2010

Mathijs Schuts

IMPROVING SOFTWARE DEVELOPMENT

The introduction and implementation of ASD at Philips Healthcare

Radboud University Nijmegen

Master Thesis: Computer Science

Improving Software Development

The introduction and implementation of ASD at Philips Healthcare[1]

Radboud University Nijmegen, 2010

Version:	1.0
Thesis number:	632
Company:	Philips Healthcare, BU Interventional X-ray Embedded Systems Institute
Student:	M.T.W. (Mathijs) Schuts
Supervisor:	prof. dr. J.J.M. (Jozef) Hooman
Supervisor:	dr. A. (Arjan) van Rooij
External Supervisor:	dr. ir. R.J. (Robert) Huis in ’t Veld

[1] This work has been carried out as a part of the Darwin project at Philips Healthcare under the responsibility of the Embedded Systems Institute. This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

Abstract

To improve the software development process, the business unit Interventional X-ray of Philips Healthcare wants to reduce the test & integration phase by introducing Analytic Software Design (ASD). ASD is a tool that can be used for designing control-based software in a component-based way. In ASD, a system is specified in a Sequence-Based Specification (SBS), which is a large table. The table describes for all states of the system how it should respond to all possible stimuli. A complete design specification can be verified formally. The Sequence-Based Specification can be used to generate source code.

We have investigated the transition from the existing development approach to a new situation where ASD will be used. In our analysis, we observed that currently ASD is positioned as a tool and therefore requires only changes in the skills of the persons that need to apply it. We also noticed that making a design following the current practices did result in designs that could not be checked by the ASD tool. Moreover, the engineers followed an ASD course that hardly explained how to create a design that suits ASD. In general, a method to apply the ASD tool is missing.

The literature indicates that ASD and Cleanroom are related technologies. Cleanroom Software Engineering describes a complete software development process and method. We propose to introduce a Cleanroom-like method and refer to the combination with ASD as Cleanroom/ASD. Cleanroom's process model is adapted to fit into the generic process model of the business unit. The Cleanroom method describes the steps to obtain a global and a detailed design. In the final step of the method, ASD can be applied for control-based software. For non-control-based software a correctness proof could be made by hand. The benefit of using a Cleanroom-like method is that there is one integral approach to design high quality software.

Applying Cleanroom/ASD will only become a success if the organisation is prepared to make concessions on the design. This is the most important hurdle that needs to be taken. Applying ASD is not just a change of skills for the persons that need to apply it, but rather a cultural change. Literature suggests that the required change in mindset may take at least a year to accomplish.

Additionally, we have looked at some techniques that could increase the re-use of system control behaviour in the ASD context. We implemented a proof-of-concept of the following two techniques: template models and merging template models. We propose that first an architecture is decomposed with Cleanroom. Secondly, common and partial behaviour in interfaces should be identified and put into template models. Lastly, the ASD models are implemented bottom-up to maximize re-using common behaviour. Our implemented techniques can then be used to compose interfaces from the template models.

Preface

This thesis is the result of conducting my master’s thesis project at Philips Healthcare. In this preface, I would like to thank the persons without whom it would not be possible to successfully conduct the research and write the thesis.

I would like to start with a special acknowledge of Jozef Hooman, who arranged the assignment, for his involvement in the research, support, and his patience in guiding the difficult process of writing the thesis. I would like to acknowledge my other supervisors Arjan van Rooij and Robert Huis in ’t Veld for their guidance, tips and help.

My research took place at a software development team at Philips Healthcare. Thanks to the team members Ben Nijhuis, Johan Gielen, Hans van Wezep, André Postma and Ammar Osaiweran for adopting me into the team, and for a pleasant time at Philips Healthcare. Additionally, I would also like to thank the team members and managers I have interviewed for their time.

Contents

1. Introduction

1.1 Problem Statement

Organisations are continuously searching for improvements of their software development process. One can think of the improvements of software quality, predictability of the process, and cost savings. The high-tech industry is not an exception, since it increasingly uses software to build complex machines and applications. Typically, products are extended with additional functionalities over many years, through constant incremental innovations. Because of the extended functionalities, the software architecture needs frequent rework in order to maintain important software quality attributes, such as maintainability, efficiency, flexibility, interoperability, security, performance, testability, and extendibility.

A software architecture usually consists of a number of software blocks which are independently developed. When finished, these software blocks are integrated into increasingly larger compositions and finally into a complete working product. The test & integration phase is often hard to control and predict, because usually many problems are found relatively late in the software development process. Sometimes major problems have to be solved at this phase, which may even require architectural changes.

To improve this situation, a tool called Analytical Software Design (ASD) has been devised by a Dutch company called Verum. ASD is a tool to mathematically specify interfaces and designs, to model check these, and to generate source code, and to generate compliance tests from the models.

At Philips Healthcare the ASD tool will be used to redesign and to re-implement parts of a new software architecture to be able to cope with new market and technical developments for the coming decade. ASD should improve the development efficiency, because of fewer problems during the test & integration phase. ASD is intended for the design of control-based behaviour. Hence, only part of the architecture will be implemented with ASD. Moreover, some aspects, such as (meta-) data and real-time requirements, cannot be modelled and verified with the ASD tool.

This thesis will show that introducing the ASD tool into a development organisation is not just a technical aspect but also involves changes in organisation, method, and mindset of the engineers that need to use the tool. The managers and project leaders of the team can use the thesis to read about the transition from a situation without ASD to one with ASD. Henceforth, this is called the “transition situation” in which we examined the current state of the organisation, the nature of the technology, the goals of applying the new technology, and the steps necessary to reach the desired goals [8]. Additionally, team members can use the thesis to enhance their ASD knowledge. In addition, we present a few techniques that are intended improve the ease of use of the ASD tool during development.

The above leads to the following two research question:

- What is the transition situation?
- How can re-use and migration enhance the technology?

1.2 Approach

In this section, we describe the approach used to answer the research questions. For answering the first research question, the transition situation as described by [8] is used. This paper describes questions for examining the transition situation. We slightly adapted the questions for our purpose which resulted in the following sub-questions.

1. What is the transition situation?
1.1 What is the state of the organisation that will incorporate the new technology?
1.2 What is the nature of the technology?
1.3 What is the ultimate goal for acquiring and using the technology?
1.4 What are the steps to reach the desired goals given the state of the organisation?

Answering question 1.3 resulted in questions about the re-usability of models and the migration to the new architecture. This led to the following sub-questions:
2. How can re-use and migration enhance the technology?
2.1 How can re-use enhance the technology?
2.2 How to migrate from the current situation to the new technology?

These questions have been investigated at Philips Healthcare and we describe for each sub-question how it is answered:

- Question 1.1: analysing internal documents and interviews. Since internal documents not always describe the real practices within an organisation, additional information is gathered by interviewing the persons involved.
- Question 1.2: studying literature about the technology being transitioned and hands-on experience with it by working on question 2.
- Question 1.3: interviewing the persons involved and observing the team's first steps in applying the new technology.
- Question 1.4: analysis of the answers to sub-questions 1.1-1.3 and relating the outcome of the analysis to the literature.
- Question 2.1: implementing a proof-of-concept and applying it to a real case to test its effectiveness.
- Question 2.2: applying another proof-of-concept onto a real case.

1.3 Scope

The research has been taken place during a preparation phase of a pilot project that will implement a new reference architecture. The observations described here are only valid for the time the research took place.

The thesis describes an analysis about the infusion of the technology into the organisation and can be used to create an operational transition plan. It does not describe operational details itself such as changes to the archive to support ASD, new build procedures, validation of ASD generated source code, etc.

The research is a snapshot in time. The developments during the research were very dynamic. Additionally, ASD itself is a moving target because it is frequently improved. Our implemented re-use and migration techniques work with ASD:Suite version 3. It has not been tested with higher versions. On the other hand, it should be possible to adapt the proof-of-concept relatively easy.

1.4 Overview

The thesis is organised according to the approach described in Section 1.2. The thesis is split into five parts each will answer a (sub-) research question.

I Part I answers the question: What is the state of the organisation that will incorporate the new technology? It contains Chapter 2 which describes the company and the product, and Chapter 3 which describes the current state of the software development organisation.

II Part II answers the question: What is the nature of the technology? It contains Chapter 4 which describes the basics of ASD, and Chapter 5 which is about ASD's ancestor Cleanroom Software Engineering (CSE).

III Part III answers the question: What is the ultimate goal for acquiring and using the technology? It contains Chapter 6 which describes the objectives and expectations about ASD.

IV Part IV answers the question: What are the steps to reach the desired goals given the state of the organisation? It contains Chapter 7 which analyses the previous parts and proposes some recommendations about the transition.

V Part V answers the question: How can re-use and migration enhance the technology? It contains Chapter 8 which describes some solutions for re-use, Chapter 9 which describes a proof-of-concept for applying re-use, and Chapter 10 which describes a proof-of-concept for the migration of an application.

I What is the state of the organisation that will incorporate the new technology?

Before we can analyse the transition situation, some important aspects of the situation need to be examined. Every transition situation is unique. There are major differences between organisations in general but also for software development organisations specifically.

This part gives an overview of the context where the new technology is going to be introduced. The context is outlined by a description of the company, the product, and the current software development organisation with its involved roles, process and method.

2. Philips Healthcare

This chapter provides a global overview of the company involved and the product on which ASD is going to be applied. Subsequently, the following is described in this chapter: the company, the product Allura, the decomposition of Allura, and the FEC, which is one of Allura's components.

2.1 Company

Philips is an electronics company founded in 1891 by Gerard Philips in Eindhoven, the Netherlands. Today, Philips is organized into three main branches: Philips Customer Lifestyle, Philips Lighting, and Philips Healthcare (former Philips Medical Systems). The Healthcare branch focuses on five businesses: Healthcare Informatics, Diagnostic Monitoring, Patient Monitoring, Defibrillators, and Imaging Systems. The research described here took place in business unit Interventional X-ray (formerly called Cardio Vascular), which is part of the Imaging Systems business.

2.2 Allura

The business unit Interventional X-ray makes products for several medical segments. Some of these segments are cardiology, radiology, neuro-radiology, electro-physiology, and surgery. The general product name is Allura. There are multiple variations of this product for the several medical segments depending on the chosen (hardware) configuration and the (software) packages. In Figure 1 a possible monoplane configuration of the Allura product is depicted. The common factor of the product variations is that x-ray movies of a patient's body can be made in real-time.

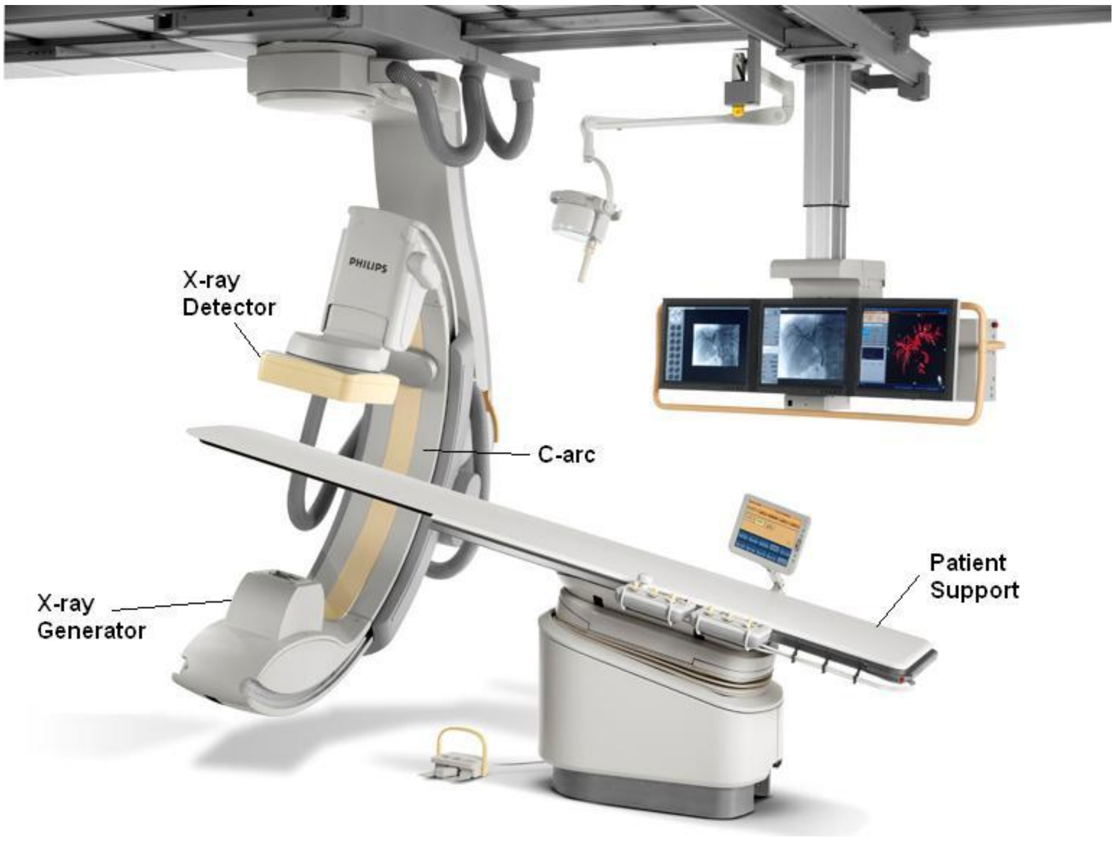


Figure 1, Allura Monoplane

The patient lies on the table and is positioned on the table between the x-ray generator and detector on the c-arc of the product. The table and c-arc of the product can be manoeuvred by means of a software user interface. One end of the c-arc transmits an x-ray beam through the patients' body and the other end of the c-arc receives the residual radiation. This received radiation is transformed into an image which can be processed and viewed by the physician and other operating room personnel. The variations of the product are for a large part determined by the software applications needed for specific medical segments. If, for example, a physician wants to place a stent into the aorta of a patient, then the product is used to navigate the stent through the patient's arteries to the target position. The arteries of the patient can be made visible by injecting a contrast media. When the contrast media is injected, Allura can be used to make the pictures.

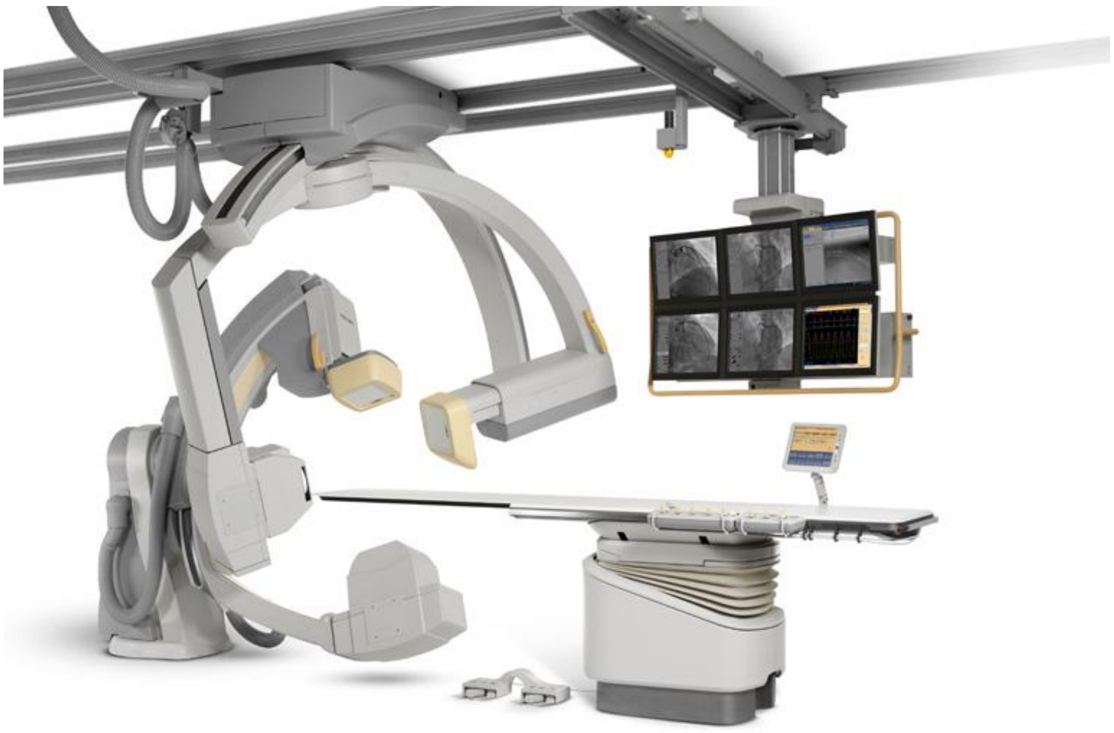


Figure 2, Allura Biplane

The product can consist of several hardware variations, for example, there is a product variant with two c-arcs. Figure 2 is a picture of the biplane variant where x-ray images from two sides can be made simultaneously.

2.3 Decomposing Allura

For creating the Allura product, many disciplines are involved such as mechanical engineering, electrical engineering, and software engineering. For this research, we are solely interested into the software of the product. Moreover, we concentrate on a small part of Allura's software.

Figure 3 shows that the software of the Allura product is decomposed into three subsystems: the front-end controller (FEC), back-end (BE), and image processing (IP). The FEC is responsible for interfacing with the devices, for example, movement and positioning of the product's arcs, activating and deactivating the x-ray beam, focusing the x-ray beam, and control the detector that captures the x-rays. The devices are depicted below the FEC in the picture. The IP subsystem processes the stream of acquired x-ray images for visualisation, and the BE is the user interface for the physician and other operating room personnel. The user interface is used to configure the device and visualize the pictures.

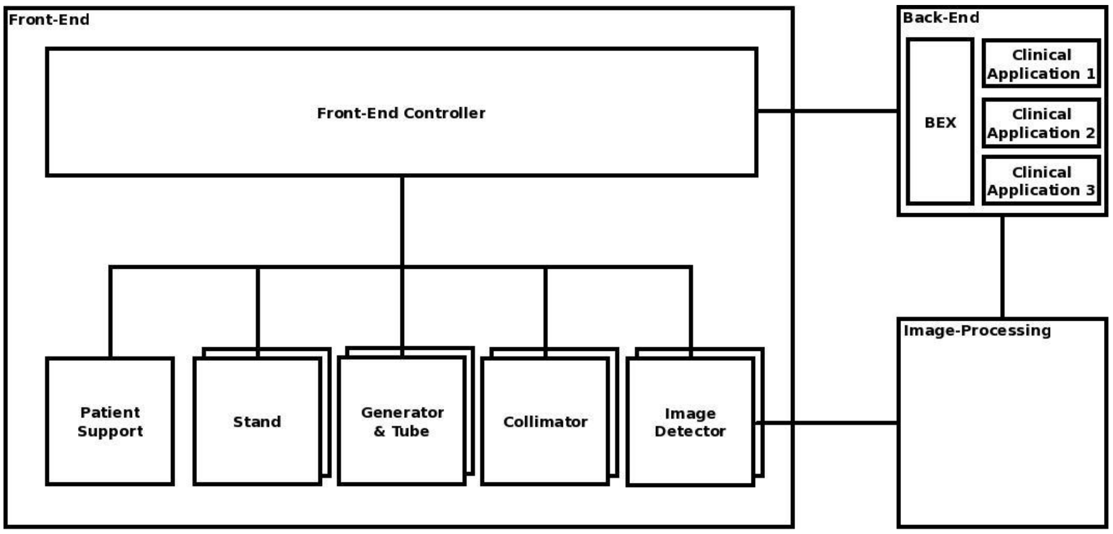


Figure 3, Allura's Architecture

The system is decomposed in such a way that the functional parts of the system are rather independent of each other. The Front-End (FE), which includes the hardware, needs more time to get released because of all the safety constraints. The decomposition accomplishes that FEC, BE and IP can be upgraded individually.

2.4 Front-End Controller

The FEC is the software that manages and controls the FE's hardware. ASD will be introduced in this FEC. The interfaces of FEC with BE and IP are already modelled with ASD. The FEC's architecture consists of three layers. From top to bottom these layers are:

- user-interaction layer,
- application layer, and
- technical-services layer.

The next chapter explains software development regarding the Front-End Controller: the organisation, the roles involved, and the software development process and method.

3. Software Development Organisation

The business unit Interventional X-ray makes products in multidisciplinary teams where the main disciplines are software, electronics, and mechanics. All disciplines need to work together in order to successfully develop Interventional X-ray products. Making the Front-End Controller (FEC) is a small but essential part of the whole development effort. This chapter starts large with describing the overall development organisation. Next the focus will be more on the FEC part of the development organisation. This chapter is based on internal documents and interviews with persons that work on FEC.

The chapter is organized as follows. Firstly, the matrix organisation is described in Section 3.1. Secondly, the roles involved in making the FEC are explained in Section 3.2. And, finally, the software development process and method is described in Section 3.3.

3.1 Matrix Organization

In this section, we describe the development organisation. A partial organisational chart of the development organisation is given by Figure 4.

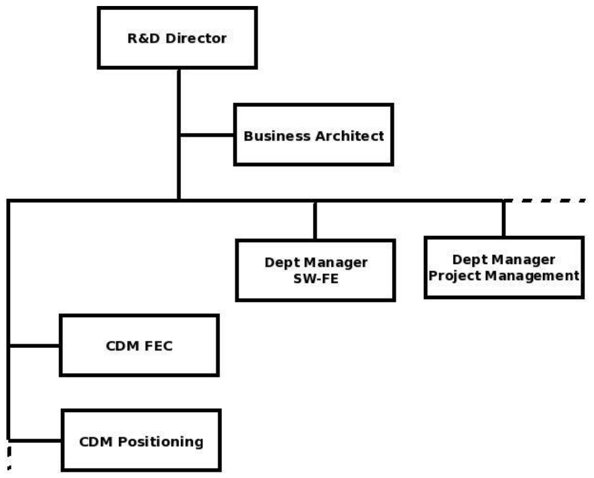
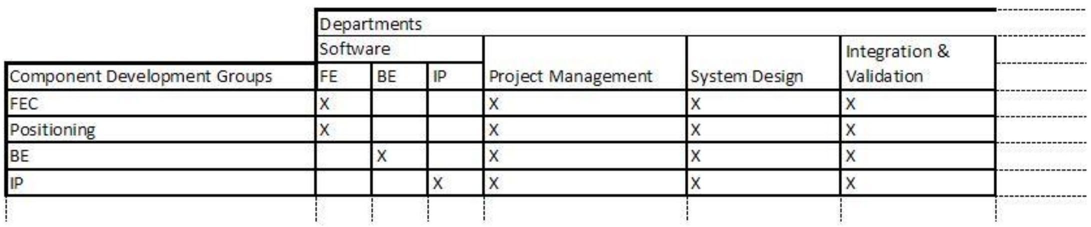


Figure 4, Organisational Chart

As can be seen in Figure 4, the development organisation is headed by the R&D director. Within the business unit Intervention X-ray, the business architect is responsible for aligning the different products technically. Additionally, the business architect is responsible that future products will incorporate visions of the future made by market forecasts.

The business unit Interventional X-ray has several organisational units, for example: innovation, R&D, marketing, customer service, etc. The organisation where the FEC is developed is part of the Research & Development (R&D) organisation. Figure 5 shows that the R&D organisation consists of a matrix. On one axis there are Component Development Groups (CDG), which are headed by Component Development Managers (CDM), and on the other axis there are the departments.



	Departments					
	Software					Integration &
Component Development Groups	FE	BE	IP	Project Management	System Design	Validation
FEC	X			X	X	X
Positioning	X			X	X	X
BE		X		X	X	X
IP			X	X	X	X

Figure 5, Matrix Organization

In Figure 6, the matrix organisation is mapped onto the hardware decomposition of Allura. In the following subsections, we give a description of the CDGs and departments relevant for this research.

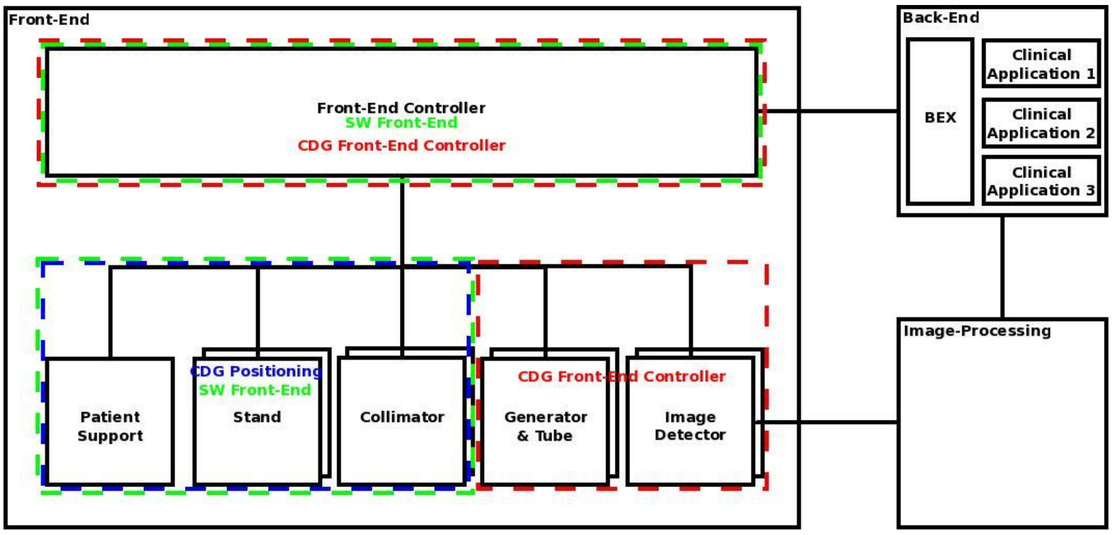


Figure 6, Front-End Organisation

3.1.1 Component Development Groups

The Allura device consists of several component, such as FEC, Positioning, BE, and IP. These are explained in a previous chapter about the Allura's decomposition, but there are also other components.

The component development group FEC is responsible for the generation of the x-ray pictures. Making x-ray pictures involves the hardware to generate an x-ray beam and to acquire the residual x-rays which will be turned into a picture. The component development group FEC's task is to build and maintain software that manages and controls the devices. Additionally, the component development group FEC's task is to build and maintain the devices required to make the x-ray pictures. As can be seen in Figure 6, these devices are: Generator & Tube and Image Detector.

The component development group Positioning is responsible for positioning the patient with respect to the x-ray beam in a proper way to acquire pictures from the exact location needed. As depicted in Figure 6, the task of CDG Positioning is to build and maintain the devices Patient Support, Stand, and Collimator. These devices also incorporate software.

To accomplish its objectives, a component development group gets its needed resources, in terms of personnel, from the departments.

3.1.2 Departments

The R&D organisation consists of several departments. The departments are grouped by discipline. Examples of departments are SW-Front End (SW-FE), Project Management, System Design, and Integration & Verification. The software department (SW) is split into several departments. SW-Front End is the software department that, for example, provides resources for the component development groups FEC and Positioning. The resources that can be provided by the department SW-Front End are software architects, software designers, and software engineers.

Inherently, working with a matrix organisation implies that the work on existing or new products will be done in projects. Projects can be used, for example, to develop a new product or to upgrade an existing product. The component development managers will initiate projects. Depending on the projects' needs for the coming year, the component development groups request the departments for resources on an annual basis. For example, a number of software designers from the SW-Front End department and a few testers from the Validation & Release department. The roles of the software department front end are explained in Section 3.2.

Summarizing, the organisational component development group FEC and the Positioning component development group will work on the technical FEC with personnel from SW-FE. Every person working on the FEC has two managers: a component development manager and a department manager.

3.2 Roles Involved

A software development organisation for creating and maintaining the FEC can distinguish several roles for the different activities needed to develop software. We concentrate on the roles involved in the creation and maintenance of the FEC, and omit others such as certain management roles.

3.2.1 System Architect

The system architect develops a conceptual view on the system. This basic view involves arranging the functionality into subsystems and to describe the relation between those subsystems. This view is cross functional and will involve multiple disciplines.

Among others, the system architect makes the conceptual view of the software architecture. The software of a subsystem is not isolated. It should work and interact with other software parts and with hardware devices; therefore negotiation about the interfaces with other parts of the system is also part of the role. How the interface will look like and what the responsibilities are have to be discussed by the software architect and possibly others such as an architect or designer from the electronics department. The activities of the system architect are done within a study. Studies are explained in Section 3.3. After studies, projects start to implement the architecture. Usually, the system architect will not be involved in the project activity but stays technically responsible for the end result.

Every component development group has a system architect who is the owner of the architecture and is technically responsible. This means that the architect should keep a long-term vision on how the product will evolve in the foreseeable future. The system architect has to be sure that current work on the architecture will not undermine the vision of the future.

3.2.2 Software Architect

The role of the software architect is to make a global design of the complete software architecture of a subsystem or a part of the system. The software architect develops the most high level, abstract, view of the software architecture. In this case a part of the whole software architecture is done by a software architect. The other parts of the architecture have their own software architects.

3.2.3 Software Designer

When the software architect has decomposed the system into units, and has defined the responsibilities of these units and how they should interact with each other, then for every module a detailed design is made by the software designer. Such a detailed design can be seen as a refinement of the conceptual view made by the software architect. The refinement will decompose a unit into even smaller entities called modules, which describe more details. When it is not feasible to decompose the entities further, then these modules need to be implemented by the software engineer.

