Universität Stuttgart
Fakultät Informatik

Studiengang: Softwaretechnik

Prüfer: Prof. Dr. rer. nat./ Harvard Univ. Erhard Plödereder

Betreuer: Dr. rer. nat. Rainer Koschke (Universität Stuttgart)
Dipl. Inf. Gerd Bleher (Agilent Technologies)

Beginn am: 03. Januar 2002

Beendet am: 03. Juli 2002

CR-Klassifkation: D.2.7, D.2.11, K.6.3

Diplomarbeit Nr. 1987

Process Models for the Reconstruction of Software Architecture Views

Jens Knodel

Institut für Informatik
Universität Stuttgart
Breitwiesenstraße 20-22
D-70565 Stuttgart

Abstract

In the literature, there are a lot of techniques for the reconstruction of software architectures. But they are lacking precise instructions how to apply and combine these techniques depending on the objectives and the boundary conditions of the reconstruction.

Therefore, this diploma thesis introduces two new process models being developed for the reconstruction of software architecture views. The one, called “Design Hypothesis”, accomplishes the goal of creating an up-to-date high-level design model. The other named “Bottom-Up Analysis” establishes a segmentation of the software architecture into different layers. To achieve these goals, both process models integrate reverse engineering techniques and several tools. In a case study with an industrial partner, the process models are evaluated for a particular legacy software system and the results are discussed.

Keywords:

software architecture, architecture recovery, architecture view, process model, reverse engineering, reflexion model, program slicing, concept analysis, design hypothesis, bottom-up analysis

Zusammenfassung

In der Literatur existieren eine Reihe von Techniken zur Rekonstruktion von Software-Architekturen. Genaue Anleitung, wie diese Techniken abhängig von Zielen und Randbedingungen angewandt und kombiniert werden, sind jedoch nur unzureichend vorhanden.

Deswegen werden in dieser Diplomarbeit zwei neue Prozessmodelle zur Rekonstruktion von Software-Architekturen präsentiert. Das eine Prozessmodell, genannt “Design Hypothesis”, hat das Ziel, ein aktuelles Modell der Architektur zu erstellen. Das andere Modell mit dem Namen “Bottom-Up Analysis” beschäftigt sich mit Aufteilung der Architektur in verschiedene Ebenen. Um diese Ziele zu erreichen, integrieren beide Prozessmodelle Reverse Engineering Techniken und mehrere Tools. In einer Fallstudie bei einem industriellen Partner werden die Prozessmodelle an einem Software-Altsystem evaluiert und die Ergebnisse werden präsentiert.

Schlüsselwörter:

Software-Architektur, Architektur Wiederherstellung, Architektursicht, Prozessmodelle, Reverse Engineering, Reflexion Modell, Program Slicing, Begriffsanalyse, Design Hypothesis, Bottom-Up Analyse

Acknowledgements

I wish to thank all the people who directly supported the writing of this thesis.

I wish to acknowledge the assistance and the contributions of the developers at Agilent Technologies, Böblingen. Thank you for your cooperation, especially Wolfgang Horn and Hans Bijker.

I also wish to thank Jörg Franke for proofreading services.

And I would like to give a special thanks to my tutors, Gerd Bleher and Rainer Koschke. Both contributed to this thesis with proofreading, excellent ideas in several discussions and for being available in problem situations.

The text of this thesis was written using Adobe Framemaker 5.5.6, the process model diagrams have been drawn with Microsoft Visio 2000.

Table of Contents

Table of Figures

List of Abbreviations

ADT	Abstract Data Type
DPS	Device Power Supply
DUT	Device under Test
GUI	Graphical User Interface
IC	Integrated Circuit
IDEF	Integrated Definition Language
IML	InterMediate Language
KLOC	Kilo Lines of Code
LOC	Lines of Code
MCD	Message Constructor/Distributor
MVC	Model-View-Controller
PAC	Parser and Constructor
RFG	Resource-Flow-Graph
SADT	Structured Analysis and Design Technique
SOC	Systems-on-a-Chip
SOCBU	Systems-on-a-Chip Business Unit
UML	Unified Modeling Language

Chapter 1 Introduction

ABSTRACT: After introducing the problem field of software maintenance, this chapter will discuss architecture recovery in general. Then the approach taken is presented. This chapter will conclude with an overview of this diploma thesis.

1.1 Software Maintenance

Lehman and Belady (see also [Lehman 1985]) examined the growth and evolution of a number of large software systems and proposed the so-called five Lehman's laws that were derived from these measurements. [Sommerville 1989] (p. 15) states that two of these laws (hypotheses really) appear to be valid, these are:

- The law of continuing change: A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.
- The law of increasing complexity: As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.

In order to manage the effects of this laws, a software system (also called legacy system once released) has to be maintained. [IEEE Std 610.12-1990] defines maintenance as the process of modifying a software system or component after delivery or to correct faults, improve performance or other attributes, or adapt to a changed environment. [Pressman 1997] (p. 791) states that the maintenance of existing software can account over 60% of all effort expended by a development organization. Furthermore he says that 80% of maintenance activities is spent adapting systems to changes in their external environment, making enhancements requested by users, and reengineering an application for future use. When maintenance is considered to encompass all of these activities, it is relatively easy to see why it absorbs so much effort. [Boehm 1973] affirms the statement that maintenance is the most significant cost factor in the lifetime of a piece of software.

[Fjelstad 1983] reports that more than 50% of time spent on maintenance is devoted to comprehension activities. Therefore [Rugaber 1994] proposes to facilitate the process of comprehending programs as an approach to improve software maintenance. Reverse engineering provides a direct attack on the program comprehension problem, but it is difficult, because it must bridge the gap between high-level descriptions and solutions on source code level and the gap between the documented and the actual structure of the software architecture which may have diverged over time.

The work presented in this thesis aims at bridging these gaps with the introduction of two process models for the reconstruction of software architecture views. The detailed task description for the diploma thesis “Process Models for the Reconstruction of Software Architecture Views” can be found in [Koschke 2001] and appendix E.

Architecture recovery supports the process of program understanding for software maintenance and system evolution. Ideally, the documentation of the architecture should be kept up-to-date. But when it is lost, due to undocumented changes, system evolution, or employee turnover, the description of the software architecture has to be recovered.

1.2 Architecture Recovery

[Clements 1996] states that although source code is often the most reliable arbiter of what a system does, it does not reflect all of the attributes of an application necessary to develop a true system-level understanding, i.e. quality attributes (e.g. robustness, reliability, performance, portability, etc.) must also be understood. And these quality attributes are related more to the software architecture than to its source code. Software architecture can be used as both as a framework for integrating sets of quality attribute domain models, and as a juncture between top-down and bottom-up strategies for program understanding.

[IEEE Std 1471-2000] states that every software system has a software architecture, which might be not or no more known. Therefore an architectural description of the existing system has to be constructed in order to guide maintenance or evolution activities.

Several people have written about software architecture recovery and processes to reconstruct views on it. Some ideas will be discussed in the next paragraphs:

- A framework introduced by [Kazman 1998] is called the horseshoe model consisting of four different levels: the source level (source code in textual representation), the code structure level (the source code in an intermediate representation that enables syntax-aware analyses), the function level (relationships among functions, data, and modules, providing a global system overview) and the architectural level (architectural elements, i.e., connectors and components). Architecture recovery usually starts at the source level. Three related processes are operating across the four levels, namely the architecture recovery, the architecture transformation and the architecture-based new development.

- [Kutscha 1997] introduces the task artifact cycle, an approach to system design which is based on knowledge of the system in use as well as on actual and future requirements. Based on a requirements definition (called task), the future system (called artifact) is designed in so-called design scenarios. Scenarios of use show the system in operation and it is important to monitor these for the system further development.
- [Kölsch 1997] presents a process model divided into two partial processes. A smaller process, called micro process, is able to manage processes that are limited to tasks which reengineer or maintain a special secluded functional and structural part of a system. The second and larger process, called the macro process, manages the controlling and guiding activities, while reacting flexibly and dynamically to all changes and replanning requirements.
- [Ponsignon 1999] proposes a framework that is based on three steps: In the reverse engineering step, a source code is mapped into a graphical view composed of low-level objects and actions. Then, in the design recovery step, this low-level view is transformed into a high-level view. Finally, the high-level constructs are intended to be reused.
- A process for object-oriented rearchitecting consisting of four steps is proposed by [Klösch 1996]. In the first step the design is recovered by various classic reverse engineering steps. Then an application model is constructed that is independent of the actual implementation. The third step is to match elements of the recovered design and the application model. In the last step the software is transformed.
- [Guo 1999] introduces a software architecture reconstruction method. This method is a semi-automatic analysis method for reconstructing software architectures based on the recognition of architectural and/or design patterns. It works only, if patterns were used during the development of the software system.
- An approach to capturing and assessing software architectures for evolution and reuse is presented by [Lung 1997]. It consists of four phases, gathering of available information, modeling the information in categories, analyzing of artifacts, and evaluating the results. With these four activities, several views on the architecture are constructed.

All of the approaches mentioned above are relatively abstract. They can be paraphrased as [Mendoca 1996] does: the process of reverse engineering an existing system is traditionally divided into two phases:

1. the extraction phase - identification and extraction of source code artifacts of interest for a given maintenance task
2. the analysis phase - analysis of the extracted in order to compose high-level design models of the system

But the problem is that the approaches do not fulfill all of the following objectives:

- high automatization by tools where possible
- reasonable integration of available techniques
- applicable to real world problems, i.e. easy to apply to large scale software systems
- realizable by one person or small teams
- explicit, precise work instructions
- fast reconstruction of useful views on the software architecture

Therefore two new process model presented in the thesis here have been developed to obtain these goals and towards the bridging of the gap between high-level descriptions and solutions on source code level, and between the documented and the actual structure. The process models operate with several tools and different methods for the purpose of gaining synergy effects when combining the different results. The one process model called “Design Hypothesis” aims at the achievement of an up-to-date high-level design model, the other named “Bottom-Up Analysis” segments the software architecture into different layers.

1.3 Basic Approach

In the beginning of this thesis (see table 1-1), a literature research took place to assess which tools, processes and techniques exist. Then it was evaluated to what degree the findings could be applied to the problem of architecture recovery. Since no appropriate process models (see above) have been found, two new ones have been developed. The choice of the specific tools is discussed in chapter 4.3.

For the process model, it was considered which attributes have to be taken into account for an architecture and how the architecture recovery can be performed. The models should be a step by step guide that is applicable in the small and works for large scale systems, as well. And it should be supported by tools where possible.

The attention was turned to methods and tools that work with large scale software systems, since both process models had to be evaluated in analyzing a legacy software system of an industrial partner. These case studies were done at Agilent Technologies, located in Böblingen, Germany.

The case studies were performed in two iterations. The first iteration was to verify that the process model work. Here weaknesses and possible improvements have been identified. After a refinement of the process models, the second iteration of the case studies began for the purpose of validating the refinements. The case study of the second process model (“Bottom-Up Analysis”) had to be shifted due to some problems with a tool (see chapter 9.5 for the reasons). In this thesis, only the latest version of the process models is presented. Differences between the process models of the two iterations will be sketched out explicit.

Phase	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Planning																											
Literature																											
Tools																											
Techniques																											
Process Models																											
Case Study 1																											
Evaluation																											
Refinement																											
Case Study 2																											
Writing																											
Presentation																											

Table 1-1. **Phases of the Diploma Thesis per Calendar Week**

The thesis will finish with a final representation being held after the reaching in of this document.

1.4 *Overview of the Remainder of this Thesis*

After this introduction, chapter 2 introduces the industrial partner, Agilent Technologies. The problems of the company with a particular legacy software and the contributions of the process models in resolving these problems are presented.

The terminology and process notation used throughout this document will be explained in chapter 3. Chapter 4 then discusses the tools and methods needed for the process models.

The main part of this thesis is presented in chapters 5 and 7. Here the two newly developed process models are described. It is explained how to work with the models and what external interfaces they have. The detailed description of the single activities is followed by the discussion of the improvements made during the iterations and still not removed weaknesses. Further on, possible fields of application are discussed.

The report of the result of the two case studies, one for each process model, is given in the chapters 6 and 8. Beyond the results, comments of the developers and collected data are presented. My lessons learned are discussed reflecting the experiences I gained during the evaluation.

Finally, the thesis concludes with a summarization of this work. The two process models introduced newly in this thesis will be compared to those presented in chapter 1.2. Open points and possible future steps will be discussed and the course of this thesis will be commented in retrospective.

Chapter 2 Agilent Technologies

ABSTRACT: The diploma thesis was done in cooperation with an industrial partner. The case studies took place at Agilent Technologies, Böblingen. This chapter introduces the company, describes their current situation and outlines their future goals concerning software architecture and architecture recovery with a particular software system. Finally, the contributions of this diploma thesis to the goals are discussed.

2.1 Business

Established in 1999, Agilent Technologies (see also [Agilent Technologies]) spun-off from Hewlett-Packard focussing on the fields of life sciences, optical and wireless communication, semiconductor products and chip test and measurement products. The automated test group comprises automated test of integrated circuits (IC) among other things. The software to operate those different IC testers is administrated, developed and maintained in the System-on-a-Chip Business Unit (SOCBU). In the next sections, some fundamentals of the semiconductor test are presented and the latest IC tester system is introduced.

2.1.1 What are Integrated Circuits?

Semiconductors are materials that can be used to build transistors. Transistors can be combined to create more complex circuits like logical gates which allow to construct decoders, latches, flip-flops, state machines, registers, and finally arithmetical units, memories, processors, and so on.

In the early seventies it became possible to integrate many of these electronic components into one, so called integrated circuit (IC) sometimes also called microchip or simply chip. Due to the demand for faster and more powerful computing, today’s chips are composed of many dedicated functional units (processors, custom logic, standard interfaces, analog/digital or vice versa converters, etc.). They are therefore called systems-on-a-chip (SOC) or silicon systems (based on [Bleher 2001]).

2.1.2 Semiconductor Test and the Agilent 93000 SOC Series

The purpose of semiconductor testing is to ensure that the IC complies with its specification, i.e. it performs the expected functions under the specified conditions within certain limits. The functions, conditions and limits are specified in the product's data sheet. During the production and packaging of a chip, it is tested several times, in order to sort out defective parts.

Agilent Technologies offers its customers a single scalable platform for testing ICs that combines a wide application coverage with full test capability for digital, analog, radio frequency SOCs.

Figure 2-1. **Agilent 93000 SOC Series Digital IC Test System**

A 93000 SOC series test system (see figure 2-1) connects the IC via the device under test (DUT) interface, and tests the chip by sending signals to it as input and compares the output with the reference values. The software suite controlling the tester runs on a HP-UX workstation and is called SmarTest (see also [User Manual Part 1]). The software comprises several tools and graphical user interfaces to setup test cases and to visualize the test results. Test cases are downloaded and then executed on the hardware of the IC tester. The interface managing the execution on the hardware side and the communication with the test software is called firmware.

2.1.3 The Firmware

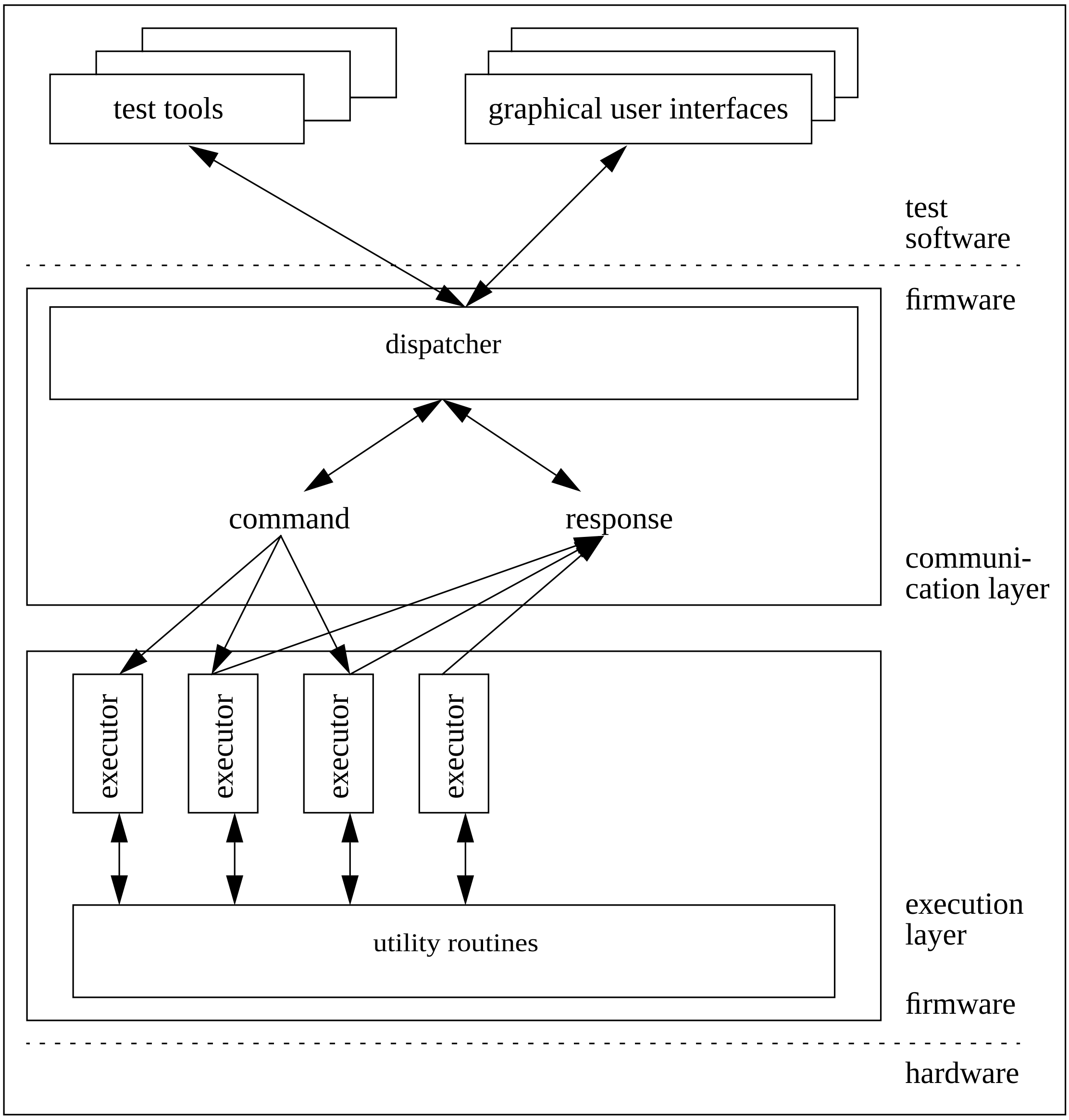

Figure 2-2. **Software Architecture of the Agilent 93000 SOC Series**

The SOCBU (see chapter 2.1.4) is responsible for the whole software of an IC tester, and therefore for the firmware, too. The two case studies described in chapter 6 and 8 examine the architecture of the firmware which was first developed for the HP82000 series about approximately 15 years ago. Since then the firmware was ported to new hardware requirements, adjusted to new versions of the operating system, and changed several times due to enhancements and enlargements of the functionality. These maintenance activities have let the firmware become very complex and quite inscrutable, hence these activities have grown to be very expensive. Up to the present the firmware consists of about 500 KLOC, more than 1,2 million LOC with comments.

The firmware is accessed by a message based command interface (similar to the General Purpose Interface Bus, [IEEE 488]). The interface offers approximately 400 firmware commands, either to execute a command, or to query the results or the state of the test. A detailed description of each command can be found in the [Command Reference].

The firmware consists substantially of two different layers, a communication layer and a execution layer. The communication layer propagates the control commands, also called firmware commands, to the corresponding functions in the execution layer. The execution layer contains the firmware executors which execute the command. There is an executor (i.e. a routine) for each firmware command.

A test case is a sequence of firmware commands. The firmware parses and interprets each command in the communication layer. The firmware is accessed and used simultaneously by different tools each running in separate operating system processes. These tools and the firmware interact via a dispatcher which is used to synchronize these interactions between the firmware and other tools. The interface for these interaction is called message parser and constructor (PAC).

2.1.4 System-on-a-Chip Business Unit (SOCBU)

Since Agilent Technologies is a global company, the SOCBU is also geographically distributed into five departments, two of them located in the USA, one in Japan, and two in Germany. The departments are more or less oriented at the market structure.

The case studies were conducted in the SOC Platform Division (SOCPD) and the Computation and Communication Solution Test Division (CCST). These two departments, are located at Böblingen, Germany.

2.1.5 Elements of the Digital Firmware

In this chapter an overview about some elements of the firmware is given. The explanation of the elements serves for a better understanding of the approaches in the two case studies. The case studies try to reveal the structure of parts of the firmware, and therefore some knowledge about the underlying concepts is needed. For an in-depth explanation see the [Command Reference] and the [User Manual Part 1].

The IC tester is physically connected to the device under test through at least one but possible up to 1024 pins. The type of a pin can be input, output, bidirectional, digital, analog or connected to the device power supply (DPS). Only the digital part of the firmware was analyzed in the case studies, the rest was omitted.

In the firmware, several pins can be grouped together, i.e. it is possible to write test cases that adjust and measure many different parameters for individual pins and whole pin groups of the tested chip.

Pins have to be configured before executing a test case. Configuration commands allow to assign pin names to a test or power supply channel and to define the pin type and the operation mode.

Level commands specify the required driver amplifier and receiver comparator voltage levels. They also define the device power supply voltage levels, both for a single pin or for whole groups.

Each pin of a device is associated with a channel consisting among others of a vector memory, and a sequencer. A channel generates the input for the device under test and/or compares the output against the expected values.

The sequencer is a simple, programmable microprocessor that generates vector data out of the raw data in the vector memory. For instance, loops, conditional jumps and comparisons can be done during the data generation. The sequencer controls which vectors are sent to the device under test via a waveform that transforms the vector data into signals, called drive and receive edges.

The test vectors are downloaded into the vector memory. Each test vector is referenced via a test label. In order to save space, test vectors are normally stored on the workstation in compressed format. When downloading, the data is decompressed by the compression utility.

The timing on the Agilent 93000 SOC series is based on device cycles, where each device cycle can consist of several clock cycles, or in other words, tester cycles. The concept of device cycles can be illustrated with microprocessors. For example, a microprocessor may need a device cycle consisting of two clock cycles to read data from the addressed memory location. One cycle to issue an address and one cycle to fetch the required data from the addressed memory location. The timing setup commands define the device cycles for all configured pins and set the system clock. A system clock is needed for the synchronization of the clock cycles.

During a device cycle, a stimulus is sent to the tester. This stimulus is based on a waveform. a timing, a level and a vector and a sequencer program. There exist several sets of elements needed for generation of the stimulus, but only one element of each set can be active at a time.

2.2 Current Situation

This chapter outlines the major problems emerging and the approaches to solve them at the development departments in Böblingen.

2.2.1 The Development Process

The development process at Agilent Technologies consists of four main phases:

1. The concept phase
2. The investigation phase
3. The design and development phase
4. The qualification phase and shipment

Although the overall process is well-defined, repeatable and has several quality assurance activities, the design and development phase is based on an ad-hoc development of new enhancements. This means that one person (or one team) can access the whole source code in order to realize a feature as a whole without being supervised by a control instance. There is no further decomposition for the implementation of several features, i.e. no central instance coordinates the implementation and maintenance activities and takes care about the architecture of the firmware. That has the following consequences:

- Documentation gets outdated when changes are made randomly and not documented. Due to the ad-hoc process, quality assurance activities to support the documentation are not executed.
- Reuse is made difficult. Now a developer has to figure out for himself whether there are routines that he can use for accomplishing his work. So the possibility of reuse could be better supported when having an up-to-date documentation.
- Architectural information and changes are not collected and documented. The structure of the system remains only in the minds of the developers, but due to the fact that another person can change parts one used to know very well, the risk of losing more and more architectural information increases as time progresses and changes evolve.

The consequences described above become more and more obvious in the firmware now. Due to the 15 years of evolution, the porting to new hardware requirements, the new introduced features, the ad-hoc development process and the globally distributed development departments, the architectural information about the structure of the firmware got lost.

Maintenance has now become a very difficult and expensive activity. The porting to a new hardware is very risky.

2.2.2 Problems with the Firmware

The main problem in the firmware is that there is no explicit separation of hardware dependent routines, variables and types, i.e. those elements which have a direct physical counterpart, and those which are a logical abstraction. Little changes on the hardware can have deep impact on the firmware, because that separation is not existent. Therefore Agilent Technologies will introduce a virtual hardware layer. This layer is a scalable interface which should capture the hardware dependent parts of the firmware and hide them from the rest. This has the advantage of having one firmware with its concepts working for several different types of hardware. If new hardware is introduced, only the virtual hardware layer has to be adjusted to the new characteristics. The executors and the concepts of the firmware remain unchanged.

2.2.3 Goals of Agilent Technologies

The consequences and problems mentioned above need to be addressed. Therefore restructuring activities are planned by Agilent Technologies:

1. Module identification
2. Separation of hardware dependent and independent code
3. Encapsulation of parts which operate directly on the hardware
4. Remodularization

Refactoring activities (see also [Fowler 1999]) should accompany the steps. The first two steps result in an up-to-date documentation where the developers should try to understand design decisions and rationales. The last two steps change the source code. Here new concepts can be introduced, for instance design patterns (see [Gamma 1995]). The remodularization should be followed by a regression test to prove the functionality is still given.

These steps when performed successfully one after another lead to a higher maintainability. Defects can be fixed faster, new hardware can be adopted easier, and the reaction to customer wishes and new requirements can be done quicker than before. Hence, these steps leads to better market position and monetary savings.

2.3 Contributions of the Diploma Thesis

Within the framework for the goals mentioned above, the Process Model “Design Hypothesis” (see chapter 5) addresses the first step, the module identification. The process model “Bottom-Up Analysis” (see chapter 7) will be instantiated in order to separate hardware dependent as well as hardware independent parts of the firmware, i.e. partly step one and two. Both process models are evaluated and measured so that Agilent Technologies has a foundation for the estimation of effort and time planning.

On the contrary to the ad-hoc activities done normally at Agilent Technologies the process models combine applicable tools and methods in a framework, that does work with such a large scale software system as the firmware is. The specific tools and methods are linked together via processes that are structured and well-defined.

The results of the case studies can be a starting point for Agilent Technologies to proceed and to extend the approaches to the whole firmware.

Furthermore, the results can be used by Agilent Technologies to document the software architecture of the firmware, and hence, to reduce the training time of new employes.

Chapter 3 Terminology, Concepts and Notations

ABSTRACT: This section introduces the terminology used throughout this thesis. It also describes the different roles of the process models and explains the process notation.

3.1 Terms

In software engineering several terms have been defined more than once or with more than one exact meaning. This chapter will give the definitions used throughout this document.

3.1.1 Process Model

A process is a sequence of steps performed for a given purpose; for example, the software development process [IEEE Std 610.12-1990].

A process is explained with a graphical and textual process model, which is the formalized representation of a process. A model describes in detail the activities, objects, and roles of the process, as well as how they are related. The primary purpose is to communicate an understanding of the process.

3.1.2 Activity

An activity is any step taken or function performed, both mentally and physically, toward achieving some objective. Thus, an activity is work accomplished within a process.

3.1.3 Forward Engineering

[Chifkovsky 1990] defines that forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.

3.1.4 Reverse Engineering

[Chifkovsky 1990] says that reverse engineering is the process of analyzing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction

Reverse engineering generally involves extracting design artifacts and building or synthesizing abstractions that are less implementation-dependent.

3.1.5 Reengineering

[Chifkovsky 1990] defines reengineering as the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

Reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring.

3.1.6 Restructuring

Restructuring is the transformation from one presentation form into another at the same relative abstraction level, while preserving the subject's external behavior, i.e. functionality or semantics (see at [Chifkovsky 1990]).

3.1.7 Architecture

The architecture is a model of the software system used to demonstrate certain properties, e.g. decomposition, design rationale. Depending on which properties are of interest one needs one or several architectural views on the architecture.

