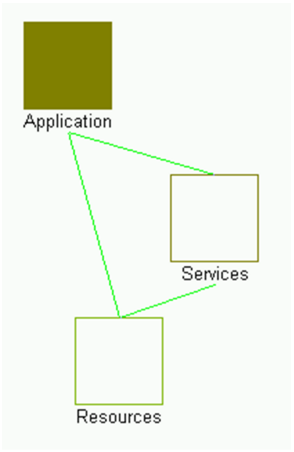
The computational units of the system are modeled using components. We do not use the UML "component" notation for the component because our definition does not match the UML definition. In our terminology, a component can be an abstract entity that might not have physical representation in the source code. UML components find a better use in the development view than in the logical view. Moreover, the graphical notation for the UML component is not commonly used by the developers. Services are modeled using interfaces. A component or a package can offer a service through an interface. An interface describes the operations that a component can handle.



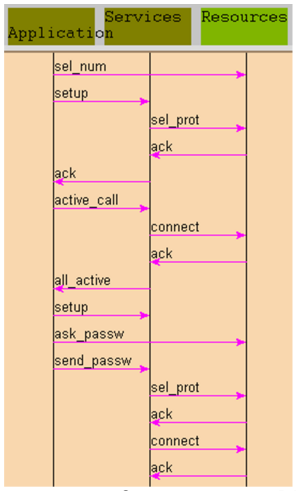
The high-level static view (a).



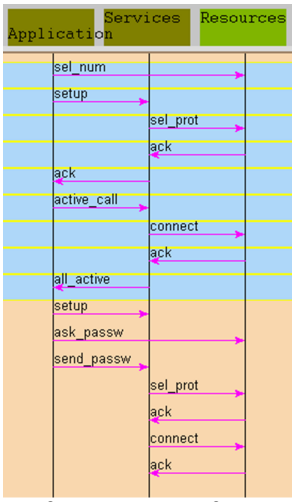
The high-level dynamic view(b).



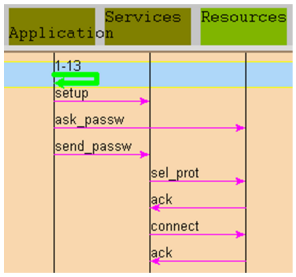
Content of GUI is expanded (c).



The messages of GUI are visible (d).



Selection of a sequence of messages (e).



The messages are collapsed (f).

Figure 8.6: Scenarios of usage of HAVA .

Architectural concept	UML element	Description
Package	Package	A package contains a set of logically related elements.
Subsystem	Package	A subsystem contains a set of functionally related elements.
Component	Stereotyped Class	A component represents a unit of computation or data storage. The granularity can vary from the a single procedure to an entire application. The stereotype defines the type of the component.
Interface	Stereotyped Interface	An interface represents the interaction point between a component and the environment. A component can offer services by means of its externally visible interfaces and can require services from other components. The type of the interface indicates the type of interaction that is supported by the interface (i.e. function call, asynchronous message).
Generalization	Generalization	A generalization is the relationship that exists between a more general element and more specific element.
Dependency	Stereotyped Dependency	A dependency exists between an element and the interfaces that is uses. The type indicates the type of dependency and it has to be supported by the interface.
Realization	Realization	A realisation is a relationship between an entity that implements an interface and the interface itself.

Table 8.3: The mapping between the architectural concepts and UML.

Concerning the dependencies, we have limited the notation to three relationships that are commonly used in architectural descriptions: generalization, dependency and realization. Generalization is used to show that a component derives some architectural properties from a more general or abstract component. At the architectural level, dependency is often a UML usage dependency among the components. Realization is used to show that a component implements an interface. In our description, we have not included the composition relationship because it is already achieved by using the package inclusion.

The conversion from the reconstructed views to the Rational Rose format is achieved by a a Python script. We have also implemented a set of scripts in Rational Rose for traversing the model and extracting the information that we need. An example can be seen in Figure 9.26.

8.5.4 WEB INTERFACE

The web interface represents an information system for publishing and distributing the architectural views within the organization through the intranet. The main design goal is to provide a simple-to-use interface not requiring special modeling expertise from the user expect a general knowledge about the architecture of the software system (end users are architects, programmers, managers, testers, designers). In our vision, the content of the web interface should be regularly updated (weekly or biweekly) in order to reflect the current implementation and to allow the users to find the latest architectural information. Our goal has not been to develop a general purpose web interface but rather a set of web components that we can reuse for creating web interfaces tailored to the particular software system under analysis. In other words, we want that the web interface speaks the same language of the developers. In section Section 9.7.5 we have presented the web interface that we designed for the case study 1. In this section we briefly describe its design.

The web interface consists of an APACHE [9] web server, a MYSQL database [10] for storing the architectural views, DOT from GRAPHVIZ [11] for the generation of graphs and a set of CGI scripts written in Python. From the RE model, we precalculate all the dependencies for the architectural views and we store them in the database. The CGI scripts query the data from the database to generate the HTML pages and to create the graphical diagrams on the fly. The user can access the information with any web browser. The architecture is shown in the diagram of Figure 8.7.

[9] APACHE : http://www.apache.org/
[10] MySQL: http://www.mysql.org
[11] GraphViz: www.graphviz.org

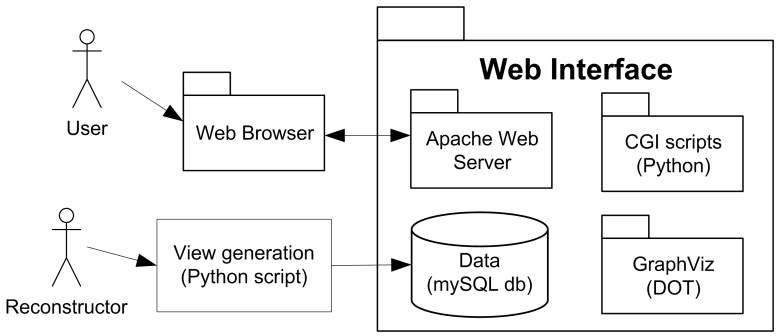


Figure 8.7: Overview of the web interface.

8.5.5 SOFTVIS

SOFTVIS , developed with the Eindhoven University of Technology, is an open visualization toolkit that we have adapted to support the reverse engineering activities (Telea *et al.* , 2002) and for the visualization of architectural views (Sillanpää, 2004). SoftVis's is based on two concepts: *datasets* and *operations*. As in a classical SciViz dataflow, operations can read and write parts of datasets. The desired functionality is achieved by sequencing several operations in a specific order starting from a particular dataset. In our toolkit, we use a centralized data model where there is a single dataset DS across the whole system. This dataset is read and written by all the operations that we need to support. The operations can be initiated by the user or the system, and can be applied in any desired order.

The architecture of the toolkit consists of two major components: the *toolkit core* and the *user interface*. The toolkit core contains the data set DS and the implementation of the operations for the creating the graphical visualizations. It is implemented as a C++ class library for performance and it is based on the Open Inventor toolkit[12] for the graphical visualizations (Wernecke, 1993). The user interface and the scripting layer are implemented in Tcl/Tk for flexibility. All the major functionality offered by the toolkit core is exported to the user with a Tcl API as a set of operations. This approach allows us to optimize the visualization engine for performance and to quickly prototype different type of visualization and interactions with the Tcl/Tk GUI. Below we describe the data model, the operations and the user interface of the toolkit

The basic data model is a typed attributed graph and contains three basic elements: *structure*, *attributes* and *selections*. The structure is the set of nodes and edges of the graph. Nodes and edges may have key-value pair attributes. We implement the keys as string literals and the values as primitive types (integer, floating-point, pointer, or string). Each node

[12]Open Inventor project: http://oss.sgi.com/projects/inventor

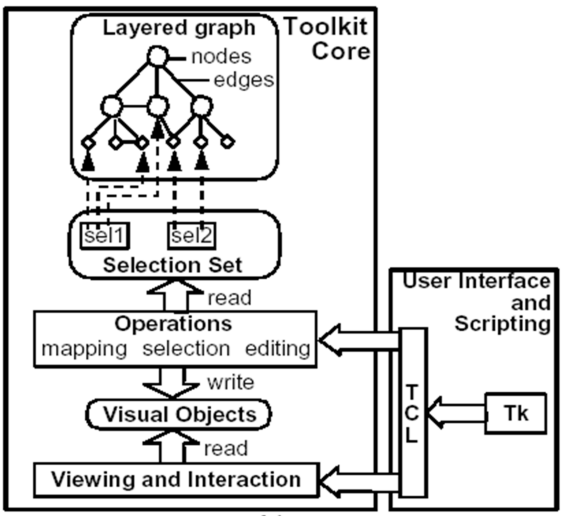


Figure 8.8: Overview of the web interface.

and edge has a set of attributes with distinct keys, managed in a hash-table-like fashion. Attributes automatically change type if written with a value of another type. Several attribute planes can coexist in the graph. An attribute plane is defined implicitly as all attributes of a given set of nodes/edges for a given key. Our attribute model differs from the one used by most SciVis (Schroeder *et al.* , 1998) and RE applications like RIGI (Wong *et al.* , March 1993) which choose a fixed set of attributes of fixed types for all nodes/edges. Our choice is more flexible, since a) certain attributes may not be defined for all nodes, and b) attribute planes are frequently added and removed in a typical RE session. Selections, defined as sets of nodes and edges, allow us to execute the toolkit operations on a specific subset of the whole graph. To make the toolkit flexible, we decouple the definition of the operations from the selections on which they are executed, similarly to the dataset-algorithm decoupling in SciViz. Selections are named and stored in variables accessible to the user as key-value pairs selection-set, similarly to attributes. Overall, our graph and selection data model is quite similar to the one used by the GVF toolkit (Marshall *et al.* , 2001). Our graphs are structurally equivalent to the node-and-cell dataset model in SciViz frameworks, whereas our selections do not have a direct structural equivalent. Selections are functionally equivalent to SciViz datasets, since they are the operations' inputs and outputs. As pointed out by (Marshall *et al.* , 2001), this is one of the main differences between SciViz and graph-based toolkits which leads to different architectures for the two. The Figure 8.9 shows an example of graph visualization with SoftVis. The graph in the bottom of the figure shows the containment structure of the hierarchical graph. The graph on the top-right shows the high-level dependencies between the top-level elements that have been selected from the containment tree. The graph in the top-left shows the same selection in a nested diagram

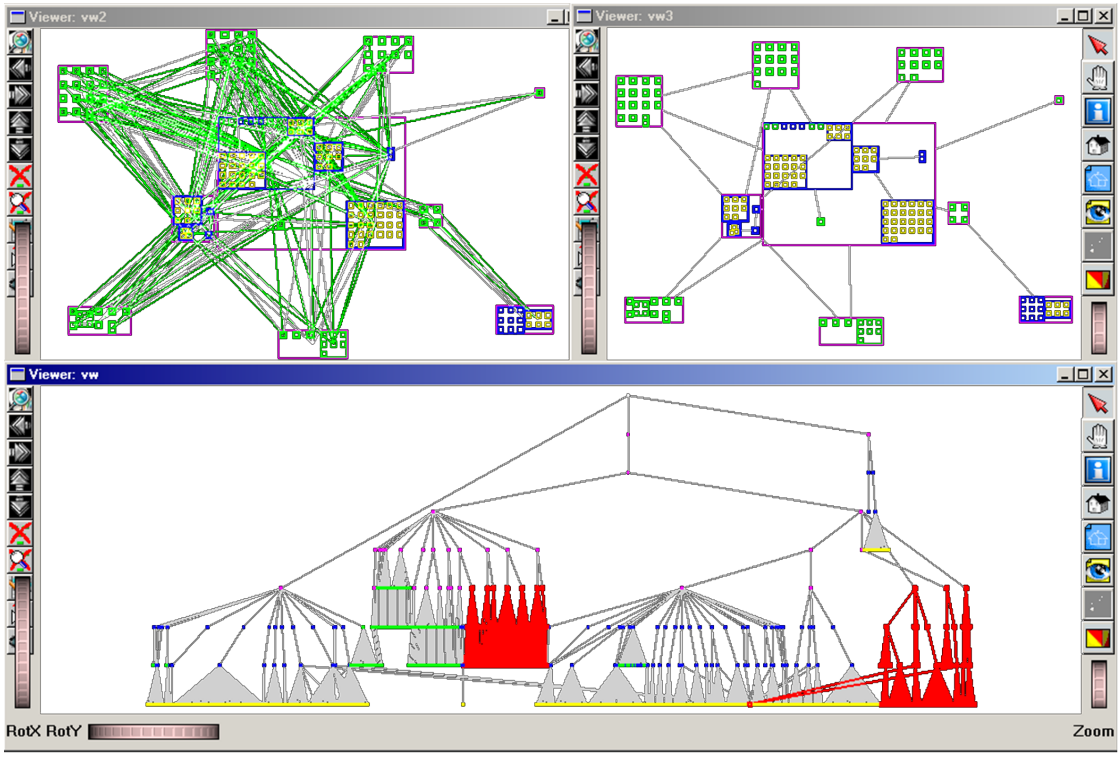
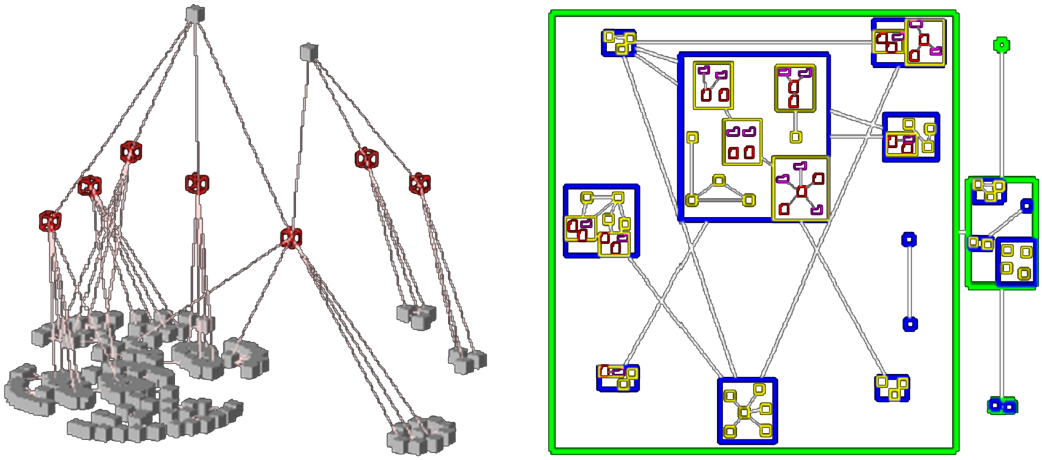


Figure 8.9: Multiple graph visualization in SoftVis.

where all the low-level dependencies are also visible.

The *operations* implement a particular functionality of the toolkit and modify the unique dataset DS. The operations have three types of inputs and outputs: *selections* that specify on which nodes and edges to operate, *attribute keys* that specify on which attribute plane(s) of the selection to work, and *operation-specific parameters* such as thresholds or factors. We can categories the operations according to their read/write data access: selection operations, graph editing operations and mapping operations. The toolkit architecture (based on the dataset-operation paradigm) allows the system to automatically update all components that depend on the modified data after the execution of any operation. For example, the selections are automatically updated after a structure editing operation which deletes selected nodes or edges. Similarly, the data viewers are updated when the selections that they monitor change. Although this is largely similar to the SciViz dataflow mechanism, we do not explicitly construct an operation pipeline. The reason is that, in contrast to SciViz applications, reverse engineering operations are seldom executed in the same order in different RE sessions. The typical RE task can be summarized as follows: (1) select a subset of nodes/edges of interest, (2) apply some editing operations on the selection, (3) map the selection to a particular visual representation.



The stacked layout (a). The nested layout (b).

Figure 8.10: Different layouts with SoftVis.

We treat graph layouts simply as attribute editing operations and thus decouple them completely from mapping and visualization. This has several benefits. First, we can lay out different subgraphs separately, e.g. using spring embedders for call graphs and tree layouts for containment hierarchies. Second, we can precompute several layouts e.g. to quickly switch between them. Finally, we can cascade different layouts on the same position attributes. We have implemented several custom layouts by cascading simple ones, as follows. Stacked layouts (show in Figure 8.10-a) lay out a selection spanning several layers of a graph by applying a given 2D layout (e.g. spring embedder) per layer and then stacking the layers in 3D. Stacked layouts visualize effectively both containment (vertical) and association (horizontal) relations of a software system. Nested layouts (show in Figure 8.10-b) lay out a similar selection as above, by recursively laying out the contents of every node separately and then laying out the bounding boxes of the containing nodes. Nested layouts produce images similar to UML class diagrams and are very helpful in RE applications. Users can easily combine any 2D layouts as the building bricks for the stacked and nested layouts For example, in Figure 8.10-a we use a tree layout, whereas in Figure 8.10-b we use a spring embedder as basic layout.

8.5.6 ART ENVIRONMENT

In conjunction with the Tampere University of Technology, we have started the development of an architecting environment, called ART, that is based on UML (Riva *et al.* , 2004b) (Riva *et al.* , 2004a). The purpose of ART is to facilitate the architecture design, reconstruction and maintenance in the entire life cycle of a software product line. The environment consists of tools for architecture model validation, architecture model analysis and processing, and

for reverse architecting. ART largely reuses the work that has been presented in this thesis for the reverse architecting part of the environment.

The ART environment is based on three modeling concepts: architecture profiles, architecture design models and recovered architecture models. They allow the architects to carry out the following operations:

- The architect can specify the *architectural profile* that defines the fundamental architectural concepts, the architectural style and the architectural rules through UML diagrams.
- The architect can create the *architecture design* model of the system with UML diagrams and validate it against the architectural profile. The architecture design contains the structural design of the system in terms of desired functionality, interfaces, partitioning and dependencies.
- The architect can reverse architect the architecture design models (the *recovered architecture* model) from the implementation and check their conformance against the architectural profile and the architectural design. The reconstruction is carried out according to the architectural profile with the process and the tools presented in this dissertation.

UML is the basic formalism and ADL for ART. Although UML is a general purpose modeling language, the use of the architectural profiles allow us to limit and customize the UML metamodel. A profile, defined through UML's extension mechanism, is an UML profile that presents the architectural rules and the legal elements allowed in the UML description of the architecture design. The UML diagrams in Figure 8.11 show an example of the stereotype definitions for the components and the independencies respectively. (the example is based on the case study 1). The diagram in Figure 8.12 defines the architectural rules for the profile. The architecture design model must conform to the rules in the profile in order to be valid. We can also check the reconverted architecture if the rules are satisfied. We use xUMLi processing platform for implementing the operations on the UML model (Selonen and Xu, 2003).

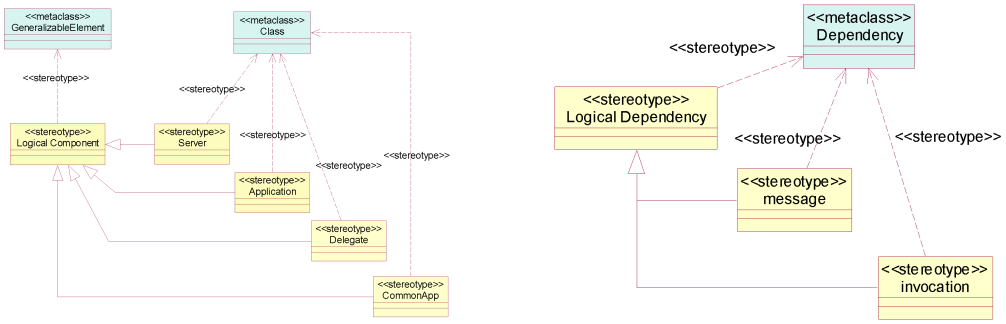


Figure 8.11: The stereotype definitions for the components (a) and for the dependencies (b).

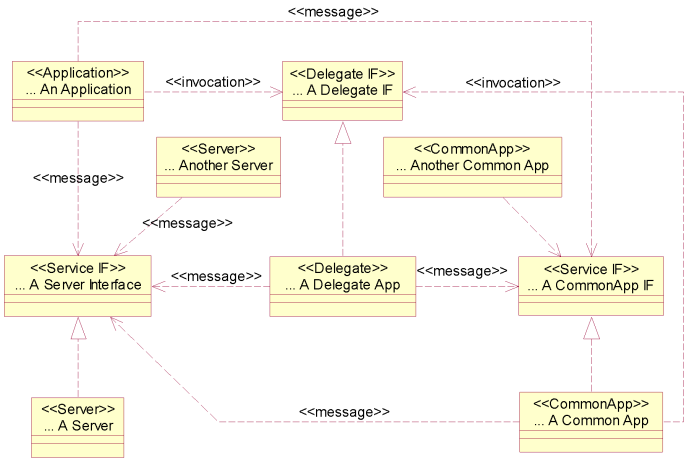


Figure 8.12: Example of constraint definition for the architectural profile.

CHAPTER 9

CASE STUDY 1: NMP PRODUCT FAMILY

To think is easy. To act is hard.
But the hardest thing in the world is to act
in accordance with your thinking.

- Johann Wolfgang von Goethe

This chapter describes the case study that we have conducted over a period of three years for assisting the architects of a large Nokia product family. We present an introduction of the domain and the motivations for the architecture reconstruction. We describe in detail the three iterations that we have carried out.

9.1 INTRODUCTION

The present case study has been conducted with one department of Nokia Mobile Phones (NMP) [1]. NMP is the world's largest mobile phone manufacturer. It develops mobile phones for all major telecommunication standard (mainly GSM, TDMA, CDMA and 3G) and customer segments in 130 countries. It has a comprehensive product portfolio that consists of several categories for every diverse needs, lifestyle and preferences of customers. More than hundred millions mobile phones are sold per year. It is characterized by a highly dynamic domain and fast changing requirements where new features have to be continuously added to the new products. NMP implements a fast product development in order to launch several new products every year. The products are organized in several product families that share common requirements and, consequently, share common features, chuck of functionality,

[1] Nokia Mobile Phones is part of Nokia Group: www.nokia.com

architectural concepts or code typically in the form of components. Among the various products the variance factors are: handset category (low-end, high-end and niche products), telecommunication protocol, user interface, operating system and customer/country customization.

As a case study, we have selected one large product family developed by NMP. Over the last three years, we have been consulting the architects of the product family and we have developed an architecture reconstruction process. The duration of the case study is influenced by two factors. The first one is that our reconstruction method and the related techniques have been developed concurrently with their application on the case study. The second reason is related to the maturity of the development process. When we started the case study the product family had only one product in the portfolio. The architects had the long-term target of increasing the throughput of the family to tens of products every year. The motivation for the reconstruction originates from the need of creating an adequate architectural governance for the coming years when the product family will be in full operation mode. During the case study, the family followed the evolution pattern that we have summarized in Section 2.4.2 and it is currently reaching the fourth maturity level (the maturity levels are described in Section 7.4). The code base increased from about one MLOC up to three MLOC. The number of logical components is about hundreds included in each product. The system is written in a dialect of C where macros are used to support the object orientation.

Throughout the chapter, we will refer to the product family with the acronym NPF (Nokia Product Family). Nokia sensitive details have been omitted as much as possible and the diagrams have been simplified, as long as this does not harm the explanation. Due to the complexity of the NPF's software architecture, we have also left out technical details that would require a deep understanding of NPF's architecture. We concentrated on those concepts that required by for understanding the essence of the case study.

9.2 Overview of the system

In Section 2.4.1, we have presented the architecture of the product family in this case study. The mobile phone products are perceived by the users and developers as a set of features. Features are pieces of functionality that offers some service or benefit to the users. Features are typically implemented with one or more software components and they can ultimately need special hardware support.

The diagram in Figure 9.1 sketches the simplified development process of NPF. The requirements for the various products are analyzed and merged in order to define a common roadmap for the products and the SW/HW platform. The roadmap is globally defined and

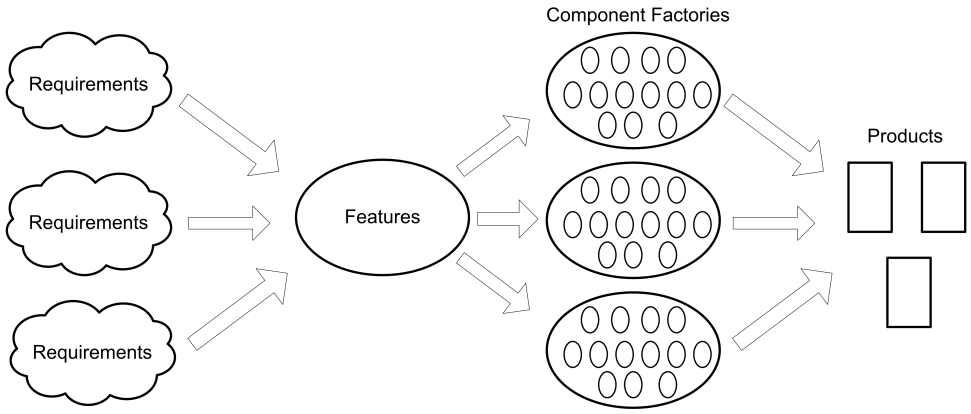


Figure 9.1: Simplification of the development process.

the platform's releases are synchronized with the needs of the products. The requirements are decomposed in a set of features. Features are pieces of functionality that offers some service or benefit to the users. Features are typically well perceived by the users who can recognize a value, by the marketers who can use if for advertising the product, by the developers for which it represent a concrete functionality to implement and by the testers who are responsible to test that the functionality is correctly implemented (for the term 'feature' we refer to the definition in Section 2.1. Features are typically implemented with a combination of hardware and software components. In the NPF, the software components are developed by the component factories, each factory having the responsibility for a set of components. The factories have their own roadmap and they regularly release the components. A particular release of the NPF platform contains a snapshot of the components that have been released by the the factories. The creation of the products is responsibility of the product projects which integrate the various components and create a proper and coherent user interface for the product.

In a component-based development approach, the work is focused on the development of generic reusable software components that are shared among the various products. However, this ideal approach has many organizational implications. The component development has to be technically organized by the architecture team for managing the platform and correctly partitioning the functionality in components. The component factories have to be able to develop and maintain their components and they are also responsible for assisting the product projects, correcting the problems and further developing the components.

One critical issue concerns with the integration of the components in the the products. The integration phase often puts a strong pressure on the project. The feature interaction (functional or logical dependency between features) is often unavoidable and difficult to control. To avoid costly delays in the integration phase, we identify and specify the possible

Iteration	Target viewpoints	Data gathering	Knowledge inference	Presentation
1^{st}	Logical view	manual extraction with GREP	manual grouping according to hypothetical logical view	UML diagrams
2^{nd}	Component view,	regular expressions in Perl scripts	manual grouping according to hypothetical logical view	hierarchical graphs with `Rigi`,
3^{rd}	Component view, Development view, Deployment view, Task view, Feature view, Organizational view,	regular expressions in Python scripts	relational algebra, linker output, mapping tables	hierarchical graphs, UML diagrams in Rational Rose, hyperlinked web pages

Figure 9.2: The summary of the iterations for the case study 1.

feature interactions as early as possible during the design phase. The the work of (Lorentsen *et al.* , 2001) proposes a method for modeling the feature interactions in mobile phones with explicit behavioral models of the features. The approach is based on the Colored Petri Nets and allows the UI designers to simulate particular features and analyze their behavior with automatically generated Message Sequence Charts.

The component factories often represent a bottleneck for the change requests made by the product projects. Change requests are regularly made by the projects that need modifications or bug fixes to the the components. The change request is forwarded to the factory that is responsible for prioritizing it and queuing in the change list. It can happen that the change request is reject and this can lead to lead to a stall in the development of the product. This often happens when the roadmaps of the platform and the product are not well synchronized.

9.3 SUMMARY OF THE ITERATIONS

Table 9.2 shows the summary of the three iterations. The first iteration was focused on defining the requirements of the reconstruction. The second iteration delivered an initial component view that we use to get a feedback from the stakeholders. The third iteration defines a complete process for architecture reconstruction and conformance checking.

9.4 THE ACTORS

We can identify three types of actors that are involved in the reconstruction process:

Reconstructor is the expert of the reconstruction process and is responsible for designing and conducting the reconstruction activity. In this case study, the reconstructor played also the role of the process designer. The reconstructor is also knowledgable of the domain. We have impersonated the role of the reconstructor.

Architects are the customers and the main stakeholders of the reconstruction activity. The architects are responsible for the architectural design of the overall product family. We have reported our results to the chief architect of the architecture team.

Experts have been interviewed for clarifications on specific areas of NPF. The experts have provided very direct and precise answers to our questions.

9.5 THE FIRST ITERATION

The first iteration is mainly focused on getting acquainted with NPF and collecting the requirements from the architects. Our team conducted a short manual reconstruction of one subsystem and we discussed the results with the architects. It was our intention to create a set of real use cases that could demonstrate how to document the architectural designs. The use cases were created in UML. At that time, UML was not in use by the development teams.

9.5.1 PROBLEM DEFINITION

When the first iteration was started, NPF represented the software platform for a new generation of mobile devices. Only one product was based on NPF and several other products were in production. Since NPF was designed to be in operation for the next decade, the architects were concerned that the architecture of NPF was robust and properly documented. Two activities were already started to document the *reference architecture* and the *logical view* of NPF.

The reference architecture of NPF was documented by interviewing the architects and the experts of the various areas (user interface, protocols, OS, core services). The reference

architecture summarizes the main architectural concepts, styles and patterns that are in use. It is a reference document that we have widely used during the next iterations.

The logical view was the first attempt to create an overall view of the structural aspects showing the logical components, the subsystems, their dependencies and the interfaces. The view was created manually by interviewing the architects and from the existing documentation. The logical view consists mainly of several UML diagrams showing the various subsystems grouped in layers and the logical dependencies.

During the initial discussions, the architects had a clear idea of their requirements for the reconstruction activity. They were lacking of an overall view of the structure of NPF. At that moment, the logical view was the closest approximation of what they needed. However, the logical view was imprecise for several reasons: (1) it was based on interviews and guesses but it did not reflect the real situation in the implementation, (2) the dependencies among the components were not complete, and (3) its maintenance was laborious and unsystematic. We can summarize the requirements of the architects in the following points that the logical view should document:

- what are the components and their interfaces.
- how the components are organized in subsystems and how the subsystems are allocated to the various layers.
- the logical dependencies between the components and the subsystems.

We quote some of the requirements that we noted during the initial meetings:

> **UI architect**: Does ANYONE have 'The Big Picture' of all applications and how they are all interlinked? How can we create one and maintain it? I think this might aid our error-corrections, and help making decisions about when and where to change/create applications/servers.

> **Chief architect**: What I want is an effective, and easy to manage, method of maintaining the overall design of the system. The simple questions would be ... What components are there? What services do they provide? What services of other components do they use? You could complicate this with ... How are they configured? Are they variants etc.?
>
> A system would be one product therefore the method would have to work with components existing in many systems and therefore many different configurations. Components are being released from many different sites.

> From my own point of view the only information I am interested in are the relationships between the logical components. i.e. application to server, application to delegate, server to server. The include information and the directory tree are not relevant. The OS interfaces are not relevant.
>
> Ideally what we need is to put some of these entities into packages, so we can see the relationship between packages at a higher level. I would also like that we could provide a way for a designer to focus on 'my application' or 'my server' and see all their relationships.

> **Architect**: We must find the mechanism how we can create models of our system design / architecture. First step for me is to find a way of reverse engineering the architecture of one just launched mobile phone software. How we should visualize these architectures? How we describe interactions and relationships between the components and entities: clients and servers?
>
> In a nutshell: to create architecture description of implemented mobile phone software and how to describe it.
>
> One way to do that is to explore the code (what are the dependencies between subsystems) and of course a lot of information can be found from product documentation but not so exact information like message interactions.

Based on these discussions, our goal was to prepare an initial demonstration of what information we can extract from the implementation and how we can present it. We focused on one single and well documented subsystem to avoid of being overwhelmed by the complexity of NPF. This demonstration would become the base for further discussing the requirements of the second iteration with the architects.

9.5.2 Concept Determination

In the first iteration, the target view is the logical view as it was intended by the architects. NPF is a distributed systems were the components can interact at runtime with asynchronous messages passed through a software bus. The entities in the logical view are the logical components: applications, delegates and servers. The logical dependencies are caused by the asynchronous messages and the function invocations between applications and delegates. An examination of the code showed that in most of the cases the recipients of the messages were statically bound, so a static analysis would be a sufficient starting point. The architects also proposed to group the components in packages in order to show the high-level dependencies. Therefore, we had to define also a containment relationship in the logical view.

9.5.3 DATA GATHERING

Due to the limited size of the subsystem to analyze, we opted for a quick and simple approach for extracting the data from the code. We took one release of the only complete product based on NPF and we used the UNIX tool GREP to find the code patterns representing the logical dependencies (asynchronous message and function invocation). The extracted information was manually added to a set of UML diagrams. The complete size of the build we analyzed was approximately 0.6 MLOC.

9.5.4 KNOWLEDGE INFERENCE

The containment relationship between packages and components was undefined and not available in the documentation. We proposed our own decomposition for the subsystem under analysis. The subsystem was decomposed in several packages, each containing a set of logical components.

9.5.5 PRESENTATION

We presented the reconstructed logical view using UML diagrams. Although UML was not commonly used by the developers of NPF, we decided to exploit UML for documenting the architecture. The Figure 9.3 shows an example of one of the diagrams that we have delivered in the first iteration.

9.6 THE SECOND ITERATION

Based the comments we got from the first iteration, we started the second iteration. The main goals were (1) to automate the reconstruction process and (2) to deliver a reconstructed logical view that could satisfy the architects. In this section we report the second iteration that has been presented in our work (Riva, 2000) and (Riva, 2002).

In the second iterations, we analyzed several products based on NPF platform. Since the platform was still rather young, we found many exceptions to the mappings rules that we define below. We reported those exceptions to the architects who fixed them in the following builds.

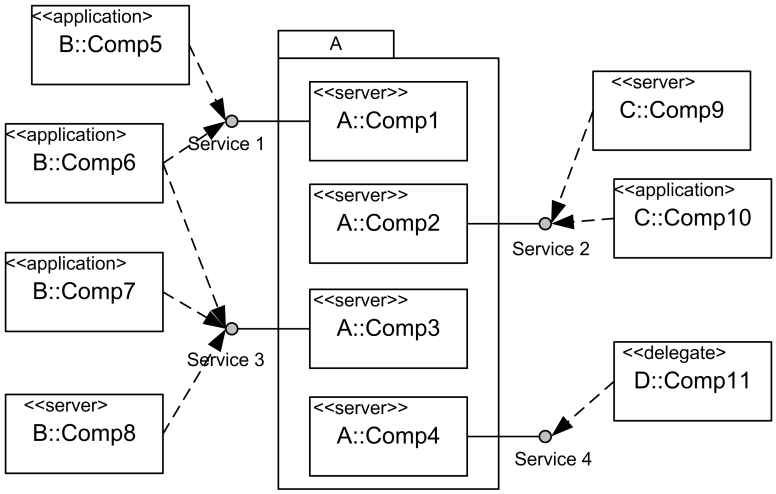


Figure 9.3: An example of the manually reconstructed logical view in UML.

9.6.1 Problem Definition

The manually reconstructed UML diagrams triggered useful discussions with the architects. This allowed us to clearly define the requirements of the reconstruction. The architects need an architectural model of the overall structure of the system (in terms of packages, components and interfaces). The model should allow them to look at the high-level and low-level logical dependencies among the packaged and components. In particular, it should be possible to query the model for the clients and suppliers of a particular component. In this way, the component owners (e.g. the factories) can identify their users and assess the impact of the changes.

The architects also would like to recover the architecture of the single products, compare them and analyze how they depend on the software platform of NPF. As discussed in Section 2.4.3, the evolution of the product family is mainly driven by two forces: the consolidation of the assets in the platform and the creation of new products. The platform evolves by incorporating the new architectural requirements, while new products with new features are added. Activities of the consolidation phase include:

- recovering the concrete product architecture (as opposed to the intended architecture that was in the minds of the architects)
- monitoring the organization of the components in the platform
- coping with the architectural dependencies within the platform and among products.

- enforcing the conformance to the architectural rules

The main goal of our architecture reconstruction method is to recover architectural models that the architects can use to comprehend the actual implementation of the products. The focus of our reconstruction is mainly on the architectural significant aspects of the products (e.g. the logical dependencies among the software components).

9.6.2 CONCEPT DETERMINATION

During the second iteration our goal was to prove that we could deliver the architecturally relevant information required by the architects. Our primary goal was to recreate the logical view as it was intended by the architects. The logical view shows the logical components and their logical dependencies. But what did the term *logical* really mean in the terminology of the architects ? What is the granularity of the logical components ? What are the logical dependencies ? Answering these questions is the core of the second iteration and in particular of the *concept determination* activity.

THE ARCHITECTURAL CONCEPTS

The reference architecture document contains the conceptual model of NPF. We use it for the identification of the architectural concepts at the correct level of granularity. The Figure 9.4 shows an excerpt that shows the three main building blocks (applications, servers, delegates) and the communication infrastructure for exchanging the asynchronous messages and the events. Discussions with programmers, architects, designers, testers and managers proved that these concepts represented the common terminology when talking about the architecture of NPF. This fact also confirmed that those concepts play a key role in the design of NPF and they are the first-class entities of the reconstruction process.