3.2.4 Software Engineer

The software engineer implements the unit, or a part of the unit, made by the software designer. The activity starts by making the units that build the unit. The software engineer is responsible to test the modules with so-called module tests. In practice, the software designer and software engineering role are often combined. Hence, one person can have multiple roles.

3.2.5 Project Leader

A project consists of a group of persons that should accomplish a certain objective. The activities of the software designer and software engineer are done in a project. Project objectives are short-term objectives. In contrast, the system architect should keep the long-term objectives to mind. The project leader’s main responsibility is the on-time delivery of the planned work. The estimates on how long it should approximately take to build or change an architectural module come from the system architect and software architects.

3.3 The Development Process and Method

This section roughly describes the development process. Before a project starts, an advanced development study is done to prepare the start of the project. For example, the work to design the new reference architecture is done in a study, but the actual building of the new architecture will be done as project work.

The business unit has a generic meta process model for developing components. The generic process model is used by all disciplines. The software discipline has mapped the V-model as a reference to guide the process on the business unit’s generic process model. The V-model defines some clear phases, see Figure 7. For every phase on the left side of the V there is an associated test activity on the right side of the V.

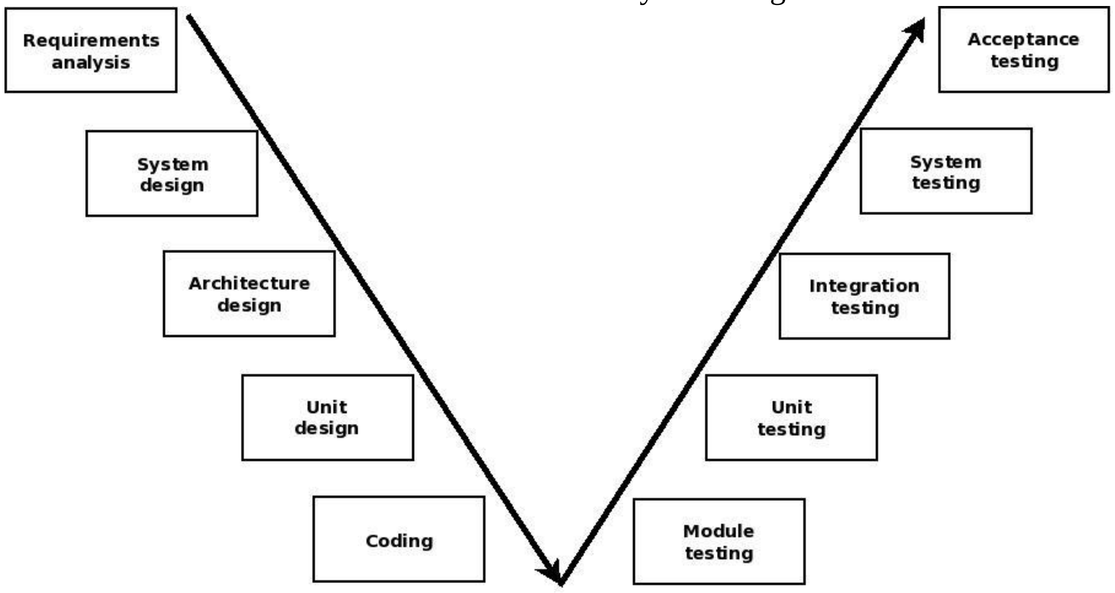


Figure 7, V-Model

During a study, the global design of the newly or adapted system is developed. The system architect builds a conceptual view of the whole system including all disciplines necessary. For a software architecture, the system architect and software architects refine the conceptual view further into units. This is the global design and it describes the responsibilities per unit. At the end of a study, the system architect together with the software architects make a work breakdown estimate for the work necessary to implement the global design. After the work of the study is finished, the project can start.

The software designers first make per unit a detailed design. Later the software designer or software engineer implements the unit and creates tests for testing the unit. When the unit is implemented, the unit tests are used to test it. The integration and Integration testing of the units is done by persons of the Integration & Verification department during the test & integration phase. Verification is very important for medical products because the products have to comply with all kinds of standards such as that of the Food and Drug Administration (FDA) from the United States.

For the FEC, the phases Architectural design, Unit design, Coding, Module testing, and Unit testing of Figure 7 will be done by SW-FE staff. Currently, the Object Oriented Analysis and Design (OOAD) method is used to make the designs. Object oriented designs are described by means of Unified Modelling Language (UML). The implementation of the source code of a unit is done with the object oriented programming language C++.

In the remainder of this thesis, we distinguish the following three major phases: global design, detailed design, and test & integration. In Figure 8, we have depicted the three major phases into the V-model.

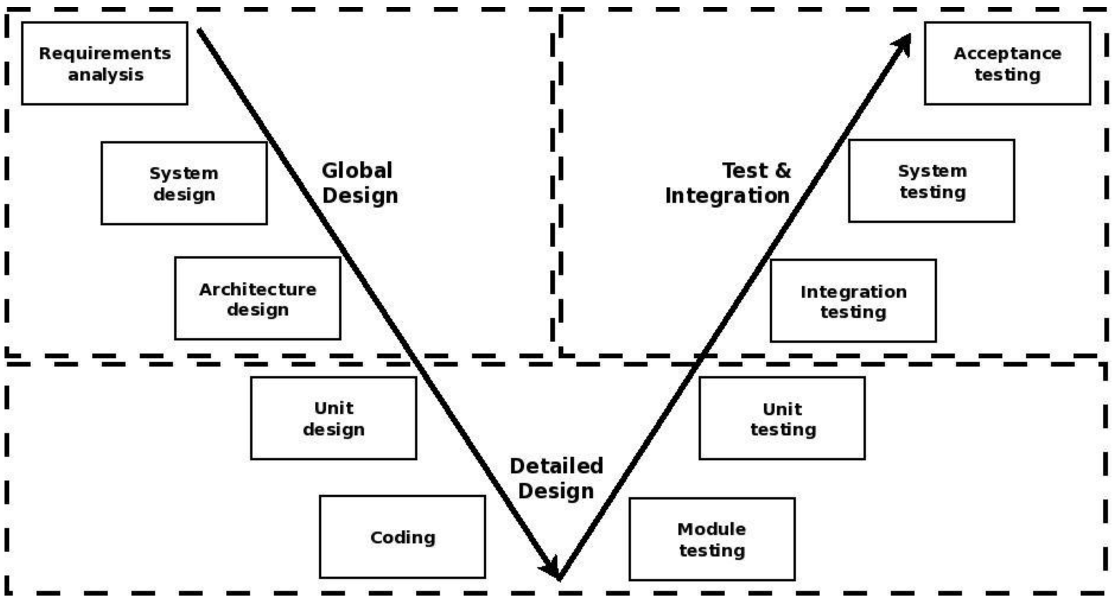


Figure 8, V-Model (Three Phases)

The system and software architects are responsible for the global design, and the software designers and software engineers are responsible for the detailed design. Test & integration is done by staff of the Integration & Validation department.

I Concluding Remarks

An important aspect of the transition situation is the current state of the organisation that will incorporate the new technology. Therefore, Part I of the thesis sketched the global outline of the organisation while answering the question: What is the state of the organisation that will incorporate the new technology?

The business unit Interventional X-ray has a generic process model used for all development activities, regardless of the engineering discipline. Hence, mechanical engineering, electrical engineering and software engineering all use the same meta process model. The software development organisation currently uses the V-model which fits into the generic process model. In this thesis, we distinguish the following three main phases: global design, detailed design, and test & integration.

Components of products are made in projects. For each project, the required persons are claimed from the departments, which are organized by discipline. The software departments can offer software architects, software designers, and software engineers. These persons are trained and used to develop software with an Object Oriented Analysis and Design (OOAD) method. Moreover, for the Front End component considered in this thesis, the software is developed using the Object Oriented programming language C++.

II What is the nature of the technology?

Part I described a global overview of the organisation, in which the new technology should be infused. Besides the current state of the software organisation, the nature of the technology is relevant for the transition situation. Not only organisations differ, technologies are also different. Therefore, in Part II, we explore the nature of the new technology.

Part II provides a global description of the new technology Analytic Software Design (ASD). We also describe the Cleanroom methodology, because it provides an historical perspective and methodological context. The aim is not to provide an in-depth introduction into the technology; this can be found in user manuals and course material. This part only describes the technology’s basics and the aspect of the technology relevant to the transition situation.

4. Introduction to ASD

Traditionally, in industry, software is specified and designed informally. The use of natural language will result in ambiguous and unclear specifications. This leads to errors which are usually removed during testing. This is fairly late in the software development process. The earlier errors are eliminated, the lower the costs and the more predictable, effective and efficient the software development process becomes. In academia formal methods are usually applied to formally verify the correctness of ("toy-") programs or program fragments. However, this is a very time consuming non-trivial task. The ASD method, from a company called Verum Consultants B.V., tries to bridge the gap between the formal methods from academia and the current informal practices in industry. [4][5]

ASD is based on the process algebra CSP. CSP is a modelling language that can be used to describe the behaviour of a system. Verum's main contribution is the development of a compositional approach and tool support to make it suitable for industrial use. When a model has been built, properties of the model can be checked such as absence of livelock, deadlock, and race conditions. In addition, the Verum tool can be used to generate source code and tests from the models. The latter can be used for testing handwritten source code. [2]

In Section 4.1, we provide a description of ASD's main concepts, and in Section 4.2 we describe the properties that the model checker verifies. For more details about ASD, we refer to the user manual [27] and the course material [28]. This material was also used to write this chapter.

4.1 The Technology

This section provides a description of ASD's main concepts. The objective of the description is to provide a global overview of the technology. We use a digital camera example to illustrate the basic concepts of ASD. It is used because of the similarity with the Allura product. As shown in Figure 9, the camera consists of three groups of buttons that the user can push: Power, MainMenu, and ChooseTheme. Furthermore, the camera has the ReturnAction group of events which are used to inform the user. The camera uses the interfaces of four devices: Aperture, Film Speed, Shutter Speed, and Battery.

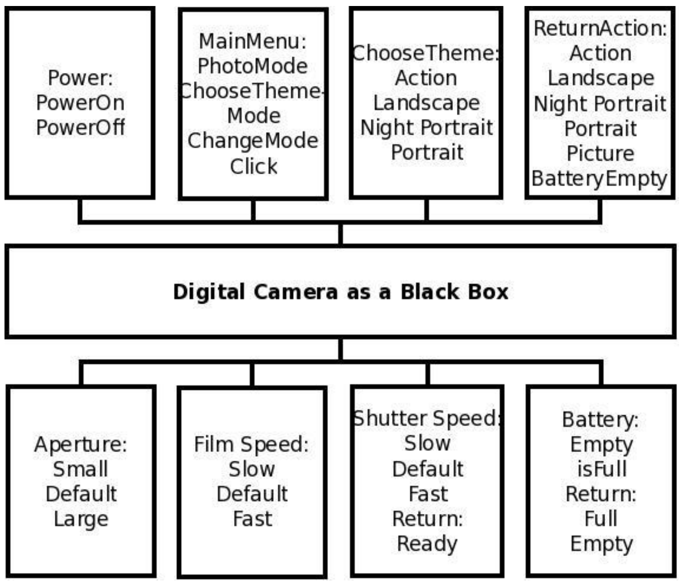


Figure 9, Camera

The settings of the devices are different for each theme. According to the application theme, the configuration settings are loaded to the devices. Of course, a digital camera does not have a film but the film speed device sets the sensitivity of the acquisition element. This has a similar effect as changing the film speed of an analogue camera. The aperture device of a camera controls the amount of light that reaches the acquisition element. When making a photo, the shutters open and light falls on the acquisition element.

4.1.1 Components

ASD is component-based which means that a design consists of components. Figure 10 depicts five components. Each component consists of an implemented interface (ellipse) and a design (box).

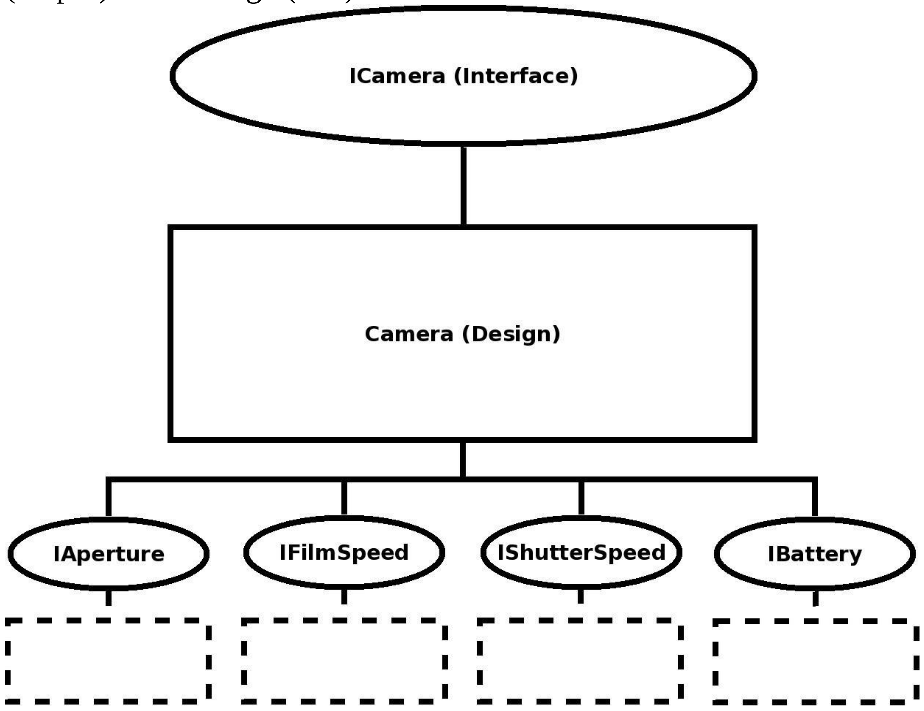


Figure 10, Components of Camera

In ASD the interface and the design are described in different models. A design model implements an implemented interface and uses used interfaces. The Camera (Design) implements the ICamera interface and uses the interfaces IAperture, IFilmSpeed, IShutterSpeed, and IBattery.

ASD can only describe control-based behaviour. To describe non-control-based behaviour, foreign components can be used. Additionally, foreign components can be used to glue ASD code to the rest of a system. A foreign component is handwritten code which conforms to an ASD interface model. For instance, some of the dashed-lined boxes in Figure 10 might be implemented as foreign components.

4.1.2 Decomposition

The used interface of one design model can be the implemented interface of another design model. Figure 11 is an extended version of Figure 10 which illustrates the decomposition of the camera example. The IServo's are added interfaces. The Aperture design implements the IAperture implemented interface and uses the IServo's used interfaces.

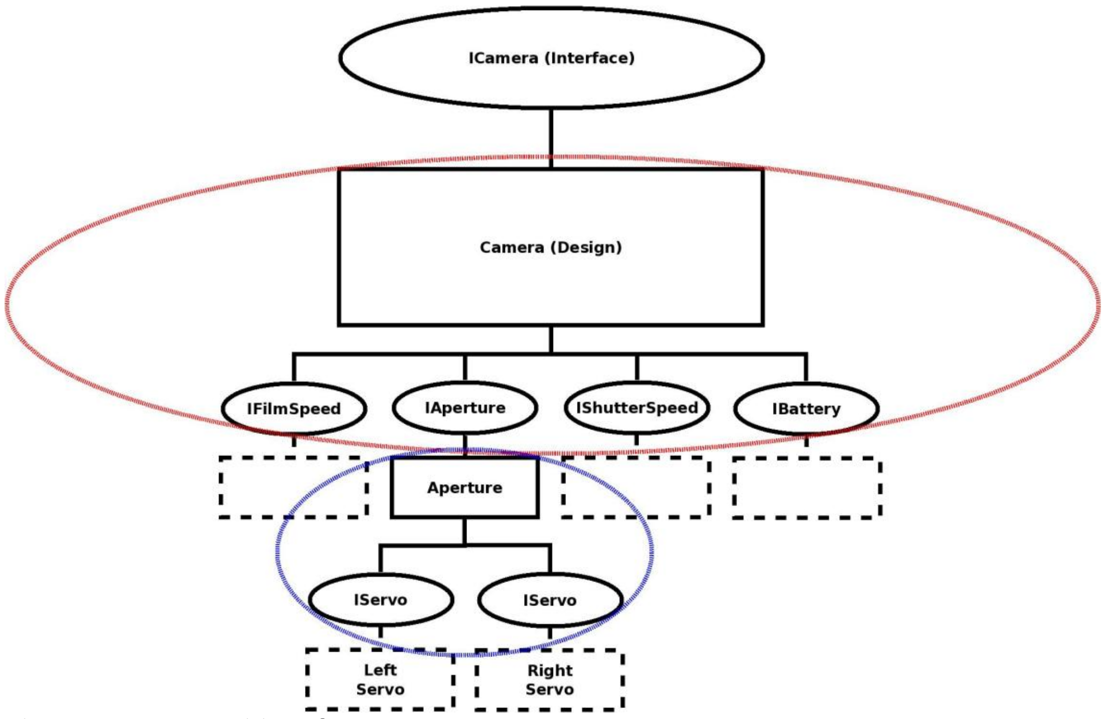


Figure 11, Decomposition of Camera

The ASD tool contains a model checker which can be used to verify that an implementation conforms to the specification. The specification refers to an implemented interface and the implementation refers to the composition of a design and its used interfaces. The composition of the implementation is depicted by the blue and red ellipses in Figure 11. ASD verifies a system in a compositional way. Recall that a used interface of one design can be the implemented interface of another design. For example, the model checker can be used to verify if the Camera design and its used interfaces IFilmSpeed, IAperture, IShutterSpeed and IBattery are conform the ICamera interface. Independently, the model checker can be used to verify if the Aperture design and its IServo used interfaces are conform the IAperture interface. Verifying a system in a compositional way makes model checking scalable. More details of the model checker are described in Section 4.2.

4.1.3 Communication

ASD is used to describe control-based behaviour in terms of function calls and callbacks. In an architecture, calls travel top-down and callbacks travel bottom-up as depicted by Figure 12.

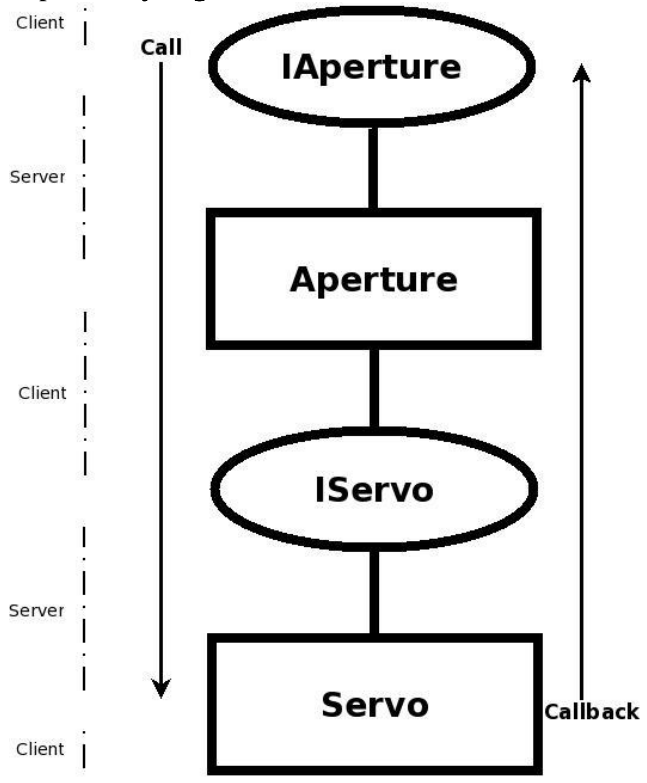


Figure 12, Communication

The Servo component acts as a server for client component Aperture. A call represents synchronous communication that travels top-down. A call is invoked on a server by a client. After completion, a call may provide the client with a return value. This communication is called synchronous because the client waits until the server has completed processing the call. During the waiting period, the client is unable to process requests in its server role for its clients. Figure 13 depicts synchronous communication.

A callback represents asynchronous communication that travels bottom-up. A callback is a communication from a server to a client. Received callbacks are decoupled at the client side by placing them into a First-In-First-Out (FIFO) queue. To prevent the queue from overflowing and to avoid race-conditions, the queue is always emptied before new calls on the implemented interface can be processed.

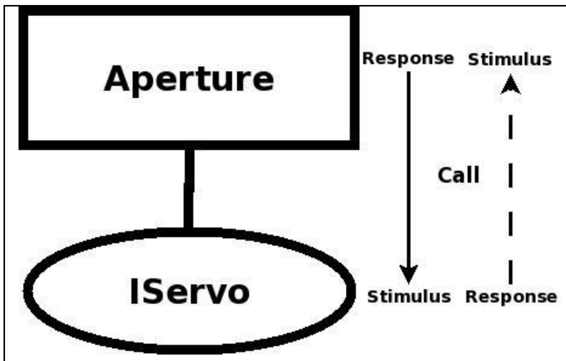


Figure 13, Call

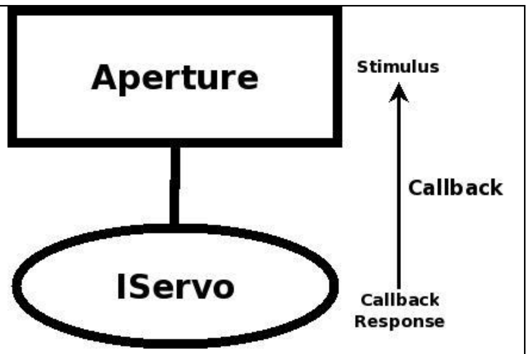


Figure 14, Callback

A call has either a void or valued return. Depending on the perspective, client or server, ASD terminology talks about responses and stimuli. A call may be invoked by a response at the client side. At the server side, the call is viewed as a stimulus. After the server has completed processing the call, it returns to the client. Optionally, a return value may be returned; this is called a response on the server side and a stimulus on the client side. Recall that actions are synchronous communication so the client waits until the call returns with or without return value.

A server can asynchronously send a callback to the client. At the server side, the callback is viewed as a response, whereas at the client side the callback is viewed as an asynchronous stimulus.

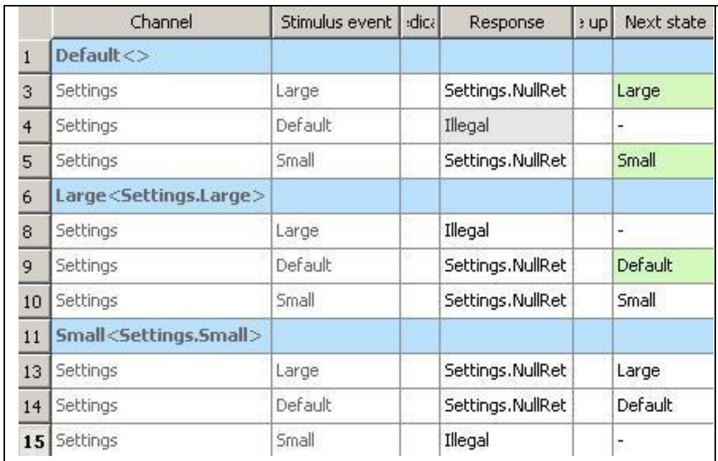
4.1.4 Channels

In a component's interface, it is possible to group related calls or callbacks into channels. Per interface, multiple channels are allowed. A Client API consists of a set of function calls. A Client CB consists of a set of callbacks.

Each Client API has a unique name and also all elements of a channel must have a unique name. However, elements of one Client API may have the same name as an element in another Client API. This also applies for Client CBs.

4.1.5 Sequence-Based Specification

As described before, each component consists of an interface model and a design model. A model describes the relation between stimuli and responses in a tabular form called a Sequence-Based Specification (SBS). A complete Sequence-Based Specification describes for all stimuli what its corresponding responses are and what the next state is. A SBS begins at the start state. If a stimulus is applied responses are instantiated and the state is changed to the next state. Figure 15 shows an example of the SBS of an interface model and Figure 16 represents the SBS as a Finite State Machine (FSM).



	Channel	Stimulus event	dica	Response	up	Next state
1	Default<>					
3	Settings	Large		Settings.NullRet		Large
4	Settings	Default		Illegal		-
5	Settings	Small		Settings.NullRet		Small
6	Large<Settings.Large>					
8	Settings	Large		Illegal		-
9	Settings	Default		Settings.NullRet		Default
10	Settings	Small		Settings.NullRet		Small
11	Small<Settings.Small>					
13	Settings	Large		Settings.NullRet		Large
14	Settings	Default		Settings.NullRet		Default
15	Settings	Small		Illegal		-

Figure 15, IAperture Interface Model (SBS)

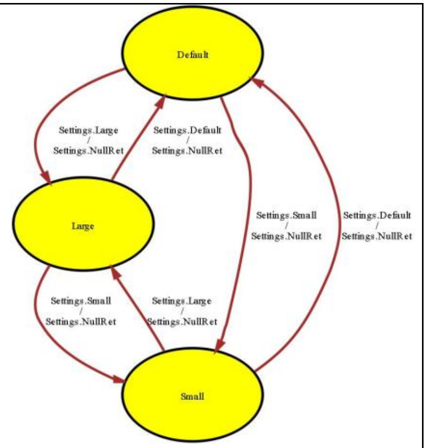
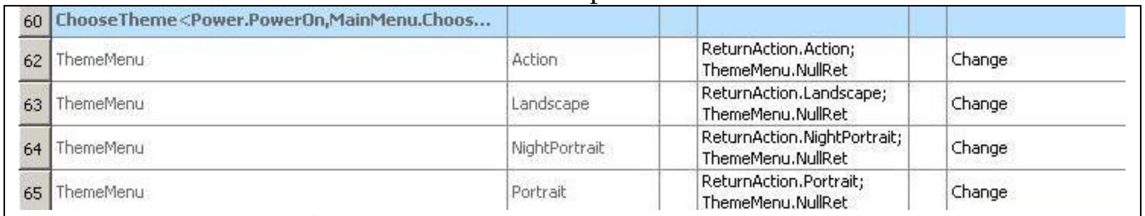

Default
Large
Small
Settings.Large / Settings.NullRet
Settings.Default / Settings.NullRet
Settings.Small / Settings.NullRet
Settings.Default / Settings.NullRet
Settings.Small / Settings.NullRet
Settings.Large / Settings.NullRet

Figure 16, SBS as FSM

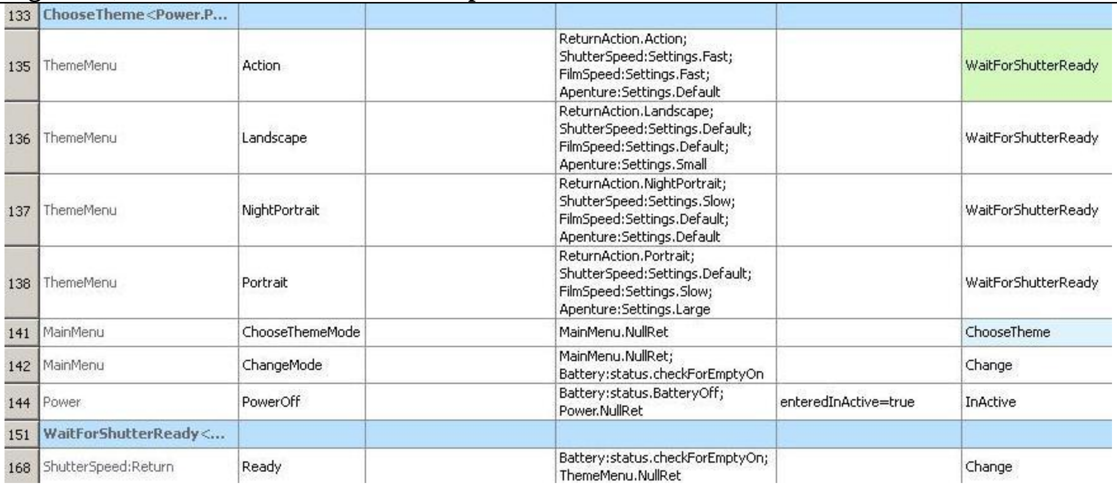
The most elementary columns of a Sequence-Based Specification are Channel, Stimulus event, Response, and Next state. The example SBS has one channel called Settings which consists of the three Stimulus events: Large, Default, and Small. Settings is a Client API and therefore Large, Default, and Small are calls. NullRet is a void return explained in the next subsection and the Next state defines the transition to the next state. Lines 1, 6, and 11 define the state names Default, Large, Small which – in this case - have the same names as the Stimuli events. Lines 3-5, 8-10, and 13-15 are called rules.

The SBS of a component's design model relates the implemented interface to the used interfaces. The camera design of Figure 18 relates the implemented interface of Figure 17 with the used interface of Figure 15 and the other used interfaces mentioned before. The ReturnAction channel will be explained in Subsection 4.1.8.