[Koschke 2000] states that there are still debates about the definition of software architecture, but most agree that it should include at least components and connectors and their hierarchical decomposition. Components are the computational parts and connectors describe the interactions between these components.

[Shaw 1996] defines that software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their decomposition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions (connectors) among those components.

[IEEE Std 1471-2000] defines architecture as 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.

3.1.8 Architecture Recovery

Architecture recovery is a discipline of reverse engineering that is aimed at recovering the software architecture of a system.

3.1.9 Architecture View

[IEEE Std 1471-2000] defines a view as a representation of a whole system from the perspective of a related set of concerns.

[Bachmann 2001] says an architectural view represents a set of system elements and their relationships. A view documents a particular aspect of the system's architecture while intentionally suppressing others. Different views will highlight different system elements and/or relationships.

Several authors prescribed specific views one should employ, for example, [Kruchten 1995] states that in order to describe a large scale software architecture, a model of multiple views or perspectives is needed. [Hofmeister 2000] introduces four different views. The process models for the reconstruction of software architectures introduced in this thesis will focus on the reconstruction of the conceptual and the module view (see [Hofmeister 2000] pp.12).

- The Conceptual View:

  In this view, the functionality of the system is mapped to architectural elements called components with coordination and data exchange handled by elements called connectors.
- The Module View:

  In the module view, the components and connectors from the conceptual view are mapped to subsystems and modules. Here it is outlined how a conceptual solution can be realized. The main concerns of this view are reuse, dependencies between modules and the regard of connections to other external software or hardware.

3.1.10 Component

A component is a group of related elements with a unifying common goal or concept relevant at the architectural level. An atomic component is a non-hierarchical component that consists of related global constants, variables, subprograms, and/or user-defined types. As opposed to an atomic component, a subsystem is a hierarchical component consisting of related atomic components and/or lower-level subsystems [Koschke 2000].

3.1.11 Connector

Connectors mediate the interactions among the components. A connector's roles define the behavior of the participants in the interaction (see [Hofmeister 2000] p. 359).

3.1.12 Feature

[Eisenbarth 2002] states that a feature is a realized functional requirement. A feature also subsumes non-functional requirements, but in the context of this thesis only functional features are relevant, i.e. features which provide the user with an observable result of value.

3.1.13 Subsystem

[Hofmeister 2000] defines a subsystem as a collection of subsystems and modules grouped by functionality. A subsystem corresponds usually to a higher conceptual component (one that is decomposed into other components and connectors).

3.1.14 Module

A module encapsulates data and operations to provide a service. These services are defined by the interfaces they provide. For example, a module may also need the services of other modules to perform its function, then this is named the use-relation (see at [Hofmeister 2000] p. 362).

A module is ideally a portion of code with high cohesion (see [Fenton 1997]) and a narrow interface, i.e. low coupling (see [Yourdon 1979]) to other modules. Each module has some specific tasks grouped around a single logical element. It should have limited scope and be clearly distinguishable to other modules.

3.1.15 Relations

3.1.15.1 The Call Relation

The call relation between two routines exists when a routine calls another routine. The completeness of all calls is named the call graph. The call graph is a directed graph.

Call relations based on function pointers have not been extracted and are not considered, i.e. the extracted call graph used in the process models is a subset of the complete one.

3.1.15.2 The Data Relation

A data relation is the summary of four different kind of relations. It is summarized for the purpose of having filtered information so that the views do not get too complicated. The four kinds of subsumed relations are:

- The Variable Use Relation:

  This relation indicates that a global variable is accessed within a routine.
- The Variable Set Relation:

  If a global variable is set in a routine, then the variable set relation is existent between the routine and the variable.
- The Member Use Relation:

  This relation exists if the member of a type is accessed in a routine.
- The Member Set Relation:

  If a member of a type is set in a routine, then a member set relation exists between the member and the routine.

The data relation is a subset of the data flow graph. It contains no dynamic information and it can not resolve pointers. There may be obsolete elements in this relation because one can not decide for all statements whether it is reachable or not.

3.1.15.3 Other Relations

Here some other relations are presented for which the process models can be extended:

- The Is Part Of Type Relation:

  If one type is the part of another type, then this relations exist between those types, i.e. the one type is needed for the declaration of the other type.
- The Parameter of Type Relation:

  If a type is in the signature of a routine, then this is called a parameter of type relation.
- The Local Variable of Type Relation:

  If a routine has local variable of a type, then this relation is existent.
- The Actual Parameter Relation:

  If a routine has a global variable as actual parameter, then the actual parameter relation exists.
- The Address Taken Relation:

  This relation exists when the address of a variable is taken by another variable.

3.1.16 Module Relations

A module relation is a direct dependency between two modules, i.e. the one module uses the other module in order to work properly. A module can also have relations to itself.

There can exist several, different relations between two modules which can be extracted by reverse engineering tools.

The call relation (see chapter 3.1.15.1) and the data relation (see chapter 3.1.15.2) have been propagated on the module level, e.g. the call relation between the modules A and B exists when a routine in module A calls a routine in module B.

This diploma thesis considers basically only the call relation and the data relation. Other relations have been introduced in chapter 3.1.15.3, but they were not considered for the sake of simplicity, but the process models can be easily extended to those relations. When taking all other relations into account, the quantity of the extracted relations increases fast and problems occur in managing the information. The other relations deliver additional information about the module relations, but it is not necessarily needed when reconstructing software architectures. Therefore, in my opinion the most important relations will be investigated, namely the call and the data relation.

3.1.17 Layer

[Hofmeister 2000] states that layers organize modules into a partially ordered hierarchy. A module is assigned to a layer, which then constrains its dependencies on other modules.

3.1.18 High-Level Design Model

A high-level design model concentrates on a certain part of the whole module view. It may consist of subsystems, modules and their relations.

3.1.19 Source Code Element

A source code element is a distinguishable item of the specific programming language. One source code element can include several other source code elements of the same or a lower abstraction level. For the software system under investigation (programmed in C) there were four source code elements declared:

- Directory
- File
- Function
- Variable

For other programming languages like Java there may be other source code elements like class or member.

3.2 *Concepts in UML*

In this chapter concepts concerning the terms mentioned above are presented and explained (based on [Hofmeister 2000]).

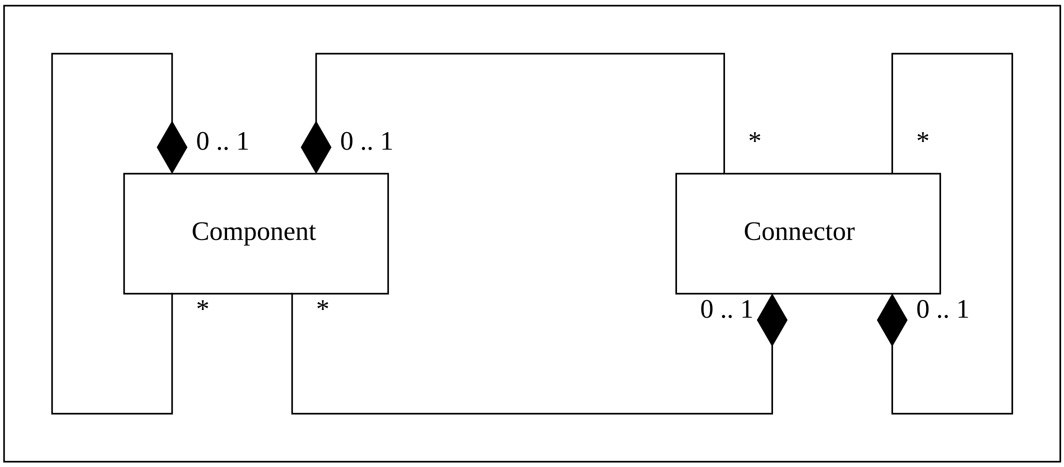

Figure 3-1. **Components and Connectors**

A component can be decomposed into components and connectors, the same is valid for a connector. The functional behavior of a system is concentrated in the components, while the control is concentrated in the connectors. A subsystem usually corresponds to a higher level component or connector, a module can correspond to a single component or connector or to a set of them.

Figure 3-2 shows a conceptual system decomposition. Subsystems and modules are static architectural components that differ in the degree of granularity. A module, as the leaf item, corresponds directly to implemented code, i.e. it may consist solely of functions, subprograms, and type declarations. Large systems are decomposed into subsystems that can be managed individually.

Modules can be assigned to exactly one layer. Layers consist of further layers and can only use layers that are on the same or a lower level in the hierarchy.

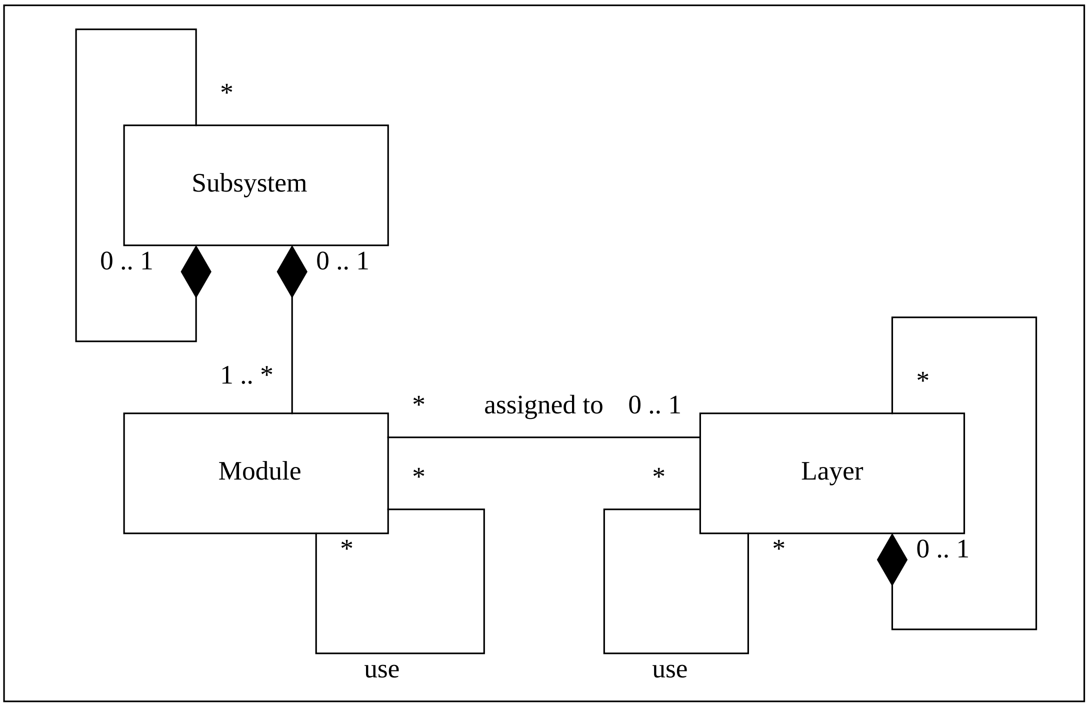

Figure 3-2. **System Decomposition**

3.3 Roles

A role implements one or more activities by accomplishing a goal, creating an artifact, or using a tool. A role can be performed by a single person or a group. Otherwise one person can fulfill several roles, too. Both process model operate with some roles defined in this section.

3.3.1 Developer

A person being involved in the software development life cycle of the software system is called developer, i.e. a person that creates the software or changes it. This includes designers, programmers, maintainers and all other persons who can edit the software system on the source code level.

3.3.2 Analyst

An analyst is the person who executes a process model. He is responsible for the availability of the tool set and must have access to all items concerning the software system under investigation. The analyst should also know whom to address when he needs specific information about the software system or its underlying concepts.

3.3.3 Software Architecture Expert

The software architecture expert is a person who knows the software architecture or at least parts of it very well. He has in-depth knowledge of his working area and experience in the decomposition of the software system.

The software architecture expert can provide the analyst with information about the software system and can advise him if necessary. Normally the analyst cooperates with several software architecture experts because in a large scale software system it is nearly impossible for one person to be familiar with all parts.

3.4 IDEFØ Process Notation

IDEF or Integrated Definition language is the modeling technique used to model functional processes and corresponding information to support Functional Process Improvement. IDEFØ is a notation designed to model the decisions, actions, and activities of an organization or system. IDEFØ was derived from a well-established graphical language, the Structured Analysis and Design Technique (SADT), for detailed information see [IDEFØ].

Both process models are described in the process notation introduced here.

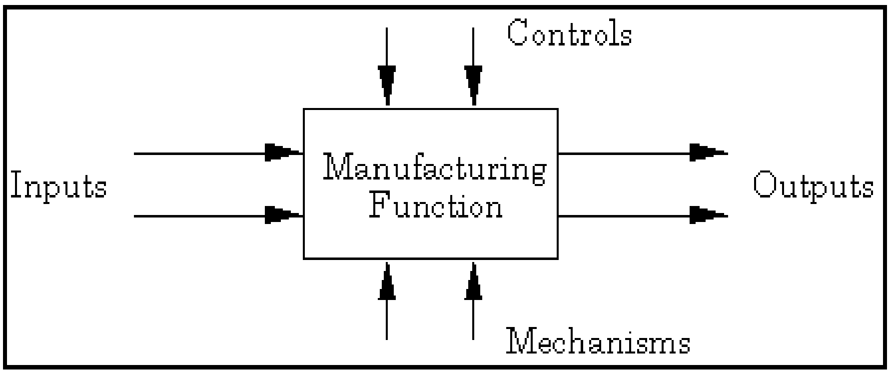

Figure 3-3. **IDEFØ Function Box and Interface Arrows**

3.4.1 Diagram

The diagrams can be refined featuring a hierarchical structure, with the major diagram at the top and with successive levels of sub diagrams revealing well-bounded detail breakouts. Diagrams are based on simple box and arrow graphics.

3.4.2 Box

An IDEFØ diagram shows activities as a box. An activity can be decomposed into several sub activities.

3.4.3 Arrow

The interfaces to or from a function are shown as arrows entering or leaving a box. To express functions, boxes operate simultaneously with other boxes, with the interface arrows constraining when and how operations are triggered and controlled.

3.4.4 Input

Inputs will be transformed by the activity or process. This includes material and information. An output of one activity can be the input to another one.

3.4.5 Control

Control elements are related to the activity that constrains or governs how the activity will be conducted. Examples include policies, budget constraints, and customer requirements.

3.4.6 Output

The result of the activity is the output. This is the input after it is transformed by the activity.

3.4.7 Mechanisms

Mechanisms conduct or support the activity. These may be people, systems, facilities or equipment necessary to accomplish the activity. The direct costs of conducting an activity are associated with the mechanisms.

Chapter 4 Methods and Tools

ABSTRACT: This section introduces the methods and techniques used throughout this thesis as well as the evaluated tools. Each method is explained with a trivial example for a better understanding.

4.1 Used Techniques

For the following chapters it is assumed that the techniques presented here are known. They are the basis and are needed for the application of the process models.

4.1.1 The Reflexion Model Technique

The software reflexion model technique provides the analyst with the summary of extracted source code information from the viewpoint of a high-level design model. It is an approach to reveal the structure of an existing software system by validating the assumptions about structural information of the software system with the source code. The high-level design model of the structure of the software system is used as a lens through a model of that system’s source code can be viewed. The computed software reflexion model shows where the model agrees with and where it differs from the extracted information of the source code. Detailed information about the software reflexion model can be found at [Murphy 1995], an overview about the file formats needed for the reflexion model is given in appendix B.

The reflexion model technique is suitable for large scale software systems, as [Murphy 1997] shows in a case study where Microsoft Excel was investigated, a product that comprises about 1,2 million LOC.

A high-level design model consists of modules (described in chapter 3.1.14) and their relations. A relation (see chapter 3.1.15) represents a direct dependency between two modules (see chapter 3.1.16). A relation between two modules is in the context of the reflexion model called an edge.

4.1.1.1 The Edges of the Reflexion Model

Two modules can have interactions with each other, in the context of the reflexion model called edge. Each edge has explicit one of three different types:

- Convergencies are interactions in the high level design model that were found in the source code, too.
- Divergences describe where the source code differs from the high-level design model.
- Absences are interactions stated in the high-level design model that do not correspond to any interactions in the source code, i.e. the model differs from the source code.

Between two particular modules there can only exist one of the edges mentioned above for each direction and each relation type (e.g. there can only be one call relation for each direction between two modules).

4.1.1.2 Inputs of the Reflexion Model

The reflexion model needs four things as input:

- the description file of the source code elements
- the source code translated into the input format
- the high-level design model
- the mapping file

For a detailed description of the four file formats see appendix B.

4.1.1.3 An Example for the Reflexion Model

Figure 4-1 shows three high-level design models which include the modules Main, Output and Print. Each module represents one file containing the source code below:

```
Main.c:
void Main()
{
    do_Output();
}

Output.c:
void do_Output()
{
    do_Print();
}

Print.c:
```

```
void do_Print()
{
    print(%c, “1”);
}
```

The left high-level design model is an assumption about the structure of the software system made by the analyst. The model in the middle shows the real structure the source code above has. And the third model on the right shows the computed software reflexion model based on the hypothesis and the as is structure.

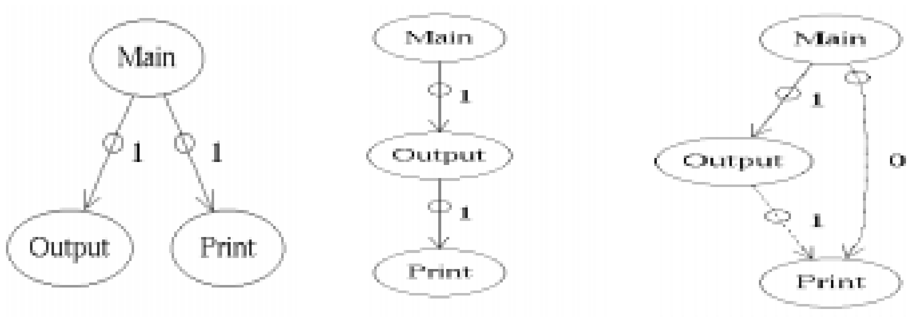

Figure 4-1. **The Hypothesis, the “As Is” Model, and the Reflexion Model**

Convergencies are drawn with a solid line, divergences as a dashed line and absences with a dotted line. The number attached to each edge indicates how broad the interface for this interactions is, i.e. how many interactions were found in the source code. The null at the absence edge means that there is no interaction in the source code.

As one can see in this trivial example, the hypothesis about the module dependencies is only partly correct.

4.1.2 Concept Analysis

The formal concept analysis is a mathematical method for the analysis of binary relations. The mathematical foundations of the concept analysis are explained in appendix C and in [Ganter 1999]. A concept is pair of objects and attributes. A concept lattice establish an order of the concepts. When displaying a concept lattice one can see which objects have common attributes.

4.1.2.1 Example for a Formal Concept Analysis

In the following example, the routines are the objects and the accessed members of the specific types are the attributes for the formal concept analysis (r stands for routines, t for types). An “X” in the relations table 4-1 means that the routine r accesses a member of the type t.

	r1	**r2**	**r3**	**r4**
t1	X			
t2	X			
t3		X	X	X
t4		X	X	X
t5		X		X
t6			X	X
t7			X	X
t8			X	X

Table 4-1. **A Relation Table**

The concept lattice corresponding to the formal context of table 4-1 is presented in figure 4-2. The concept lattice is shown in a sparse representation, i.e. a routine or a member will only appear once in the lattice, routines at the lowest, attributes at the highest level.

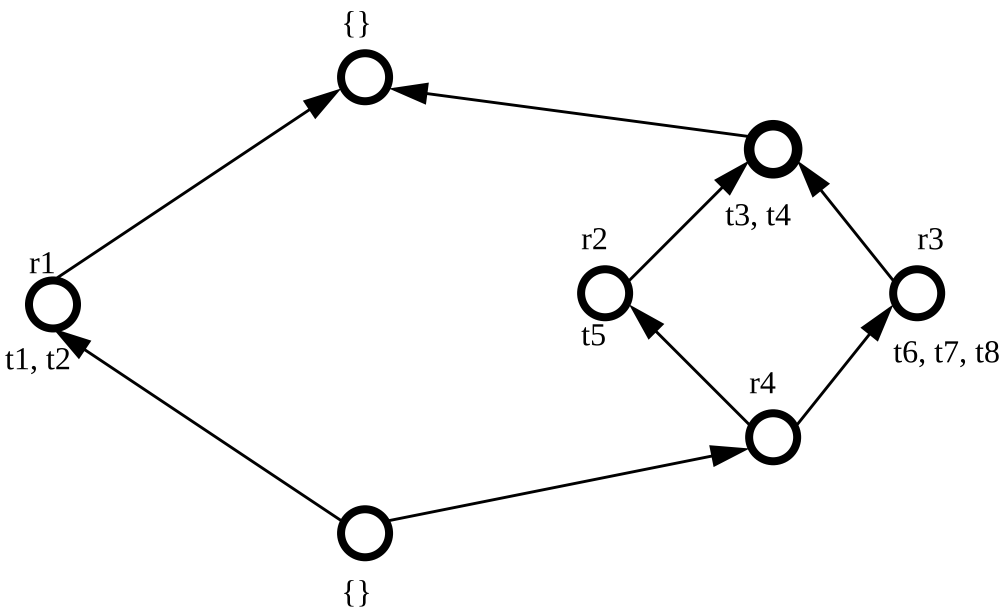

Figure 4-2. **Example of a Concept Lattice**

4.1.2.2 Interpretation of the Concept Lattice

As figure 4-2 shows, the top element of the concept lattice indicates that there is no type that is accessed by all routines. And the bottom element shows that no routine is accessing all types exists.

The concept lattice can be split into two parts. The left and the right side have no coupling, i.e. their are no dependencies between the two parts. The type t1 and t2 are specific to the routine r1, they are not used by other routines. The cohesion between these three source code elements (t1, t2, and r1) is high because the routine only accesses the member of those types. And they have a low coupling to the other elements, i.e. no other routines access the members of t1 or t2, and r1 does not operate with members of other types. So these are candidates to be grouped together into one single module, since it fulfills the criteria given by [Fenton 1997] and [Yourdon 1979].

The right side can be interpreted as one big module, too. But also a more detailed separation can be useful. For instance, the routines r3 and r4 operate with members of the types t3, t4, t6, t7 and t8. It might make sense to group them to one module, although the routines have dependencies to other types, or the types have operate with other routines.

The interpretation of the concept lattice provides the analyst with information routines that access common members of types. With this information he can identify source code elements, that may be grouped in modules. And he gets aware of grouped set of routines and types that a high amount of relations to other source code elements.

4.1.3 Program Slicing

Program slicing is a technique that examines the source code referring to exactly one starting point, i.e. a statement in the source code. The influences of this point in the control and data dependency graph are examined, i.e. for all other statements can be determined whether there is dependency of the starting point or not. All statements of the source code, that depend on the starting point, or that is starting point is dependent of, are called slice. The idea of program slicing was first introduced by [Weiser 1984] and then refined and improved by [Horwitz 1990]. The technique is used to gain a better program understanding in order to support reengineering, maintenance and to collect metrics like the cohesion of a software system.

4.1.3.1 Types of Slicing

A slice always refers to exactly one point in the source code. A forward slice contains all statements that are influenced by the starting point, i.e. the execution of the statements is dependent from the starting point. In contrast to that, all statements influencing the behavior at a certain point compose a backward slice.

Intraprocedural slices consider only statements within a single routine. The basis for this is the program dependency graph, an intermediate representation for data and control dependencies. The system dependency graph can be examined for interprocedural slices, i.e. the slices can be spanned over several routines. The program dependent graphs for all routines combined with additional information for routine calls result in the system dependency graph.

In the context of the diploma thesis, only static program slices are relevant. But dynamic slicing as described in [Beszedes 2001] can be implemented, too. Dynamic slices can be used to make a rough estimate of a static slice. Since in a dynamic slice, the software is running, only really executed statements are collected. So dynamic slices are based on test cases, set up and executed in order to produce the results. Static slices record all statement, even those which are never executed, but they can be collected without running the software, and no test cases are needed.

So when slicing is mentioned in the description of the process models, always static slicing is meant.

4.1.3.2 Example of Static Program Slicing

The following source code is the basis of the examples presented in this section:

```
void main()
{
int num;
num = 5;
calc(num);
}

void calc(int n)
{
i = 0;
sum = 0;
product = 1;

while (i <= n)
  {
  sum = sum + 1;
  product = product * i;
  i = i + 1;
  }

write (sum);
write (product);
}

void write(int number)
```

```
{
print(%i, number);
}
```

Then the intraprocedural forward slice of the statement “sum := 0” is the following extract:

```
sum = 0;
while ( ... )
  {
  sum := sum + 1;
  }
write (sum);
```

And for the statement “write (product);”, this is the resulting intraprocedural backward slice:

```
i = 0;
product = 1;
while (i <= n)
  {
  product = product * i;
  i = i + 1;
  }
write (product);
```

The interprocedural backward slice for the statement “write (product);” would consider the extract of the source code above as well as the statements concerning the variable “num” in the routine “main” would be taken into account. The routine “write” has only to be considered when generating the forward slice of the “write (product);” statement. The interprocedural slices are connecting the intraprocedural slices for the starting point.

4.2 Working Environment

The working environment where I assessed the tools and evaluated the process models was a PC running under Linux. Collected data concerning time measurements was made on this machine. In detail the hardware was:

- Pentium III 750 MHz
- 768 MB RAM
- > 1 GB hard disk

The software configuration of the working environment was the following:

- RedHat Linux 7.1

- Kernel 2.4.2-2
- KDE 2.1.1

The programming language of the software system under investigation was C.

4.3 Tools

In this chapter the tools evaluated are presented and the decisions made for using or not using the tool are given. There exist a lot more reverse engineering tools than presented here but due to the time restrictions only the tools mentioned below were evaluated. The analyst is free in his choice and can choose others tools if he likes to.

4.3.1 Bauhaus

The Bauhaus tool provides the user with methods and tools to analyze and recover the software architecture, derivation of different views on the architecture of legacy systems, identification of reusable components, and estimation of change impact. Advanced compiler technologies for data flow and control flow analysis are used as basis analyses to support these reverse engineering activities (see appendix D, [Bauhaus] and [Eisenbarth 1999] for further information).

The Bauhaus tool is used in both process models because it provides the analyst with a variety of reverse engineering methods and it had the best support.

4.3.2 CIAO

CIAO is a customizable and extensible source code navigator. It allows users to query, analyze, visualize, and track structures of various software and document repositories. CIA which is the instantiation for C is part of the CIAO tool set (see [CIAO] for detailed information).

This tool is not considered because it did not work properly with the software system under investigation.

4.3.3 CodeSurfer

CodeSurfer is a maintenance, understanding, and inspection tool. CodeSurfer provides the user with precise inter procedural backward and forward program slicing (see also chapter 4.1.3) and pointer analysis (for detailed information see [CodeSurfer]).

This tool is used for program slicing.

4.3.4 Graphviz

Graphviz is a set of graph drawing tools offering hierarchical layouts of trees and DAGS (directed acyclic graphs) and virtual physical (“spring model”) layouts of undirected graphs. See [Graphviz] for detailed information.

This tool was used for visualization purposes because it is needed by the jRMTool to layout the results.

4.3.5 Imagix 4D

Imagix 4D is a tool that provides the user with program information about the source code, call graphs, variable usage, test coverage, system performance and class hierarchies (see also [Imagix 4D]).

This tool is not used because it offers similar functionality as Bauhaus and CodeSurfer.

4.3.6 ISVis

ISVis provides a developer with browsing and analysis of execution scenarios, derived from actual program executions. The key features are its use of visualization techniques to depict the large amounts of information available to a user, and the notion of recurring scenarios, or interaction patterns, as abstractions which help bridge the gap between low-level event traces and high-level design models (see also [ISVis]).

Since ISVis is a tool for dynamic analysis and running only under Solaris, it was not appropriate.

4.3.7 jRMTool

The jRMTool is a tool for viewing and displaying reflexion models (see also [jRMTool]). What reflexion models are is described in chapter 4.1.1.

This tool is used since it is the only tool available that has this functionality.