With the help of the system experts we identified how the architectural concepts (applications, delegates and servers) are mapped to the implementation. Conceptually, they belong to three different layers in the architecture. The *servers* belong to the low layer that is responsible for controlling the system resources. The intermediate level contains *applications* and *delegates* implementing reusable functionality. The topmost layer contains *applications* implementing the features. Each entity is implemented in one directory in the file system, though there are exceptions. At runtime, each entity has a unique identifier that is statically assigned at compile-time. The identifier is contained in a configuration file and it can be used to create the link directory-identifier-entity.

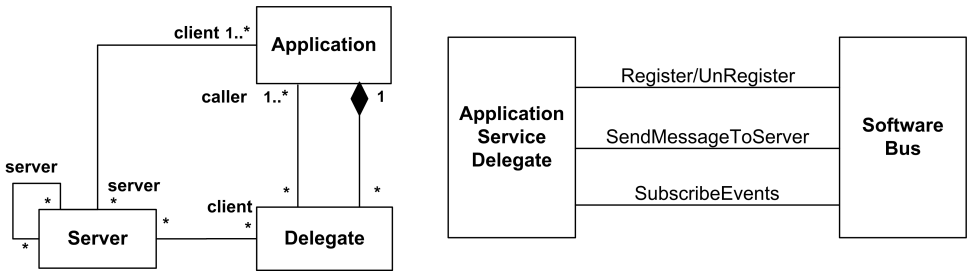


Figure 9.4: Excerpt from the reference architecture.

There are three mechanisms of interaction: asynchronous messages, the events and function invocations. They are implemented using well-defined patterns that we can easily detect in the source code.

THE TARGET VIEWPOINT

Our goal is to recreate the logical view. We define the target viewpoint in the following way. It contains four types of entities: `Server`, `Application`, `Delegate` and `Package`. There are three types of relations: *message*, *event* and *invocation*. The `Interface` represents the set of all messages that can handled by one server. Although logically we should model the interaction of a server through its interface, this was not explicitly required by the architects during the second iteration. The architects were interested in the logical dependencies among the components without the details of the interface. We also define a containment relationship that groups an `Application`/`Server`/`Delegate` in a `Package`. A package *A* depends on another package *B* if any of the components of package *A* has a relation with a component from package *B*. This relationship is expresses with the *dependency* relationship. The Figure 9.5 shows the diagram of the target viewpoint.

THE SOURCE VIEWPOINT

The diagram in Figure 9.6 shows the elements of the source viewpoint. The source viewpoint contains all the elements that are necessary to build the target view. We can distinguish between the elements in gray that represent programming language constructs and the elements in white that represent architectural concepts (used in the target viewpoint). We use two different techniques for their extraction from the implementation.

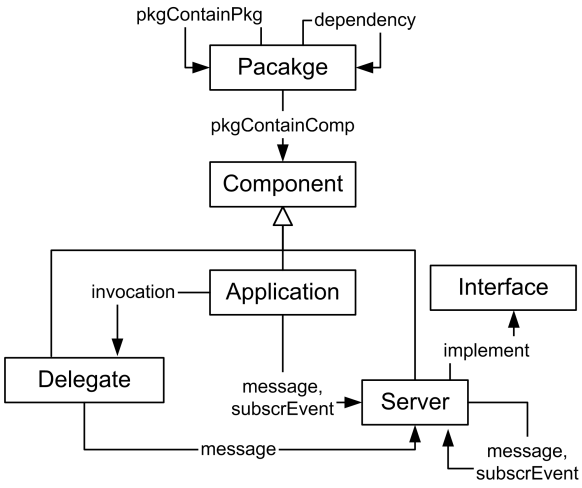


Figure 9.5: The target viewpoint of the second iteration.

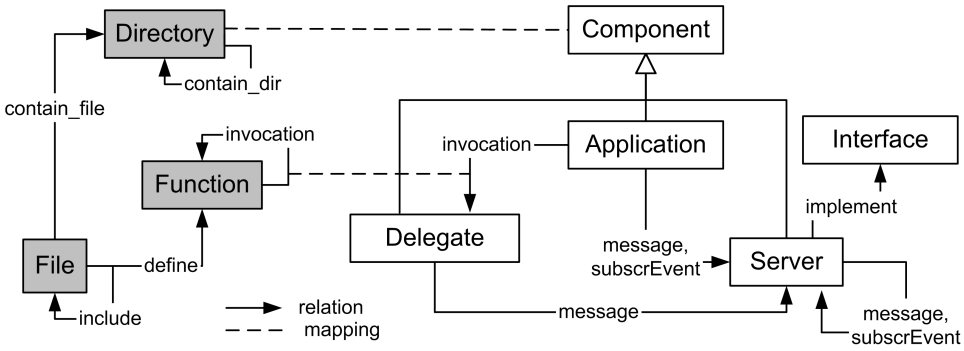


Figure 9.6: The source viewpoint for NPF.

THE MAPPING RULES

We list the rules that map elements from the source viewpoint to elements of the target viewpoint.

- The `Component` is directly mapped to a `Directory`. Although this is not a precise one-to-one mapping, it delivers a satisfactory level of accuracy for this iteration.
- The *invocation* between an application and a delegate is mapped to the *invocation* between two functions that follows a particular pattern. There is a particular code pattern for invoking a delegate.
- `Components` are manually grouped into a `Package`. There is no automatic way to define this mapping.

9.6.3 DATA GATHERING

We decided to analyze the source files without using existing analyzers or parsers for C/C++. One reason is that the system is written with a variant of C that makes an extensive use of macros. We do not have a parser for that variant and we did not have the resources to write one. The second reason is that we need to extract architectural information that is not semantically expressed using the programming language constructs (e.g. messages or events). Our choice was to rely on regular expressions. Each element of the source viewpoint is mapped to a code pattern that we can identity with one or more regular expressions. We wrote several scripts in PERL that can recursively examine all the source files in the file system and detect the code patterns. The scripts extract the following information:

- file system structure (directories and files)
- function definitions
- identifiers of the components
- interface of the servers
- messages and events
- function calls

We give an example of one regular expression for detecting the asynchronous messages:

```
/^.*send_message\s*\(\s*(\S+)\s*,.+,.+\)/
```

The source view is stored in a plain ASCII file in the RIGI Standard Format (RSF). The size of the build was approximately 3.6MLOC. The source model contained approximately 12,000 files, 200 applications and delegates, 120 servers, 80 invocations, 500 messages, and 100 events.

9.6.4 KNOWLEDGE INFERENCE

In the source view we have all information for creating the logical view, except the relationship of package containment that we created manually. We carried out this activity manually because it was not possible to automatically map components to packages. The mapping required a lot of mental reasoning. This was due to the low level of maturity of the organization during the second iteration. We relied on several indicators or heuristics for guessing the mapping component-package:

- Directory structure often suggested the package decomposition.
- Naming conventions (e.g. component/directory names with the same prefix).
- The hypothetical logical view that was maintained by the architects.

We grouped manually the components using the *collapse* functionality of RIGI . We manually selected the nodes and *collapsed* them in the `Package` nodes. RIGI automatically calculated the *dependency* relation among the packages while we grouped the nodes (the *dependency* relation is visualized with the *composite* arcs). The *pkgContainComp* and *pkgContainPkg* relation is the *level* arc in RIGI .

The result was the first reconstructed logical view that we delivered to the architects. We automated the grouping process with the scripting interface of RIGI . However, this was not satisfactory because the heuristics were not precise and we had to change the scripts for every build that we analyzed. We iterated this activity several times before we could deliver an adequate mapping. We held frequent meetings with the architects to show the reconstructed logical views and discuss the mappings. During the second iteration, the implementation was rapidly changing and this required frequent modifications to our process.

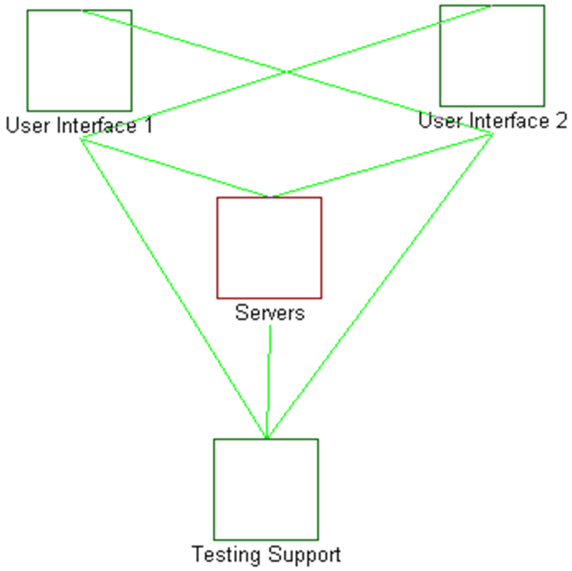


Figure 9.7: The top-level dependencies in the target view.

9.6.5 PRESENTATION

We presented the reconstructed views as graphs in RIGI . We created a new RIGI domain where the nodes are the entities of the target viewpoint and the edges are the relations. Since the visualization format is rather simple and intuitive, the end-user did not require a special training to learn how to use the reconstructed views. Since the architectural concepts are familiar to the developers, they could rapidly grasp the meaning of the diagrams. We distributed the logical views of several products and we got a positive feedback from the users. Below we describe several use cases that the end-users found particularly useful.

What are the top-level dependencies ?

This is a typical question for the architects who need to understand he overall view of the system. The graph is shown in Figure 9.7. The system is decomposed in four main packages that contain other subpackages and the components. There are two packages containing the components related to the user interface, one package containing all the servers and one package containing the testing components. The diagram revealed a suspicious cross dependency between the two UI packages that was not supposed to exist. This situation was further analyzed by the architects who browsed the view by opening the various packages (*expanding* in RIGI 's terminology).

What are the clients of a particular server ?

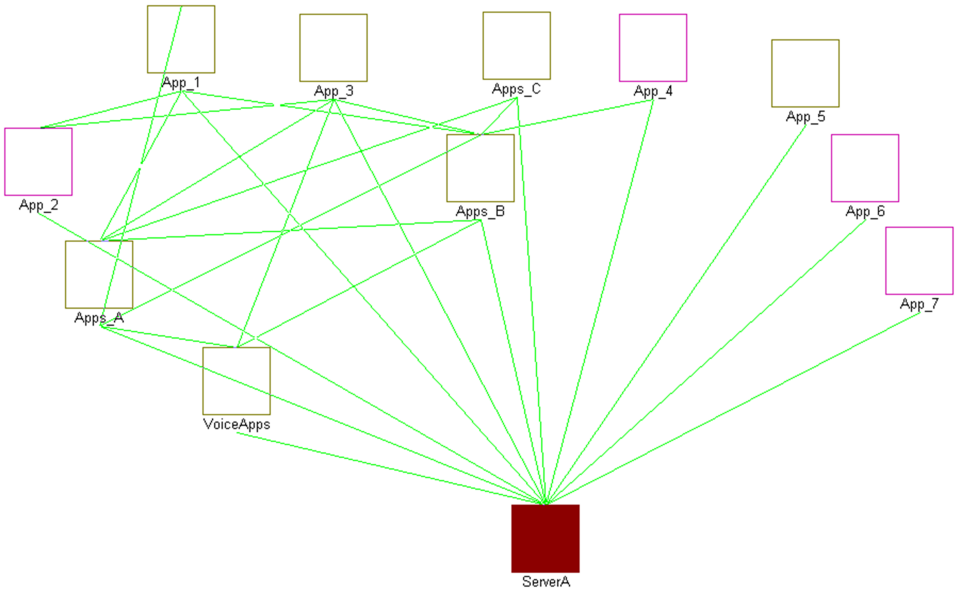


Figure 9.8: The clients of one server.

This is a typical question for the component factory that owns a server and wants to estimate the impact of changes in the server. When publishing a new version of the server, all the clients have to be informed. The graph can be obtained by selecting all the incoming nodes of the server and by filtering out all the rest, like shown in Figure 9.8.

What are the dependencies of one component ?

This question concerns with the context diagram of one component. The context diagram shows the clients and suppliers of the component. The graph in Figure 9.9 shows all the dependencies of the component `Apps_A`. At the top there are the clients, at the bottom the suppliers and on the left side there are the two components that have a cross-dependency with `Apps_A`.

How is an application using a server ?

This is a typical situation during debugging or testing. The user wants to find all the possible paths of execution between two components. In the example, we investigate how an application implementing one particular feature is using the services from a server. The graph in Figure 9.10 shows this situation with the application `MainApp` and the server `ServerB`. The application can directly access the server (by sending asynchronous messages) or indirectly through various delegates (by invoking the function of the delegates). We can identify six different paths of execution. We can also note that there are three layers

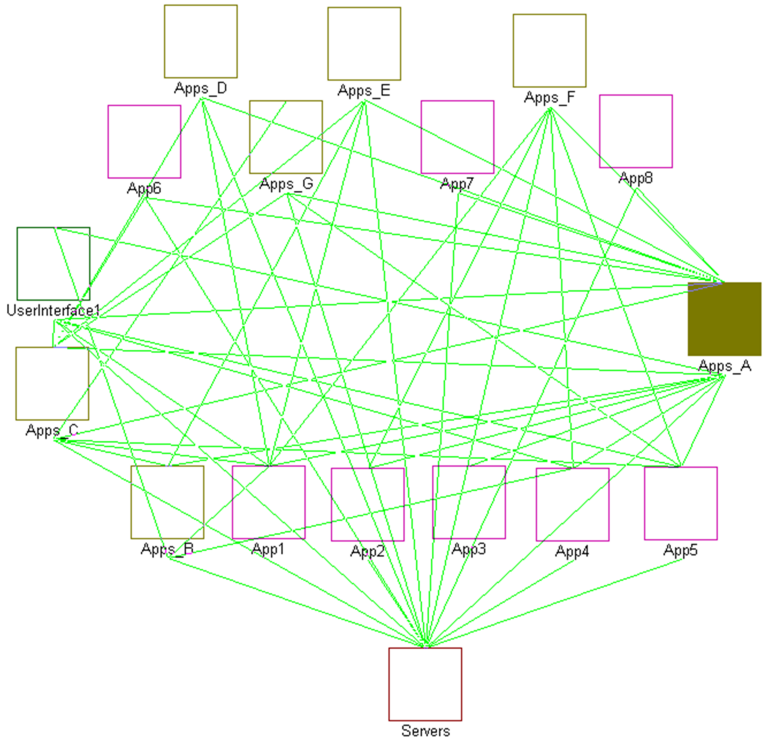


Figure 9.9: The context diagram of one component.

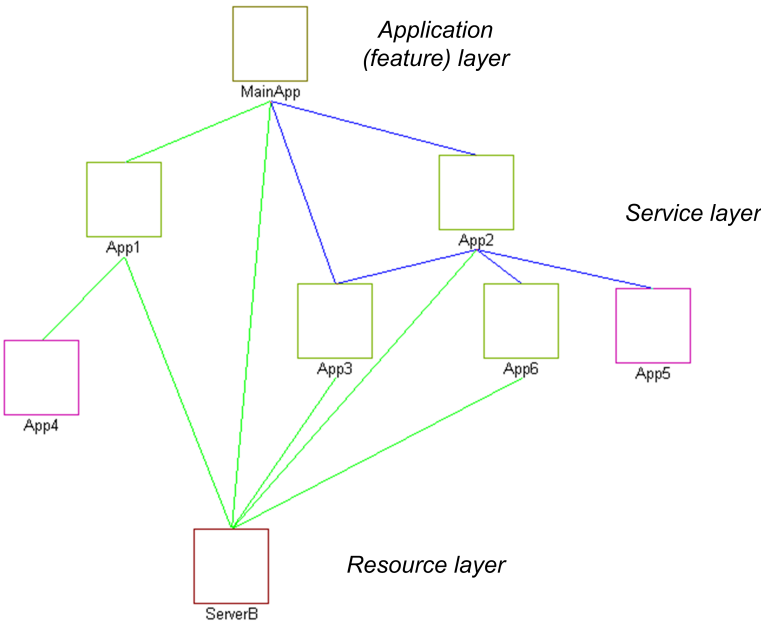


Figure 9.10: The clients of one server.

of functionality. At the top layer there is the application `MainApp` that implements the feature. We call this layer the *application layer* or *feature layer*. The intermediate layer is the *service layer* that contains the delegates that are implementing reusable functionality. The lower layer is the *resource layer* that contains the servers that are controlling the device's resources. It is an intention of the architects to enforce this three-layers design style in the implementation of the products. According to this rule, the direct communication between `MainApp` `ServerB` is forbidden. In the third iteration, we will automate the detection of these anomalies.

9.7 The Third Iteration

The architects gave us a positive feedback from the results of the second iteration. The recovered views delivered the architectural information they were interested in. However, the reconstruction process was still immature and not ready for its integration with the development process of NPF. This was the goal of the third iteration. We extended the reconstruction process to the whole platform of NPF, we made the data gathering activity more precise and we formalized the information required by the knowledge inference phase. We also defined more viewpoints and supported more presentation formats.

Since the beginning of the case study, NPF underwent profound changes. The platform increased its size with many new components. The source file system was largely reorganized and it became easier to map the architectural concepts to the implementation. NPF served as a platform for tens of products. At this moment, it was the interest of the architects to enforce a strong architectural governance on NPF that is the goal of this iteration.

9.7.1 Problem Definition

Besides the requirements that we identified in the previous iteration, the stakeholders of NPF wanted to enforce a strong architectural governance of the platform. This requirement involved to a create a precise mapping between the architectural concepts and the implementation.

9.7.2 Concept Determination

The architectural concepts that we have identified in the second iteration were satisfactory for the architects. We included few more concepts (like Library, HW Driver) and other that we do not mention for simplicity. The Figure 9.11 shows the meta-model of the concepts that we have identified. The elements in gray represent programming langauge constructs, while the elements in white represent architectural or organizational concepts.

The target viewpoints

The reconstructed logical view addressed most of the questions of the architects and we believe it is the most valuable view they need. However, we reorganized the output of the reconstruction in a set of views that address specific architectural concerns. The target views of the third iterations are listed below:

Component viewpoint : is concerned with the logical organization of the components and their logical dependencies. It describes the components, their interfaces, their logical relationships, the hierarchical composition of components in packages and the package-level dependencies.

Task viewpoint : is concerned with task allocation of the architectural entities and their inter-task communication.

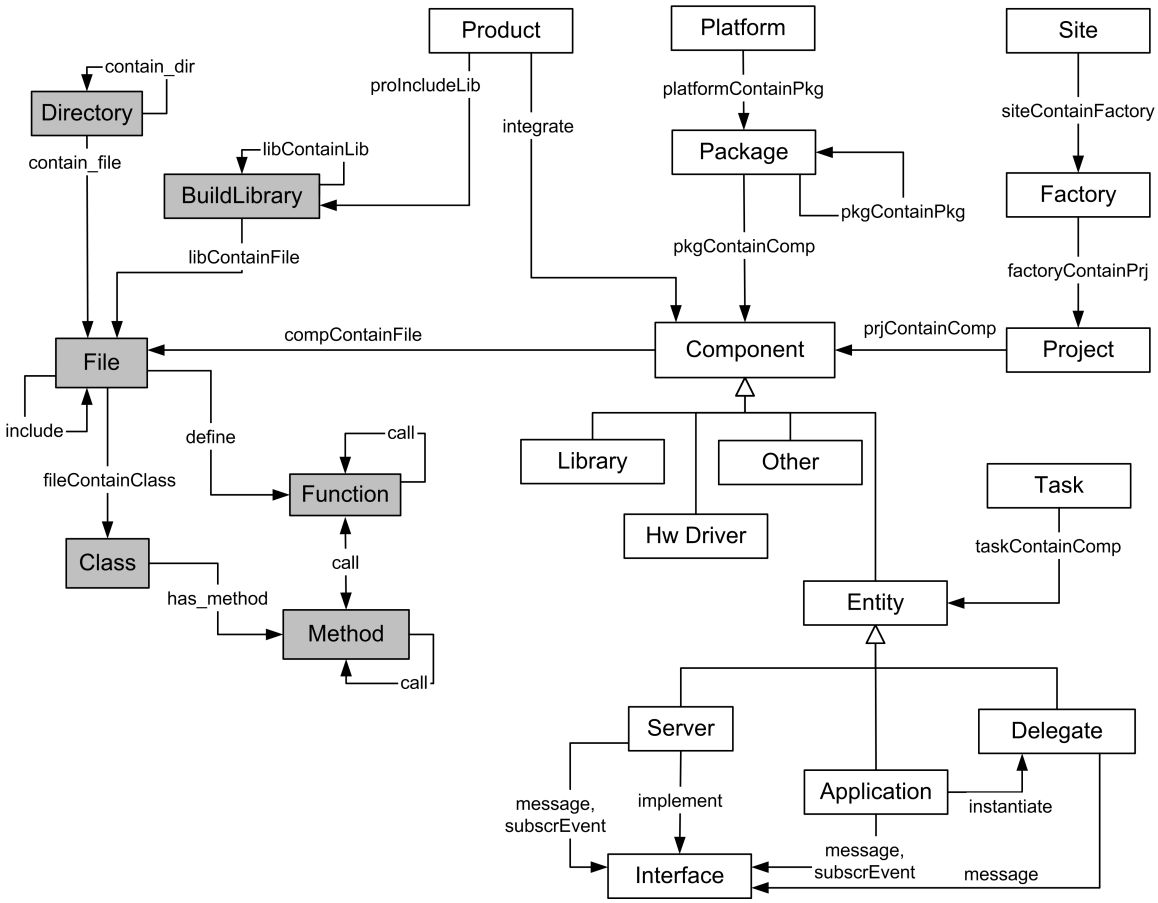


Figure 9.11: The meta-model of the NPF case that shows the architectural concepts.

Viewpoint		Entity (T_E)	Relation (T_R)	Containment (T_C)
Component		Component Package	message + instantiate + susbcrEvent	platformContainPkg + pkgContainPkg + pkgContainComp
Task		Task, Entity	message + instantiate + subscrEvent	taskContainComp
Development	dir	Directory, File,	fileCallFile[2] + include	contain_dir + contain_file +
Development	lib	BuildLibrary, File	fileCallFile + include	libContainLib + libContainFile
Development	comp	Component, File	fileCallFile + include	compContainFile
Feature		Application, Delegate, Server,	message + subscrEvent	taskContainComp/ pkgContainPkg + pkgContainComp
Organizational		Component, Project, Site Factory	message + call + subscrEvent	prjContainComp + factoryContainComp + siteContainFactory

Table 9.1: The target viewpoints.

Development viewpoint : describing the organization of the source code files and their relationships (for example, include dependencies)

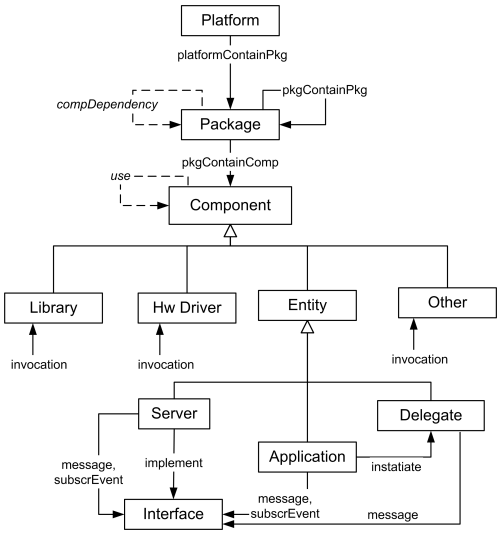
Organizational viewpoint : describing the organization of the development activities (projects, programs, sites).

Feature viewpoint : describing the run-time implementation of a feature at a high level of abstraction. The feature view is presented in the Section 9.8

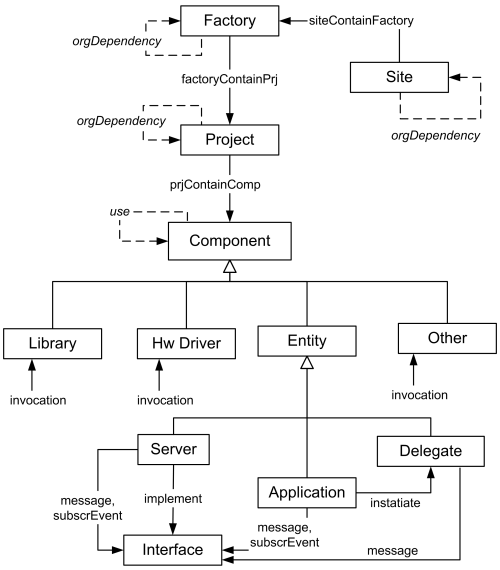
Table 9.1 defines the various viewpoints according to the viewpoint definition given in Section 5.5.

THE SOURCE VIEWPOINT

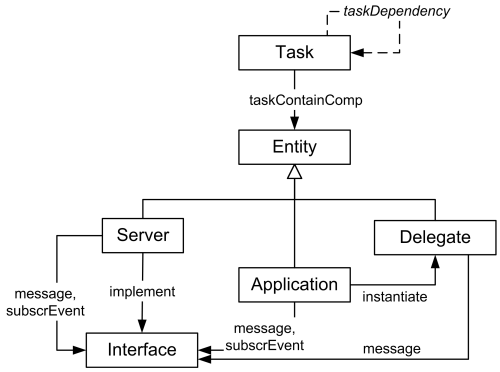
We split the source viewpoint in two independent parts that we can recover separately. The diagram in Figure 9.13 shows the first part of the source viewpoint that contains the implementation related concepts. It shows the file-level relations that we can extract by analyzing the source files. Those relations are necessary for inferring the relations in the target viewpoints. The elements in gray represent programming langauge constructs, while the elements in white represent architectural or organizational concepts. The diagram in



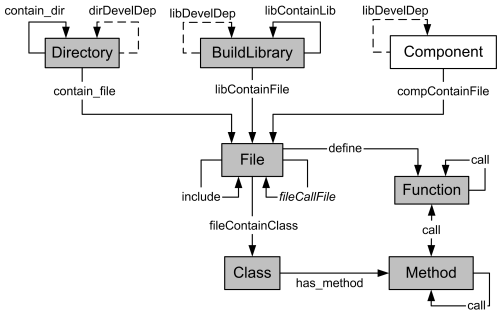
The component viewpoint.



The organizational viewpoint.



The task viewpoint.



The development viewpoint.

Figure 9.12: The target viewpoints.

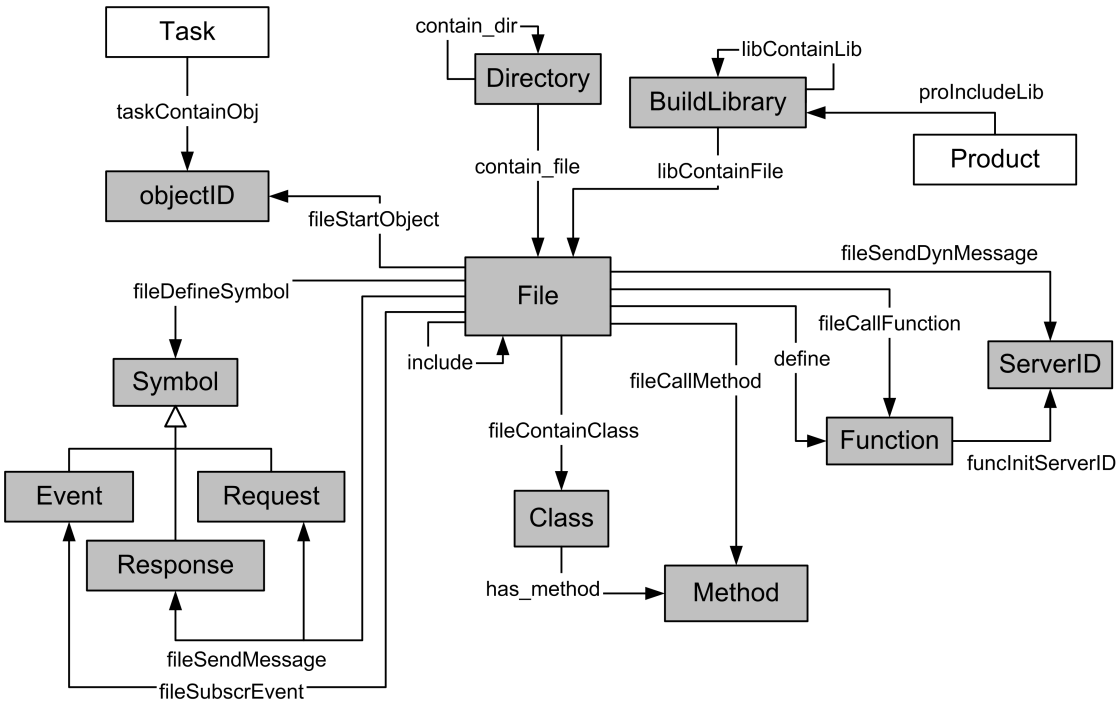


Figure 9.13: The source viewpoint for programming language concepts and the code patterns of NPF.

Figure 9.14 shows the second part of the source viewpoint that contains the concepts from the domain knowledge. Table 9.2 shows the relations that are part of the source viewpoint.

THE MAPPING RULES

We refined the mapping rules from the second iteration and we list them below.

- A `Component` is mapped to a group of `File` elements that represent its implementation. The relation *compContainFile* contains the one-to-many mapping between components and files. The component-directory mapping that we used in the second iteration is not enough fine grained because often files that belong to different components are located in the same directory. The *compContainFile* allow us to map elements from the target views to the elements of the source views. The map is created during the knowledge inference activity.
- The *instantiate* relation represents the instantiation of a delegate from an application. This instantiation is achieved by calling the delegate's initialization function whose

Relation	Source	Destination
contain_file	Directory	File
contain_dir	Directory	Directory
libContainLib	BuildLibrary	BuildLibrary
libContainFile	BuildLibrary	File
porIncludeLib	Product	BuildLibrary
define	File	Function
fileContainClass	File	Class
has_method	Class	Method
fileCallMethod	Method	Method
fileCallFunction	Function	Function
include	File	File
funcInitServerID	Function	ServerID
fileSubscrEvent	File	Event
fileDefineSymbol	File	Event|Request|Response
fileSendMessage	File	Response|Request
fileSendDynMessage	File	ServerID
fileStartObject	File	objectID
taskContainObj	Task	objectID
pkgContainComp	Package	Component
pkgContainPkg	Package	Package
platformContainPkg	Platform	Package
prjContainComp	Project	Component
factoryContainPrj	Factory	Project
siteContainFactory	Site	Factory
type	*element*	*type*

Table 9.2: The relations of the source viewpoint.

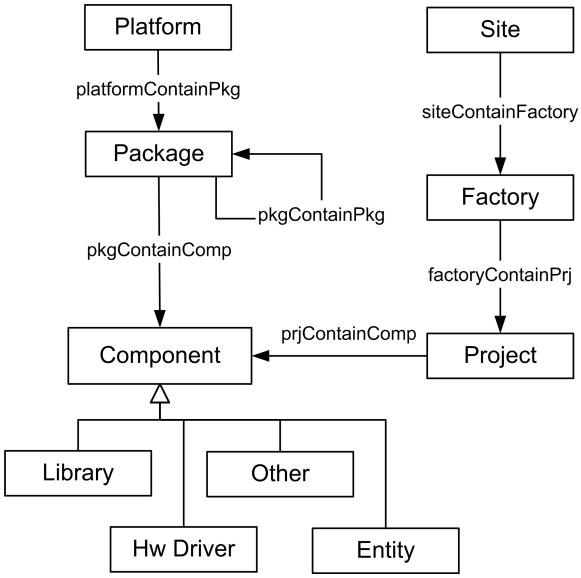


Figure 9.14: The source viewpoint for the domain knowledge of NPF.

name follows the convention: $\backslash w + \_Initialize$ (expressed as a regular expression). We can derive the *instantiate* relation from the *fileCallFunction* relation.

- The *message* and *subscrEvent* relations can be derived from the *fileSubscrEvent* and *fileSendMessage* through the *compContainFile* relation. The *fileSendDynMessage* is also mapped to the *message* relation for those cases where only the recipient of the message is known and not the message type (the message type is selected at runtime).

- The `Interface` elements represent the identifiers of the asynchronous messages and events. The *implement* relation is created from the *fileDefineSymbol* relation. The mapping can also be refined with the precise naming conventions that have been used for naming the identifiers.

- The *taskContainComp* relation is derived from the *taskContainObj* through the *compContainFile* relation.

9.7.3 Data gathering

In the third iteration we aim at creating a reconstruction process that can be integrated in the development process. We require an high-level of accuracy for the reconstructed architectural model. The accuracy of the target views is directly dependent on the accuracy

of the source view, and, therefore, we have largely refined the data gathering activity by including more concepts and by being more precise than in the second iteration.

We summarize our extraction strategy for each of the three categories (as defined in Section 7.3.3):

Programming language concepts This category contains the gray elements in the Figure 9.13. They represent programming language concepts that can be directly identified in the source code. `Directory` and `File` elements are detected by scanning the file system. `BuildLibrary` element maps to a group of files and the mapping relation (*libContainFile*) can be recovered from the configuration files used during the build process. The `Class` and `Method` elements are identifiable through particular macro definitions. The `Function` elements are the standard C-like functions. The *invocation* relation is built by detecting the C function calls and the method invocations (realized through a particular macro).

The relation *proIncludeLib* is extracted from the output of the linker when the products are built. There are configuration files that are used to include and configure the various components for the products. Analyzing the output of the linker turned out to be the simplest solution. The configuration files are rather complex to parse.

For each element, we have defined a set of regular expressions that are searched through all the source files by a Python script. Since certain parts of the system are written in plain C, the architects can optionally choose to parse the source code with SOURCE NAVIGATOR and merge the output with the results of the Python script. This is the preferable choice for creating a detailed development view. However The Python script is the standard extraction procedure.

Code patterns The information about the `Component` elements is extracted by analyzing particular code patterns in the sources. The type, the name, the runtime identifier and the task of the components is extracted from the configuration files that are located in the same directory of the sources (although there can be more than one component defined in one directory). From the configuration file we also extract the initialization function of the component.

The `Interface` element represents the set of asynchronous messages handled by one server and it is extracted from the message definition table in the main C file of the server. The *implement* relation is also extracted in this way.

The *subscrEvent* relation is extracted from the registration table of servers and applications. The *message* is identified with the same pattern shown in the second iteration (see Section 9.6.3).

The code patterns are expressed as a set of regular expressions and they are searched by the Python script.

The source view is stored in a RSF file that contains the relations show in the Table 9.2. All the elements in the RSF file have a unique name. The analysis of the source files is carried out by a Python script that recursively traverse all the the directory tree and search the source files for the regular expressions. The relations are extracted at the level of files. This means that the source of the relations is often a `File` element. The relations for the source viewpoint can be directly computed from the file level relations through the relational algebra. This step is done during the knowledge inference activity when the mapping *compContainFile* is also available.

NPF is a product family and we are interested in analyzing the whole NPF platform. Since each product is configured differently, we had to analyze each separate product and merge the results in a unique model. This has been achieved executing the extraction script on the various configured products. A typical build consisted of about 4MLOC.

9.7.4 KNOWLEDGE INFERENCE

The knowledge inference activity is carried out in two steps. In the first step we complete the source view by recovering the domain concepts and the *compContainFile* relation. Since it is mainly a reasoning operation, we have located it in the knowledge inference. [3] In the second step we have generated the target views from the source views and from the mapping rules defined in Section 9.7.2.

RECOVERY OF THE DOMAIN KNOWLEDGE

The domain concepts of the source view (shown in Figure 9.14) are not directly available in the source code but they need to be recovered from the domain knowledge. We carried out this operation with the help of the architects who represent the main domain experts for this task. We created a component inventory that describes each logical component with various details like its type, logical interfaces, the organizational and functional ownership. Below we describe the various relations in detail.

At first, we created a list of all the logical components. We started with the incomplete list of the components from the hypothetical *logical view* that was created during the previous iterations. We gradually refined the list by adding the missing components that we iden-

[3] It could be argued that the recovery of the domain concepts is a data gathering activity. In the case of NPF the domain knowledge was not well-formalized and it has been mainly a conceptual process rather than a simply data gathering operation. In the future we expect that the domain knowledge will be complectly formalized, hence, it will be possible to extract it during the data gathering activity.

tified from the list of `BuildLibrary` elements and by manually inspected the directory structure. At the same time, we also created the *compContainFile* relation that maps the logical components to the source files. The relation *libContainFile* gave us a rough partition of files to build libraries to start with. We manually refined the decomposition to the level of components. More than 20,000 files were mapped to about 1,000 components.

Once the list of components was completed, we added the type information, the functional decomposition and the organizational decomposition. For the functional decomposition, we started with the decomposition that was available in the hypothetical *logical view* and we gradually refined it by adding new packages and allocating all the components of the platform to one package. In this way, we created the `Package` elements, the *pkgContainPkg* and *pkgContainComp* relations. For the organizational decomposition, we relied on the organizational charts of NPF. We identified the `Site`, `Factory` and `Project` elements and their mappings.

GENERATION OF THE TARGET VIEWS

The source view for NPF is complete and is the basis for creating the target views. We use the mapping rules defined in Section 9.7.2. The *compContainFile* relation is the only mapping relation between the two parts of the source viewpoint: the implementation and the domain concepts. Below we define the sequence of actions for creating the target views in binary relational algebra. The relations listed in Table 9.2 represent the initial set of relations for the source viewpoint. The relations of the target viewpoints are listed in Table 9.1.