60	ChooseTheme<Power.PowerOn,MainMenu.Choos...					
62	ThemeMenu	Action		ReturnAction.Action; ThemeMenu.NullRet		Change
63	ThemeMenu	Landscape		ReturnAction.Landscape; ThemeMenu.NullRet		Change
64	ThemeMenu	NightPortrait		ReturnAction.NightPortrait; ThemeMenu.NullRet		Change
65	ThemeMenu	Portrait		ReturnAction.Portrait; ThemeMenu.NullRet		Change

Figure 17, ICamera Interface Model Snapshot



133	ChooseTheme<Power.P...					
135	ThemeMenu	Action		ReturnAction.Action; ShutterSpeed:Settings.Fast; FilmSpeed:Settings.Fast; Apenture:Settings.Default		WaitForShutterReady
136	ThemeMenu	Landscape		ReturnAction.Landscape; ShutterSpeed:Settings.Default; FilmSpeed:Settings.Default; Apenture:Settings.Small		WaitForShutterReady
137	ThemeMenu	NightPortrait		ReturnAction.NightPortrait; ShutterSpeed:Settings.Slow; FilmSpeed:Settings.Default; Apenture:Settings.Default		WaitForShutterReady
138	ThemeMenu	Portrait		ReturnAction.Portrait; ShutterSpeed:Settings.Default; FilmSpeed:Settings.Slow; Apenture:Settings.Large		WaitForShutterReady
141	MainMenu	ChooseThemeMode		MainMenu.NullRet		ChooseTheme
142	MainMenu	ChangeMode		MainMenu.NullRet; Battery:status.checkForEmptyOn		Change
144	Power	PowerOff		Battery:status.BatteryOff; Power.NullRet	enteredInActive=true	InActive
151	WaitForShutterReady<...					
168	ShutterSpeed:Return	Ready		Battery:status.checkForEmptyOn; ThemeMenu.NullRet		Change

Figure 18, Camera Design Model Snapshot

Line 63 of the implemented interface describes that when a Stimulus event Landscape is invoked, the Next state is Change. On line 136, the design invokes as a response -

among others - the call Small on the used interface and goes to the Change state via the WaitForShutterReady state. Subsection 4.1.8 explains why the Next state after the Stimulus event Landscape is WaitForShutterReady and not Change.

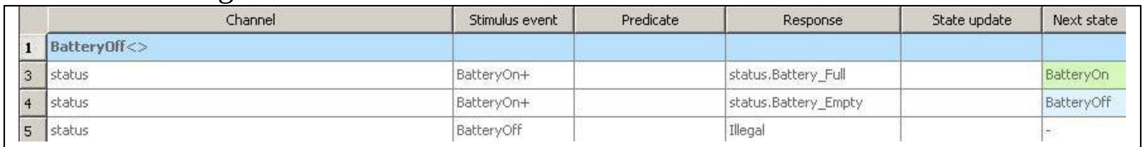
Line 136 of Figure 18 depicts that a call Aperture:Settings.Small, which is a response on the server side and a stimulus on the client side (lines 5, 10 and 15 of Figure 15).

4.1.6 Responses

Remember that calls can have either void or valued return. A call in a SBS can have a void return which is presented by a NullRet Response (Figure 15). In case of a valued return, in the Client API defines the values that can be returned.

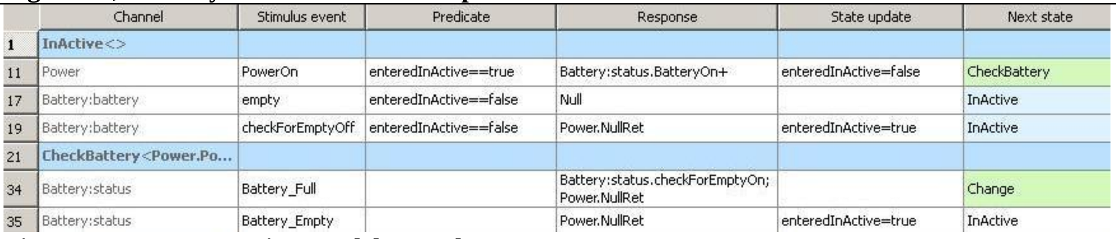
Figure 19 shows a snapshot of the IBattery interface model. The model describes that if in the state BatteryOff a call BatteryOn is invoked, the return value indicates whether the battery is full or empty. Line 3 describes that if a BatteryOn call is provided, the response may be Battery_Full and the next state is BatteryOn. Likewise, line 4 describes that if a BatteryOn call is provided, the response may be Battery_Empty and the next state is BatteryOff. Hence, in the interface model the result of the call BatteryOn is non-deterministic. As described, the return values Battery_Full and Battery_Empty are defined the Client API.

In Figure 20, the camera design model invokes BatteryOn when PowerOn is invoked and the Next state is CheckBattery (line 11). On lines 34 and 35, the Battery_Full and Battery_Empty call return values are responses of the IBattery interface and stimuli to the camera design.



	Channel	Stimulus event	Predicate	Response	State update	Next state
1	BatteryOff<>					
3	status	BatteryOn+		status.Battery_Full		BatteryOn
4	status	BatteryOn+		status.Battery_Empty		BatteryOff
5	status	BatteryOff		Illegal		-

Figure 19, IBattery Interface Model Snapshot



	Channel	Stimulus event	Predicate	Response	State update	Next state
1	InActive<>					
11	Power	PowerOn	enteredInActive==true	Battery:status.BatteryOn+	enteredInActive=false	CheckBattery
17	Battery:battery	empty	enteredInActive==false	Null		InActive
19	Battery:battery	checkForEmptyOff	enteredInActive==false	Power.NullRet	enteredInActive=true	InActive
21	CheckBattery<Power.Po...					
34	Battery:status	Battery_Full		Battery:status.checkForEmptyOn; Power.NullRet		Change
35	Battery:status	Battery_Empty		Power.NullRet	enteredInActive=true	InActive

Figure 20, Camera Design Model Snapshot

We have described void and valued returns of actions; however, there are more response possibilities. A response can indicate that a call is:

1. Not allowed, by the keyword Illegal. In Figure 19, line 5 describes that in state BatteryOff the Stimulus event BatterOff is not allowed.
2. Blocked, this is indicated by the keyword Blocked. When blocked should be used, is explained in Subsection 4.1.8.
3. Null, meaning that no response is given.

In the ASD tool rules with an illegal or blocked response can be hided to condense the tabular view.

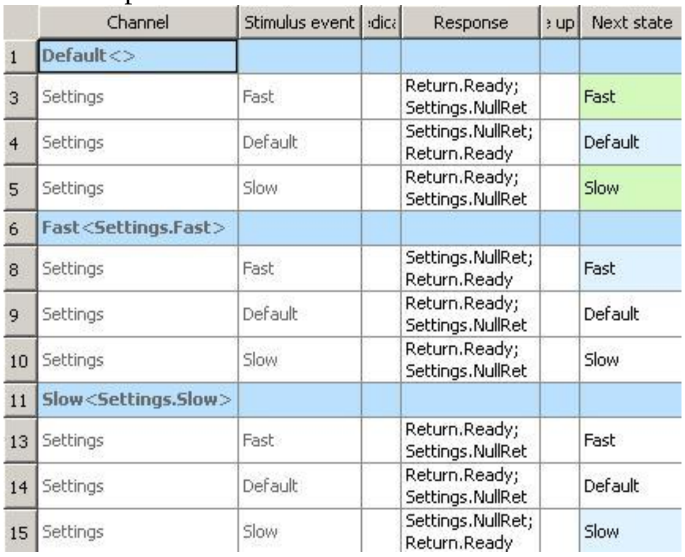
4.1.7 State variable

To reduce the number of states, state variables can be used. If two states do more or less the same or are in another way related, they can be combined into one state by means of state variables. State variables are defined by a name and an initial value. In a state, for one stimulus multiple rules can be created with a different predicate. The predicate is a Boolean expression containing state variables. In the state update part of a rule, the value of a state variable can be updated. In Figure 20, a snapshot of a SBS with a Boolean state variable enteredInActive is shown. As an example, we explain line 19. If in the state InActive, enteredInActive is false, and a Stimulus event checkForEmptyOff is invoked, then a void return is given, enteredInActive is updated to true, and the next state is also InActive. Note that the Illegal and Blocked responses are hidden.

4.1.8 Durative vs. Non-Durative

In Section 4.1.3, we have explained that calls are synchronous and that clients wait while a call is processed by the server. When the server has completed processing, the client is released by a void or valued return of the call. This communication scheme is called non-durative.

The durative communication scheme is used to release the client before the processing of a call is completed. When the server has completed processing the call, the client is notified by the server with a callback. Figure 21 is the IShutterSpeed interface model. Setting the shutter speed takes time. Figure 18 shows a snapshot of a design that uses IShutterSpeed as used interface.



	Channel	Stimulus event	dica	Response	up	Next state
1	Default<>					
3	Settings	Fast		Return.Ready; Settings.NullRet		Fast
4	Settings	Default		Settings.NullRet; Return.Ready		Default
5	Settings	Slow		Return.Ready; Settings.NullRet		Slow
6	Fast<Settings.Fast>					
8	Settings	Fast		Settings.NullRet; Return.Ready		Fast
9	Settings	Default		Return.Ready; Settings.NullRet		Default
10	Settings	Slow		Return.Ready; Settings.NullRet		Slow
11	Slow<Settings.Slow>					
13	Settings	Fast		Return.Ready; Settings.NullRet		Fast
14	Settings	Default		Return.Ready; Settings.NullRet		Default
15	Settings	Slow		Settings.NullRet; Return.Ready		Slow

Figure 21, IShutterSpeed Interface Model

IShutterSpeed has Client CB channel Return with a callback response Ready. The Ready callback is used to notify the client that the shutter speed is set. When the user of the camera changes the application theme of the camera into Action mode (line 135 of Figure 18), the configuration of the devices shutter speed, film speed, and aperture is set and the state is changed to WaitForShutterReady. In the state WaitForShutterReady all Stimuli events except Ready are blocked. When the Ready callback is received, the Next state is Change (line 168 of Figure 18).

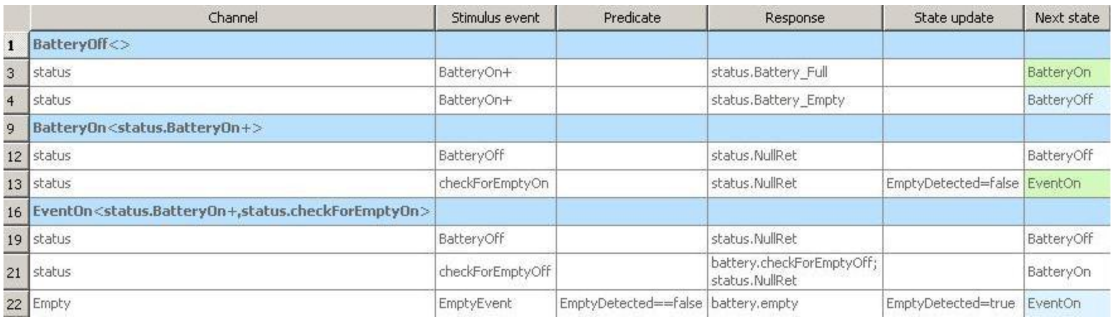
4.1.9 Solicited vs. Unsolicited

Callbacks can either be solicited or unsolicited. The callback that is sent in a durative communication scheme is solicited. The word “solicited” indicates that the client who invokes a call expects a callback when a durative action has been finished.

A callback can be unsolicited, meaning that it can be received at a random moment in time. If a component can receive unsolicited events, the design model has to define for each state what to do when such a callback is received.

4.1.10 Modelling Event

A modelling event is used in an interface specification to model an internal event, such as a callback sent by a foreign component. A modelling event may or may not happen, and may happen once or multiple times. Hence, such an event is unsolicited. Figure 22 shows the IBattery interface model. Note that Illegal and Blocked responses are hidden.



	Channel	Stimulus event	Predicate	Response	State update	Next state
1	BatteryOff<>					
3	status	BatteryOn+		status.Battery_Full		BatteryOn
4	status	BatteryOn+		status.Battery_Empty		BatteryOff
9	BatteryOn<status.BatteryOn+>					
12	status	BatteryOff		status.NullRet		BatteryOff
13	status	checkForEmptyOn		status.NullRet	EmptyDetected=false	EventOn
16	EventOn<status.BatteryOn+,status.checkForEmptyOn>					
19	status	BatteryOff		status.NullRet		BatteryOff
21	status	checkForEmptyOff		battery.checkForEmptyOff; status.NullRet		BatteryOn
22	Empty	EmptyEvent	EmptyDetected==false	battery.empty	EmptyDetected=true	EventOn

Figure 22, IBattery Interface Model

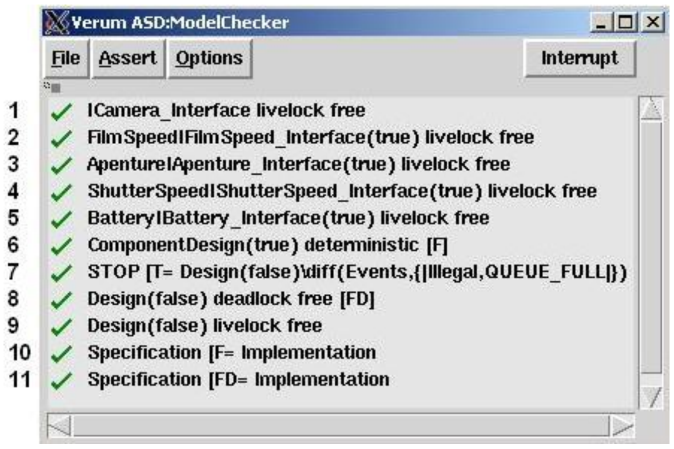
Line 22 describes a modelling event that notifies the camera when the battery is empty. Because a modelling event may happen multiple times, a state variable is used to send only one notification.

4.1.11 Sub-State

To manage the complexity of a design model ASD provides the possibility to use sub-states. The main state can change control to a sub-state which can implement partial behaviour. The sub-state gives control back to the main state when finished with its job. For more explanation about the described ASD concepts, we refer to the user manuals and course material. [27][28].

4.2 Model Checking

Model checking can be used to provide evidence about the correctness of a design. The model checker verifies whether certain properties hold for the design. When the property does not hold, the sequence of stimuli with leads to the failure is provided so that the failure can be analysed and solved. In the remainder of this section, we call a sequence of stimuli a trace. Figure 23 presents the model checker’s output for the camera design which satisfies all properties.



1 ✓ ICamera_Interface livelock free
2 ✓ FilmSpeedIFilmSpeed_Interface(true) livelock free
3 ✓ ApentureIApenture_Interface(true) livelock free
4 ✓ ShutterSpeedIShutterSpeed_Interface(true) livelock free
5 ✓ BatteryIBattery_Interface(true) livelock free
6 ✓ ComponentDesign(true) deterministic [F]
7 ✓ STOP [T= Design(false)\diff(Events,{|Illegal,QUEUE_FULL|})
8 ✓ Design(false) deadlock free [FD]
9 ✓ Design(false) livelock free
10 ✓ Specification [F= Implementation
11 ✓ Specification [FD= Implementation

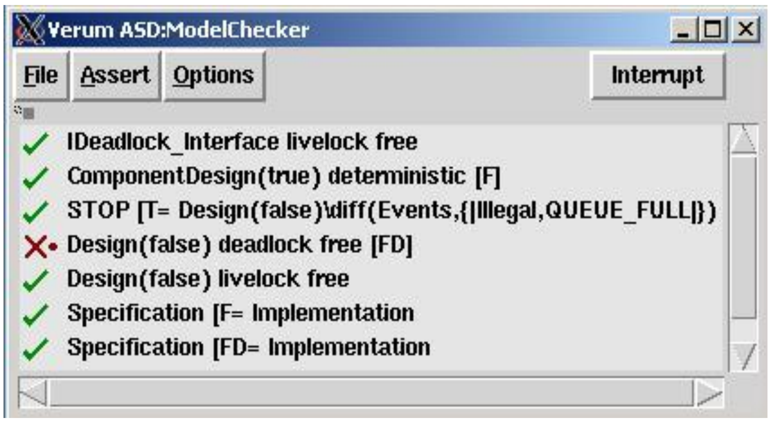
Figure 23, Model Checker

Subsection 4.2.1 describes line eight of Figure 23, which checks whether the design is deadlock free. Subsection 4.2.2 describes the first five lines and the line nine, concerning livelock freedom. Subsection 4.2.3 describes line six which checks whether the design is deterministic. Subsection 4.2.4 describes line seven. Line ten is described by Subsection 4.2.5 about failures refinement. Finally, Subsection 4.2.6 describes the last line of the model checker about failures-divergences refinement.

The explanations come from the model checkers manual [9] and [13].

4.2.1 Deadlock

The deadlock verifies whether there is a state in which no new call is possible. If such a state is present, then the model checker will give a trace leading to the deadlock. In the Figure 24 a state machine is depicted that will lead to deadlock after the first transition from state S0 to S1. Once in S1 it can never do a new call.



✓ IDeadlock_Interface livelock free
✓ ComponentDesign(true) deterministic [F]
✓ STOP [T= Design(false)\diff(Events,{|Illegal,QUEUE_FULL|})
✗ Design(false) deadlock free [FD]
✓ Design(false) livelock free
✓ Specification [F= Implementation
✓ Specification [FD= Implementation

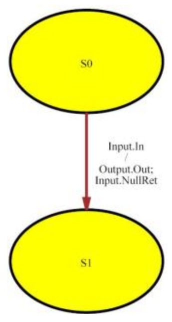
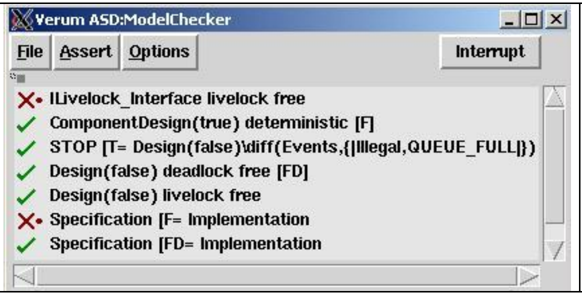


Figure 24, Deadlock

4.2.2 Livelock

A model has a so-called livelock if there is a trace from which there is an infinite loop of internal actions that lock up the process. Hence, after this trace external actions are impossible.



```
✗ ILivelock_Interface livelock free
✓ ComponentDesign(true) deterministic [F]
✓ STOP [T= Design(false)\diff(Events,{|Illegal,QUEUE_FULL|})
✓ Design(false) deadlock free [FD]
✓ Design(false) livelock free
✗ Specification [F= Implementation
✓ Specification [FD= Implementation
```

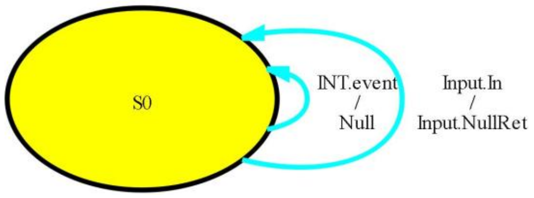
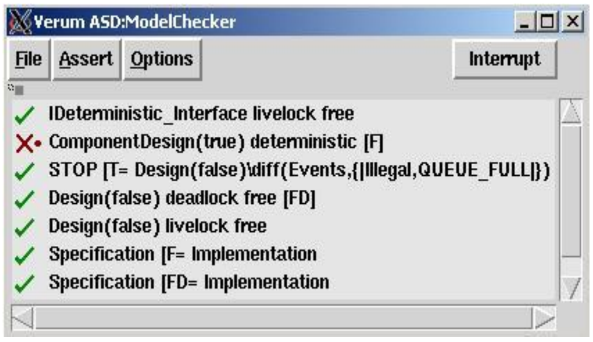


Figure 25, Livelock

In Figure 25, Input.In is an external call. The external call can become blocked by the modelling event INT.event which might lead to an infinite loop of internal events. Then, the system will no longer respond to external actions.

4.2.3 Deterministic

The model checker will verify whether a design model is deterministic. Deterministic means that, after a trace, at most one transition can be taken when a certain stimulus is provided to the system. In Figure 26, it is possible to take two different transitions. The first possibility is that the response Output.Out0 is given when Input.In is applied. The second possibility is that the response Output.Out1 is given when Input.In is applied.



```
✓ IDeterministic_Interface livelock free
✗ ComponentDesign(true) deterministic [F]
✓ STOP [T= Design(false)\diff(Events,{|Illegal,QUEUE_FULL|})
✓ Design(false) deadlock free [FD]
✓ Design(false) livelock free
✓ Specification [F= Implementation
✓ Specification [FD= Implementation
```

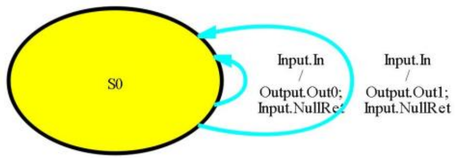
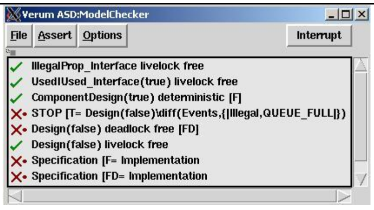

S0
Input.In / Output.Out0; Input.NullRet
Input.In / Output.Out1; Input.NullRet

Figure 26, Deterministic

4.2.4 Illegal and Full Queue

For the “Illegal” property, shown on line seven of Figure 23, the model checker verifies whether the design calls the used interface correctly. Figure 27 is an example of a used interface with two states. S0 is the initial state. When applying an “On” on the interface of Figure 27, a transition to S1 will be made. From S1 there can only be made a transition to S0 when “Off” is applied on the interface. Hence, in state S1 “On” is an illegal call. Figure 28 and Figure 29 are the interface and design of a component that uses the used interface in Figure 27. In Figure 29, the design applies two sequential “On” actions on the used interface which is not allowed by the used interface description and therefore this property does not hold.



✓ IllegalProp_Interface livelock free
✓ UsedIUsed_Interface(true) livelock free
✓ ComponentDesign(true) deterministic [F]
✗ STOP [T= Design(false)\diff(Events,{|Illegal,QUEUE_FULL|})
✗ Design(false) deadlock free [FD]
✓ Design(false) livelock free
✗ Specification [F= Implementation
✗ Specification [FD= Implementation

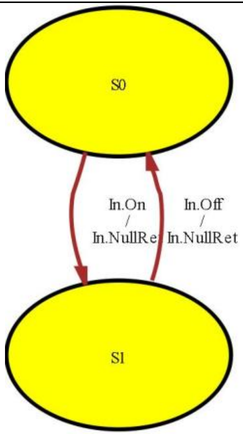


Figure 27, Used Interface

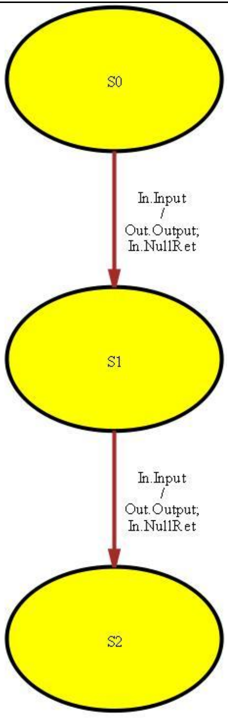


Figure 28, Interface

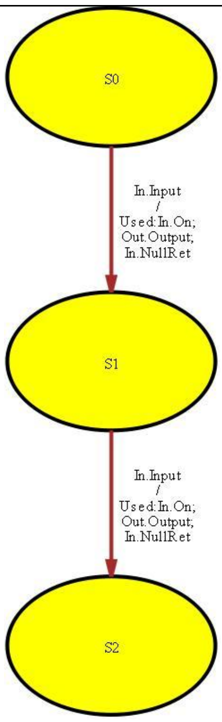
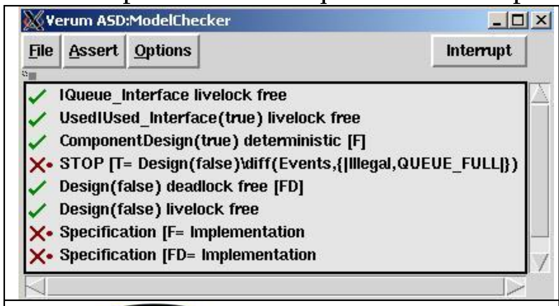


Figure 29, Design

For the “QUEUE_FULL” part of the property, the model checker verifies whether the design allows the queue to become full. In Figure 30, an example with the use of a modelling event is given. The used interface can always generate a callback event Out.Output. When the queue is not emptied fast enough, the queue is full.



✓ IQueue_Interface livelock free
✓ UsedIUsed_Interface(true) livelock free
✓ ComponentDesign(true) deterministic [F]
✗ STOP [T= Design(false)\diff(Events,{|Illegal,QUEUE_FULL|})
✓ Design(false) deadlock free [FD]
✓ Design(false) livelock free
✗ Specification [F= Implementation
✗ Specification [FD= Implementation

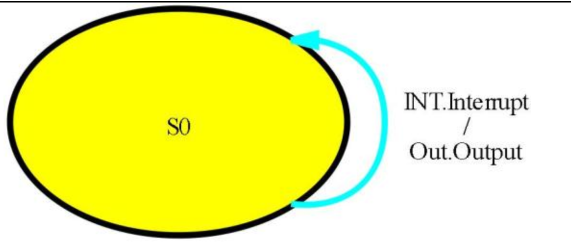


Figure 30, Used Interface

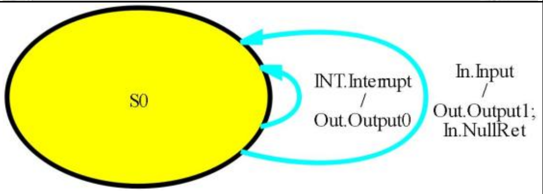


Figure 31, Interface

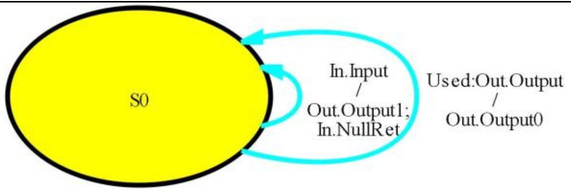


Figure 32, Design

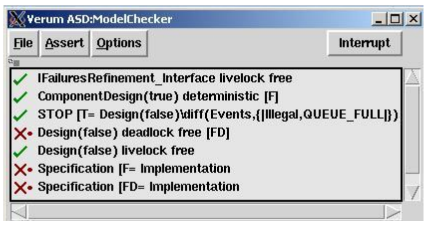
4.2.5 Failures Refinement

Failures Refinement or “Specification [F= Implementation” in the picture of the model checker, verifies if the implementation only has specified behaviour. Note that the specification refers to the implemented interface and that the implementation refers to the design and its used interfaces, see the ellipses of Figure 11. The specified behaviour expresses that no other actions than those specified are allowed by the caller and only the specified output than is given.

Whether the implementation only has the specified behaviour is verified by two checks:

- the traces of the implementation are a subset of the specification, and
- the failures (i.e., the refused calls) of the implementation are a subset of those of the specification.

Figure 33 and Figure 34 are an implementation and specification, respectively, for which the traces of the implementation are not a subset of the specification. The implementation has a trace with Ouput.Out1 while the specification does not. Failures are actions that refused after a trace. Figure 35 and Figure 36 are an implementation and specification respectively for which the failures of the implementation are not a subset of the specification. The implementation has the trace Input.In followed by Output.Out0 after which Output.Out1 is refused while the specification does not has this trace.