4.3.8 Lackwit

Lackwit investigates the use of type systems and type inference to statically deduce properties of C programs (see [Lackwit]).

Because of its limited functionality, this tool is not taken into account.

4.3.9 Rigi Workbench

Rigi is an interactive, visual tool designed to help the developer better understand and re-document the software. It supports the developer with the analysis, representation, and visualization of software structure. Software structure refers to a collection of artifacts including software components such as subsystems, procedures, variables, calls, data accesses, and interfaces, dependencies among components such as client-supplier, composition, and control and data-flow relations and attributes such as component type, interface size, and interconnection strength (see [Rigi Workbench] and [Müller 1994]).

The Rigi Workbench is not considered although it has several useful feature because the Bauhaus tool provides the analyst with even more functionality and a better performance when layouting elements in the graphical user interface.

4.3.10 RMGuiMapper

This tool was implemented during the diploma thesis to accelerate the first process model, namely “Design Hypothesis”. It is written in the programming language Java.

Due to the fact that this tool was implemented in order to improve one process model, it is obvious that this tool is considered (see also chapter 5.6).

4.3.11 Portable BookShelf

The Portable BookShelf is a tool kit for the presentation and navigation of information representing large software systems (see also [Portable BookShelf]).

This tool is not considered because it was not working properly under the working environment described in chapter 4.2.

4.3.12 SNiFF+

Wind River´s SNiFF+ tool is a source code analysis environment for Unix or Windows application using C, C++, Java, or ADA or a mixture of any of those languages. It assists the developer with source code browsing, a graphical representation of the source code and automatically-managed makefiles (for detailed information see [SNiFF+]).

SNiFF+ remains unused in the process model because of two reasons. The first one is that the functionality is already made available by other tools and the other one is that the developers at Agilent Technologies already know this tool and they are not satisfied with it because SNiFF+ did not fulfill the expectations.

4.3.13 TraceGraph

TraceGraph being part of the software reconnaissance tool kit is a free software tool that assists C programmers in understanding unfamiliar code. It locates where program features are implemented by tracking the execution of test cases. For further information see [Wilde 2000] and [TraceGraph].

Due to the fact that in the end only two process models have been developed which are based on static analysis, this dynamic analysis tool is not considered.

4.3.14 Xrefactory

Xrefactory is a development tool for Emacs and XEmacs with code completion, source code browsing and refactoring functions. It supports the programming languages C and Java, on Unix, Linux and Windows platforms (see [Xrefactory] for detailed information).

This tools is chosen since it enables the analyst to find source code elements fast and to browse their reference in the source code.

4.3.15 Summary

This section recapitulates the decisions about the selection of the tools. The jRMTool and RMGuiMapper are unique in its functionality, i.e. no other product was found, that can be used instead of these. The jRMTool requires GraphViz for the visualization. Xrefactory accelerates the source code navigation. Slicing is done with CodeSurfer. Therefore these tools are used. Bauhaus subsumes the functionality of several reverse engineering tools like the CIAO, Lackwit, Imagix 4D and the Rigi Workbench. Therefore only Bauhaus is used instead of several of the other tools.

The absence of dynamic analyses leads to the left out of tools designed for those tasks, namely ISVis and Tracgraph. The Portable Bookshelf did not work properly.

Chapter 5 Process Model “Design Hypothesis”

ABSTRACT: In this section the process model “Design Hypothesis” will be introduced. The external interfaces will be described in detail and the activities to perform will be explained. The major changes to prior versions of this process model are sketched and it is discussed why these have been applied. Finally, further possible refinements and potential fields of application will be discussed.

5.1 Overview

This chapter describes a process model which is based on design hypotheses. The analyst makes an assumption of the structure of the software system under investigation. The assumption consists of several logical modules and the relations among them representing the software.

This high-level design model of the software is compared to the source code and the results are analyzed. During several iterations the high-level design model is refined further until the model is in a stable state.

The main idea of this top-down approach is that the analyst starts with a quite inaccurate view on the software architecture. Based on the structural information gained through the analysis he can successively adjust the content of the view so it will become precise.

The purpose of this process model is the generation of a high-level design model of the software architecture which provides the analyst with an up-to-date view on the software system. Then further steps in the maintenance or reengineering process can be executed or the documentation can be improved or validated.

5.2 Premises

There are some premises which should be taken into account before implementing the process model. The analyst should try to gather all available documentation of the investigation subject, even if existing documentation may be outdated or not complete. The found pieces can serve as a foundation for analysis. It is assumed that even after years of evolution and several major changes to a software system that at least some of the main design decisions are still present. Or if there exists multiple, contradictory documentation, one can decide which is the correct one.

Even more important than the documentation due to the fact that most information is in the mind of the developers are the people. The analyst has to find software architecture experts who can support him when thinking about the structure of the software system. These people may be experts for specific parts or the system as a whole. In order to gain best results, the analyst should work with them closely and ask for advice if necessary.

The third and the last premise is that all the required tools should work properly because most activities of this process model are supported by tools. If this is not the case, the analyst has to perform the steps manually which would cost significant more effort.

5.3 External Interfaces of the Process Model

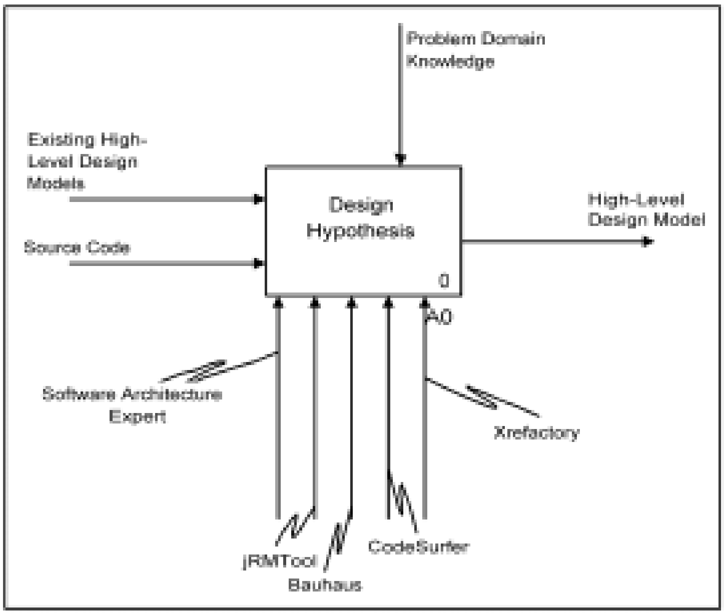

Figure 5-1. **Top-Level Diagram of the Process Model “Design Hypothesis”**

5.3.1 Inputs

Source Code: The input for this process model is the source code of the software. The source code and its associated source files are the basis for the analysis. The source code is needed when proving the hypothesis about the module structure.

Existing high-level design models:Existing high-level design models, even if they are not up-to-date are a good starting point. One can assume that at least some structures once designed have survived the evolution of the software system. Further analysis will prove whether and to what degree this assumption is correct.

5.3.2 Controls

Problem domain knowledge: For the establishing of high-level design models one needs problem domain knowledge. The more one understands the problem domain and the environment, the better one can model the high level design. For reaching best results, the cooperate work with one or several software architecture experts is advantageous and in my opinion necessary.

5.3.3 Outputs

High-level design model: The result of this process model is a high-level design model, which is up-to-date and presents several modules (or subsystems) including their interactions. In this model call and data relations have been taken into account.

5.3.4 Mechanisms

This process model needs advice of a software architecture expert (see at chapter 3.3.3) and uses the following tools:

- Bauhaus (see at chapter 4.3.1)
- CodeSurfer (see at chapter 4.3.3)
- jRMTools (see at chapter 4.3.7)
- RMGuiMapper (see at chapter 4.3.10)
- Xrefactory (see at chapter 4.3.14)

The analyst can extend the tool set if appropriate.

5.4 Activities

The diagram A0 (see figure) shows the main activities of the process model. It starts with an initial creation of a high-level design model where a hypothesis about the architecture of the software system is set up. Then an iterative approach is used where the following steps are repeated:

- The hypothesis is checked and examined by way of the result of the reflexion model.
- The reflexion model provides the analyst with the spots in the source code which need further consideration. These points are analyzed carefully.
- With the new gained structural information about the architecture, the high-level design model is refined and improved.

The process starts again with the execution of these three activities. It is repeated iteratively until the high-level design model is in a stable state where no or just little changes occur between the single iterations.

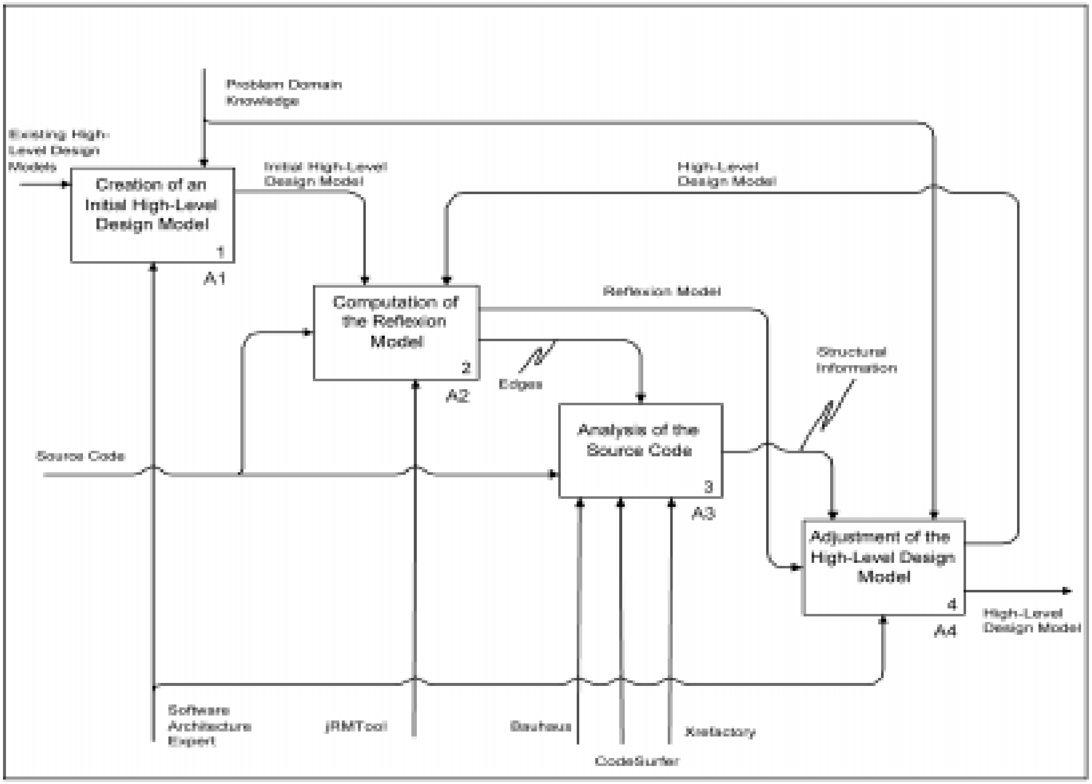

Figure 5-2. **The Process Model "Design Hypothesis"**

5.4.1 Activity A0.1: Creation of an Initial High Level Design Model

The first activity in this process model is the creation of an initial high-level design model. The high-level design model created in this activity contains the assumed modules of the software architecture and their relations.

5.4.2 Activity A0.2: Computation of the Reflexion Model

The high-level design model then is used for the computation of the reflexion model. The reflexion model is an approach to reveal the structure of an existing software system by computing a model that shows where the analyst's high-level design model agrees with and where it differs from the source code (see chapter 4.1.1).

Therefore the elements of the high-level design model are mapped to the source code. After the computation one can see the three different types of edges between the model and the source code, the convergence, the divergence and the absence edges.

The reflexion model is the output of this activity as well as any exceptions and peculiarities concerning the edges. These are the candidates of further investigation in the next activity where it is checked how to handle them.

5.4.3 Activity A0.3: Analysis of the Source Code

The next activity is the analysis of the source code. The peculiarities which attracted attention in the phase before, are now examined direct in the source code. The reflexion model provides the analyst with the points in the source code which have to be checked. A convergence edge for example should be investigated whether it is really a existing relation in the source code and not a false positive due to a mapping error. The same has to be done with the other edges. Three tools are used to perform the analysis: the Bauhaus tool, Xrefactory and Code Surfer (the detailed description of this activity in chapter 5.5.3 shows how these tools are used).

The hypothesis about the architecture have now been tested against the source code. The output of this activity is a better knowledge of the source code. Missing and redundant modules and module relations are identified so that the high-level design model can be adjusted with the gained structural information.

5.4.4 Activity A0.4: Adjustment of the high level design model

In this activity the high-level design model is adjusted with the knowledge gained from the activities before. The interaction of the different modules can be made more precise and the hypothesis can be proven as wrong or correct. The changed model is the input for further examination. The activities 1 to 3 are repeated until there are no more changes in the high-level design model.

The high-level design model can now be considered as correct or at least as partially correct due to the possibility that there may exist exceptions to the model which are now known but now yet resolved or that some source code elements may have been mapped twice which need to split up for a correct mapping.

Beyond the gained up-to-date high-level design model, the analyst now knows where the specific source code elements of a module are located physically and which routines and variables belong to it. Hence, the module itself is documented, too.

5.5 Decomposition of the Main Activity Diagram

In this chapter the main activities of the process model “Design Hypothesis” are decomposed in several more detailed activities. Each decomposition is explained by a diagram and describing text.

5.5.1 Diagram A1: Creation of an initial high level design model

5.5.1.1 Activity A1.1: Setting of the Scope

The first step is the setting of the scope for the specific instantiation of the process model. The analyst has to decide whether the whole software system is under investigation or just a part of it. When examining only a specific part, the rest of the software system can be left out for the analysis.

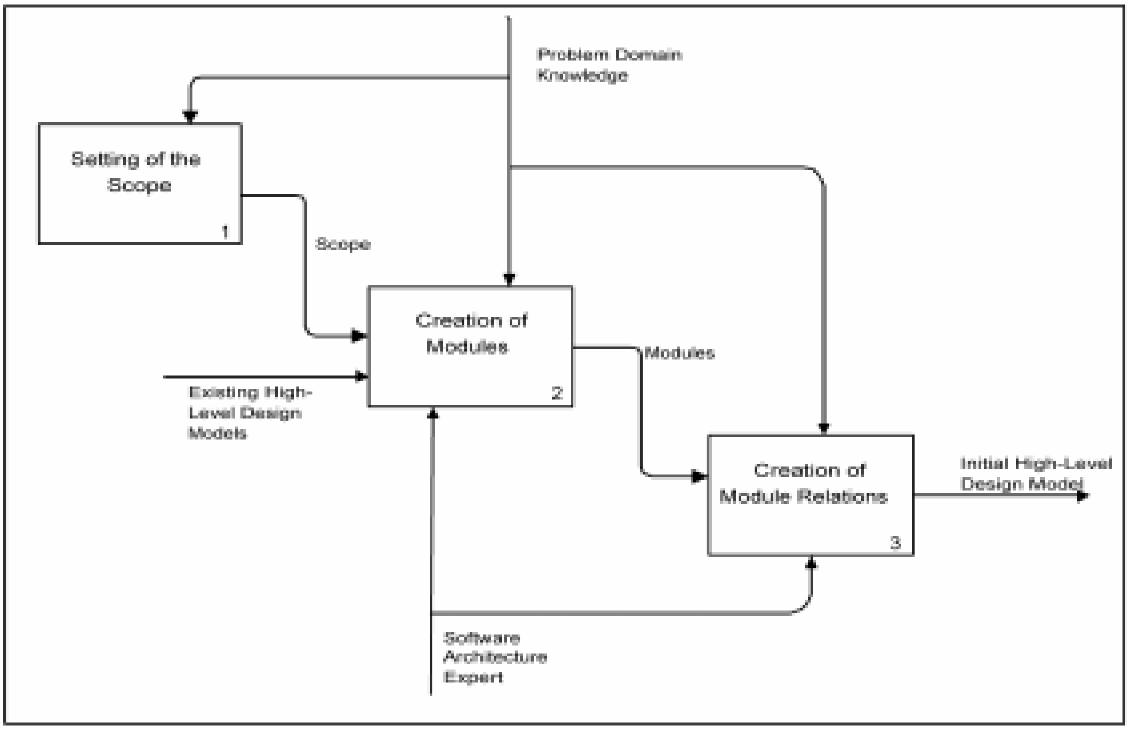

Figure 5-3. **A1: Creation of the Initial High Level Design Model**

The analyst has to identify which components (see chapter 3.1.10) and connectors (see chapter 3.1.11) belong to the chosen scope.

5.5.1.2 Activity A1.2: Creation of Subsystems and Modules

With the input of existing design documentation about the software architecture if available and controlled by the problem domain knowledge of the software architecture expert the software system is decomposed into several subsystems (see chapter 3.1.13) and modules (see chapter 3.1.14). Depending on the scope of the analyst can concentrate on the specific parts of interest. The functional behavior of the components and the control of the connectors has to be assigned to the subsystems and modules.

In this activity at least an initial set of modules should be found. If there is no existing documentation of the structure of the software system, then the analyst has to work together with the software architecture expert on the following points in order to reach this goal:

- which functionality does the system provide and how could it be decomposed
- which logical elements could be identified
- what might be a possible realization of a part or the whole system

The created initial high-level design model may be partly incorrect or even completely wrong. The next steps of this process model try to verify the assumptions and to improve the quality of the high-level design model.

5.5.1.3 Activity A1.3: Creation of Modules Relations

In this activity the set of modules is enhanced with two types of relations between two different modules:

- the call relation (see chapter 3.1.15.1)
- the data relation (see chapter 3.1.15.2)

The goal of this activity is to link the modules with each other in order to sketch out how they cooperate for the purpose of achieving a common task. The module relations can be propagated to the subsystem level. The software architecture expert advises the analyst in this step with his knowledge of the software system.

5.5.2 Diagram A2: Analysis of the Reflexion Model

5.5.2.1 Activity A2.1: Mapping

In this activity the mapping is created or modified. The mapping consists of the relations between the specified modules and their corresponding elements in the source code. Each source code element, the domain of this mapping, (see chapter 3.1.19) can be mapped to each module, the range. Mappings can include regular expressions on the names of the elements so that more than one element can be mapped at once.

This activity includes the decision of the analyst which source code elements are assigned to which specified module. He has to determine whether a source code element is a part of a module or whether it can be neglected. The mapping is supported by the RMGuiMapper where the analyst can map source code elements and the mapping is generated automatically.

The mapping will be more and more refined during the iterations so that wrong mappings of the first pass are modified and replaced by correct ones.

5.5.2.2 Activity A2.2: Translation into the Input Format

The source code has to be translated into a specific input format so that it is possible to compute the reflexion model. This has only to be done once, unless the source code is changed during the iterations.

This special input format is needed for the computation of the reflexion model. The complete source code is translated into a single file containing all source code elements, their position in the source code and all existent relations between the source code elements. This activity is taken over by the dumpreflexion tool, a component of the Bauhaus tool set.

5.5.2.3 Activity A2.3: Computation of the Reflexion Model

With the help of the jRMTool the reflexion model is computed. In this activity the source code is checked whether the assumed module relations in the high-level design model exist between the mapped source code elements. The convergence, divergence and absence edges are shown in a graphical view of the high-level design model. A number of found edges indicates how broad the interface between the modules is, i.e. how many times the specific relation was found. For each edge the proper references and the related nodes can be displayed.

5.5.2.4 Activity A2.4: Identification of the Edges

In this activity the different edges are investigated. Their location in the source code is provided by the jRMTool. A software system can be very large and so the number of edges can be so, too. In order to check the hypotheses it is necessary to examine the edges. The analyst will find a certain amount of edges which do not need further analysis because it is trivial to judge if they were modeled and mapped correct. But there may also be some edges which can not be checked at once.

These peculiarities are the output of this activity and they have to be investigated directly on the source code level.

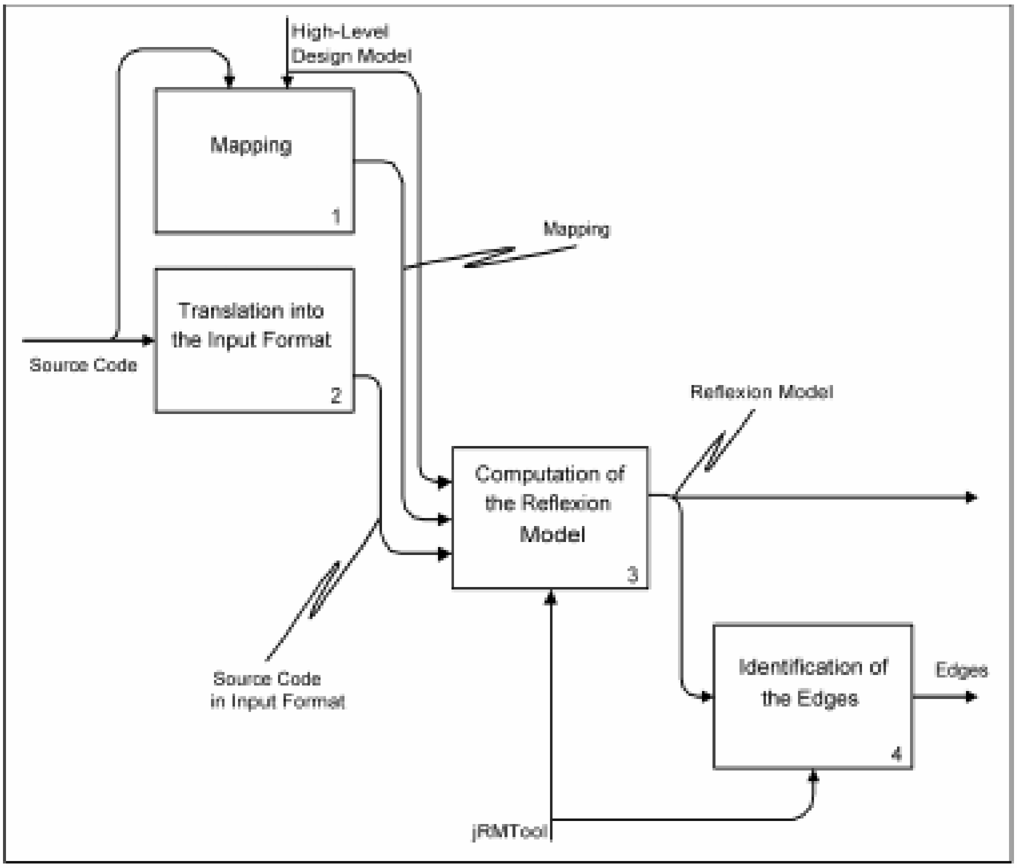

Figure 5-4. **A2: Analysis of the Reflexion Model**

5.5.3 Diagram A3: Analysis of the Source Code

The analysis of the source code is done with the help of three tools, namely the Bauhaus tool, CodeSurfer and Xrefactory. The edges, output of the preceding activity are processed here. Each identified edge has to be investigated with the help of at least one tool. The analyst should use the tool that is appropriate for the specific edge. He also can combine two or all of the tools.

The results of the source code analysis are used for further refinement of the high-level design model and for an improved mapping between the model and the source code. The goals of the following steps are the gaining of detailed structural information, the recognition of groups and the identification of false positives and true negatives.

Detailed structural information is needed for elements the analyst has been in doubt when he mapped them. The more information the analyst knows about an edge or the nodes of an edge, the less the doubts should remain.

During the analysis of the source code, groups of source code elements can be recognized. All elements of one group belong to same module, for example, an abstract data type and its access routines can be considered as a group. When the analyst investigates an edge, he can also investigate the nodes of that edge. A node that for certain belongs to a specified module can be taken as a starting point for a group recognition. The analyst has to find further source code elements referring to the starting point that belong to the module, too. This can be repeated until the elements are considered as non-group members. Then in the next refinement all members of the found group can be mapped to that module. The references of the starting point can be derived by naming conventions, program slicing, dominance analysis, source code browsing, any other reverse engineering method or a combination of these.

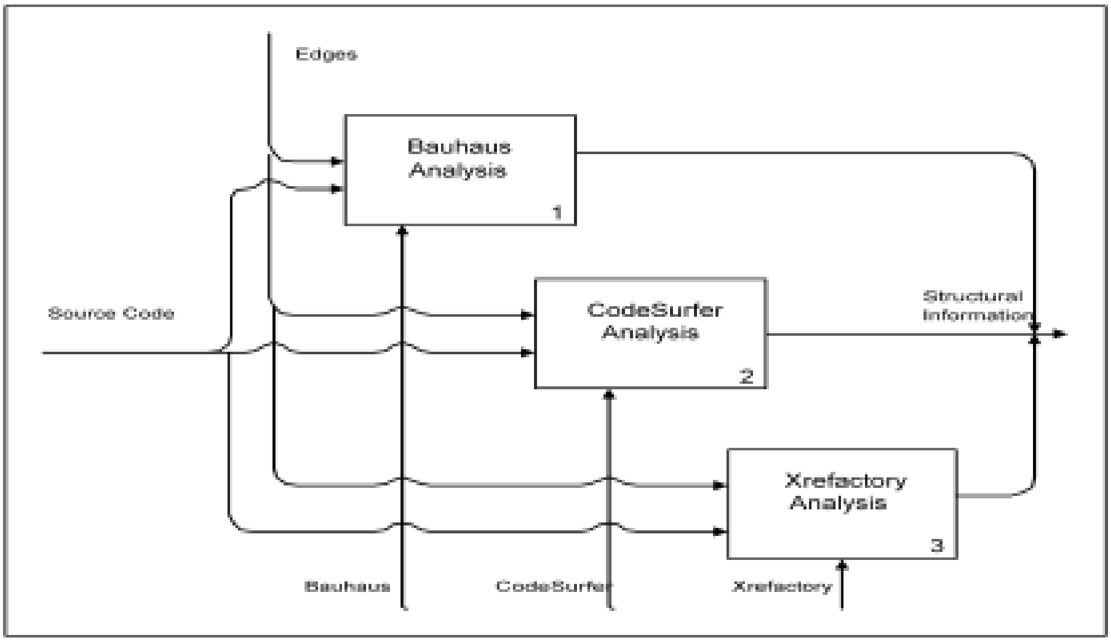

Figure 5-5. **A3: Analysis of the Source Code**

A false positive is an item of a convergence edge in the reflexion model that is shown because the high-level design model or the mapping was wrong. On the contrary, true negatives are items of divergence or absence edges that occur because the mapping was not complete or the analyst has forgotten to map something.

5.5.3.1 Activity A3.1: Bauhaus Analysis

The Bauhaus tool provides the analyst with a lot of information about the source code. For example, he can use the Bauhaus tool to examine the position of a edge in the call graph. The neighbors are taken into account to decide whether the nodes of an edge belong to the specified modules or the relations is modeled incorrect in the high-level design module.

Not only the call graph but the declaration of function and data types, the dominance tree, part type relations, etc. and the combination of different views are used for the inspection of the edges. If one of the methods mentioned above, does not lead to a decision, the analyst has to combine several of them.

5.5.3.2 Activity A3.2: CodeSurfer Analysis

The results derived from the preceding activity, the identification of the edges, are now investigated with the tool CodeSurfer. CodeSurfer, a tool using program analysis techniques for providing the user with intra- and interprocedural slicing (see chapter 4.1.3) and pointer analysis can also be helpful in investigating the edges and their dependent nodes. The program slicing is done on the statement level, but only routines and variables being in the slice are considered in the structural information.

For example, if it is certain that a node belongs to a specified module, then the analyst can slice the source code forward and backward with the node as a starting point. He can investigate the elements of the slice and decide whether these belong to the module, too or not. With this gained structural information, the quality of the mapping and the high-level design model is increased in the next iteration.

5.5.3.3 Activity A3.3: Xrefactory Analysis

The Xrefactory tool is used for fast navigation in the source code. The analyst can find out about the declaration and all other reference in the source code of the nodes and the edges. With this information it is easier to decide how to handle the computed edges of the reflexion model.

The structural information, the output of this activity is needed for the adjustment of the high-level design model.