We calculate the interface of the components. The interface is represented by the set functions, asynchronous messages and events that are defined by the components. For the `Server` components, we also map the server identifier to the component through the *funcInitServerID* relation. We can distinguish if a symbol s is an a request or an event with a regular expression that is based on the naming conventions of the symbols.

$$
\begin{aligned}
compDefineServerID &= compContainFile \circ define \circ funcInitServerID \\
compDefineSymbol &= compContainFile \circ fileDefineSymbol \\
compDefineRequest &= \{(c, s) \in compDefineSymbol \wedge s \text{ is a request}\} \\
compDefineEvent &= \{(c, s) \in compDefineSymbol \wedge s \text{ is an event}\} \\
compDefineFunction &= compContainFile \circ define
\end{aligned}
$$

We calculate the logical dependencies among the components from the file-level dependen-

cies:

$$\begin{aligned} implement &= compDefineRequest + compDefineEvent \\ subscrEvent &= compContainFile \circ fileSubscrEvent \circ compDefineEvent^{-1} \\ message &= compContainFile \circ (fileSendMessage \circ compDefineRequest^{-1} + \\ &\quad fileSendDynMessage \circ compDefineServerID^{-1}) \end{aligned}$$

The *instantiate* relation is calculated by filtering all the function call relations that match the name of delegate initialization function. The file-level dependencies are lifted to the level of components through the relation *compDefineFunction* and *compContainFile*.

$$\begin{aligned} initDelegate &= \{(file, func) \in fileCallFunction \wedge func \text{ is like } < \backslash w+\_Initialize >\} \\ instantiate &= compContainFile \circ initDelegate \circ compDefineFunction^{-1} \end{aligned}$$

We calculate the component-level dependencies for the function call by lifting through the *compContainFile*.

$$\begin{aligned} fileCallFile &= fileCallMethod \circ has\_method^{-1} \circ fileContainClass + \\ &\quad fileCallFunction \circ define^{-1} \\ invocation &= fileCallFile \uparrow compContainFile \end{aligned}$$

We define the *use* relation among the components as the union of all the logical dependencies.

$$use = message + invocation + instantiate + subscrEvent$$

We define the containment relationship for the target views according to the list in Table 9.1.

$$\begin{aligned} compContain &= pkgContainPkg + pkgContainComp + platformContainPkg \\ orgContain &= siteContainFactory + factoryContainPrj + prjContainCOmp \\ taskContain &= taskContainComp \end{aligned}$$

We lift the *use* relation with the containment relations for each view.

$$\begin{aligned} compDependency &= use \uparrow compContain \\ orgDependency &= use \uparrow orgContain \\ taskDependency &= use \uparrow taskContain \end{aligned}$$

Finally, we define the target views as the following graphs:

$$V_{Component} = G(T, use + compDependency, compContain)$$
$$V_{Organization} = G(T, use + orgDependency, orgContain)$$
$$V_{Task} = G(T, use + taskDependency, taskContain)$$

where the set T contains the type information.

For the development viewpoint, we define three different containment relationship for the three types of development views.

$$dirDevelContain = contain\_dir + contain\_file$$
$$libDevelContain = libContainLib + libContainFile$$
$$compDevelContain = compContainFile$$

We lift the function call and include dependencies with the containment relations [4].

$$develUse = fileCallFile + include$$
$$dirDevelDep = develUse \uparrow dirDevelContain$$
$$libDevelDep = develUse \uparrow libDevelContain$$
$$compDevelDep = develUse \uparrow compDevelContain$$

We define the development views as the following graphs:

$$V_{dirDevelopment} = G(T, develUse + dirDevelDep, dirDevelContain)$$
$$V_{libDevelopment} = G(T, develUse + libDevelDep, libDevelContain)$$
$$V_{compDevelopment} = G(T, develUse + compDevelDep, compDevelContain)$$

where the set T contains the type information.

In the environment NIMETA , the operations are implemented with Python scripts. Starting from the source views, it is possible automatically generate the target views. The target views are stored in a RSF file and in the component repository.

[4] We note that in the description of the case study we do not explicitly base the development view on the *call* relation among functions and methods as. In practise, we calculate the *call* relation only for certain parts of the system and in most of the cases the *fileCallFile* is satisfactory.

9.7.5 PRESENTATION

The goal is to effectively communicate the reconstructed architectural information to the development teams. The stakeholders desire that the architectural information becomes easily accessible to all the developers (managers, programmers, designers, testers). This represents a wide variety of people with different background, interests and knowledge about the system. The reconstructed architectural models cover the whole platform, therefore we expect that they will be queried either by (1) people who have a deep knowledge in a particular domain and need to find more information, or by (2) people who are just knowledgable and need to get an overview. The presentation should convey views at different levels of abstraction and should allow experts to dive into the details.

We extended the scenarios from the second iterations(see Section 9.6.5) to the following list:

- The chief architects of NPF want to look at the top-level structure of the platform and to analyze the dependencies by descending the hierarchy through the various packages to the level of components.
- The project managers want to look at the top-level organizational aspects of the architecture in order to keep the coupling among the sites and factories under control.
- The architects of certain functional domain want to look at the design of certain areas of NPF and how the various components collaborated to implement the features.
- The designers want to look at the design of their and other components' interfaces and how these interfaces are implemented.
- The testers want to trace the dependencies among the components in order to solve a bug and they want to know who are the owners of certain parts of NPF.
- The programmers want to look at the low-level dependencies among the components that they are developing.

The rest of the section is structured in the following way. First we review our experiments with visualization tools and the we create a list of requirements for the presentation activity. Second, we describe the presentation approaches that we support for NPF: web interface, UML models and hierarchical graphs.

QUALITATIVE EXPERIMENTS WITH VISUALIZATION TOOLS

We conducted several usability tests with the stakeholders about the presentation formats. We focused our tests on the component view. We proposed them various representations based on existing tools. In order to conduct a realistic demonstration, we presented the tools with the real architectural views for a particular release of the system. We report the results in the Table 9.3.

From the experiments we observed that the stakeholders found the textual representations more informative than the graphical representation. In a typical component with about 20 clients/suppliers the graph becomes unreadable (for example the diagram in Figure 9.20) while a simple table or a list can quickly convey the essential information. However, the stakeholders agreed that the graphical representation is more intuitive than text if the information is properly represented. They also appreciated the hyperlinked pages for the possibility of navigating through the components and their dependencies. All the approaches lacked the ability of making direct queries. There was also a remarkable resistance towards the introduction of new tools (e.g. Rational Rose) who required deployment, training and support.

Concluding, we defined the following requirements:

Text and graphs Textual representation is the main presentation format that we should use to convey the architectural views. Graphical diagrams represent an additional format for presenting selected views on the data.

Navigability Due to the large size of the models, navigability is a main concern for the presentation. We need to support multiple ways for navigating through the views (e.g. hyperlinks, navigator bar).

Easy-to-use and intuitive The presentation format should require a minimal training, be very intuitive and easy to use.

Queryable The presentation format should allow the users to query the views according to several criteria.

Availability The architectural views are produced biweekly and we need a efficient mechanism for managing the various versions and delivering them to the users.

Tool	Presentation	Strong(+)/Weak(-) points
RIGI	Hierarchical typed graphs. The domain is defined according to the viewpoint.	+ visualization of hierarchical graphs + customization of domain of graphs + visualization of top-level dependencies + interactive navigation/filter of graphs - visualization of nested graphs - complex queries - layout algorithms on large graphs
Rational Rose	Graphs in UML.	+ UML conformance of the graphs + commercial support + usability + textual navigation bar - limited API for accessing the repository - graph layout
Venice [a]	UML nested graphs	+ nested graphs - prototype - not scalable
Web pages	Hyperlinked HTML pages containing summary of the components and links to the clients/suppliers. The users can follow the chain of dependencies through the hyperlinks. Described in (Riva and Yang, 2002).	+ very informative for components and dependencies + hyperlinks for navigating through the dependencies + web publishing in the intranet - static pages
Graphs with SVG[b]	RIGI -like diagrams rendered in a web browser through SVG and JavaScript for the interactivity	+ web publishing
DOT	Nested graphs	+ layout + Compact visualization of nested graphs
SoftVision	Following RIGI 's philosophy, the tool supports nested graphs, layout algorithms and possibility to customize the graphical appearance of the nodes and arcs.	+ layout + multiple synchronized views of the same data + navigability + scalable - training is required - no support for UML
SWAG Toolkit [c]	Web interface and nested graphs	+ web publishing - poor visualization of the graphs
Microsoft Excel	Spreadsheets listing all the components and dependencies	+ compact textual representation + queryable - not navigable

[a] Venice is a prototype for the visualization of nested diagrams in UML developed with the University of Helsinki. Our goal was to experiment RIGI 's philosophy of graph visualization with the nested graphs and UML. Available at http://www.cs.helsinki.fi/group/venice

[b] Scalable Vector Graphics: http://w3c.org

[c] SWAG Toolkit

Table 9.3: Summary of the tested presentation tools.

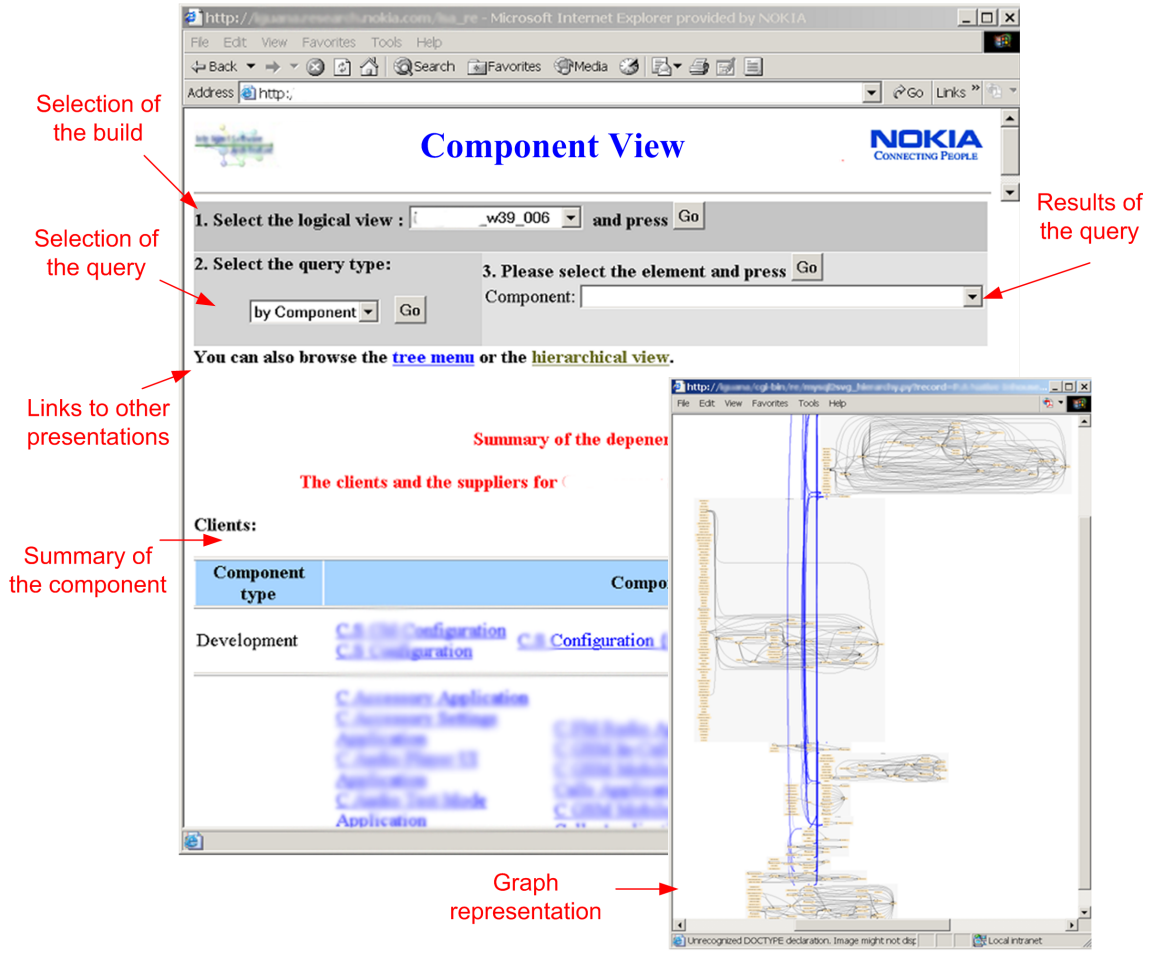


Figure 9.15: The overview of the web application.

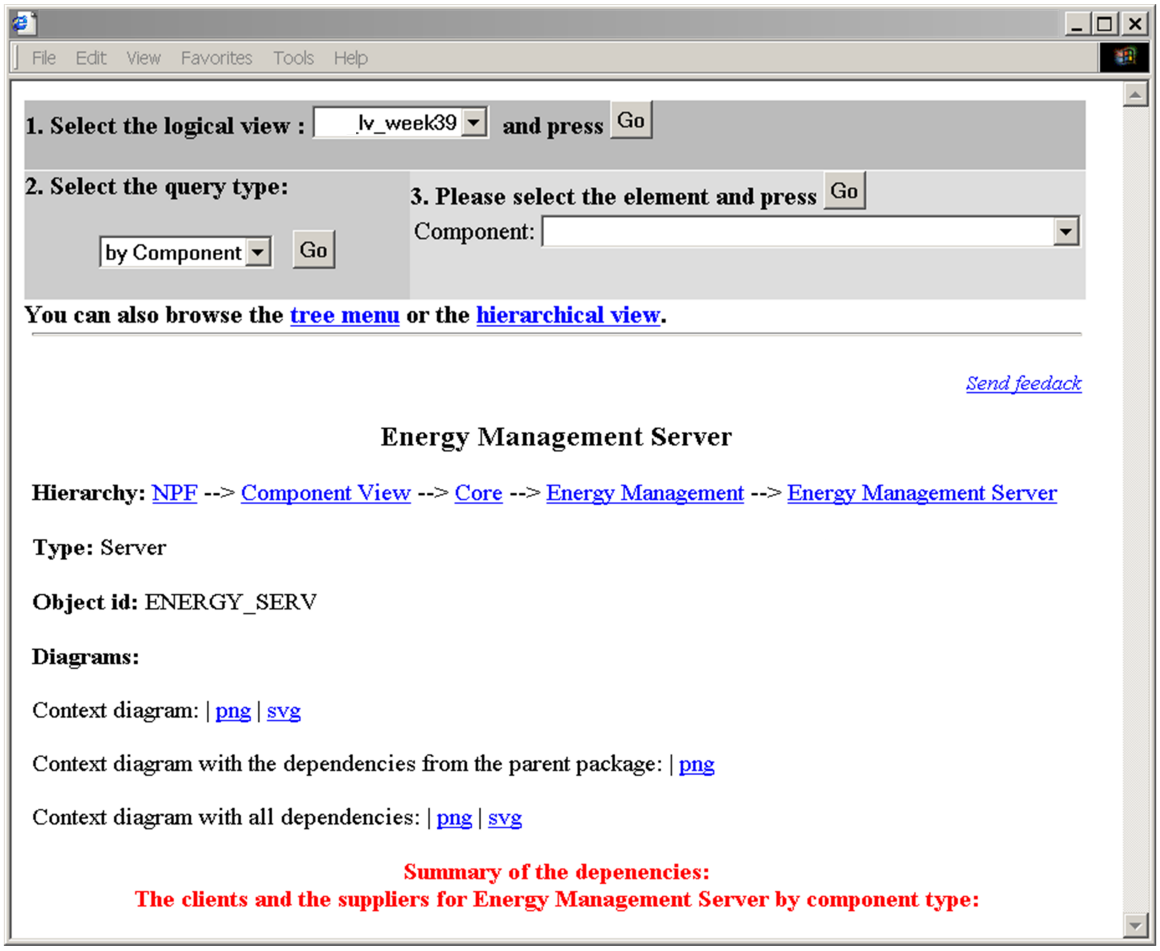


Figure 9.16: Overview of the web interface.

WEB INTERFACE

The architecture of the web interface is described in Section 8.5.4. In this section we focus on its application for NPF. From the user perspective, the web interface consists of five parts: the query section, the component summary, the graphical diagrams, the tree menu and the hierarchical view. The Figure 9.15 shows an overview of the web interface.

The query section allows the user to select the particular build from the database and to query the components. There are four types of queries: by component's name, by type, by package and by matching a particular pattern. The results of the query are shown in the drop-down list where the user can select the element to show in the component summary.

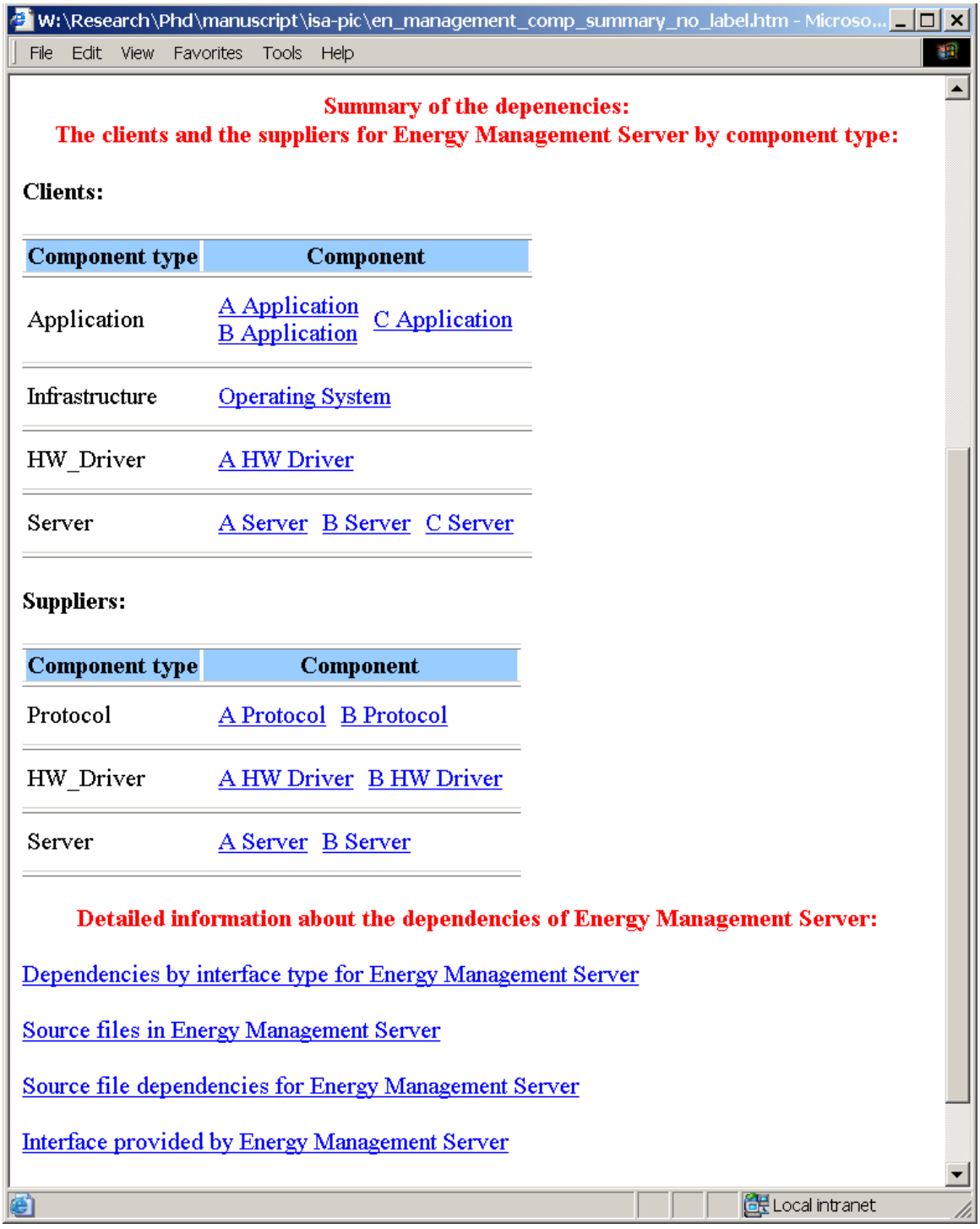


Figure 9.17: The summary of the dependencies.

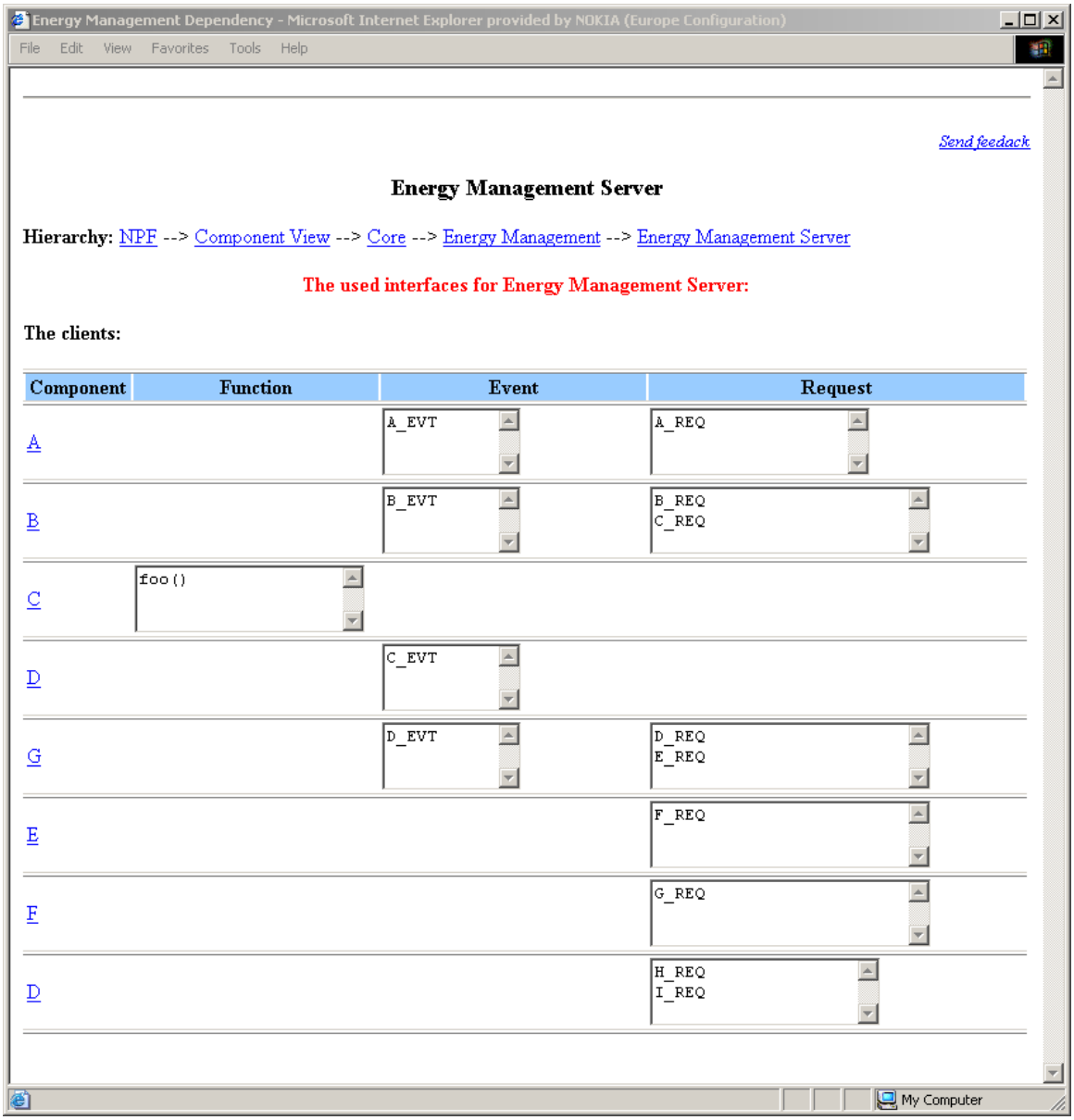


Figure 9.18: The detailed dependency table.

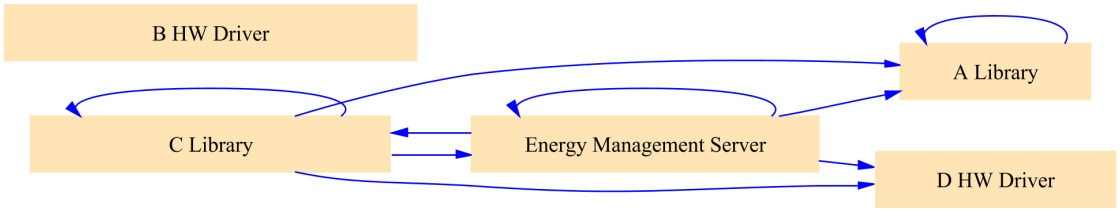


Figure 9.19: The memory management.

The component summary contains a summary of the information about the selected component: location in the hierarchy, the type, other information, a list of context diagrams, a summary of the main dependencies and a list of links to other details about the component. The Figure 9.16 show the summary page for one server and the Figure 9.17 shows the summary of the dependencies. The user can easily look at the clients and suppliers for the selected component. All the elements are hyperlinked to other components' summary page. The table in Figure 9.18 shows the detailed information about the interface usage for the selected component.

From the summary page, the user can automatically generate several diagrams about the context of the selected component. The context diagrams show the external relationships with other components. For example, the graph Figure 9.20 shows the context diagram of one component with respect to the components from the same package. The arcs represent any kind of dependency between the components. The graph in Figure 9.20 shows the entire context diagram for the the same component. The selected component is marked with a circle and there are 46 external components. The graph is rather complicated and is an example of the complexity of large systems like NPF.

The hierarchical view allows the user to interactively browse the component view through the various packages starting from the top-level diagram. The diagram in Figure 9.21 shows the 5 top-level packages of NPF and the top-level dependencies. The user can click on the packages or on the components to navigate through the view. For example, by clicking on the top-level package in Figure 9.21 the user can expand the packages and obtain the diagram in Figure 9.22. The diagram shows the top-level packages, their contents, the intra-package and inter-package dependencies. There are two major packages: *Apps* and *Core*. The user can click on them to obtain more detailed diagrams as shown in Figure 9.23 for *Apps* and in Figure 9.24 for *Core*. The diagram in Figure 9.25 shows the content of one package contained in the *Apps* package.

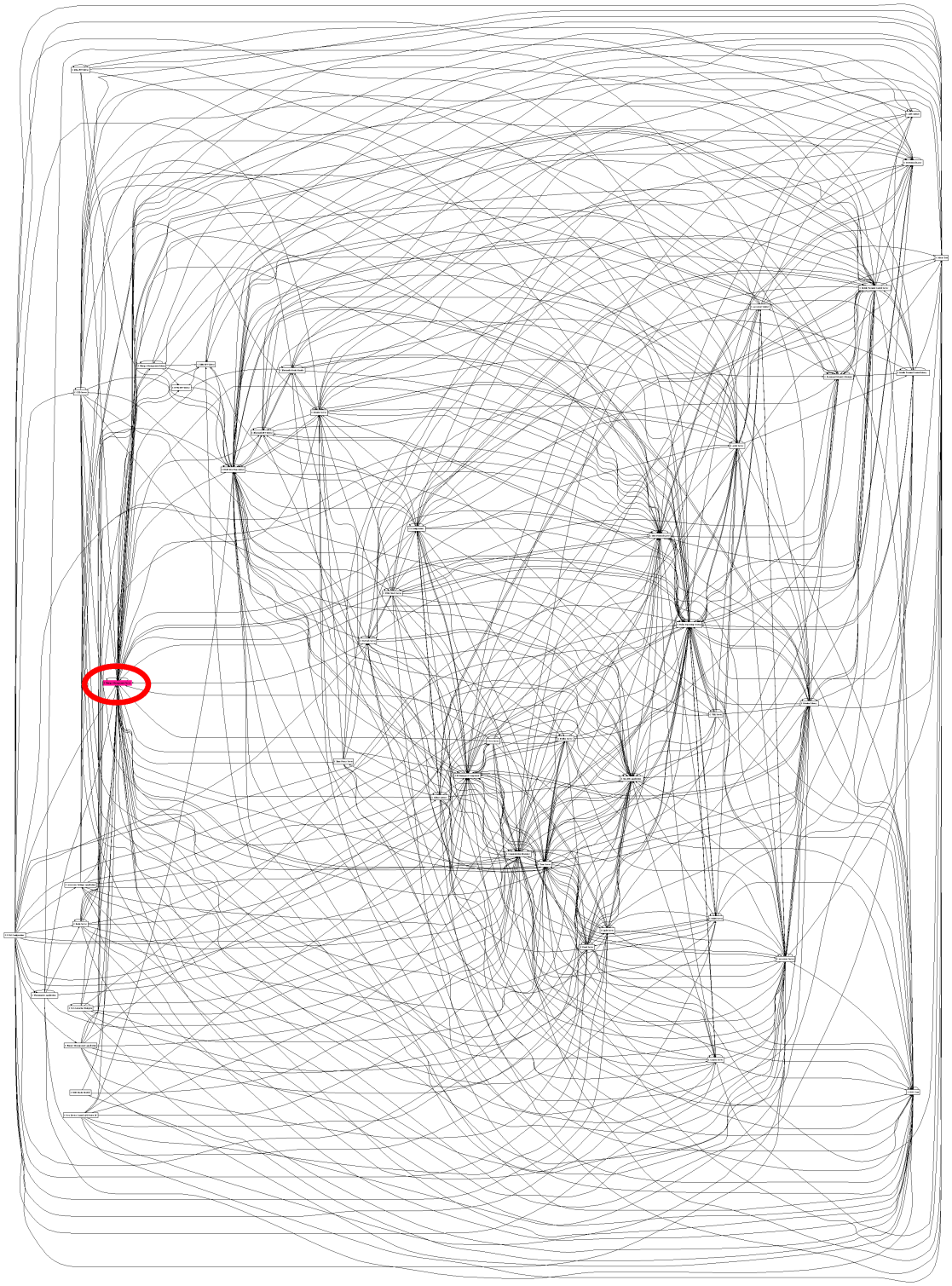


Figure 9.20: The context of memory management.

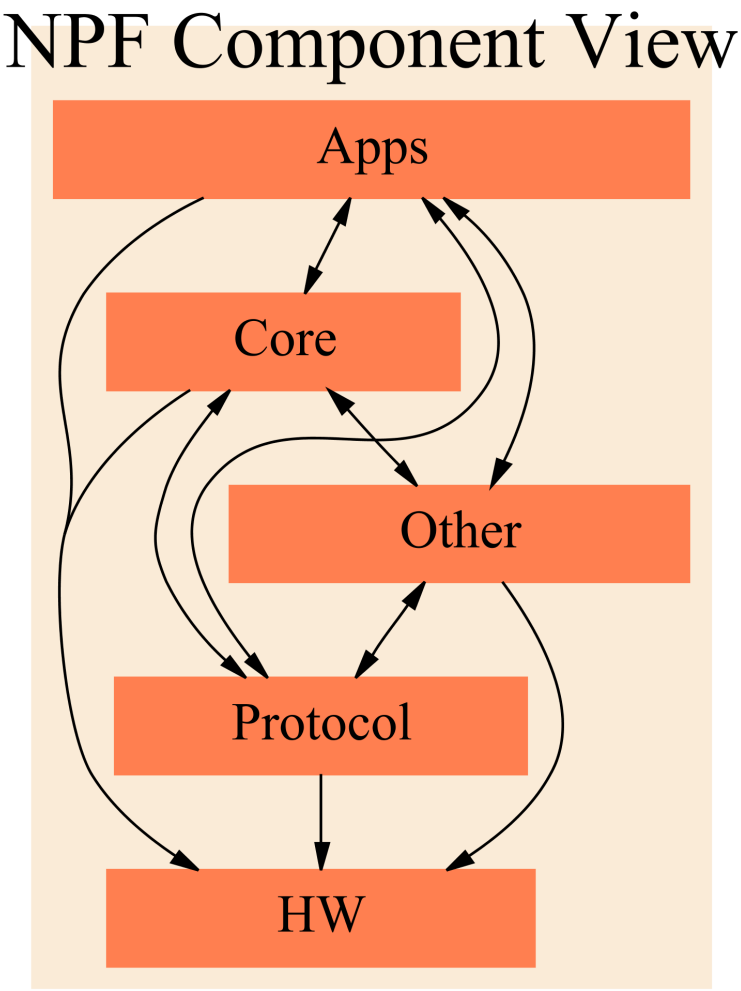


Figure 9.21: The top-level diagram for NPF.

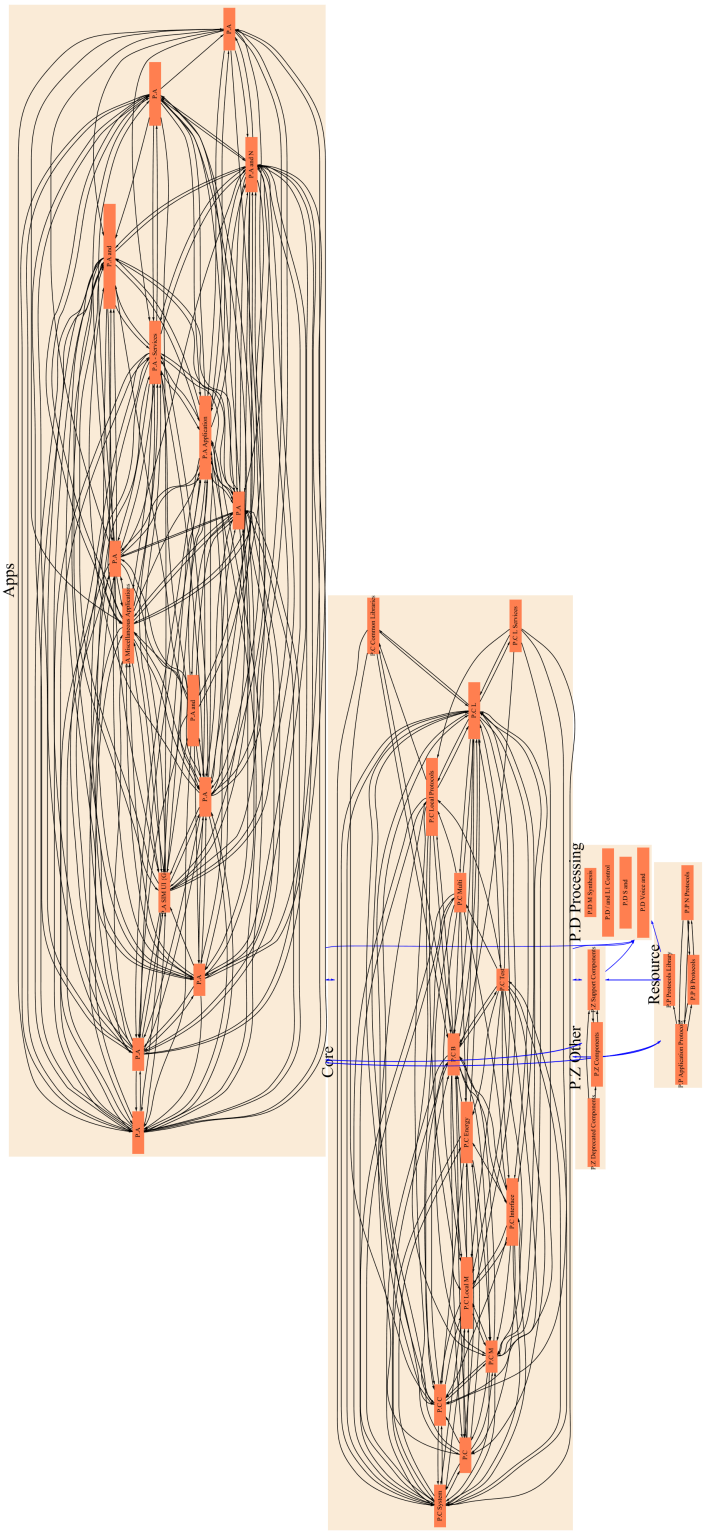


Figure 9.22: The top-level diagram showing the content of the packages.

The stakeholders of NPF appreciated the simplicity of the web interface and the easiness for retrieving information. At the end of the third iteration, they decided to deploy the web interface within organization and to integrate it in the development process of NPF.

UML

The architectural views can be converted to a UML model and visualized with Rational Rose following the indications presented in Section 8.5.3. The result is show in Figure 9.26. Although the architects are not especially interested in this CASE tool, in the future we plan to provide a better support for UML. All the packages and components are listed in the navigation bar on the left. For each package the user can show the context diagram as shown in the figure.

HIERARCHICAL GRAPHS

We can visualize the reconstructed views as hierarchical graphs with RIGI or DOT from GraphViz. The hierarchy of the graphs is defined by the containment relationship of the particular viewpoint. Using RIGI the end-users can navigate the hierarchical graphs and extract the details about the dependencies.

