Integrated simulation in EUV source

3TU. School for Technological Design

STAN ACKERMANS INSTITUTE

Integrated Simulation in EUV Source

Tom Boshoven

Date: September 2015

TU Delft
Delft University of Technology

TU/e Technische Universiteit Eindhoven University of Technology

UNIVERSITY OF TWENTE.

Integrated Simulation in EUV Source

Eindhoven University of Technology
Stan Ackermans Institute / Software Technology

Partners

ASML Netherlands B.V.

Eindhoven University of Technology

Steering Group

Tom Boshoven
Dirk Coppelmans
Ernest Mithun Xavier Lobo
Pieter Cuijpers

Date September 2015

Document Status Public

The design described in this report has been carried out in accordance with the TU/e Code of Scientific Conduct.

Contact Address
Eindhoven University of Technology
Department of Mathematics and Computer Science
MF 7.090, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands
+31402474334

Published by
Eindhoven University of Technology
Stan Ackermans Institute

Printed by
Eindhoven University of Technology
UniversiteitsDrukkerij

ISBN
A catalogue record is available from the Eindhoven University of Technology Library

ISBN: 978-90-444-1389-2
(Eindverslagen Stan Ackermans Instituut ; 2015/051)

Keywords
Integration, Simulation, Software Interfaces, Integration Testing, Simulation Architecture, EUV Source

Preferred reference
Integrated Simulation in EUV Source, SAI Technical Report, September 2015. (978-90-444-1389-2)

Partnership
This project was supported by Eindhoven University of Technology and ASML Netherlands B.V.

Foreword

At ASML, the software test challenge increases with the complexity of the systems being developed. With a price tag of tens of millions of euros apiece, it is impossible to provide enough systems for testing. Many people within the organization have a full time job in making it possible to test the software without the need for an actual machine (or parts thereof).

We are making the system a reality, by faking it:

- to cope with limited physical capacity, we fake it
- to cope with hardware that is still under development, we fake it
- to cope with conditions that are hard to test, we fake it

Within the Source program, we focus on simulation. The goal is to provide developers with a *Virtual Source*. This is a (software only) virtual representation of the machine being developed. Virtualized computer platforms are combined with simulated hardware (sensors and actuators) and a simulated environment. The simulated environment can be perceived by the sensors and manipulated by the actuators.

Multiple disciplines are involved to create this type of simulation. Where complex products are being developed, there is a need for complex development facilities. As such, simulation is not a matter of choice, it is a necessity.

Current simulators have limitations. They are isolated and provide simulated behavior for a narrow part of the hardware and environment. On integration, the simulators are unaware of each other, making it impossible to test the integrated software as a whole.

Tom has been given the assignment to investigate simulator integration. Based on a dedicated use case, he has investigated how to integrate the isolated simulators. He has closely cooperated with software engineers, and challenged and improved the simulation model with physics engineers. He integrated available simulators and added simulator functionality. As a result, it was made possible to run the use case on a virtual environment instead of a physical one.

The short time span in which he got to know the problem field and the way he single-handedly developed his own network within the ASML organization deserves respect. In time, Tom took ownership and convinced others of the necessity of simulator integration.

Toms work and results have already led to finding a number of defects in the actual product. With that, he has successfully demonstrated the benefits of the approach and the potential of available isolated simulators. This is a step towards standardization of the design, which is not to be underestimated.

Dirk Coppelmans
Test Architect of the Source Program

August 23, 2015

Preface

This report documents the final project of Tom Boshoven, for completing the *Software Technology* (ST[1]) program, thereby acquiring the Professional Doctorate in Engineering (PDEng). This two-year post-master program was executed at Eindhoven University of Technology under the banner of the 3TU. Stan Ackermans Institute. It was concluded with a nine-month project, executed at ASML Netherlands B.V. (referred to as ASML). This report describes various aspects of this project, with a focus on design.

It should be noted that this report is a *public* version. In this version, specific information, such as the names and behavior of software components, is omitted. This information can be found in the confidential version of this report, which is available in ASML.

September 2015

[1] Also known under its Dutch name: Ontwerpersopleiding Technische Informatica (OOTI)

Acknowledgements

The completion of this project was only possible through the help of several people. I would like to thank the people who supported me during the course of this project.

In the first place, I would like to thank my project supervisors, who have been a big help in the successful completion of this project. In particular, I would like to thank my company supervisors Dirk Coppelmans and Ernest Mithun Xavier Lobo for their continuous support and guidance. Dirk's help in sharpening my soft skills and his input for my presentations were very useful during the project and will continue to be useful in the future. Ernest helped take my work to a higher level through his extremely valuable input in the technical aspects of the project and his extensive reviews of my documentation. I would also like to thank my TU/e supervisor Pieter Cuijpers, whose invaluable insights made him an important part of the project steering group.

During the project, many of my ASML colleagues have contributed to my project. I would like to thank Ludovico Verducci in particular, for his detailed explanations that helped me understand the existing software. Furthermore, the contributions of Pieter Koper to the architecture, and Maarten Dam, with whom I cooperated on the optical model, are appreciated.