5.5.4 Diagram A4: Adjustment of the High-Level Design Model

The adjustment of the modules and the relations is done iteratively because the two activities are strongly interconnected, and changes in the one activity may have consequences for the other activity.

5.5.4.1 Activity A4.1: Adjustment of Modules

In this activity the reflexion model is transformed in a new high-level design model. This happens under control of the problem domain knowledge of a software architecture expert and the structural information gained during the source code analysis. The modules are checked against the source code, and the hypothesis about the modules may have to be modified, otherwise it is considered correct.

In the first iterations a lot of changes are expected here, but the longer one executes the process model, the more the modules and their dependencies should stay stable.

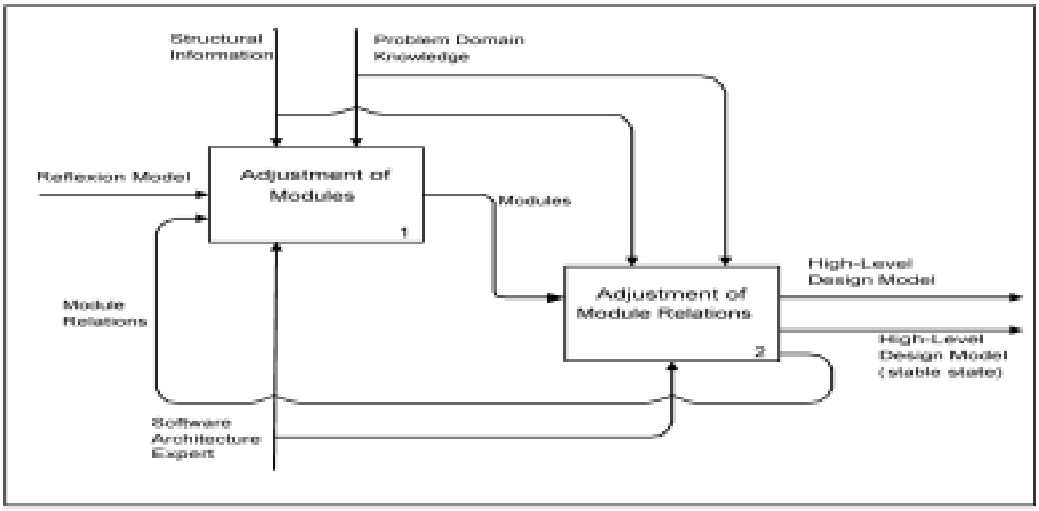

Figure 5-6. **A4: Adjustment of the High Level Design Model**

5.5.4.2 Activity A4.1: Adjustment of Modules Relations

In this step the relations between the modules are redefined. Due to the fact that the modules will change often in the early stages it is clear that the relations will change, too. Once the modules are in a stable state, one can concentrate mainly on the relations between the modules.

The goal of this activity is to improve the quality of the high level design model. The analyst has to consider to what degree the divergences have to be adopted in the new high level model. Some divergences can be left out, because these are unwanted exceptions to the chosen structure and can be marked for restructuring.

5.6 Applied Improvements

After the initial development of the process models some refinement have been made. The activity “Setting the Scope” (see chapter 5.5.1.1) is now part of the process model in order to have an explicit step where the analyst defines the realization of which components and connectors he plans to investigate.

The mapping when done manually can be a very long activity if the software system is large and the analyst has to work with many source code elements. Therefore a tool for the mapping was implemented. The tool provides the analyst with a view on the source code elements and the modules with a graphical user interface. He can now map elements to modules and vice versa with a mouse click and it generates automatically the mapping file. One can also see whether a source code element is already mapped via status information.

Another applied improvement is the ability to group several modules into a subsystem. So the analyst can on the one hand see how such a group is connected to the rest of the system and on the other hand he can analyze the dependencies within a group.

5.7 Possible Further Refinements

The main point is that the jRMTool can only display the reflexion model of exactly one level. Hierarchies are not supported yet. In order to reflect the software architecture, consisting of subsystems and modules, this feature is useful and reasonable. The next step for improving this process model should be the support of hierarchies.

Another refinement could be module relation attributes which represent the state of the examination. For example, there is the need of a relation attribute that indicates exceptions to the high-level design model, that are already known and documented. This has the effect that the analyst recognizes at once that these relations are already captured and he can concentrate on other parts of interest.

Furthermore the possibility of suppressing the elements of the reflexion model that are already in a stable state would accelerate the analyst in doing his work. When examining the system decomposition he can focus on the parts which need further attention. If necessary he can revert to the whole model.

Another point is the extension to the recognition and extraction of macros in the source code. Up to now, only the source code elements directories, files, functions and variables are taken into account, whereas macros are omitted. So the integration of macros would a desirable refinement.

5.8 Fields of Application

Beyond the possibility of recovering the architecture of a software system, the main goal for which this process model was developed, there are some other fields of application where the outcome of this process model can be used. These fields are presented in the following paragraphs.

The code ownership principle means that a developer (or a group of developers) is responsible for a specific part of the source code. Normally it is a part that is logically connected or physically located in the same file(s) or directory. However, there are a lot of possibilities how to split up a software system, but the important thing is that only the assigned developer is allowed to make changes in his owned code. In the case that some other developers need, for example, a new interface or change in the behavior of a function, they have to contact the responsible owner and both have to agree that this change request is useful, feasible and does not have impact on other features of the software system. With the help of the process model “Design Hypothesis” a module owner can examine to which other modules a function is connected and if there can occur side effects. Another aspect is the delimitation of the code belonging to different owners. The analyst can check if there are parts of the source code which are not assigned to a developer or which overlap, i.e. they are owned by two or even more developers. The analyst captures the owned code of each developer, and can compare these modules pair wise. Then he identifies where problems can occur, i.e. if two modules have interactions, then one has to pay attention to these when reimplementing, restructuring or refactoring the owned code. With the help of this process model these interactions are known and documented so no errors can be introduces due to the fact that one developer does not know that there exists another parts of the software that depends on his work.

Another benefit of the process model is that it is a good starting point for additional documentation like pre and post conditions, constraints, design rationale, behavior description and trans-view information (see also [Bachmann 2001]). As one knows which parts of the source code belong to a module and which interactions exist the analyst can use this view to make his documentation about the design decisions easier to read and more comprehensible. He can also assess unwanted design exceptions and make other developers aware of them. This leads to an better understanding of the problem domain for new employees and shortens the training times. And if a software system is properly documented, the maintenance is definitely better than if it were not. Later on after some further evolution of the software system the analyst can easily perform an up-to-date check for the documentation.

Furthermore this process model can be used when performing restructuring activities. Restructuring activities have to be well planned due to the fact that the software system should have at least the same (or even better) quality attributes than the old system. Since it is not an easy task to restructure a whole system at once, this is done step by step. The analyst responsible for the restructuring activities develops a plan how to transform the actual state of the software structure into the target state and which steps have to be done in what order. At milestones, after some major steps have been performed the analyst can assess with the process model “Design Hypothesis” to what degree the steps have been successful and how far the target state is still away.

Chapter 6 Case Study “Design Hypothesis”

ABSTRACT: The evaluation of the process model “Design Hypothesis” at Agilent Technologies is the topic of this chapter. Three different studies are presented, the first two examine the subsystem structure and the last one concentrates on one specific subsystem, the sequencer control.

6.1 Overview

The case study performed at Agilent Technologies was conducted to reveal weaknesses and to improve the process model. Data was collected for the purpose of having criteria for effort estimation and planning for further application of this process model. The experiences gained during the execution flew into the refinement of the process model. Overall, there have been two cycles, one before and one after the refinement. The latter was to verify that the adjustments in the process have been successful. Here, only the results of the second cycle are presented because they correspond to the process model “Design Hypothesis” described in chapter 5 and the results are similar to those of the first cycle.

The three different studies distinguish in the scope of the process model. Two analyze the subsystem structure of the firmware, and one concentrates on one specific subsystem. Hence, there are three different, assumed high-level design models including the subsystems and modules. And the reconstructed views on the software architecture are different for each instantiation, too.

The collected data and the measured times and efforts were gathered in the working environment described in chapter 4.2.

The input file for the reflexion was generated with the Bauhaus tool set. The compilation of the firmware took 10 hours and the linking of the intermediate language files to a resource-flow-graph (RFG) only 2 hours. The extraction of the input format from the RFG required 15 minutes by way of the dumpreflexion tool. Overall, 127753 relations have been extracted, 58065 call and 69688 data relations.

In the following sections several graphs of computed reflexion models will be shown. Each node stands for a module or subsystem, a blue edge represents the call relation (see chapter 3.1.15.1) and the red ones stand for the data relation (see chapter 3.1.15.2).

6.2 Directories as Subsystems

6.2.1 Approach

The source code of the firmware is physically split up in many files which are subsumed in a couple of directories. In the first development of the firmware, about 15 years ago, there was the design rationale that each subdirectory should contain exactly one subsystem. This assumption is the basis for this approach. The goal was to see to what degree the original made decomposition is still present.

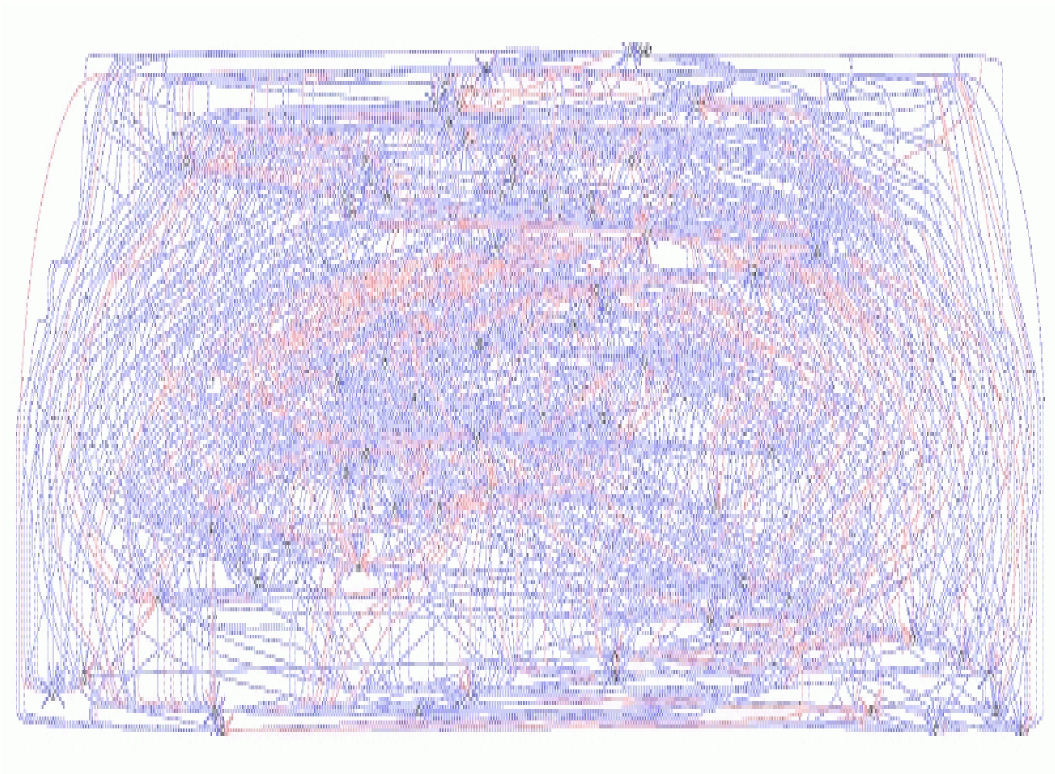

Figure 6-1. **Directories as Modules - an Overview**

69 directories have been modeled as a subsystem. No relations between the subsystems have been assumed in the high-level design model so that only divergences appear in the computed reflexion model.

This approach was executed in only one single iteration because the hypothesis that a subdirectory stands for a subsystem was realized then. No more adjustment to the mapping or the high-level design model was needed.

6.2.2 Results

Figure 6-1 shows the computed reflexion model. It contains 69 nodes (each node stands for one subsystem) and 988 edges between the nodes. As one can see in this overview the original subsystem structure got lost, there are a lot of relations between the specific subsystems.

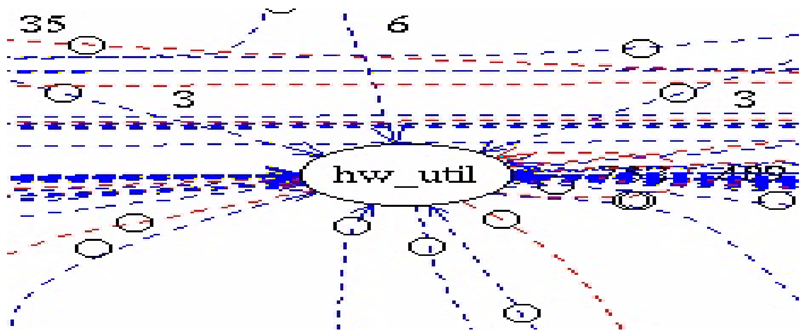

Figure 6-2. **The Subsystem “hw_util”**

Figure 6-2, a cutout of the whole, shows one of the subsystems with the highest number of incoming and outgoing relations. A change in this subsystem can have deep impact because of the high number of dependencies. It has a call relation to 40 other subsystems and data relations to 17. 55 subsystems have a call relation to “hw_util”, only 3 a data relation. Therefore a developer has to handle the subsystem “hw_util” very carefully to avoid unwanted side-effects.

As figure 6-3 shows there is one exception to the rest with respect to the number of relations to other subsystems. The subsystem “ibm_aldc_sw” is responsible for the simulation of a compression chip which compresses or decompresses the data sent to the device under test of the IC tester when operating in offline mode. It has only a small number of call relations into it and no data relation to other subsystems. The subsystem “ibm_aldc_sw” is called by “brst” with 5 calls, and by “mem_mgt” with 4 calls. The 15 call and 338 data relations within the subsystem “ibm_aldc_sw” indicate that there exists a high internal cohesion. So this subsystem can be taken as an example for a “good” architecture, since it fulfills the criteria proposed by [Fenton 1997] (high internal cohesion) and [Yourdon 1979] (low coupling with the rest of the firmware).

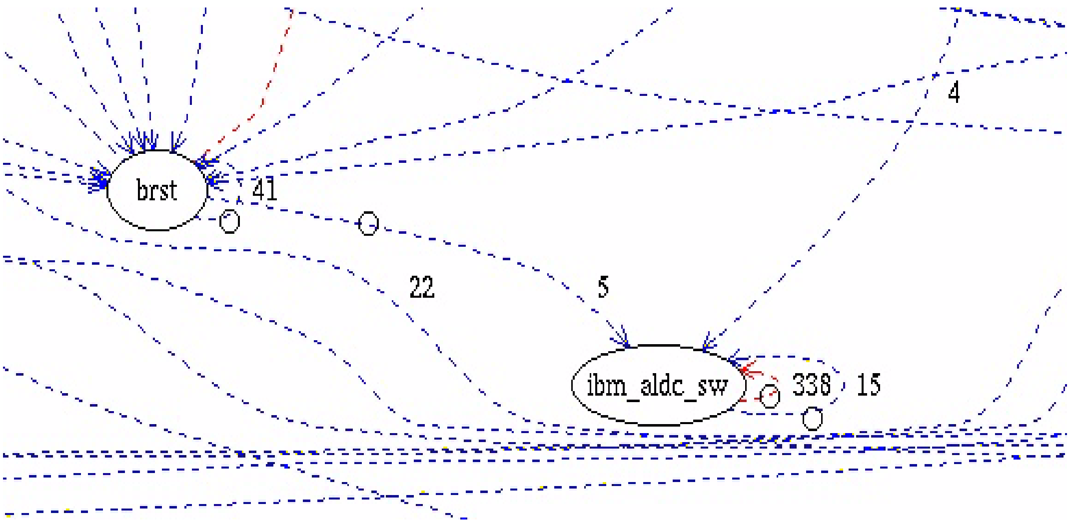

Figure 6-3. **The Subsystem “ibm_aldc_sw” and its Relations**

The reason for this exception is that the compression feature is the software which simulates a chip being part of the IC tester in offline mode, i.e. it was developed by another company, independently from the rest of the firmware. The developers of Agilent Technologies operate with the software via interfaces provided by the other company and the internals of the subsystem were not known until the chip accompanied with the simulation software was delivered.

It is very difficult to handle the results of this study for the reconstruction of a view on the architecture, no structure can be recognized at once and even after some time of analysis, the analyst still has problems with mastering of the large amount of relations. Therefore this approach is difficult to apply for improving the current situation, but it shows that improvements to the software architecture are really needed.

6.2.3 Collected Data and Developer’s Comments

For this study, there were 69 subsystems in the high-level design model. The mapping file also contained 69 entries, one for each subsystem.

The mapping was done via a shell-script. The creation of the script required half an hour and the execution took for less than a minute. The computation of the reflexion model took about 5 minutes, and the visualization of the results about 10 minutes.

The results showed the 69 subsystems and 988 relations, that means an average of about 14 per subsystem. The relations can be divided in 240 data relations and 748 call relation. The broadness of interfaces varies from 10 to several thousands, the average is well above 100.

The developers said that they expected a lot of dependencies between the specific subsystem (directories), but not as much as figure 6-1 shows. Furthermore, they stated that this approach produced an interesting view on the architecture of firmware. But they saw no starting position in this view for any activities in order to improve the situation. So they attested that this study is difficult to apply because of the inherent complexity of the firmware.

6.2.4 Lessons Learned

The process model can help to identify subsystems which have to be handled with caution due to a high coupling. If a developer wants to change the source code in a subsystem, he is now aware of the fact, that there might be a lot of side-effects due to this kind of analysis.

The results raised the awareness among the developers to take preventive actions. The once assumed decomposition into subsystems was even worse than imagined. Now it has been proven that there is really a need for the restructuring activities and that a view on the software architecture has to be reconstructed. First of all, modules with low coupling have to be identified and then grouped to subsystems.

Another point becoming obvious now is that the restructuring activities should start with a pilot project concerning only a small part of the firmware, because restructuring the whole firmware at once is a nearly impossible task due to the fact that there is such a high amount of dependencies between the source code elements. So the restructuring has to be done step by step.

The mapping on the subdirectory level is in case of the firmware too coarse-grained. The specific directories are interconnected far too often, so the analyst could not work with the high-level design model. It would have been too much effort to adjust the mapping and the high-level design model. But nevertheless this approach can work for other software systems where the design decision, that a subsystem is a directory, is more respected.

6.3 Global Model

6.3.1 Approach

This instantiation of the process model “Design Hypothesis” takes the coarse-grained architecture of the firmware (see figure 2-2) as foundation. In this figure, the execution layer of the firmware is divided in the executors, which realize the firmware commands, and the underlying utility routines. Three major iterations have been performed each consisting of several smaller steps.

In the first iteration, the executors and their initialization were mapped, and separated from the rest of the system. Due to naming conventions the realization of the firmware commands could easily be located in the source code. Further on, global modules, including a global database, a pin database, the error handling and some other things accessible by nearly all other source code elements, were assessed. In addition to this, a subsystem concerned only with the access of the hardware was synthesized. The source code elements related with the dispatching are mapped to module “MCD”, which stands for message constructor and distributor.

The identification of utility routines belonging to a specific feature or a set of related features was the task of the second iteration. The source code was mapped to a specific concept of the hardware elements of the firmware (see chapter 2.1.5), e.g. all vector specific source code was mapped to a subsystem called “vector”, or the level specific code was subsumed in the subsystem “level”. Step by step the high-level design model was extended by one of these concepts.

I performed the first two iterations alone only relating to the knowledge I gained during the vocational adjustment, informal talks to the developers and my work with the firmware source code.

In the last major iteration the single concepts (i.e. the subsystems standing for the realization of a specific feature) were grouped in one huge subsystem to reveal what the relations between the concepts as a whole and the other subsystems are. The mapping activities were refined with the help of a software architecture expert. This way the number of subsystems was reduced.

6.3.2 Results

During the first iteration, five subsystems with 21 relations among them have been created and mapped. The reflexion model (see figure 6-4) shows that there is not an absence edge, so the high-level design model does not contain a relation not found in the source code. But 10 divergence edge indicate that the model is not fully correct. The subsystem “hardware” has many relations to the other subsystems, which contradicts the high-level design model. The hypothesis was that the subsystem “hardware” operates only with global accessible elements and is accessed only by the firmware commands.

There exist some relations between the initialization (subsystem “inits”) and the realization (subsystem “executors” of the firmware commands which are not part of the high-level design model). The reason for this is that a few firmware commands check before execution that some other firmware commands are already initialized, and therefore they have call and data relations into the “inits” subsystems.

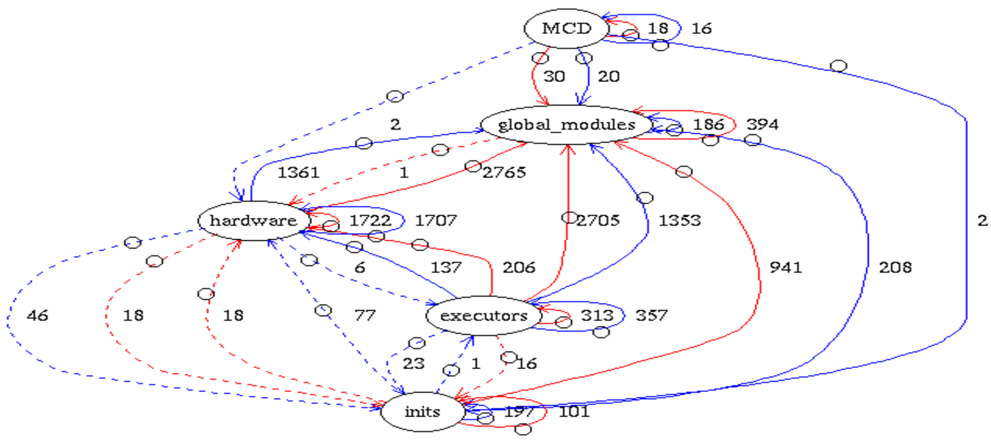

Figure 6-4. **The First Iteration**

There is no relation between “MCD” and “executors” due to the fact that the firmware commands are called via function pointers, that have not been taken into account when the input file for the reflexion model was generated.

Another finding of this iteration is that there exist a high amount of relations to the subsystem “global_modules”. In most cases the relations to it consist of more than 1000 different call or data dependencies. Since a lot of data dependencies exist, one can assume that data encapsulation as proposed by [Parnas 1972] should be applied more often. The developers later confirmed this.

In the second iteration, it was started with the five subsystems, defined the iteration before. The it was successively extended. For each concept of the firmware a new subsystem was introduced. Hence, as shown in figure 6-5, there are now 13 new subsystems, and an overall of 339 relations. The specific subsystems are highly related to each other with an average of about 19 outgoing relations per subsystems.

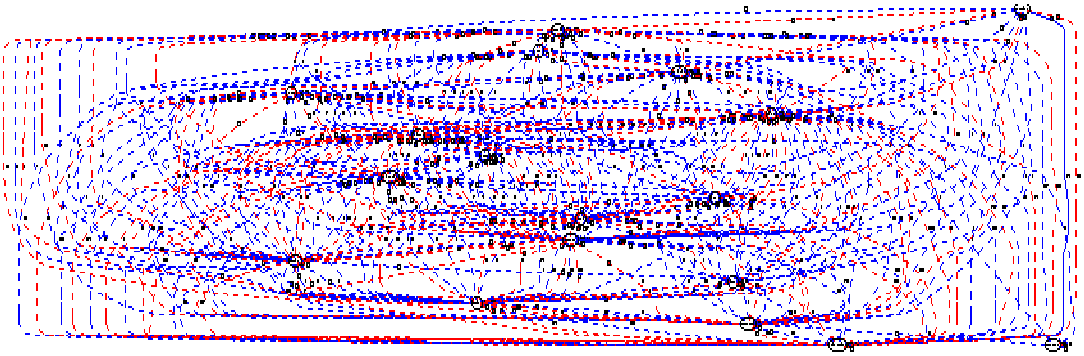

Figure 6-5. **The Specific Concepts of the Firmware**

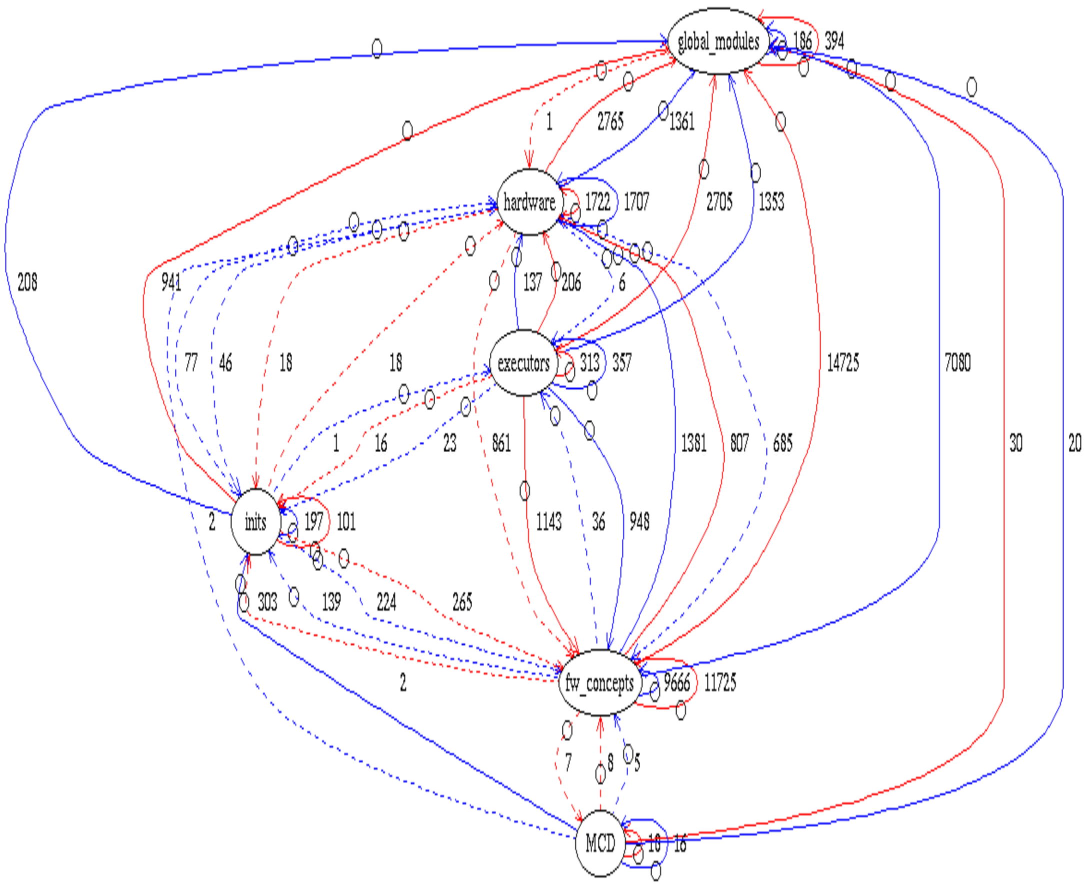

Figure 6-6. **Grouped Concepts**

Since the high number of relations shown in figure 6-5 is hardly manageable, all of the specific concepts have been grouped together in a single subsystem, called “fw_concepts” (see figure 6-6). Hence, in the third major iteration the number of modules is reduced as well as the number of relations. Now there are six subsystems, 29 convergencies and 20 divergences, i.e. resulting in overall 49 subsystems.

The new subsystem is connected to all other subsystems. Many cycles between the subsystems are still present, and some of them seem to be introduced by an incorrect mapping, i.e. some source code elements have been mapped to a subsystem where they do not belong to. For example, a whole file was assigned to a subsystem, but in this file there is a function that should have assigned to another subsystem.

The subsystems are too much coupled, so that a mapping on function and variable level would be necessary. Hence, this approach did not lead to a view containing a full reconstruction of the software architecture, but it showed some unwanted connections. No underlying utility routines of the firmware could be identified simply because a distinction of the subsystems of the specific concepts could not be made. Even the support of the software architecture expert in the third iteration lead only to slight improvements in the mapping activities.