The graph in Figure 9.27 is the development view for NPF showing the top-level dependencies (call and include dependencies) among the root directories directories. With the development view the architects can examine the source code organization. The graph in Figure 9.28 is the geographical view showing the sites and the cross-site dependencies for NPF. Each site consists of several component factories. The cross-site dependencies are lifted from the low-level logical dependencies between the components developed by the sites. With this view, the managers can look at the geographical distribution of the development and analyze the cross-site dependencies (that can cause synchronization and communication problems). The development of NPF is distributed over more than 20 different sites around the world (not all are shown in the figure). The graph in Figure 9.29 is the organization view showing the projects and the inter-project dependencies. Each project, owned by a factory, is responsible for the development of a set of components. With this view the project managers can check the dependencies among the projects and, they can plan correctly the schedule of the projects. The task view is show in Figure 9.30. The view presents the tasks of NPF and the inter-task dependencies (caused by messages exchanged among components). The inter-task dependencies should be minimized as they can cause performance problems due task context switch. This view help the architects in their analysis. The task view allowed us also to validate the task start order for NPF. The graph in Figure 9.31 shows the task view laid out using RIGI 's reverse tree layout. The reverse tree

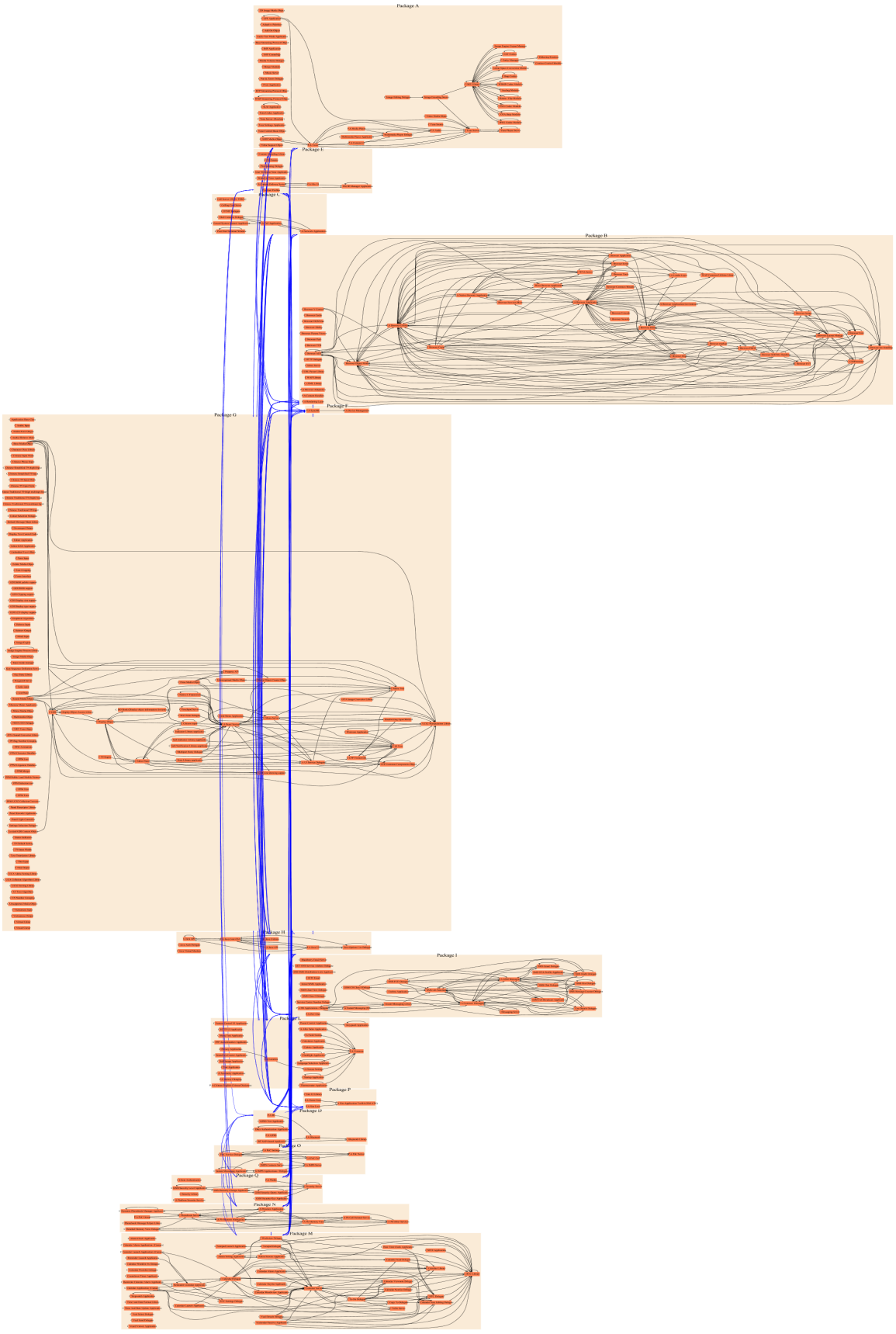


Figure 9.23: The content of the *Apps* package.

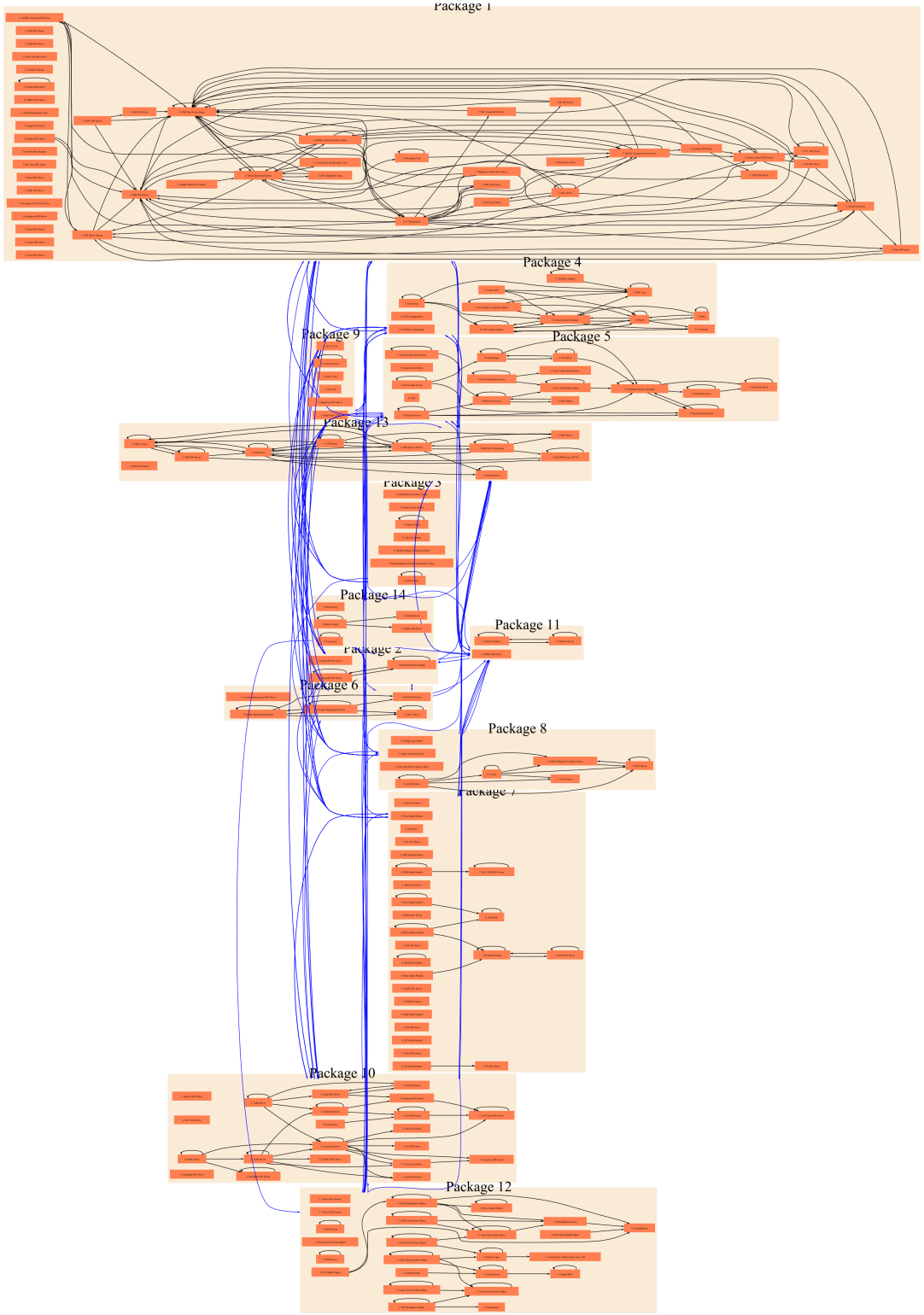


Figure 9.24: The content of the *Core* package.

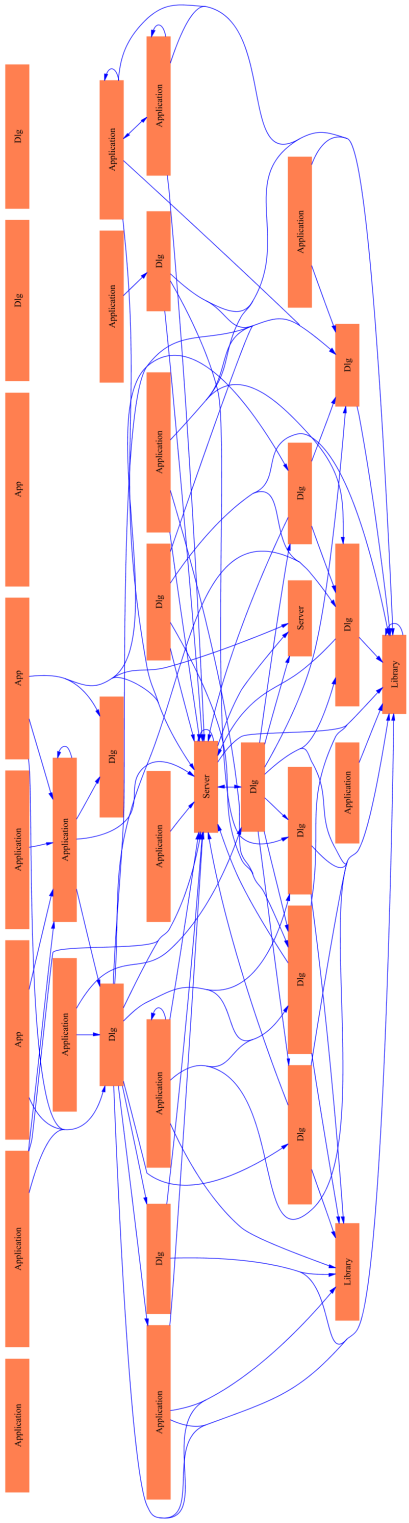


Figure 9.25: The content of one package of NPF.

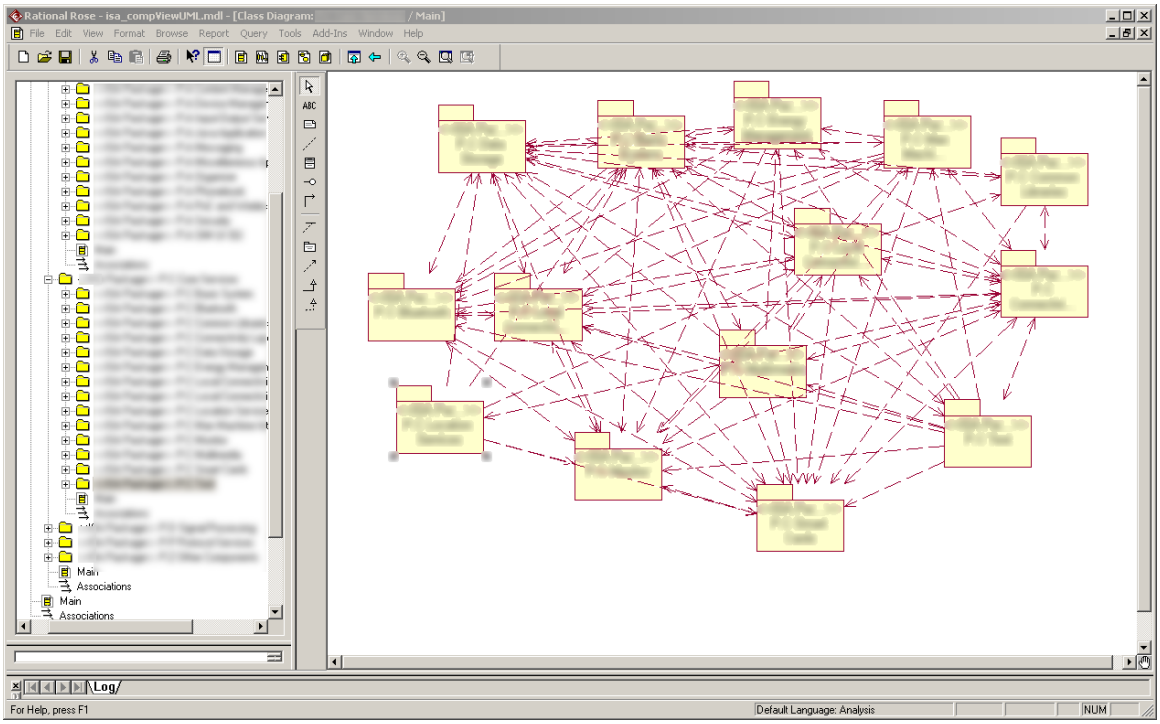


Figure 9.26: The content of one package of NPF.

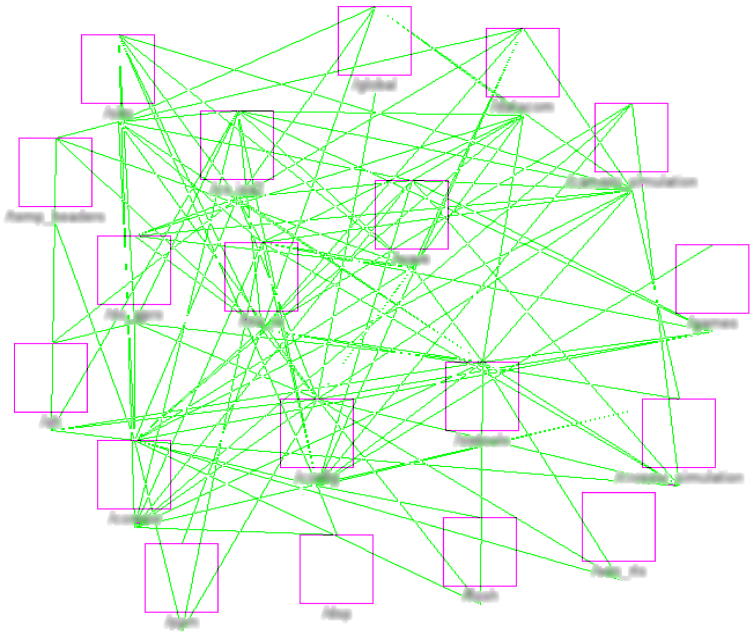


Figure 9.27: The development view for NPF.

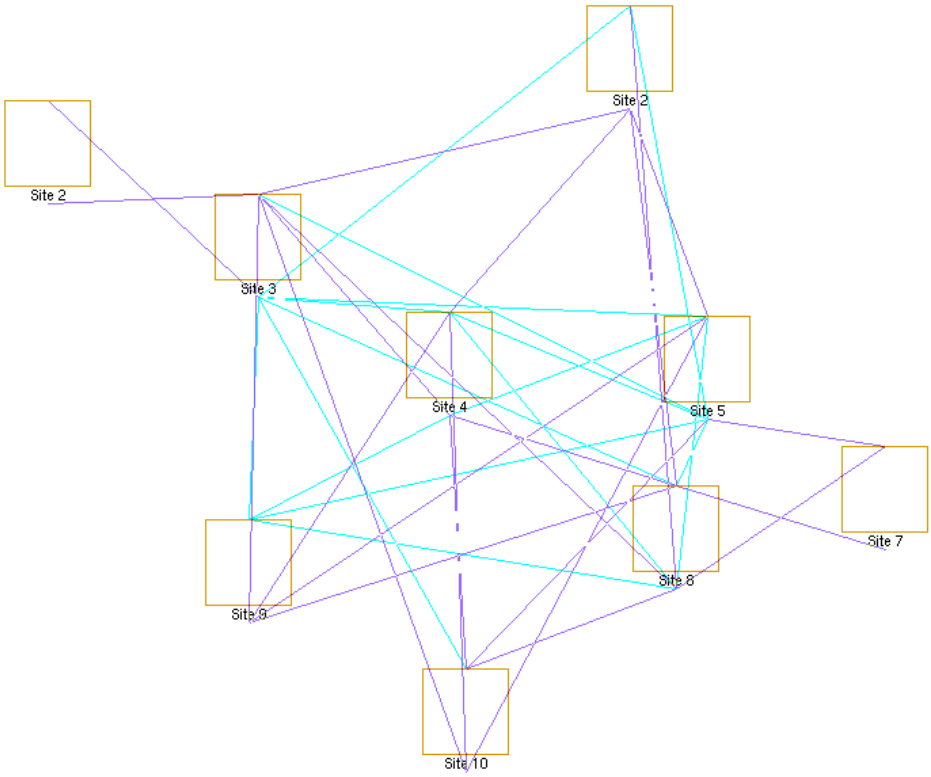


Figure 9.28: The geographical view for NPF.

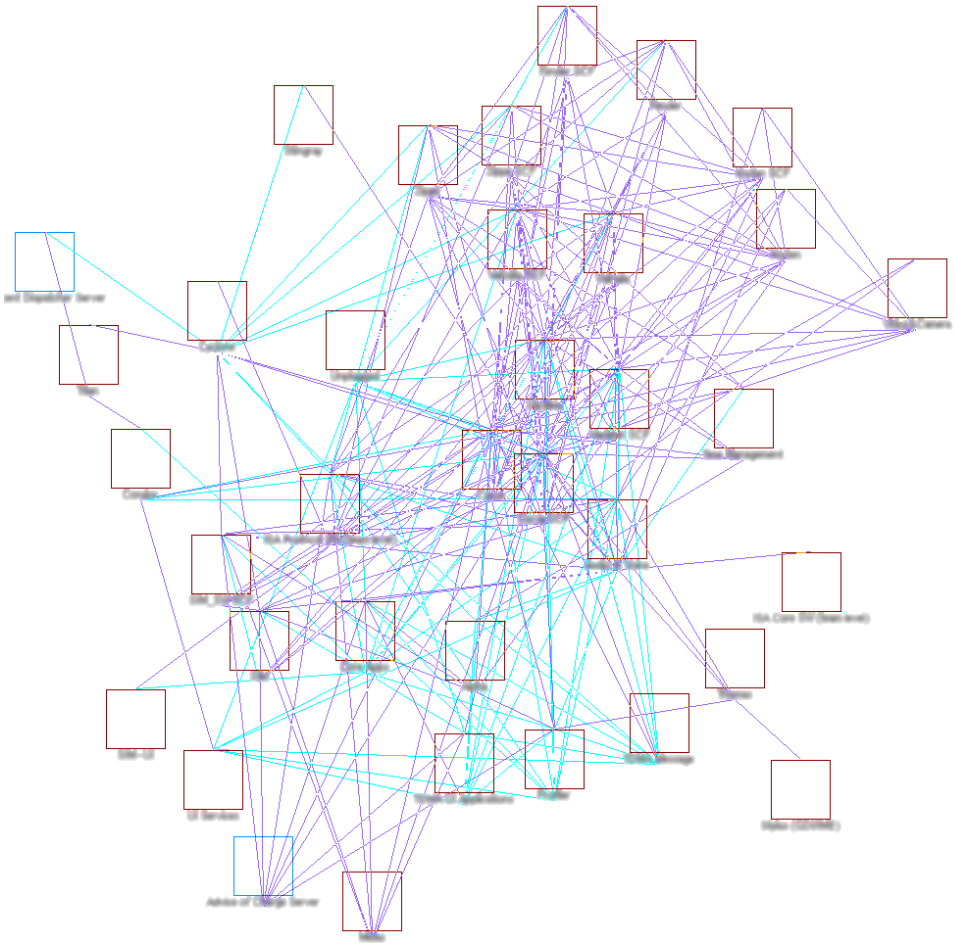


Figure 9.29: The organizational view for NPF.

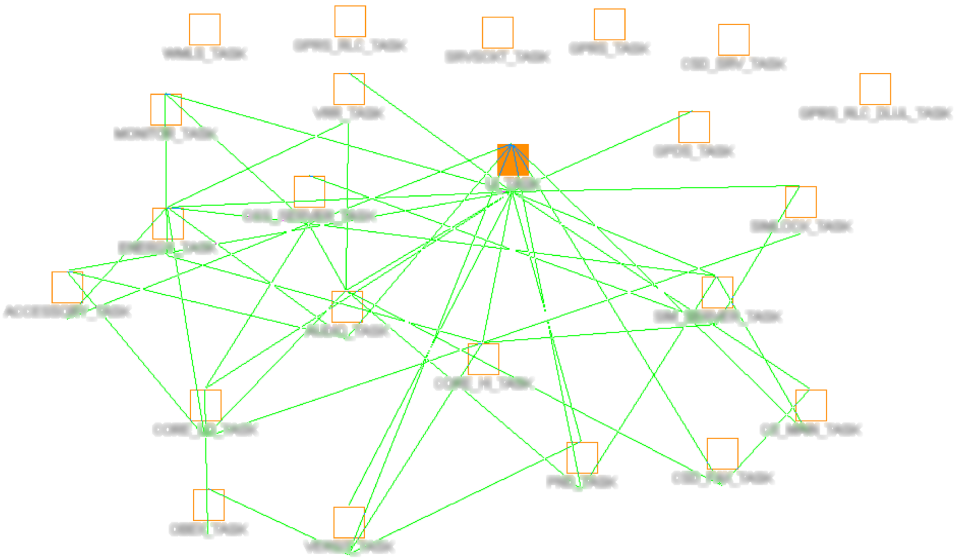


Figure 9.30: The task view for NPF.

layout arranges in a tree layout all the nodes on which a selected node depends. Starting from the tasks with no dependencies we can reversely lay out all the other tasks. The tasks in the lower part of the graph need to be started before the tasks in the higher part. This diagram has been used to validate that the booting sequence is correct.

9.7.6 Architecture Conformance Checking

The architectural style of NPF defines the architectural rules that have to be enforced in the implementation of the platform and of the products. The architects must enforce these rules in order to preserve the architectural integrity of NPF and avoid the architectural decay. Some of the rules are listed below:

- To check that the implementation conforms to the reference architecture. We have to check that only the permitted relationships exist between the entities (e.g. an Application can only use the services of a server through the message and event interfaces).
- To check that the top-level dependencies conform to the indented design of the architects.
- Validate that NPF conforms to the there-layer architectural style.

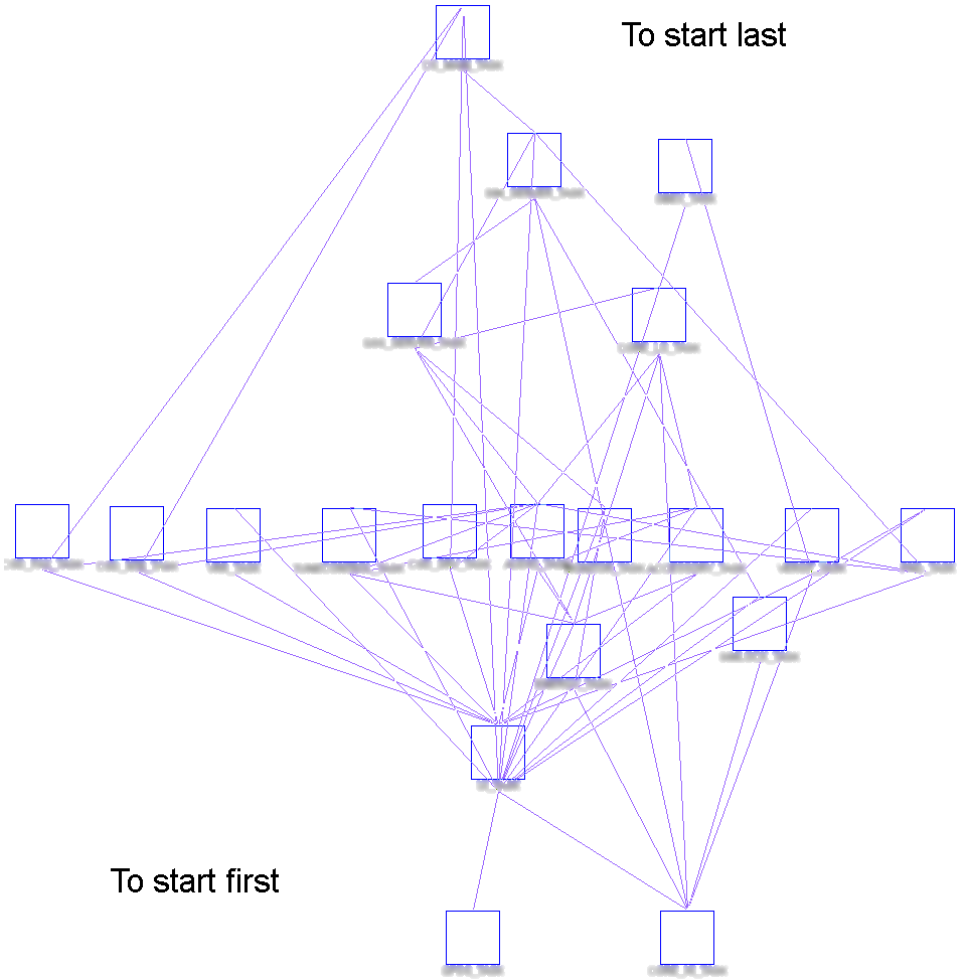


Figure 9.31: The starting order of tasks for NPF.

The result of the conformance checking is a detailed report of the violations that is handed to the architects who can take the proper follow-up actions.

We have defined about 100 different rules for all the architectural concepts of NPF. For each concept, the rules define what other components can be used and what dependencies are allowed. In addition, we have also defined rules for enforcing the layering structure of NPF. We checked the conformance of one build of NPF and the result showed about 3700 illegal dependencies and 650 layer violations. We investigated the dependencies and we made the following observations:

1. Some legal dependencies have been detected as illegal because the the reference architecture of NPF is not complete. This fact will lead to an update of the reference architecture
2. Some components have an inappropriate component type. This has to be fixed in data gathering phase.
3. There are illegal dependencies between components that are implemented by the same team or that implement the same feature or feature set. For example, some components illegally share functions with other components. This is often the result of incomplete refactoring or simply because the components are developed by the same team. These violations should be fixed because they are against the design principles of NPF and hinder reusability.
4. Some illegal dependencies represent short-cuts across the layers and have been introduced for special reasons (like performance or other non-functional requirements). Although against the design principles, they are permitted on limited scale.
5. Illegal function calls are responsible for most of the illegal dependencies. These represent short-cuts, legacy code, global functions. These violations need to be fixed and monitored.

Below we present two cases of conformance checking against two types of rules and the implementation in relational algebra. The work that we present in this section conforms with the work that we have presented in (Riva *et al.* , 2004b) on introducing a UML-based conformance checking method.

CONFORMANCE CHECK AGAINST THE REFERENCE ARCHITECTURE

For each architectural concept we can check if the interfaces are properly used. As an example, we can consider the servers. The reference architecture defines the following

rules:

- A component can access a server through the asynchronous messaging interface.
- A component can subscribe to the events offered by the server.
- A server can invoke the functions of a library, an hardware driver, a protocol and other infrastructure elements.

We can formalize the rules in binary relational algebra. We can calculate the components that violate the first and second rule because they are accessing the servers by function calls:

$$\begin{aligned} servers &= type.\{'\text{Server}'\} \\ illegalClients &= invocation.servers \end{aligned}$$

The *illegalClients* relation contains all the components that are calling the functions of the servers. For a build of NPF we calculated that about 200 components are violating this rule. According to the third rule, we can calculate the illegal suppliers of the servers that are accessed through function calls:

$$\begin{aligned} allowedSuppliers &= type.\{'\text{Library}',' \text{ HW Driver}',' \text{ Protocol}'\} \\ serverInvocation &= invocation|_{dom} servers \\ legalInvocation &= serverInvocation|_{ran} allowedSuppliers \\ illegalInvocation &= serverInvocation - legalInvocation \end{aligned}$$

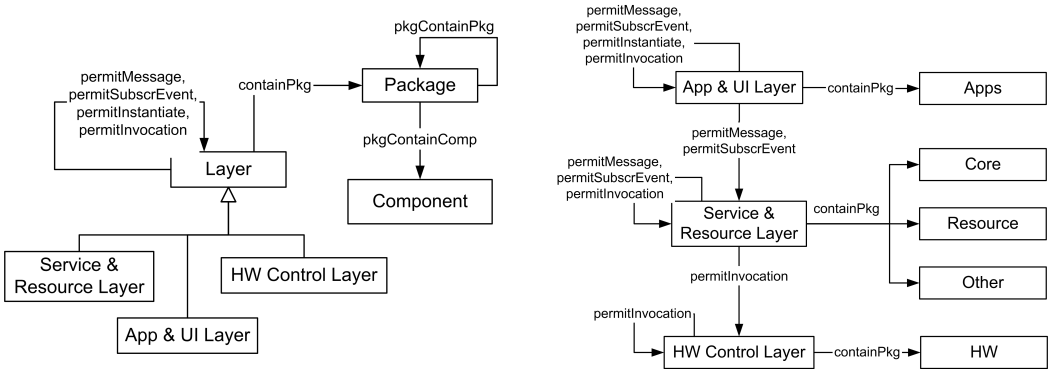
The *illegalInvocation* relation contains the list of illegal invocations. We calculated that there are about 100 illegal dependencies that are caused by 45 components that are supplying their services through functions.

CONFORMANCE CHECKING OF THE LAYERS

At the top-level, NPF is organized in three layers of functionality that are responsible for the UI applications (`App & UI` layer), the global services and system resources (`Service & Resource` layer) and the hardware abstraction layer (`HW Control` layer). Each layer contains several logical packages and there are precise rules what dependencies are allowed between the layers. The architects of NPF need to enforce that these rules are respected for the whole product family.

In conjunction with the architects, we have defined the layered view for NPF. A simplified version is shown in Figure 9.32 together with the layered viewpoint. The layered

view shows that the `App & UI` layer is allowed to use components in the `Service & Resource` layer only by sending asynchronous messages or by subscribing to the events. The `Service & Resource` layer can use the components of the `HW Control` layer only by invoking their functions. The components have also constraints within their own layers.



The layered viewpoint. The layered view.

Figure 9.32: The viewpoint and the view for the layering conformance checking.

We have formalized the process of conformance checking with the binary relational algebra. The check is fully automated and it can applied every time a new build is available. The output is a detailed report of the components and dependencies that are violating the layering. The report is regularly handed to the architects who can take further actions. Below we present a simplification of the conformance check.

Each layer contain one or more logical packages. We define the containment relation for the layers in the following way:

$$
\begin{aligned}
containPkg \; = \{ \; & (\text{'App \& UI Layer'},\text{' Apps'}), \\
& (\text{'Service \& Resource Layer'},\text{' Core'}), \\
& (\text{'Service \& Resource Layer'},\text{' Resource'}), \\
& (\text{'Service \& Resource Layer'},\text{' Other'}), \\
& (\text{'HW Control Layer'},\text{' HW Driver'})\}
\end{aligned}
$$

The overall containment relation for the layered viewpoint is defined as:

$$contain = containPkg + compContainPkg$$

where the *compContainPkg* is the containment relation for the component view as defined in Section 9.7.4.

The allowed relations for the layered view are defined as:

$$
\begin{aligned}
permitMessage &= \{ \ (\text{'App \& UI Layer'},\text{'Service \& Resource Layer'}),\\
&\quad (\text{'App \& UI Layer'},\text{'App \& UI Layer'}),\\
&\quad (\text{'Service \& Resource Layer'},\text{'Service \& Resource Layer'})\}\\
permitSubscrEvent &= \{ \ (\text{'App \& UI Layer'},\text{'Service \& Resource Layer'}),\\
&\quad (\text{'App \& UI Layer'},\text{'App \& UI Layer'}),\\
&\quad (\text{'Service \& Resource Layer'},\text{'Service \& Resource Layer'})\}\\
permitInstantiate &= \{ \ (\text{'App \& UI Layer'},\text{'App \& UI Layer'})\}\\
permitInvocation &= \{ \ (\text{'App \& UI Layer'},\text{'Service \& Resource Layer'}),\\
&\quad (\text{'Service \& Resource Layer'},\text{'HW Control Layer'}),\\
&\quad (\text{'App \& UI Layer'},\text{'App \& UI Layer'}),\\
&\quad (\text{'Service \& Resource Layer'},\text{'Service \& Resource Layer'}),\\
&\quad (\text{'HW Control Layer'},\text{'HW Control Layer'})\}
\end{aligned}
$$

We can calculate the violations among layers by lifting the component level dependencies (*compDependency)* with the *contain* relation to the level of the layers. The *compDependency* is the dependency relation for the component view as defined in Section 9.7.4. In relational algebra we have:

$$
\begin{aligned}
permittedDependecy &= permitMessage + permitsubscrEvent + permitInstatiate\\
&\quad + permitInvocation\\
layerDependency &= compDependency \uparrow contain\\
layerViolation &= permittedDependency - layerDependency
\end{aligned}
$$

The *permittedViolation* contains the list of illegal dependencies among the layers. We can calculate the component level dependencies that are causing the layer violations by lowering the violations:

$$compViolation = (layerViolation \downarrow contain) \cap compDependency$$

We can refine the information about the violations according to the dependency type:

$$
\begin{aligned}
allowedMessage &= (permitMessage \downarrow contain) \cap message\\
forbiddenMessage &= message - allowedMessage
\end{aligned}
$$

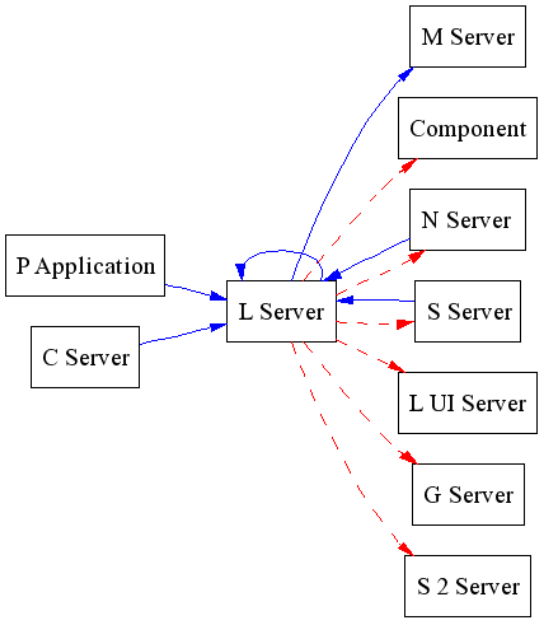


Figure 9.33: The conformance diagram of one server.

$$allowedSubscrEvent = (permitSubscrEvent \downarrow contain) \cap subscrEvent$$
$$forbiddenSubscrEvent = message - allowedSubscrEvent$$

$$allowedInstantiate = (permitInstantiate \downarrow contain) \cap instantiate$$
$$forbiddenInstantiate = message - allowedInstantiate$$

$$allowedInvocation = (permitInvocation \downarrow contain) \cap invocation$$
$$forbiddenInvocation = message - allowedInvocation$$

The forbidden relations contain the list of dependencies at the level of components.

For a given component, we can create a diagram that shows the illegal dependencies. The diagram in Figure 9.33 shows the legal (plain arcs) and illegal (dashed arcs) dependencies for one server. The diagram in Figure 9.34 shows the illegal dependencies for the context diagram of one server.

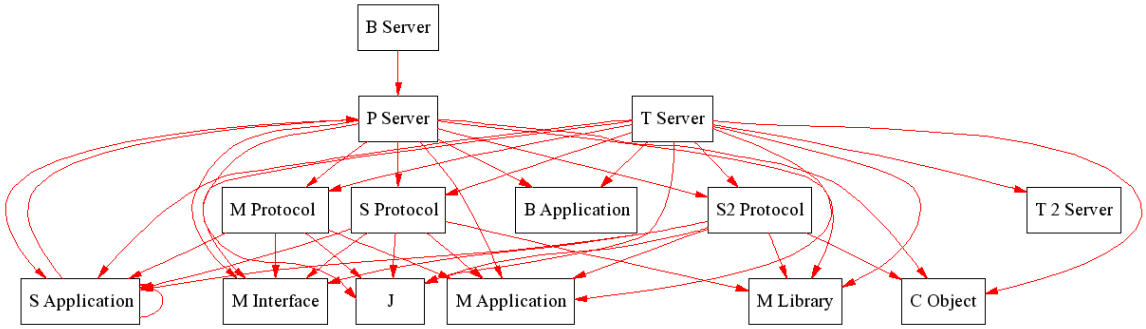


Figure 9.34: The conformance diagram of one server.

9.8 The Dynamic Analysis

We have conducted a dynamic analysis of NPF for calculating the feature view of selected features. In this section, we present the an overview of the approach that we have followed.

9.8.1 Problem Definition

The goal of the dynamic analysis was to recover the feature views for a selected set of features from the implementation of the products. There were two major motivations: to re-document the implementation of key features and to compare the implementation the same features across various products. Understanding the feature implementation is an important activity in a product family. The features offered by the platform represent the assets that are reused among several products. A clear comprehension of these assets is necessary when deriving new products by combining existing and new features.

9.8.2 Concept Determination

The architects were mainly interested in the run-time behavior at the same level of abstraction as the component and task viewpoints defined in the third iteration (Section 9.7.2). Our task was to recover the run-time interactions among the elements of the component and task viewpoints, and present them in the feature views.