I would like to express my thanks to Ad Aerts, the program director for the Software Technology program, and management assistant Maggy de Wert, for their support and care throughout the program. Additionally, I would like to thank the coaches and trainers for this program. They really helped improving my skills as a designer.

I am also grateful to my colleagues, the other Software Technology trainees, whose support and feedback were indispensable in the preparation for this final project.

Finally, I would like to thank my friends, family, and everyone else I did not thank explicitly, for their help and support during the nine months of this project.

Tom Boshoven

September 2015

Executive Summary

ASML is the world's leading provider of photolithography systems for the semiconductor industry. The new generation of these systems makes use of extreme ultraviolet (EUV) light for exposing wafers in order to create integrated circuits. Generation of this EUV light, which is done in the EUV Source system, requires coordination of various subsystems. Testing the software that drives the EUV Source is important for satisfying the ASML business drivers.

Part of this software is tested on the software-only Devbench platform using simulation. This gives the following benefits over testing on a physical machine:

- Higher availability of test environment
- Higher coverage of bad-weather behavior
- Higher configurability of machine
- Lower risk and cost of testing
- Higher reproducibility of test results

Because of these benefits, the goal is to cover as much of the software as possible using testing on Devbench.

Current simulation solutions simulate only parts of the system in isolation. This makes integration testing difficult because these parts may interact with each other. The interactions between these parts are not simulated.

By integrating existing simulation solutions, it is possible to achieve simulation of a larger part of the system. Such a simulation supports execution of tests that require multiple subsystems. As a result, integration test coverage on Devbench is increased.

The integration of simulators can be done by applying a generic approach and architecture guidelines to specific test cases. This leads to a high-level architecture that serves as a basis for the design of the interfaces between the simulators. By following this approach, a solution can be designed that fits in the software architecture of the EUV Source.

This principle was demonstrated by designing integrated solutions for testing two applications for the EUV Source. The design was validated by means of prototype implementations. It was shown that that the generic approach works, but also that its application leads to better integration test coverage using simulation. This leads to a lower dependency on a physical machine for testing, which reduces cost and development cycle time.

Based on these results, the recommendation is to apply this approach in other integration test cases.

Table of Contents

List of Figures

List of Tables

1 Introduction

ASML is the market leader in the production of photolithography systems for the semiconductor industry. The new generation of lithography systems from ASML makes use of extreme ultraviolet (EUV) light. Producing this light follows a highly complex process, which requires close cooperation of various software and hardware components. Because of the high cost and low availability of hardware for testing, simulation is used in the qualification of the software.

1.1 Context

In order to understand the factors that play a role in the project, it is necessary to understand the project context first. In the project, the core concepts are simulation and generation of EUV light for photolithography. An introduction to these concepts is given in the following sections.

1.1.1 Lithography

The wafer scanners that are produced by ASML make use of photolithography (simply referred to as lithography) to imprint a pattern on a silicon disk. This process is used in the production of integrated circuits (chips). By projecting light onto such a disk after coating it with photoresist material, it is possible to write detailed patterns onto the disk.

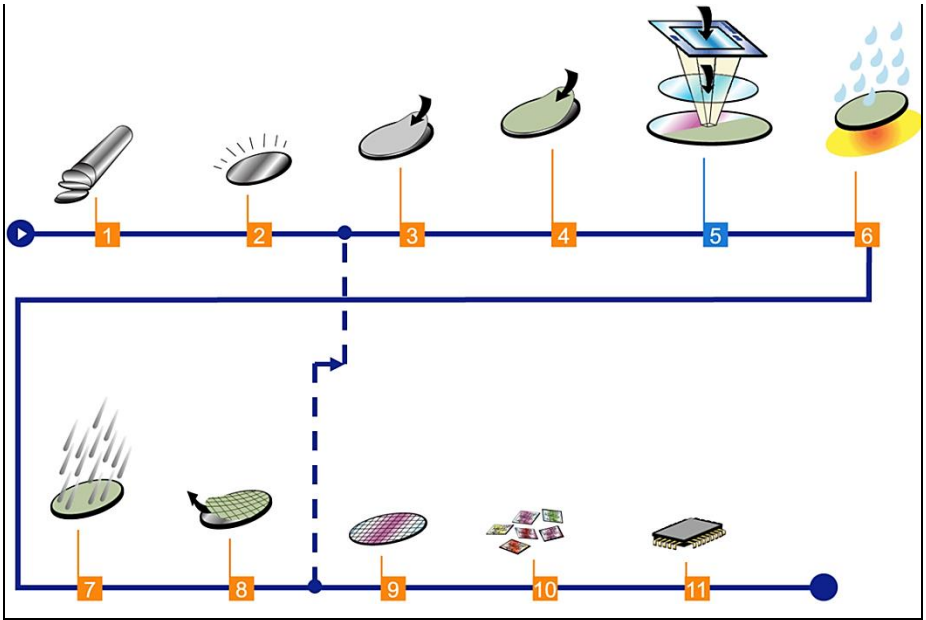

Figure 1.1 – Schematic representation of the lithography process

The process of creating an integrated circuit, which is shown in Figure 1.1, consists of several steps:

1. *Slicing*
   A single disc is sliced off from a silicon "boule."
2. *Polishing*
   The disc is polished to make the surface as smooth as possible.
3. *Material Deposition or Modification*
   Materials are transferred to the disc.