But this view on the architecture exhibits the relations of the firmware executors, their initialization, and their relations to the other subsystems.

6.3.3 Collected Data and Developer’s Comment

The first iteration took approximately 8 hours. The duration was quite high, but the reason for this is that the mapping was done manually. Later on, shell scripts were used to accelerate the mapping activity.

In the second iteration, it took approximately 1 hour to integrate each new subsystem in the high-level design model and to map the source code elements to it, with the exception of refinement of the subsystems “hardware” and “executors”. Those required three hours each.

In the last iteration no new modules were introduced, just the mapping was refined. This activity required approximately 4 hours, performed by me and the software architecture expert. Here the a couple of mappings were verified in the source code, and due to the source code inspections with the tools Bauhaus, Xrefactory and CodeSurfer, several new entries have been inserted in the mapping file.

Regular expression matching provided a high number a matches between the mapping and the source code in the first two iteration. Then in the last iteration the mapping was refined by the tool RMGuiMapper. About 100 different maps were done in the first iteration, then in the second, there were approximately 1000 mapping entries. In the end the mapping file contained more than 2000 entries.

The visualization of the different reflexion models required always less than ten minutes since the number of the relations between the modules was relatively low.

6.3.4 Lessons Learned

The role of the software architecture expert is very important. His knowledge is definitely needed, because the analyst normally can not decide, whether a routine or a variable is to be mapped to one module or another. It has become clear, when working with a system as large as the firmware, the analyst has to cooperate with several software architecture experts since it is impossible for one person to have in-depth knowledge of all parts of the software system.

Another point is the bigger the subsystems are, the more imprecise and less accurate the mappings get due to the inherent complexity of the firmware. As a module or a subsystems comprises more source code elements, the number of possible relations increases. Therefore should smaller excerpts yield a better result.

The mapping done manually with editors or with shell scripts was not sufficient and took quite a long time, therefore after some iterations, the tool RMGuiMapper was developed to accelerate the mapping procedure, and hence, to improve the efficiency of the process model. The combination of manual mappings for regular expressions, the mapping tool for single source code elements and the shell-scripts for the automatic repetition of mappings with slight changes lead to a noticeable faster performance of the mapping activity than before.

This instantiation has shown that it is very difficult to reconstruct a view on a large software system as a whole. The main reason for that is when working with the whole system, the analyst has to handle significant more dependencies than when concentrating only on a limited sector of the software system. Therefore better results in the reconstruction of views on the architecture are expected when investigating smaller parts.

6.4 The Sequencer Control

6.4.1 Approach

The sequencer is a simple, programmable microprocessor chip responsible for generating vector data sent to the device under test. The subsystem “Sequencer Control” of the firmware controls the chip. This subsystem is part of restructuring project at Agilent Technologies. A developer already analyzed the actual state and will implement a new version of the sequencer control. The developer first set up a high-level design model and mapped the source code to the modules. Then he evaluated how his hypothesis fits to the source code.

In a second iteration, he then refined and adjusted the high-level design model with the knowledge gained. Beyond the sequencer control itself, he also modeled the relations of that subsystem to the rest of the firmware.

In the future, when the restructuring activities have been finished, it is planned to compare the new module structure of the sequencer control against the target state. For example, one goal of the restructuring activities is, that there are no more cycles between two modules and it will be proven with the help of this process model whether the goal is obtained or not.

The high-level design model, the modules, the assumed relations and the mapping were created by the developer alone. I only assisted in using the tools and gave information about mode of operation of the process model.

6.4.2 Results

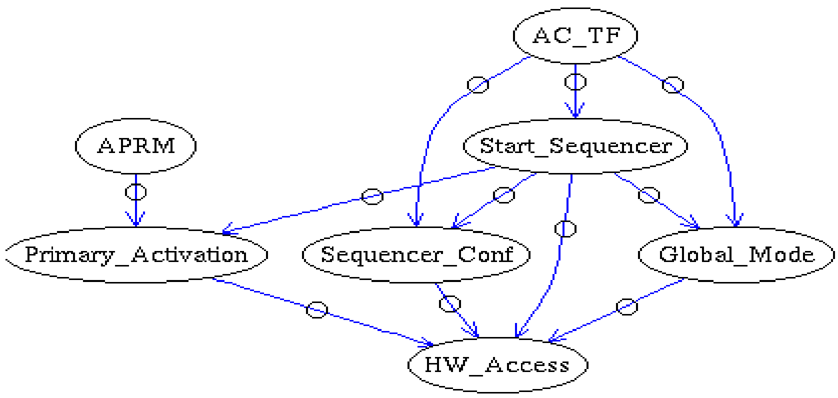

Figure 6-7. **The Sequencer Control - High-Level Design Model**

The high-level design model as specified in the first iteration is shown in figure 6-7. There are seven modules and 10 assumed call relations between the modules. The model is a idealized one, and it was expected that it is only partly correct.

Figure 6-8 shows the reflexion model after the first iteration. The number of relations between the seven modules has significantly increased, which is on the hand due to the fact that no data relations have been modeled, and on the other hand some absence edges and some divergence edges for the call relation are introduced by the reflexion model. Another reason for the high amount of divergences is the fact that relations of modules to themselves have not been modeled.

Four convergencies (APRM to Primary_Activation, AC_TF to Global_Mode, Start_Sequencer to Sequencer_Conf and Global_Mode to HW_Access) correspond to the high-level design model. Five absence edges show the ideal sequencer control can not be found in the source code. The divergences are partly known exceptions, for instance the data relation between the modules Primary_Activation and HW_Access was already known by the developer. He stated that this will be restructured in the future. But also some unwanted exceptions have been revealed, even though it was just the first iteration. The three calls between the modules Sequencer_Conf and Primary_Activation are not wanted. Other divergences occur because the mapping was not done carefully enough. These divergences will then disappear in later iterations when the mapping and/or the modules are refined.

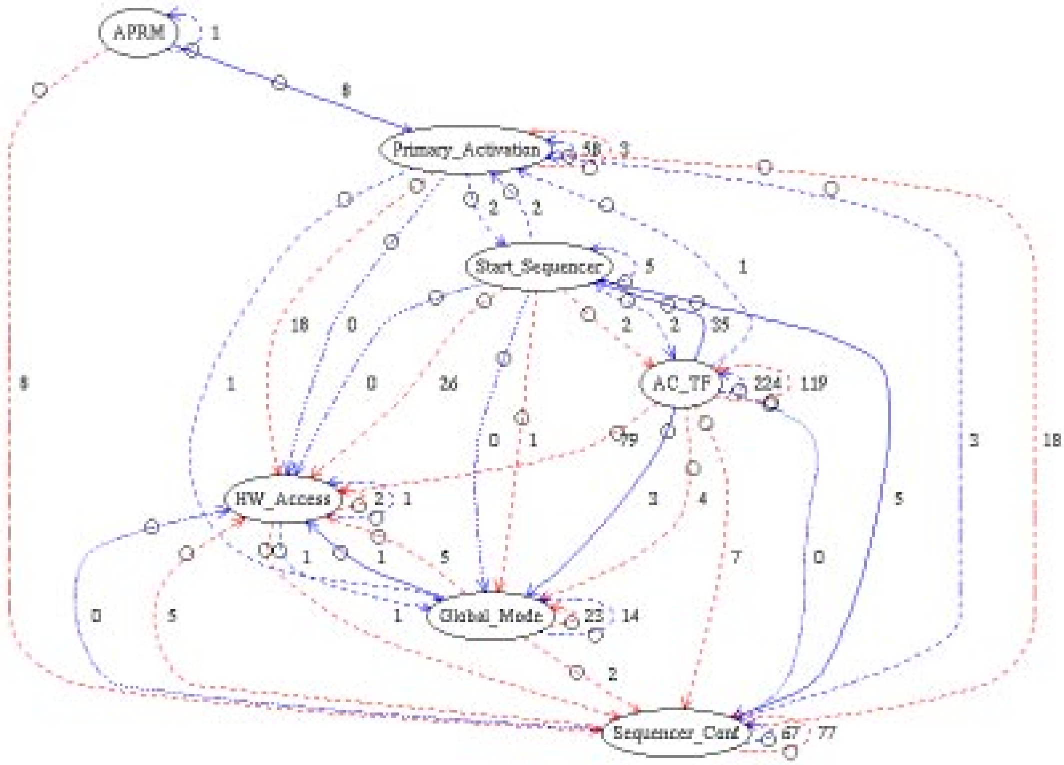

Figure 6-8. **The Sequencer Control - First Iteration**

Further on, several cycle can be found in figure 6-8. For example, there exists a call relation in both direction between the modules Primary_Activation and Start_Sequencer or

HW_Access and Global_Mode. The resolution of these cycles is planned in the restructuring activities. The developer now knows which routines he has to modify when doing so.

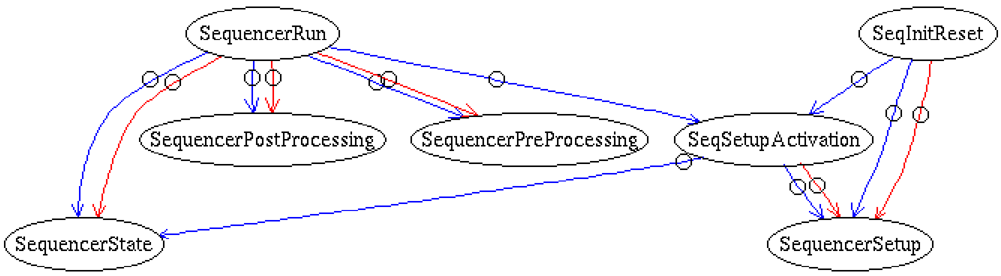

Figure 6-9. **The Sequencer Control - Adjusted High-Level Design Model**

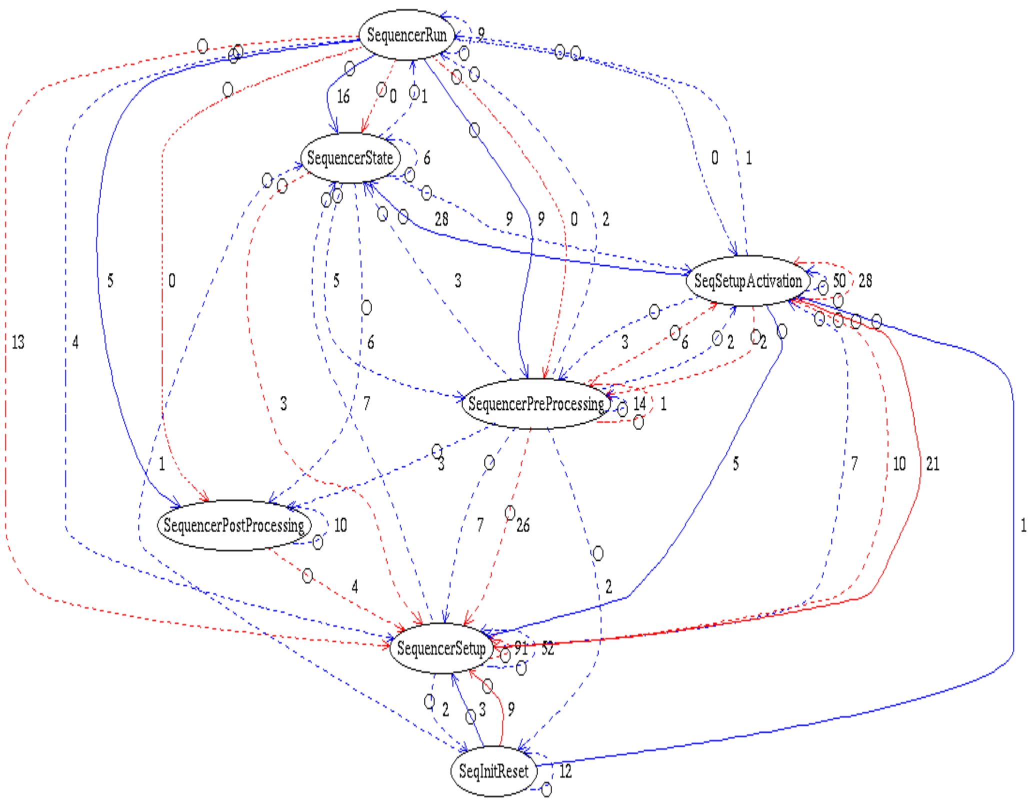

Figure 6-10. **The Sequencer Control - Second Iteration**

In the second iteration more time was spent on the creation of the high-level design model and the effort for the mapping, now better supported by a tool, was increased. That resulted in eight modules and 56 relations between the modules.

As shown in figure 6-9 the analyst has created 7 new modules, and omitted the modules of the first iteration. In addition to this, 13 module relations describe the calls (8 times, blue) and data dependencies (5 times, red) of the specific modules. This high-level design model is again quite idealized with the exception of the data relations. In the future, the goal of data encapsulation should be established which contradicts figure 6-9, i.e. no data relation should be present in the model, variables and members are accessed only by routines.

As figure 6-10 shows the three data relations from the module SequencerRun to SequencerState, SequencerPreProcessing and SequencerPostProcessing as absence edges. This means that there are no data dependencies (variables or member of types set/ use). Routines access those modules only via calls. The conclusion of this is that further data encapsulation is not needed. On the contrary to this, the analyst still has to attend the data relations to the modules SequencerSetup from SeqSetupActication and SeqInitReset.

Regarding the call relations in the computed reflexion model (see figure 6-10), it becomes obvious that there are several cycles indicated by mutual relations of two different modules. The analyst can now clearly recognize that cycles exist and where they are located in the source code. With this information the next step of the restructuring activities is significantly facilitated. Other relations, for instance from the module SequencerPreProcessing to SequencerPostProcessing are unwanted. These three calls contradict the hypothesis of the high-level design model.

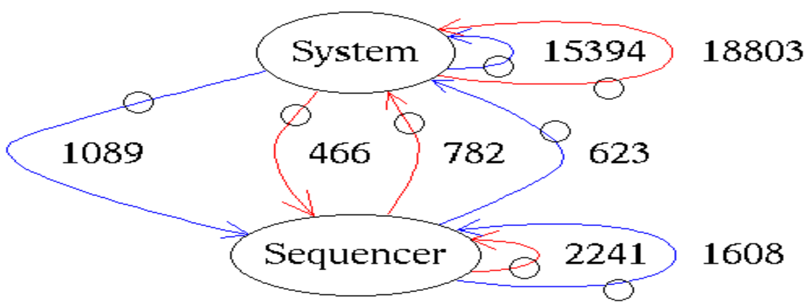

Figure 6-11. **The Sequencer Control - A Subsystem View**

The subsystem of the sequencer control and rest of the system as one huge subsystem is shown in figure 6-11. The subsystem “System” can be decomposed in 19 smaller subsystems. It has become obvious to the developers that when they start the restructuring activities, they have to regard the rest of the system. And they know now which routines definitely have to be included in a regression test after the termination of the restructuring and where these routines are located. In my opinion, the risk of the introduction of side-effects can be reduced because the developers are now aware of the problem.

When comparing the results of the two iterations, the developer stated that the second iteration provided him with even more useful information than the first one. The reasons for this underlie mainly in the increased effort spent for the high-level design model and the mapping. In addition, the experiences of the prior iteration improved the result as well.

6.4.3 Collected Data and Developer’s Comment

During first iteration of the investigation of the sequencer control, 7 modules and 10 call relations among them result in the high-level design model. The generation of the model took about 30 minutes. Then the source code was mapped to the specific modules, which required approximately 45 minutes. Overall, 515 different mappings have been applied to the mapping file. The computing of the reflexion model and the display of it took 15 minutes, so that totally, this iteration took 90 minutes.

The second iteration consumed three times more time, namely 270 minutes. The high-level design model was designed in 1 hour, the mapping required approximately 3 hours. Due to the fact that both the sequencer and the parts of the firmware that are related to the sequencer have been mapped, the number of maps noticeable increased. In this iteration, more than 10000 mapping entries have been made. The computing and displaying of the reflexion model now took 30 minutes.

The developer states that the process model gives a good overview about the current state and can give an insight in to what degree the source code meets the target state.

Further on, the developer tells that this process model and the results of the two iterations are very useful for his work in the restructuring project. And he also planned another iteration after the reimplementation of the sequencer control, in order to verify to what degree the restructuring activities have been successful. Furthermore, he would welcome the integration of the tools needed to apply the process model into the working environment at Agilent Technologies.

6.4.4 Lessons Learned

The process model allows the modeling of subsystems and modules, i.e. a hierarchy in the decomposition of the software system. This is not supported by the jRMTool which can only visualize one level at a time. Therefore an enhancement of this tool or another form of visualization is imaginable and should be addressed in the future.

The results for analyses in the small lead to very useful information about the software architecture. Insights can be gained, that ease the work of the developers. Working with small extracts leads to better results than working with the whole system, several analyses of smaller parts can be combined to reconstruct a view on the whole system and can lead to a coarse-grained view of the software architecture.

The role of the software architecture expert was in this instantiation of the process model played by the same person as the role of the analyst. Due to the limited scope, the quality of the mappings was very high, and almost no incorrect mappings were done.

Furthermore, this instantiation of the process model confirms the assumption that an iterative approach yields better results. Consequently the analyst should always plan to perform at least more than one iteration.

Another positive point is that the developer could apply the process model almost alone just after a short time of introduction. It is nearly no training needed, neither for using the tools, nor for performing the activities of the process model. This leads to a fast production of results.

6.5 Summary

The tools worked all properly with the source code of the firmware, even though sometimes it took some computing time, so that the analyst has to wait several minutes or hours (depending on the task) before he can carry on with the next activities. But fortunately, the activities which require relatively long computing time (i.e. the generation of the RFG file) have only to be performed once.

Some problems occurred with handling the amount of available information when working with the whole software system. When examining smaller parts of the firmware, the results were considered useful.

The feedback of the developers was positive and the results were useful. Hence, the process model “Design Hypothesis” fulfills its goals in reconstructing useful views on the software architecture. It is easily applicable to real world problems, especially with a defined scope and iterative refinements. Further on, the developers at Agilent Technologies already plan to continue the use of the process model in the future.

Table 6-1 gives a view on the collected data in comparison of all case studies. The second iteration of the sequencer is subdivided since for the results, one time only the sequencer was analyzed, and the second time the sequencer and its relations to the rest of the system have been modeled. The total time for the whole second iteration is the sum of both (5 hours).

Instantiation	# Subsystems or # Modules	# Relations	# Mappings	Total Time
Directories as Subsystems	69 S	988	69	1
Global Model				
Iteration 1	5 S	21	97	8
Iteration 2	18 S	339	1078	15
Iteration 3	6 S	49	2184	4
Sequencer Control				
Iteration 1	1 S / 6 M	42	515	1.5
Iteration 2 - Sequencer	1 S / 7 M	56	607	3
Iteration 2 - System	2 S / 19 M	4	10754	2

Table 6-1. **Summary of Gathered Data**

Chapter 7

Process Model "Bottom-Up Analysis"

ABSTRACT: The process model "Bottom-Up Analysis" will be introduced in this section. In detail, the external interfaces, the specific activities, improvements and possible fields of application will be discussed.

7.1 Overview

The process model "Bottom-Up Analysis" combines two different static analyses of the source code and an analysis of the directory and file hierarchy. Architectural information is derived during this bottom-up approach.

The goal of this process model is to identify layers (see chapter 3.1.17) in the architecture of the software system. The different layers have to be distinguishable by a separation criterion.

Therefore the analyst examines the features (see chapter 3.1.12) of the software system one after another. For each feature, the relevant dependency graph is constructed. The relevant dependency graph contains only elements which contribute directly to the realization of the functional requirement. Other elements, such as range checks or error handling are omitted. In addition to this, a concept analysis (see chapter 4.1.2 and appendix C) regarding common function and data types is performed and the file hierarchy is investigated. Then the results are delimited against the separation criterion and assigned to one of the layers.

These three different views of the features, namely the relevant dependency graph, the concept lattice and the file structure, are combined in a final analysis. Then the analyst identifies modules and their relations. If possible, the found modules are assigned to the different layers. The purpose of the process model is to gain a minimum coupling between these different layers in the software system and to reveal the inner structure of the layers.

7.2 Premises

The main premise for this process model is an explicit, distinguishable and well-defined separation criterion. Ideally it must clearly indicate whether a part of the source code belongs to one layer or not. The analyst has to know this for constructing the different layers because the less ambiguous the criterion is, the better the results will be. The separation criterion should be refined and adjusted, if possible.

For the investigation of the specific features of the software system, it is necessary to locate them in the source code. If that is not possible, the analyst has the problem that he does not know exactly where to start the examination.

In this process model, the analyst has to handle a huge amount of information about the software system under investigation. In order to manage this, he has to use filters that reduce the information to the minimum needed to implement the process model. Further information has only to be accessed if necessary.

The last premise is that the needed tools work properly in order to support the analyst during his work.

7.3 External Interfaces of the Process Model

7.3.1 Inputs

Source Code: For this process model, the source code is the basis for the analysis of the software system. The source code and its associated source files within the directory structure are the input for the analysis. The views on the source code are created in a bottom-up manner.

7.3.2 Controls

Problem Domain Knowledge: When producing the relevant dependency graph and dividing it into different layers, it is necessary to cooperate with a (or several) software architecture experts with problem domain knowledge. Only they can decide whether a performed separation is correct or not.

Concept Analysis: For detailed information about the concept analysis see chapter 4.1.2.

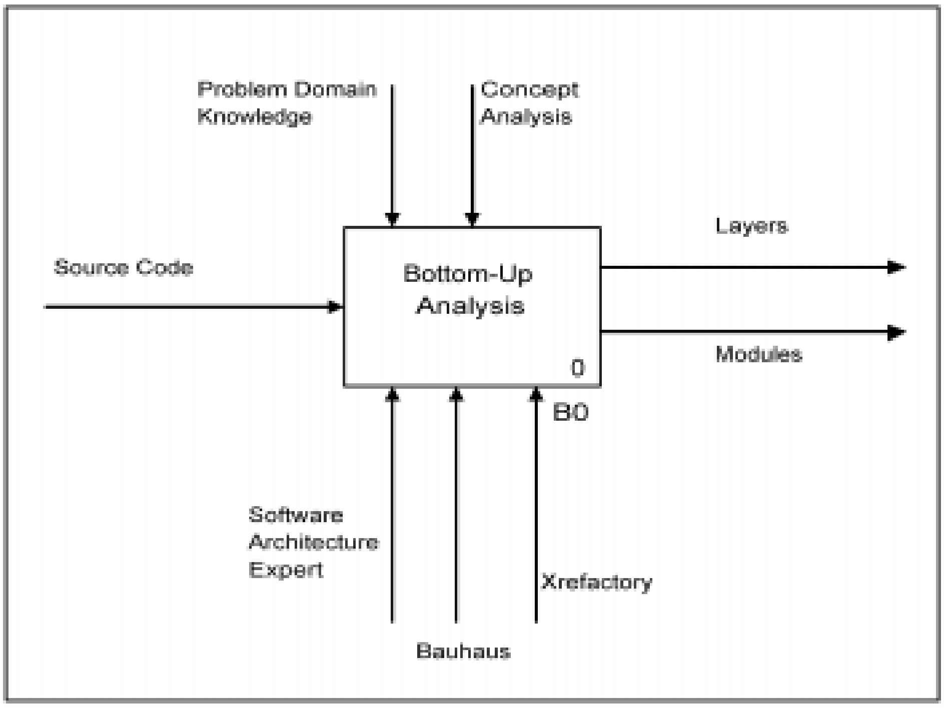

Figure 7-1. **Top-Level Diagram of the Process Model "Bottom-Up Analysis"**

7.3.3 Outputs

Layers: The process model produces the different layers delimited by the separation criterions. There may be only two layers, but it is also possible to separate several layers in one execution of the process model.

Modules: Each layer contains at least one module. The module is physically located in one or several files. The routines and the data structures of the module have been revealed. The relations between the identified modules are also tracked and part of the output of this process model.

7.3.4 Mechanisms

This process model needs advice of a software architecture expert (see at chapter 3.3.3) and uses the following tools:

- Bauhaus (see at chapter 4.3.1)
- Xrefactory (see at chapter 4.3.14)

The analyst can extend the tool set if appropriate.

7.4 Activities

The main activity diagram of process model “Bottom-Up Analysis” (see figure 7-2) is source code based. It can be divided into three main sections. In the first section, the analyst defines the separation criterion for the process model, then three analyses are performed concurrently.

In detail, the dependency graph and the directory and file hierarchy are examined, and a concept lattice is considered. The results are then segmented into layers. Finally, the three results are combined. The analyst identifies modules and their relations belonging to specific layers.

So the goal of this process model is to clarify the layers of the software architecture, to separate them and to reveal which functions and data types grouped in modules belong to them.

7.4.1 Activity B0.1: Definition of the Separation Criterion

The first activity of the process model is the definition of the separation criterion. At least one criterion has to be defined, but it is also possible to define several which would lead to a combined criterion. This step is very important, because the following analyses are based on the definition made here.

Each separation criterion divides the source code into layers. Ideally each source code element can be assigned to exactly one layer. But there may be source code elements where the decision can not be made. These are the members of several layers or none.

The separation criterion hence provides the analyst with a definition of the layers and rules whether to assign source code elements to one layer or to the other.

For instance, all source code elements that are part of a specific layer are named with the same prefix. Then this naming convention is a separation criterion that results from a design decision. It can be used to reconstruct this layer and to separate it from the rest of the software system. In this example the definition and the separation can be easily applied, but that is not always the case.

The separation criterions can be testable as well as logical or combinations of both. A testable separation criterion has the advantage that the assignment can be automated by tools. A logical separation can establish or hold up design rationale.

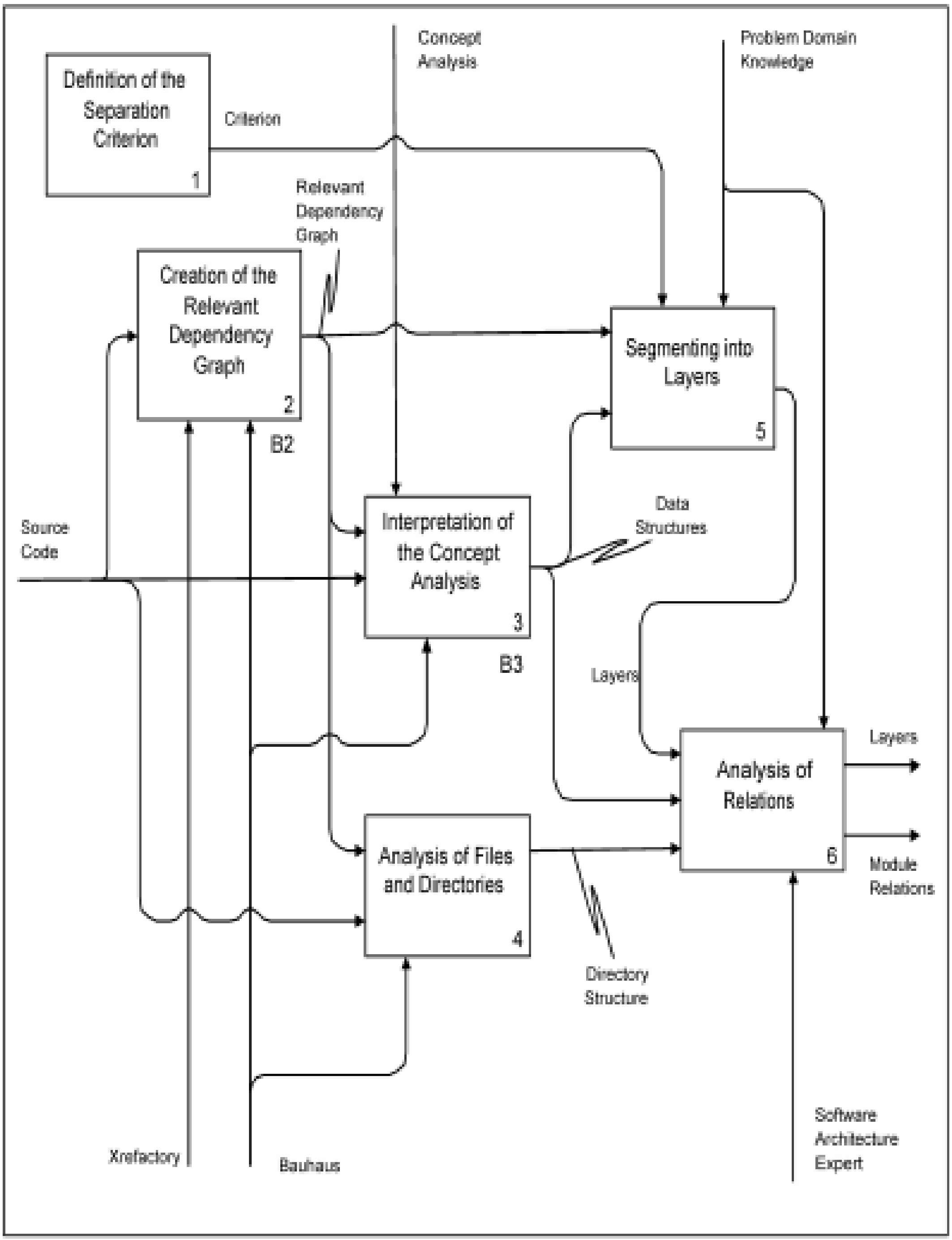

Figure 7-2. **The Process Model „Bottom-Up Analysis“**

7.4.2 Activity B0.2: Creation of the Relevant Dependency Graph

The activity in this process model is the manual slicing of the dependency graph. Normally a dependency graph of a software system is very complex and one has to handle too much information. But the point of interest here is not the whole dependency graph, just some relevant parts are needed to gain a better understanding of the system. Therefore the dependency graph is sliced manually with the help of the Bauhaus tool and the tool Xrefactory. Starting points of this activity are the calls of the specific features, a realized functional requirement. The analyst considers only the relevant calls, others, e.g. the error handling or range verification etc. are neglected.