The target viewpoints are the feature viewpoints of the component and task viewpoint. The feature viewpoints also include the time line. The source viewpoint is represented

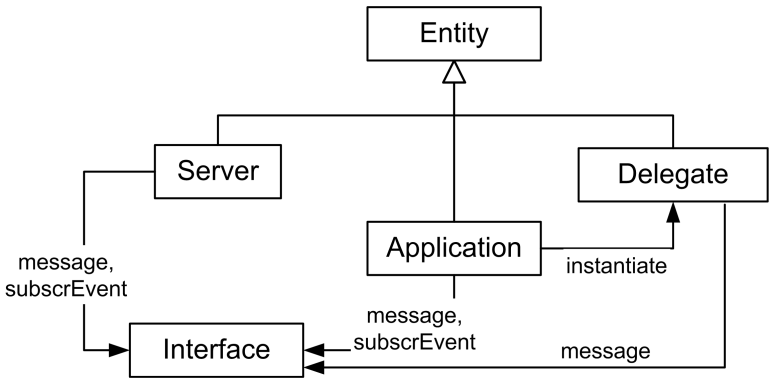


Figure 9.35: The elements of the feature viewpoint.

in Figure 9.35. It includes the architecturally relevant interactions for creating the feature views.

9.8.3 DATA GATHERING

In order to gather the data for the feature views we instrumented NPF and execute several scenarios. The instrumentation was done accordingly to the architectural concepts from the component and target viewpoints. We instrumented the communication bus for tracing the asynchronous messages (*message*) and the event registrations (*subscrEvent*). For tracing the delegate instantiation, we instrumented the initialization functions of the delegates.

We defined a set of scenarios that cover the features to analyze. We tried to create the shortest scenario possible for each feature and we tried to avoid the feature interaction.

We executed the scenarios on the instrumented system and we collected the long trace logs. We had the possibility of executing the scenarios either in a simulator or on the target hardware. From the trace logs we removed the startup and shutdown traces. Then, we converted them in order to conform with the source view. We stored the result in a sorted RSF-like file containing tuples in the format: $< timestamp >< relation >< source >< destination >$

9.8.4 KNOWLEDGE INFERENCE

The trace logs contained thousands of traces and hundreds of participants. For this reason it was important to compress the traces with abstraction. We applied the horizontal abstraction provided by the tool HAVA (Section 8.5.2). We grouped the participants according to the relation *compContain* and *taskContain* respectively for the component view and the task view. The result shows the feature views grouped according to the package structure and according to the task structure. This operation allowed to significantly reduce the size of the traces.

9.8.5 PRESENTATION

The feature views are presented with the HAVA extension of NIMETA . NIMETA allow us to visualize the static and dynamic views at the same time. The static views are shown as graphs in RIGI 's windows, while the dynamic views are shown in HAVA 's sequence diagrams. HAVA supports the possibility of synchronizing the abstractions in the static and dynamic views (importing the abstractions from rigi to hava or viceversa), applying the synchronized collapse/expand commands and slicing the static view based on the content of the dynamic view.

We have studied the feature "Call release" that is executed when the user terminates a phone call. We have created a simple scenario where the user initiates a call and terminates it immediately. The top-level static and sequence diagrams are show in Figure 9.36. The static diagram (shown as a RIGI 's graph) is based on the component view calculated in the third iteration (see Figure 9.21). The message diagram is based on the dynamic analysis and the participants have been grouped according to the static view. The messages exchanged within the collapsed participants are not shown and this allows us to examine the feature implementation at an high-level of abstraction. In the sequence diagram we can note the messages "call_create_req" and "call_terminated_event" that are exchanged between one component in the `Apps` package and one server in the `Resource` package. They represent the effect of the user's actions for initiating and terminating the call. We can note that the main benefit of the HAVA approach is to visualize a trace log consisting of hundreds messages in a very compact format. This is the main advantage of the HAVA approach.

The analysis can continue by expanding the top-level packages and navigating the details of the sequence diagram. We expand the package `Apps` and the result is shown in Figure 9.37 and Figure 9.38 for the static and dynamic views respectively. The sequence diagram shows another part of the implementation of the "call release" feature.

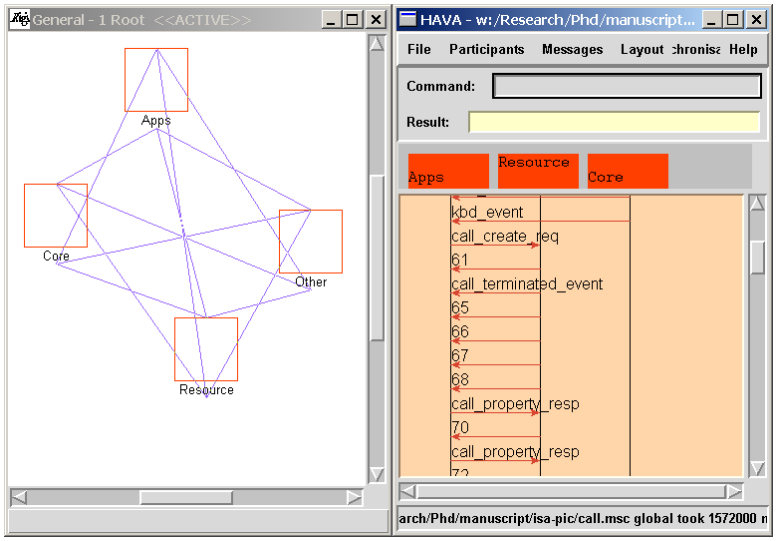


Figure 9.36: The top-level feature view for the ”call release” feature (static diagram and sequence diagram).

We can also slice the static view with the participants in the sequence diagram. The Figure 9.39 shows an example of the slice. In the sequence diagram we have focused on a particular interaction sequence with only three participants. The static view has been sliced in order to show the static dependencies among the same participants. This permits us to conduct a consistent analysis of the dynamic traces and the feature implementation.

9.9 Conclusions and Lessons Learned

The third iteration delivered a robust reconstruction process that satisfies most of the requirements set by the stakeholders. They decided to integrate the reconstruction process with the development process of NPF and we are currently conducting the technology transfer. One architect has been nominated as a reconstructor and is responsible for the maintenance of the reconstruction process. The target is to make biweekly releases of the models for the whole NPF family (including the platform and the products). These are the main tasks for the reconstructor:

- To maintain the reconstruction process up to date with the reference architecture of NPF (i.e. introducing new architectural concepts, updating the conformance rules)
- To maintain the mapping tables (i.e. introducing the mapping for new components)

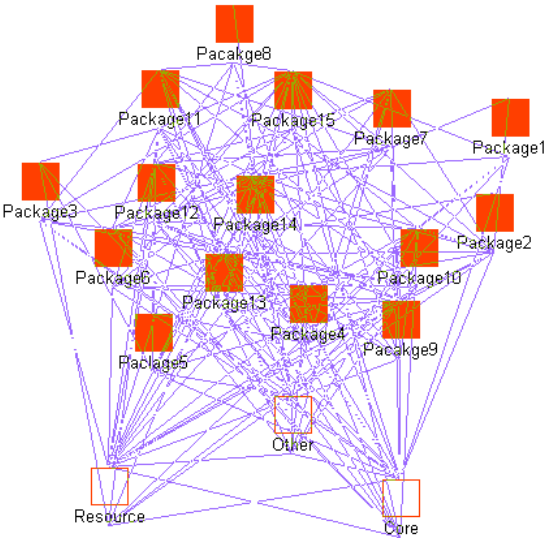


Figure 9.37: The expanded component view.

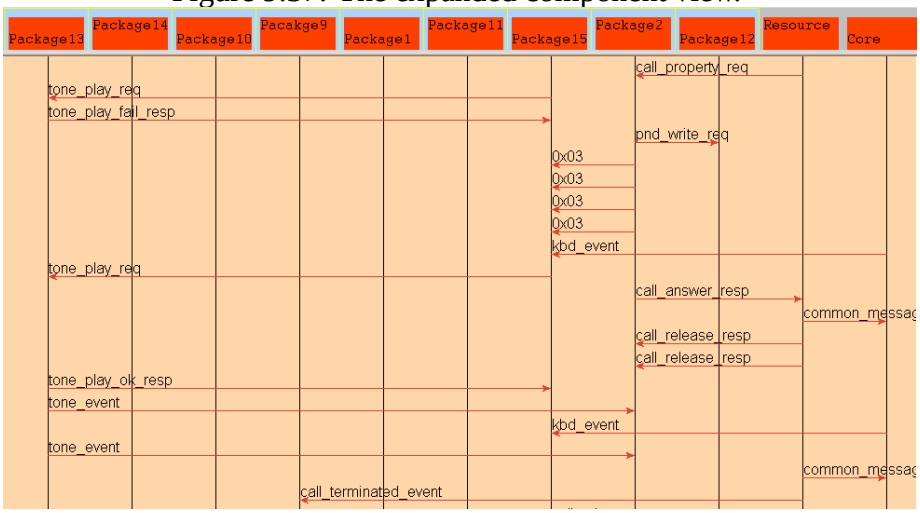
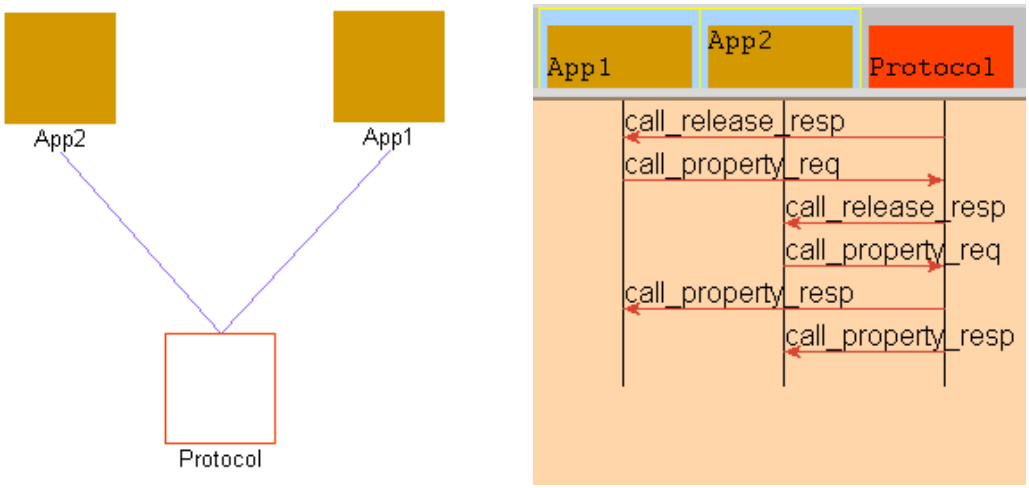


Figure 9.38: The expanded sequence diagram.



The static view. The dynamic view.

Figure 9.39: The details of the call release feature between two applications and one server.

- To execute the reconstruction scripts on the build and to publish the models on the web interface
- To validate the architectural models
- To report the architectural violations to the architects

The overall approach results in a very architecture centric development process that can guarantee the proper architectural governance to NPF and avoid the architecture decay. In the long term, we expect that the architectural models will contain precise and reliable architectural information and the conformance checking will preserve the architectural integrity. The reconstruction process has been implemented as a pipeline of tools (as described in Section 8.2 that can be executed automatically on a given build with minor human interaction. In this way the reconstructor can easily generate the new architectural models as the build become available. In the future, we are planning to integrate the reconstruction scripts with the build process in order to create the architectural documentation every time the system is built. At the end of the technology transfer, we can consider NPF at the fourth level of maturity.

We summarize below the main lessons that we have learned during the case study:

- The development of the reconstruction process has to be properly supported by the organization where the minimum requirement is the awareness of the architectural problems. The reconstruction process is efficient if it is designed according to the

domain of the system and with clear requirements from the stakeholders. Hence, the requirement that for starting a reconstruction process the organization or the team has to be at least at the first maturity level (see Section 7.4).

- At least three iterations are required before integrating the reconstruction process within the organization. The first iteration is focused on the detection of the requirements, the second iteration is focused on delivering the initial models and getting the feedback from the stakeholders and, the third iteration is focused on creating a robust reconstruction process that can be integrated in the development process.

- Run-time dependencies are difficult to trace with the static analysis. This may lead to systems that are difficult to evolve. If they cannot be avoided, they should be properly designed in a way that they are easy to recover with static or dynamic analysis. Not only the systems should be designed for change, but they should also be designed for the reconstruction.

- The visualization of the architectural diagrams is often overwhelming and the textual representation is preferable.

- The reconstruction process is currently not supported by commercial CASE tools. The tool environment that we used for NPF is mainly based on loosely integrated academic or ad-hoc tools.

- Creating and validating the mapping tables took (especially the *compContainFile* took a considerable amount of time. Their maintenance is conducted manually by the reconstructor. An automatic mechanism should be developed.

- NPF is a product family and several components can be considered variants of basic ones. Components can also be configured according to the products were they are included. We did not address the problem of variability explicitly, but it is our goal for the future work.

- Naming conventions were not appropriate at the beginning but they improved with time. Naming conventions are very important as they facilitate the automatic reconstruction.

- The architecture views have been used by the architects for conducting various architecture assessments of NPF and the results have been used to improve the overall quality of the platform.

CHAPTER 10

CASE STUDY 2: NMP PLATFORM

I have not failed 700 times. I have not failed once.
I have succeeded in proving that those 700 ways will not work.
When I have eliminated the ways that will not work,
I will find the way that will work.

- Thomas Edison

This chapter describes the case study that we have conducted to support the migration of a monolithic product towards a platform for a product family. We introduce the domain, the motivations and the details of the three iterations that we have carried out.

10.1 INTRODUCTION

The second case study was conducted with a department within Nokia that was in charge of developing a new platform (Nokia New Platform, NNP) for a new generation of telecommunication products. The decision to create a new platform was mainly driven by the trends in the mobile terminal market The original intention was to build the new platform on top of a third-party operating system and to reuse the implementation from other Nokia products for certain telecommunication functionality.

The department adopted the evolutionary approach that we have discussed in Section 2.4.2: developing a single product with the platform design in mind and, then, creating the new platform out of the existing product. The approach consists of an initial one-shot development phase where the implementation is made according to the desired architectural targets. While there is no guarantee that the implementation will follow the architectural targets, at

the end of the one-shot phase the software reaches the release quality and the first product is released. Most of the time is spent on struggling with the implementation details rather than with the architectural quality. The second phase concerns with the architecture evolution and the platform development. The targets are met with frequent release-quality where new features are slowly introduced and carefully assessed. More effort can be put in improving the quality of the architecture and checking the conformance between the real implementation against the architecture.

Over the past four years, our group consulted the NNP department and helped the architects to recover architectural models from the implementation. The case study presents the three most significative experiences that we have collected during this time. The three experiences represent three different moments of the evolutions towards new new platform.

10.2 OVERVIEW OF THE SYSTEM

The NNP platform consists of a *specialized operating system*, a *graphical user interface*, an *application framework* and the support for a set of *telecommunication services*. The various parts have different origins and have been integrated uniquely in the NNP platform. The OS is a third-party component that has been designed for an efficient use of memory resources and battery power, and it provides the basic services of an operating system. The graphical user interface is developed by Nokia and it includes a graphic library and a comprehensive set of UI elements. The application framework is shipped with the OS and modified by Nokia. It allows the development of native and third-party applications. The telecommunication services are integrated from other Nokia's products. There are strict architecturally significant requirements that have to be considered by the architects: startup/shutdown times, real-time requirements (mainly for the telecommunication protocols), memory size, power saving measures and the user interface style.

The NNP platform is partitioned into several logical subsystems that are developed by Nokia or external subcontractors. Each subsystem consists of closely coupled and coherent components. The integration process requires to guarantee that all the parts work together and that they conform to NNF's architectural style. The platform runs on proprietary hardware and is developed in C/C++. The programmers follow precise coding rules. The size of the whole platform exceeds the six millions lines of code.

Iteration	Target viewpoints	Data gathering	Knowledge inference	Presentation
1^{st}	Component view	SourceNavigator and regular expressions	SQL	hierarchical graphs with RIGI
2^{nd}	Component view	SourceNavigator regular expressions	Prolog according to reference architecture	hierarchical graphs with `Rigi`, UML diagrams in Rational Rose
3^{rd}	Component view	SourceNavigator	relational algebra mapping tables	hierarchical graphs, UML diagrams in Rational Rose,

Table 10.1: The summary of the iterations for the second case study.

10.3 SUMMARY OF THE ITERATIONS

Table 10.1 shows the summary of the three iterations that we have conducted. The first iteration was mainly intended for testing the reconstruction process. The goal of the second iteration was to support the creation of the reference architecture. The third iteration was mainly focused on consolidating the reconstruction process in a proper way.

In the third iteration, the build reached the approximative size of 6 MLOC. The source model contained approximately 28,000 files, 27,000 classes, 231,000 methods, 40,000 functions, 23,000 inheritances, 581,000 invocations and 313,000 accesses.

10.4 THE FIRST ITERATION

The first iteration was executed when the development of the first product based on NNP technology was almost ended. We analyzed the implementation of various parts of the product in order to assess the possibility of including them in the NNP platform. Our goal was also to get familiar with the system.

10.4.1 PROBLEM DEFINITION

The overall goal is to define the new platform based on the implementation of an existing product. The architects need to define what components to include, the required interfaces,

the offered APIs, the points of extensibility and the owners of the different elements. The diagram in Figure 10.1 shows the general architecture of the product at the time we started the iteration. The NNP platform already includes large parts of the original product. NNP depends on the hardware layer and on the Nokia Old Platform (NOP) for certain telecommunication services. NNP also includes a part of the original graphic library. The NNP GUI Library package contains the new GUI library that needs to be factored out from the product and included in the platform. The focus of our analysis is on three gray packages in the diagram. The architects need to analyze the dependencies of the NNP GUI Library package and the Nokia Old Platform.

The STD Graphic Library is the outcome of a long evolution. The library evolved in the direction of satisfying UI products with a wide range of different requirements. The NNP GUI Library represents a new graphic library for a new generation of products that has been partly developed using the STD Graphc Library. The assessment aims at understanding the dependencies between the new library and the STD library. Parts of the STD Library are not necessary and will be removed, and the rest will be merged with the NNP graphic library.

The Nokia Old Platform contains legacy code that have been reused from previous products mainly concerning telecommunication protocols. Although NPP depends on NOP, the intention of the architects to define clear interfaces between NPP and NOP because in the future NOP may be replaced or NPP may run on a different telecommunication hardware. Therefore, the architects concentrated in NOP the device dependent implementation of the platform.

The architects also provided us with a large collection of design documents. The product was surprisingly well documented. Each subsystem had its own design document describing the its context, the design decisions and its internal design. The architects also had their own reverse engineering tools for analyzing the static dependencies between the binary files and creating certain design models. The need for architecture reconstruction originated from the fact that an understanding of the overall picture was missing. Although the single parts were well-documented, it was not clear how they all fit together in the product. There was no guaranteed that the borders and the dependencies shown in Figure 10.1 matched the reality. A high-level design document existed, but it was not reliable. Understanding the high-level design was one requirement for the reconstruction activity.

10.4.2 Concept Determination

We organized a one week workshop with the experts and the customers of the reconstruction project. In the first day we had a plenary meeting with all the participants in order to

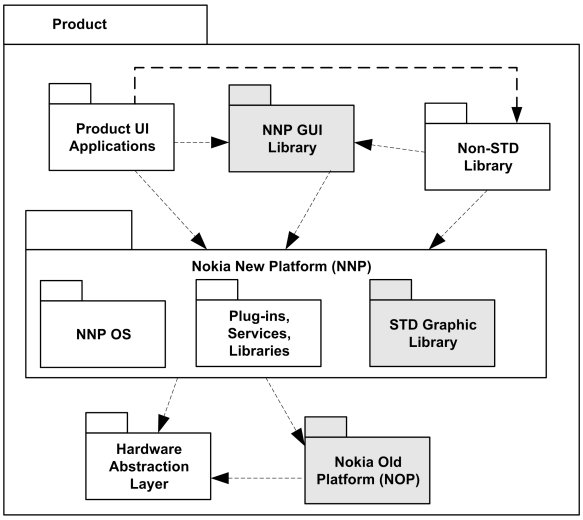


Figure 10.1: General architecture of the product based on the NNF platform.

determine the requirements and the architectural concepts. The agenda consisted of a general overview of the design goals of the future NPP platform, an overview of the existing product, a discussion about the goals of the reconstruction and a detailed discussion about the architectural concepts. We spent the other days analyzing the source code manually and with the reverse engineering tools. We discussed the problems and the initial results directly with the experts. In this way we received an immediate feedback on the results and we precisely focus the direction of the reconstruction work.

THE ARCHITECTURAL CONCEPTS

In order to start the discussion on the architectural concepts, we asked the experts to briefly describe the implementation of a typical feature of the product. The diagram in Figure 10.2 shows the original message sequence chart that the experts drew during the workshop . The MSC shows the participants and the interactions in the implementation of a typical feature. Below we list the architectural concepts that we identified [1]:

Application Applications are the building blocks for implementing the top-level functionality and the UI.

Engine An engine provides a reusable set of services. It consists of a client interface and an internal server.

[1]For simplicity and for preserving the confidentiality, we have changed the labels and we omit the details.

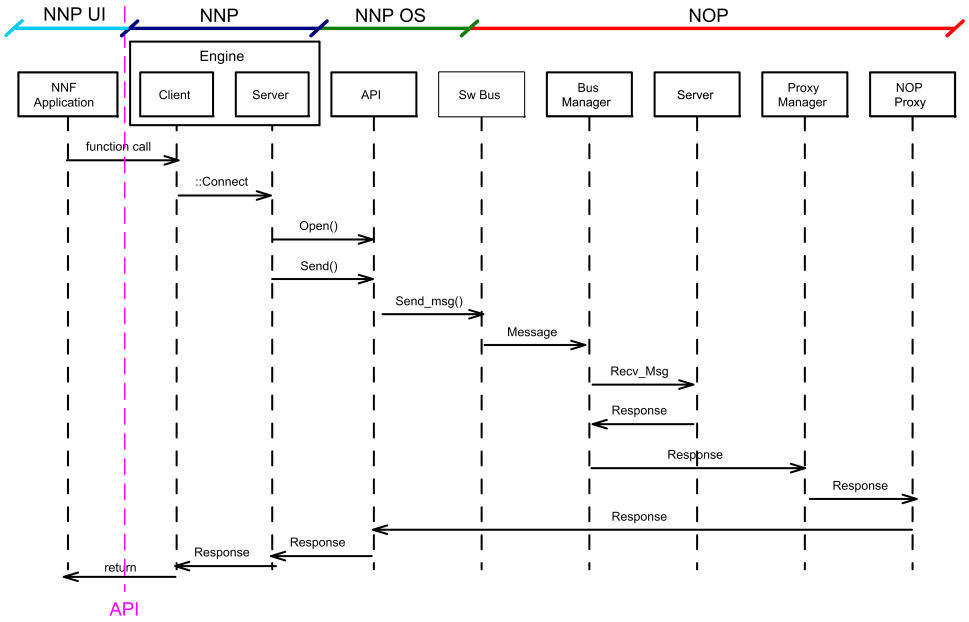


Figure 10.2: MSC drawn during the workshop with the experts.

OS API The OS provides its services through a well-defined API that can be accessed by the applications and engines.

SW BUS There is an internal software bus that connects the NNP OS with the NOP. The communication happens via asynchronous messages.

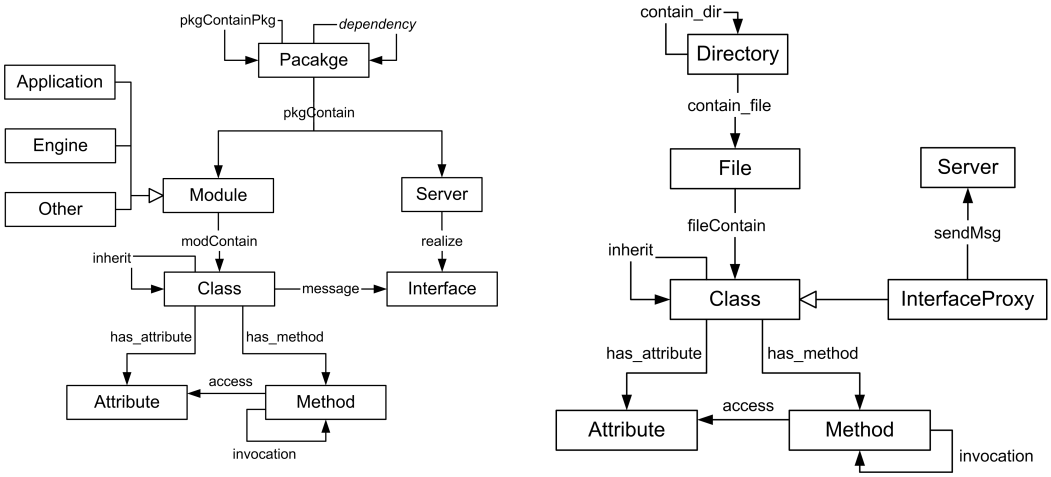
Server On the NOP side there are servers that provide access to the NOP functionality. The NOP servers return the messages through a NOP proxy.

The MSC shows that the current implementation is the result of the integration of elements originated from different systems having different architectural styles. NOP is implemented as a distrusted system while NNP follows an object-oriented style. The integration required to create ad-hoc interfaces in order to make the two architectural styles to coexist (e.g. the NOP proxy).

THE TARGET AND SOURCE VIEWPOINTS

The diagrams in Figure 10.3 show the target and source viewpoint for the first iteration. The target viewpoint has been derived from the FAMIX viewpoint presented in Section 6.4 and shows the concepts that are relevant for the first iteration. Our goal is recover the object-

oriented design, the package hierarchy and the dependencies with the NOP servers. The relation *message* indicates that a class sends a message to a NOP server. The source viewpoint shows the elements that we need to gather from the implementation. The `InterfaceProxy` is a class that represents the messages that can be sent to the NOP servers.



The target viewpoint. The source viewpoint.

Figure 10.3: The viewpoints for the first iteration of NNP.

THE MAPPING RULES

We list the rules for mapping the elements from the source viewpoint to the target viewpoint:

- The `Module` is mapped to a `Directory`. For the level of granularity of the first iteration there is a one-to-one mapping between modules and first-level directories.
- The *pkgContain* groups module in logical packages. In the first iteration we do not explicitly calculate this relation.
- The `Interface` is mapped to an `InterfaceProxy`.
- The relation *message* is calculated by abstracting the invocation between the methods of the class and the `send_msg` method of the class `InterfaceProxy`.

10.4.3 DATA GATHERING

The source view contains the FAMIX concepts and NPP specific concepts. We decided to extract the information with two different techniques and to combine the results. NPP

is written in C/C++ and we can extract the FAMIX related concepts with SOURCENAVIGATOR and the snav2nimeta script (as described in Chapter 8). For the NPP specific concepts, we extended the snav2nimeta script with a section that searches for particular regular expressions with the GREP functionality of SOURCENAVIGATOR . The following code shows an example of a code pattern that we search with the regular expressions. An object that instantiates the class `CRequestMsg` can send the `Create` request to the server `O_SERVER`.

```
{
CCreateReqMsg* CCreateReqMsg::New() {
    CCreateReqMsg* msg = new CCreateReqMsg();

    msg->set_server(O_SERVER);
    msg->set_msg(MCreate);
    ...
    return msg;
}
```

10.4.4 KNOWLEDGE INFERENCE

The goal of the first iteration is to show the module-level dependencies. Modules are directly mapped to directories and all the relations for the target view are available in the source view. We lift the low-level dependencies (*invocation*, *access*, *inherit* and *message*) to the class-level and then to the module-level. We lifted the elements using RIGI and two RCL scripts. With the script *collapse_classes* (see Appendix A.2.2) we group the methods and the attributes of a class in a new class node. With the script *collase_dirs* (see Appendix A.2.3) we group all the content of the top-level directories in the module elements. The *dependency* is automatically calculated by RIGI while collapsing the nodes.

10.4.5 PRESENTATION

We presented the target views with graphs in RIGI . The graph in Figure 10.4 shows the high-level dependencies among the modules. It is a highly connected graph that demonstrates the complex inter-dependencies at this stage of the development. The module on the top-left represents the `NNP GUI Library` and it is the focus of the next section.

One requirement of the architects is the possibility of analyzing the dependencies between the NNP and NOP. The graph in Figure 10.5 demonstrate one example of such analysis. We have selected one module and we have filtered all the classes that are connected to a NOP

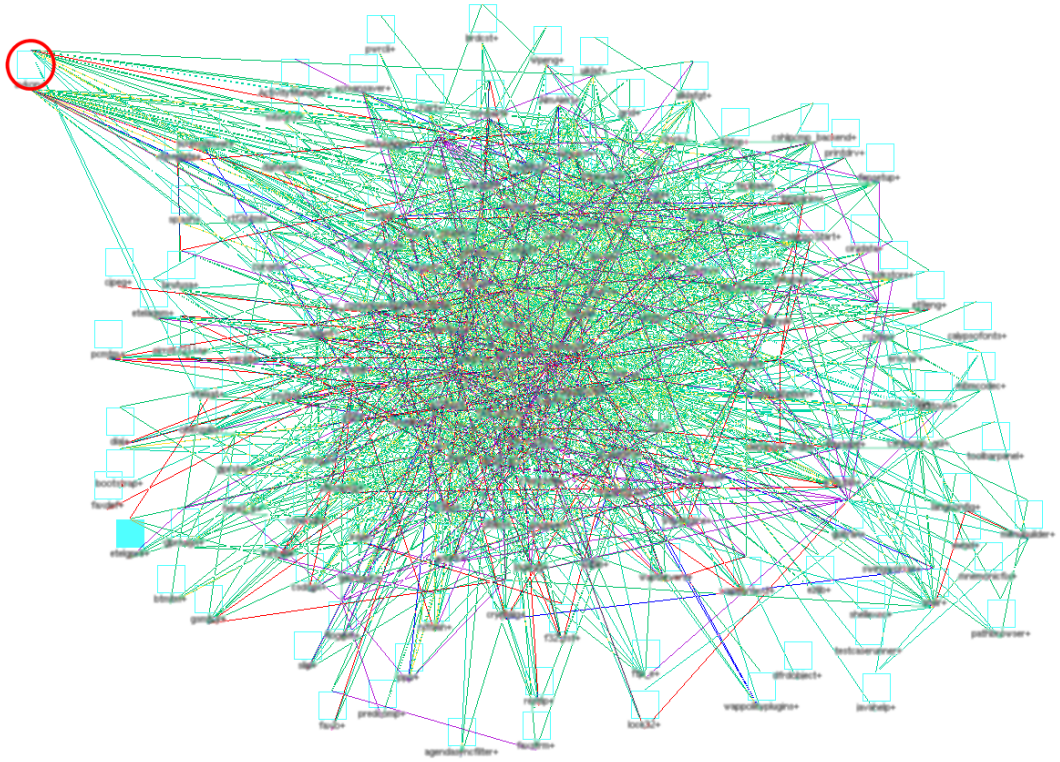


Figure 10.4: Top-level dependencies among the modules of the prodcut based on NNP.

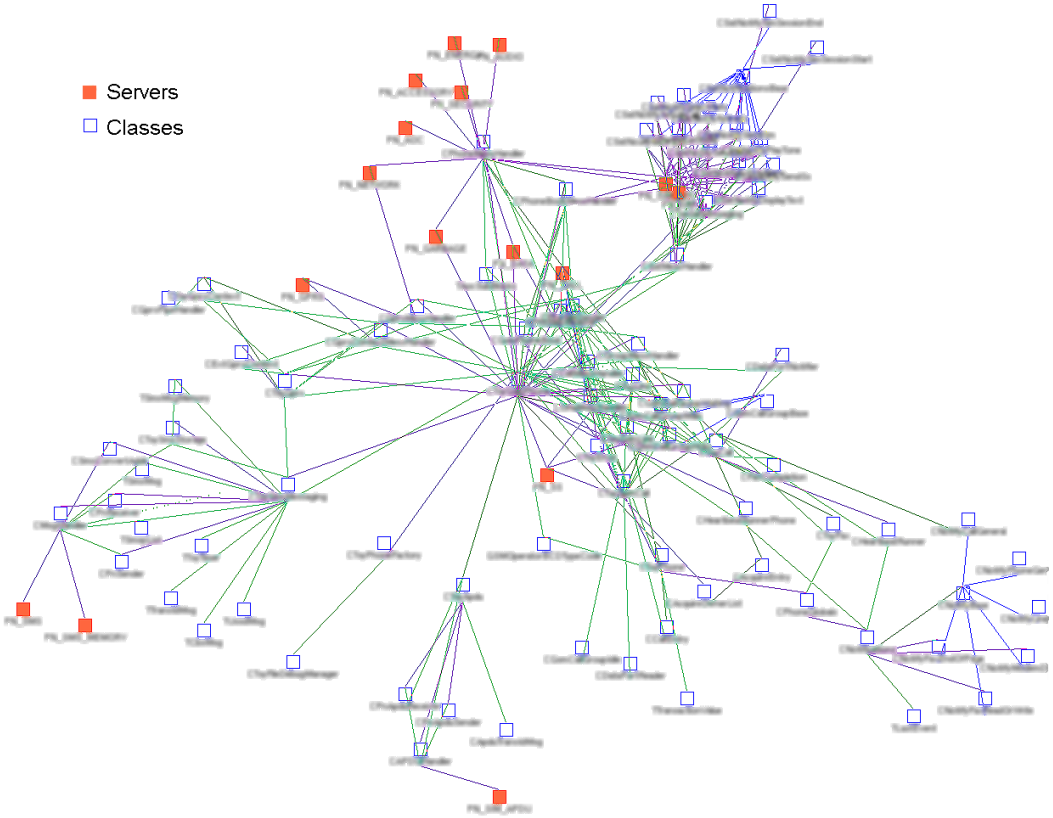


Figure 10.5: Dependencies between NNP classes and NOP servers in one module.

server. The unfilled nodes represent classes and the filled nodes represent the NOP servers. The composite dependencies among the classes are induced by low-level dependencies (like method invocation, variable access and class inheritance). The composite dependencies between classes and servers are induced by the exchange of the asynchronous messages.

The graph in Figure 10.6 shows the dependencies between the a set of modules and the NOP servers. We can note that most of the servers are used by one module (the marked node) that is actually responsible for managing most of the telecommunication services of the product.

We can also analyze the what interfaces of the NOP servers are used by NNP. The graph in Figure 10.7 shows the interfaces that are used by one module.

The architects appreciated the possibility of analyzing the dependencies with old Nokia platform and this was used for creating NNP's reference architecture in the second iteration.

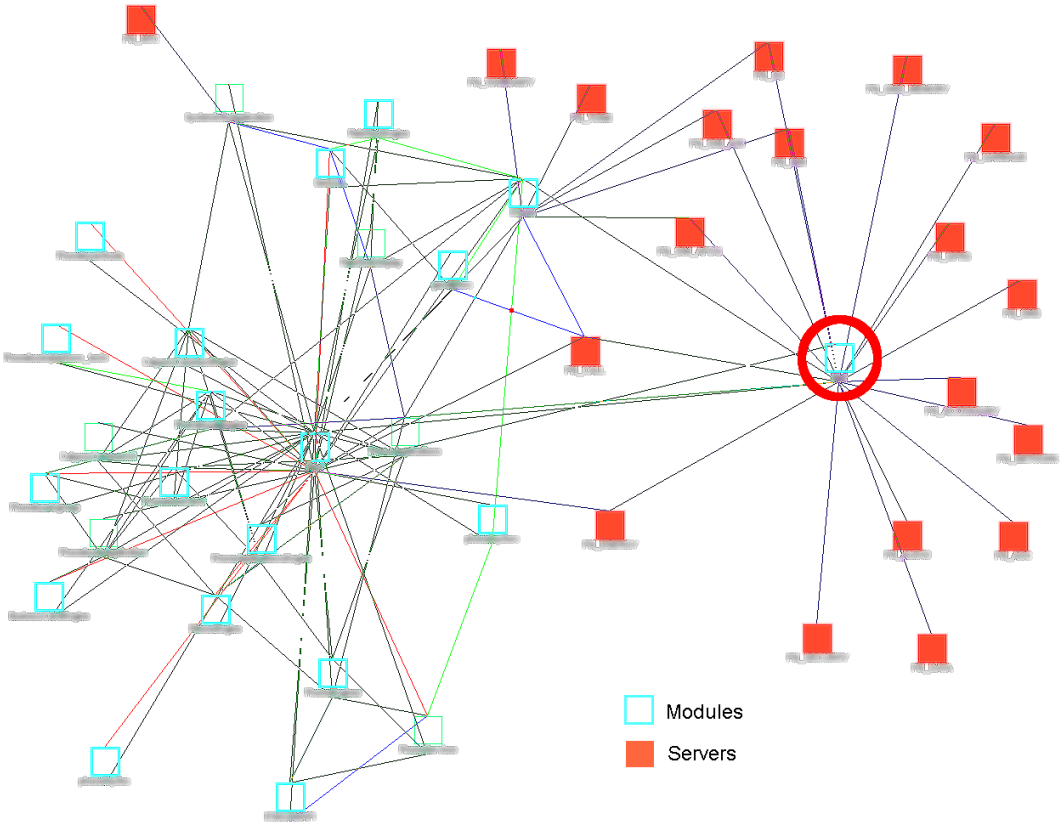


Figure 10.6: Dependencies between NNP modules and NOP servers.