4. *Photoresist Coating*
   A photosensitive coating is applied to the disc. This coating protects the unexposed parts from being etched away in later steps.
5. ***Exposure*** *(only step in which ASML wafer scanners play a role)*
   The wafer is aligned and exposed to light, imprinting a pattern on the photoresist layer. This is the only step of the process that is done inside a wafer stepper or wafer scanner.
6. *Developing and Baking*
   The photoresist layer is developed and baked onto the wafer.
7. *Etching*
   The part of the material on the wafer that is no longer covered by photoresist is etched away.
8. *Ashing*
   The photoresist layer is removed. After this step, it is possible to continue with step 3, in order to form a three-dimensional structure. This is typically done 20 to 30 times.
9. *Testing*
   The result of this process is a disc with a wafer-like pattern. We call these discs wafers. Measurements are done to ensure the quality of the wafer.
10. *Dicing*
    The wafer is separated into the integrated circuits of which it is now built up.
11. *Packaging*
    The result is packaged and can be further processed into, for example, a processing unit.

The wafer scanners developed by ASML perform only the exposure step.

1.1.2 Extreme Ultraviolet

In photolithography, the wavelength of the used light can limit the detail of the patterns that can be projected onto a wafer. By using light with a shorter wavelength, it is possible to produce smaller, more detailed, and hence more efficient chips. The amount of detail that is currently required by the industry is on the order of nanometers (for comparison: a human hair grows about five nanometers per second).

By using lasers, it is possible to generate light in the deep infrared spectrum. To achieve even shorter wavelengths, the technique that is shown in Figure 1.2 can be used. Here, a laser beam (1) hits a droplet of molten tin (2), forming plasma (3). This plasma then emits light at extreme ultraviolet wavelengths. A collector (4) then focuses this light, after which it is propagated through the rest of the machine. The efficiency of this method is increased by performing an additional laser pulse (prepulse) to shape the droplet before hitting it with the main pulse.

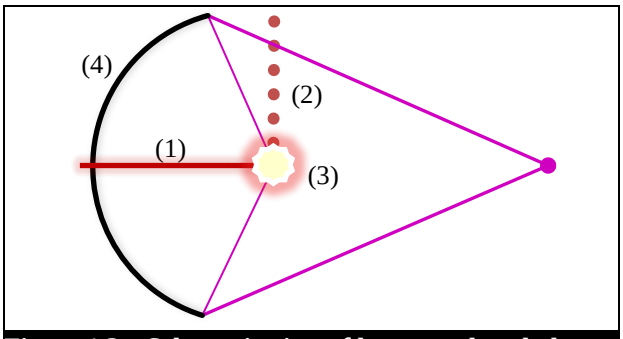

Figure 1.2 – Schematic view of laser-produced plasma

The system that performs this task of generating extreme ultraviolet light, called the EUV Source, is indicated in Figure 1.3. It is partially located in the wafer scanner (its computers and electronics exist outside the machine) and takes a laser beam as its input. Its EUV light output is propagated through the machine to the wafer using mirrors.

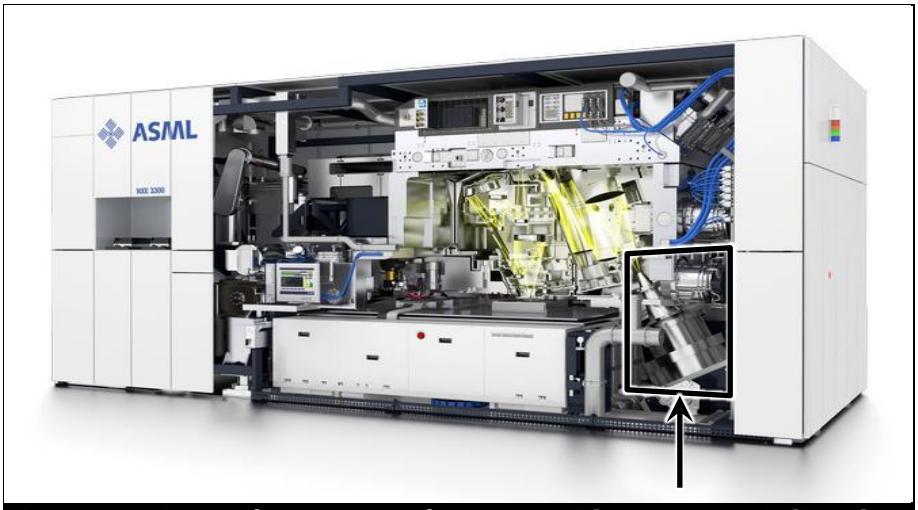

Figure 1.3 – Picture of open NXE wafer scanner with EUV Source indicated

In order to create a stable source of plasma, it is necessary to generate molten tin droplets, focus the laser, and time the laser pulses. This has to happen in a coordinated way. These concerns are separated over multiple subsystems.