When working with several features, there is one relevant dependency graph for each feature. These relevant dependency graphs may or may not overlay. So it is possible to work with a manageable subset of the dependency graph for each feature. This view on the software architecture differs for each feature (see chapter 7.5.1 for a detailed description).

7.4.3 Activity B0.3: Interpretation of the Concept Lattice

The method of the formal concept analysis is used in this activity of the process model. With the Bauhaus tool the following concept lattice is created: which routines access common members of types.

The goal is to reveal the relations functions and data types with a high cohesion. This is also taken into account in the final analysis at the end of this process model (see chapter 7.5.2 for a detailed description).

7.4.4 Activity B0.4: Analysis of Files and Directories

The directories and files including the functions used for a specific routine are investigated. The relevant dependency graph for each feature provides this activity with the routines as input. The location of these routines is the start for the examination. Further feature-specific routines or variables belonging to the feature might be identified in the same file. If a file provides other functionality as well, then this is an indicator that this file is shared by several features. The software analyst knows now what parts of the source code are needed exactly for a feature.

The output of this activity is the used file system structure for the investigated feature.

7.4.5 Activity B0.5: Segmenting into Layers

Then the relevant dependency graph is segmented into the different layers. The decision where to put a source code element should be made in cooperation with the software architecture expert. The problem domain knowledge and the defined rules lead to a deterministic processing of the source code elements. The elements of the dependency graph are segmented into one of the layers, or assigned to several.

The result of this step is necessary for the final analysis of the process model where the elements of the layers are grouped into modules. Therefore the determination where to segment the elements of the relevant dependency graph has to be done very carefully.

7.4.6 Activity B0.6: Analysis of Relations

In this activity the final analysis is performed. All information gained during the steps before is now combined and reconceived as a whole. The relations between the identified layers of the architecture, the revealed data structures and their location in the source files and directories are analyzed. The goal of this activity is to delimit the layers from each other following the separation criterion and to group the source code elements into modules. Sometimes it is necessary to inspect the relations and the data structure directly in the source code, in order to get a better understanding of it, or to verify the decisions.

If this is successful, it is possible to restructure the layers with well-defined interfaces and sound responsibilities. Source code elements or modules assigned to more than one layer have either to be subdivided or a decision is finally made to which layer it is assigned.

For example, there is a software system consisting of 2 files providing the user with the possibility of executing three different features. Now, this system should be divided into two distinguishable layers. The relevant dependency graph shows that the first feature uses three relevant function which really do the job. In the concept lattice, the analyst can see that all three functions operate with the members of same types. The separation criterion leads to the decision that all of these source code elements are assigned to layer two. The analysis of the files containing the types and functions results in the fact, that all of these functions are located in the same file. In addition to this, there are some more functions logically belonging to the same feature. Hence, all source code elements are grouped to one module assigned to layer two.

The investigation of the second feature results in two different modules for which the separation criterion is ambiguous, i.e. the analyst can not exactly determine where to assign the two modules. This is an indicator that the principle of a layered architecture is not observed. Here, a remodularization should take place while restructuring.

Feature three is examined and can be assigned to layer one. The relevant dependency graph of it contains several functions accessing several members of types. Here, the analyst can group the source code elements to one big module, or he can try to decompose the big module into several smaller ones.

7.5 Decomposition of the Main Activity Diagram

In this chapter the main activities of the process model “Bottom-Up Hypothesis” are decomposed in several more detailed activities. Each decomposition is explained by a diagram and describing text.

7.5.1 Diagram B2: Creation of the Relevant Dependency Graph

7.5.1.1 Activity B2.1: Feature-Centric Program Analysis

This optional activity has only to be performed if there is no possibility to identify the starting points. The feature-centric program analysis localizes the implementation of a specific set of features in the source code. The process combines static and dynamic analysis and is automated to a great extent. It maps the software system’s externally visible behavior to the corresponding parts in the source code. (for detailed information see [Eisenbarth 2002]).

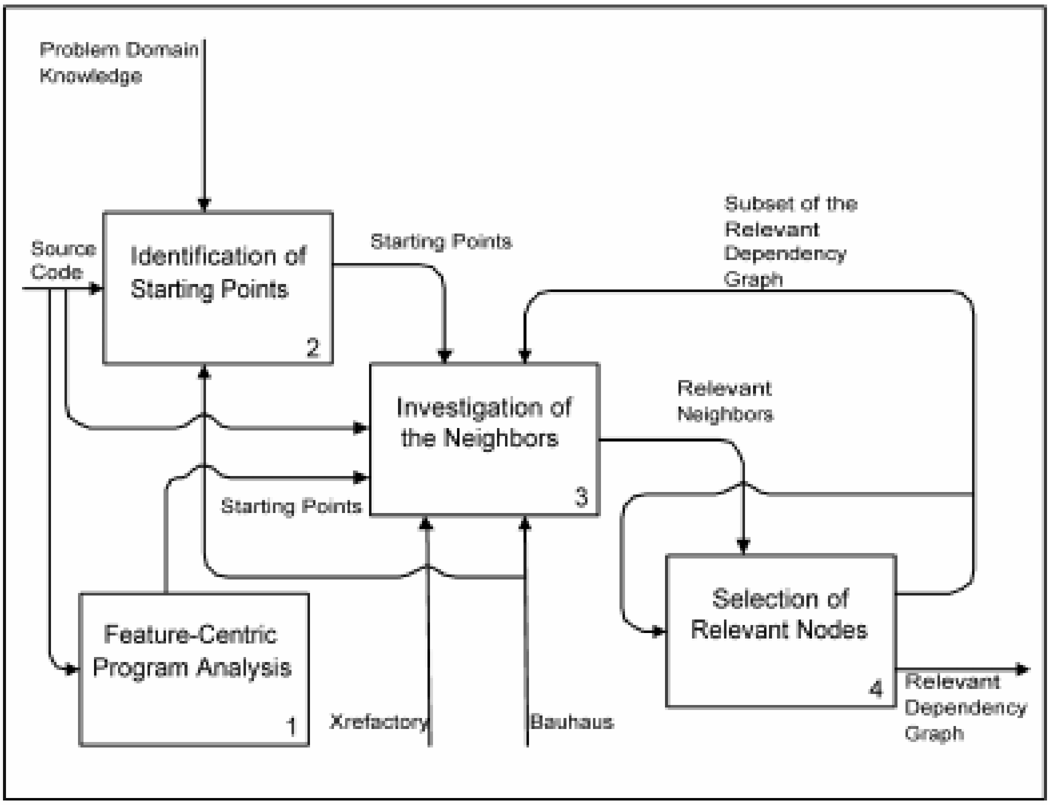

Figure 7-3. **B2: Manual Slicing of a Dependency Graph**

The starting points are the output of this optional activity for the following steps of the process.

7.5.1.2 Activity B2.2: Identification of the Starting Points

In this activity the dependency graph is automatically generated with the help of the Bauhaus tool set. The dependency graph can now be viewed and the analyst can select the starting points for the manual slicing of each feature.

For instance, the starting points have been identified in the preceding step, the feature-centric program analysis. Or a naming convention has been established and the feature can then be identified by a search about all nodes of the dependency graph.

The starting points are the output of this activity which is controlled by the problem domain knowledge of the software analyst. He has to know where to locate the points.

7.5.1.3 Activity B2.3: Investigation of the Neighbors

Beginning with the starting point of a feature the relevant dependency graph is constructed. It is determined for each node in the dependency graph whether it is relevant or not. A node is relevant when it contributes directly to the creation of an observable result for the user, i.e. routines that perform range checks, assert assumptions, or manage lists, stacks, etc. are not considered.

The software analyst displays all neighbors with the help of the Bauhaus tool. Then he investigates them directly in the source code with help of Xrefactory, applying a manual backtracking within the actual code segment. It is examined for indicators for relevance. If it is not decidable, then further investigation takes place by stepping deeper into the code. When there is a dead end or a found non-relevance, then it is backtracked up one level. Otherwise the neighbor is marked as relevant. See the algorithm in pseudo code below:

```
procedure investigate_neighbors(root)

  if is_relevant(root) then
    mark_as_relevant(root);
    add_to_relevant_dependency_graph(root);
    for each child do
      investigate_neighbors(child);
    end for;

  else if is_not_relevant(root) then
    mark_as_not_relevant(root);

  else
    mark_as_not_relevant(root);
```

```
    for each child do
      investigate_neighbors(child);

      if is_relevant(child) then
        mark_as_relevant(root);
        add_to_relevant_dependency_graph(root);
      end if;
    end for;
  end if;

end procedure;
```

The output of this activity are the relevant neighbors which build the relevant dependency graph.

7.5.1.4 Activity B2.4: Selection of Relevant Nodes

This activity processes the relevant nodes, constructs the relevant dependency graph, and merges the existing subset with the new neighbors. It may be possible that the neighbors of a subset of a relevant dependency graph may be investigated again, or that some nodes are no longer taken into account.

When there are no more neighbors left to investigate this repetition of the investigation and the selection comes to an end. The relevant dependency graph is the output of this activity.

7.5.2 Diagram B3: Interpretation of the Concept Lattice

7.5.2.1 Activity B3.1: Implementation of the Concept Analysis

The Bauhaus tool kit (the tool add_member_lattice) automatically performs the formal concept analysis on the source code and the concept lattice is produced. For detailed information about the concept analysis see chapter 4.1.2. The created lattice with the data structure information is passed on to the next activity.

7.5.2.2 Activity B3.2: Analysis of the Concept Lattices

The concept lattices are analyzed and the data structures, that means the routines that access common members of types are passed on as the output of this activity. The investigation is limited to the routines of the relevant dependency graph.

For example, the software analyst can identify the members of types used by the routines. The member and the corresponding types are highly connected to those routines and they can be grouped in one or several modules depending on the actual data structure.

So these revealed data structures are part of the input of the final analysis of the process model.

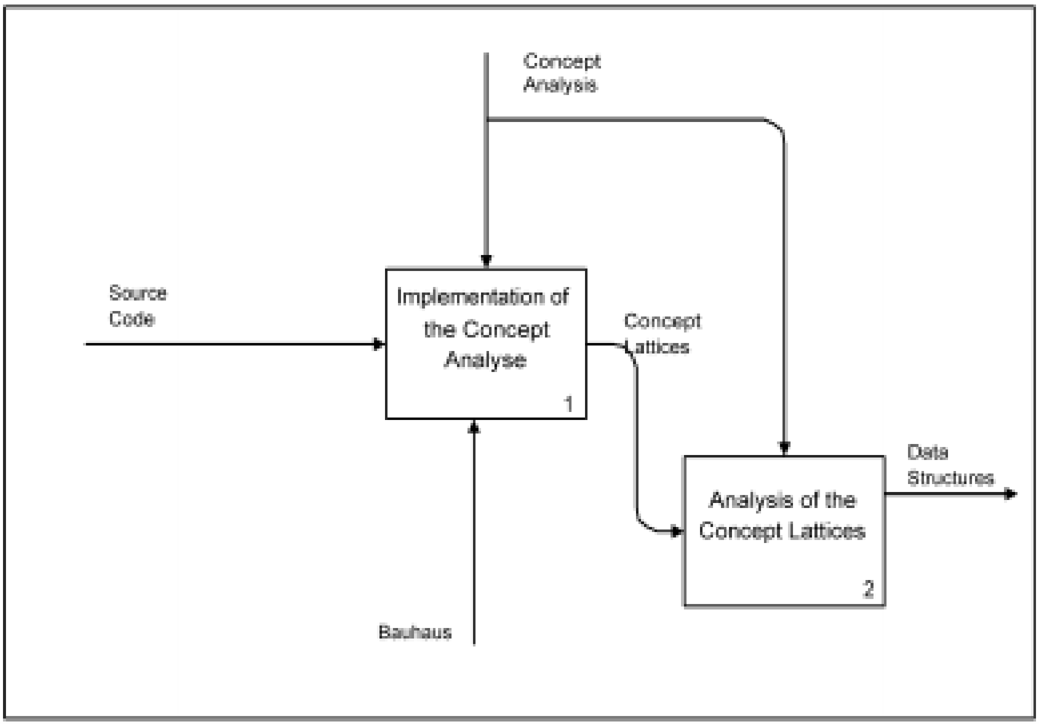

Figure 7-4. **B3: Implementation of the Concept Analysis**

7.6 Applied Improvements

One of the major improvements applied to the process model “Bottom-Up Analysis” is the generalization of it. In prior versions, it was centered on the specific situation of Agilent Technologies (see also chapter 8.1) and therefore only applicable for the problems there. Now the process model has been generalized so that it is applicable in a broad variety of situations (see chapter 7.8).

A new activity introduced in the process model is the explicit definition of the separation criterion (see chapter 7.4.1). Here rules for the separation of the layers are set up. A benefit is that the criterion can now be reviewed by other developers because it is documented.

Another new point is the optional activity “feature centric program analysis” (see also chapter 7.5.1.3). This is a possibility to extract starting points out of the source code, when it is not obvious which source code elements implement a feature. It may be needed due to the fact that typically existing documentation is outdated or the developers who programmed the feature are not available.

7.7 Possible Further Refinements

In addition to already applied improvements there are still a couple of slight weaknesses which could not be removed yet due to timing reasons. There exist a lot more analyses beyond the three performed (the analysis of the relevant dependency graph, the concept analysis and the analysis of the directory structure). In further refinements it has to be investigated whether and to what degree other activities can be integrated into the process model or not. For example, these techniques can be employed:

- Plan and cliche detection [Rich 1990]: A plan is a stereotypical coding pattern for algorithms or data structures used to implement frequently used concepts, comparable to design pat-terns at the design level. Hence, this technique can be used for the detection of modules which provide utility functionality.
- Clone detection [Baxter 1998]: In a large scale software system, a considerable fraction of the source code is duplicate code (“clones”). Detection and removal of such clones might help to simplify the internal structure of a layer.
- Atomic component detection [Koschke 2000]: An atomic component is a non-hierarchical component that consists of related global constants, variables, subprograms, and/or user-defined types. There exist several techniques for the atomic components detection which can be used for component recovery.
- Another concept analysis with other object and attributes could be incorporated in this process model in order to improve the results.

Another point is when examining large scale system the amount of available information grows quickly so the analyst can have problems in mastering the information. Not all available information is needed for the success of the process model. On the contrary, there exists much information that can be left out during the analysis. Therefore a concept is needed that filters and collapses the information, and places the information only at the disposal of the analyst. Then he can better concentrate on the relevant parts.

Furthermore, an automation of the decision, where to assign a source code element based on the separation criterion, would be very useful. This could be implemented, if the separation criterion can be tested by an algorithm. The job of the analyst then would be restricted to the cases where it is ambiguous.

The concept analysis is now done regarding all routines resulting in one single concept lattice. But since not all routines in this lattice are investigated, it could be sufficient to compute the concept analysis only for those routines, that are under investigation, i.e. the routines being part of the relevant dependency graph. So it could be interesting and easier to manage if the analyst performs several concept analyses instead of one. Then only the routines of the relevant dependency graph of each feature will be regarded, while the is left out.

7.8 Fields of Application

With this process model, the analyst can recover different layers of the architecture of a software system, but there are further fields of application described in the following sections.

Whenever porting activities are planned for a software system, i.e. the adjustments to a new version of the operating system, then the analyst can use this process model to identify the source code elements where changes may occur. The rules of the design criterion for the decision have to group the operating system dependent source code together leading exactly to two layers. Some source code elements can be assigned to both layers, here the porting activities have to be done carefully. With the help of this process model the developer knows where changes may occur and the porting activities are support better than without.

Another point of concern the use of design patterns (see also [Gamma 1995]), for instance the facade pattern. The design criterion has to match the facade, one layer is behind the facade, containing the interface and the functionality, and the other layer is the rest of the system, which should only access the functionality through the interface of the facade. The process model “Bottom-Up Analysis” helps the analyst to check whether the facade is broken or not. When there are module relations into the facade layer not using the interface, then it is obvious that the design pattern is outflanked.

Chapter 8 Case Study “Bottom-Up Analysis”

ABSTRACT: The case study for the second process model is presented in this chapter. Several firmware commands were investigated and analyzed in order to show that the process model works properly.

8.1 Overview

In order to evaluate the process model “Bottom-Up Analysis” (see chapter 7) a second case study took place at Agilent Technologies. Again, in order to support effort estimation and planning activities, data was collected. The experience gained in the course of the two cycles of the case study flew into the refinements of the process model and have been verified in the latter cycle. In the following sections, only the results of the second cycle are presented.

The separation criterion for the evaluation was the establishing of two layers in the firmware, one for hardware dependent source code elements, and one for those elements that are independent of the hardware (see chapter 8.2 for a detailed definition of the criterion). Therefore the features of the firmware have been investigated. A feature in the context of the firmware stands for a single firmware executor, since these provide the users with a realized functional requirement, which produces observable results. The starting points for the firmware commands can be easily found since the implementation of a firmware commands follows a naming convention. This case study dealt with the examination of several firmware commands, namely “mema”, “tstl”, “aprm”, “etim” and “xclk”. These five firmware commands differ in complexity ranging from low to high and in functionality ranging from little to large. In this classification functionality measures how much work is done, and complexity stands for the difficulty of the implementation of the specific firmware command. The call graph, a subset of the dependency graph, is used to construct the relevant call graph (see chapter Algorithm 7.4.2) of the specific firmware commands. The relevant call graph includes only those routine which the firmware command needs to perform its job. Routines like error handling or so can be omitted. This leads to a reduction of the call graphs of the firmware commands.

The following sections show several figures of intermediate and final results. Nodes can stand for routines (colored red squares), types (blue triangles) member of types (in green squares) and concepts (blue squares). Edges (not to be confused with the edges of the reflexion model) interconnect the nodes. The meaning of the edges will be given in the context of the figures.

The collected data and the measured times and efforts were taken under the working environment described in chapter 4.2. The values may differ on other software or hardware configurations.

For all analyzed firmware commands, the following values are the same. The generation of the resource flow graph (RFG) required 10 hours for the compilation of the firmware and 2 hours for the linking of the intermediate language files to a resource-flow-graph. Computing the concept analysis and adding the result to the RFG took approximately two hours.

The call graph of the firmware contains 13164 routines and 57415 edges, i.e. there have been 57415 different calls among the routines. The concept lattice establishes an order of 22034 concepts, in a flat view of all concepts including the objects and the attributes the analyst could see a total number of 43765 nodes and 355071 edges.

Loading an RFG can take up to 10 minutes with the Bauhaus tool set, depending on the number of different views saved in the RFG. The same time is required for the saving. The visualization of the call graph lasts for 5 minutes, while the concept lattice needs 15 minutes until it has been displayed. The copy and paste functionality is related to the amount of nodes and edges that are handled. For instance, pasting 100 nodes into a view requires about 10 minutes, while with about 1000 nodes and about 3000 edges among them it takes a little more than an hour for this operation.

8.2 What is Hardware Dependent and What Not?

The firmware is responsible for the control of the IC tester. The testers are enhanced and extended with new chips, more memory, or changed configuration at least once every two years. Of course, the software has to be adjusted each time the hardware of a tester is modified. Therefore the porting activities should be facilitated by dividing the firmware in a hardware dependent and a hardware independent part, which remains unchanged whenever such a porting occurs.

In the firmware, so the software architecture expert stated, there are two different indicators for hardware dependence. One is a naming convention: several routines and variables related to the hardware start with the prefix “FW_HW”. “FW_HW” represents a data structure containing all registers of the firmware. So all routines accessing this data structure are considered as hardware dependent.

The other is a logical assumption. Hardware dependent are those source code element that represent a physical property of the IC tester. For example, a variable standing for the exact number of waveforms is hardware dependent, as well as the direct access of a register or the concrete voltage level, hard coded in the software is hardware dependent, too.

Since in a firmware all source code is somehow related to the hardware, the distinction for hardware independence is made in an abstract representation of the values. For instance, variables or types representing pins are not hardware dependent, if they do not know details about how many pins are physically possible.

An example for the distinction is the voltage level of a digital channel: the level voltage can be set to low and high values. The abstract representation of low and high is hardware independent since it is not known for what values high stands for. Due to different test configurations, the concrete values for a high voltages can differ. So the concrete specification of the high voltages can be found in the source code, too. There are source code elements which stand for three volts, other specify five volts for the high voltage. And other routines check if the voltage level is within a certain range. If the concrete values are known in a routine, then this is an indicator for hardware dependence. These source code elements have direct consequences on to the hardware, and hence, they are hardware dependent, in contrast to those, which only operate with the abstract value high voltage. For instance, if there is a loop in a routine which is repeated six times because there six edge timings to be processed, then this routine is hardware dependent because the number six is can be changed when adapting the source code to new hardware requirements.

To paraphrase this chapter, hardware dependent are the parts of the source code which have to be modified when there are physical changes on the hardware side.

8.3 The Firmware Command “mema”

8.3.1 Description

The firmware command “mema” (MEMory Access) is a command which performs relatively little functionality, compared to the other firmware commands, investigated the following sections.

The command “mema” accesses the “FW_HW” data structure and writes a specified value into a given hardware location. First, addresses will be checked for legal range. The corresponding query reads a value from a specified hardware location.

8.3.2 Results

The firmware command “mema” and its query have a call graph consisting of only 11 routines (see figure 8-1) and 14 calls among the specific routines. The slicing of a relevant call graph could be neglected since 11 routines are manageable without further filtering of the amount of information. The two selected, green nodes show two hardware dependent routines, namely “setjmp” and “signal”.

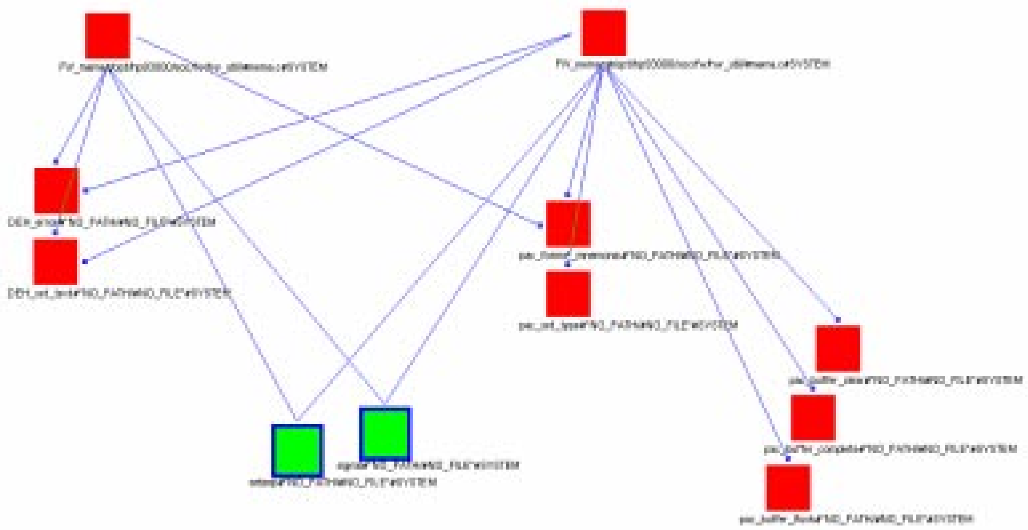

Figure 8-1. **“mema” - The Call Graph**

These two routines are responsible for signal handling and non local jumps. The writing of the value to the address happens in “mema” itself.

The other routines (colored in red) are hardware independent. The nodes with prefix DEH_ display error message, and the others with the prefix pac_ are responsible for parsing and constructing of the firmware command and its parameters. For this feature the segmentation can be easily made. Since no members are accessed, the concept lattice analysis can be neglected for this firmware command.

The analysis of the files shows that both the firmware command and the query are located in the same file, so that the analyst can group both in a module concerned with the “mema” functionality.

8.3.3 Collected Data and Developer’s Comment

The investigation of the “mema” firmware command was done in half an hour. The time was spent in visualization of the “mema” specific call graph (about 15 minutes) and approximately 15 minutes were required for the verification on the source code level.

The developers stated that the firmware command “mema” implements almost no functionality so that it is not astonishing that there is no interaction with any member of types.

8.3.4 Lessons Learned

For firmware commands like “mema” which perform only small functionality (in this case the writing of a value into a specified hardware location) the process model is oversized. The overhead and effort for computing the concept analysis and the RFG does not justify the results. The analyst in this case can obtain the same results as shown above in faster time with source code inspections. Therefore this process model breaks a fly on the wheel when analyzing firmware commands with a complexity like “mema”.

But even though the process model is over-dimensioned, it produces nevertheless useful results. And when analyzing several firmware commands of low complexity and little functionality, then it can pay off to use this process model because of expected synergy effects.

8.4 The Firmware Command “tstl”

8.4.1 Description

The firmware command “tstl” (TeST Label) selects the label of the test on which the vector edit commands work. The label is selected from a list and can be modified in a following edit operation.

This command has medium complexity and little functionality.

8.4.2 Results

During the analysis of the firmware command “tstl” one module has been identified. This module implements a stack to push and pop the sate of the sequencer. Reducing the call graph of the firmware command “tstl” of the 1081 call nodes and the 3151 edges between these nodes, there remain only 76 nodes and 101 edges of “tstl”. It is an indicator that the sequencer state stack is a module that is quite large.

The relevant call graph (see figure 8-2) of the firmware command “tstl” consists only of two calls. The routine “FW_get_test_label_by_name” extracts the label from another underlying utility module, a tree. Since this tree is a generic data structure of the firmware, it was not considered for the relevant call graph. Since no hardware call has been revealed for this specific firmware command, it is assigned completely to the hardware independent layer.

“tstl” operates with 15 different members of types. These members are attributes of an overall of 1068 concepts, that indicates that these members are widely used throughout the source code of the firmware.

The analysis the file containing the “tstl” firmware executor showed that there is more functionality included related to labels performed by other firmware commands. Further analysis will show whether the other commands operate with the same members of types, but the other commands were not examined in the case study.

Figure 8-2. **“tstl” - Relevant Calls and Related Members of Types**

8.4.3 Collected Data and Developer’s Comment

The construction of the relevant call graph took 2 hours. The investigation of the members of types also took two hours. The rest of the overall time of 8 hours was needed to display the different graphs and to manipulate them.