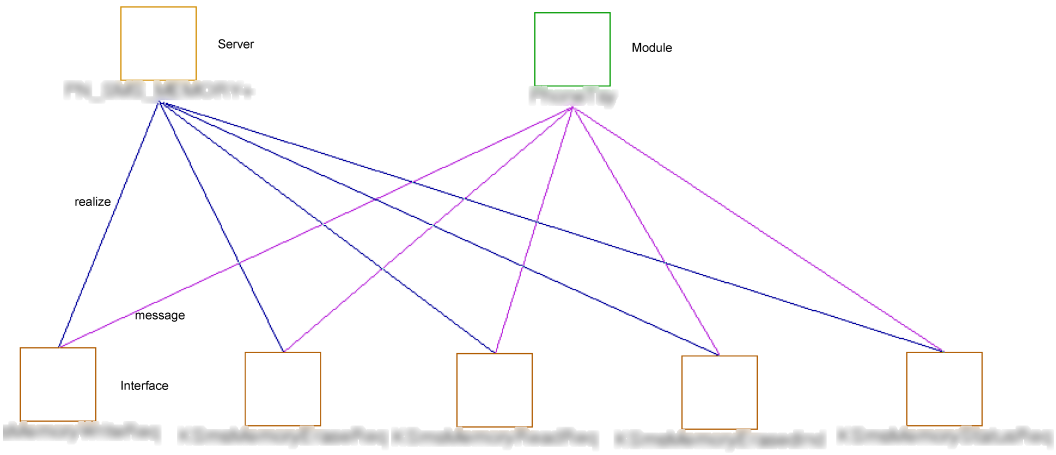


Figure 10.7: Dependencies between NNP modules and NOP servers.

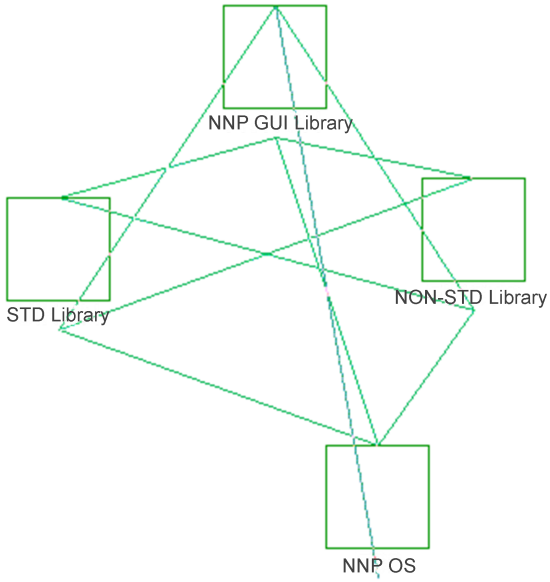


Figure 10.8: The top-level dependencies between the packages under assessment.

10.4.6 Architecture Assessment

The main goal of the first iteration was to assess the dependencies of the NNP GUI Library. As discussed earlier, the graphic library that is used by the product is the result of a long process of evolution. The graphic library was originally developed in conjunction with the third party OS and then reengineered several times to adapt it to new scenarios. The current implementation contains parts that are not in use anymore, parts that are under the control of a third-party and parts that are developed by Nokia. The goal of the architects was to re-architect the current implementation in a coherent and comprehensive GUI library for a new generation of products. The library should provide an high-level API for the products based on the NNP platform. We conducted an assessment of the current implementation that resulted in a detailed report that was handed to the architects. The architects considered the results of the report to base their future actions.

The graph in Figure 10.8 shows the top-level dependencies among the packaged under assessment. The packages contains selected modules that we grouped manually with RIGI . The graph shows that the NNP GUI Library package has cross-dependencies with the the modules from the three other packages. From this view, the architects can inspect the details of the dependencies.

The graphs in Figure 10.9 and in Figure 10.10 show the details of the dependencies from

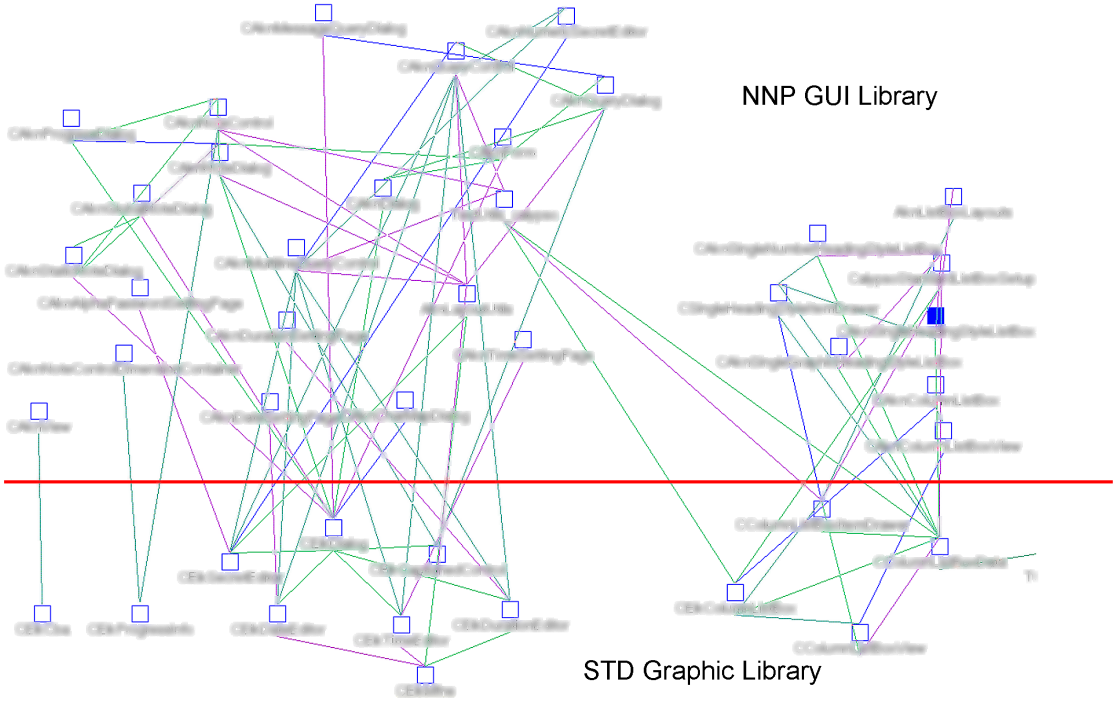


Figure 10.9: Class-level dependencies from the NNP GUI Library to the STD Graphic Library.

the classes of NNP GUI Library to the classes of STD Graphic Library and viceversa. The graphs can inspect the nature of the dependencies: inheritance, method invocation or variable access.

10.5 The Second Iteration

Since the reegnineering work continued, the architects assigned us the task of recovering the as-implemented structure of the NNP platform. The task lasted about one year and was conducted by two reconstructors working part-time.

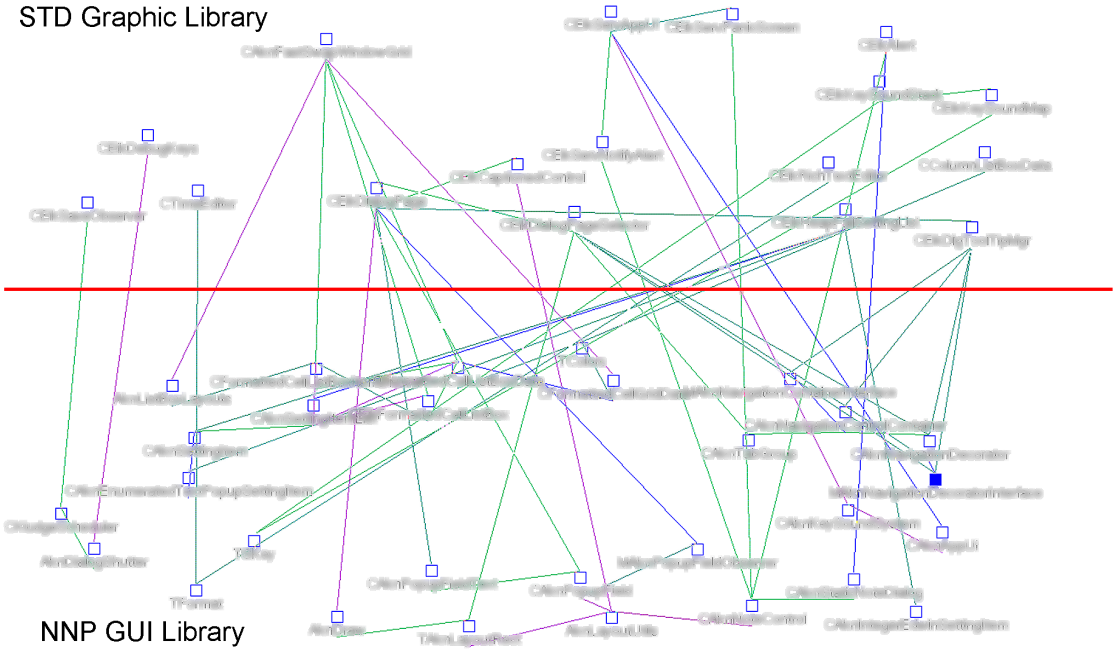


Figure 10.10: Class-level dependencies from the STD Graphic Library to the NNP GUI Library.

10.5.1 Problem Definition

When we started the second iteration the NNP platform started to assume a more important role in the organization and more investments were allocated. The reegineering effort was still progressing. The design of NNP was improved since the first iteration: well-defined APIs, consistent partition of functionality and clear context of the platforms (in terms of required/offered interfaces). The architects started the activity of defining the reference architecture of NNP. The reference architecture should provide an overview of the system, a description of the architecturally significant requirements, the rationale of the design choices, the description of the architectural concepts and overview of the concrete design (with information on the functionality, interfaces and ownership). Our task was to support the creation of the reference architecture with the reconstruction activity. In particular our focus was on the help the creation of the concrete design. Out work should also pave the way for implementing a strict practise of architecture governance when NNP will enter in full operational mode and will serve as a basis for a large product family. The goal of the architects was to provide the platform with a robust reconstruction mechanism that would allow them to constantly monitor the conformance of the implemented architecture against their architectural decisions. The diagram in Figure 10.11 shows the general structure of the NNP platform as it was defined during the second iteration. The NNP platform includes

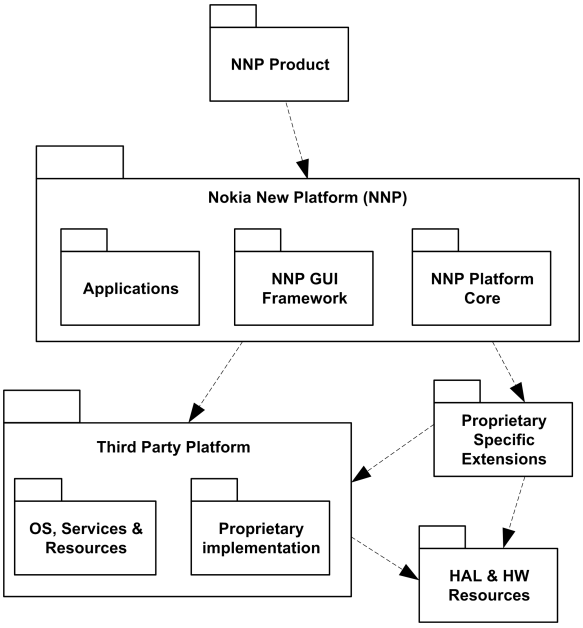


Figure 10.11: General structure of the NNP platform.

the NNP GUI framework, the platform core and a set of basic applications. The platform depends on the third party OS and on proprietary specific extensions (this includes the reengineered functionality from NOP). The scope of the second iteration is on the NNP platform package.

10.5.2 Concept Determination

In the second iteration we reused the concepts and the viewpoints defined in the first iteration with minor changes. Differently from the first iteration, we explicitly define the relations *pkgContainPkg* and *pkgContain* during the knowledge interference activity.

10.5.3 Data gathering

We followed the same extraction procedure like the first iteration. Differently form the first iteration when we analyzed the build of the product, in the second iteration we analyzed one build of the platform without any product specific implementation. (We also excluded the implementation of the OS (except the header files for the interfaces) since it was not the

focus of the analysis).

10.5.4 KNOWLEDGE INFERENCE

The architect responsible for the reference architecture activity created a list of all the modules that belong to the NNP. The list contained the details about what source directories belong to the module, the functional location in the package structure and the ownership. Although this information was still incomplete, it was the base for defining the relations *pkgContainPkg* and *pkgContain*. We manually allocated the remaining modules by examining the target APIs for the platform and the hypothetical logical structure of NNP. The servers have also been allocated to one package. The work resulted in about 300 containment relationships between packaged and modules.

We defined the generation of the target view with the relational algebra. The first operation is to lift the *invocation* and *access* relations to the class level:

$$\begin{aligned} classUsage &= invocation + access \\ classContain &= has\_method + has\_attribute \\ useDep &= classUsage \uparrow classContain \end{aligned}$$

For the dependencies with the NOP servers, we make a left lift:

$$msgDep = message \upharpoonleft has\_method$$

The relation that contains the complete class-level dependencies is defined as:

$$classDep = useDep + msgDep + inherit$$

In the second iteration we maintain the one-to-one mapping between modules and directories as in the first iteration. Therefore, we can calculate the relation *modContain* in the following way:

$$modContain = contain\_file \circ fileContain$$

We can define the relation *contain* as the union of the package containment, the module containment and the interface realization (the interfaces of the servers are logically grouped within the servers themselves):

$$contain = pkgContainPkg + pkgContain + realize$$

We can calculate the high-level dependencies for the target view by lifting the class-level

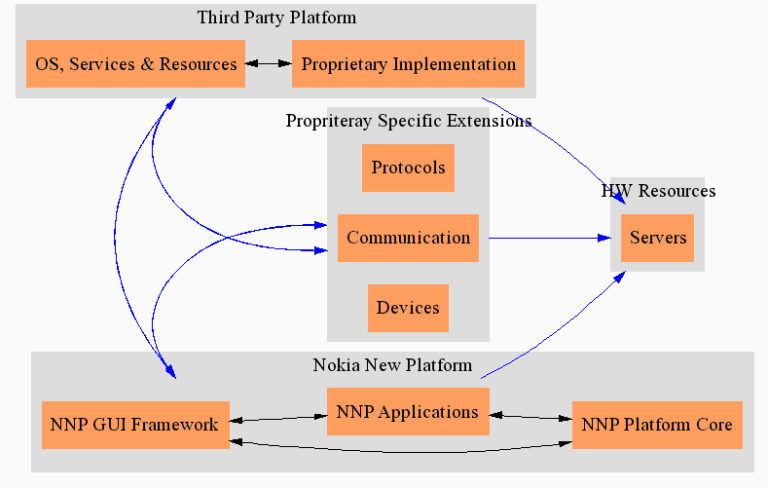


Figure 10.12: The top-level structure of the NNP platform.

dependencies:

$$dependency = classDep \uparrow contain$$

10.5.5 Presentation

We presented the final results with graphs generated from RIGI and DOT . The graph in Figure 10.12 shows the top-level result of the reconstruction. We can compare it with the intended design shown in Figure 10.11 and we can note that there are several unwanted dependencies that need to be fixed in the implementation. The graph in Figure 10.13 shows the internal dependencies among the packages of the NPP platform. This view shows the information that was requested from the architects. This material has been used also for the preparation of the reference architecture of NNP.

10.6 The Third Iteration

NNP has matured into a fully functional platform and it is the base for the development of several products. The focus of the architects is to establish a proper architectural control of the platform. The third iteration is focused on this goal.

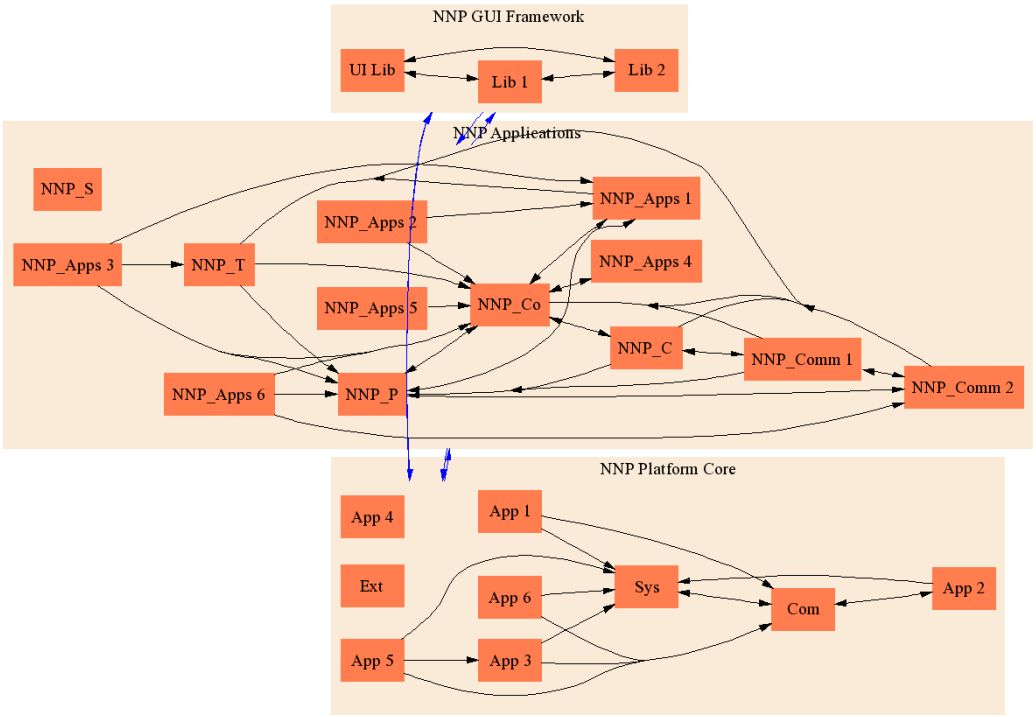


Figure 10.13: Structure of the NNP platform.

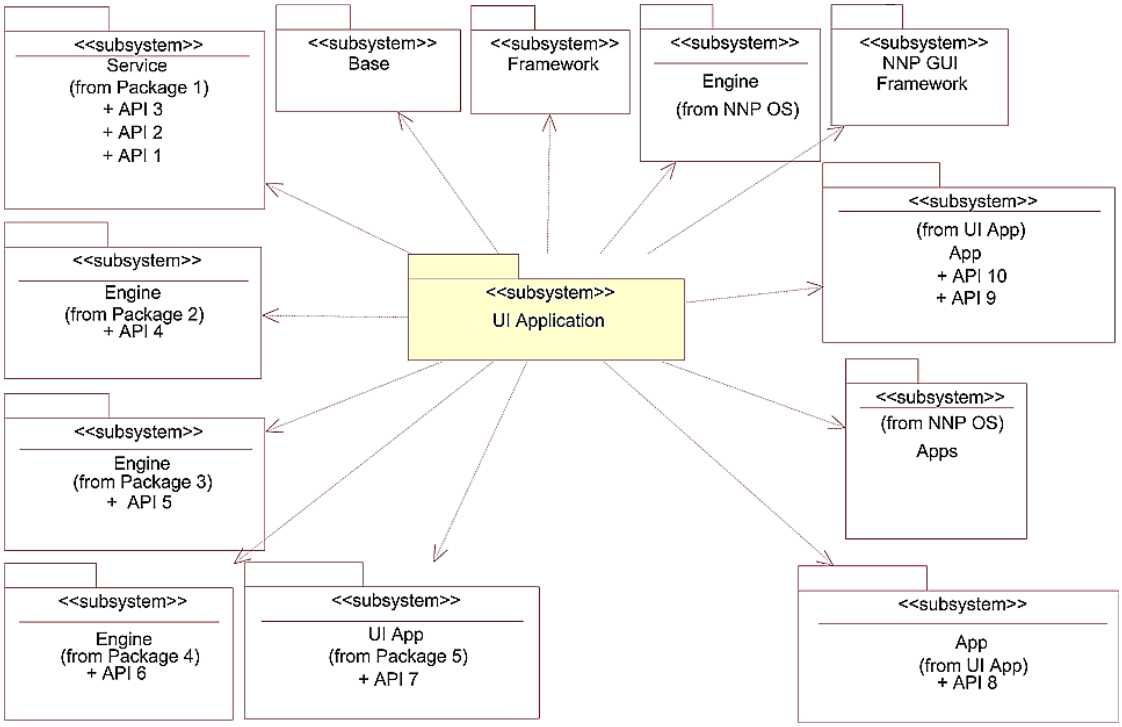


Figure 10.14: Structure of the NNP platform.

10.6.1 Problem Definition

The third iteration started when the NNP was well-established and the architects were focused on improving the overall quality of the architecture. One key requirement is to ensure that the architectural specifications are properly implemented in the platform.

The organization established a solid process for the architectural design of NNP. Every architect is responsible for the design of one or more subsystems. The design includes the specicifcation of the context of the subsystem, offered/required interfaces and the internal design in terms of components/classes. Due to the size of the system and to the subcontracting, the specifications are very strict on what the subsystem must offer and can require. The whole design activity is based UML and all the design artifacts are stored in Rational Rose. The UML diagram in Figure 10.14 shows an example of a context diagram for one subsystem. The diagram clearly specifies what other subsystems and interfaces are required. The UML diagram in Figure 10.15 shows the internal design of one subsystem in terms of components and APIs.

We discussed with the architects the requirements for the reconstruction and we concluded

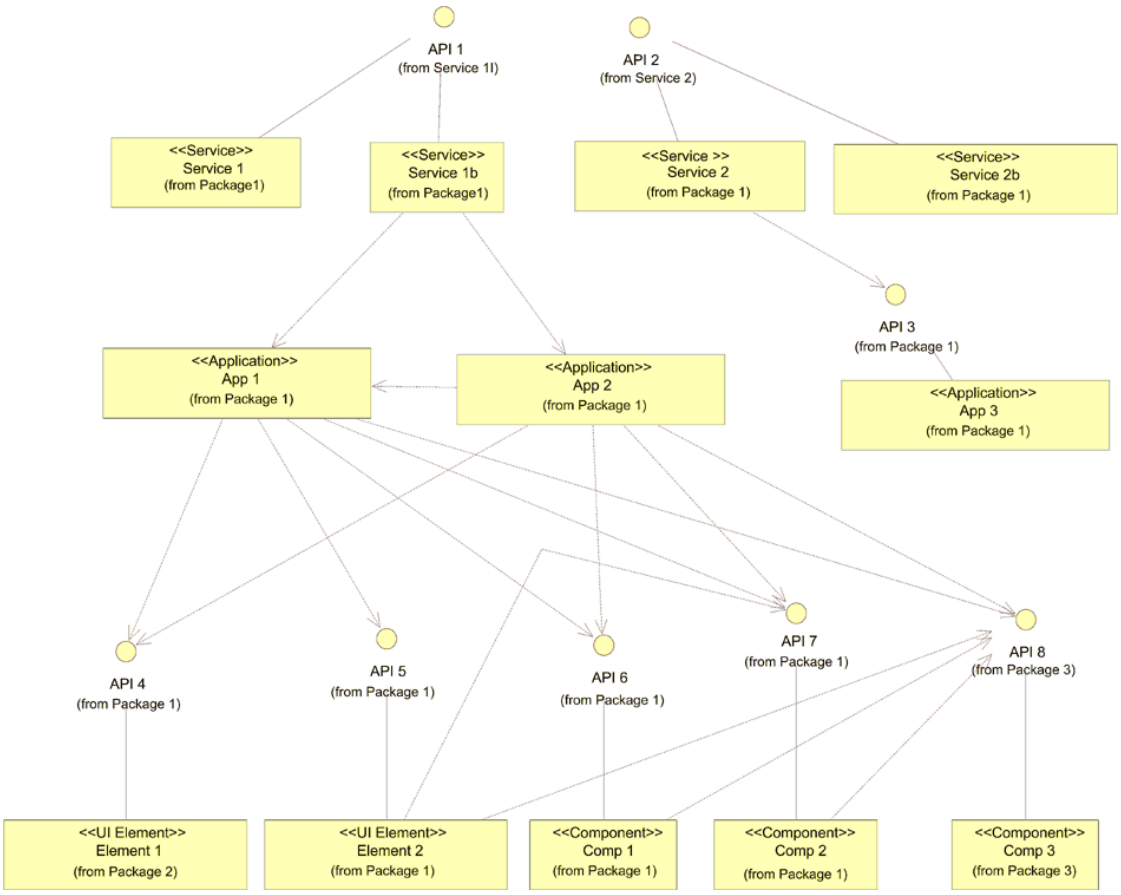


Figure 10.15: Structure of the NNP platform.

that two views are required: the *logical* and the *build* view. The logical view shows the logical subsystems, the logical components and the interfaces. One typical scenario is that an architect can select one subsystem and can create subsystem context view at the level of interfaces. The build view shows the relationship between logical and physical components. The user should be able to select one subsystem and find what binary files are included and what source files are included.

The architects already have developed tools for analyzing the static dependencies between subsystems based on the binary dependencies. The added value of our reconstruction process is being able to show the dependencies at the level of interfaces. The dependencies among the components are only allowed through the well-defined interfaces.

10.6.2 Concept Determination

We organized a one day workshop with the architects of NNP in order to define the architectural concepts for the third iteration. We had to redefine most of the concepts that we used in the previous iterations. The previous iterations were mainly focused on the platformization work while in the third iteration the reconstruction is mainly focused on the logical aspects of the NNP platform.

We identified the following key architectural concepts:

Component A logical component consists of a set of source files and provides interfaces to other components. The interfaces are defined in the header files.

Subsystem A subsystem is an aggregate of components that implement a set of related functionality.

Package Packages are used for organizing the subsystems in logical folders.

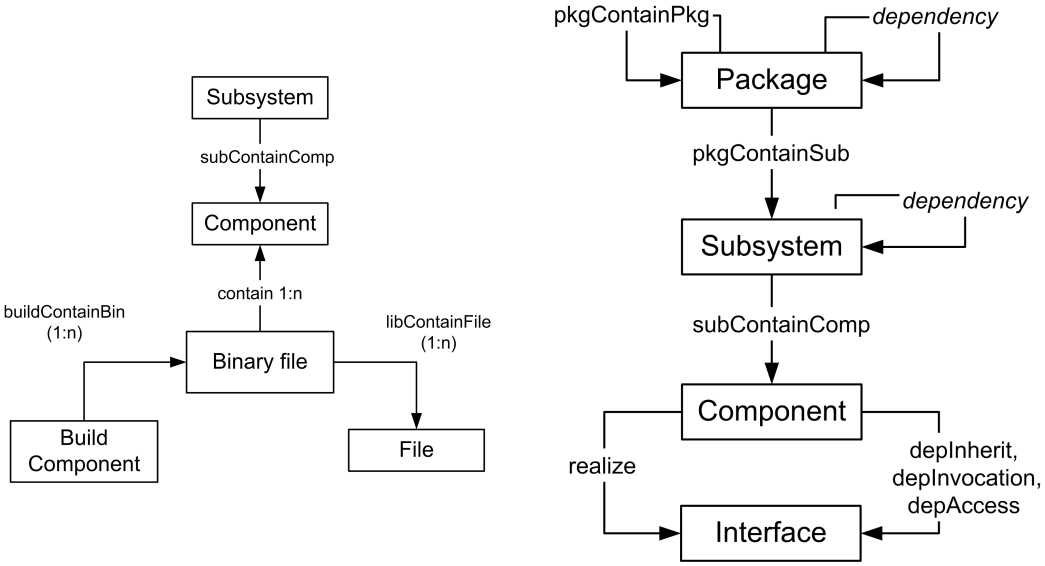
Interface An interface is represented by a class declared in a header file.

Binary File A binary file is generated during the build process. A binary configuration file lists the files that are included in the binary file.

Build Component A binary component is a collection of binary file. In most of the cases a binary component is directly mapped to one directory.

THE TARGET VIEWPOINT

The diagram in Figure 10.16 show the target viewpoints for the third iteration. The logical viewpoint reflects the main interest of the architects on the component/interface dependencies. In this iteration we assume that one interface can only be realized by one and only component. In the future we may change this assumption.



The build viewpoint. The logical viewpoint.

Figure 10.16: The viewpoints for the third iteration of NNP.

THE SOURCE VIEWPOINT

The diagram in Figure 10.17 shows the source viewpoint for the third iteration. It includes the design concepts from FAMIX (in gray) as in the previous iterations and the new concepts that we need to create the target views.

THE MAPPING RULES

We list the mapping rules for the third iteration:

- The relations *depInherit*, *depInvocation* and *depAccess* are respectively calculated from the relations *inherit*, *invocation* and *access* among the classes that are contained in the components.

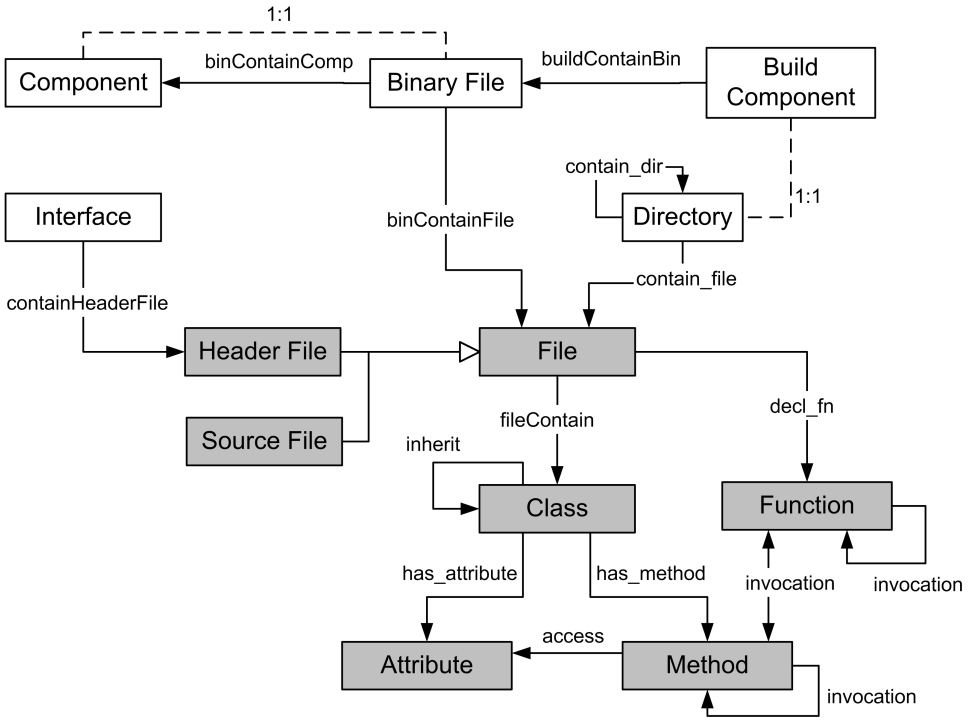


Figure 10.17: The source viewpoint.

- The relation *realize* is calculated from the relations *binContainFile* and *binContainComp*.
- The relation *subContainComp* is available from a spreadsheet that contains detailed information for each binary file (like the name, the containing subsystem and the size).
- The relation *pkgContainSub* and *pkgContainPkg* are available from the design artifacts and can be extracted from Rational Rose.
- A `Component` is a logical entity and in this iteration is mapped directly to a binary file. In the future the mapping will be refined.
- A `Build Component` is mapped one-to-one to a `Directory`.

10.6.3 DATA GATHERING

We followed different strategies for gathering all the data. For the FAMIX concepts (in gray in Figure 10.17) we relied on Source Navigator as in the first iteration. The output of Source Navigator has been stored in a RSF file. For the architectural elements we used the following methods:

- An `Interface` is mapped to a set of header files through a mapping table. The table is maintained by the architects in a spreadsheet from where the relation *containHeaderFile* is extracted. We stored the relation in a RSF file.
- The relation *subContainComp* is extracted from the spreadsheet of the binary files and stored in a RSF file.
- The *binContainFile* is extracted from the configuration files. A Python script traverses all the file system, parse the configuration files and stores the result in a RSF file.
- The relation *pkgContainSub* and *pkgContainPkg* are available from the design documents and can be extracted from Rational Rose.
- The relation *buildContainBin* is based on the configuration files that are contained in the directory of the build component.
- The relation *binContainComp* is a one-to-one mapping between logical components and binary files.

We merged the results from the various sources by concatenating all the RSF files. The scripts have been implemented using Python.

10.6.4 KNOWLEDGE INFERENCE

We defined the generation of the target view with the relational algebra. In this iteration, we assume the one-to-one mapping between components and binary files, therefore we can define the relation *compContainFile* that maps components to files:

$$compContainFile \;=\; binContainFile$$

We can calculate the interfaces that are realized by the components in the following way:

$$realize \;=\; compContainFile \circ containHeaderFile^{-1}$$

The classes and functions that belong to one interface are calculated as:

$$interfaceContain = containHeaderFile \circ (fileContain + decl\_fn)$$

We can also calculated the classes and functions that belong to the components through their interfaces:

$$compContain \;=\; realize * interfaceContain$$

Now we can lift the low-level dependencies to the level of classes and functions that represent the basic elements of interfaces:

$$\begin{aligned} classInvocation &= invocation \uparrow has\_method \\ classAccess &= access \uparrow (has\_attribute + has\_method) \\ funcInvocation &= invocation \upharpoonleft has\_method + invocation \upharpoonright has\_method \end{aligned}$$

We can calculate the dependencies between components and interfaces in the following way:

$$\begin{aligned} depInvocation &= compContain \circ funcInvocation \circ interfaceContain^{-1} + \\ &\quad\; compContain \circ classInvocation \circ interfaceContain^{-1} \\ depAccess &= compContain \circ classAccess \circ interfaceContain^{-1} \\ depInherit &= compContain \circ inherit \circ interfaceContain^{-1} \end{aligned}$$

We can define the relation *contain* as the union of the package containment and subsystem containment. The interfaces are logically grouped together with their components through

the relation *realize*. We can do this operation because we assume that an interface can only be realized by one component. In the future we may loose this constraint and implement a different grouping for interfaces (in order to support variability):

$$\begin{aligned} contain &= pkgContainPkg + pkgContainSub + subContainComp + \\ &\quad subContainComp \circ realize \end{aligned}$$

We can calculate the high-level dependencies for the target view by lifting the component-interface dependencies:

$$usage = depInvocation + depAccess + depInherit$$
$$dependency = usage \uparrow contain$$

10.6.5 PRESENTATION

Since the architects prefer to store the reconstructed models in the same format as the design artifacts, we decided to converted the component view to UML. We have converted the models with the following conventions:

- An `Interface` is mapped to a UML interface element.
- A `Component` is mapped to a UML stereotyped classifier called Component.
- The relations *depInvocation*, *depInheritance* and *depAccess* have been mapped to UML stereotyped dependencies.
- A `Subsystem` and a `Package` have been mapped to UML stereotyped packages.
- The relation *dependency* has been mapped to the UML dependency.

The result is a model that can be visualized with Rational Rose. An example is shown in Figure 10.18. The architects appreciated that the reconstructed models can be integrated with their architecting tool. In the future, we will focus in achieving a seamless integration with UML in order to create a unique model for architecting/rearchitecting as we produced for the case study 1.

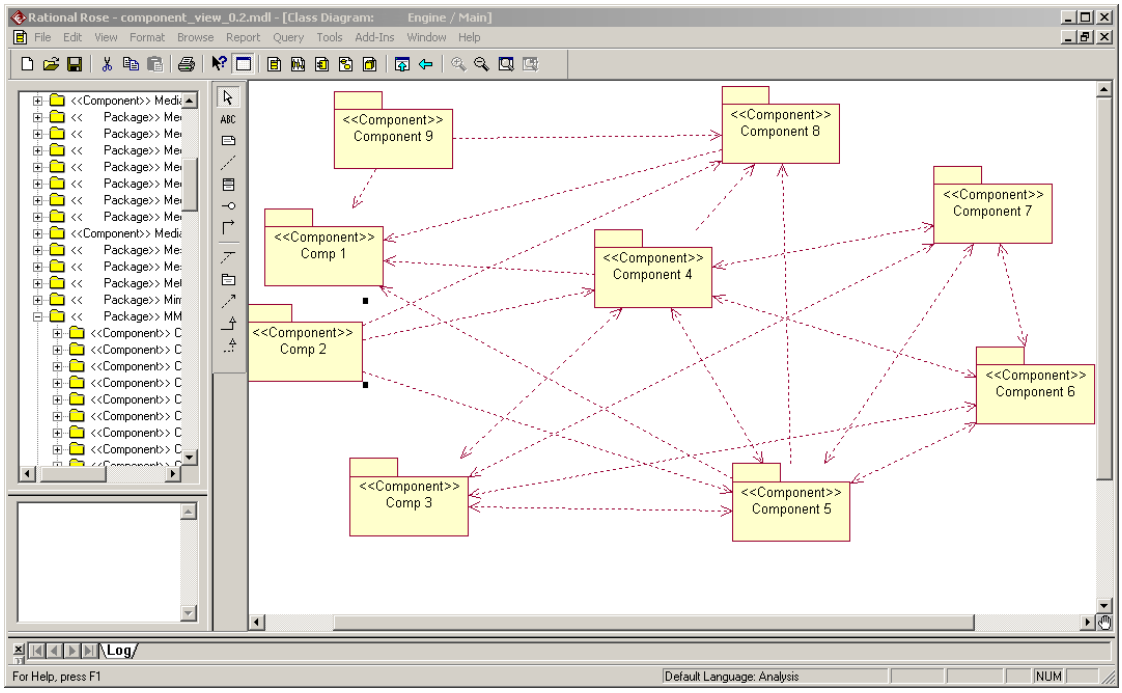


Figure 10.18: The context diagram of one subsystem.