1.1.3 Important Drivers

ASML emphasizes the following drivers:

- Critical dimension (minimum size of etched features)
- Image quality
- Overlay (position error when exposing multiple layers)
- Focus (focus stability of the projection)
- Throughput
- Yield (number of good ICs produced)
- Availability
- Cost of Ownership
- Reliability
- Time to market

Focus in this project is on the reliability and time to market.

The EUV Source is a vital part of a complex and expensive machine. Because any period during which this machine is unavailable leads to high cost, customers expect high availability from the system. In direct relation to this availability is the reliability of both software and hardware of the system.

Because the creation of integrated circuits happens in a highly competitive field, new features should be available to manufacturers as quickly as possible. This makes reducing the time to market an important driver for the company. Improving the development efficiency can reduce this time to market and lead to lower cost for the company.

1.1.4 Testing

Maintaining high software quality is important for satisfying the drivers. Because of the size and complexity of the software, ensuring software quality is a significant challenge. Software testing is one of the mechanisms that are used to tackle this challenge. Software testing on multiple levels (such as unit tests and software integration tests) is part of the development and maintenance process.

Platforms

In the EUV Source, three testing platforms are used. Proto is a test platform that consists of a complete machine. Testbench is a platform that contains the real electronics, but is missing mechanics and optics. It is not capable of producing EUV

light. Devbench is a software-only platform that is deployed on a virtual machine. Figure 1.4 compares these platforms based on functional completeness and availability. From Devbench to Proto, each platform is more functionally complete but has lower availability and exponentially greater cost.

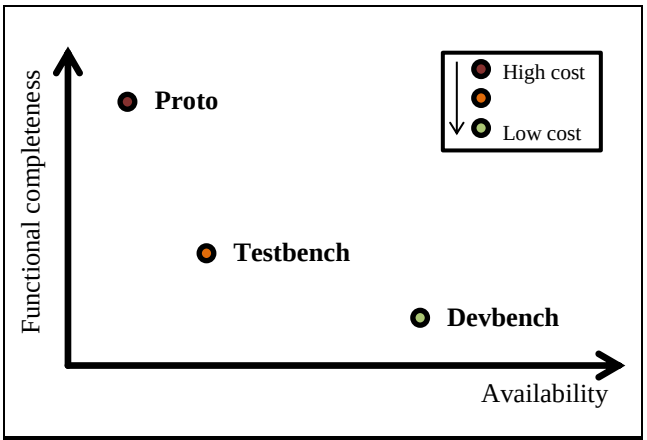

Figure 1.4 – Comparison of test platforms

The choice of platform depends on the testing requirements. Currently, most of the software is tested on Testbench and Proto. Extending test coverage on the Devbench should lead to more of the tests being executed on the Devbench before using the Testbench and finally Proto. This increases testing capabilities and reduces development cycle times, which leads to more reliable software and a shorter time to market.

Because the capabilities of the Devbench are limiting factors in the choice of the testing platform, improving these capabilities will lead to more software being verified on this platform. The use of simulation in testing can help improve these capabilities.

1.1.5 Simulation

In simulation, a part of the system under test (SUT) is isolated using simulators. By replacing a hardware component by a software model, testing can be done without using the physical hardware component. The software model then fulfills the role of this hardware component in the system.

Such a software model describes the physical aspects of the hardware component in terms of the interactions between actuators and sensors in a system, as shown in Figure 1.5. An example of an interaction is the effect of a motor on its encoder. The movement of a motor is transferred to its motor encoder, which measures it. This transfer can be modeled in a simulator.

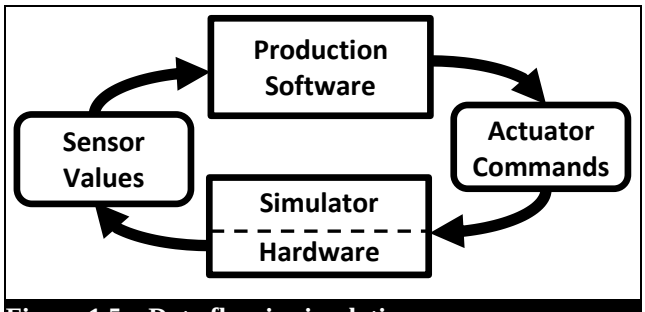

Figure 1.5 – Data flow in simulation

A specific type of simulator with fixed (as opposed to dynamic) behavior is called a *stub* in the company context. Such a stub implements one or more software interfaces but does not provide a model of the behavior of the simulated system. The term *stub* is often used loosely, as changing requirements can cause stubs to evolve into more complex simulators.