That the two underlying modules are utility modules was confirmed by the developers. The tree module is hardware independent as well as the whole command “tstl”. The sequencer state stack was not examined further during this case study due to time limitations.

8.4.4 Lessons Learned

It makes sense to construct the relevant call graph, since the number of routines that remains for analysis can be reduced significantly. Therefore the analyst can concentrate on the source code elements which really perform the task. Later, the other elements can be investigated in further analysis, but this should be an easy task since the main decisions where to assign and how to divide the source code elements have been already made with the relevant call graph.

There are some firmware commands like “tstl” which consist of source code elements not being part of the hardware dependent layer. These commands do not access the underlying hardware, but perform logical tasks, e.g. the setting of a label, or a query of a value not related to the hardware.

8.5 The Firmware Command “aprm”

8.5.1 Description

The firmware command “aprm” (Activate PRiMary sets) activates the primary set values in the hardware made known to the firmware by another command. Sets specify the conditions for the test case, i.e. which voltage level, which waveforms etc. are used. Several different sets are downloaded into the tester due to performance reasons. During the test, the activated sets can be changed. The “aprm” command activates the specific sets.

It is a firmware command with medium complexity and medium functionality.

8.5.2 Results

The total number of 1114 nodes with 3081 calls among them build the call graph of the firmware command “aprm”. Again the sequencer state functionality with push and pop operations can be identified as a utility module used by this feature as well. The reduction of this utility module leads to the relevant call graph, in which only 11 nodes and 17 relations remain (see figure 8-3).

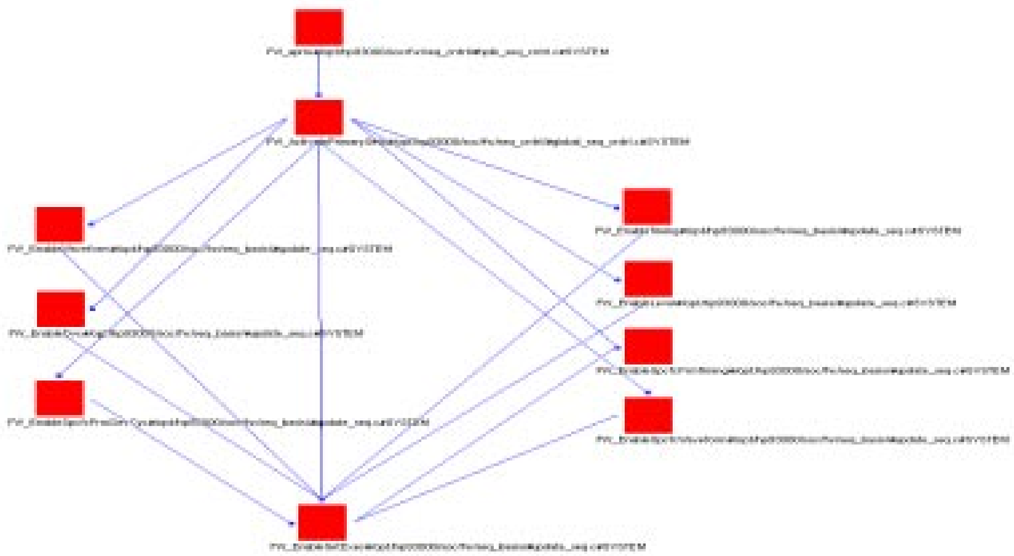

Figure 8-3. **“aprm” - The Relevant Call Graph**

Figure 8-4. **“aprm” - Hardware Dependent Routines and Members**

The relevant call graph is divided in several “FW_Enable” routines. Each of these routines prepares the activation of the specific set. The calls flow together in the routine “FW_EnableSetExec” which does the activation of the specific set.

Figure 8-4 shows the hardware dependent part of this firmware command. The routine “FW_EnableSetExec” sets three members of hardware dependent types, as seen in the lower left side of the graph. The right side shows several member use relations to types of a global database. The members are hardware independent, so that at this point, the separation cannot be done completely, i.e. the routine accesses hardware dependent as well as independent members.

The other routines of the relevant call graph are considered as hardware independent and are not assigned to the hardware layer as well as the types and their members. The accessed members are on the one hand part of a global database and on the other hand members of the timing measurement, which log the timing measurement.

8.5.3 Collected Data and Developer’s Comment

The analysis took 12 hours overall. Of these 12 hours, 3 were needed for the construction of the relevant call graph. It required 3 hours to analyze the different members accessed by the routines of the relevant call graph. The segmentation into hardware dependent and independent layer took one hour. The rest of the time was needed for loading and saving the graphs, performing analysis on the different graphs, and for copying and pasting nodes and edges during the creation of new subsets.

The partially assignment of the firmware command “aprm” to the hardware dependent layer is appropriate since it is strongly related to the sequencer control, which is the software implementation of a microchip on the IC tester.

8.5.4 Lessons Learned

Again the utility module sequencer state stack was found which let the analyst presume that this is a module widely used throughout the firmware, since it was found in the call graph of two firmware commands not logically related to each other.

The segmentation into different layers is sometimes ambiguous. The analyst should try to refine the separation criterion if possible. This way he extends the foundation for the decision of the assignment.

8.6 The Firmware Command “etim”

8.6.1 Description

The firmware command “etim” (Edge TIMing) command defines the edge timing for a specified waveform set, timing set and pins. The edge timing specifies the actions occurring at a certain point of time. An action defines a part of the signal sent to the device under test.

Large functionality and high complexity are the attributes of the firmware command “etim”.

8.6.2 Results

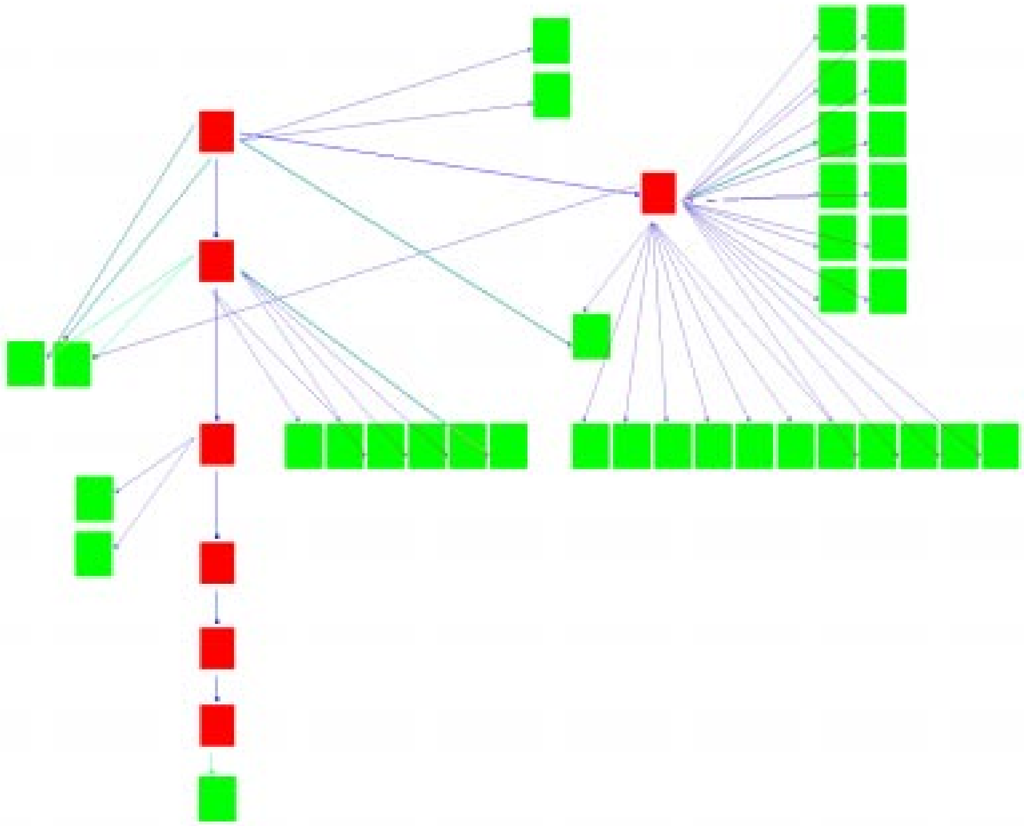

Figure 8-5. **“etim” - Hardware Dependent Routines and Members**

The call graph of the firmware command “etim” consists of 1300 nodes and 3613 edges, i.e. 1300 different routines may be executed when running this command. During the construction of the relevant call graph, several utility modules were secluded. Again, the sequencer state stack is part of this command. Furthermore, the identification of the modules “FW_OutOfDate”, “expr” and several “activate_set” was possible. The “FW_OutOfDate” mark several values for an update. The “expr” module resolves an expression, consisting of arithmetic operations. In addition to these modules, there are several “activate_set” modules which collect the current activated sets (e.g. the primary sets activated by the firmware command “aprm”).

The reduction of these utility modules lessens the call graph of this firmware command. The subset is then used for the construction of the relevant call graph comprising of 14 nodes and 19 edges

During the analysis, 7 routines were assigned to the hardware dependent layer as well as 37 members of types (see the cutout figure 8-5, the labels of the routines and the members have been left out for brevity).

The member of the type “FW_Channel_Addr_Space_t” (the member at the bottom of figure 8-5) is a type accessed by several other routines since it points directly to the registers of the hardware. Hence, the member is located near the bottom of the member access lattice.

8.6.3 Collected Data and Developer’s Comment

The construction of the relevant call graph required 6 hours. The segmentation into the hardware dependent and independent layer took 2 hours. The analysis of the concept lattice took 4 hours. The rest of 4 hours was needed for loading and saving the graphs, as wells as for the manipulation or the creation of new graphs containing only subsets of the other graphs.

The developers stated that most of the members assigned to the hardware dependent layer belong there, but some elements could be considered as hardware independent as well. A precise decision in these cases can not be made.

8.6.4 Lessons Learned

One point becoming obvious while analyzing the firmware command “etim” was that the software architecture expert is very important for segmentation of the elements into the different layers. Only he has the knowledge needed to perform this task.

When investigating a feature with large functionality, significantly more modules can be identified which are used to realize a functional requirement. When concentrating the analysis first on the firmware commands with large functionality, then several utility modules can be identified. These modules then can probably be recognized when proceeding with the investigation of firmware commands with less functionality.

8.7 The Firmware Command “xclk”

8.7.1 Description

The firmware command “xclk” (eXternal CLocK) sets global clock parameters. Normally, a tester is controlled by an internal clock. The command “xclk” disables the internal clock and replaces it by an external one. This is done for the synchronization of externally connected equipment, for example an oscilloscope.

Like “etim”, “xclk” is a firmware command with high complexity and large functionality.

8.7.2 Results

Figure 8-6. **“xclk” - The Relevant call Graph**

Most of the underlying modules of the firmware command “etim” were found for “xclk” as well, namely, “FW_OutOfDate”, “expr”, and several “activate_set”. This lead to a faster reduction of the call graph. Starting with 1299 nodes and 3601 relations among the nodes, the relevant call graph finally consisted only of 13 nodes and 13 relations.

Another result is that the propagation to registers of the hardware starting with the routine "write_vm_words" is the same as in the firmware command "etim". So this can be assumed as a module for the register access.

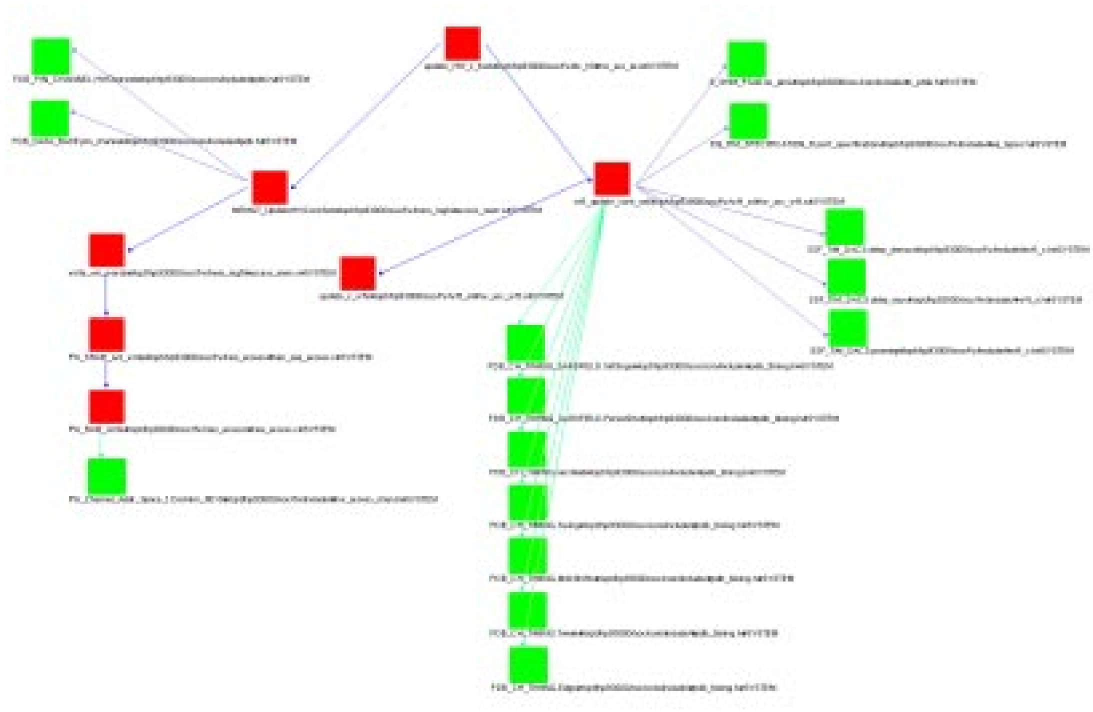

Figure 8-7. **"xclk" - Hardware Dependent Members and Routines**

There were 7 routines and 15 members of types considered as hardware dependent (see figure 8-7). The other routines and members are assigned to the hardware independent layer.

8.7.3 Collected Data and Developer's Comment

The construction of the relevant call graph required only 2 hours. The segmentation into the hardware dependent and independent layer took 2 hours as well. Then analysis of the concept lattice took 4 hours. The rest of the time, 4 hours, was needed for the visualization or the manipulation of the graphs.

8.7.4 Lessons Learned

Modules providing underlying utilities can be assigned to the hardware dependent layer as well as to the hardware independent layer. Some of them consist of elements of both layers, e.g. the sequencer state stack.

The more firmware commands are investigated, the better the results are. The reason for this is that some modules are not identified until they can be recognized in the context of several firmware commands. And the more firmware commands are investigated, the more elements can be assigned to the different layers. This way it is possible to synthesize the different layers throughout the firmware.

8.8 Summary

The evaluation showed a few shortcomings of the Bauhaus tool set operating with a large scale software system as the firmware is. Due to the standby between two consecutive operations, the analyst often had to wait for some time. Another problem was that the concept lattice was only available in a grid layout, due to the fact, that all other layout algorithm were running more than a week nonstop, before the attempt was aborted. This lead to the problem that only views with significantly less than 5000 nodes could be layouted in another way than the default grid layout. When operating with single features, the dimensions can be downsized, so that only a cut-out of the whole has to be handled. However, the Bauhaus tool set has by far the best performance when compared to the other tools with similar functionality mentioned in chapter 4.3.

Regardless this negative points, the process model “Bottom-Up Analysis” has shown its abilities in recovering layers and modules in the architecture of a software system. Several features (i.e. firmware commands) have been investigated and the routines and types have been assigned to one of the two layers. Furthermore, a few of the underlying utility modules could be identified. For instance, the sequencer state stack was found, with push and pop operations, used by all firmware commands except “mema”. Or the module “FW_OutOfDate” was found, which sets flags for values that have to be updated. The firmware commands “etim” and “xclk” both used a module “expr” to calculate equations, and they are both based on several utility modules, which provide the firmware commands with the current active sets, for example the active level set, or the active waveform. The analyst could assess these utility modules and routines beyond the assignment to different layers. In further investigation the utility modules can then again be investigated in detail.

Table 8-1 contains an overview about values for the different firmware commands. The number of members of types refers only to the relevant call graph as well as the number of concepts. The number of hardware elements shows how many elements (i.e. routines and members) have been assigned to the hardware dependent layer. The total time is measured in hours.

Firmware Command	# Calls	# Call Relations	# Relevant Calls	# Member of Types	# Concepts	# Hardware Elements	Total Time
mema	11	19	3	0	0	2	0.5
tstl	1181	3252	2	15	1068	0	8
aprm	1141	3152	11	40	126	4	12
etim	1300	3613	14	76	565	44	16
xclk	1299	3601	13	29	54	22	12

Table 8-1. **Summary of Gathered Data**

Table 8-2 shows the times of the specific firmware commands in comparison. The times for the reconstruction of the relevant call graph of the firmware command “xclk” are relatively low since knowledge gained during the analysis of “etim” could be used. This is a synergy effect, which will probably reduce the times for other firmware commands when extending the case study to commands not examined yet.

Firmware Command	Construction of the Relevant Call Graph	Segmentation	Analysis of the Concept Lattice	Visualization and Manipulation	Total Time
mema	0,2	0,1	0	0,1	0.5
tstl	2	0	2	4	8
aprm	3	1	3	5	12
etim	6	2	4	4	16
xclk	2	2	4	4	12

Table 8-2. **Summary of Times in Hours**

The examination of a single firmware command required a bit more time than expected, but in my opinion, the times can be reduced when operating only with a concept lattice based on the relevant call graph, since most of the time is spent with the analysis of the concept lattice and the visualization of the graphs.

Chapter 9 Conclusions

ABSTRACT: This final chapter summarizes the conclusions and contributions of this thesis as well as some unresolved issues and starting points for further work. As is customary, a short critique of the tasks and the work progress will be given as well.

9.1 Scope of this Work

This work had the task of developing process models for reconstructing software architecture views and of evaluating them at an industrial partner, making a particular legacy software available for the analysis activities. In the literature no process models have been found, that fulfill the following criteria:

- automated by tools
- applicable to real world problems
- realizable by one person or small teams
- explicit, precise work instructions
- easy to apply to large scale software systems

Consequently, two new process models have been developed.

The process models profit from reverse engineering techniques like the reflexion model, program slicing or the concept analysis as well as from several tools (namely Bauhaus, CodeSurfer, jRMTool, RMGuiMapper, Xrefactory) that are incorporated in the specific activities.

In comparison to the seven process model for architecture recovery presented in chapter 1.2, the two process models introduced in this thesis differ in the achievements as follows:

- [Kazman 1998] presents with the horseshoe model a generic framework for reengineering and architecture recovery as a part of it. The horseshoe model tries to integrate many analytical and transformational processes between the code-level and architectural views. Therefore precise, non-abstract working instruction are missing as well as a concrete description for what purpose specific techniques are used.
- The task artifact cycle introduced by [Kutscha 1997] differs from the two process models because it is an abstract process description, i.e. the general idea behind the process is mentioned, but no techniques, tools or explicit working instructions.
- The process model of [Kölsch 1997] for performing reengineering tasks concentrates objectification of the source code elements. No high-level elements, like modules, subsystems or components, are considered. Further on, no tool support has been integrated in the process model yet.
- [Ponsignon 1999] proposes to build graphical views of low-level objects and after a transformation of these objects a high-level view. But it is left out, how to build these two views. In contrast to the process models presented in this thesis, it is not clear which steps for what purposes the analyst has to perform for obtaining the goal of reconstruction of software architecture views.
- The object-oriented rearchitecting by [Klösch 1996] proposes to recover the design with various classic reverse engineering techniques, but it is not said, which technique is used for which purpose. The comparison shows an advantage for the two process models regarding the deployed reverse engineering techniques as well as the used tools, because their use is described precisely.
- The main difference to the approach of [Guo 1999], the software reconstruction method, is that [Guo 1999] is based on the recognition of patterns. The analyst has first to define concrete patterns, then it is attempted to detect the patterns in the source code, and finally, the software architecture is derived around the detected instances of the patterns. The approach has the shortcoming that it is restricted to software systems that really use such patterns and where these patterns are known. In contrast to this, the two process models introduced in this thesis work for all software systems. Another vagueness of the software architecture reconstruction method is the usefulness for large scale software systems, since it has only been applied in case studies where the analyzed systems consisted of noticeable less than 100 KLOC.
- The framework of [Lung 1997] is described in a relatively abstract manner. Precise working instructions for the four different phases are missing. For instance, it is not said, which information should be gathered in the first phase or how the molding activity should be performed. When following the processes “Design Hypothesis” or “Bottom-Up Analysis”, an analyst knows which activities to perform because both process models are documented in detail.

The goals for the process models to reconstruct architectural views positioned in the introduction of this thesis are namely the following:

- high automatization by tools where possible

- reasonable integration of available techniques
- applicable to real world problems, i.e. easy to apply to large scale software systems
- realizable by one person or small teams
- explicit, precise work instructions
- fast reconstruction of useful views on the software architecture

When considering these objectives, the results show that the process models “Design Hypothesis” and “Bottom-Up Analysis” succeed in the achievement of all of these goals contrary to the seven other process models (which are partly based on other premises, or have slightly different objectives).

In developing these two process models, contribution was done to the fields of software architectures in general, architecture documentation and recovery, and in the domain of reverse engineering and reengineering.

9.2 Unresolved Issues

The development of another process model integrating dynamic analysis, i.e. that gathers architectural information while the software system is running, has been left out due to time limitations. There are several tools and methods for dynamic analyses, and the question to what degree these are applicable to a large scale software system remains unanswered yet.

A further step is to find possibilities to collapse and to cluster information when wanted and accompanied by a better navigation through the different views on the architecture of a software system. Imaginable would be a step-by-step generation of the different views, i.e. only the elements are viewable the analyst wants to see in the context of his actual work, contrary to other elements which stay hidden.

When applying the process models to real world problems, some of the activities are quite time consuming, e.g. the visualization of a computed reflexion model, the generation of the RFG, the layout of the whole concept lattice failed (after the layout algorithm was running longer than 2 weeks, it was aborted), or some of the functionality of the Bauhaus tool set, e.g. copy and paste, load or save a RFG. Hence, the huge amount of data lead to delays. With a better performance the work could be facilitated since the standby times would be reduced. Hence, the optimization of the used tools is of great importance.

Other unresolved issues are the possible further refinements already mentioned in the chapters 5.7 and 7.7. There are a lot of potentials for improvements in these proposals, that in my opinion, could lead to significant performance and quality improvements of both process models.

9.3 Future Work

In order to continue the work done so far, further evaluation with controlled experiments and case studies of other projects at Agilent Technologies and in other development environments have to be performed to serve the following purposes:

- assurance that the process models meet their goals with other software systems than the firmware
- collection of data for a better effort estimation
- verification of further improvements
- refinement of both process models on basis of the newly gained practical experiences if necessary

Another point is the extension of the module relations (see chapter 3.1.15.3) of the process model “Design Hypothesis”. This could be easily done since the analyst only has to extract those relations from the source code. But in my opinion, this can only be recommended when the high-level design model is very stable. Nice to have would be an optional computing of the reflexion models where the analyst can choose which relations are important, and hence are visualized.

A point worth some effort of analysis is to check whether the two process models can be combined. For example, it is imaginable that the analyst first establishes a high-level design model (as in the process model “Design Hypothesis”), and then adopts the process model “Bottom-Up Analysis” for the investigation of the structure of a specific module. This approach may combine the advantages of both, the top-down process model “Design Hypothesis” and the “Bottom-Up Analysis”.

9.4 Learnings

Through my work, I could deepen my knowledge in the fields of software maintenance, software architectures, architecture recovery and reverse engineering. By means of the firmware, I learned a lot about possible reasons and factors that let software systems degenerate during program evolution.

Further on, I dealt especially with views on software architecture, and how to document the structures of a system. It became clear that it depends on the focus of the viewer to what degree he can use a specific view. Separation of concerns is a good instrument to improve the quality of software architecture views.

Beyond the experiences interrelated with the topic of this thesis, I gained deep insight in the corporate culture and the company policies of Agilent Technologies. Further on, the work provided me with a view into a global player of semiconductor test. I learned about the advantages and disadvantages of such a large, distributed organization. For example, it requires quite a long time for the company-wide transition of decisions and ideas.

9.5 Practical Critique

Now, as customary, some practical issues encountered during the progress of the thesis are discussed shortly.

I made one major mistake in the evaluation phase of the tools. I looked at the tools with examples, not with the whole firmware because I first wanted to know which services the tools provide and what is the usefulness of the specific features. The better approach would have been to start the evaluation at once with the firmware source code, because some characteristics (or bugs) are faced only when working with large scale software. The tools related risks of this work would have been recognized earlier and more time to mitigate the risks or to work out alternatives would have been available.

Another problem that emerged and partly resulted of the mistake described above was that the first releases of the Bauhaus tool set included some bugs, e.g. the copy and paste functionality did not work properly. These defects had to be fixed, and due to the fact that they were not revealed in the tools evaluation phase the case studies had to be shifted. Especially the evaluation of the process model “Bottom-Up Analysis” suffered from this situation so that the results were achieved not until barely before the deadline of the thesis. The reason for this is the fact that the Bauhaus tool set was in an experimental state up to late in the course of this thesis, since a new graphical user interface was deployed. On the other hand, without the new graphical user interface, Bauhaus would not be usable at all, since already some major performance improvements have been achieved.

The instantiation of the process model “Design Hypothesis” with the purpose of creating a layered architecture had to be postponed due to timing reasons. It was planned that a team of software architecture experts organize the source code elements in modules assigned to several layers. Relations in this high-level design model were supposed to be in a top-down manner, and modules belonging to the same layer could also communicate among each other.

Remarkable is the pleasant working atmosphere at Agilent Technologies. I enjoyed to work together with software engineers who shared their wide, in many years acquired experience with me in various discussions and several informal talks.

This thesis contributes much from the support of both, the developers at Agilent Technologies and at the Institute of Informatik, University of Stuttgart.

9.6 Contributions

The contributions of my work may be summarized as follows: two process models, one with a top-down approach, the other bottom-up, have been developed for the reconstruction of architectural views. They integrate several reverse engineering techniques. A person or a small team can apply the process model to real world problems following precise working instructions.

After reading the process model description of chapter 5 and 7, developers of Agilent Technologies not concerned with my work before stated that they think they could leverage both process models and they expect useful results when applying them to their problems in architecture recovery. Furthermore they quoted that the working instructions for the specific activities are clear and precise. They feel very confident in deploying the process models.

Agilent Technologies already plans restructuring activities with the help of this process models, and the company can profit from the results already gained.

In summary, the results of both case studies allow to make sound statements about the software architecture of the firmware. Subsystems have been detected, modules have been identified, and the hardware dependent layer has been partly exposed. The developers at Agilent Technologies stated that the case studies provided them with useful, insightful, interesting results.

As the two process models have different objectives, the results and times of the two case studies can not be compared. Therefore, only advantages and disadvantages of the process models are presented in the following sections.

The strength of the process model “Design Hypothesis” is that it produces relatively fast views on the software architecture. Another positive point is that the analyst can try out several different assumptions about the architecture of the software system. But when investigation is done at a very fine-grained level, it is expected that a many mappings have to be done. So the scope of this process model should be set in a way that the mapping activity is manageable.

The “Bottom-Up Analysis” provides the analyst with detailed information about specific features and it has served well for the identification of underlying utility modules and the separation of different layers. The weakness of the “Bottom-Up Analysis” is that no subsystem except the layers are identified.

In my opinion, the results of the evaluation show that both process models are generally applicable and the analyst will benefit of the created architectural views on the software.

9.7 Final Remarks

Large scale software should be maintained carefully for the purpose of not losing important information about the architecture. During the software life cycle, programs evolve due to maintenance, porting, enhancements or new requirements. In cases of outdated documentation or no documentation available at all, it is reasonable to first recover views on the architecture and then to apply the modification.

It is difficult to reconstruct software architectures, since in the broad field of literature about software architectures in general (at least in the publications I found), only relatively abstract approaches have been proposed for this problem. Therefore the two process models presented in this thesis have been developed to address this problem, and as the results of the two case studies and the intention of the developers to employ show, the two process models are successful in doing so.