10.7 Conclusions and Lessons Learned

This case study demonstrated how architecture reconstruction has supported various tasks of a platformization activity. Initially, we helped the architects with the analysis of the dependencies among the modules of the original product. Successively, we helped them with the creation of the reference architecture by recovering up to date models of the concrete structure of the platform. As the one-shot development was completed, in the third iteration we have helped them to introduce a systematic method for checking the as-implemented architecture against the architectural specifications. This work will continue in the future in order to provide a robust and automatic method of architecture conformance checking as we introduced in the case study 1.

We summarize below the lessons that we have learned during the experience:

- At the beginning of the case study, it took considerable time to generate enough interest for the reconstructed models. We believe that this was caused by the fact that the architects and developers were overwhelmed by the one-shot development and they had to carefully allocate their resources. There was a natural resistance against introducing new methods in their development practise, unless they could see immediate

benefits. For this reason, in the first and second iteration we carefully targeted our reconstruction efforts on very precise goals. As the one-shot development completed, we could set long-term goals for supporting the architecture maintenance.

- Our reconstruction method revealed to be flexible against changes in the architectural concepts and in the reconstruction process. We could easily adapt the process to the changing goals of the architects over the three iterations.

- We experienced a noticeable resistance against the introduction of new tools. The choice to deliver the reconstructed models in Rational Rose following UML was largely appreciated by the architects. This fact supports the belief that reverse engineering has to be integrated with the same tools that are used during forward engineering. In the future, we will work on achieving a closer integration with Rational Rose as we have presented in (Riva *et al.* , 2004a).

CHAPTER 11

CONCLUSIONS

Quality isn't something you lay on top of subjects and objects
like tinsel on a Christmas tree.

- Robert Pirsig

In this chapter we summarize the results of this dissertation and we discuss the possible future research topics.

11.1 SUMMARY

In this dissertation we have presented the method NIMETA for reconstructing the software architecture of large embedded software systems as developed at Nokia. These systems are typically organized in a product family, consists of millions of lines of code. Since they represent a considerable long-term investment for Nokia, they require a robust and well-understood software architecture. NIMETA provides the method and tools to support not only the recovery of the architecture description but also the conformance check against the architectural constraints.

In Chapter 2, we introduce the background information about software architecture and reverse engineering. In our working definition, a software architecture comprises the important design decisions taken for the development of the system. Architecture reconstruction is the process of creating an architecture description that document those design decisions that are considered vital by the developers. The overview of the related work (Chapter 3) shows that the existing reverse engineering approaches are not able to raise the level of

abstraction at the architectural level and they are limited on one or few architectural viewpoints. In Chapter 4 we have presented the main problem addressed by this dissertation: developing a methodology for reconstructing a documented architecture from the available evidence (implementation, documentation and people). In particular, we focus on (1) recovering the architecturally relevant views that are considered vital by the stakeholders of the system, and (2) the problem of automating the conformance checking of the architecture against specific architectural rules. In Chapter 5, we have introduced the mathematical foundations for the NIMETA approach: the binary relational algebra. The algebra provides us a formal and expressive formalism for modeling the design and architectural viewpoints, as we have presented in Chapter 6. For the design viewpoint, we have selected the FAMIX meta-model as it provides a language-independent and abstract representation of the source code. We selected the architectural viewpoints from the unified catalogue proposed in (Clements *et al.* , 2003). In Chapter 7, we introduce the NIMETA architecture reconstruction process. It consists of three phases: process design, view recovery and result interpretation. The process design is concerned with the definition of the goal of the reconstruction (problem definition) and the selection of the architectural concepts and views (concept determination). The view recovery delivers the architectural views and consists of three activities: data gathering, knowledge inference and presentation. The last phase, result interpretation, is concerned with making an efficient use of the reconstructed architecture for architecture conformance checking, architecture assessment and re-documentation. In Chapter 8, we present the NIMETA tool environment that supports the NIMETA architecture reconstruction process. The environment consists of an integrated set of reverse engineering and presentation tools for the three tasks of extraction, abstraction and presentation. Chapter 9 describes the first case study. The three iterations demonstrate the development of an architecture reconstruction process for a Nokia product family. The case study also show how we supported the architecture conformance checking with the NIMETA formalism. Chapter 10 presents the second case study that demonstrates how the architecture reconstruction has been used to support the migration of monolithic product towards a platform for a product family.

11.2 RECOMMENDATIONS

We provide several recommendations for applying NIMETA to other systems or to other domains. The NIMETA process has been designed to be tailored around the particular architectural style of the system under analysis. There are two important factors that one should consider: the architectural concepts and the architectural views.

The correct selection of the *architectural concepts* guarantees that the recovery is well-focused on the architecturally relevant aspects and it is coherent with the architectural style.

In a distributed software system, asynchronous messages cause logical dependencies that are necessary for understanding the overall design of the system. Our recommendation is that the architectural concepts are first-class elements of the reconstruction.

The *architectural views* guarantee the correct scoping of the reconstruction. The description of an architecture consists of multiple views concerned on particular aspects of the system. Moreover, the software architecture does not capture all the details of the implementation but only certain aspects that considered important by the architects. Our recommendation is that the reconstruction process is well focused on the architectural views that have been defined with the stakeholders, especially the architects. NIMETA 's approach is strongly centered on the recovery of the architectural views.

As the system evolves, the reconstruction process has to be correctly maintained. Ideally, the reconstruction process should be embedded in the build process and jointly maintained. In this way it can deliver the architectural views for every new build. NIMETA has been designed as a pipeline of tools and script in order to satisfy this requirement. Our recommendation is that the vendors of architecture modeling tools should enable the users to easily configure the reconstruction pipeline for a seamless integration with the build process.

11.3 CONCLUSION

A properly documented architecture is a vital factor for the success of large software systems containing millions of lines of code. The architects need to create simplified mental models of the important design decisions in order to cope with their inherent complexity. Software architecture reconstruction is a key technique for recovering and maintaining the architecture description. Maintaining the architectural control is especially vital for large product families consisting of hundreds different components, developed by hundreds people concurrently and located in multiple geographical sites. The architects are particularly interested in recovering the logical dependencies among the components and in preserving the integrity of the structural organization of the various software parts. The NIMETA approach addresses these issues in the first place, providing a configurable, adaptable, scalable and robust reconstruction technique.

11.4 FUTURE WORK

We conclude this dissertation by summarizing the future research topics:

Architecture evolution. The evolution of the architecture has to be properly supported with techniques for comparing the architectural differences between different releases, calculating the architectural metrics and, in general, monitoring the overall quality of the architecture.

Design for reconstruction. Our claim is that software systems should be designed not only for change (in order to anticipate the future changes) but also for reconstruction. There are architectural styles that facilitate the reconstruction more than others. The architecture should be properly designed in order to support its own extraction. In the future, we will concentrate on creating several guidelines for supporting the design for reconstruction and on integrating those guidelines with NIMETA .

UML conformance. UML is a widely accepted modeling language that has also been proposed as an architecture description language. Our intention is achieve a seamless integration of UML with NIMETA . In the future, we will exploit how to support a complete architecting and re-architecting with UML. In the Section 8.5.6 we have already presented our initial results with the tool ART.

Architect assistant. The increasingly complexity of existing software systems is reaching the limits of human comprehension. In a recent invited talk, P. Grahm speculated on what programming language will be like in a hundred years when the hardware computational power will be much higher than now (Graham, 2003). Our concern is not only how these systems will be developed but how they will be maintained. Since their complexity will probably exceeds humans' comprehension, we believe that artificial intelligence techniques will be required. In the future we will concentrate on developing an expert system, called architect assistant, that can help the work of the architects. The assistant should be able to automatically extract the data, detect anomalies and prioritize the detected architectural problems.

APPENDIX A

NIMETA SCRIPTS

A.1 EXTRACTION SCRIPTS

This appendix lists the source code of various programs that we have discussed during in other sections.

A.1.1 SNAV2NIMETA.TCL

```
#!/usr/local/apps/source-navigator/latest/bin/hyper
# Snav2nimeta.tcl
#
# Converter from the RedHad SourceNavigator to the FAMIX model
# The output is stored in RSF or GXL format
#

#Global variables
set project_name ""
set project_path ""
set project_dir ""
set ext "*"
set cl 0
set attr 0
set meth 0
set meth_imp 0
set fl 0
set func_decl 0
set func_def 0
set mac 0
set gv 0
set tdef 0
set inh 0
set acc 0
set inv 0
set incl 0
set output stdout
set ofile stdout
set format "rsf"

set GXLIDCounter 0
set array IDToName
set array NameToID

# Is this object a function or method implementation? Returns 1 if so, 0
# otherwise.
proc isfunc {objtype} {
    if {[string compare "$objtype" "fu"] == 0 || \
        [string compare "$objtype" "mi"] == 0 || \
        [string compare "$objtype" "fd"] == 0 } {
    return 1
    }
```

```
    return 0
}

# Is this object a global or local variable implementation? Returns 1 if so, 0
# otherwise.
proc isvar {objtype} {
    if {[string compare "$objtype" "gv"] == 0 || \
        [string compare "$objtype" "iv"] == 0} {
    return 1
    }
    return 0
}

# A class ?
proc isclass {objtype} {
    if {[string compare "$objtype" "cl"] == 0} {
    return 1
    }
    return 0
}

# A method attribute ?
proc ismacro {objtype} {
    if {[string compare "$objtype" "ma"] == 0} {
    return 1
    }
    return 0
}

# A type definition ?
proc istypedef {objtype} {
    if {[string compare "$objtype" "t"] == 0} {
    return 1
    }
    return 0
}

# Transform a class/method name into a fully qualified C++ method (ie.
# class.method).
proc qualifyFunction {class method argument} {
    set signature [createSignature $method $argument]
    if {[string compare $class "\#"] == 0} {
        return $signature
    }
    return "${class}.${signature}" ;# C++ style naming
}

# Transform a class/variable name into a fully qualified C++ variable (ie.
# class.variable).
proc qualifyVariable {class variable} {

    if {[string compare $class "\#"] == 0} {
        return $variable
    }
    return "${class}.${variable}" ;# C++ style naming
}

# Create a complete signature for functions and methods
proc createSignature {routine argument} {

    regsub -all " " "${routine}(${argument})" "" signatureShort

    return $signatureShort
}

# printFiles(path)
# Print the file system structure of the files in the project
proc printFiles {path} {
    global ext

    # Open the .f file of SNavigator
    set filesDB [dbopen nav_file $path.f RDONLY 0644 btree {cachesize=200000}]

    #Extract all the files
    set files [$filesDB seq ]

    foreach entry $files {
        set filePath [lindex $entry 0]
        set fileType [lindex [lindex $entry 1] 0]

        # Print only .c* files
        if [string match $ext $fileType] {
            set parts [split $filePath /]
            set counter [expr [llength $parts]-2]

            #Extract the name of the directory
```

```
            set fileDir [join [lrange $parts 0 $counter] /]

            if {$fileDir == "" } {
                set fileDir "./"
                printNode $fileDir "Directory" {}
            }

            set attr [list "file $filePath"]

            printNode $filePath "File" $attr
            printArc $fileDir $filePath "contain_file"

            while {$counter > 0} {
                set element [lindex $parts $counter]
                set parentDir [join [lrange $parts 0 [expr $counter-1]] /]
                set dirName [join [lrange $parts 0 [expr $counter]] /]
                printNode $parentDir "Directory" {}
                printNode $dirName "Directory" {}
                printArc $parentDir $dirName "contain_dir"
                incr counter -1
            }
            if {$counter == 0} {
                printNode $fileDir "Directory" {}
            }
        }
    }
    unset files
}


# printFuncDef (path)
# Print the functions definitions
proc printFuncDef { path } {
    global functions

    # Open the .fu file of SNavigator
    set funcDefDB [dbopen nav_funcf $path.fu RDONLY 0644 btree {cachesize=200000}]

    #Extract all the function definitions
    set funcDef [$funcDefDB seq]

    foreach entry $funcDef {
        set name [lindex [lindex $entry 0] 0]
        set file [lindex [lindex $entry 0] 2]
        set argument [lindex [lindex $entry 1] 3]
        set signature [createSignature $name $argument]
        set attr [list "file $file"]
        printNode $signature "Function" $attr
        printArc $file $signature "def_fn"
    }
    unset    funcDef
}

# printFuncDecl (path)
# Print the functions declarations
proc printFuncDecl { path } {
    global functions

    # Open the .fd file of SNavigator
    set funcDeclDB [dbopen nav_funcd $path.fd RDONLY 0644 btree {cachesize=200000}]

    #Extract all the function declarations
    set funcDecl [$funcDeclDB seq]

    foreach entry $funcDecl {
        set name [lindex [lindex $entry 0] 0]
        set file [lindex [lindex $entry 0] 2]
        set argument [lindex [lindex $entry 1] 3]
        set signature [createSignature $name $argument]

        set attr [list "file $file"]
        printNode $signature "Function" $attr
        printArc $file $signature "decl_fn"
    }
    unset    funcDecl
}

# printClass (path)
# Print the functions declarations
proc printClass {path} {
    # Open the .cl file of SNavigator
    set classDB [dbopen nav_class $path.cl RDONLY 0644 btree {cachesize=200000}]

    #Extract all the classes
    set class [$classDB seq]
```

```
    foreach entry $class {
        set name [lindex [lindex $entry 0] 0]
        set file [lindex [lindex $entry 0] 2]
        #This is to remove the spaces
        regsub -all " " $name "" nameShort
        set attr [list "file $file"]
        printNode $nameShort "Class" $attr
        printArc $file $nameShort "contain"
    }
    unset class
}

# printMethod (path)
# Print the methods of the classes
proc printMethodDef {path} {

    # Open the .md file of SNavigator
    set methDefDB [dbopen nav_methdef $path.md RDONLY 0644 btree {cachesize=200000}]

    #Extract all the methods
    set methDef [$methDefDB seq]

    foreach entry $methDef {
        set className [lindex [lindex $entry 0] 0]
        set methodName [lindex [lindex $entry 0] 1]
        set file [lindex [lindex $entry 0] 3]
        set argument [lindex [lindex $entry 1] 3]
        set signature [qualifyFunction $className $methodName $argument]

        set attr [list "file $file" "belongsTo $className" "signature $methodName"]

        printNode $signature "Method" $attr
        printArc $className $signature "has_method"
    }
    unset methDef
}

# printMethod (path)
# Print the methods of the classes
proc printMethodImp {path} {

    # Open the .md file of SNavigator
    set methDefDB [dbopen nav_methdef $path.mi RDONLY 0644 btree {cachesize=200000}]

    #Extract all the methods
    set methDef [$methDefDB seq]

    foreach entry $methDef {
        set className [lindex [lindex $entry 0] 0]
        set methodName [lindex [lindex $entry 0] 1]
        set file [lindex [lindex $entry 0] 3]
        set argument [lindex [lindex $entry 1] 3]
        set signature [qualifyFunction $className $methodName $argument]

        set attr [list "file $file" "belongsTo $className" "signature $methodName"]

        printNode $signature "Method" $attr
        printArc $file $signature "fileImplementMethod"
    }
    unset methDef
}

proc printAttribute {path} {
    global ofile

    # Open the .md file of SNavigator
    set attrDeclDB [dbopen nav_attribute $path.iv RDONLY 0644 btree {cachesize=200000}]

    #Extract all the methods
    set attrDecl [$attrDeclDB seq]

    foreach entry $attrDecl {
        set className [lindex [lindex $entry 0] 0]
        set attrName [lindex [lindex $entry 0] 1]
        set file [lindex [lindex $entry 0] 3]
        set signature [qualifyVariable $className $attrName]

        set attr [list "file $file" "belongsTo $className"]
        printNode $signature "Attribute" $attr
        printArc $className $signature "has_attribute"
    }
    unset attrDecl
}

proc printMacro {path} {
```

```
    # Open the .md file of SNavigator
    set macroDB [dbopen nav_macro $path.ma RDONLY 0644 btree {cachesize=200000}]

    #Extract all the methods
    set macro [$macroDB seq]

    foreach entry $macro {
        set macroName [lindex [lindex $entry 0] 0]
        set file [lindex [lindex $entry 0] 2]
        #set signature [qualifyVariable $className $attrName]

        set attr [list "file $file"]
        printNode $macroName "Macro" $attr
        printArc $file $macroName "contain"
    }
    unset macro
}

proc printTypedef {path} {
    # Open the .md file of SNavigator
    set typedefDB [dbopen nav_typedef $path.t RDONLY 0644 btree {cachesize=200000}]

    #Extract all the methods
    set typedefs [$typedefDB seq]

    foreach entry $typedefs {
        set typeName [lindex [lindex $entry 0] 0]
        set file [lindex [lindex $entry 0] 2]

        set attr [list "file $file"]
        printNode $typeName "TypeDef" $attr
        printArc $file $typeName "contain"
    }
    unset typedefs
}

proc printGlobalVariable {path} {
    # Open the .md file of SNavigator
    set globalvarDB [dbopen nav_globalvar $path.gv RDONLY 0644 btree {cachesize=200000}]

    #Extract all the methods
    set globalvars [$globalvarDB seq]

    foreach entry $globalvars {
        set varName [lindex [lindex $entry 0] 0]
        set file [lindex [lindex $entry 0] 2]

        set attr [list "file $file"]
        printNode $varName "GlobalVariable" $attr
        printArc $file $varName "contain"
    }
    unset globalvars
}

proc printInclude {path} {
    # Open the .md file of SNavigator
    set inclDB [dbopen nav_include $path.iu RDONLY 0644 btree {cachesize=200000}]

    #Extract all the methods
    set include [$inclDB seq]

    foreach entry $include {
        set includedFile [lindex [lindex $entry 0] 0]
        set includeFrom [lindex [lindex $entry 0] 2]

        printArc $includeFrom $includedFile "include"
    }
    unset include
}

proc printInheritance {path} {

    # Open the .md file of SNavigator
    set inherDB [dbopen nav_inheritance $path.in RDONLY 0644 btree {cachesize=200000}]

    #Extract all the methods
    set inheritance [$inherDB seq]

    foreach entry $inheritance {
        set className [lindex [lindex $entry 0] 0]
        set baseClassName [lindex [lindex $entry 0] 1]

        printArc $className $baseClassName "inherit"
    }
    unset inheritance
}
```

```
# Print the references
proc printRefTo {path} {
    global ofile
    global sn_sep
    global func_def
    global meth

    # Open the .to file of SNavigator (for references to)
    set refToDB [dbopen nav_refto $path.to RDONLY 0644 btree {cachesize=200000}]

    if {$meth} {
        #Open the .mi file of SNavigator (for method implementation)
        set methImplDB [dbopen nav_methdef $path.mi RDONLY 0644 btree {cachesize=200000}]
        set methImpl [$methImplDB seq]

        #Print all the references for the methods of the classes
        foreach entry $methImpl {
            set className [lindex [lindex $entry 0] 0]
            set methodName [lindex [lindex $entry 0] 1]

            printInfoRefTo $className $methodName $refToDB
        }
    }

    if {$func_def} {
        set funcDefDB [dbopen nav_funcf $path.fu RDONLY 0644 btree {cachesize=200000}]
        set funcDef [$funcDefDB seq]

        #Print all the references for the functions
        foreach entry $funcDef {
            set name [lindex [lindex $entry 0] 0]

            printInfoRefTo "#" $name $refToDB
        }
    }
}

proc printInfoRefTo {className methodName db} {

    global ofile
    global sn_sep
    global inv
    global acc
    global mac
    global tdef


    # Query only the references of functions (not methods of classes)
    append pattern $className $sn_sep $methodName
    set refTo [$db seq $pattern]

    foreach entry $refTo {
        #Extract names and methods of caller and callee
        set callerClassName [lindex [lindex $entry 0] 0]
        set callerMethodName [lindex [lindex $entry 0] 1]
        set callerMethodType [lindex [lindex $entry 0] 2]
        set callerMethodArgument [lindex [lindex $entry 1] 0]
        set calleeClassName [lindex [lindex $entry 0] 3]
        set calleeMethodName [lindex [lindex $entry 0] 4]
        set calleeMethodType [lindex [lindex $entry 0] 5]
        set calleeMethodArgument [lindex [lindex $entry 1] 1]

        if { [isfunc $callerMethodType] && [isfunc $calleeMethodType] } {
            set caller [qualifyFunction $callerClassName $callerMethodName $callerMethodArgument]
            set callee [qualifyFunction $calleeClassName $calleeMethodName $calleeMethodArgument]

            if {$inv} { printArc $caller $callee "invocation" }

        } elseif { [isfunc $callerMethodType] && [isvar $calleeMethodType] } {
            set caller [qualifyFunction $callerClassName $callerMethodName $callerMethodArgument]
            set accessed [qualifyVariable $calleeClassName $calleeMethodName]

            if {$acc} {printArc $caller $accessed "access"}

        } elseif { [isfunc $callerMethodType] && [isclass $calleeMethodType] } {
            set caller [qualifyFunction $callerClassName $callerMethodName $callerMethodArgument]
            set accessed $calleeMethodName

            if {$acc} {printArc $caller $accessed "access"}

        } elseif { [isfunc $callerMethodType] && [ismacro $calleeMethodType] } {
            set caller [qualifyFunction $callerClassName $callerMethodName $callerMethodArgument]

            if {$mac} {printArc $caller $calleeMethodName "expansion"}
```

```
        } elseif { [isfunc $callerMethodType] && [istypedef $calleeMethodType] } {
            set caller [qualifyFunction $callerClassName $callerMethodName $callerMethodArgument]

            if {$tdef} {printArc $caller $calleeMethodName "use_type"}
        }
    }
    unset pattern
    unset refTo
}

## GXL
proc printGXLHeader {modelName} {

    global ofile

    puts $ofile "<?xml version=\"1.0\"?>"
    puts $ofile "<!DOCTYPE gxl SYSTEM \"gxl.dtd\">"
    puts $ofile "<gxl>"
    puts $ofile "\t<graph id=\"$modelName\" edgeids=\"true\">"
}

proc printGXLFooter {} {

    global ofile

    puts $ofile "\t</graph>"
    puts $ofile "</gxl>"
}

proc toXML {str} {
    regsub -all {&} $str {\&amp;} str
    regsub -all {<} $str {\&lt;} str
    regsub -all {]]>} $str {]]\&gt;} str
    return $str
}

## RSF
proc toRSF {str} {
    regsub -all {\s+} $str {} str
    return $str

}

proc printNode {nodeName nodeType attr} {

    global ofile
    global format
    global GXLIDCounter
    global IDToName
    global NameToID

    if {[info exists NameToID("$nodeName")] == 0} {
        set IDToName($GXLIDCounter) $nodeName
        set NameToID("$nodeName") $GXLIDCounter
        set ID $GXLIDCounter
        incr GXLIDCounter
    } else {
        #Node already exists, so quit
        #set ID $NameToID("$nodeName")
        return
    }

    switch $format {
     "rsf" {
            set name [toRSF $nodeName]
        puts $ofile "type $name $nodeType"
        foreach entry $attr {
            set type [lindex $entry 0]
            set val [toRSF [lindex $entry 1]]
            if {$val != ""} {
                puts $ofile "$type $name $val"
            }
        }
     }
     "gxl" {
        puts $ofile "\t\t<node id=\"\_$ID\">"
        #puts $ofile "\t\t\t<type xlink:href=\"famix.gxl#$nodeType\"/>"
        puts $ofile "\t\t\t<attr name=\"type\">"
        puts $ofile "\t\t\t\t<string>$nodeType</string>"
        puts $ofile "\t\t\t</attr>"
        puts $ofile "\t\t\t<attr name=\"name\">"
        puts $ofile "\t\t\t\t<string>[toXML $nodeName]</string>"
        puts $ofile "\t\t\t</attr>"
        foreach entry $attr {
            set type [lindex $entry 0]
```

```
            set val [lindex $entry 1]
            if {$val != ""} {
                puts $ofile "\t\t\t<attr name=\"$type\">"
                puts $ofile "\t\t\t\t<string>[toXML $val]</string>"
                puts $ofile "\t\t\t</attr>"
            }
        }
        puts $ofile "\t\t</node>"
    }
   }
}

proc printArc {srcName destName arcType} {
    global ofile
    global format
    global GXLIDCounter
    global IDToName
    global NameToID

    switch $format {
     "rsf" {
        puts $ofile "$arcType [toRSF $srcName] [toRSF $destName]"
     }
     "gxl" {

         if {[info exists NameToID("$srcName")] == 0} {
            printNode $srcName "Unkown" {}
        }
         if {[info exists NameToID("$destName")] == 0} {
            printNode $destName "Unkown" {}
        }
        set srcID $NameToID("$srcName")
        set destID $NameToID("$destName")

        puts $ofile "\t\t<edge id=\"\_$GXLIDCounter\" from=\"\_$srcID\" to=\"_$destID\">"
        #puts $ofile "\t\t\t<type xlink:href=\"famix.gxl#$arcType\"/>"
        puts $ofile "\t\t\t<attr name=\"fromNodeName\">"
        puts $ofile "\t\t\t\t<string>[toXML $srcName]</string>"
        puts $ofile "\t\t\t</attr>"
        puts $ofile "\t\t\t<attr name=\"toNodeName\">"
        puts $ofile "\t\t\t\t<string>[toXML $destName]</string>"
        puts $ofile "\t\t\t</attr>"
        puts $ofile "\t\t\t<attr name=\"type\">"
        puts $ofile "\t\t\t\t<string>$arcType</string>"
        puts $ofile "\t\t\t</attr>"
        puts $ofile "\t\t</edge>"
        incr GXLIDCounter
      }
    }
}


#Main function

if { $argc < 1 } {
    puts "Snav2Nimeta v1.0 - Converter from RedHad SourceNavigator to Famix model"
    puts ""
    puts "Syntax: hyper snav2gxl.tcl <lang> <format> <options> <SNavigator project> <output file>"
    puts "\n<lang>:"
    puts "\tjava:   language type is java"
    puts "\tc++:    language type is c++"
    puts "\n<format>:"
    puts "\tgxl:    output in GXL format"
    puts "\trsf:    output in RSF format"
    puts "\n<options>:"
    puts "\tcl+/-:   classes"
    puts "\tattr+/-: attributes"
    puts "\tmeth+/-: methods"
    puts "\tmeth_imp+/-: method implementations"
    puts "\tfl+/-:   directories, files"
    puts "\tfunc+/-: functions"
    puts "\tgv+/-:   global variables"
    puts "\tmac+/-:  macros"
    puts "\ttdef+/-: typedefs"
    puts ""
    puts "\tinh+/-:  inheritance"
    puts "\tacc+/-:  access"
    puts "\tinv+/-:  invocation"
    puts "\tincl+/-: include"
    puts "\n by default:"
    puts "    c++:  +cl +fl +func +gv +inh +acc +inv"
    puts "    java: +cl +fl +inh +acc +inv"
    puts "    tcl: fl+ func+ gv+ acc+ inv+"
    puts "    none: for no language selection
    puts "    format: gxl"
```

```
    puts "    output on stdout"
    puts ""
    puts "Example: hyper snav2nimeta.tcl java /proj/test\n"
    puts "If hyper doesn't work, please try to invoke it with full path"
    exit
}

for {set i 0} {$i < $argc} {incr i} {

    set entry [lindex $argv $i]

    switch $entry {
     "java" {
         set cl 1
        set attr 1
        set meth 1
        set fl 1
        set inh 1
        set acc 1
        set inv 1
        set ext "java"
        }
     "c++" {
        set cl 1
        set attr 1
        set meth 1
        set meth_imp 1
        set fl 1
        set func_decl 1
        set func_def 1
        set gv 1
        set inh 1
        set acc 1
        set inv 1
        set ext "c*"
          }
     "tcl" {
        set fl 1
        set func_def 1
        set gv 1
        set acc 1
        set inv 1
        set ext "tcl"
         }
     "none" {
        }
     "cl+" { set cl 1}
     "cl-" { set cl 0}
     "attr+" { set attr 1}
     "attr-" { set attr 0}
     "meth+" { set meth 1}
     "meth-" { set meth 0}
     "meth_imp+" { set meth_imp 1}
     "meth_imp-" { set meth_imp 0}
     "fl+" { set fl 1}
     "fl-" { set fl 0}
     "func_decl+" { set func_decl 1}
     "func_decl-" { set func_decl 0}
     "func_def+" { set func_def 1}
     "func_def-" { set func_def 0}
     "gv+" { set gv 1}
     "gv-" { set gv 0}
      "mac+" { set mac 1}
     "mac-" { set mac 0}
     "tdef+" { set tdef 1}
     "tdef-" { set tdef 0}
      "inh+" { set inh 1}
     "inh-" { set inh 0}
     "acc+" { set acc 1}
     "acc-" { set acc 0}
     "inv+" { set inv 1}
     "inv-" { set inv 0}
     "incl+" { set incl 1}
     "incl-" { set incl 0}
     "gxl" {set format "gxl"}
     "rsf" {set format "rsf"}
     default {
         if {$project_path == ""} {
            #Set the project file
             set project_path [lindex $argv $i]

            regsub -all {\.proj} $project_path {} tmp
            set index [expr [string last "/" $tmp] - 1]
                set project_dir [string range $tmp 0 $index]
            incr index 2
                set project_name [string range $tmp $index end]
```

```
        } else {
            #Set the output file
            set output [lindex $argv $i]
            set ofile [open $output w]
        }

      }
    }
}

########## Main function #################

#Set the complete path to the SNavigator project dir
set project_path $project_dir/.snprj/${project_name}

puts stderr "\nExtracting information from the project: $project_path"

if {$format == "gxl"} { printGXLHeader $project_name}

#Extract the file system structure of the source files
if {$fl} {
    puts stderr "Extracting the files..."
    printFiles $project_path
    puts stderr "   ...done."
}

#Print the file inclusion
if {$incl} {
    puts stderr "Extracting the include relationship..."
    printInclude $project_path
    puts stderr "   ...done."
}

#Extract the function definitions
if {$func_def} {
    puts stderr "Extracting the function definitions..."
    printFuncDef $project_path
    puts stderr "   ... done."
}

#Extract the function declaration
if {$func_decl} {
    puts stderr "Extracting the function declarations..."
    printFuncDecl $project_path
    puts stderr "   ... done."
}

#Extract the class, methods and attributes
if {$cl} {
    #Extract the classes
    puts stderr "Extracting the classes..."
    printClass $project_path
    puts stderr "   ... done."
}

if {$meth} {
    #Extract the methods
    puts stderr "Extracting the methods..."
    printMethodDef $project_path
    puts stderr "   ...done"

}
if {$meth_imp} {
    #Extract the methods implementation
    puts stderr "Extracting the method implementation..."
    printMethodImp $project_path
    puts stderr "   ...done"

}

if {$attr} {
    #Extract the attributes
    puts stderr "Extracting the attributes..."
    printAttribute $project_path
    puts stderr "   ...done"
}

#Extract the typedefs
#Be careful when enabling typedefs because there
#can be a problem with uniqueNames. For example,
#a global variable and a typedef can have the same name
if {$tdef} {
    puts stderr "Extracting the typedefs..."
    printTypedef $project_path
    puts stderr "   ...done"
}
```

```
#Extract the macros
if {$mac} {
    puts stderr "Extracting the macros..."
    printMacro $project_path
    puts stderr "   ...done"
}

#Extract the global variables
if {$gv} {
    puts stderr "Extracting the global variables..."
    printGlobalVariable $project_path
    puts stderr "   ...done"
}

#Extract the inheritances
if {$inh} {
    puts stderr "Extracting the inheritances..."
    printInheritance $project_path
    puts stderr "   ...done"
}

#Extract the refrences
if {$inv || $acc || $mac || $tdef} {
    puts stderr "Extracting the references..."
    printRefTo $project_path
    puts stderr "   ... done."
}

if {$format == "gxl"} { printGXLFooter }

#Close the output file
if {$output != ""} {
    close $ofile
}

exit
```

A.1.2 EXTRACT.PY

```
#!/usr/bin/python

# Example of a an extraction script

# Directory to parse
dir = "/system"

# Location of the script for removing the comments
stripCommand = "strip-comments-file.pl"


import os, re, fileinput, string, shutil, tempfile, sys, time
from fnmatch import fnmatch
from os.path import *


def detect(dir, patterns):
""" Functions for traversing the directory tree """
        oldDir = os.getcwd()
        os.chdir(dir)

        if (not exists(dir)):
                print >> sys.stderr, "ERROR: Specify an existing path\n"
                return

        walk("./", visit, patterns)

        os.chdir(oldDir)

def visit(patterns, dirname, names):
        parentDir = normpath(dirname)

        for name in names:
                fullName = normpath(join(parentDir, name))

                for pattern in patterns:
                        pattern(parentDir, fullName, name)


def pRSF(rel, src, dst):
""" Print the tuple in RSF """
        print '%s "%s" "%s"' %(rel,src,dst)
```

```
def printDir(parentDir, fullName, name):
""" Print the contain_dir relation """
        if (isdir(fullName)):
                print 'contain_dir "%s" "%s"' % (parentDir, fullName)
                pRSF('type', parentDir, 'Directory')
                pRSF('type', fullName, 'Directory')

        return

def printFile(parentDir, fullName, name):
""" Print the contain_file relation """
        if ( not isfile(fullName)):
                return

        if ( (not fnmatch(name, "*.c*")) & (not fnmatch(name, "*.h*")) ):
                return

        if (isfile(fullName)):
                pRSF('contain_file', parentDir, fullName)
                pRSF('type', parentDir, 'Directory')
                pRSF('type', fullName, 'File')

        return

detectTask = re.compile("^\s*(\S*_TASK)\s*")

def printTask(parentDir, fullName, name):
""" Detect the OS tasks from """
        if (not fnmatch(name,"tasks.h")):
                return

        for line in fileinput.input(fullName):
                result = re.match(detectTask, line)

                if result != None:
                        task = result.groups()
                        print 'type "%s" Task' % (task)
                else:
                        continue

detectInitFunc = re.compile("^\s*\{\s*(\w+)\s*,\s*\d+\s*,\s*\d+\s*,\s*(\w+)\s*\},")

def printInitFunction(parentDir, fullName, name):
""" Detect the initialization functions """
        if (not fnmatch(name,"init.h")):
                return

        for line in fileinput.input(fullName):
                result = re.match(detectInitFunc, line)

                if result != None:
                        (obj, func) = result.groups()

                        pRSF('functionInitObject', func, obj)
                        pRSF('type', func, 'Function')
                else:
                        continue


#################
#Calculate the type from the object name
objectNames = (
(re.compile("PN_OBJ_\S+_APPL"), "Application"),
(re.compile("PN_OBJ_\S+_DELEG"), "Delegate"),
(re.compile("PN_OBJ_\S+_SERV"), "Server"),
(re.compile("PN_OBJ_\S+_CM"), "CommManager"),
)

def fromObjectIDToType(name):
        for (pattern, objectType) in objectNames:
                result = re.match(pattern, name)
                if result != None:
                        return objectType

        return None


###########################
#Code patterns

#Definition of pattern handelrs

messageID = ""
```

```
def processSendMsg(fullName, list):
        print 'fileSendMsg "%s" "%s"' %(fullName, list)

def processInitialize(fullName, list):
        print 'fileInitObject "%s" "%s"' %(fullName, list)

def processInclude(fullName, list):
        print 'fileIncludeFile "%s" "%s"' %(fullName, list[0])

def processDefFunction(fullName, list):
        pRSF('fileDefineFunction', fullName, list)
        pRSF('type', list, 'Function')

def processFunctionCall(fullName, list):
        pRSF('fileCallFunction', fullName, list)
        pRSF('type', list, 'Function')

#Definition of the pattern table

codePatterns = (

( re.compile("^[^,]*send_msg.*(ID_\w+)\s*,.*,.*\)"), processSendToServer),
( re.compile("^[^,]*connect_server.*(ID_\w+)"), processSendToServer),
( re.compile("^[^,]*send_layer\s*\(.*,.*,(ID_\w+)"), processSendToServer),
( re.compile("^[^,](\w+_Initialize)\s*(ID_\w+)"), processInitialize),
( re.compile("#include\s*[\"|\<](\S*)[\"|\>]"), processInclude),
( re.compile("^\s*void\s+(\w+)\s*\(.*\)"), processDefFunction),
( re.compile("^\s*int\s+(\w+)\s*\(.*\)"), processDefFunction),
( re.compile("^\s*objref\s+(\w+)\s*\(.*\)"), processDefFunction),
( re.compile("\(\s*objref\s*\)\s*(\w+)\s*\([^;]*;"), processFunctionCall),
)

def detectCodePatterns(parentDir, fullName, name):
""" Detect the code patterns """
        if ( not isfile(fullName)):
                return
        if ( (not fnmatch(name, "*.c*")) ):
                return

        # Remove the comments from the file.
        tmpFile = tempfile.mktemp()

        shutil.copy(fullName,tmpFile)
        os.spawnlp(os.P_WAIT,'perl','perl',stripCommand,tmpFile)

        for line in fileinput.input(tmpFile):
                for (pattern, handler) in codePatterns:
                        results = re.findall(pattern,line)
                        if results != None:
                                for result in results:
                                        handler(fullName, result)


#######################################
#Define the list of patterns to execute

patterns = (
        printDir,
        printFile,
        printTask,
        printInitFunction,
        detectCodePatterns,
)


#######################################
# Main

if (not exists(dir)):
        print >> sys.stderr, "ERROR: Invalid path\n"
        exit

if (not exists(stripCommand)):
        print >> sys.stderr, "ERROR: Invalid strip command\n"
        exit

#Detect the patterns in the directory
detect(dir, patterns)
```