Approaches

We distinguish two approaches to simulation based on whether the entire hardware platform is simulated or only the mechanics. How these approaches map to the testing platforms in Section 1.1.4 is shown in Table 1.1.

The simulation approach in which the entire hardware platform is simulated is called Software-in-the-Loop (SiL) simulation. Because none of the actual hardware is required, the entire system software can be run as a software package on general-purpose hardware. This type of simulation is limited, because specific properties of the hardware platform, such as time behavior, cannot be simulated accurately.

Another simulation approach is Hardware-in-the-Loop (HiL) simulation. In this approach, the real hardware and software infrastructure are present in the system under test, but mechanical elements are replaced by simulation. Using HiL simulation, it is possible to execute tests on a system where certain components are not available.

Table 1.1 – Mapping between testing platforms and simulation approaches

Testing Platform	Simulation Approach
Devbench	Software-in-the-Loop
Testbench	Hardware-in-the-Loop
Proto	*None*

Benefits

Using simulation over testing on physical hardware provides the following benefits for the company:

- *Increased availability of systems for testing*
  Not many physical machines are available for testing. By supplementing these test environments with simulation, the number of engineers who can test at the same time can be increased significantly. Furthermore, waiting times for testing can be reduced drastically. By increasing the availability of the physical test environments, the required time for the software delivery process can be decreased.
- *Higher coverage of "bad weather" behavior*
  The behavior of the system can be influenced by problems in the system hardware. It is hard to verify that the software runs correctly under such conditions. By simulating these bad weather conditions, specific parts of the software that are hard to test can be evaluated.
- *Higher configurability*
  The machines that are available for testing represent a small subset of all machine configurations used by customers of the company. By providing a larger set of configurations through simulation, more problems can be prevented before the software is deployed at the customers.
- *Lower risk and cost of testing*
  The cost of the resources used for testing on the machine is high. Additionally, testing on a machine may lead to defects in the machine hardware, which adds to the cost and limits the availability.
- *Higher reproducibility of test results*
  Because of inaccuracies and noise in sensors, tests do not necessarily produce the same results in two identical executions. This complicates the tracing of identified problems. In simulation, it is more often possible to produce the same behavior in two identical runs.

SiL simulation using Devbench can give the most benefit during development. Because of its high availability, it allows software to be tested more often than when using one of the other test platforms. Because of this, the goal is to allow as much of the software as possible to be tested on the Devbench.

1.2 Outline

This document describes a method for the integration of simulators within the EUV Source. The first part of this report focuses on the project context. First, the various project stakeholders and their roles within the project are given in Chapter 2. After this, the opportunities are described in Chapter 3. Chapters 4 through 6 focus on the design itself, from requirements to design. The technical part of the report is concluded by Chapter 7. Finally, in Chapter 8, the project is described from a project management point of view. ■

2 Stakeholder Analysis

In order to understand the project context, it is necessary to be aware of the project stakeholders. This chapter gives an overview of these stakeholders, including their main interests in the project and possible conflicts in these interests.

2.1 Stakeholders

In the project context, we can identify three main groups of stakeholders. The first group of stakeholders is the university. The stakeholders in this group are interested in the process and design aspects of the project. The second group of stakeholders is the company. The main interest of these stakeholders is the business value of the project. The final stakeholder is the trainee. The main interest of the trainee is successfully completing the project as judged by stakeholders from the other two groups. An overview of the relevant stakeholders is given in Table 2.1.

Table 2.1 – Relevant stakeholders

Name	Role	Stakes
University		
Ad Aerts	3TU. Stan Ackermans, Software Technology Program Director	Ensuring program quality by: – Ensuring quality of the projects and their results – Maintaining good relations with high-tech companies in the Eindhoven area (including ASML)
Pieter Cuijpers	TU/e Project Supervisor	Ensuring the project follows a correct process and progresses at the expected pace Ensuring report quality on a content level
Name	**Role**	**Stakes**
Company		
Ernest Mithun Xavier Lobo	ASML Project Mentor	Ensuring the project follows a correct process and leads to a desirable result for ASML
	EUV Source Simulation Expert	Improving development time and code quality in the EUV Source by: – Improving the number of tests that can be run in simulation – Improving the availability of test environments – Determining the architectural approach to simulation
Dirk Coppelmans	ASML Project Mentor	See above
	EUV Source Test Architect	
Pieter Koper	Simulation Competence Owner	See above

Table 2.1 – Relevant stakeholders

Name	Role	Stakes
Ludovico Verducci	Main Developer / Maintainer Plasma Simulation	Changes in code of the Plasma Simulator should have: – high quality – clear documentation – minimal impact on existing design
Daniël Patty	Team Leader	Qualification of as many CPD applications as possible without the need for a machine
David Hols Reis	Engineer working with Simulator	Qualification of CPD applications
	Developer of camera simulation	Integrating camera simulation with plasma simulation and image processing to show that the simulator works
Qiaowei Zhang	Engineer working with image processing and laser focus	Having a highly available testing environment for software integration Increased capability of testing bad weather scenarios
Maarten Dam	Functional owner of CPD application	Successful qualification of the CPD application
Name	**Role**	**Stakes**
Trainee		
Tom Boshoven	PDEng Trainee	Acquiring the PDEng degree by finishing the project to a satisfactory degree (as judged by a committee) Acquiring technical knowledge as well as soft skills during the project

From an organizational perspective, three different groups of stakeholders exist within the company. Major stakeholders are located in the Architecture department, in the EUV Source Testing and Integration group, and in the EUV Source Embedded Software group. The first two of these groups contain a project mentor.