✓ IFailuresRefinement_Interface livelock free
✓ ComponentDesign(true) deterministic [F]
✓ STOP [T= Design(false)\diff(Events,{|Illegal,QUEUE_FULL|})
✗ Design(false) deadlock free [FD]
✓ Design(false) livelock free
✗ Specification [F= Implementation
✗ Specification [FD= Implementation

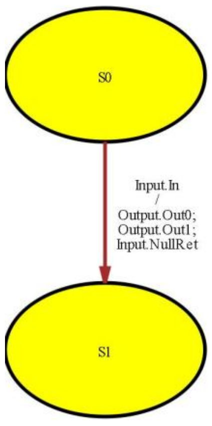


Figure 33, Implementation

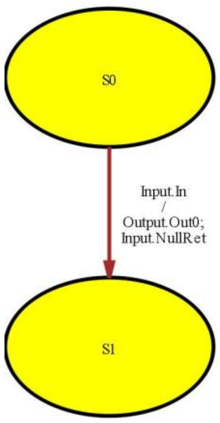
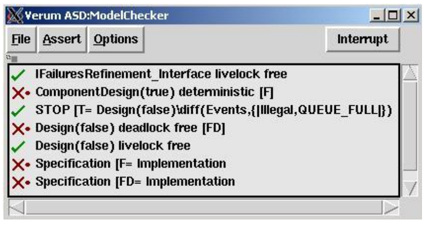


Figure 34, Specification



✓ IFailuresRefinement_Interface livelock free
✗ ComponentDesign(true) deterministic [F]
✓ STOP [T= Design(false)\diff(Events,{|Illegal,QUEUE_FULL|})
✗ Design(false) deadlock free [FD]
✓ Design(false) livelock free
✗ Specification [F= Implementation
✗ Specification [FD= Implementation

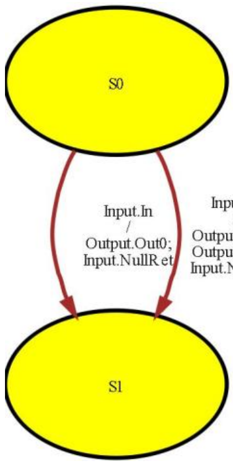


Figure 35, Implementation

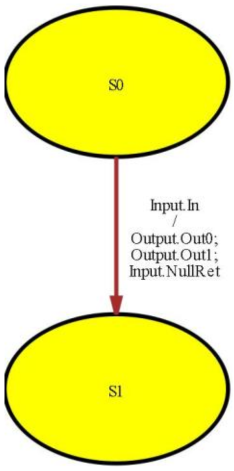


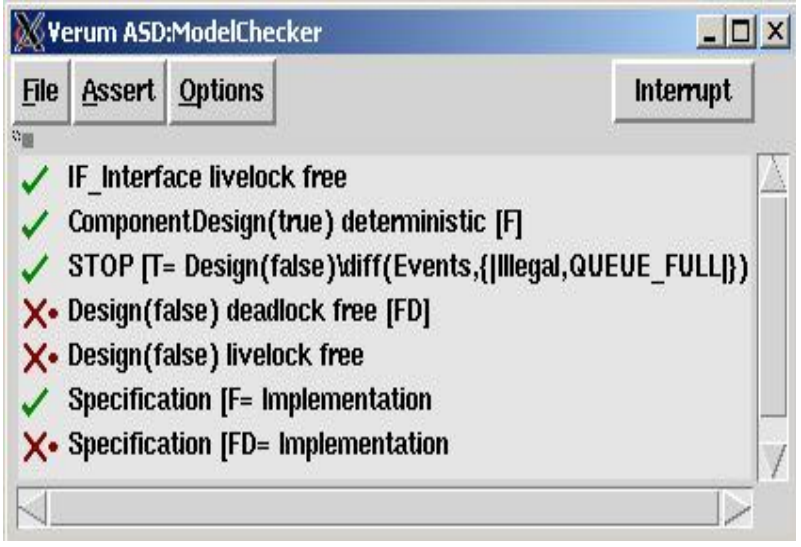
Figure 36, Specification

4.2.6 Failures-Divergences Refinement

"Specification [FD= Implementation" or failures-divergences refinement checks:

- failures refinement, and
- whether there is no trace leading to a livelock.

In Figure 37, is an example of state that produces a livelock. When the first call is applied, the super-state machine and the sub-state machine will infinitely loop and no external actions are accepted; this results in a livelock.



```
✓ IF_Interface livelock free
✓ ComponentDesign(true) deterministic [F]
✓ STOP [T= Design(false)\diff(Events,{|Illegal,QUEUE_FULL|})
✗ Design(false) deadlock free [FD]
✗ Design(false) livelock free
✓ Specification [F= Implementation
✗ Specification [FD= Implementation
```

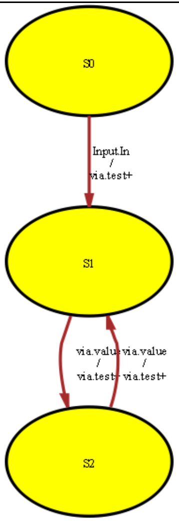


Figure 37, Divergences

5. Cleanroom

Whereas Chapter 4 described the main techniques of ASD and the tool support, this chapter describes a process and method. ASD fits into Cleanroom Software Engineering's development process and method. The following references are used Section 5.1 [18], Section 5.2 [20] and [24], and Section 5.3 [17].

5.1 *The Characteristics*

Cleanroom Software Engineering (CSE) is a software development method that, in analogy with the semiconductor industry, will focus on error prevention. The semiconductor industry makes integrated circuits. When making this hardware, the focus of the development process lies in error prevention through engineering discipline. The reason for the focus on error prevention is that it is very costly and time consuming to build prototypes. Therefore, they want to be sure that all errors are eliminated when the prototypes are built and thereby reducing development costs. Because of this analogy with the semiconductor industry, the method is called Cleanroom.

In fact, most engineering industries focus on error prevention whereas current methods in software industry mainly concentrate on error removal. In the software engineering industry, typically, the errors are removed in the last stage of development during test & integration. The general idea is that in this final stage of development, quality can be tested into the final product. However, errors identified in this final stage are very costly to remove. Depending on the error, it is even possible that at this stage it is necessary to change the architecture of the product which will further increase cost and development time. Cleanroom concentrates on error prevention to make the development process more predictable. The Cleanroom's philosophy is: why make erroneous code, if it is possible to make it right the first time? Cleanroom has been developed in the late 1970s by Harlan Mills and colleagues at IBM.

Cleanroom does not depend on tooling and is language independent; it has been used for C, Ada, and object oriented languages. It can be used for all software types such as control-based, algorithms, and data-centred. Legacy code can also be incorporated into the CSE.

5.2 *The Method*

A functional specification is made with boxes which are obtained by a process of stepwise refinement. This leads to a hierarchy of boxes which describe the needed system behaviour. Figure 38 is an example of such a box structure.

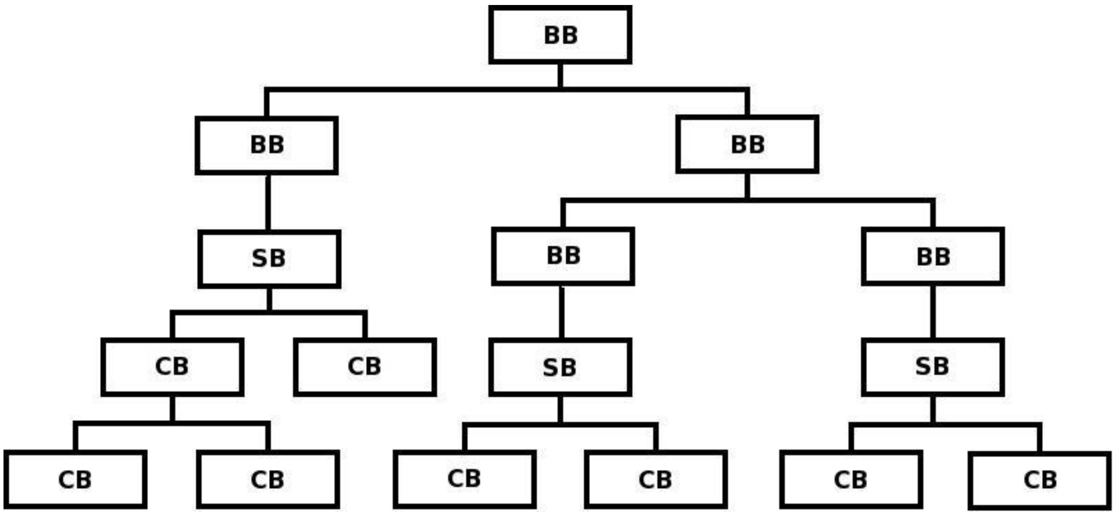


Figure 38, Box Structure

Cleanroom distinguishes three levels of data abstractions: Black Boxes (BB), State Boxes (SB), and Clear Boxes (CB). The black box describes the intended external behaviour the system should implement. The clear boxes are at the lowest level of abstraction describing structured programming constructs. While the state boxes are used as an intermediate step from external behaviour to programming constructs. State box specifications are a refinement of a black box specification, because both are specified formally state box specifications can formally be verified against its black box specification to ensure that the behaviour conforms to the specification. Similarly, clear box specifications can be formally verified against the specification of its corresponding state box specification.

A black box describes system or partial system behaviour in terms of stimuli and responses.

```
BB: S* -> R
```

S is a sequence of successive stimuli that are transformed by function BB into an observerable response R. A black box can be refined into new black boxes. The refinement builds a tree structured hierarchy of black boxes. This function can be given in a formal or informal language, or a mixture of both.

A state box refines a black box by adding a state machine; states are used to remember the history of previous applied stimuli. The state machine will take a transition from the current state to the next state when an allowed stimulus is applied. During this transition it may invoke a response, possibly a stimulus of another state box.

```
SB: S* x T* -> R x T
```

T represents the internal state of a box which is in fact an internal black box inside the state box. A state box can be further refined into clear boxes. For clear boxes structured programming constructs are used. Making clear boxes is done in the design activity.

The CSE approach scales because how large a system is, the top-level clear box will only be verified against its invoked clear boxes. The invoked clear boxes can represent large subsystems, but for the verification this does not matter. The verification will be recursively done until the clear boxes at the bottom of the hierarchy are verified. Also, the top-level clear box must be a refinement of its calling state box. Next this state box must be a refinement of its calling black box. This black box must be a refinement of its calling black box, etc.

To obtain a structure such as that of Figure 38, an 11-step box-structure process is described by [20]. For more detailed information, we refer to this paper.

Define the black box

1. Define black-box stimuli
   Determine all possible stimuli for the black box.
2. Define black-box behaviour
   For each possible stimulus, determine its complete response in terms of its stimulus history.

Design the state box

3. Discover state data requirements
   For each response to be calculated, encapsulate its stimulus history into a state data discontinuity without a lot of engineering requirement.
4. Define the state
   Select a subset of the required state data items to encapsulate stimulus histories.
5. Design the state box
   For the selected state, determine the internal black box required for the state box.
6. Verify the state box
   Verify the correctness of the state box with respect to the required black-box behavior.

Design the clear box

7. Discover state data accesses
   For each item of state data and each possible stimulus, determine all possible accesses of the item.
8. Define data abstractions
   Organize state data into data abstractions for effective access.
9. Design the clear box
   Define sequential or concurrent uses of the data abstractions defined to replace the internal black box of the state box.
10. Verify the clear box
    Verify the correctness of the clear box with respect to the state-box behavior.

Continue the process

11. Repeat stepwise expansion until design completion
    For each new data abstraction, repeat steps l-10 until suitable data and program specifications are reached.

5.3 The Process

In this section, the CSE process, as shown in Figure 39, is used to explain the technology. The process can be described by the following activities: specification, incremental planning, box structure specification and design, usage modelling and test case generation, and statistical testing. These activities are explained in the following subsections.

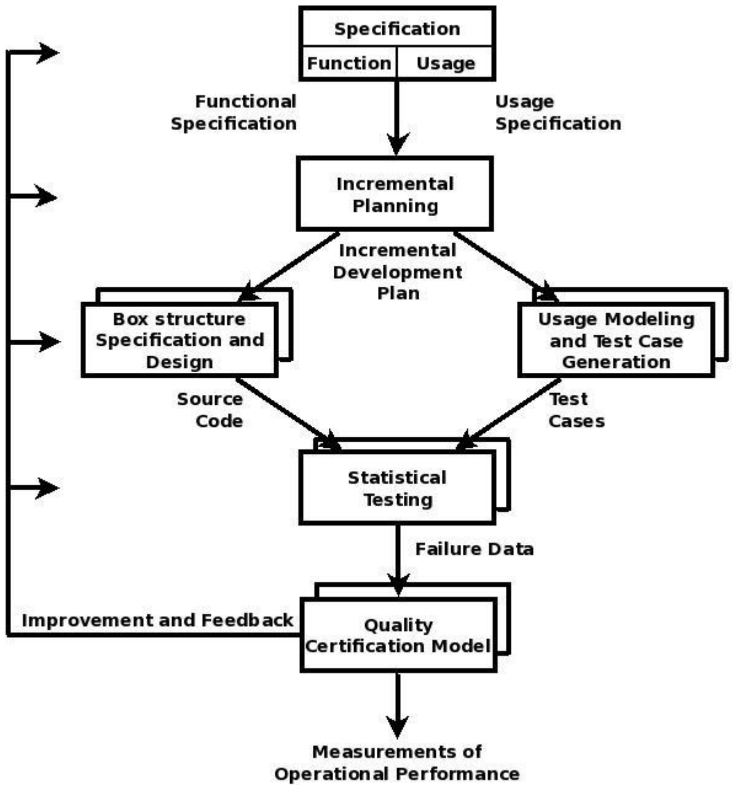

Specification
Function
Usage
Functional Specification
Usage Specification
Incremental Planning
Incremental Development Plan
Box structure Specification and Design
Usage Modeling and Test Case Generation
Source Code
Test Cases
Statistical Testing
Failure Data
Improvement and Feedback
Quality Certification Model
Measurements of Operational Performance

Figure 39, The CSE Process [17]

5.3.1 Specification

The Cleanroom process starts with a specification activity. During the specification activity, the system is decomposed into black boxes, see Figure 38.

Depending on the size of the project this is done by a separate team, or this is done by the development team and the certification team. Two specifications are made: a functional specification and a usage specification. The functional specification describes the external behaviour of a system or subsystem. The usage specification is used by the certification team, and defines usage scenarios and the probability of these scenarios for test case generation.

5.3.2 Incremental Planning

From a historical perspective, the idea of incremental development was radically new. Today most software methodologies advocate incremental development. Based on the specification of the external needed behaviour of a system, an incremental development plan is made. The plan describes the schedule, the resources, and what should be in each increment. In Cleanroom, the needed system behaviour is implemented top-down; consequently, for the first increment(s) stubs are made in order to be able to execute the source code. These stubs are replaced by later increments until all behaviour has been implemented. Figure 40 and Figure 41 are examples of the incremental implementation of a system by stubbing. The figures also show (with the A) that it is possible that clear boxes can be reused at different leaves of the tree.

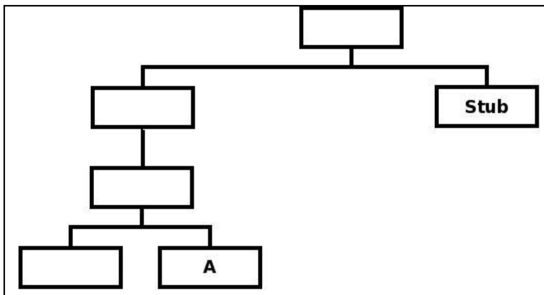


Figure 40, First Increment

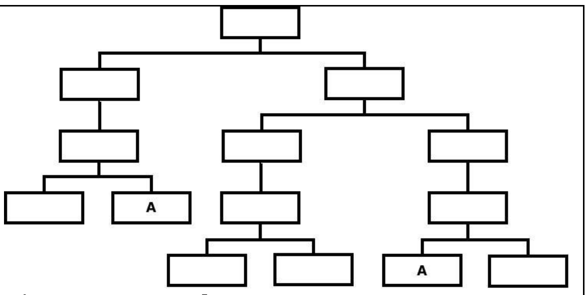


Figure 41, Second Increment

Each increment is integrated and tested. For feedback, the result of each increment is shown to the customer to validate requirements and behaviour. This feedback can be incorporated in subsequent increments, as can be seen in Figure 39.

5.3.3 Design

During specification the black boxes are made. Figure 38 shows that then the state and clear boxes need to be created. This is done during the design activity. First the state boxes and then the clear boxes are refined into elementary boxes by the development team. The idea behind this refinement is that these small, elementary clear boxes can easily be formally or informally verified by a proof. The correctness verification checks if the derived function of a clear box is equal to the intended function. The verification is done by the complete software team in the form of team reviews. All team members should agree that a certain proof is correct. Therefore, most present faults will not be spotted during the verification activity.

Figure 38 illustrates how a box structure developed with Cleanroom Software Engineering could look like. The final box structure is developed during the specification and design activities. Figure 42 illustrates which part of the box structure of Figure 38 is developed during the specification activity and design activity.

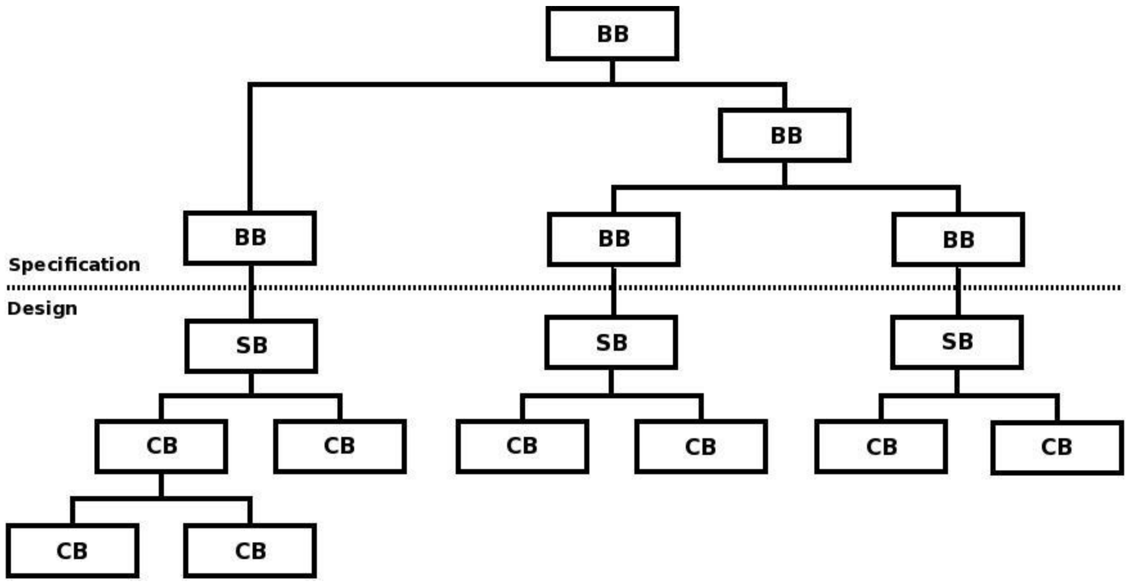


Figure 42, Box Structure during Specification and Design

5.3.4 Test Case Generation

While the development team develops the source code, concurrently the certification team develops usage scenarios from the usage specification. The usage scenarios and the probability of these scenarios are used to generate test cases. When generating test cases, the focus on external behaviour of the system for a certain increment. This concerns the behaviour of how the user of the system will experience interaction with the system.

5.3.5 Statistical Testing

The development team will not execute the source code. This is done for the first time by the certification team. Every increment, the certification team will integrate the developed system and run the test cases against the system. From the tested behaviour of the system, it is possible to statistically say something about the quality of the whole system. If it is not the last increment, this information is used for subsequent increments.

5.4 Drawbacks of Cleanroom

According to [12] there are three reasons why Cleanroom is not widely adopted by the industry: it "is too theoretical, too mathematical, and too radical". Traditionally, software is developed with software engineers that test their own source code with unit testing. Instead of unit tests, Cleanroom advocates that correctness verification and statistical quality control should be used. These two activities are radically different than how software is developed traditionally.

II Concluding Remarks

Part II provided an answer to the question: What is the nature of the technology? Below are the part’s concluding remarks.

Verum’s Analytic Software Design (ASD) is a new tool that can be used for designing control-based software in a component-based way. In ASD, a system is specified in a Sequence-Based Specification (SBS), which is a large table. The table describes for all states of the system how it should respond to all possible stimuli. A complete design specification can be verified formally. The Sequence-Based Specification can be used to generate source code. The source code can then be integrated into the final product. As said before, the tool is relatively new and therefore, during the research, new releases have further matured ASD.

The literature indicates that ASD is related to Cleanroom Software Engineering (CSE). Specifically, the component-based aspect of ASD originates from CSE. CSE has been developed in the late 1970s by IBM and describes the process and method that could be used to develop high quality software. The philosophy of CSE is error prevention instead of error removal during the test & integration phase of the software development process. The error prevention of CSE is accomplished by formally verifying designs by hand.

CSE is not widely used in the software industry because formal verification had to be done by hand. Verum's ASD tool eliminates this drawback by hiding the theoretical and mathematical processing. The user of the ASD tool is not bothered too much with the formal foundations the tool relies on.

Part III describes the ultimate goal of introducing and using the new technology in the organisation described by Part I.

III What is the ultimate goal for acquiring and using the technology?

In Part I and Part II the context of the software development organisation and the new technology are described. Part III explores the goal for acquiring and using the technology. The objectives and expectations of the introduction of the technology and the organisation's knowledge about the technology are described in this part of the thesis.

Part III investigates the following questions:

- What is the objective of introducing the new technology?
- What are the expectations of the new technology?
- What is the current knowledge of the managers and designers about the new technology?

6. Expectations of ASD

This chapter explores the expectations and objectives of introducing ASD into the FEC. To acquire the expectations and objectives, interviews with managers and software designers have been taken place. Additionally, the team's first steps in thinking about and applying ASD are observed. For both techniques [25] is used as reference. As described in Section 1.3, the research took place during the preparation phase of a pilot project; therefore, none of the interviewed software designers has made production software with ASD at the time of interviewing. The software designers are split into software designers responsible for the global design and software designers responsible for the detailed design of Allura's FEC because the implications of ASD for both groups differ. For the interviews, question lists are made for the designers (Appendix A) and managers (Appendix B). This chapter is a summary of the results of the interviews and reflects the relevant thoughts and opinions of the interviewees regarding their expectations about ASD.

6.1 Management

This section provides an overview of what management's objectives for introducing ASD are, their knowledge about ASD, what they expect from the introduction of ASD for the software development organisation and the technology, and how they prepared the transfer of technology. The business architect, component development manager, and department manager have been interviewed; Chapter 3 provides a description of these roles.

6.1.1 Objective

According to the managers, the software development process has become increasingly expensive. As shown in Figure 43, the effort needed for test & integration, drawn as a function of the time will look like an exponential curve.

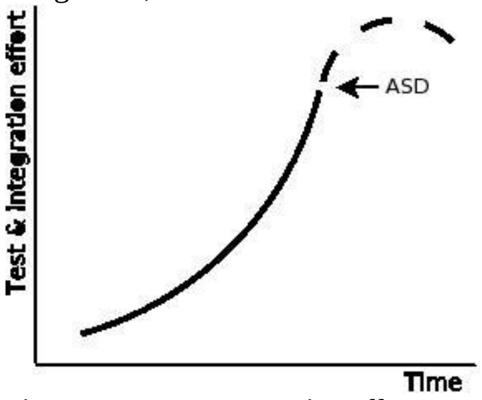


Figure 43, Test & Integration Effort

ASD should bend the curve, as depicted in Figure 43, and bring the test & integration time in proportion to the global design and detailed design effort.

Another driver for using ASD is composition (see Figure 45) instead of decomposition (see Figure 44). Decomposition is an activity when one product is built and decomposed in several subsystems. However, if multiple product variations can make use of the same software architecture, one of many variations can be composed of existing subsystems. Hence, not every product needs its own software architecture anymore. This will allow for maximal reuse of subsystems.

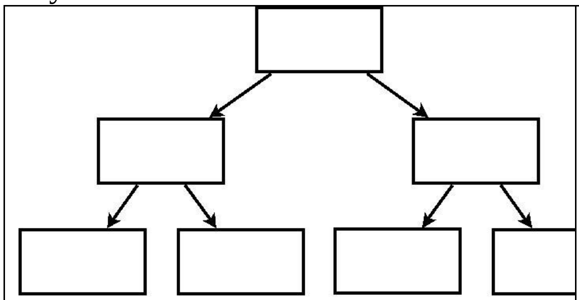


Figure 44, Decomposition of System

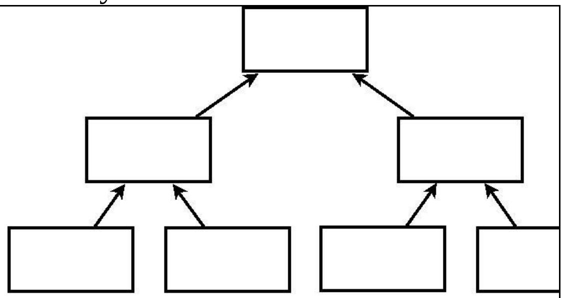


Figure 45, Composition of System

The business unit Interventional X-ray has numerous product variations and also many products in the field that need bug fixes and upgrades. If the composition of software is possible, only subsystems of the software can be updated or made that should work flawlessly with other (existing) components. When using subsystems, the interfaces between them are very important. ASD can be used to correctly describe the control behaviour of the interfaces.

6.1.2 Knowledge

According to the managers, ASD is a mathematical method. A software designer builds a model of the state behaviour of a component or subsystem with Verum's ASD tool. The model can then be tested on consistency and feasibility. From the behaviour of this correct model, source code can be generated. Components with clearly described interfaces can be used to compose products. ASD can clearly describe interfaces and the behaviour of these interfaces. This will help build higher quality software and there are less reparation cycles needed to acquire the needed quality.

ASD helps to ask questions about the requirements in an early stage of development instead of, as is current practice, in a later stage. This is less expensive then solving problems at a later stage. Another benefit is that if an existing component or subsystem needs to change, then only the design and not the source code needs to be refactored. This means that over time the design remains maintainable. With the current practice it is extremely difficult to let the design be or remain equivalent with the source code. It is always a question what should and what should not be included into the design document. Of course, the source code describes everything but the design document describes the source code at a higher level of abstraction.

The consequences of ASD are:

- ASD forces software designers need to think in another paradigm.
- The software designers need to describe behaviour more explicit in ASD.
- The current practice of software development needs to be adapted to ASD.
- Certain design concepts which are currently used are unsupported by ASD.

6.1.3 Software Development Organisation

According to the managers, the software industry is searching for means to go as quickly through the V-model as possible, so the software can be tested. Going quickly through the V-model is the currently used alternative of ASD. Software organisations are searching for methods where software can be composed into testable parts. Today, one can only test software when it is coded. Designing software is done on paper. When a design document is finished, the design document can be checked with a review. However, there are no support tools to check a design more formally. Additionally, it is impossible to know if the design is consistent and conforms to the requirements. The managers think that with ASD the organisation no longer need to spent resources on small or trivial software faults. With ASD, the organisation can focus and spent resources on essential system challenges.

During the current software development practice about 50 to 60 percent of the state behaviour is thought of by the designer and written down in a design document. The other half of the state behaviour is incrementally developed during engineering. In this phase, the software engineers need to make decisions for which the software engineer cannot see the implications on a larger scale then for the involved module the engineer is working on. It is difficult to predict what the impact and consequences of such decision are. Consequently, it is challenging to keep a consistent architecture. Today it is often the case that the behaviour of modules or subsystems is unintended or inconsistent with other parts of the system. All kinds of introduced race conditions and erroneous behaviour make that a lot of time is needed to find and restore the faults, especially with complex behaviour. ASD should solve this. ASD should help identify such issues in the design phase. One could say that with ASD, the behaviour that has been specified is the behaviour one gets. The generated source code is correct. Consequently, ASD is a method to manage complexity resulting in software with a better quality. As stated before, the design phases will take longer while the test & integration phase will take less time. The prediction is that during integration fewer faults will be found and also the validation on system level will reveal fewer faults which results in fewer Problem Reports (PRs).

Management thinks that in the future they need software designers that can think more abstractly and do not want to start coding immediately. This is in contrast with the current pragmatic approach many designers take. There will be a shift to more abstract and out-of-the-box thinking which should be challenged and supported by the organisation. When this is done, applying ASD is relatively easy. In the future the ASD tool will be used more frequently. All important control interfaces will be modelled by ASD. The future population of the software departments will change; there are more architects and designers needed and less engineers. However, this process will go gradually. For a very long period, if not forever, legacy or current code will be used which is not generated by ASD for data handling.

The managers have prior experience with changing processes but despite their experience they had expected it to be more smoothly. One of the managers said: “Changes in software development are difficult, but this is hard.” The different pilots and projects with ASD are a process of learning by doing. From its experience, management has seen that if you invest in an ASD team you see that it works. For a number of people this becomes a pleasant surprise. First they are sceptical but later when working with ASD they see that it works. Management is convinced that if they proceed and more and more groups are adopting ASD and share knowledge, then at a certain moment this will be enough to cross the critical mass. People will then think it is naturally, but there is a long way to go.

The expectations about ASD are based on pilot projects at, for example, BE. In small scale projects the managers have seen that the biggest advocate of ASD will become the project leader. When using ASD the relationships between the development phases shift. Requirement & global design takes more time, detailed design stays the same, and the test & integration takes less time.

Management has experienced that many software designers are unsure that the software development process becomes more effective with ASD. However, the software designers do think that there are fewer faults present in the source code. The managers have noticed that most designers expect little from ASD. The managers are unsure why they are unable to capture the big picture. Management has told designers the expectations they have about ASD on previous pilots, but management does not get the feedback from the designers that they agree with the benefits. The managers think that the designers need to get used to the idea.

6.1.4 Technology

There are some frequently used design concepts which are not (yet) supported by ASD. Despite the solved tool problems, management observes that it is very difficult to get started with the ASD tools. This is because the developers have to think in another paradigm to make a design. When designing a component in ASD the designer must have a complete idea about the component before the ASD tool can be used. The way that problems are made explicit changes radically. A software designer has to learn something new. The pilots learn that for most software designers this not a problem, but for some this is undoable. Additionally, the interface of the ASD tool could be improved.

6.1.5 Technology Transfer

The preparation to make the transition to ASD has gone stepwise. On the management level this has been done by the business architect. During the past three years management, together with Verum, has organized extra information days and courses. This includes refreshment courses because the tool has changed a lot during the past three years. Additionally, there have been user information days which are used to exchange experiences with other companies.