However, there are still some open points concerning the two process models and architecture recovery in general. Hence, a lot of tasks remain for additional work.

Appendix A Bibliography

[Agilent Technologies] Agilent Technologies
http://www.agilent.com
last accessed: July 2002

[Bachmann 2001] Felix Bachmann, Len Bass, Paul Clement, David Garlan, James Ivers, Reed Little, Robert Nord, Judy Stafford
Documenting Software Architectures: Organization of Documentation Packages
Carnegie Mellon, Software Engineering Institute, Pittsburgh PA, CMU/SEI-2001-TN-010, 2001

[Bauhaus] Projekt Bauhaus
http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus/
last accessed: July 2002

[Baxter 1998] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, Lorraine Bier
Clone Detection Using Abstract Syntax Trees
Proceedings of the International Conference on Software Maintenance, IEEE, 1998

[Beszedes 2001] Arpad Beszedes, Tamas Gergely, Mihaly Szabo, Janos Cisirik, Tibor Gyimothy
Dynamic Slicing Method of Large C Programs
Proceedings of the Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, 2001, pp. 105-113

[Bleher 2001] Gerd Bleher
Fundamentals of Semiconductors
Agilent Technologies, 2001

[Boehm 1973] Barry W. Boehm
Software and its Impact: A Quantitative Assessment
Datamation, Vol. 19, No. 5, 1973

[Chifkovsky 1990] Elliot J. Chifkovsky, James H.Cross
Reverse Engineering and Design Recovery: A Taxonomy
January 1990

[CIAO] CIAO: A Graphical Navigator for Software and Document Repositories
http://www.research.att.com/~ciao/
last accessed: July 2002

[Clements 1996] Paul Clements, Robert Krut, Ed Morris, Kurt Wallnau
The Gadfly: An Approach to Architetural-Level System Comprehension
Fourth IEEE Workshop on Program Comprehension, Berlin, March 1996

[CodeSurfer] CodeSurfer
http://www.grammatech.com/products/codesurfer/
last accessed: July 2002

[Command Reference] Agilent Technologies
Command Reference - Agilent 93000 SOC P-Series and C-Series
Revision 3.0.1, January 2001

[Eisenbarth 1999] Thomas Eisenbarth, Rainer Koschke, Erhard Plödereder, Jean-Francois Girard, Martin Würthner
Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen
Workshop Software-Reengineering, Bad Honnef, Universität Koblenz-Landau, Fachberichte Informatik, Nr. 7-99, pp. 17-26, 1999

[Eisenbarth 2002] Thomas Eisenbarth, Rainer Koschke, Daniel Simon
Feature-Centric Program Analysis
March 2002

[Fenton 1997] Norman E. Fenton, Shari L. Pfleeger
Software Metrics: A Rigorous and Practical Approach
Int. Thomson Computer Press, 1997

[Fjelstad 1983] R. K. Fjelstad, W. T. Hamlen
Application Program Maintenance Study: Report To Our Respondents
Tutorial on Software Maintenance, IEEE Computer Society, April 1983

[Fowler 1999] Martin Fowler
Refactoring - Improving the Design of Existing Code
Addison-Wesley Longman, Reading MA, 1999

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Patterns: Elements of Reusable Object Oriented Software
Addison Wesley Longman Inc., Reading M.A., 1995

[Ganter 1999] Berhard Ganter, Rudolf Wille
Formal Concept Analysis - Mathematical Foundations
Springer Berlin-Heidelberg, 1999

[Graphviz] Graphviz - Open Source Graph Drawing Software
http://www.research.att.com/sw/tools/graphviz/
last accessed: July 2002

[Guo 1999] G. Y. Guo, J. M. Atlee, R. Kazman
A Software Architecture Reconstruction Method
Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, USA, February 1999.

[Hofmeister 2000] Christine Hofmeister, Robert Nord, Dilip Soni
Applied Software Architecture
Addison-Wesley Longman, Reading MA, 2000

[Horwitz 1990] Susanne Horwitz, Thomas Reps, David Binkley
Interprocedural Slicing Using Dependence Graphs
ACM Transactions on Programming Languages and Systems, vol. 12, no. 1, 1990, pp. 26-60

[IDEFØ] Integrated Definition Laguage
http://www.idef.com/idef0.html
last accessed: Julz 2002

[IEEE 488] General Purpose Interface Bus
http://www.ieee.org
last accessed: July 2002

[IEEE Std 610.12-1990] IEEE Std 610.12-1990
IEEE Std 610.12-1990
http://www.ieee.org
last accessed: July 2002

[IEEE Std 1471-2000] IEEE Std 1471-2000
IEEE Recommended Practice for Architectural Description of Software-Intensive Systems
IEEE-SA Standards Board, September 2000

[Imagix 4D] Imagix 4D -- C/C++ Reverse-Engineering
http://www.imagix.com/products/imagix4d.html
last accessed: July 2002

[ISVis] ISVis - Interaction Scenario Visualizer
http://www.cc.gatech.edu/morale/tools/isvis/isvis.html
last accessed: July 2002

[jRMTool] jRMTool
http://www.cs.ubc.ca/~murphy/jRMTool/doc/
last accessed: July 2002

[Kazman 1998] Rick Kazman, Steven G. Woods, S. Jeromy Carriere
Requirements for Integrating Software Archi-tecture and Reengineering Models: CORUM II
IEEE Computer Society Press, Working Conference on Reverse Engineering, Hawaii, October 1998, pp. 154-163

[Klösch 1996] Rene R. Klösch
Reverse Engineering: Why and How to Reverse Engineer Software
Proceeding of the California Software Symposium, April 1996

[Kölsch 1997] Ulrike Kölsch, Mehcthild Wallrath
A Process Model for Controlling and Performing Reengineering Tasks
Proceedings of the 1st Euromicro Working Conference on Software Maintenance and Reengineering (CSMR ’97), 1997

[Koschke 2000] Rainer Koschke
Atomic Architectural Component Recovery for Program Understanding and Evolution
Ph.D. Thesis, Institute for Computer Science, University of Stuttgart, 2000

[Koschke 2001] Rainer Koschke
Prozessmodelle für die Rekonstruktion von Sofware-Architektursichten
Tender, Institute for Computer Science, University of Stuttgart, 2001

[Kruchten 1995] Philippe Kruchten
Architetural Blueprints - The “4+1“ View Model of Software Architetcure
IEEE Software 12, November 1995

[Kutscha 1997] S. Kutscha, K. Henning, H. Kesselmeier
The Task Artifact Cycle: A Conceptual Approach to Maintenance and Reengineering
Proceedings of the 1. Euromicro Conference on Software Maintenance and Reengineering, 1997

[Lackwit] Lackwit: Practical Program Understanding With Type Inference
http://www-2.cs.cmu.edu/~roc/Lackwit.html
last accessed: July 2002

[Lehman 1985] M.M. Lehman, L. Belady
Program Evolution
Processes of Software Change, Academic Press, London, 1985

[Lung 1997] C. H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman
An Approach to Software Architecture Analysis for Evolution and Reusability
Proceedings of CASCON ’97, November 1997

[Mendoca 1996] Nabor C. Mendoca, Jeff Kramer
Requirements for an Effective Architecture Recovery Framework
SIGSOFT 96 Workshop, San Francisco, 1996

[Müller 1994] H. A. Müller, K. Wong, and S. R. Tilley
Understanding software systems using reverse engineering technology
The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS) 1994

[Murphy 1995] Gail C. Murphy, David Notkin, Kevin Sullivan
Software Reflexion Models: Bridging the Gap between Source and High-Level Models
Proceedings of SIGSOFT '95, ACM Press, New York, 1995, pp. 18-28

[Murphy 1997] Gail C. Murphy, David Notkin
Reengineering with Reflexion Models: A Case Study
IEEE Computer 30, 8, 1997, pp. 29-36

[Parnas 1972] David L. Parnas
On the Criteria To Be Used in Decomposing Systems into Modules
Communication of the ACM, Vol. 15, No. 12, December 1972, pp. 1053-1058

[Ponsignon 1999] J.-F. Ponsignon, M. Maranzana, R. Aubry, Y. Martinez
Reengineering of Multitasking Applications for Easier Maintenance
Proceedings of the Third European Conference on Software Maintenance and Reengineering, March 1999

[Portable BookShelf] The Portable Bookshelf
http://swag.uwaterloo.ca/pbs/
last accessed: July 2002

[Pressman 1997] Roger S. Pressman
Software Engineering - A Practioner's Approach
Fourth Edition, McGraw-Hill Companies Inc., Cambridge, 1997

[Rich 1990] Charles Rich, Linda M. Wills
Recognizing a Program's Design: A Graph-Parsing Approach
IEEE Software, Vol. 7, No. 1, January 1990, pp. 82-89

[Rigi Workbench] Rigi - a visual tool for understanding legacy systems
http://www.rigi.csc.uvic.ca/
last accessed: July 2002

[Rugaber 1994] Spencer Rugaber
White Paper on Reverse Engineering
Software Engineering Research Center, Georgia Intistute of Technology, Atlanta, March 1994
http://www.cc.gatech.edu/reverse/repository/white_paper.ps
last accessed: July 2002

[Shaw 1996] Mary Shaw, David Garlan
Software Architecture - Perspectives on an Emerging Discipline
Prentice-Hall, Upper Saddle River NJ, 1996

[Sommerville 1989] Ian Sommerville
Software Engineering
Third Edition, Addison-Wesley , 1989

[Snelting 1997] Gregor Snelting
Reengineering of Configurations Based on Mathematical Concept Analysis
ACM Transactions on Software Engineering and Methodology 5, 2, April 1997, pp. 146-189

[SNiFF+] SNiFF+
http://www.windriver.com/products/html/sniff.html
last accessed: July 2002

[TraceGraph] TraceGraph
http://www.cs.uwf.edu/~recon/index.html
last accessed: July 2002

[User Manual Part 1] Agilent Technologies
Agilent 93000 SOC Series User Training Part 1
Revision 3.0.3, October 2001

[Weiser 1984] Mark Weiser
Program Slicing
IEEE Transactions on Software Engineering, vol. SE-10, no. 4, pp. 352-357, July 1984

[Wilde 2000] Norman Wilde, Kazimiras Lukoit, Scott Stowell Tim Hennessy
TraceGraph: Immediate Visual Location of Software Features
International Conference on Software Maintenance (ICSM'00), October 2000

[Xrefactory] Xrefactory - A Refactoring Browser for Emacs
http://www.xref-tech.com/
last accessed: July 2002

[Yourdon 1979] E.Yourdon, L.L. Constantine
Structured Design
Prentice Hall, Englewood Cliffs NJ, 1979

Appendix B The Reflexion Model

ABSTRACT: In this chapter, the different file formats needed for the reflexion model are described. The description is based on [Murphy 1995].

B.1. The Language Description File

The language description file describes how to refer to the source code elements in the map and which elements are available. The structure description file lets you describe a hierarchical naming scheme to support these different ways of referring to an entity. For example, the C description file used to analyze the firmware is the following:

```
dir
  dir.file
    dir.file.function
    dir.file.var
```

The language description file dictates which source code elements have to be extracted for the input format.

B.2. Input Format

Each line in a input file describes a relationship between two source code elements and the type of the relation. Each line is of the form:

Source_Code_Element_1 Source_Code_Element_2 type

A source code element has to follow the language description file dictating numbers to each element.

```
dir -> 1
dir.file -> 2
dir.file.function -> 3
dir.file.var -> 4
```

A source code element is encoded by the number in “@” symbols followed by the name of the specific source code element. The hierarchy of the elements is concatenated together.

```
@1@aDir@2@aFile@3@aFunction@4@
@1@aDir@2@aFile@3@@4@aVariable@
```

A relation in the input file, i.e. two encoded source code elements is followed by the type of the relation in plain text. So an example (an extraction of the input file of the firmware) for a correct input file would be:

```
@1@/opt/hp93000/soc/fw/aacu/@2@aacu_ctrl_hw.c@3@aacu_write_ctrl
@4@ @1@/opt/hp93000/soc/fw/include/@2@global_dbase.h@3@
@4@FW_TESTHEAD_INFO.dut_if_type data

@1@/opt/hp93000/soc/fw/aacu/@2@aacu_ctrl_hw.c@3@aacu_write_ctrl
@4@ @1@/opt/hp93000/soc/fw/include/@2@global_dbase.h@3@
@4@FW_GlobalDbase_t.testhead_info data

@1@/opt/hp93000/soc/fw/aacu/@2@aacu_ctrl_hw.c@3@aacu_write_ctrl
@4@ @1@/opt/hp93000/soc/fw/aacu/@2@aacu_ctrl_hw.c
@3@modify_util_line_assignment@4@ call
```

Note that each relation extracted from the source code has to be exactly one line in the input file.

B.3. High-Level Design Model

A line in a high-level design model file either introduces a node or a relation according to the following syntax:

- node
- node_1 node_2 <type>

The first defines a node with no dependencies, the node name may be a string. The second defines two nodes with a relation from node_1 to node_2, the type of the relation is optional, but should be used when there are more than one type of possible relations. Nodes may be introduced simply by defining relations.

As an example, the high-level design model of the first iteration when the sequencer control was analyzed file is shown below. There seven modules, and nine call relations have been assumed:

```
AC_TF
APRM
Global_Mode
HW_Access
Primary_Activation
Sequencer_Conf
Start_Sequencer
AC_TF Global_Mode <call>
AC_TF Sequencer_Conf <call>
AC_TF Start_Sequencer <call>
Start_Sequencer Sequencer_Conf <call>
Start_Sequencer Global_Mode <call>
APRM Primary_Activation <call>
Global_Mode HW_Access <call>
Sequencer_Conf HW_Access <call>
Start_Sequencer HW_Access <call>
```

B.4. The Mapping

A mapping file consists of an ordered sequence of map entries. Each entry is placed within square brackets. A source code element is mapped by the first line in the map it matches. A mapping file entry names source code elements and associates them with entities in the high-level design model. Source code elements are named by using the keywords specified in the language description file.

Regular expressions may be used when naming the source code elements (but not when naming high-level design model entities). The regular expression syntax is that of Perl. By default, wildcards are places around the start and end of a source item regular expression. For instance, stating file=pager means file=.*Parser.*. This wildcarding can be overridden through the use of the ^ and $ operators. For instance, file=^Parser˙cpp$ states only the file named Parser.c (and not Parser.h). Multiple high-level model entities may be named on the right hand side of a mapping entry. For example, the following statement maps the entities in Parser.c to both Parser and AnotherEntity. :

```
[ file=Parser.c mapTo=Parser mapTo=AnotherEntity]
```

Lines beginning (in the first column) with # in the mapping file are considered comment lines.

An example for the mapping is the following extraction of the sequencer mapping of the first iteration:

```
[ dir=^/opt/hp93000/soc/fw/ac_tf/$ file=^ac_tf_.h$ mapTo=AC_TF ]
[ dir=^/opt/hp93000/soc/fw/ac_tf/$ file=^get_edge.c$
function=^CheckEdgeDirection$ mapTo=AC_TF ]
[ dir=^/opt/hp93000/soc/fw/ac_tf/$ file=^get_edge.c$
function=^CheckEdgeNumber$ mapTo=AC_TF ]
[ dir=^/opt/hp93000/soc/fw/ac_tf/$ file=^get_edge.c$
function=^CheckPinList$ mapTo=AC_TF ]
[ dir=^/opt/hp93000/soc/fw/ac_tf/$ file=^get_edge.c$
function=^CheckPinListEq$ mapTo=AC_TF ]
[ dir=^/opt/hp93000/soc/fw/ac_tf/$ file=^get_edge.c$
function=^CheckPinListPort$ foo=foo mapTo=AC_TF ]
```

The whole file “ac_tf_.h” is mapped to the module AC_TF, but just a couple of function of the file “get_edge.c” belongs to the same module, too.

Appendix C Formal Concept Analysis

ABSTRACT: This chapter introduces the formal concept analysis and its mathematical foundations.

C.1. Foundations

This section acquaints with the basic ideas of concept analysis. Concept analysis provides a way to identify groups of entities that have common attributes. Its mathematical foundation was laid by Birkhoff in 1940. [Snelting 1997] introduced concept analysis to software engineering. The following mathematical background is based on [Ganter 1999].

C.2. Mathematical Background

Concept analysis is based on a relation R between a set of objects $\mathcal{O}$ and a set of attributes $\mathcal{A}$, hence $R \subseteq \mathcal{O} \times \mathcal{A}$. An object in the sense of concept analysis can be anything. Within this section on concept analysis, the term object will be used in the sense of concept analysis.

The triple $C = (\mathcal{O}, \mathcal{A}, R)$ is called a formal context. For any set of objects $O \subseteq \mathcal{O}$, the set of common attributes is defined as

$$\sigma(O) = \{a \in \mathrm{A} | \forall(o \in O)(o, a) \in \mathrm{R}\}$$

Similarly, for any set of attributes $A \subseteq \mathcal{A}$, their set of common objects is

$$\tau(A) = \{o \in \mathrm{O} | \forall(a \in A)(o, a) \in \mathrm{R}\}$$

As an example, let us consider the fictitious binary relation described by Table C-1. An object Oi has attribute Aj when there is a ✕ in row i and column j in Table C-1. In this table, for example, the following two equations hold:

$$\sigma(\{O_1\}) = \{A_1, A_2\} \qquad \tau(\{A_7, A_8\}) = \{O_3, O_4\}$$

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
O_1	✕	✕						
O_2			✕	✕	✕			
O_3			✕	✕		✕	✕	✕
O_4			✕	✕	✕	✕	✕	✕

Table C-1. **Example relation.**

The two functions σ and τ form a Galois connection, i.e., a pair of two antimonotone functions:

$$O_1 \subseteq O_2 \Rightarrow \sigma(O_2) \subseteq \sigma(O_1) \quad \text{and} \quad A_1 \subseteq A_2 \Rightarrow \tau(A_2) \subseteq \tau(A_1)$$

and both σ ° τ and τ ° σ are closure operators: e.g., $\sigma \circ \tau(O)$ determines the biggest set of objects that have the same attributes as O.

A pair (O, A) is called a concept, if $A = \sigma(O) \wedge O = \tau(A)$, i.e., all objects share all attributes. For a concept c = (O, A), O is the extent of c, denoted by extent(c), and A is the intent of c, denoted by intent(c).

Informally, a concept corresponds to a maximal rectangle of filled table cells modulo row and column permutations. For example, Table C-2 contains the concepts for the relation in Table C-1.

C_1	$\{O_1, O_2, O_3, O_4\}, \varnothing$
C_2	$\{O_2, O_3, O_4\}, \{A_3, A_4\}$
C_3	$\{O_1\}, \{A_1, A_2\}$
C_4	$\{O_2, O_4\}, \{A_3, A_4, A_5\}$
C_5	$\{O_3, O_4\}, \{A_3, A_4, A_6, A_7, A_8\}$

Table C-2. **Concepts for Table C-1**

C_6	$\{O_4\}, \{A_3, A_4, A_5, A_6, A_7, A_8\}$
C_7	$\emptyset, \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$

Table C-2. **Concepts for Table C-1**

The set of all concepts of a given relation forms a partial order via

$$(O_1, A_1) \le (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2 \qquad \text{or, equivalently}$$

$$(O_1, A_1) \le (O_2, A_2) \Leftrightarrow A_1 \supseteq A_2 \qquad .$$

If c1 ≤ c2, then c1 is said to be a subconcept of c2 and c2 is a superconcept of c1. For example, ({O2, O4}, {A3, A4, A5}) ≤ ({O2, O3, O4}, {A3, A4}) in Table C-1.

The sets of concepts and the partial order form a complete lattice, called concept lattice:

$$\mathrm{L(C)} = \{(O, A) \in 2^O \times 2^A \mid A = \sigma(O) \wedge O = \tau(A)\}$$

In this lattice, the infimum (or join) of two concepts is computed by intersecting their extents:

$$(O_1, A_1) \wedge (O_2, A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2))$$

Note that $A_1 \cup A_2 \subseteq \sigma(O_1 \cap O_2)$, as $O_1 \cap O_2$ has at least common attributes $A_1 \cup A_2$. Thus, an infimum describes the set of attributes common to two sets of objects.

Similarly, the supremum (or meet) is computed by intersecting the intents:

$$(O_1, A_1) \vee (O_2, A_2) = (\tau(A_1 \cap A_2), A_1 \cap A_2)$$

Again, $O_1 \cup O_2 \subseteq \tau(A_1 \cap A_2)$. Thus, a supremum describes a set of common objects which fit to two sets of attributes.

Graphically, the concept lattice for the example relation in Table C-1 can be represented as a graph whose nodes are the concepts in Table C-2 and whose edges denote the < relationship as shown in figure C-1 (a).

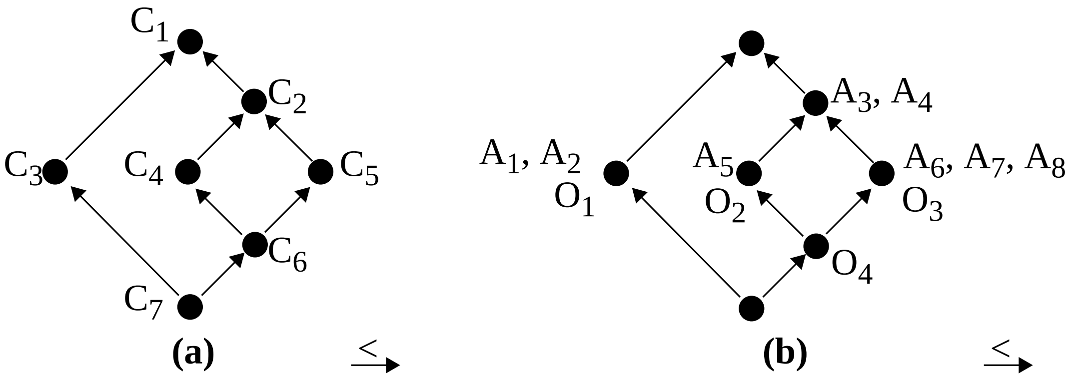

Figure C-1. **Example lattice.**

The graph for a concept lattice would be difficult to read when each node showed its complete concepts, i.e., if the nodes Ci were replaced by their contents according to Table C-2. Fortunately, there is a better strategy for labelling nodes. A graph node in figure C-1 (b) is labelled with attribute $a \in A$ if it is the largest concept having a in its intent; it is labelled with an object $o \in O$ if it is the smallest concept having o in its extent. The (unique) lattice element labelled with a is thus:

$$\mu(a) = \bigvee \{c \in \mathrm{L}(C) | a \in intent(c)\}$$

The element labelled with o is

$$\gamma(o) = \bigwedge \{c \in \mathrm{L}(C) | o \in extent(c)\}$$

The equivalent graph for figure C-1(a) using this labelling strategy is shown in figure C-1(b). A concept represented by a node N in this graph consists of all objects at and below N and of all attributes at and above N. For example, the concept labelled with O2 and A5 in (b) is ({O2, O4}, {A3, A4, A5}).

The concept lattice and the table, T, originally used to represent the relation are two equivalent ways to represent the relation, i.e., they can be reconstructed from each other, formally stated as:

$$(o, a) \in T \Leftrightarrow \gamma(o) \le \mu(a)$$

However, the concept lattice is a much more comprehensible representation allowing direct insight into the structure of the original relation. For example, we can immediately derive from figure C-1 that there are two disjoint sets of objects: O1 that has attributes A1 and A2 and objects O2, O3, and O4 share the other attributes. Furthermore, among O2, O3, and O4, O4 has all attributes of O2 and O3 but not vice versa. A3 and A4 is common to all objects O2, O3, and O4. This information can, of course, also be derived from the table, but for larger tables, this is more difficult.

Appendix D The Bauhaus Tool Set

ABSTRACT: Here the Bauhaus tool set is described. The used tools of this set are introduced here.

D.1. The Bauhaus Project

The Bauhaus project is a research cooperation of the department Programmiersprachen und Compiler at the Institut für Informatik, Universiät Stuttgart, and the Fraunhofer Institut für Experimentelles Software-Engineering (FhG IESE), Kaiserslautern. The Bauhaus project explores various techniques to increase the understanding of a system.

IML (InterMediate Language) is designed to represent different procedural programming languages (although only C programs are currently mapped onto IML) by offering general primitive concepts to model the semantics of all language constructs explicitly. On the other hand, by semantic annotations and specialization of modeling concepts for specific language constructs, IML allows to regenerate the original (preprocessed) source.

Using the IML code, a resource flow graph (RFG) is derived. Base nodes of the graph are subprograms, user defined types, and (global) variables, for example. The relations between the nodes are represented by edges. The RFG is used as a data base for the identification and visualization of architectural components. A Bauhaus analysis takes a view as input.

RFGs are visualized by a graph editor, being part of the Bauhaus tool set, too. The editor provides information about the resource flow graph and is the start for various kinds of analyses on the graph.

D.2. add_member_lattice

The tool add_member_lattice computes the concept lattice and adds it to the RFG. The concept lattice is depending on the formal definition of the concept analysis. The tool can be adjusted in order to perform concept analyses with other objects or attributes.

D.3. dumpreflexion

For the extraction of the input format for the computation of the reflexion model from the RFG, this tool is used. It extracts the nodes, the relations between them, and the specific type of the relations. When the analyst wants to introduces new types of relations, he has to adjust this tool. I developed the dumpreflexion tool for the purpose of transforming the RFG format into the input format for the reflexion model. Originally, the [CIAO] tool set should take over this task, but since it did not work properly, this solution was implemented.

Appendix E Task Description of the Diploma Thesis

ABSTRACT: Here the tender of the diploma thesis is presented. The tender includes the detailed task description.

E.1. The Tender

Diplomarbeit

Prozessmodelle für die Rekonstruktion von Software-Architektursichten

Hintergrund

In der Literatur existieren eine Reihe von Techniken zur Rekonstruktion der Software-Architektur von Altsystemen. Genaue Anleitungen, wie diese Techniken angewandt und kombiniert werden sollen (abhängig von Zielen und Randbedingungen der Rekonstruktion), existieren jedoch nur unzureichend. Solche Prozessmodelle sind jedoch notwendig, um die Techniken effektiv einsetzen zu können.

Aufgabenstellung

In dieser Diplomarbeit sollen Prozessmodelle zur Rekonstruktion von Software-Architektursichten ausgehend vom Quellcode und etwaig existierender (jedoch nicht unbedingt zuverlässiger) Dokumentation entwickelt und evaluiert werden.

Als konkrete Fallstudie soll die Architektur eines ca. 400 KLOC großen Teilsystems der Firma Agilent Technologies untersucht werden. Ein Katalog spezifischer Fragestellungen bezüglich architekturrelvanter Information seitens Agilent Technologies und dessen Generalisierung für Architekturrekonstruktion im Allgemeinen findet sich im Anhang.

Die erwarteten Ergebnisse sind im Einzelnen:

- Literaturstudie zur Frage, inwieweit bereits einsetzbare Lösungsmöglichkeiten für die spezifischen und verallgemeinerten Fragen existieren,
- Auswahl von verfügbaren Techniken/Werkzeuge sowie Erstellung von Prozessmodellen zur Rekonstruktion der Architektursichten (spezifisch und verallgemeinert) basierend auf den ausgewählten Techniken
- Evaluation der Prozessmodelle mit Hilfe verfügbarer Techniken an der Software von Agilent Technologies zur Beantwortung der spezifischen Fragen unter Einbeziehung eines Expertens seitens Agilent Technologies (eventuell nur an einem Teilsystem, abhängig von der verfügbaren Zeit) sowie Erfahrungsbericht über die Eignung der benutzten Techniken,
- Überarbeitung der ursprünglichen Prozessmodelle und Beschreibung weiterer notwendiger Informationen, die bislang durch existierende Techniken nicht hergeleitet werden, auf der Basis der gemachten Erfahrungen.

Voraussetzungen

Vorlesung *Software Reengineering*

Betreuer
Rainer Koschke (interner Betreuer) und
Gerd Bleher (externer Betreuer bei Agilent)

Erklärung zur Diplomarbeit:

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Stuttgart, den 03. Juli 2002

Jens Knodel