2.2 Possibly Conflicting Stakes

The major risk for conflicts between the interests of the stakeholders exists between the university and the company. This risk is based on the level of abstraction of the solution. The university stakeholders place most emphasis on accurate descriptions of a design for a generic problem and the process to achieve this design, where the company stakeholders put more emphasis on the design for the concrete case. Because of this, it is important to demonstrate not only that the design works as intended, but also that it provides value for the company.

This conflict is approached by focusing on the company value first. The business case is at the core of the project definition. The requirements and the design follow from this. As much as possible, a high-level design is made for a generic version of the problem. At the same time, detailed designs and prototype implementations are made for demonstrating the suitability of the design for solving the concrete problem. This way, the core interests of both parties are satisfied. ■

3 Opportunity Analysis

Combining the views of the major stakeholders, we can formulate the questions that play a central role in this project. We sketch the opportunities in the company context and summarize them in a concise problem statement.

3.1 Limitations in Current Simulation Solutions

Simulators are used for executing various test cases on components. These simulators are used for testing isolated parts of the system. Test cases that are not limited to these sets of components cannot be tested using these simulators, because not all interactions in the system are simulated. Executing system-wide tests using simulation requires simulation of three distinct elements:

- *Actuator / sensor interactions within a subsystem*
  Example: A motor encoder changing value based on the movement of this motor.
- *Simulation of the environment*
  Example: The hardware platform on top of which the software components are running.
- *Interactions between subsystems*
  Example: Multiple subsystems collaborate to create EUV light.

The interactions within a subsystem can be simulated by using existing simulators. The environment can be simulated on Devbench using an operating system abstraction layer in combination with a specific network configuration. The third element, which relates multiple subsystems, cannot be satisfied using the existing simulation approaches.

Because of this missing third element, various system-wide test cases can only be completed successfully on Proto. This provides an opportunity for improvement.

3.2 Opportunities

By allowing simulations to be combined into a single solution, it becomes possible to simulate more aspects of a system at the same time in a coherent way. This leads to an increased test coverage using simulation.

This is especially the case for integration tests. Whereas unit tests test for the correct behavior of the elements in a software component, integration tests focus on the interaction of the component with other components. To this end, a set of components is aggregated and the combined behavior of the components is validated.

The evaluated method for enabling testing of this type of tests is the integration of existing simulators. There are two main reasons for integrating existing simulators instead of designing a single simulator that simulates all parts of the system:

- *Domain knowledge is reused from the existing simulators.*
  Creating a simulator requires a significant amount of research in addition to the investment for design and implementation. Existing simulators already store the required knowledge. Furthermore, some of the knowledge that is present in existing simulators may no longer be present anywhere else in the company.
- *Separation of concerns is maintained*
  Although there is the added concern of the integration of the simulators and their combined behavior, the concern of the internal behavior of the simulators is not changed.

This leads to the hypothesis that combining existing simulators is an effective way of increasing integration test coverage.

3.3 Problem Statement

Evaluation of the opportunities leads to the following problem statement:

Achieving better integration test coverage in simulation in a software-only environment:

- *Is it possible using only existing simulators?*
- *How can we achieve it?*
- *What patterns and guidelines can be found?*
- *Does the result provide the expected benefits?*

3.4 Design Opportunities

The envisioned solution for this is an infrastructure in which multiple simulation solutions can be combined to increase integration test coverage. Each simulator provides a model for its own part of the system, but may rely on information that is provided by other simulators. Designing a solution in which simulators can exchange information helps achieve accurate simulation of the entire system.

Because various simulators exist, a generic approach needs to be designed for the integration of simulators, so this integration can be achieved in various parts of the system. This generic approach is the basis of the concrete designs of an integrated simulation solution.

3.4.1 Design Criteria

Several criteria play an important role in the design:

- **Genericity**
  The solution should be as generic as possible, so it can be applied to other use cases as well.
- **Realizability**
  One of the main goals of the concrete designs is to demonstrate the feasibility of the approach.
- **Documentation**
  In order to apply the approach in other use cases, it must be documented in an understandable way, so it is as easy to reproduce as possible.
- **Impact**
  The design is important in determining the approach for similar projects in the future.