At the moment of writing this thesis, the organisation regarding the FEC team is in the preparation phase for making the transition to ASD. People are sent to the ASD course. At the same moment, there is an activity to build a new reference architecture for which ASD will be applied. Next, the parts of the reference architecture that will be made with ASD have to be chosen. Some parts of the architecture are suitable for using ASD, others are unsuitable. ASD is a means, not a goal. The new reference architecture is the goal and where it is useful ASD will be applied. The work on the reference architecture is done by architects. Although, the introduction of ASD for the designers and engineers has more consequences, also architects have to approach problems in a different way. FEC is a stable group, people work there relatively long, so ASD can be a major change for them.

Only a limited number of people are working on establishing the reference architecture. The communication of the introduction of ASD to the rest of the SW-FE people can be improved; they are not officially informed about ASD. Only a few talks have taken place. This is because, at this moment, management does not know the implications of ASD. According to management the communication can be improved by giving more talks about what ASD is and what its accomplishments are. These should not be management talks, but talks by software designers. Management thinks that this works better within the organisation and that peers are more open for each other and should create critical mass. The critical mass should then pull the others over the line. Management can inform the software designer about ASD's benefits but it would be better if peers make other peers enthusiastic. Therefore, there is a need for an ASD core team that propagates the benefits, and understands and addresses the difficulties. Furthermore, train people and give them assignments to work with ASD. Let them learn how it works and guide them when necessary. Preferably with more experienced people. People with more experience from prior projects can be assigned to new projects with people with less or zero ASD experience. Management has to think about the introduction for the large group. For example, with a key user group that makes a presentation and training material. Management wants to build knowledge about ASD. It is possible to hire Verum consultants for every new group that makes the transition but building knowledge about ASD this way costs a lot of money. So share the experiences there are within the organisation regarding ASD. Let other software designers show where it can be used for, how it works, and where it is infeasible.

Some managers expect that they will encounter difficulties with ASD during the first years of the transition. They think it is a large change in terms of the required mindset and the new practices. Every change brings a lot of resistance. This step is comparable with the transition from C to Object Oriented and therefore asks a very big change. People like to hold on to their old values. At first management has to talk and convince. Later people start working with it.

6.2 Software Designers

The software designers are split into two groups: global designers, and detailed designers. For the global design two software architects are interviewed and a system architect. For the detailed design, software designers which also have the software engineering role are interviewed.

This section contains three subsections; these are: common view, global designers, and detailed designers. In the common view section, the shared view of the detailed software designers and the global software designers is discussed.

6.2.1 Common View

6.2.1.1 Software Development Organisation

With the introduction of ASD, the software designers are forced to make complete specifications in an early stage of the software development process. This will result in a more efficient process because some faults can be found in an early stage and no longer only at the test and integration phase.

The software designers see how the business unit Interventional X-ray invests in ASD. So they think it is wise to go with this flow, and they think that there will be a bright future for this methodology within the business unit. They expect that in a few years everybody will apply ASD. So if someone cannot, or will not use, ASD, he or she has a problem.

6.2.1.2 Technology

The software designers expect that there is some learning curve before ASD can be applied successfully. During the first steps of applying ASD, the software development process will take longer because it is new. The design phase will take longer than before but this time will be gained back during the implementation and validation. When generating source code with ASD, the tip of the V-model disappears. There will always be acceptation tests because the customer needs the see what functionality is implemented.

All software designers think that the right method should be used to address a certain problem. In other words, use different methods for different problems. Hence, use ASD only for problems for which it is beneficial.

6.2.1.3 Technology Transfer

Most software designers have followed Verum's ASD course. The general perception is that the course is an introduction to the tool, but not to a method. Also, the timing between following the course and using the tool in practice is something to take into account. Both groups agree that this period should be at most one month because otherwise the momentum is gone.

6.2.2 Global Designers

6.2.2.1 Software Development Organisation

The architects know something about the architectural units up to a certain level of detail. They will decompose the system and assign the responsibilities for certain functions or features to units. The architects will not build ASD models themselves. The designer gets an assignment to build the detailed design of a unit, the architect tells in the assignment what the architect expects from the unit in terms of dynamic behaviour. When the designer has finished the interface description, the designer will present it and the architect has the ability to comment on it.

Today, when making a global design, the requirements are used to design an interface of a component on a certain level of abstraction. With ASD this is no longer possible because the interface needs to be complete. Therefore, the architect needs to approach an interface in a different way. The interface needs to be viewed from different angles to make it complete. This will take more time than currently and it is something people have to learn. ASD needs to be applied on interfaces which are control-based and not on interfaces which are data-centred. Also, the benefits of ASD are unclear for interfaces which are made once and which are not expected to be any change on the interface for the lifespan of the reference architecture. However, for interfaces for which it is expected that they will change frequently the benefits of ASD are clear. Also, the benefits of describing interfaces with the environment of architecture are clear. For example, some devices are made in other plants. Describing interfaces with the environment can help to abstract from the used components. Furthermore, it helps testing these components. What the suppliers of components do internally does not matter for the FEC as long as they adhere to the interface.

In the current global design practice the interfaces are not leading, the architectural units are. The feeling is that this will shift with the introduction of ASD. The architects think that they must shift their paradigm from decomposition to interfaces.

Every architectural unit, which can consist of multiple ASD components, has a requirements document and design document but not its own designer. The allocation of the units is done on the basis of experience, knowledge and availability. In the current architecture some units are stable for 5 years, but others have to be modified with every change. The same people have worked on these units, but currently there have been some shifts in the allocation to distribute the knowledge about these units among more persons. The organisation will always have some domain specialists and some people know a lot about the system because they work on FEC for a long time. ASD could make some units more accessible for others. The FEC group is a stable group; most people work on it for a long time and know the difficulties with the current architecture. With the new reference architecture it should be more understandable and accessible for newcomers. Some information is burden; therefore, there is always a balance between what should be documented and what not. The global design contains the requirements and the overall design concepts. In some way one has to make the translation from the overall concepts to the details of a unit.

6.2.2.2 Technology

Some of the architects have prior experience with formal methods in industry. These formal methods failed because the specifications became too formal; therefore, they could no longer be communicated with the customer and therefore could no longer be validated with the requirements. This was a major problem at the time but this should not be the case with ASD.

Currently, state machines are used but they are hidden in if-statements or case-statements. In the future the notion of state machines will increase and all designers will have a representation of a Sequence Based Specification (SBS) on their desks. The interface description will give insight in the essence of the unit. So the terminology will change. But on the conceptual level of abstraction the functionality or interaction of the system will stay unchanged. Therefore, the work of the architects will not change much.

Depending on how much of the reference architecture will be realized with ASD, legacy or current code will become an issue. The legacy code will, for example, be used to make calculations because in ASD it is impossible to do this. ASD generated source code and legacy code will become intertwined; ASD generated code will invoke functions of the legacy code. Normally, it will not be the case that a designer will only do a part of a unit, for example, the part with ASD. It is possible that some units are made without ASD. For the parts for which no code will be generated with ASD, there should still be an interface model made. How far ASD will be integrated into the architecture is currently unknown. It will depend on early experiences. Also, some parts are non-control-based and ASD is therefore unable to cope with it. There will always be large parts of the architecture solely consisting of handwritten source code.

Concluding, the architects do not think that ASD is the silver bullet that will solve all the problems. The architects think they should be open for the change. They have seen that with previous changes people raised a blockade. They want to try it at several places in the architecture and use this experience to apply ASD where it is beneficial and not apply ASD where this is not the case.

6.2.2.3 Technology Transfer

Persons with the architect role do not need to know all details; they can and need to use experiences from others. For them it is impossible to know everything in the field. Therefore, following the ASD course is not perceived as a necessity for this group. They need to know the ASD's concepts, but they do not have to use the tooling themselves. However, they need to be able to read a SBS either in tabular or graphical form in order to communicate with the software designers which make the detailed designs. It should be possible to read a SBS without following the ASD course.

6.2.3 Detailed Designers

6.2.3.1 Software Development Organisation

ASD is a tool. The ideas have to come from the interaction with peers. The development starts with an idea of what will be built and then a model of this idea is made. Currently, this is done with UML and in the future with ASD. The main shift when using ASD will be the reviewing of the model. With ASD, the reviewing will only address if the model is conform the requirements while today reviewing also addresses the consistency and correctness of the model.
Some designers think requirements engineering could be improved. They think it is hazy and hope that ASD will solve this because the interfaces need to be complete. Currently, interfaces are very generic, data-driven and stateless such that it is always possible to make tiny adjustments. This is one of the reasons test & integration takes relatively long, because the tiny adjustments could break interaction between architectural units. With ASD, the consequences of tiny adjustments of the interface can be checked automatically by means of the model checker. This is considered as a benefit of ASD.

Currently, the specifications are regarded to be brief. This is caused by postponing choices and decisions until the engineering phase which results into an unclear design document. With ASD the designer is forced to make a 100 percent clear and complete specification of the control behaviour. There will no longer be a discrepancy between an implementation and a design. The assurance of knowledge is considered the biggest plus of using ASD. The design phase will take longer but the lost time will be gained back during the test & integration phase. Test & integration is considered to be a major problem, especially for units which depend on lots of other units. Regression testing is used to see of changes in a unit does not break other units. If it does break, then it is a Problem Report, PR. These faults make the development process unpredictable because it is unknown what will be encountered. If ASD is used, changes on a unit will not break other units. Therefore, the process becomes more predictable and there will be far less PRs regarding test & integration. However, the software designers expect that there will always be a lot of PRs in the legacy code. Less PRs will be very welcome because PRs are experienced as very unpleasant to solve especially when the next project is already started. A large group of PRs can be prevented by using ASD between the interfaces of units. Of course, ASD will not avoid all PRs.

Whether the interaction with peers will change, depends on the portion of the architecture that will be done in ASD. When a large part of the architecture will be done in ASD, peers will no longer talk about the source code but about the ASD model. It will also depend on where peers are comfortable with; some will rather talk about source code and others about ASD models. Some source code will be generated by the ASD tool; sometimes the generated code might be more readable than the model itself. The design model is relatively abstract. There will be a separation between designers that can cope with the abstract model and designers that unable to do so. The introduction of ASD could improve the interaction between designers and testers because there are lots of testers that make models of specifications. There could be earlier discussions about testing. The tester can more quickly see where to focus on while testing because the tester will get a formal ASD description.

The software designers have no idea how to communicate an ASD specification (SBS) with their client. An SBS is not considered very readable. It is possible to make a graphical representation of it, but they do not know if this is better.

A fear among the designers is that the organisation becomes too ASD-centric in the sense that all problems can only be solved with ASD. They fear that this could limit their creativity to choose for certain solutions. However, creativity will also be determined by the possible solutions the ASD tool provides. A designer could think that an ASD solution is too difficult and therefore decides to create a design without ASD for a particular solution. If the designer presents the problem clearly enough, it may be possible that a workaround will be proposed. Eventually, the use of ASD is a decision of the CDG or software architect. When this proposed solution is not in the current tooling, Verum can maybe adapt the tool. The ASD tool is in development and it would be wise if Verum should listen to such change requests.

There will be resistance against the introduction of ASD. Resistance to the unknown, because it is different from the current practices. People in this organisation are used to work in a certain way for years and this will be a major change for them. The software designers think they should start working with ASD and learn from there. They also think that it should be used where it is beneficial, otherwise it should not be used.

Often the designers get a new tool from the organisation, and need to learn how it works themselves. For example, everybody uses the include structure of Rational Rose in a different way. Some years ago there was a group responsible for how a tool should be used within the organisation. Also, for example, how a C++ project file should look like. In those days, there was a lot of standardisation which is now gone. The designers would like to see standardisation of the use of ASD within the organisation.

6.2.3.2 Technology

The interfacing between the ASD generated code and the legacy or current code is a big question mark. It is impossible to rebuild a new architecture at once from scratch;

therefore, at some point it has to interface with the legacy code. To combine the activity of making a new architecture and introducing ASD is considered a good combination because a new architecture implies large software changes. Everything that is data-centred will be done with handwritten code, or maybe legacy code.

When using the model checker, the danger is that the software designer will hunt the “green ticks” which are the pass or fail of certain properties. The danger of satisfying the model checker is that the user solves problems that have nothing to do with functionality. The software designer needs to be careful because the design can suffer from it. The user of the tool is busy solving problems the tool finds but with solving the problems the user could forget the original requirements. For example, deadlocks can be checked when the user of the tool has a complete design; however, the deadlock can already be introduced in the interface. The first question should be: are the requirements for the interface correct? Then the user needs to iterate between the interface and design. If the user hunts for green ticks, the requirements are erroneous.

Using ASD, the designer needs to think in a component-based way; this implies a change. It does not matter if ASD looks like OO as long as it provides enough possibilities to do the things needed to solve a problem.

Some basic things in the tool do not work. Very simple copy and pasting does not work. It will help the satisfaction of the user using the tool if such basic things will be fixed.

6.2.3.3 Technology Transfer

Before a software designer can start, the software designer has to be prepared by an introduction to ASD. This can be done by following a course or reading some tutorial. This material is considered to be essential for the acceptance of the tool. Most ASD documentation is in the course material. However, it is possible that someone within the organization has to start using ASD without following the course. For such persons, a book describing ASD would be useful. Also, knowledge sharing within the organisation should be used for teams that start working with ASD. Currently, information is available on request basis.

The course is considered to be useful; however, there were some missing aspects. In the course examples are used and modified. However, building a model from scratch is not explained while this is perceived as very important. Some explanation about design patterns is missing. According to the detailed designers, it would be a plus if the course provide the theory behind the tool. The tool tries to shield the mathematical foundation on which it is build, but some knowledge about the mathematical foundation will be useful for understanding the output of the model checker. Without this knowledge the output of the model checker is very hard to interpret. The model checker is believed to be the hardest part of understanding ASD.

6.3 Observations

Besides interviews, the researcher also used participant observation to acquire data on the pilot project’s preparations. Here a description of the most important observations.

To interpret the interviews, one has to realise that the aim of the work on FEC has changed during this research project. First, the assignment was to develop a new reference architecture. Just before the research started, the assignment was changed to include the use of ASD for describing the interfaces between the different software units. Later, positive results became available of another pilot project. In this project, complete designs were made by ASD and code has been generated. Because of these positive results, the assignment for FEC has been changed to fully apply ASD wherever possible, so including design and code generation. Therefore, in the interviews described in the first two sections of this chapter the designers talk about applying ASD on interfaces. At that time it was unknown that ASD should also be applied on complete designs including code generation.

According to Verum consultants, ASD can be used for every control-based design. However, on several occasions problems with the application of ASD were encountered. Designers had made a design and wanted to apply ASD to verify the design. However, the tool did not support the intended design solution. The general idea within the organisation was that Verum should adapt the tool in order to make it possible to verify such designs. However, support for such designs does not fit into the overall concept of ASD. By developing design solutions that are not supported by the ASD tool, it is easy to point out the incapabilities of the tool.

Another observation is that at some point in time there were two teams. One team was responsible for building a reference architecture. The architecture was built in a way common to the members of the team. The second team was responsible for developing new ASD design concepts such that the developed architecture by the other team could be built by applying the extended version of ASD. Hence, a team was responsible for developing concepts that should extend the ASD tool in order to be able to build an architecture in the way that was common to build an architecture within FEC.

III Concluding Remarks

We summarize the results of the interviews and observations which answer the question: What is the ultimate goal of acquiring and using the technology?

The objective of introducing ASD is to improve software development. Currently, the test & integration phase, compared with the other phases, takes too long and makes the process difficult to manage. The quality of the software in products that leave the factory is unprecedented but according to some of the interviewed persons the reason for the long test & integration phase is the quality of the software which is supplied to this phase. An important part of the problem is that independently developed software units do not work together seamlessly.

The current approach to manage test & integration problems is to go quickly through the V-model. In this way there is sooner something that can be tested, which gives management the perception of control. By going fast through the V-model, the focus is on testing the quality into the software during the test & integration phase.

As a new approach, management puts an effort into improving the software quality by providing software designers with a new tool, namely ASD. The tool should be used to make designs, but the introduction of a new tool is not the objective. By improving the software quality, the test & integration phase should become shorter and thereby improve the whole software development process.

The ASD tool can be applied for designing control-based software. Of course, not all software is control-based. Additionally, not all software will be instantaneously made with ASD. Hence, there will be large portions of handwritten legacy code during migrating to the new reference architecture. However, the objective for improving the software development also applies for the non-control-based software.

Everybody that has been interviewed and has followed the ASD course agrees that the ASD course is an introduction to the ASD tool. During the course some prefabricated models are adapted, but none of the developers has an idea on how to start building a model. The required method to apply the tool is not addressed.

The perception of the software designers is that, a software architecture can be designed in the way architects and designers are used to make designs. Hence, the current practice does not need to be changed for applying ASD. When such a design has been made, units can be chosen to apply ASD.

In Part IV, we further analyse the data from Part I, Part II, and Part III. This results in a complete picture of the transition situation with all relevant aspects.

IV What are the steps to reach the desired goals given the state of the organisation?

Given the previous parts, describing the current state of the organisation, the nature of the technology, and the goal for using and acquiring the new technology, Part IV describes how the software development process can be improved. The improvement should lead to a reduction of the test & integration phase. Given the results of the previous parts and the literature we formulate recommendations regarding the transition to ASD.

7. Implications of Introducing ASD

In this chapter, we combine all the data of the previous chapters to describe the implications of the introduction of ASD into the organisation. In Section 7.1, we start with the desired goals by analysing the interviews and observations. From this analysis, we conclude that a method to apply ASD is missing and that a Cleanroom-like method might solve the method problem. Therefore, in Section 7.2, we explore how the combination of Cleanroom and ASD would look like. In Section 7.3, we further describe the implications of introducing ASD by analysing the interviews under the assumption Cleanroom/ASD. Then, we globally describe some important aspects regarding the transition from the current state of the organisation to a possible new state with Cleanroom/ASD in Section 7.4.

7.1 *Analyse Desired Goals*

During the past three years the organisation has experimented with ASD during several pilot projects. The objective is to improve the software development process by reducing the time necessary for test & integration. In Section 6.3 some observations are described on how the technology is applied during the research. An important observation is that some solutions that have been designed do not work with ASD. The tool is blamed that something is impossible and Verum is called to make a particular solution work with the tool. In Figure 46, we try to capture the current state of the organisation into a model about adopting a new technology.

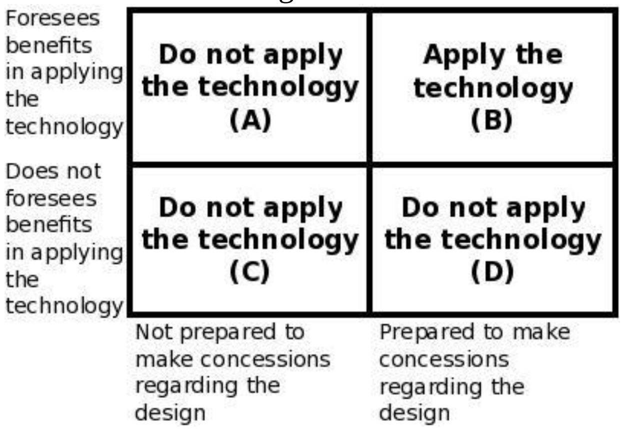

Foresees benefits in applying the technology
Do not apply the technology (A)
Apply the technology (B)
Does not foresees benefits in applying the technology
Do not apply the technology (C)
Do not apply the technology (D)
Not prepared to make concessions regarding the design
Prepared to make concessions regarding the design

Figure 46, Success Factor

Figure 46 describes an organisation's state regarding a new technology. The outcome of a decision process to introduce a new technology depends on the following:

B)	If an organisation foresees benefits in applying a new technology and is prepared to make concessions to be able to fully benefit from the technology, then it can successfully apply the new technology.
A,C,D)	Otherwise, do not apply the new technology.

We apply the model on the business unit Interventional X-ray and ASD. The managers and designers indicate in the interviews that both foresee the benefits ASD could bring. However, the organisation is not (yet) prepared to make concessions regarding the design. Therefore, during the research described here, the organisation was in state A.

We want to emphasize that the new technology is more than just a tool. Therefore, in the following, we make a distinction between the tool and a method that may be used to apply the tool. How an architectural decomposition should be made, in terms of method and process, is not addressed by ASD. When combining the insights of the interviews and observation, the method to organize the software development process, with or without ASD, is missing. Therefore, the first step to reach the objective of reducing the test & integration phase is to choose a method for applying ASD.

From the interviews, we have retrieved the following technical constraints for the required method. The method:

1. should work with ASD, which is:
    - control-based, and
    - component-based;
2. should work with legacy or current code, which could be:
    - control-based, and/or
    - non-control-based;
3. should work with new non-control-based software, which could be:
    - data-centred, such as algorithms; and,
4. should improve test & integration.

The published literature about ASD [2][3], as mentioned in Chapter 4, describes that ASD is a combination of CSP and Cleanroom. Additionally, Cleanroom satisfies the technical constraints for the required method as described above. Therefore, as a hypothesis, in this thesis, the use of Cleanroom is investigated such that an integral approach for the software development process with Cleanroom is possible. We will call the combination Cleanroom/ASD.

The business unit was during the research in the process of learning the technology. Because the lack of knowledge about the method that should be applied, the solutions that the designers develop for problems do not fit with ASD. The gap in knowledge about the method should first be filled before sustainable solutions can be developed. For a successful transition, the organisation needs to realize that it cannot continue to design systems with the current practices. It makes no sense to apply ASD to an architecture that is not developed with a component-based methodology like Cleanroom. The designers, as the managers correctly noted, need to undergo a paradigm change. The gap between the current practices and, for example, Cleanroom/ASD makes the transition nontrivial. To successfully adapt ASD, the current state of the organisation should shift from state A to state B. If the decision is made that the organisation is not prepared to make concessions on the design, then the transition will fail on the long-term. The next section describes how the combination Cleanroom/ASD could look like.

7.2 Cleanroom/ASD

As described in Chapter 5, Cleanroom describes a software engineering method and process. In this section, we describe how the Cleanroom/ASD combination could look like an software development organisation.

7.2.1 The Method

In this subsection, we describe how the software development organisation with ASD combined with a Cleanroom-like method could look like. We use the technical constraints of the previous section as a guideline.

1. The method should work with ASD.

Cleanroom, as can be read in Chapter 5, describes a method on how an architecture should be decomposed into components. ASD uses this principle from Cleanroom in order to scale ASD to industrial sized applications. Therefore, a component-based method, like Cleanroom, is required to create an architecture suitable for applying ASD. Architectural components that solely contain control-based behaviour can then be made with ASD. The behaviour described by the ASD components will be used to automatically generate code.

2. The method should work with legacy or current code.

Because the size of the code base, it cannot be altered all in a limited amount of time. Therefore, during migration the legacy or current code should be used in combination with the new reference architecture and the new technology. With a Cleanroom-like method it is possible to isolate legacy or current code into separate components. In ASD terminology, as described in Chapter 4, the legacy or current code can be placed in foreign components. For foreign components, ASD interface models can be built to describe the control-based behaviour. This behaviour can tested (not certified) by the Compliance Test Framework (CTF).

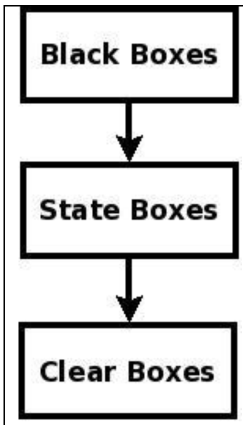
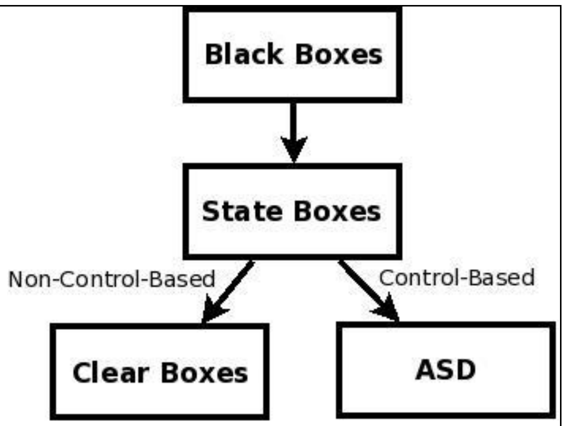
3. The method should work with new non-control-based software.

We define non-control-based components as components that not solely contain control-based behaviour. Using new non-control-based software is similar to the previous technical constraint. But instead of using it for legacy or current code, it can be used for data-centred software. Data-centred and algorithmic software is isolated and placed into foreign components. These components will be implemented manually by handwritten source code. When using Cleanroom, the handwritten components are developed with error prevention in mind.

The implemented interface ASD model for a non-control-based component is needed to model-check a component that uses the non-control-based component as a used component. Therefore, the limited control-based behaviour the component does have should, according to Cleanroom, be statistically tested. An ASD interface model for the control-behaviour can be built and, possibly, the Compliance Test Framework (CTF) could be used to certify the handwritten parts.

4. The method should improve test & integration.

The method should improve test & integration for the previous three technical constraints. In order to improve test & integration, we propose to use ASD combined with a Cleanroom-like method as an integral software development approach. When a Cleanroom-like method is used, the last step of the design will be done with the ASD tool for control-based software, or an informal or formal proof is made for the data-centred and algorithmic software. In Figure 47, the development with Cleanroom is depicted while in Figure 48, the development with the combination of ASD and a Cleanroom-like method is depicted.

 **Figure 47, Design with Cleanroom**	 Black Boxes State Boxes Non-Control-Based Control-Based Clear Boxes ASD **Figure 48, Design with Cleanroom/ASD**

All components are created with quality in mind and therefore test & integration will be improved. With Cleanroom/ASD, the test & integration phase will no longer be used to test the quality into the software, but to certify the quality of the software.

7.2.2 The Process

As described in Chapter 3, the business unit has a generic process model which is used for all development disciplines. This process model is essentially the Waterfall model. For the software development organisation the V-model is fit into the generic process model.

In Chapter 5 the process model of Cleanroom is described. The Cleanroom process describes that the activities design and test case generation occur concurrently. The organisation adopting Cleanroom can choose to execute these activities concurrently or sequential. For sequential execution, we propose to adapt the Cleanroom process model to fit into the generic process model. In the thesis, we have used the following three phases for software development: global design, software design, and test & integration. We map the Cleanroom activities onto the generic phases. Cleanroom's specification activity can be mapped onto the global design phase. The design activity onto the detailed design phase, and the activities test case generation and statistical testing can be mapped onto the test & integration phase. This exactly corresponds to the three teams required for the Cleanroom process. Figure 49 depicts how this mapping corresponds to the box structure.

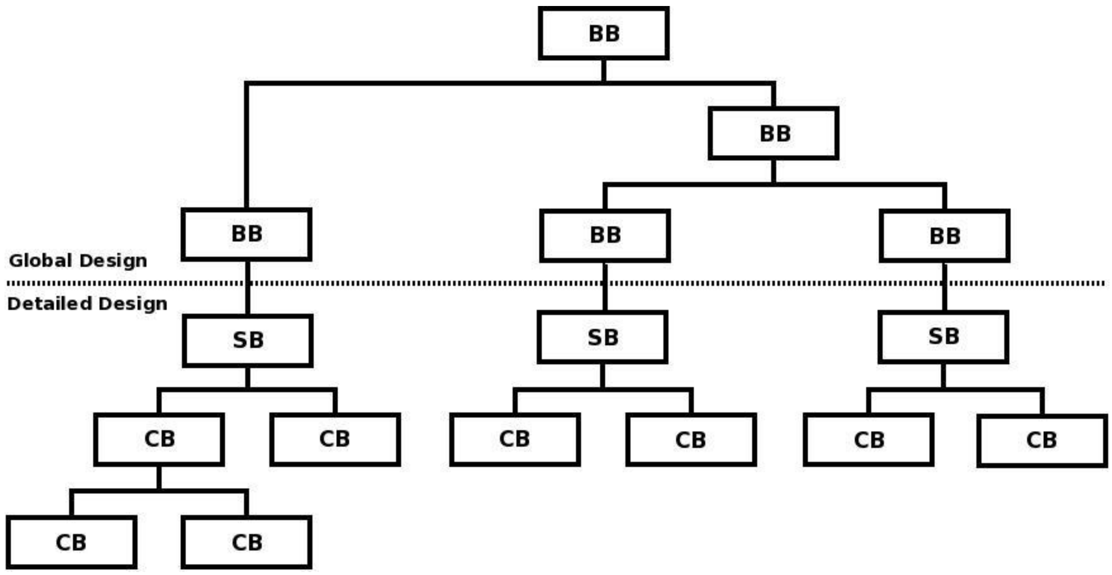


Figure 49, Box Structure

In the remainder of the chapter, we refer to Cleanroom's method when describing about Cleanroom.

7.3 Analysis of the Expectations

In the following two subsections, we continue to analyse the interviews under the assumption of using Cleanroom/ASD.

7.3.1 Managers

Managers indicated that it is currently very difficult to keep the design document and source code consistent. We think that with the introduction of Cleanroom/ASD there still is a need for design documents especially when large and complex models are built. However, if a system is decomposed sufficiently with the Cleanroom method, then models should not be too complex. For the global design there will always be a need of a document. In Cleanroom this should describe the Black Boxes (BB) and the refinements of black boxes.

The assignment by management was to make a new reference architecture and then choose which parts could be done with ASD. We propose to make future reference architectures with a Cleanroom-like method and then apply ASD on components which solely describe control-based behaviour.