A.1.3 STRIP-COMMENTS-FILE.PL

```
#!/usr/bin/perl -w
```

```
#-------------------------------------------------------------------------
# Name: strip-comments-file 1.0
#
# Synopsis: strip-comments-file <file>
#
# Brief: Removes all the comments in C/C++ files.
#
# Description:
#     The script removes all the comments that are in the C/C++ style:
#          -single line comment in the format: // ...
#          -multiple line comment in the format: /* ... */
#     The script removes all the tabulation charactes
#     The script also puts all the statements on one line
#     The subdirectories are recurivelly processed.
#
# Example:
#
#     strip-comments-file a.c
#
#-------------------------------------------------------------------------

if (!($ARGV[0])) {
        die "! Syntax: strip-comments-file <file name>";
}

$name = $ARGV[0];


$_ = $name;

open (FILE, $name) or die "! Unable to open the file $name\n";

undef $/;
$_ = <FILE>;

close FILE;

# 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;

# remove tab
s/\t/ /g;


# Add ; to # statements, and remove it later
s/^(#.*)$/$1;/gm;

# Remove the space after;
s/(;|{|})\s*$/$1/gmx;

# Put all the statements on one line
s/([^;{}])\n/$1/gx;

# Remove the ; at the end of # statements
s/^(#.*);$/$1/gm;

open (FILE, ">$name") or die "! Unable to open file $name for writing\n";

print FILE;

close FILE;
```

A.2 ABSTRACTION SCRIPTS

A.2.1 NIMETA MODULE

```
#!/usr/bin/perl -w

#-------------------------------------------------------------
```

```
# Name: nimeta.py
#
# Brief: Nimeta extension for Python
#
# Description:
#           It provides several relational algebra operators
#           that are not available in kjBuckets
#
#-----------------------------------------------------------

import fileinput, sys, re, os

from kjbuckets import *
from string import *

def cproduct(X, Y):
        """ Cartesian product of X and Y
                X, Y are a kjSet
                result is a kjGraph

        """
        result = kjGraph()

        for x in X.items():
                for y in Y.items():
                        result.add(x,y)

        return result

def setRimage (S, k):
        """ Create a graph with the keys taken from S and the value k
                S is a KjSet
                k is a value
                result is a kjGraph

        """
        result = kjGraph()

        for node in S.items():
            result.add(node, k)

        return result

def setLimage (k, S):
        """ Create a graph with the key k and the values taken from S
                k is a value
                S is a kjSet
                result is a kjGraph of size size(S)
        """
        result = kjGraph()

        for node in S.items():
            result.add(k, node)

        return result

def dom(R):
        """ Domain of the relation R """
        return kjSet(R.keys())

def ran(R):
        """ Range of the relation R """

        return kjSet(R.values())

def car(R):
        """ Carrier of the relation R """

        return kjSet(R.keys()) + kjSet(R.values())

def ident(R):
        """ Identity of the relation R """

        return (kjSet(car(R))).ident()

def lImage(R, y):
        """ Left image of R
            Returns all the keys of R that have value y"""

        return kjSet((R*kjSet([y])).keys())

def rImage(R, x):
        """ Righ image of R
            Returns all the values of R that have the key x"""

        return kjSet((kjSet([x])*R).values())
```

```
def lProj(R, Y):
        """ Left projection of R over Y
            Returns all the keys of R that have values in Y"""

        return kjSet((R*kjSet(Y)).keys())

def rProj(R, X):
        """ Righ projection of R over X
            Returns all the values of R that have keys in X"""

        return kjSet((kjSet(X)*R).values())

def domRest(R, Do):
        """ Domain restriction"""

        return kjSet(Do)*R

def ranRest(R, Ra):
        """ Range restriction"""

        return R*kjSet(Ra)

def carRest(R, S):
        """ Carrier restriction"""

        return kjSet(S)*R*kjSet(S)


def domExcl(R, Do):
        """ Domain exclusion"""

        return domRest(R, kjSet(R)- kjSet(Do))

def ranExcl(R, Ra):
        """ Range exclusion"""

        return ranRest(R, kjSet(R.values()) - Ra)

def carExcl(R, S):
        """ Carrier exclusion"""

        return carRest(R, car(R)-S)

def power(R, n):
        """ Power"""
        if n <= 0:
                return ident(R)
        result = R
        for i in range(n-1):
                result = result * R
        return result

def rtclosure(R):
        """ Reflexive transitive closure"""

        return R.tclosure() + power(R, 0)

def top(R):
        """ Top of tree R"""
        return (dom(R) - ran(R))

def bottom(R):
        """ Bottom of tree R"""

        return (ran(R) - dom(R))

def treduction(R):
        """ Transitie Reduction"""

        return (R - (R * R.tclosure()))

def cycleFree(R):
        """ Check if R is cycle free"""
        if len(ident(R)& R.tclosure()) > 0:
                return 0
        else:
                return 1

def cycles(R):
        """ The cycles of R"""
        return kjSet(ident(R) & R.tclosure())

def lift(R, C):
        """ Lift R over C"""
```

```
        D = C.tclosure()
        A = (~C).tclosure()
        result = D*R*A - D - A
        result = result - ident(result)

        return result

def lLift(R, C):
        """ Left lift of R over C"""

        D = C.tclosure()
        result = D*R - D
        result = result - ident(result)

        return result

def rLift(R, C):
        """ Right lift of R over C"""

def fullLift(R, C):
        """ Full lift of R over C"""

        D = C.tclosure()
        A = (~C).tclosure()
        result = D*R*A + D*R + R*A - D - A
        result = result - ident(result)

        return result

def lowering(R,C):
        """ Lowering of R over C"""

        D = C.tclosure()
        A = (~C).tclosure()
        result = A*R*D - D - A
        result = result - ident(result)

        return result

def readGraphRSF(fileList=None):
        """Read in the RSF file"""

        relsIn = {}

        RSFPattern = re.compile(r'^\s*\"?((?:(?<=\")[^\"]+)|\S+)\"?\s+\"?((?:(?<=\")[^\"]+)|\S+)\" \
        ?\s+\"?((?:(?<=\")[^\"]+)|\S+)\"?\s*$')

        # Read lines from input
        for line in fileinput.input(fileList):

                result = re.match(RSFPattern, line)

                if result != None:
                        try:
                                [type, source, target] = result.groups()[0:3]
                        except ValueError:
                                pass
                else:
                        continue

                if relsIn.has_key(type):
                        relsIn[type].add(source, target)
                else:
                        relsIn[type] = kjGraph( [ (source, target) ] )

        return relsIn

def writeGraphRSF(r, Root=1):
        """Print graph r in RSF format on stdout.
           r is an hash that contain the relations of the graph. The key
             is the relation type. The value is a kjGraph.
           If Root=1 it assumes that the level and type relations are
           defined and it prints an unstructured RSF file that can be loaded
           Rigi. """

        if Root:
                if r.has_key('level'):
                        TopNodes = kjSet(r['level'].keys()) - kjSet((~r['level']).keys())
                        for node in TopNodes.items():
                                r['level'].add('Root', node)

                        if r.has_key('type'):
                                OrphanNodes = kjSet(r['type']) - kjSet(r['level']) - kjSet(~r['level'])
                                for node in OrphanNodes.items():
                                        r['level'].add('Root', node)
```

```
                print "type Root Syntactic"
            elif r.has_key('type'):
                r['level'] = kjGraph()
                for node in r['type'].keys():
                    r['level'].add('Root', node)

                print "type Root Syntactic"

    nodesOut = kjSet()

    # Print all relations
    for (type, relation) in r.items():
        if type != 'type':
            for (source, target) in relation.items():
                # Print type, source and target separated by tabs
                nodesOut.add(source)
                nodesOut.add(target)
                print "%s \"%s\" \"%s\"" % (type, source, target)

    if r.has_key('type'):
        for (node) in nodesOut.items():
            try:
                print "type \"%s\" \"%s\"" % (node, r['type'][node])
            except KeyError:
                pass

def printRel(rel, name = ""):
    """Print the relation rel"""

    for (a,b) in rel.items():
        print name, a, b
```

A.2.2 COLLAPSE_CLASSES.RCL

```
#Group methods and attributes of classes

proc collapse_classes { } {
    rcl_select_type Class

    set winnodes [rcl_select_get_list]
    rcl_select_none

    foreach node $winnodes {

        rcl_select_id $node
        set name [rcl_get_node_name $node]

        rcl_set_node_name $node $name+

        select_neighbors has_method out 1
        select_neighbors has_attribute out 1
        if {[rcl_collapse Class $name] == 0} { return }
    }
}
```

A.2.3 COLLAPSE_DIRS.RCL

```
# Collapse all the selected directories. The parameter stype is the
#target type for the collapsed nodes.
```

```
proc collapse_dirs { stype } {

    set winnodes [rcl_select_get_list]
    rcl_select_none

    foreach node $winnodes {
        collapse_dir $node $stype
    }
}

# Recursively collapse all the subdirectories

proc collapse_dir {node stype} {

    set name [rcl_get_node_name $node]

    set arcs [rcl_node_get_arclist $node contain_dir out 0 0]

    if {[llength $arcs] != 0} {
        foreach arc $arcs {
            set dstNode [rcl_get_arc_dst $arc]
            collapse_dir $dstNode $stype
        }
     }
     collapse_all_content_in_dir $node $stype
}

# Collapse all the nodes contained in one directory

proc collapse_all_content_in_dir {node stype} {
    rcl_select_none
    rcl_select_id $node

    set name [rcl_get_node_name $node]

    #Select all files
    select_neighbors contain_file out 1

    #Select all the contained nodes
    select_neighbors contain out 1

    #Select all the functions
    select_neighbors def_fn out 1

    #Select all methods and attributes of classes
    select_neighbors has_method out 1
    select_neighbors has_attribute out 1

    if {[rcl_collapse $stype "$name+"] == 0} { return }
}
```

REFERENCES

Anquetil, Nicolas, and Lethbridge, Timothy C. 1999. Recovering software architecture from the names of source files. *Journal of Software Maintenance*, **11**(3), 201–221.

Bass, L., Clements, P., and Kazman, R. 1998. *Software Architecture in Practice*. Reading, MA: Addison-Wesley Longman.

Bassil, S., and Keller, R. K. 2001. Software visualization tools: Survey and analysis. *Pages 7–17 of: 9th International Workshop on Program Comprehension (IWPC 2001), 12-13 May 2001, Toronto, Canada*. IEEE Computer Society.

Berghammer, R., Schmidt, G., and Winter, M. 2003. RELVIEW and RATH - Two systems for dealing with relations. *Theory and Applications of Relational Structures as Knowledge Instruments*, 1–16.

Booch, Grady, Rumbaugh, Jim, and Jacobson, Ivar. 1999. *The Unified Modelling Language User Guide*. Addison-Wesley.

Bosch, Jan. 2000. *Design and Use of Software Architectures: Adopting and evolving a product-line approach*. Addison Wesley.

Bosch, Jan. 2002. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. *Pages 257–271 of: Proceedings of the Second International Conference on Software Product Lines*. Springer-Verlag.

Bowman, T., Holt, R. C., and Brewster, N. V. 1999. Linux as a Case Study: Its Extracted Software Architecture. *Pages 555–563 of: Proceedings of the 1999 International Conference on Software Engineering, May 16-22, 1999, Los Angeles, CA, USA*. IEEE Computer Society.

Casais, Eduardo. 1998. Re-engineering object-oriented legacy systems. *Journal of Object-Oriented Programming*, **10**(8), 4552.

Chikofsky, Elliot J., and Cross II, James H. 1990. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, January, 13–17.

Clements, P., and Northrop, L. 2001. *Software Product Lines: Practices and Patterns*. Addison Wesley.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. 2003. *Documenting Software Architectures: Views and Beyond*. Addison Wesley.

Demeyer, Serge, Ducasse, Stéphane, and Tichelaar, Sander. 1999. Why Unified is not Universal. UML Shortcomings for Coping with Round-trip Engineering. *Pages 630–644 of:* France, Robert B., and Rumpe, Bernhard (eds), *Proceedings of the UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999*. Lecture Notes in Computer Science, vol. 1723. Springer-Verlag.

Demeyer, Serge, Tichelaar, Sander, and Ducasse, Stéphane. 2001. *FAMIX 2.1 – The FAMOOS Information Exchange Model*. Tech. rept. University of Bern.

Ebert, J., Kontogiannis, K., and Mylopoulos, J. 2001 (jan). *Interoperability of Reengineering Tools*. Seminar 01041. Schloss Dagstuhl.

Eden, Amnon H., and Kazman, Rick. 2003. Architecture, Design, Implementation. *Pages 149–159 of: Proceedings of the 25th International Conference on Software Engineering (ICSE), May 3-10, 2003, Portland, Oregon, USA*. IEEE Computer Society Press.

Eisenbarth, Thomas, and Koschke, Rainer. 2003. Locating Features in Source Code. *IEEE Transactions on Software Engineering*, **29**(3), 210–224.

Eixelsberger, W., Ogris, M., Gall, H., and Bellay, B. 1998. Software architecture recovery of a program family. *Pages 508–511 of: Proceedings of the 20th International Conference on Software Engineering, ICSE 98, April 19-25, 1998, Kyoto, Japan*. IEEE Computer Society Press.

Fahmy, Hoda, Holt, Richard C., and Cordy, James R. 2001. Wins and Losses of Algebraic Transformations of Software Architectures. *Pages 51–62 of: 16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*. IEEE Computer Society.

Faust, D., and Verhoef, C. 2003. Software Product Line Migration and Deployment. *Software Practice and Experience, to appear*.

Feijs, L. M. G., and Krikhaar, R. L. 1999. Relational Algebra with Multi-Relations. *International Journal of Computer Mathemathics*, **70**, 57–74.

Feijs, L. M. G., and van Ommering, R.C. 1999. Relation Partition Algebra - Mathematical Aspects of Uses and Part-Of Relations. *Sience of Computer Programming*, **33**, 163–212.

Feijs, L. M. G., Krikhaar, R. L., and van Ommering, R.C. 1998. A relational approach to Software Architecture Analysis. *Software Practice and Experience*, **28**(3), 371–400.

Ferenc, Rudolf, Sim, Susan Elliott, Holt, Richard C., Koschke, Rainer, and Gyimothy, Tibor. 2001. Towards a Standard Schema for C/C++. *Pages 49–58 of: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01), 2-5 October 2001, Suttgart, Germany*. IEEE Computer Society.

Ferenc, Rudolf, Beszedes, Arpad, Tarkiainen, Mikko, and Gyimothy, Tibor. 2002. Columbus - Reverse Engineering Tool and Schema for C++. *Pages 172–181 of: Proc. of the 18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3-6 October 2002, Montreal, Quebec, Canada*. IEEE Computer Society.

Finnigan, P. J., Holt, R. C., I, I. Kalas, Kerr, S., Kontogiannis, K., Müller, H. A., Mylopoulos, J., Perelgut, S. G., Stanley, M., and Wong, K. 1997. The software bookshelf. *IBM Systems Journal*, **36**(4), 564–593.

Fowler, Martin. 2003. Who Needs an Architect ? *IEEE Software*, **20**(5), 11–13.

Gacek, Cristina, Abd-Allah, Ahmed, Clark, Bradford, and Boehm, Barry. 1995. On the Definition of Software System Architecture. *In: ICSE 17 Software Architecture Workshop*.

Garlan, D., and Perry, D. 1995. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, **21**(4).

Garlan, David, Allen, Robert, and Ockerbloom, John. 1995. Architectural Mismatch, or, Why it's hard to build systems out of existing parts. *IEEE Software*, **12**(6), 17–26.

Girard, J.-F. 2003. *SARA: A Semi-automatic Architecture Reconstruction Approach*. Ph.D. Dissertation, Fraunhofer Institute for Experimental Software Engineering.

Godfrey, Michael W., and Lee, Eric H. S. 2000. Secrets from the Monster: Extracting Mozilla's Software Architecture. *In: Proc. of the Second Intl. Symposium on Constructing Software Engineering Tools (CoSET'00)*. IEEE Computer Society Press.

Graham, Paul. 2003 (March). *The Hundred-Year Language*. Invited talk at the Python Conference 2003 (PyCon DC 2003), March 26-28 2003, Washington DC.

Gschwind, Thomas, and Oberleitner, Johann. 2003. Improving Dynamic Data Analysis with Aspect-Oriented Programming. *Pages 259–268 of: Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 2003), Benevento, Italy, March 2003*. IEEE Computer Society.

Gschwind, Thomas, Oberleitner, Johann, and Pinzger, Martin. 2003. Using Run-Time Data for Program Comprehension. *Pages 245–250 of: Proceedings of the 11th International Workshop on Program Comprehension, May 2003*. IEEE Computer Society.

Guo, George Yanbing, Atlee, Joanne M., and Kazman, Rick. 1999. A Software Architecture Reconstruction Method. *Pages 15–34 of:* Donohoe, Patrick (ed), *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1), 22-24 February 1999, San Antonio, Texas, USA*, vol. 140. Kluwer Academic Publisher.

Harris, David R., Reubenstein, Howard B., and Yeh, Alexander S. 1995. Reverse engineering to the architectural level. *Pages 186–195 of: Proceedings of the 17th International Conference on Software Engineering, April 23-30, 1995, Seattle, Washington, USA*. ACM Press.

Hassan, A. E., and Holt, R. C. 2000. A Reference Architecture for Web Servers. *In: Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE Computer Society Press.

Hofmeister, C., Nord, R. L., and Soni, D. 2000. *Applied Software Architecture*. Object Technology Series, Addison Wesley.

Hofmeister, Christine, Nord, Robert L., and Soni, Dilip. 1999. Describing Software Architecture with UML. *Pages 145–160 of:* Donohoe, Patrick (ed), *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1), 22-24 February 1999, San Antonio, Texas, USA*. IFIP Conference Proceedings, vol. 140. Kluwer Academic Publisher.

Holt, R. C., Winter, A., and Schürr, A. 2000a. GXL: Toward a Standard Exchange Format. *Pages 162–171 of: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00), 23-25 November 2000, Brisbane, Australia*. IEEE Computer Society.

Holt, Ric. 2001 (August). Software Architecture as a Shared Mental Model. *In: ASERC Workhop on Software Architecture*. University of Alberta.

Holt, Richard C. 1998. Structural Manipulations of Software Architecture using Tarski Relational Algebra. *Pages 210–219 of: Proc. of the 5th Working Conference on Reverse Engineering, WCRE '98, October 12-14, 1998, Honolulu, Hawai, USA*. IEEE Computer Society.

Holt, Richard C., Hassan, Ahmed E., Lague, Bruno, Lapierre, Sebastien, and Leduc, Charles. 2000b. E/R Schema for the Datrix C/C++/Java Exchange Format. *Pages 284–286 of: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00), 23-25 November 2000, Brisbane, Australia*. IEEE Computer Society.

Holt, Richard C., Winter, Andreas, and Schürr, Andy. 2000c. *GXL: Towards a Standard Exchange Format*. Tech. rept. 1–2000. Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz.

Jaktman, Catherine Blake, Leaney, John, and Liu, Ming. 1999. Structural Analysis of the Software Architecture. *Pages 455–470 of:* Donohoe, Patrick (ed), *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1), 22-24 February 1999, San Antonio, Texas, USA*. IFIP Conference Proceedings, vol. 140. Kluwer Academic Publisher.

Jazayeri, M., van der Linden, F., and Ran, A. 2000. *Software Architecture for Product Families*. Addison Wesley.

Jerding, Dean, and Rugaber, Spencer. 1997. Using Visualization for Architectural Localization and Extraction. *Pages 56–65 of:* Baxter, Ira D., Quilici, Alex, and Verhoef, Chris (eds), *Proceedings of the 4th Working Conference on Reverse Engineering, WCRE '97, October 6-8, 1997, Amsterdam, The Netherlands*. IEEE Computer Society.

Kazman, R., and Carrière, S. J. 1998. View Extraction and View Fusion in Architectural Understanding. *Page 290 of: Proceedings of the 5th International Conference on Software Reuse*. IEEE Computer Society.

Kazman, R., Bass, L., Abowd, G., and Webb, M. 1994. SAAM: A Method for Analyzing the Properties Software Architectures. *Pages 81–90 of: Proceedings of the 16th International Conference on Software Engineering, May 16-21, 1994, Sorrento, Italy*. IEEE Computer Society / ACM Press.

Kazman, R., Klein, M., Barbacci, M., Lipson, H., and T. Longstaf and, S.J. Carrire. 1998. The Architecture Tradeoff Analysis Method. *In: Proceedings of International Conference on Engineering of Complex Computer Systems, August 1998, Monterey, CA*. IEEE Computer Society.

Kazman, R., O'Brien, L., and Verhoef, C. 2001. *Architecture Reconstruction Guidelines*. report CMU/SEI-2001-TR-026. Carnegie Mellon University, Software Engineering Institute.

Kazman, Rick. 1996. Tool Support for Architecture Analysis and Design. *Pages 94–97 of: Joint Proceedings of the SIGSOFT '96 Workshops (ISAW-2)*. ACM.

Koppler, R. 1997. A Systematic Approach to Fuzzy Parsing. *Software Practice and Experience*, **27**(6), 637–649.

Koschke, R. 2003. Software Visualization in Software Maintenance, Reverse Engineering, and Reengineering: A Research Survey. *Journal on Software Maintenance and Evolution*, **15**(2), 87–109.

Koschke, R., and Simon, D. 2003. Hierarchical Reflexion Models. *Pages 36–45 of:* van Deursen, Arie, Stroulia, Eleni, and Storey, Margaret-Anne D. (eds), *Proc. of the 10th Working Conference on Reverse Engineering (WCRE 2003), 13-16 November 2003, Victoria, Canada*. IEEE Computer Society Press.

Koskimies, Kai, and Mössenböck, Hanspeter. 1996. Scene: using scenario diagrams and active text for illustrating object-oriented programs. *Pages 366–375 of: Proceedings of the 18th International Conference on Software Engineering, March 25-29, 1996, Berlin, Germany*. IEEE Computer Society.

Koskimies, Kai, Systä, Tarja, Tuomi, Jyrki, and Männistö, Tatu. 1998. Automated Support for Modeling OO Software. *IEEE Software*, **15**(1), 87–94.

Krikhaar, Rene. 1999. *Software Architecture Reconstruction*. Ph.D. thesis, University of Amsterdam.

Kruchten, P. B. 1995. The 4+1 View Model of architecture. *IEEE Software*, **6**(12), 42–50.

Krug, Steve. 2000. *Don't Make Me Think: A Common Sense Approach to Web Usability*. New Riders.

Kullbach, Bernt, and Winter, Andreas. 1999. Querying as an Enabling Technology in Software Reengineering. *Pages 42–50 of: 3rd European Conference on Software Maintenance and Reengineering (CSMR '99), 3-5 March 1999, Amsterdam, The Netherlands*. IEEE Computer Society.

Kuusela, Juha. 1999. Architectural Evolution. *Pages 471–478 of:* Donohoe, Patrick (ed), *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1), 22-24 February 1999, San Antonio, Texas, USA*. IFIP Conference Proceedings, vol. 140. Kluwer Academic Publisher.

Laine, Petri K. 2001. The Role of SW Architectures in Solving Fundamental Problems in Object-Oriented Development of Large Embedded SW Systems. *Pages 14–23 of: Proc. of the 2001 Working IEEE / IFIP Conference on Software Architecture (WICSA 2001), 28-31 August 2001, Amsterdam, The Netherlands*.

Lange, Danny B., and Nakamura, Yuichi. 1995 (June). Program Explorer: A Program Visualiser for C++. *In: Proceedings of the USENIX Conference on Object-Oriented Technologies*.

Lanza, Michele, and Ducasse, Stphane. 2003. Polymetric Views-A Lightweight Visual Approach to Reverse Engineering. *IEEE Trans. Softw. Eng.*, **29**(9), 782–795.

Lorentsen, Louise, Tuovinen, Antti-Pekka, and Xu, Jianli. 2001. Modelling Feature Interaction Patterns in Nokia Mobile Phones using Coloured Petri Nets and Design/CPN. *In: Proc. of the 7th Symposium on Programming Languages and Software Tools, Szeged, Hungary, June 15-16*.

Maccari, A., and Riva, C. 2001. Architectural evolution of legacy product families. *Pages 64–69 of:* van der Linden, Frank (ed), *Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain, October 3-5, 2001, Revised Papers*. Lecture Notes in Computer Science, vol. 2290. Springer.

Mancoridis, S., B.S.Mitchell, C.Rorres, Y.Chen, and E.R.Gansner. 1998. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. *Pages 45–52 of: Proceedings of the 1998 International Workshop on Program Understanding (IWPC'98), Ischia, Italy, June 1998*.

Marshall, M. S., I., I. Herman, and Melancon, G. 2001. An Object-Oriented Design for Graph Visualization. *Software: Practice & Experience*, **31**, 439–756.

Medvidovic, N., and Taylor, R.N. 2000. A Classification and Comparison Framework for Software Architecture Description languages. *IEEE Transactions on Software Engineering*, **26**(1), 70–93.

Mendonça, Nabor C. 1999. *Software Architecture Recovering for Distributed Systems*. Ph.D. thesis, University of London.

Mendonça, Nabor C., and Kramer, Jeff. 2001. An Approach for Recovering Distributed System Architectures. *Automated Software Engineering.*, **8**(3-4), 311–354.

Mens, Kim. 2000. *Automating architectural conformance checking by means of logic meta programming*. Ph.D. thesis, Departement Informatica, Vrije Universiteit Brussel.

Moonen, L. 2001. Generating Robust Parsers using Island Grammars. *Pages 13–22 of: Proc. of the Working Conf. on Reverse Engineering (WCRE)*. IEEE CS.

Morgan, Augustus De. 1860. On the syllogism, no. IV, and on the logic of relations. *Transactions of the Cambridge Philosophical Society*, **10**, 331–358.

Müller, Hausi A., Orgun, Mehmet A., Tilley, Scott R., and Uhl, James S. 1993. A Reverse Engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, **5**(4), 181–204.

Murphy, G. C., and Notkin, D. 1997. Reengineering with Relfextion Models: A Case Study. *IEEE Software*.

Murphy, G. C., Notkin, D., and Sullivan, K. J. 2001. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE CS Transactions on Software Engineering*, **27**(4), 364–380.

Nentwich, Christian, Emmerich, Wolfgang, Finkelstein, Anthony, and Zisman, Andrea. 2000. BOX: Browsing objects in XML. *Software: Practice and Experience*, **30**(15), 1661–1676.

O'Brien, Liam. 2002. *Experiences in Architecture Reconstruction at Nokia*. Tech. rept. CMU/SEI-2002-TN-004. Software Engineering Institute, Carnegie Mellon University.

P1471, IEEE. 2000. *IEEE Recommended Practice for Architectural Description of Software-intensive Systems*.

Parnas, David L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, **15**(12), 1053–1058.

Parnas, David L. 1974. On a Buzzword: Hierarchical Structure. *IFIP Congress 74*, 336–339.

Parnas, David L. 1994. Software Aging. *Pages 279–287 of: Proceedings of the 16th International Conference on Software Engineering, May 16-21, 1994, Sorrento, Italy*. IEEE Computer Society Press / ACM Press.

Peirce, C. S. 1933. Description of a notation for the logic of relatives, resulting from an amplification of the conceptions of Boole's calculs of logic. *Collected Papers of Charles Sanders Peirce* III *Exact Logic*.

Perry, Dewayne E., and Wolf, Alexander L. 1992. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, **17**(4), 40–52.

Pinzger, Martin, and Gall, Harald. 2002. Pattern-Supported Architecture Recovery. *Pages 53–64 of: Proc. of the 10th International Workshop on Program Comprehension (IWPC 2002), 27-29 June 2002, Paris, France*. IEEE Computer Society.

Pinzger, Martin, Fischer, Michael, Gall, Harald, and Jazayeri, Mehdi. 2002. Revealer: A Lexical Pattern Matcher for Architecture Recovery. *Pages 170–180 of:* van Deursen, Arie, and Burd, Elizabeth (eds), *Proc. of the 9th Working Conference on Reverse Engineering (WCRE 2002), 28 October - 1 November 2002, Richmond, VA, USA*. IEEE Computer Society.

Postma, Andre. 2003. A method for module architecture verification and its application on a large component-based system. *Information and Software Technology*, **45**, 171–194.

Pratt, Vaughan R. 1992. Origins of the Calculus of Binary Relations. *Pages 248–254 of: Proc. IEEE Symposium on Logic in Computer Science, 248-254, Santa Cruz, CA, June, 1992*. IEEE Computer Society.

Ran, A. 2000. *ARES Conceptual Framework for Software Architecture*. Addison-Wesley. Chap. 1, pages 1–30.

Richner, Tamar, and Ducasse, Stéphane. 1999. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. *Pages 13–22 of: Proceedings of the International Conference on Software Maintenance (ICSM'99), 30 August - 3 September, 1999, Oxford, England, UK*. IEEE Computer Society.

Riva, C., Xu, J., and Maccari, A. 2001. *Architecting and reverse architecting in UML*. First International Workshop on Describing Software Architectures with UML, the 3rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada.

Riva, Claudio. 2000. Reverse Architecting: an Industrial Experience Report. *In: Proceedings of the 7th Working Conference on Reverse Engineering (WCRE2000) , 23-25 November 2000, Brisbane, Australia*. IEEE Computer Society Press.

Riva, Claudio. 2002. Architecture Reconstruction in Practice. *In:* Bosch, Jan, Gentleman, W. Morven, Hofmeister, Christine, and Kuusela, Juha (eds), *Software Architecture: System Design, Development and Maintenance, IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture (WICSA3), August 25-30, 2002, Montral, Qubec, Canada*. IFIP Conference Proceedings, vol. 224. Kluwer.

Riva, Claudio, and Del Rosso, Christian. 2003. Experiences with Software Product Family Evolution. *Page 161 of: Proc. of the 6th International Workshop on Principles of Software Evolution (IWPSE 2003), 1-2 September 2003, Helsinki, Finland*. IEEE Computer Society.

Riva, Claudio, and Rodriguez, Jordi Vidal. 2002. Combining Static and Dynamic Views for Architecture Reconstruction. *In: Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR 2002), 11-13 March 2002 in Budapest, Hungary*. IEEE Computer Society Press.

Riva, Claudio, and Yang, Yaojin. 2002. Generation of Architectural Documentation Using XML. *Pages 151–160 of:* van Deursen, Arie, and Burd, Elizabeth (eds), *Proc. of the 9th Working Conference on Reverse Engineering (WCRE 2002), 28 October - 1 November 2002, Richmond, VA, USA*. IEEE Computer Society.

Riva, Claudio, Selonen, Petri, Systä, Tarja, Tuovinen, Antti-Pekka, Xu, Jianli, and Yang, Yaojin. 2004a. Establishing a Software Architecting Environment. *In: Proc. of 4th*

Working IEEE / IFIP Conference on Software Architecture (WICSA 2004), 12-15 June 2004, Oslo, Norway. IEEE Computer Society.

Riva, Claudio, Selonen, Petri, Systä, Tarja, and Xu, Jianli. 2004b. UML-based Reverse Engineering and Model Analysis Approaches for Software Architecture Maintenance. *In: Proc. of the The 20th IEEE International Conference on Software Maintenance, Chicago, Illinois, USA, September 11th - 17th 2004*. IEEE Computer Society.

Robbins, Jason E., Medvidovic, Nenad, Redmiles, David F., and Rosenblum, David S. 1998. Integrating architecture description languages with a standard design method. *Pages 209–218 of: Proceedings of the 20th International Conference on Software Engineering, ICSE 98, April 19-25, 1998, Kyoto, Japan*. IEEE Computer Society.

Rumpe, B., Schoenmakers, M., Radermacher, A., and Schürr, A. 1999. UML + ROOM as a Standard ADL? *In:* Titsworth, F. (ed), *Proc. ICECCS'99 Fifth IEEE International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society.

Schmidt, G., and Ströhlein, T. 1993. *Relations and Graphs*. Berlin: Springer-Verlag.

Schroeder, W., Martin, K., and Lorensen. 1998. *The Visualization Toolkit, 2nd edition*. Prentice Hall.

Schroöder, E. 1895. Vorlesungen über die Algebra der Logik (Exakte Logik). *In:* Teubner, B.G. (ed), *Algebra und Logic der Relative*, vol. 10. Leipzig.

Selonen, Petri, and Xu, Jianli. 2003. Validating UML models against architectural profiles. *Pages 58–67 of: Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*. ACM Press.

Shaw, M., and Garlan, D. 1996. *Software Architecture: Perspective on an Emerging Discipline*. Prentice Hall.

Sillanpää, Matti. 2004. *Visualizing Reverse Architected Software Models with an Open Visualization Toolkit*. M.Phil. thesis, Helsinki University of Technology.

Stewart, David B. 1999. 30 Pitfalls for Real-Time Software Developers - Part 2. *Embedded Systems Programming Magazine*, **12**(12), 74–86.

Stoermer, C., O'Brien, L., and Verhoef, C. 2002. Practice Patterns for Architecture Reconstruction. *Pages 151–160 of:* van Deursen, Arie, and Burd, Elizabeth (eds), *Proc. of the 9th Working Conference on Reverse Engineering (WCRE 2002), 28 October - 1 November 2002, Richmond, VA, USA*. IEEE Computer Society.

Störrle, H. 1999. Archtiectural Modelling with the Unified Modelling Language. *In: Nordic Workshop on Software Architecture (NOSA 99)*.

Systä, Tarja. 2000. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. Report A-2000-4, University of Tampere, Dept. of Computer and Information Sciences,.

Tarski, Alfred. 1941. On the calculus of relations. *Journal of Symbolic Logic*, **6**, 73–89.

Telea, Alexandru, Maccari, Alessandro, and Riva, Claudio. 2002. An Open Visualization Toolkit for Reverse Architecting. *Pages 3–10 of: Proc. of the 10th International Workshop on Program Comprehension (IWPC 2002), 27-29 June 2002, Paris, France*. IEEE Computer Society.

Tichelaar, Sander. 2001. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. Ph.D. thesis, University of Berne.

Tilley, S., Paul, S., and Smith, D. B. 1996. Towards a Framework for Program Understanding. *Pages 19–28 of: 4th International Workshop on Program Comprehension (IWPC '96), March 1996, Berlin Germnay*. IEEE Computer Society.

Turner, C. Reid, Fuggetta, Alfonso, Lavazza, Luigi, and Wolf, Alexander L. 1999. A conceptual basis for feature engineering. *The Journal of Systems and Software*, **49**(1), 3–15.

van Deursen, Arie, and Kuipers, Tobias. 1999. Identifying objects using cluster and concept analysis. *Pages 246–255 of: Proceedings of the 21st international conference on Software engineering*. IEEE Computer Society Press.

van Deursen, Arie, Hofmeister, Christine, Koschke, Rainer, Moonen, Leon, and Riva, Claudio. 2004. Symphony: View-Driven Software Architecture Reconstruction. *Pages 122–132 of: Proc. of 4th Working IEEE / IFIP Conference on Software Architecture (WICSA 2004), 12-15 June 2004, Oslo, Norway*. IEEE Computer Society.

Wernecke, J. 1993. *The Inventor Mentor: Programming Object-Oriented 3D Graphics*. Addison-Wesley.

Wong, Kenny, Corrie, Brian D., Müller, Hausi A., Storey, Margaret-Anne D., Tilley, Scott R., and Walkowicz, Jacek. March 1993. *Rigi User's Manual for Open Look*.

Yang, Yaojin, and Xu, Jianli. 2003. Encoding Informal Architectural Descriptions with UML: an Experience Report. *In:* Stevens, Perdita, Whittle, Jon, and Booch, Grady (eds), *Proc. of the 6th International Conference on the Unified Modeling Language-(UML 2003), San Francisco, CA, USA, October 20-24, 2003*. Lecture Notes in Computer Science, vol. 2863. Springer.

CURRICULUM VITÆ ET STUDIORUM

Claudio Riva was born in Merate (LC), Italy, on the 4th July 1973. In 1992 he received the scientific diploma from the Liceo M. G. Agnesi in Merate. The same year he started the five years course in Telecommunication Engineering at the Politecnico di Milano, Italy. In 1998, he obtained the degree of Master of Science (Diploma di Laurea) in Telecommunication Engineering with the final grade of 100/100. The Master's Thesis was carried out at the Vienna University of Technology and titled: "Visualizing Software Release Histories: The Use of Color and Third Dimension". In 1999, he enrolled in the postgraduate studies (Ph.D.) at the Vienna University of Technology where he completed to write the present dissertation in 2004. In 1998, he joined the Software Architecture Group at the Nokia Research Center in Helsinki, Finland, as research engineer. His main research interests are in the field of software engineering, software architecture, reverse engineering, software evolution, programming languages, product lines, information visualization and distributed computing. He has participated in several European research projects and numerous international conferences and workshops. He is author and co-author of 16 conference articles and one journal paper. His tutorial on "Software Architecture Reconstruction" has been presented at the International Conference on Software Maintenance (ICSM) 2002, European Conference on Software Engineering (ESEC/FSE) 2003, International Conference on Software Engineering (ICSE) 2004 and serves for Nokia internal training. He has been a member of the program committee for the International Workshop on Principles of Software Evolution (IWPSE) 2003, International Workshop on Program Comprehension (IWPC) 2003, 2004, Working Conference on Reverse Engineering 2002 and IEEE Transaction on Software Engineering. He was the Program Chair of the European Conference on Software Maintenance and Reengineering (CSMR) in 2004. He is currently working as a Senior Research Engineer at the Nokia Research Center.