The following two criteria play a less important role in the design:

- **Inventiveness**
  If an existing solution can be tailored to fit the problem, it is sufficient.
- **Complexity**
  While a solution of low complexity is preferred, this is less important as long as this complexity is managed correctly (e.g. through detailed documentation of the design and decision process).

3.5 Scope

The broad scope of this project is simulation of EUV Source software on a Devbench. The goal is to enable increased test coverage in the absence of hardware using simulators. This is part of a long-term roadmap for creating and implementing an architecture for simulation in the EUV Source.

In this project, the focus was put on two specific parts of the EUV Source. Instead of the entire EUV Source software, only simulation of plasma and simulation of the laser focus were evaluated. This leads to a smaller, short-term scope, which better fits

the duration of the project. The two main use cases that are part of this project (see Section 4.1) fall within this scope.

3.6 Success Criteria

This project is completed successfully when a generic approach for integrating existing simulators is found, enabling a higher integration test coverage in simulation on the EUV Source. This approach must be documented and demonstrated by means of concrete designs based on two main use cases, as well as prototype implementations.

3.7 Agreed Deliverables

Various project deliverables were agreed upon with the company stakeholders:

- Project plan
- Requirements document
- Design documentation
- Source code of prototype implementations
- Final presentation

The project plan, requirements document, and design documentation are combined in this final report.

3.8 Roadmaps

Within the company, a simulation roadmap for the EUV Source is being built. This roadmap relates use cases, required simulators, and communication flows between these simulators. This strongly relates to this project. Whereas the goal of this roadmap is to describe the necessary steps for implementing simulation solutions for various test cases, the goal of this project is to describe how to design these integrated solutions. ■

4 System Requirements

After evaluating the opportunities for improving the current solutions, we define a set of requirements for this improved solution. These requirements result from the domain analysis and the combined views of the various stakeholders. The requirements in this chapter form the basis for the design that is presented in later chapters.

4.1 Use Cases

The central use case in this project is the qualification of a CPD (Calibration, Performance measurement, Diagnostics) application. This type of application is used for automated calibrations of the machine.

Real hardware is required for testing these CPD applications. In order to test them on Devbench, the behavior of this hardware needs to be simulated. Solving the current limitations in simulating these aspects is the main opportunity in this project.

The central use case for this project is "Qualify CPD on Devbench." This use case exists only as part of a bigger process of which the end goal is full qualification of a CPD application. In the envisioned process, this qualification is done first on Devbench. If this is successful, it is attempted on Testbench, which provides a higher level of realism. Finally, qualification on Proto determines acceptance. The eventual goal is to find all issues on Devbench or Testbench, so qualification on Proto will be successful in the first attempt.

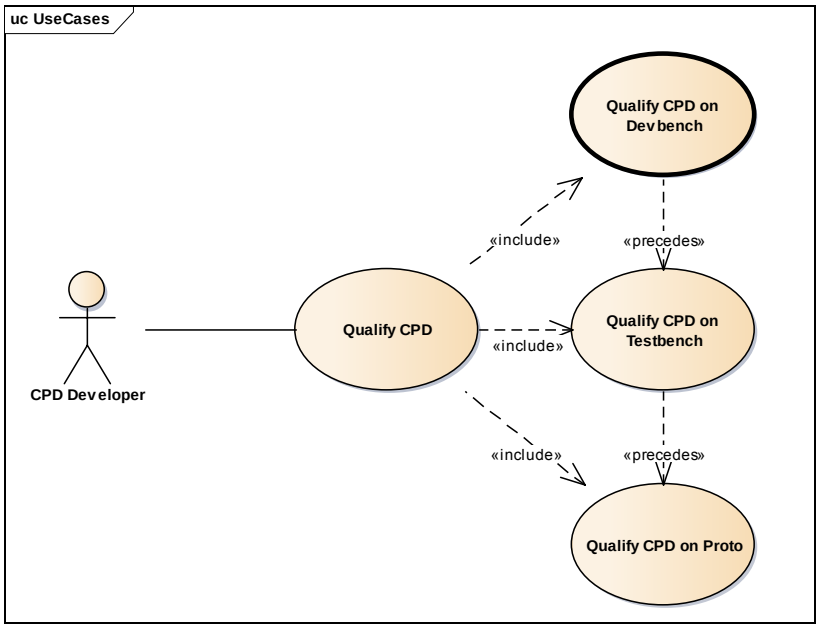

Figure 4.1 – Use case diagram for the qualification of CPD applications

Figure 4.1 gives an overview of the evaluated use cases for qualifying a CPD application (abbreviated to "CPD" in the diagram). Appendix A contains detailed descriptions of these use cases. Two specific CPD applications were chosen for specific instances of the use cases, based on their requirements.

The first CPD application ("CPD A") was chosen as a use case because it could already be qualified on a Devbench using high-level simulation. This use case was

used to show that the solution can be applied in an existing case and to demonstrate that integrating existing simulations can lead to higher code coverage in simulation.