7.3.2 Designers

Remember that when the designers where interviewed they thought that ASD should only be used for interfaces. The global designers indicated that the benefits of using ASD for interfaces that frequently change are clear. However, for interfaces that are made once and for which changes are not expected the benefits are unclear. We think that there are still benefits because it is unknown in advance if an interface will change. The detailed designers indicated that with regression testing an interface between two components can be stable but when one of the components changes there could be new problems found during test & integration. This should be solved by

ASD. The major benefit for the designers when applying ASD is therefore that they have to solve less Problem Reports (PR) during the test & integration phase.

In the current global design practice, the interfaces are not leading, the architectural units are. The feeling is that this will shift with the introduction of ASD. The architects say that they must shift their paradigm from decomposition to interfaces. We think that the paradigm needs to change but not in this way. With a Cleanroom-like method, an architecture is decomposed differently. The Cleanroom decomposition, however, is less ad hoc and formally verifiable.

According to the designers depending on how much of the architecture will be realized with ASD legacy or current code will become an issue. This will become an issue because at some level in the architecture, ASD has to interface with legacy or current code. This is addressed by the proof-of-concept of Chapter 10 about the migration of an application. Additionally, legacy or current code can be used in a Cleanroom-like architecture as is described in the previous section.

The detailed designers fear that the organisation becomes too ASD-centric in the sense that all problems can only be solved with ASD. We already indicated that ASD can only be applied successfully if a Cleanroom-like method is used to create a solution in terms of an architecture. Because solutions have to be obtained with a different method, it is possible to perceive this as an ASD-centric organisation. Being ASD-centric has benefits which the detailed designers also subscribe. However, they also think that ASD will limit their creativity. It is true that elegant certain solutions are impossible with ASD because when using a Cleanroom-like method creating an architecture is more or less a stepwise undertaking, as is explained in Section 5.2. The idea is that the structure that is created can be verified relatively easy.

In summary, some ASD related challenges that are raised by the managers and designers during the interviews, and observations can be solved by applying a suitable method. We have reasoned that Cleanroom might be a suitable candidate.

7.4 The Transition

In this section, we explore the transition from the current situation to Cleanroom/ASD.

During the interviews managers indicated that they thought that the change process to ASD would be easier. We start this section with an exploration about the magnitude of the change. In Figure 50, [8] describes a model that, given the magnitude of technological change, more or less learning is required. Depending on the learning that is required for the technological change, the approximated time needed for an organisation to adjust is short or long.

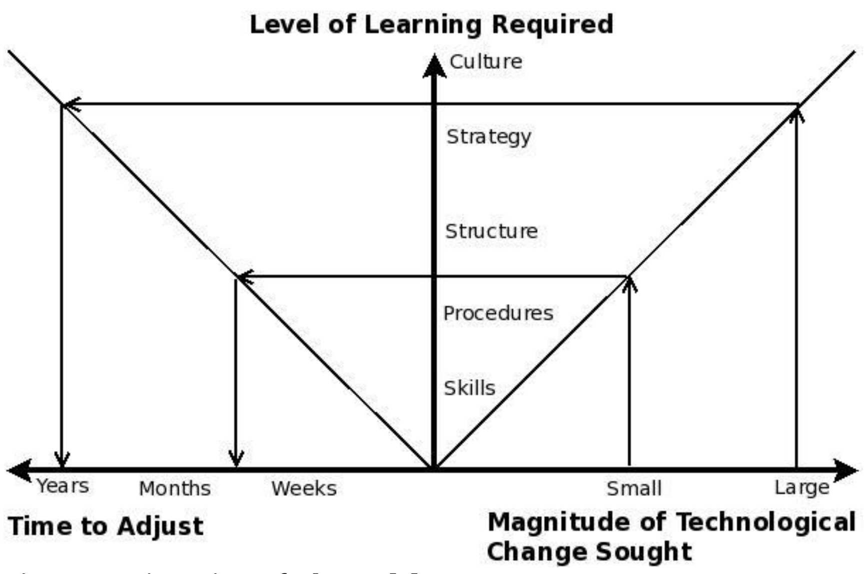


Figure 50, Dimensions of Change [8]

For the adaptation of Cleanroom/ASD within the organisation designers should acquire new skills; they should be able to use the new ASD tool. The procedures will change, for example, because with ASD it is possible to generate code. The structure of the software development organization will need to change because of the new Cleanroom-like method that is required to design solutions. The change is strategic because the strategy changes from trying to improve test & integration time by going through the V-model as quickly as possible to improving test & integration time by applying Cleanroom/ASD. Moreover, the adaptation of ASD imposes a cultural change. The new culture should be to make software that is the first time right. A shift in culture: from error removal to error prevention. Consequently, the model in Figure 50 approximates that the transition to Cleanroom/ASD will take years to accomplish.

7.4.1 Literature Model

In the past there have been software development organisations that have made a transition to Cleanroom. However, these transitions had a different starting point. The organisation that made the transition to Cleanroom in the past came from a used Structured Analysis/Structured Design (SA/SD) as a method.

Because ASD and Cleanroom are closely related, the change of the software development organisation to one of them will have comparable implications. In order to say something about the current transition and learn something from past transitions the model shown in Figure 51 has been built.

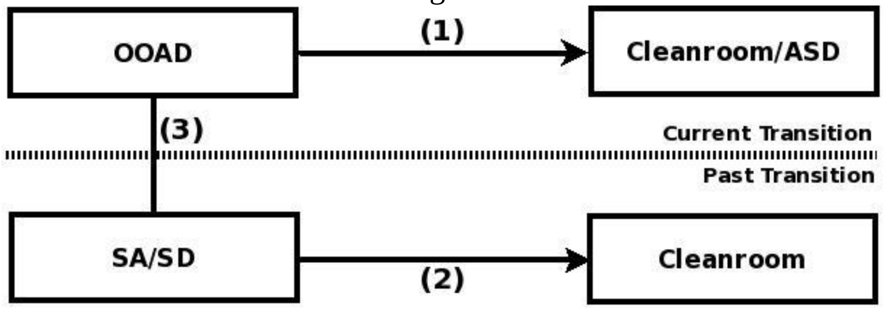


Figure 51, Literature Model

The model in Figure 51 is used as follows:

1. The objective is to say something about the transition from an Object Oriented Analysis and Design (OOAD) method to ASD/Cleanroom's method.
2. In order to say something about (1), the literature that describes the transition from a SA/SD method to Cleanroom's method is used.
3. For (2), the literature that describes the difference between SA/SD and OOAD is used.

In the following three subsections, we first describe the (3). Secondly, we describe (2) and, lastly, we describe (1).

7.4.2 SA/SD vs. OOAD

The transition from Structured Analysis (SA) and Structured Design (SD) to Object Oriented Analysis and Design (OOAD) is written in literature.

The main difference between the two methods is their objective. SA/SD works towards a design that can be used for a structured programming language and OOAD aims at a design that can be developed with an object oriented programming language. The main difference between them is that SA/SD considers processes as the main driver of system decomposition while OOAD considers data as the main driver of system decomposition. SA/SD makes a distinction between data, control and processes. However, OOAD tries to encapsulate data, control and processes into objects. [16][19]

According to [6] the transition from SA/SD to OOAD will take at least one year. The transition time does not depend on the project size, the size of the team, or the used tools and techniques. The transition time takes at least a year because the team members need to change their paradigm in developing software in another way. Every step of the software development process needs to be changed in order to infuse OO in the software development organisation.

7.4.3 Transition from SA/SD to Cleanroom

The past transition from SA/SD to Cleanroom is described in this subsection. In Table 1 a description by [10] of the differences between the traditional approach and Cleanroom's approach for software development is shown.

From	To
Informal specification	Black box specification
Informal design	Box structure refinement
Defect correction	Defect prevention
Individual unit testing	Team correctness verification
Path-based inspection	Function-based verification
Coverage testing	Statistical usage testing
Indeterminate reliability	Certified reliability

Table 1, From Traditional Software Development to Cleanroom

Of course, SA/SD and Cleanroom also have some similarities because both use decomposition to analyze a system. The techniques from both methods are comparable, for example, context diagram versus black box, state transition diagram versus state box, and data flow diagram versus clear box. [16]
In the literature, the transition from SA/SD to Cleanroom is exhaustively described by several case-studies at, for example, HP, IBM and U.S. Army, Texas Instruments (TI), and NASA. We list the aspects described [21][22][23][29]:

- Duration, how long it takes to adopt Cleanroom;
- Education, how engineers are prepared to use Cleanroom;
- Process support, for example books which describe the process;
- Creativity, how the method limits the creativity of engineers;
- Diffusion, how Cleanroom is spread among different project groups; and
- Phased Approach, how Cleanroom can be adopted in three phases. [10]

In the remainder of this section, we describe the aspects duration and education. Duration and education a chosen, because these provide a good insight on the technological change sought.

7.4.3.1 Duration

The research at NASA describes an understanding stage of 26 months. Almost half of this time is spent on training. The other half is spent on the first pilot project. Before the end of the first pilot project, additional pilot projects were started and by this they started with the transition stage. The transition stage took a little bit more than 46 months. The project at the US Army ran from the end of 1992 till 1994.

7.4.3.2 Education

Education is a required activity that is used to support engineers that make the transition. Before transition engineers are trained into the new technology and during the transition coaching help them to adapt to the new technology. Therefore, in this subsection training and coaching are described.

7.4.3.2.1 Training

The pilot project at the US army used formal classroom training together with workshops to educate the engineers. During the classroom training three topics were addressed: specification, development, and certification. The education also involved training about the support tools.

IBM's technology transfer program used four courses to transfer Cleanroom knowledge:

- Cleanroom Software Engineering for Zero Defect Software,
- Cleanroom Software Specification,
- Cleanroom Software Verification, and
- Cleanroom Software Quality Certification.

The first course was a one-day course; the other courses took three days. So, in total the duration of the courses was two weeks. They preferred not to give the courses full-time but spread the courses over, for example, four weeks in order for the engineers to digest the material. At TI they experienced that it is beneficial to train the engineers just before the need to apply Cleanroom.

7.4.3.2.2 Coaching

At the US Army coaching was used to help with planning and executing the project. Educating the engineers continued throughout the project execution. The experts visited the team on-side and helped to apply Cleanroom on their specific domain. The aim of the training and coaching was to adapt the engineers' mindset which was necessary for implementing Cleanroom. The paradigm needed to change to making software the first time right. This aim was reinforced by the involvement by of management.

The IBM Cleanroom technology transfer program appointed a consultant to a Cleanroom project. All documentation, at each stage of that documentation, the team produced was send to the consultant for feedback. The feedback and coaching of the consultant only addressed Cleanroom, not the technical aspects.

At TI the team that made the transition to Cleanroom made extensive use of an experienced Cleanroom consultant. The use of a consultant complemented the training. The consultant was also used for mentoring.

7.4.4 Transition OOAD to Cleanroom/ASD

In this subsection, we explore the current transition from OOAD to Cleanroom/ASD.

7.4.4.1 Duration

Previously, the transition from Structured Analysis (SA) and Structured Design (SD) to Object Oriented Analysis and Design (OOAD) is described. The hypothesis is that the transition from OOAD to SA/SD is the other way around and should therefore be comparable in terms of the time needed for the organisation to adjust. Hence, the time to adjust will take at least a year regardless of the project size, the size of the team, or the used tools and techniques.

From literature, in the past software developers were used to use SA/SD which has a small gap with Cleanroom because structured programming constructs are used by Cleanroom, as the explanation of clear boxes in Chapter 5 describes. The current transition imposes a larger gap; the transition from SA/SD to Cleanroom is smaller than the current transition from OOAD to Cleanroom/ASD. [1]

The model in Figure 50 and the case studies described in this subsection provides evidence that the transition to Cleanroom/ASD will take years to accomplish.

7.4.4.2 Education

7.4.4.2.1 Training

Literature describes that many courses and books were used in the past and that much training is required for a team to acquire Cleanroom. Additionally, in the interviews the designers indicated that the course they followed did not addressed a method to use the tool. Also, as described before, the current transition is more difficult than past transitions. Therefore, the organisation could gain in training the engineers that need to use ASD. Because ASD is a tool that supports the Cleanroom-like method, the focus of the first course should be about a Cleanroom-like method. An additional, second, course could train the engineers in the use of ASD, in combination with Cleanroom. Applying the tool should not be too difficult when trained for a Cleanroom-like method.

The global designers indicate that they do not think it is essential for them to follow an ASD course. We agree; however, a course about the method is a must. The detailed designers should follow both courses.

7.4.4.2.2 Coaching

Coaching can be a good supplement for education. As in the past, a coach can educate and provide feedback during project execution. An important aspect of coaching should be to apply a method correctly without going into the technical details of the domain.

Management wants to limit the use of ASD consultants and build the knowledge about ASD within the organisation. It sounds reasonable to assign people with ASD experience to new ASD project in order to transfer knowledge; however, these people are not consultants or coaches or facilitators. Nor do they master a much needed method to successfully apply ASD. Therefore, we propose to use (external) consultants which focus on correctly applying a Cleanroom-like method, without going into the domain.

IV Concluding Remarks

In this part, we sought answers to the question: What are the steps to reach the desired goals given the state of the organisation?

In our analysis, we noticed that currently ASD is positioned as a tool and therefore requires only changes in the skills of the persons who need to apply it. However, we also noticed that making a design following the current practices did result in designs that could not be checked by the ASD tool. This observation, combined with the fact that the ASD course did not explain how to create a new design with ASD, implies that a method to apply the ASD tool is missing. From Part II we know that ASD and Cleanroom are related technologies. Cleanroom Software Engineering describes a complete software development process and method. Consequently, we propose to introduce a Cleanroom-like method and call the combination Cleanroom/ASD.

Cleanroom’s process model is adapted to fit into the generic process model of the business unit. The process model of Cleanroom can be rewritten to the currently used phases. The Cleanroom method describes the steps to obtain a global and detailed design. In the final step of the method, ASD can be applied for control-based software. For non-control-based software a correctness proof could be made by hand. Of course, this will only be necessary for a fraction of the design. The benefit of using Cleanroom is that there is one integral approach to design high quality software.

Applying ASD will only become a success if the organisation is prepared to make concessions on the design. This is the most important hurdle that needs to be taken. Applying ASD is not just a change of skills for the persons that need to apply it, but rather a cultural change. Literature suggests that the change in mindset that is required may take at least a year to accomplish.

In summary, the first step is to acknowledge that a method is missing. The second step is to choose a candidate method. The third step is to adapt the method to the current situation. The last step is to infuse the technology, including the method, into the organisation.

Finally, we have described some guidelines for creating a transition plan. Creating an operational plan would be the final step with the insights given by part IV.

V How can re-use and migration enhance the technology?

Part II describes the new technology. From this part, we know that ASD is a relatively new tool which is under constant development and improvement. Among other aspects, Part III provides two aspects which are of concern for introducing the new technology to the organisation. These two aspects are:

- How can re-use enhance the technology? For instance, what can be done to avoid copy/pasting?
- How to migrate from the current situation to the new technology?

8. Re-Use

All software development methodologies emphasise the importance of re-use. ASD has some implicit mechanisms that addresses re-use which are described in this chapter, but we also propose a few new mechanisms to enhance re-use in the context of ASD.

The need for re-use can be encountered at different stages in the lifecycle of a software product. In the following, re-use during the design and the maintenance stages is distinguished.

During design the following aspects of re-use apply:

- common behaviour:
  - extending behaviour, and
  - merging partial behaviour; and,
- avoid copy/pasting.

Common behaviour is behaviour that will be used at several places. If the developer encounters common behaviour, the developer wants to describe it at a single place and re-use the behaviour from then on to avoid copy/pasting. Moreover, it is possible that for building an architectural unit, common behaviour described at different places need to be composed. This is called merging of partial behaviour. Separation of concern by splitting behaviour in a set of partial behaviours can help a designer to manage complexity. With incremental development, architectural units can be extended in each increment by new partial behaviour until the unit is complete.

When implementing a design, the developer does not want to redo previous work. Additionally, the developer does not want to copy/paste previous work into new work to avoid maintainability problems. If, for example, algorithms or requirements change, then at all places where that behaviour is implemented these changes have to be made by hand.

During the maintenance stage the following aspects of re-use apply:

- reallocation of responsibilities, and
- extending behaviour.

During the lifecycle of a software architecture it is possible that the responsibilities of a unit, represented by an interface or design model, change. It would be convenient if the current interface or design model is composed of other elementary interface or design models because when the responsibilities of a unit change, a new interface or design model can be recreated with minimal effort. Technically, extending behaviour is similar to merging partial behaviour explained above but then applied in the maintenance stage of the software product's lifecycle.

The ASD tool has incorporated some re-use aspects described above. These re-use aspects involve the re-use of complete interface and design models. This is explained in Section 8.1. To improve the tooling environment some additional techniques are described in Section 8.2. These techniques involve the re-use of partial interface and design models.

8.1 ASD Solutions

For some of the re-use aspects described above it is necessary to re-use complete interface and design models. The following solutions are described in this section:

- re-use interfaces for different designs;
- multiple instances of a component; and,
- interface refinement.

8.1.1 Re-Use Interfaces

Consider the case where one wants to use one interface model for two different design models. For instance, when two different devices that implements the same interface, such as the interface models used to drive the devices of the camera in Chapter 4. The interfaces of IAperture and IFilmSpeed are conceptually the same. They both have a Default state and the Fast state from the one can be transformed into the Large state of the other. This is also the case for the Slow state and Small state. Hence, the design model of these devices can make use of the same interface model. The camera design model then has to use two instances of the same interface to drive the two devices.

8.1.2 Multiple Instances

The second ASD solution regarding re-use is about multiple instances. For example, if a developer wants to drive two similar devices, then the developer can create a single interface model and design model, and instantiate the interface model twice; once for every device. The benefit of this possibility is that one interface and design model can be reused for multiple instances instead of making for each instance a separate interface and design model.

8.1.3 Interface Refinement

The example described in the previous subsection can be applied if the interfaces of both devices are equal. Equal interfaces can be grouped by ASD such that a mechanism can be used to invoke the same stimulus on all group members at once. However, if the interfaces of both devices are unequal, then grouping cannot be used. A solution for this is depicted in Figure 52.

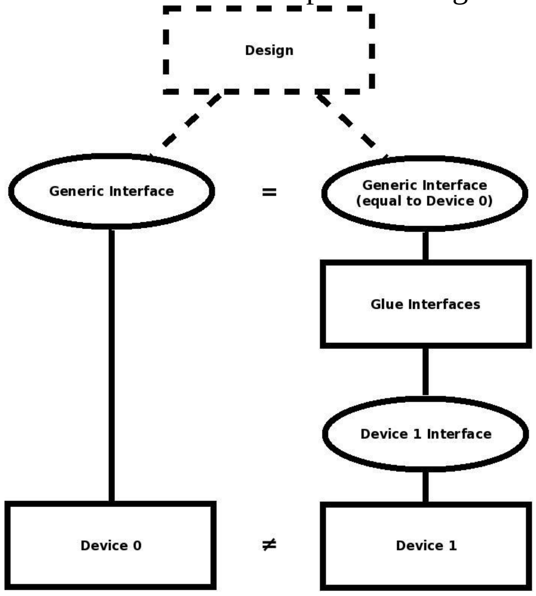


Figure 52, Interface Refinement

This solution uses an additional design model to glue the desired interface with the interface of the devices in order to be able to use the interface grouping mechanism.

8.2 Proposed new ASD Solutions

The solutions in the previous section focus on the re-use of complete interface models and design models. In this section the focus is on creating interface models or design models by re-using parts of the behaviour of other interface or design models. Here "behaviour" is not limited to states, but also includes the stimuli, responses, and rules for the corresponding states. The technologies described here make it possible to compose an interface model or a design model from other interface or design models which describe partial behaviour.

In the next subsections we discuss three techniques:

- template models;
- merge template models; and,
- expand template models.

8.2.1 Template Models

If a developer has to build several units which have common behaviour, then it is convenient to have a template model. This model template can then be extended with the specific behaviour such that the developer does not have to implement the common behaviour into a model for every unit separately.

Therefore, a library of template models can be used when building a system. If a developer wishes to build a unit, then the developer takes the unit template model from the library and starts adding extra functionality, particular for that unit, to the model.

Template models describe the common behaviour between interface or design models. Drawbacks of this solution are:

- poor maintainability; and
- difficult to combine two template models.

With respect to maintainability, it is possible that in the future, for example, by new requirements, a template model needs to be altered. A consequence of this is that every unit that has been build with the template model needs to be changed manually.

Currently, the use of template models is not supported by the ASD tool. In Chapter 9, a solution is presented to allow the use of template models.

In addition, it is not possible to build a model that needs behaviour of two or more template models. These two drawbacks could be solved by merging models, as described in Subsection 8.2.2.

8.2.2 Merge Template Models

Merging template models means that the partial behaviour of a certain template model A is merged into a model B to form a new model C. When working with template models, the following actions could be possible:

- Creation. Take a model A and a model B. Copy the behaviour of model A into the model B, this leads to model C
- Removal. Given model C, remove the behaviour of model A. The result is model B.
- Altering. There are two options:
  - Altering can be accomplished by first removing the old behaviour by the removal action described above. When the old behaviour is removed, the new behaviour can be added by the creation action.
  - A similar result can be obtained by merging new versions of model A and model B to get model C.

The altering action is needed for the maintainability of derived models. For instance, if model A changes, then all the derived models should be easily modified too.

In a software architecture, there can be all kinds of relations between units and their common behaviour. As a concrete example of creation with template models, assume that unit 1 can has some common behaviour with unit 2, and unit 1 has some common behaviour with unit 3. The common behaviour between unit 1 and unit 2 can be put in a template, for example, in Figure 53. Similarly, the common behaviour between unit 1 and unit 3 can be put into a template, for example, in Figure 54. When we start building unit 1, the model A and model B should be merged, see Figure 55.

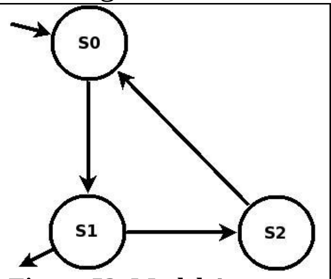


Figure 53, Model A

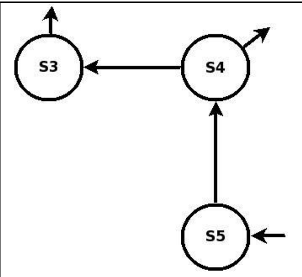


Figure 54, Model B

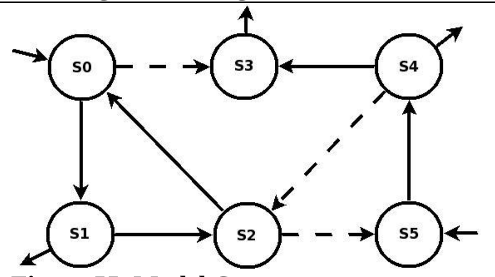


Figure 55, Model C

When the behaviour in terms of states are added, the derived model essentially consists of two separate state machines, see for example in Figure 55 without the dashed-lines. To obtain the combined behaviour, there are three ways to interconnect the two state machines:

- by hand;
- in an automated way; and,
- by merging states.

For all three solutions, it is possible that for different compositions of the models the connections differ. Connecting the two state machines by hand has the benefit that the developer has to think about the combined behaviour. The other two could be done automatically. The automated way is to somehow tell the merge mechanism or tool how to make the dashed-lined transitions. This can be done in a separate file, command-line or a popup window. The automated way would help the developer from redoing the same action multiple times. In [7][26] there are algorithms described that can be used to automatically merge partial behaviour. Merging states means that both model A and model B have some states with the same name. When merging model A and model B into model C the transitions of the states which have the same name in model A and model B, are merged.

To obtain combined behaviour, two state machines are interconnected. If one of the state machines is removed from the model, then the interconnect transitions, depicted by the dashed-lines of Figure 55, are unconnected on one side. Therefore, the previously described removal action should also delete the transitions that interconnect the two state machines. Although the example above uses two models, the number of models to be merged is not limited by the concept.

The merging functionality is currently not possible with the ASD tool, but a technical solution to include it is presented in Chapter 9.

8.2.3 Expand Template Models

Inspired by the macro-like operation described by [14][15], we describe an approach for expanding template models. Expanding template models involves the following actions:

- Creation. Make a model A with a “super” state. Replace the super state of model A by the states of model B, this leads to model C.
- Removal. Remove from model C the states of model B and replace these by a super state.
- Altering. Altering can be accomplished by first removing the old states by the removal action described above. When the old states are removed, the new states can be created by the creation action.

An example of the expanding template models solution is depicted in Figure 57 and Figure 59. Special state S2 of Figure 56 is expanded by the states of Figure 58; the dashed-lined state is replaced by multiple states from another model. Expanding models have the same benefits as merging models. There are two possibilities for applying the expand template models mechanism:

1. Figure 56 is the generic component and Figure 58 is the specific functionality for a unit. Hence, Figure 56 can be used for several applications.
2. Figure 58 is generic and Figure 56 has specific functionality for a unit. Here Figure 58 is used for multiple applications.

The states of Figure 58 replace the dashed-lined state of Figure 56 to form a new Figure 57. The three ways to connect the dashed-lined transitions of merging template models also apply for expanding template models. However, it is possible that transition labels for transitions to or from multiple states are the same. Therefore, there are several possibilities to connect the dangling transitions that leaving state S3 and S4. These two possibilities are depicted by Figure 58 and Figure 59.

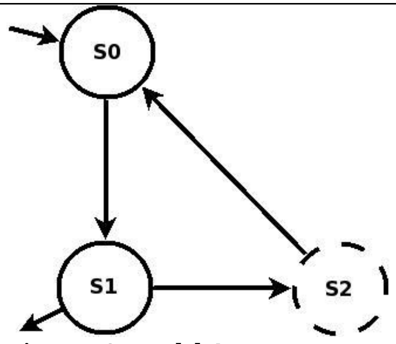


Figure 56, Model A

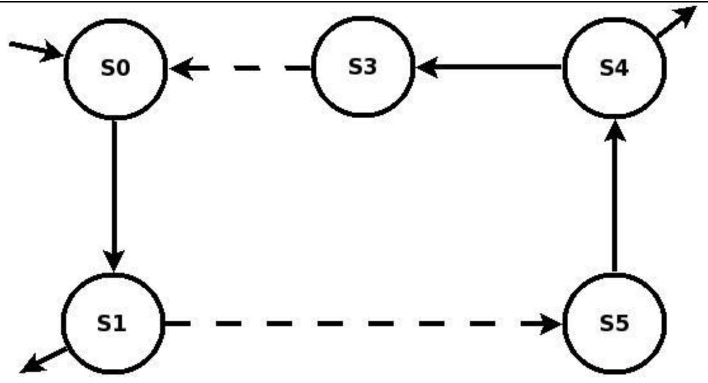


Figure 57, Model C

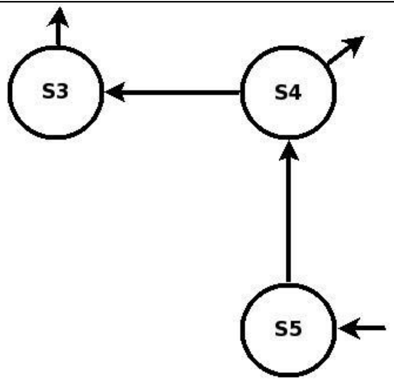


Figure 58, Model B

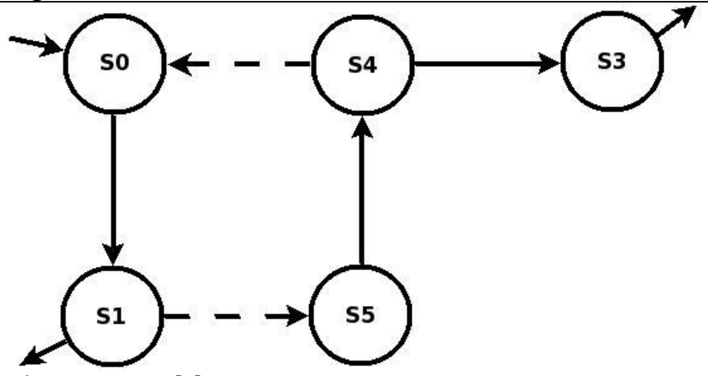


Figure 59, Model D

The expansion can be applied in two ways:

- Expand immediately. The states will be expanded, and are visible and accessible through the ASD tool.
- Expand just-in-time. This would involve a macro kind of operation that expands the states just before model checking, code generation, or test generation.

In the next chapter, we describe a script that implements the template models and the merge of template models.

9. Adapt Interface Models

As a proof-of-concept, we wrote a script that implements the usage of template models and the merge of template models, from the previous chapter, for ASD interface models. The first section of this chapter describes an approach on how to effectively apply the script on an architecture with the objective to maximize the possible re-use of ASD interface models. The second section describes the functionality of the script. The third section illustrates the functionality of the script with examples.

9.1 Re-use Approach on an Architecture

In this section, we present an approach to maximize re-use of ASD interface models. The approach consists of the following steps:

- step 1, decompose the architecture;
- step 2, identify common behaviour; and,
- step 3, compose the interfaces.

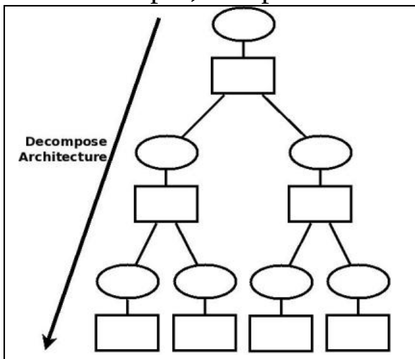


Figure 60, Step 1

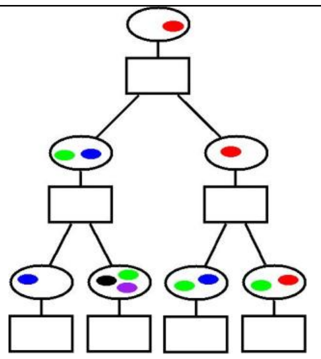


Figure 61, Step 2

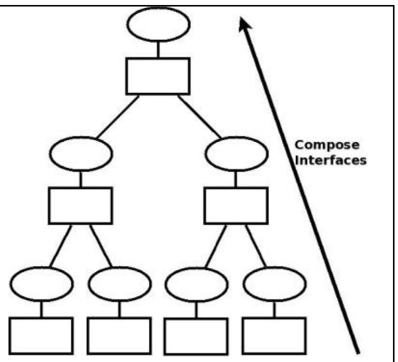


Figure 62, Step 3

9.1.1 Decompose Architecture

Chapter 5 explains that an architecture in Cleanroom/ASD should be decomposed top-down. This is done in the first step of the approach.

9.1.2 Identify Common Behaviour

The second step is to identify common behaviour among interfaces. In an architecture, typically some partial behaviour will occur in several interfaces. As mentioned before, it would be beneficial to re-use such pieces of partial behaviour by merging models. For example, in Figure 61 take the partial behaviour presented by the blue ellipses. The blue ellipse occurs twice in the same layer. The left interface of the middle layer can be composed by merging the blue and green partial behaviours.

9.1.3 Compose Interfaces

As a third step, we propose to implement architectures in ASD bottom-up layer-by-layer to maximize the possibility for re-use.

9.2 Script

The proof-of-concept is a Perl script that can be used to maximize the re-use of interface models. The script supports the usage of template models and the merging of template models as described in Chapter 8. The usage of the script is restricted to ASD interface models, ASD design models cannot be used by the script.

The script takes as an argument two files ASD interface models:
- an *extendfile*, which gets extended with new behaviour, and
- an *inputfile*, from which the behaviour serves as input to the process.

Resulting, the script can produce two files:
- an *outputfile*, which is the union of the extendfile and the inputfile, and
- a *difffile*, which is the difference between extendfile and the inputfile. The difffile cannot be read by the ASD tool.

Below the possible arguments and switches that can be applied to the script are summarised:

```
concept.pl
--extendfile <filename>
--inputfile <filename>
--outputfile <filename>
--difffile <filename>
--add
--del
--verbose [level]
--fill
--guid
--channel <to>=<from>,<to>=<from>,...
--state <to>=<from>,<to>=<from>,...
The contents of the <from> channel will be placed into the <to> channel
```

Add and *del* implement the previously described create and removal actions for the merging of template models. *Verbose*, *fill*, *guid*, *channel*, and *state* are optional arguments.

- The verbose switch prints some internal processing information of the script
- fill fills the unused stimuli with illegal rule cases for every state.
- The guid switch renumbers the model’s internal ids. Renumbering is necessary each time a template model is used.
- The channel and state arguments are strings containing a sequence of channels or states that needs to be renamed. In this context, rename means that the name of a channel or state in the inputfile is changed before it is combined with the extendfile, which results in the outputfile. The rename will also be visible in the difffile.

Note that from the outputfile and the difffile it should be possible to delete the added behaviour, but this is not (yet) implemented in the script. In Chapter 8, we describe that the template models need to be adapted when the requirements change. From these updated template models, a new interface model can be composed instead of removing the old behaviour and replace it with the new behaviour.

The script can be used for template models and the merging of template models and therefore has two basic execution scenarios. The first is renumbering unique ids for template models, depicted in Figure 63, and the other is merging interface models, as shown in Figure 64. The next section is devoted to some execution examples of the script.

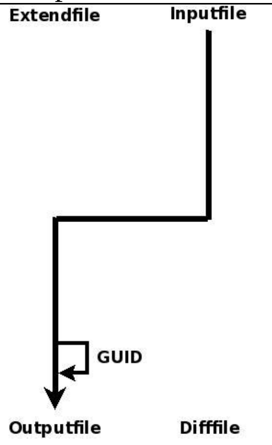


Figure 63, Renumber Unique IDs

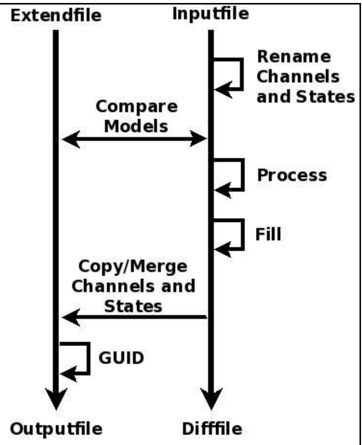


Figure 64, Merge Interface Models

9.3 Examples

In this section, we describe the basic operations the script can take in order to use template models or merge template models.

9.3.1 Template Interface Models

It is impossible to re-use an interface model file in a design, as is described in subparagraph 8.2.1. If one tries to do this, the tool will complain about duplication of unique ids. Figure 63 shows the execution scenario of the script to renumber ids. The script can be used to renumber ids in ASD interface model “in.im” leading to ASD model “out.im”, by the command:

```
concept.pl -inputfile in.im -outputfile out.im -guid
```

9.3.2 Merge Interface Models

Merging of interface models can be used to compose an interface model out of, for example, multiple template models. The script can extend one interface model with an input interface model, see Figure 64. It is possible to merge more than two models but then the script has to be executed multiple times.

The merging of two interface models involves two activities: copying and merging of channels, and copying and merging of states. Besides these two activities, some pre-processing activities such as renaming of channel and state names are optional. Renumber the unique ids (subparagraph 9.3.1) and filling empty rules with illegals are optional post-processing steps.

For merging interface models the add switch of the script is used, see:

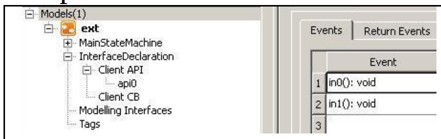
```
concept.pl -add -extendfile ext.im -inputfile in.im -outputfile out.im
```

9.3.3 Copy Channels

The merging of interface models consists of some elementary actions. One of these actions is copying channels. Copying of channels needs two interface models to generate a new interface model. One of these models is the extendfile which is used as a base file. If a channel name in the inputfile does not exist in the extendfile, then the channel is copied after the channels of the extendfile to form a new outputfile.

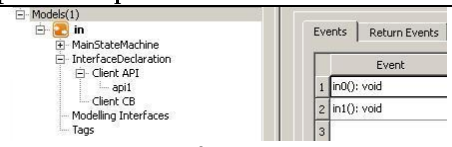
9.3.3.1 Copy API

If the inputfile contains API channels, which are a set of stimuli, then these are placed after the API channels of the extendfile in the outputfile; this can be seen in Figure 65, Figure 66 and Figure 67. The stimuli inside the channel api1 are also copied to the outputfile. Note that the channel api0 and api1 can have stimuli with the same name.



	Event
1	in0(): void
2	in1(): void
3	

Figure 65, Extendfile



	Event
1	in0(): void
2	in1(): void
3	

Figure 66, Inputfile

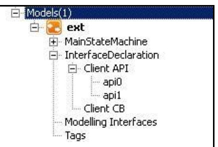
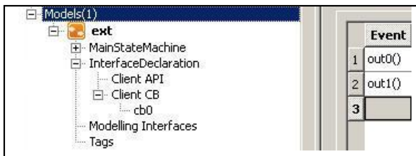


Figure 67, Outputfile

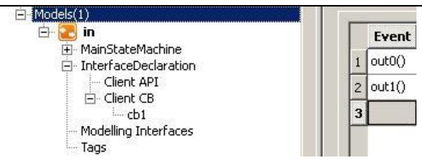
9.3.3.2 Copy CB

If the inputfile contains CallBack (CB) channels, which are a set of callbacks, then these are placed after the CB channels of the extendfile in the outputfile; this can be seen in Figure 68, Figure 69 and Figure 70. The callbacks inside the channel cb1 are copied to the outputfile. Note that the channel cb0 and cb1 can have callbacks with the same name.



	Event
1	out0()
2	out1()
3	

Figure 68, Extendfile



	Event
1	out0()
2	out1()
3	

Figure 69, Inputfile

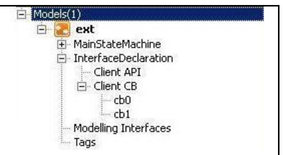


Figure 70, Outputfile

9.3.4 Merge Channels

Merging of channels needs two interface models to generate a new interface model. If a channel in the inputfile and the extendfile has the same name, then the channel is merged in the outputfile. As explained before, channel consists of a set of stimuli or callbacks. When merging two channels, the resulting channel will be the union of the two. Hence, without duplication of stimuli or callbacks.

9.3.4.1 Merge Stimuli

If the inputfile contains an API channel, which is a set of stimuli, with the same name as the extendfile, then the API channel will be merged in the outputfile; this can be seen in Figure 71, Figure 72 and Figure 73. Note that if a stimulus occurs in both channel api0 and api1, then in the outputfile this stimulus occurs only once. In the difffile can be seen that such a stimulus is removed and therefore not copied to the outputfile.

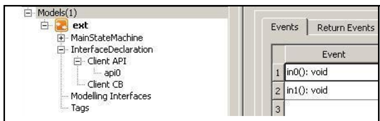


Figure 71, Extendfile

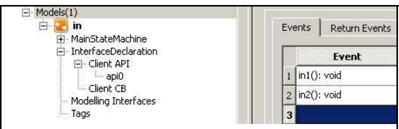


Figure 72, Inputfile

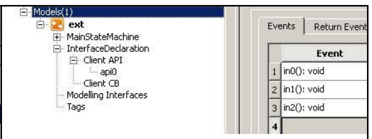


Figure 73, Outputfile

9.3.4.2 Merge CBs

If the inputfile contains a CB channel, which is a set of callbacks, with the same name as the extendfile, then the CB channel will be merged in the outputfile; this can be seen in Figure 74, Figure 75 and Figure 76. Note that if a callbacks occurs in both channel cb0 and cb1, then in the outputfile this callbacks occurs only once. In the difffile can be seen that such a callback is removed and therefore not copied to the outputfile.

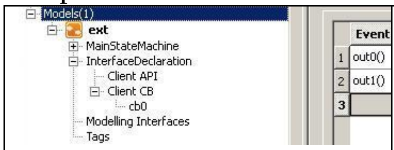


Figure 74, Extendfile

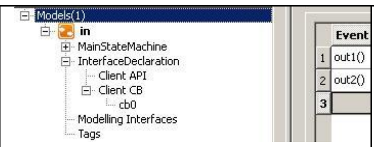


Figure 75, Inputfile

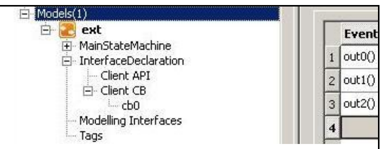


Figure 76, Outputfile

9.3.5 Rename Channels

If a template model with certain names for channels but for the composed interface model other names are required, then as a pre-processing step it is possible to rename API and CB channels. After renaming the resulting channels will either be copied or merged. Renaming channels works as follows:

```
concept.pl -add -extendfile ext.im -inputfile in.im -outputfile out.im
-channel api2=api1,cb2=cb1
```

Execution of the script on the models in Figure 77 and Figure 78 will result in a model depicted by Figure 79.

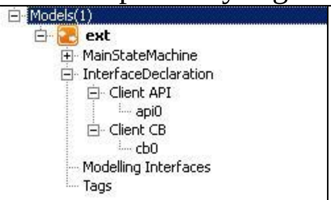


Figure 77, Extendfile

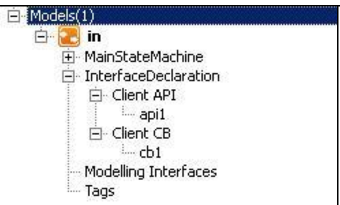


Figure 78, Inputfile

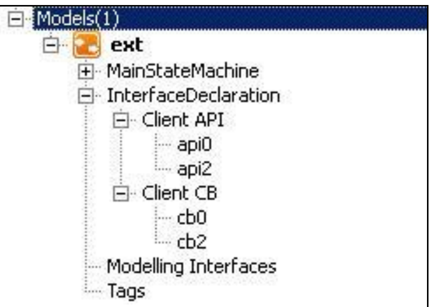
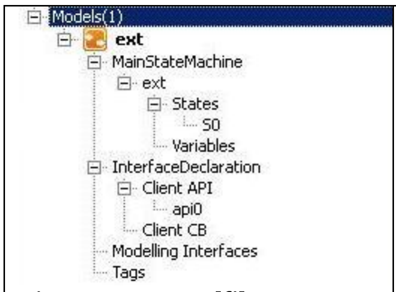


Figure 79, Outputfile

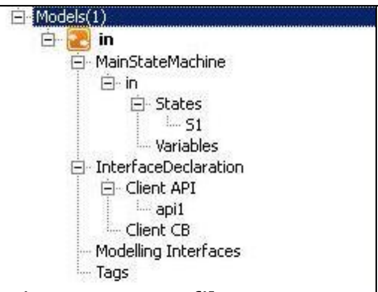
9.3.6 Copy States

The copying of states works similar as the copying of channels. If the inputfile contains states, which has a set of rule cases, then these are placed after the states of the extendfile in the outputfile; this can be seen in Figure 80, Figure 81 and Figure 82. Note that the channel api1 is also copied to the outputfile. The rule cases corresponding state S1 are copied to the outputfile. In Figure 83 the Sequence Based Specification (SBS) is depicted of the outputfile. Lines 5, 6, 9, and 10 are empty rule cases and should be filled in by hand.



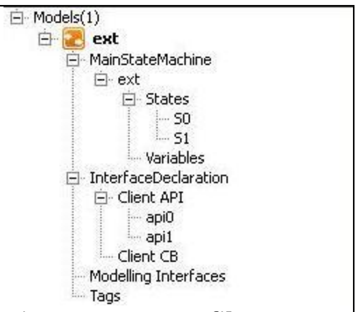
```
Models(1)
  ext
    MainStateMachine
      ext
        States
          S0
        Variables
    InterfaceDeclaration
      Client API
        api0
      Client CB
    Modelling Interfaces
    Tags
```

Figure 80, Extendfile



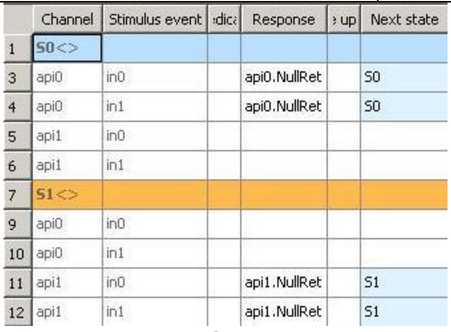
```
Models(1)
  in
    MainStateMachine
      in
        States
          S1
        Variables
    InterfaceDeclaration
      Client API
        api1
      Client CB
    Modelling Interfaces
    Tags
```

Figure 81, Inputfile



```
Models(1)
  ext
    MainStateMachine
      ext
        States
          S0
          S1
        Variables
    InterfaceDeclaration
      Client API
        api0
        api1
      Client CB
    Modelling Interfaces
    Tags
```

Figure 82, Outputfile

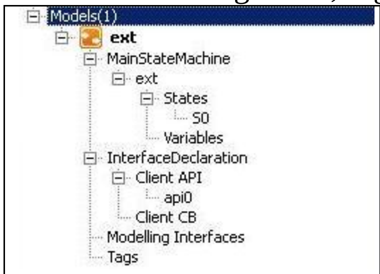


	Channel	Stimulus event	·dic	Response	› up	Next state
1	S0<>					
3	api0	in0		api0.NullRet		S0
4	api0	in1		api0.NullRet		S0
5	api1	in0				
6	api1	in1				
7	S1<>					
9	api0	in0				
10	api0	in1				
11	api1	in0		api1.NullRet		S1
12	api1	in1		api1.NullRet		S1

Figure 83, Outputfile (SBS)

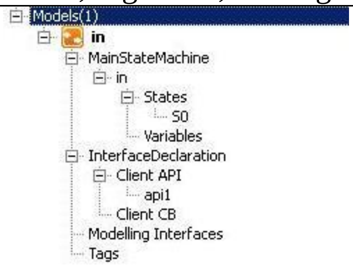
9.3.7 Merge States

If the inputfile contains a state, which has a set of rule cases, with the same state name as the extendfile, then the rule cases of the states will be merged in the outputfile; this can be seen in Figure 84, Figure 85, Figure 86, and Figure 87.



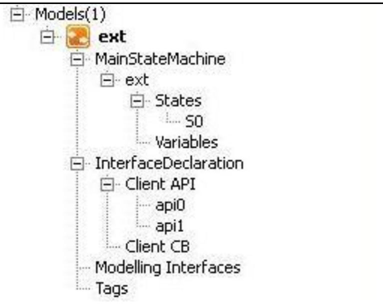
```
Models(1)
  ext
    MainStateMachine
      ext
        States
          S0
        Variables
    InterfaceDeclaration
      Client API
        api0
      Client CB
    Modelling Interfaces
    Tags
```

Figure 84, Extendfile



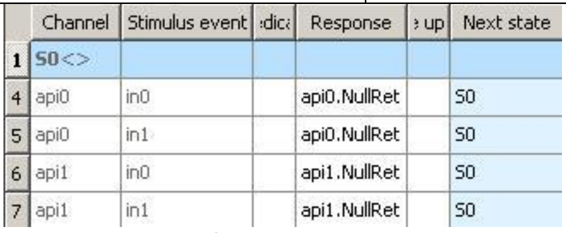
```
Models(1)
  in
    MainStateMachine
      in
        States
          S0
        Variables
    InterfaceDeclaration
      Client API
        api1
      Client CB
    Modelling Interfaces
    Tags
```

Figure 85, Inputfile



```
Models(1)
  ext
    MainStateMachine
      ext
        States
          S0
        Variables
    InterfaceDeclaration
      Client API
        api0
        api1
      Client CB
    Modelling Interfaces
    Tags
```

Figure 86, Outputfile



	Channel	Stimulus event	·dic	Response	› up	Next state
1	S0<>					
4	api0	in0		api0.NullRet		S0
5	api0	in1		api0.NullRet		S0
6	api1	in0		api1.NullRet		S0
7	api1	in1		api1.NullRet		S0

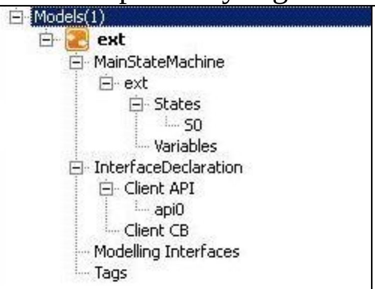
Figure 87, Outputfile (SBS)

9.3.8 Rename States

If a template model with certain names for channels but for the composed interface model other names are required, then as a pre-processing step it is possible to rename states. After renaming the resulting states will either be copied or merged. Renaming states works as follows:

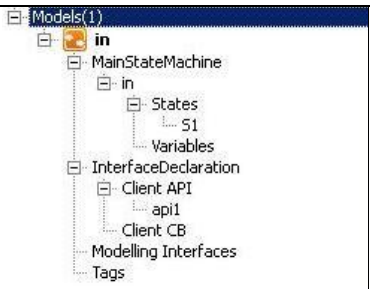
```
concept.pl -add -extendfile ext.im -inputfile in.im -outputfile out.im  -state S2=S1
```

Execution of the script on the models in Figure 88 and Figure 89 will result in a model depicted by Figure 90.



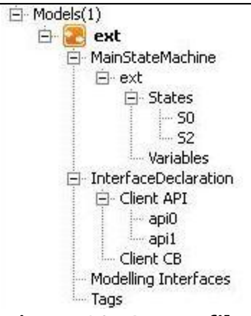
```
Models(1)
  ext
    MainStateMachine
      ext
        States
          S0
        Variables
    InterfaceDeclaration
      Client API
        api0
      Client CB
    Modelling Interfaces
    Tags
```

Figure 88, Extendfile



```
Models(1)
  in
    MainStateMachine
      in
        States
          S1
        Variables
    InterfaceDeclaration
      Client API
        api1
      Client CB
    Modelling Interfaces
    Tags
```

Figure 89, Inputfile



```
Models(1)
  ext
    MainStateMachine
      ext
        States
          S0
          S2
        Variables
    InterfaceDeclaration
      Client API
        api0
        api1
      Client CB
    Modelling Interfaces
    Tags
```

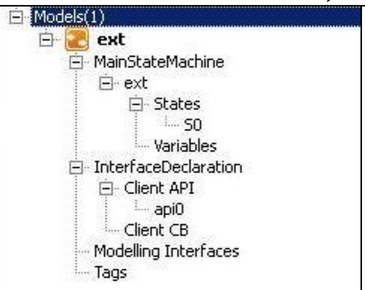
Figure 90, Outputfile

9.3.9 Fill with Illegals

In Figure 83 a SBS is depicted with empty rule cases. The empty rule cases that occur after the merging of interface models can be filled with illegals by the fill switch; this can be done as follows:

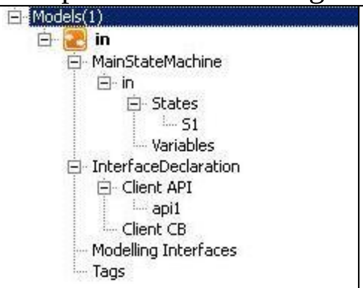
```
concept.pl -add -extendfile ext.im -inputfile in.im -outputfile out.im  -fill
```

The models extendfile (Figure 91) and inputfile (Figure 92) are merged to form an outputfile (Figure 93). In Figure 94 the resulting SBS of the outputfile is depicted. Because of the fill switch, the script has added an illegal on line 4 and 7 of the SBS.



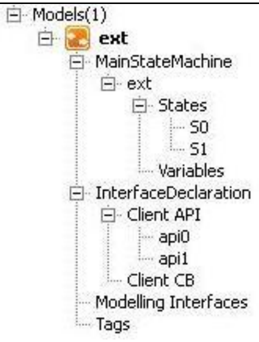
```
Models(1)
  ext
    MainStateMachine
      ext
        States
          S0
        Variables
    InterfaceDeclaration
      Client API
        api0
      Client CB
    Modelling Interfaces
    Tags
```

Figure 91, Extendfile



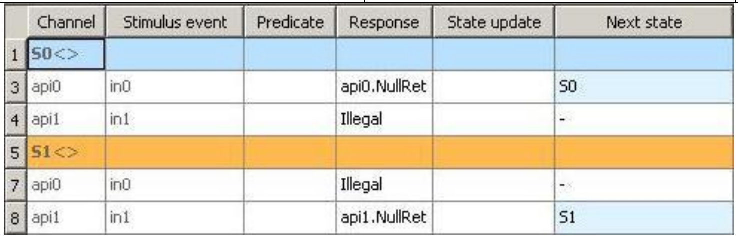
```
Models(1)
  in
    MainStateMachine
      in
        States
          S1
        Variables
    InterfaceDeclaration
      Client API
        api1
      Client CB
    Modelling Interfaces
    Tags
```

Figure 92, Inputfile



```
Models(1)
  ext
    MainStateMachine
      ext
        States
          S0
          S1
        Variables
    InterfaceDeclaration
      Client API
        api0
        api1
      Client CB
    Modelling Interfaces
    Tags
```

Figure 93, Outputfile



	Channel	Stimulus event	Predicate	Response	State update	Next state
1	S0<>					
3	api0	in0		api0.NullRet		S0
4	api1	in1		Illegal		-
5	S1<>					
7	api0	in0		Illegal		-
8	api1	in1		api1.NullRet		S1

Figure 94, Outputfile (SBS)

10. Migration of an Application

In Chapter 7 both managers and software designers pointed out that it is unclear how ASD generated source code should be intertwined with the legacy or current source code in a software architecture. To answer this question, one of the architectural units, at the application-layer, is adapted to provide an additional ASD interface. The main goal of our approach is that an application can be used during the migration from the current architecture to the new reference architecture with ASD, as indicated by Figure 95.

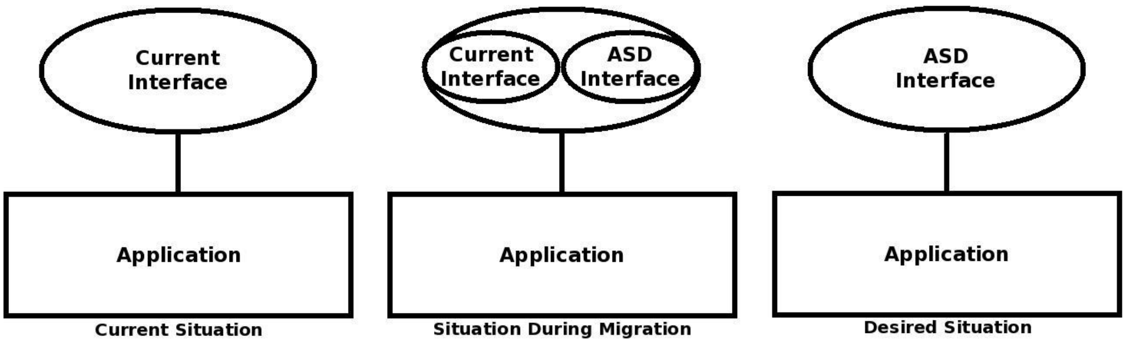


Figure 95, Migration of an Application

As a proof-of-concept, the Beam Limitation application is adapted such that ASD clients can use it. The Beam Limitation application is introduced in Section 10.1. In the desired situation of the new architecture, the application's interface will be implemented in ASD. In the new reference architecture Beam Limitation has a different name, namely Beam Limitation Controller. Beam Limitation Controller with an ASD interface incorporated is explained in Section 10.2. Because there are multiple architectural units (applications) at the application-layer of the FEC's reference architecture, Sections 10.3, 10.4, and 10.5 describe a general incremental approach to adapt applications.

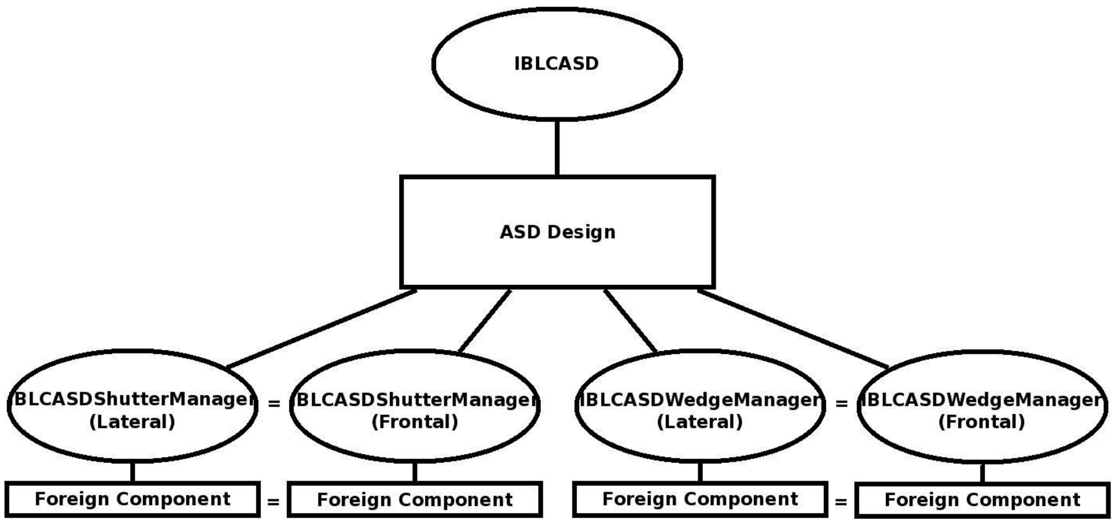
10.1 Old Architecture

Beam Limitation is an application which is responsible for shaping and filtering the x-ray beam. The objectives are to reduce the area exposed with radiation and in the same time shape the beam such that the image quality is improved. To accomplish these two objectives the Beam Limitation has shutters and wedges. Shutters are used to reduce the area exposed with radiation and wedges filter the beam to improve the image quality. The monoplane product variant can produce one x-ray beam, and has one set of shutters and one set of wedges. The biplane variant has two sets of shutters and two sets of wedges.

The application acts as a server for architectural units in the user-interaction layer which is positioned above the application layer. Additionally, the application acts as a client to the architectural units in the technical-services layer. Architectural units in the technical-services layer are hardware abstractions and therefore responsible for driving the hardware. Note that each architectural unit is implemented as a single binary. So, every architectural unit has a interface which it provides to architectural units in an upper layer.

10.2 New Architecture

In the new reference architecture, the application Beam Limitation Controller will have comparable responsibilities as the old Beam Limitation application. Architectural control units at higher-levels than the applications will be developed in ASD. Architectural units at lower levels use proven concepts that will remain the same. Hence, the architecture is split in two parts, an ASD part and a non-ASD part, which meet at the application layer. Consequently, at the top – in their server role - applications need to interact with ASD units and at the bottom – in their client role - they need to interact to units in the old way. As indicated in Figure 95, the application's interface will be specified in ASD, whereas the bottom of the application will remain the same. Figure 96 depicts how ASD will be incorporated into the upper half of the application. The foreign components are responsible for interacting with the current lower half of the application and are implemented by hand. The other interfaces in the picture and the design are made by ASD. From this design and interfaces, source code is generated and incorporated into the application. As indicated by Figure 95, the application has an interface which provides the current commands and the new commands in ASD form during migration. After migration, the current commands can be removed. Henceforth, commands in the ASD interface are called "ASD commands".


IBLCASD
ASD Design
IBLCASDShutterManager (Lateral)
=
IBLCASDShutterManager (Frontal)
IBLCASDWedgeManager (Lateral)
=
IBLCASDWedgeManager (Frontal)
Foreign Component
=
Foreign Component
Foreign Component
=
Foreign Component
Figure 96, ASD in Beam Limitation

10.3 Approach

In order to manage complexity, we propose to change the application and its test client incrementally. The test client is used to test the new functionality of the application. In this section, we list the steps used to adapt the Beam Limitation application. These steps are generic and can also be used to adapt other architectural units at the application layer. The following steps have been taken and are explained in the next sections:

- create an ASD environment;
- create a build environment;
- create a test environment;
- extend the application with one ASD command of a group of commands;
- extend the application with all related ASD commands in the same group;

- incrementally extend the application with new groups of ASD commands; and,
- optionally, remove the old commands from the application.

10.4 Prerequisites

Before the application can be changed, an appropriate environment needs to be set up. Setting up the environment involves setting up the ASD environment, the build environment, and the test environment. In the following subsections these environments are explained shortly.

10.4.1 Create an ASD Environment

To get familiar with ASD, the approach starts with a small ASD design which consists of a implemented interface model, a design model, and a used interface model. After model checking these models, the tool can be used to create source code.

10.4.2 Create an Build Environment

Microsoft Visual Studio is used by SW-FE; therefore, C++ source code needs to be generated by the ASD tool and added to a test project. Also, the ASD:Runtime files need to be added to the Microsoft Visual Studio project. Besides the generated and runtime files, the project consists of two additional files:

- A file with the main function that invokes functions provided by the source code file of the ASD implemented interface.
- A file that implements the functions that can be called with the used interface.

By using these two files the whole path through the ASD generated code can be tested.

10.4.3 Create an Test Environment

The following step is to setup an environment to compile the Beam Limitation application and the test client, and to install a VMware image that can execute both the application and the test client. The ASD:Runtime was also added to the Beam Limitation project.

10.4.4 Extend Application with One ASD Command

If it is possible to compile the application and the test client and to execute these in the VMware image successfully, then ASD can be incorporated. To test the complete path of ASD from test client to application, an ASD model with one command is built and source code is generated. The ASD command is added to the implemented interface of the application. The used component of the generated source code can, for example, toggle a Boolean value. The test client is extended with an additional button that executes the ASD command on the application when the button is pushed and notifies the user with the return value. Because the toggling of the Boolean, the test client should give another message than the prior message when pushing the button. This way the whole path from test client to applications can be tested. The next step is to integrate the ASD generated code with the application's functionality. The toggling of the Boolean value is removed and replaced by, for example, executing a shutter command on the lateral shutter manager. To this end, the shutter manager has to be extended and changed as well.

10.5 Extending the Application

Next, the application can be extended. As can be seen in Figure 96, the application has a implemented interface (IBLCASD) which should contain ASD commands from the used interfaces (IBLASDShutterManager and IBLASDWedgeManager). For both used interfaces there is a lateral and frontal instance. It is possible to create a generic channel in the IBLCASD interface with, for example, the following commands for the shutter managers (SM) and the wedge managers (WM):

- BLASDLateralSMCommand,
- BLASDFrontalSMCommand,
- BLASDLateralWMCommand, and
- BLASDFrontalWMCommand.

The consequence of this choice would be that the designer of the IBLCASD interface model has to type them all in by hand. In the light of re-use it is more convenient to generate the IBLCASD interface model from the used interfaces and used channels to establish differences between the commands, especially when there are a large number of commands. For the same commands described before this leads to four channels:

- ASDSMLateral channel with command
  - BLASDCommand
- ASDSMFrontal channel with command
  - BLASDCommand
- ASDWMLateral channel with command
  - BLASDCommand
- ASDWMFrontal channel with command
  - BLASDCommand

Where ASDSMLateral, ASDSMFrontal, ASDWMLateral and ASDWMFrontal are channel names.

For extending the application, we follow an incremental approach and the proposal of Section 9.1 to use a bottom-up approach to construct ASD interfaces for maximal re-use. To extend the application, we re-use the used interfaces to generate the implemented interface with a script.

10.5.1 Extend Application with All Related ASD Commands

When the execution of one shutter command is working, the same recipe can be used for the complete group of shutter commands with, of course, an also extended ASD model and generated code. This should result in a working ASD interface on top of Beam Limitation that can invoke lateral shutter commands.

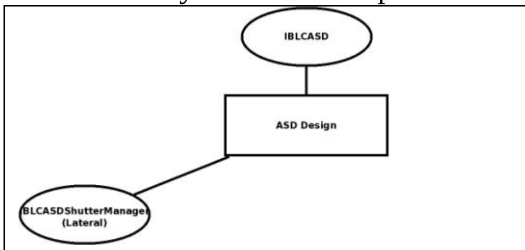
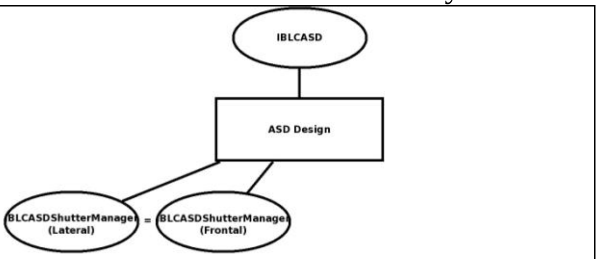
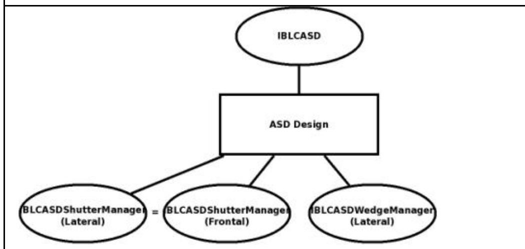
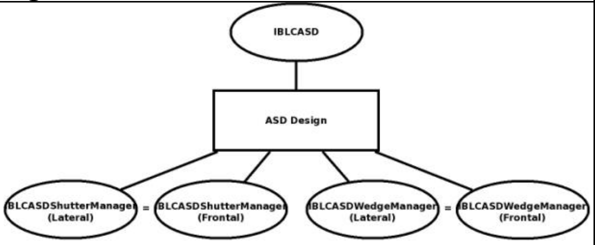
From the IBLASDShutterManager interface, the IBLCASDv1 interface is generated by executing the script as follows:

```
concept.pl -inputfile IBLASDShutterManager.im -outputfile IBLCASDv1.im -guid
```

The guid switch of the script is used because it is not allowed by the tool to use two interface models with duplication of ids for a design model. When IBLCASD is generated, a design model needs to be build that uses IBLASDShutterManager as used interface, see Figure 96. With the ASD tool the channel name ASDSMCommands has to be renamed to ASDSMLateral in IBLCASDv1 by hand.

10.5.2 Incrementally Extend the Application

Subsection 10.5.1 describes the first increment (see Figure 97) of adding the lateral shutter ASD commands. Figure 98, Figure 99 and Figure 100 depict an approach to incrementally extend the implemented ASD interface with new functionality.

 Figure 97, First Increment	 IBLCASD ASD Design BLCASDShutterManage (Lateral) = BLCASDShutterManage (Frontal) Figure 98, Second Increment
 IBLCASD ASD Design BLCASDShutterManage (Lateral) = BLCASDShutterManage (Frontal) BLCASDWedgeManager (Lateral) Figure 99, Third Increment	 IBLCASD ASD Design BLCASDShutterManage (Lateral) = BLCASDShutterManage (Frontal) BLCASDWedgeManage (Lateral) = BLCASDWedgeManager (Frontal) Figure 100, Fourth Increment

In the second increment, the frontal shutter manager is added, see Figure 98. This can be done by creating a second instance of the used component in the design model. The only difference between the two used components is that they get a different pointer from either the frontal or lateral shutter manager.

The IBLCASDv1 interface model is extended with the frontal shutter commands by executing the script as follows:

```
concept.pl -add -extendfile IBLCASDv1.im -inputfile IBLASDShutterManager.im
-outputfile IBLCASDv2.im -channel ASDSMFrontal=ASDSMCommands -guid
```

After execution of the script, the IBLCASDv2 interface model will have two sets with the same shutter manager commands but these sets are grouped by different channel names, ASDSMLateral and ASDSMFrontal respectively.

The third increment is used to add the lateral wedge commands, see Figure 99. To accomplish this, the IBLASDWedgeManager interface model is built. The foreign component that implements this interface is build and the wedge manager is adapted to execute the commands.

The IBLCASDv2 interface model is extended with the lateral wedge commands in the following way:

```
concept.pl -add -extendfile IBLCASDv2.im -inputfile IBLASDWedgeManager.im
-outputfile IBLCASDv3.im -channel ASDWMLateral=ASDWMCommands -guid
```

After execution of the script, the IBLCASDv3 interface model will have a set of wedge manager commands. Next the design model has to be adapted to correlate the wedge commands on the top and bottom interface.

In the fourth increment, the frontal wedge manager is added, see Figure 100, similar to the addition of the second shutter manager in the second step. The script can be used as follows to extend the IBLCASDv3 interface model with the frontal wedge commands:

```
concept.pl -add -extendfile IBLCASDv3.im -inputfile IBLASDWedgeManager.im
-outputfile IBLCASDv4.im -channel ASDWMFrontal=ASDWMCommands -guid
```

After execution of the script, the IBLCASDv4 interface model will have two sets with the same wedge manager commands but these sets are grouped by different channel names, ASDWMLateral and ASDWMFrontal respectively.

When all ASD commands are added to the application, the test client can be adapted to execute all shutter and wedge ASD commands and the application can be tested.

10.5.3 Remove the Old Commands

When the above described actions are taken, the result is an application that can execute both commands in the old way and the new ASD commands. Figure 95 depicts that after migration, the commands from the prior architecture may be removed. Hence, these non-ASD commands are unnecessary for the new reference architecture.

V Concluding Remarks

In Part V, we sought answers to the question: How can re-use and migration enhance the technology?

We have described techniques to re-use partial behaviour in models. The aim of re-using behaviour is to avoid redoing previous work and to avoid copy/pasting behaviour. A script implements the following two techniques: template models and merging template models. We propose that the script is used in the following manner. Firstly, decompose an architecture with a Cleanroom-like method. Secondly, identify common behaviour on the interfaces. Thirdly, place common behaviour in template models and compose interfaces, by merging template models, bottom-up.

Applying both techniques of the script on a real case provides evidence about the usefulness of re-use when implementing a design in Cleanroom/ASD. This real case was also used to investigate how to intertwine the new technology with the current technology. Therefore, an architectural unit was adapted to be able to act as a server for the new technology and as a client for the current technology.

To answer the question: both re-use and migration as is described by this part can enhance the technology.

11. Conclusion

11.1 Answers to the Research Questions

In this thesis, we have analyzed the transition situation of a software development organisation that wants to introduce and implement a new tool called ASD. By the transition situation we mean from the current situation without ASD to the desired situation with ASD. We have done a case study at Philips Healthcare, but the conclusions are generic enough to be useful for other software development organisation that wants to introduce and implement ASD into their organisation. Additionally, we have created some techniques that could improve ASD and its use. We use the structure of the thesis also in the conclusions.

Part I: What is the state of the organisation that will incorporate the new technology?
The business unit Interventional X-ray has a generic process model used for all development activities, regardless of the engineering discipline. Hence, mechanical engineering, electrical engineering and software engineering all use the same meta process model. The software development organisation currently uses the V-model which fits into the generic process model. In this thesis, we distinguish the following three main phases: global design, detailed design, and test & integration.

Components of products are made in projects. For each project, the required persons are claimed from the departments, which are organized by discipline. The software departments can offer software architects, software designers, and software engineers. These persons are trained and used to develop software with an Object Oriented Analysis and Design (OOAD) method.

Part II: What is the nature of the technology?
Verum's Analytic Software Design (ASD) is a new tool that can be used for designing control-based software in a component-based way. In ASD, a system is specified in a Sequence-Based Specification (SBS), which is a large table. The table describes for all states of the system how it should respond to all possible stimuli. A complete design specification can be verified formally. The Sequence-Based Specification can be used to generate source code. The source code can then be integrated into the final product. As said before, the tool is relatively new and therefore, during the research, new releases have further matured ASD.

The literature indicates that ASD is related to Cleanroom Software Engineering (CSE). Specifically, the component-based aspect of ASD originates from CSE. CSE has been developed in the late 1970s by IBM and describes the process and method that could be used to develop high quality software. The philosophy of CSE is error prevention instead of error removal during the test & integration phase of the software development process. The error prevention of CSE is accomplished by formally verifying designs by hand.

Part III: What is the ultimate goal for acquiring and using the technology?
The objective of introducing ASD is to improve software development. Currently, the test & integration phase, compared with the other phases, takes too long and makes the process difficult to manage. The quality of the software in products that leave the factory is unprecedented but according to some of the interviewed persons the reason

for the long test & integration phase is the quality of the software which is supplied to this phase. An important part of the problem is that independently developed software units do not work together seamlessly.

The current approach to manage test & integration problems is to go quickly through the V-model. In this way there is sooner something that can be tested, which gives management the perception of control. By going fast through the V-model, the focus is on testing the quality into the software during the test & integration phase.

As a new approach, management puts an effort into improving the software quality by providing software designers with a new tool, namely ASD. The tool should be used to make designs, but the introduction of a new tool is not the objective. By improving the software quality, the test & integration phase should become shorter and thereby improve the whole software development process.

The ASD tool can be applied for designing control-based software. Of course, not all software is control-based. Additionally, not all software will be instantaneously made with ASD. Hence, there will be large portions of handwritten legacy code during migrating to the new reference architecture. However, the objective for improving the software development also applies for the non-control-based software.

Everybody that has been interviewed and has followed the ASD course agrees that the ASD course is an introduction to the ASD tool. During the course some prefabricated models are adapted, but none of the developers has an idea on how to start building a model. The required method to apply the tool is not addressed.

According to Verum, a software architecture can be designed in the way architects and designers are used to make designs. Hence, the current practice does not need to be changed for applying ASD. When such a design has been made, units can be chosen to apply ASD.

Part IV: What are the steps to reach the desired goals given the state of the organisation?
In our analysis, we noticed that currently ASD is positioned as a tool and therefore requires only changes in the skills of the persons who need to apply it. However, we also noticed that making a design following the current practices did result in designs that could not be checked by the ASD tool. This observation, combined with the fact that the ASD course did not explain how to create a new design with ASD, implies that a method to apply the ASD tool is missing. From Part II we know that ASD and Cleanroom are related technologies. Cleanroom Software Engineering describes a complete software development process and method. Consequently, we propose to introduce a Cleanroom-like method and call the combination Cleanroom/ASD.

Cleanroom's process model is adapted to fit into the generic process model of the business unit. The process model of Cleanroom can be rewritten to the currently used phases. The Cleanroom method describes the steps to obtain a global and detailed design. In the final step of the method, ASD can be applied for control-based software. For non-control-based software a correctness proof could be made by hand. Of course, this will only be necessary for a fraction of the design. The benefit of using Cleanroom is that there is one integral approach to design high quality software.

Applying ASD will only become a success if the organisation is prepared to make concessions on the design. This is the most important hurdle that needs to be taken. Applying ASD is not just a change of skills for the persons that need to apply it, but rather a cultural change. Literature suggests that the change in mindset that is required may take at least a year to accomplish.

In summary, the first step is to acknowledge that a method is missing. The second step is to choose a candidate method. The third step is to adapt the method to the current situation. The last step is to infuse the technology, including the method, into the organisation.

Part V: How can re-use and migration enhance the technology?
We have described techniques to re-use partial behaviour in models. The aim of re-using behaviour is to avoid redoing previous work and to avoid copy/pasting behaviour. A script implements the following two techniques: template models and merging template models. We propose that the script is used in the following manner. Firstly, decompose an architecture with a Cleanroom-like method. Secondly, identify common behaviour on the interfaces. Thirdly, place common behaviour in template models and compose interfaces, by merging template models, bottom-up.

Applying both techniques of the script on a real case provides evidence about the usefulness of re-use when implementing a design in Cleanroom/ASD. This real case was also used to investigate how to intertwine the new technology with the current technology. Therefore, an architectural unit was adapted to be able to act as a server for the new technology and as a client for the current technology.

11.2 Final Remarks

This research was conducted during the preparation phase of a pilot project that will implement a new reference architecture. At the time of writing these final remarks the project is started and ASD will be applied on a substantial part of the architecture. The research contributed to the awareness process of the new technology and helped the software development organisation of the business unit interventional X-ray to make the transition to ASD.

Most of the lessons learned during the research have an impact on the current application of ASD, including:

- OO and ASD need different paradigms,
- ASD components should be relatively small,
- create a design that suits ASD, and
- there needs to be a method to apply the tool.

In the interviews management has described the intent to organise presentations to make designers enthusiastic about ASD by their peers. Shortly after the interviews took place management has successfully organized a day where such talks were given.

Verum's ASD course has been changed. Previously, the students only adapted models during the course; whereas the current course asks students to build models from scratch. Additionally, there is a tendency at Verum to incorporate re-use mechanisms into the tool. The template models technique -described in this thesis- which makes it possible to re-use interface models, is recently adopted into the ASD tool.

Bibliography

[1] Ananthpadmanabhan, H., Kale, C., Khambatti, M., Jin, Y., Usman, S. & Zhang, S. *Cleanroom Software Development.* Arizona State University.

[2] Broadfoot, G. (2005). *Introducing Formal Methods into Industry using Cleanroom and CSP*. Dedicated Systems Magazine, Q1 2005.

[3] Broadfoot, G. & Broadfoot, P. White Paper An Introduction to (ASD) *Analytic Software Design.*

[4] Broadfoot, G. & Broadfoot, P. (2003). *Academia and industry meet: Some experiences of formal methods in practice.* IEEE Computer Society, Tenth Asia-Pacific Software Engineering Conference.

[5] Graham, D., Veenendaal, van E., Evans, I. & Black, R. (2008). *Foundations of Software Testing: ISTQB Certification*. Course Technology. ISBN: 978-1-84480-989-9.

[6] Fayad, M., Tsai, W. & Fulghum, M. (1996). *Transition To Object-Oriented Software Development.* Communications of the ACM, Vol. 39, No. 2, February 1996.

[7] Fischbein, D. & Uchitel, S. (2008). *On correct and complete strong merging of partial behaviour models.* Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, 2008.

[8] Fowler, P. & Levine, L. (1993). *A Conceptual Framework for Software Technology Transition.* Software Engineering Institute; Carnegie Mellon University.

[9] *FDR2 Manual*. Retrieved on 14-9-'09 from the Internet: http://www.fsel.com/fdr2_manual.html

[10] Hausler, P., Linger, R. & Trammell, C. (1994). *Adopting Cleanroom software engineering with a phased approach.* IBM Systems Journal, Vol. 33, No. 1, 1994.

[11] Head, G. (1994). *Six-Sigma Software Using Cleanroom Software Engineering Techniques.* Hewlett-Packard Journal, June 1994.

[12] Henderson, J. *Why Isn't Cleanroom the Universal Software Development Methodology?* Loral Space Information Systems.

[13] Hoare, C. (2004). *Communicating Sequential Processes.* Prentice Hall International.

[14] Itabashi, G., Takahashi, K., Kato,Y., Suganuma T. & Shiratori N. (2004). *State Machine Specification with Reusability.* IEICE Trans. Fundamentals, Vol. E87-A, No. 11 November 2004.

[15] Itabashi, G., Takahashi, K., Kato,Y., Suganuma T. & Shiratori N. (2005) *Incremental Design of a State Machine Specification for Mobile and Real-time Systems.* Tohoku University.

[16] Jadalowen, I. (2010). *SENG 613 lecture notes: Structured Analysis and Structured Design.* Retrieved on 28-3-'10 from the Internet: http://pages.cpsc.ucalgary.ca/~jadalow/seng613/sasd_summary.html

[17] Linger, C. (1993). *Cleanroom Process Model.* Proceedings of the 15th International Conference on Software Engineering. IEEE Los Alamitos, CA., Computer Society Press, 1993.

[18] Linger, R. & Spangler, R. *The IBM Cleanroom Software Engineering Technology Transfer Program.*

[19] Meyer, B. (1989). *The new Culture of Software Development: Reflections on the practice of object-oriented design.* Proceedings of TOOLS 1989.

[20] Mills, H. (1988). *Stepwise Refinement and Verification in Box-Structured Systems.* University of Florida, Computer, June 1988.

[21] Oshana R. (1998) *An industrial Application of Cleanroom Software Engineering – Benefits Through Tailoring.*

[22] Oshana, R. & Coyle, F. (1997) *Implementing Cleanroom Software Engineering into a Mature CMM-Based Software Organisation.*

[23] Oshana, R. & Linger, R. (1999). *Capability Maturity Model Software Development Using Cleanroom Software Engineering Principles – Results of an Industry Project.* Proceedings of the 32nd Hawaii International Conference on System Sciences, 1999.

[24] Pressman, R. (2005). *Software Engineering: A Practitioner's Approach. Sixth Edition.* McGraw-Hill. ISBN: 007-123840-9.

[25] Reulink, N. & Lindeman, L. (2005) *Dictaat kwalitatief onderzoek.* College 23 november 2005.

[26] Uchitel, S & Chechik, M. (2004). *Merging partial behavioural models.* ACM SIGSOFT Software Engineering Notes. Vol. 29, Is. 6, November 2004.

[27] Verum Consultants B.V. Verum ASD:Suite 3.0.0 Version 4.0. *User Manual* (2009).

[28] Verum Consultants B.V. Verum ASD:Suite. Course Material (2009).

[29] Zelkowitz, M. (1996). *Software Engineering Technology Infusion within NASA.* University of Maryland.

Appendix A

Doel van dit onderzoek is het doen van aanbevelingen m.b.t. de introductie van een nieuwe tool in een team. Meer specifiek: de introductie van Analytical Software Design (ASD) in het Front-End Controller (FEC) team. Omdat voor een gedeelte van het team de overgang op korte termijn gaat plaatsvinden is het niet mogelijk om vooraf aanbevelingen te doen hoe dit het beste te begeleiden. Daarom zal de introductie van ASD voor een selecte groep van het FEC team worden gevolgd of gemonitord om de overgang van de grotere groep op een later tijdstip meer gestructureerd te laten verlopen. Hiervoor worden de aanbevelingen opgesteld.

De vragen moeten antwoord geven op de volgende aspecten: de achtergrond van de betrokken personen, het verloop van de omschakeling naar ASD, de implicaties voor de werkzaamheden en organisatie, en hoe de omschakeling een volgende keer beter kan verlopen.

De vragen over wat het voor personen persoonlijk en voor de organisatie betekent zijn nogal vaag. Afhankelijk van hoe het interview verloopt kunnen deze begrippen als volgt worden opgesplitst om deelvragen te stellen.
Persoonlijk:

- moeilijk om te leren,
- vast in een Object Oriented Programming (OOP) manier van denken,
- veranderende interactie met collega’s.

Organisatie:

- de huidige manier van werken,
- wat kan beter,
- wat ga je nodig hebben,
- betere software,
- snellere ontwikkeltijden.

<Inleiding: voorstellen en vertellen wat er met de antwoorden gebeurt.>

Achtergrond (voor deze ronde vragen
Wat voor opleiding heeft u genoten?
Waar hebt u gewerkt voordat u bij Philips begon?
Hoelang werkt u op deze afdeling?
Hebt u de pre-OOP tijd meegemaakt?
Bent u bekend met formele methoden?
  Zoja: welke formele methoden?
Bent u bekend met model-checkers?
  Zoja: welke model-checkers?
Hoe zien uw huidige werkzaamheden eruit?

Verwachtingen over ASD (terugkerende vragen)
Wat verwacht u van ASD voor u persoonlijk?
Wat verwacht u van ASD voor de organisatie?
Wat verwacht u van de invoering van ASD voor u persoonlijk?
Wat verwacht u van de invoering van ASD voor de organisatie?
Gaan uw dagelijkse werkzaamheden veranderen door de invoering van ASD?
  Hoe staat u daar tegenover?
Voor alle bovenstaande vragen: waar zijn deze verwachtingen op gebaseerd?

Voorbereiding op ASD (voor deze ronde vragen)
Hoe heeft “de organisatie” u voorbereid op het gebruik van ASD?
  Wie heeft u verteld dat ASD ingevoerd gaat worden?
    Hoe is dat verteld?
Wat waren de argumenten voor het invoeren van ASD?
  Hoe staat u tegenover deze argumenten?
Hoe zou u het liefst zijn voorbereid?

Kennis over ASD (terugkerende vragen)
Wat is de begeleiding die u hebt gekregen?
  Hebt u een cursus gevolgd?
    Wat hebt u opgestoken tijdens de cursus?
    Sluit deze cursus aan bij de praktijk?
  Hoe zou u het liefst zijn begeleid?

Overig (terugkerende vraag)
Ben ik nog iets vergeten te vragen wat u in dit verband toch aan mij kwijt wilt?

Appendix B

Doel van dit onderzoek is het doen van aanbevelingen m.b.t. de introductie van een nieuwe tool in een team. Meer specifiek: de introductie van Analytical Software Design (ASD) in het Front-End Controller (FEC) team. Omdat voor een gedeelte van het team de overgang op korte termijn gaat plaatsvinden is het niet mogelijk om vooraf aanbevelingen te doen hoe dit het beste te begeleiden. Daarom zal de introductie van ASD voor een selecte groep van het FEC team worden gevolgd of gemonitord om de overgang van de grotere groep op een later tijdstip meer gestructureerd te laten verlopen. Hiervoor worden de aanbevelingen opgesteld.

Naast dat ik de teamleden ga volgen, ben ik ook geïnteresseerd in wat de verwachtingen van het management zijn m.b.t. ASD en de invoering hiervan.

De vragen moeten antwoord geven op de volgende aspecten: het verloop van de omschakeling naar ASD, de implicaties voor de organisatie, en hoe de omschakeling een volgende keer beter kan verlopen.

De vraag over wat het voor de organisatie betekend is nogal vaag. Afhankelijk van hoe het interview verloopt, kan dit begrip als volgt worden opgesplitst om deelvragen te stellen.
Organisatie:

- de huidige manier van werken,
- de nieuwe manier van werken,
- voordelen ASD,
- nadelen ASD,
- wat kan beter,
- wat er is voor nodig,
- kwalitatief betere software,
- snellere ontwikkeltijden,
- beter stuurbaar,
- beter voorspelbaar.

<Inleiding: voorstellen en vertellen wat er met de antwoorden gebeurt.>

Globaal
Wat is uw rol binnen de organisatie?
Wat is uw achtergrond qua opleiding?

Kennis over ASD
Wat is volgens u ASD?
Wat zijn volgens u de voor- en nadelen van ASD?
Wat waren de argumenten voor en tegen het invoeren van ASD?
    Hoe staat u tegenover deze argumenten?
Wie heeft besloten dat ASD ingevoerd gaat worden?
Zijn er alternatieve tools overwogen (zoals I-Mathic Studio van Imtech)?
    Zoja: waarom prefereert de ASD tool?
Wat zijn tot nu toe de ervaringen met ASD?

Verum
Hoe staat u tegenover Verum (het bedrijf dat de ASD levert)?
    Wordt de organisatie niet erg afhankelijk van Verum?
        Slechte onderhandelingspositie voor nieuwe licenties e.d.
    Wat als Siemens Verum inlijft?
    Wat als Verum failliet gaat wat hebben de modellen dan nog voor waarde?

Verwachtingen over ASD
Wat verwacht u van ASD voor de organisatie?
Wat verwacht u van de invoering van ASD voor de organisatie?
Hoe denkt u het overgangsproces te gaan begeleiden?
Gaan alle software developers over op ASD?
    Blijft er nog legacy code?
    Wat als een software developer het conceptueel niet aan kan?
Voor alle bovenstaande vragen: waar zijn deze verwachtingen op gebaseerd?

Voorbereiding op ASD
Hoe hebt u "de organisatie" voorbereid op het gebruik van ASD?
    Wie heeft besloten dat ASD ingevoerd gaat worden?
        Hoe is dat gecommuniceerd?
Wat waren de argumenten voor/tegen het invoeren van ASD?
    Hoe staat u tegenover deze argumenten?

Overig
Ben ik nog iets vergeten te vragen wat u in dit verband toch aan mij kwijt wilt?