The second CPD application ("CPD B") was chosen as a use case in order to demonstrate the genericity of the solution, by showing its applicability to a wider range of tests. This use case could only be qualified on a machine.

4.2 User Requirements

Combined with the use cases, there is a list of functional requirements for the solution. These requirements, which are listed in Table 4.1, describe the end goal of the project. They exist as a means to ensure that the delivered solution is valuable in the company context. The list of requirements is based on the use cases, and is created in dialog with the main company stakeholders.

Table 4.1 – List of functional user requirements and brief verification methods

Deployment		
UR000	The solution must be deployable on a Devbench	– Deploy on Devbench – Verify other requirements using this deployment
Integration		
UR100	The solution must successfully execute CPD A in a configuration in which a simulator is used for the laser focus, with the real driver software.	– Configure the solution to use a simulator for the laser focus. – Execute CPD A successfully using default settings. – Inspect the output to verify behavior is correct.
UR101	The solution must successfully execute CPD B in a configuration in which a simulator is used for the laser focus.	– Configure the solution to use a simulator for the laser focus. – Execute CPD B successfully using default settings. – Inspect the output to verify behavior is correct.
UR102	The solution must successfully generate camera images during the execution of CPD B.	– Configure the solution to use a simulator for the behavior of the laser focus and enable camera simulation. – Execute CPD B successfully using default settings. – Inspect the output to verify behavior is correct.
Configurability		
UR200	The solution must successfully execute CPD A in a configuration in which the driver stub is used for the laser focus.	– Configure the solution to use the driver stub for the laser focus. – Execute CPD A successfully. (Note: there is no support for CPD B in this stub.) – Inspect the output to verify behavior is correct.
UR201	It must be possible to deploy the simulators independently of each other.	– Successfully run smoke tests without running any other simulator.
UR202	It must be possible for a user to switch between a configuration using the driver stub and a configuration using a simulator for the laser focus.	– Verify UR100 and UR200 in a single sequence.

The verification methods in this table describe the basic steps for verification of a requirement.

If a requirement cannot be verified successfully using these steps, it does not necessarily indicate that the solution is incorrect. When verification fails, the cause must be explored manually. Assistance from the responsible software developers or mechatronics experts may be required for this. If the cause is determined to be a fault in an existing simulator or in production software, the problem must be reported. In such cases, the requirement can be considered not verifiable until the problem is resolved.

Table 4.2 – Prioritization of the user requirements

Must do	Should do	Could do	Would do
UR000 **UR100** **UR200** **UR201**	**UR101** **UR102** **UR202**		

The requirements are categorized by priority in Table 4.2. For this prioritization, the MoSCoW model is used. In this model, priorities are described from high to low by the letters M (*must do*), S (*should do*), C (*could do*), and W (*would do* or *won't do*). The set of *must do* requirements must be satisfied in order to complete the project successfully.

4.3 Nonfunctional Requirements

In addition to the functional requirements mentioned above, various nonfunctional requirements may affect the design. These requirements are based on the use cases and the views of the technical stakeholders. The main nonfunctional requirements were found to be as follows:

- **Configurability**
  It should be easy to select a configuration that suits the use case. This means that not only the type of simulation can be selected, but also the hardware configuration. How this configuration is done by the end user is not in the scope of this project, but care should be taken to allow this in the design. Through configurability of the solution, a trade-off can be made between coverage, accuracy, and performance.
- **Time Behavior**
  Although there are no real-time requirements, the solution should be fast enough. What this means in practice is that the expected performance varies per use case. Generally, the solution must not trigger software timeouts in addition to the timeouts caused by the simulators of which it is composed. Changes with a significant impact on performance should be evaluated with the relevant stakeholders.
- **Changeability**
  Changes in components and interfaces should not have a big impact on the solution. This keeps the overhead of simulation maintenance low, which is important to keep development cycles of the simulated products as short as possible.
- **Installability**
  The solution should be deployable with minimal effort and delay for a streamlined development and testing process. ■

5 Solution Direction

Based on the use cases and requirements, we determine a solution direction. This is achieved by describing the problem in the context of the system architecture and developing a generic approach based on this. By applying this approach to the specific test cases, we find high-level solutions to these cases.

5.1 System Architecture

In order to provide a generic solution for integrating simulators within the EUV Source, knowledge about the EUV Source software architecture is required.

5.1.1 Layers

EUV Source software follows a layered architecture in which each component is located on exactly one layer. An overview of the layers is shown in Table 5.1. Control flow is directed downward (same layer or lower), while measured data is directed upward (same layer or higher). The components that are important in the context of this project reside in the *CPD Applications*, *Metrology*, and *Subsystem* layers.

Table 5.1 – Software layers in the system architecture

Layer	Description
System Interfacing	Communication from the system to external systems
Application	Production application
CPD Applications	Applications used for configuration, performance measurements, and diagnostics
CPD Facilities	Generic facilities for CPD applications
System Control	Main sequencing of activities
Metrology	Modeling system settings
Subsystem	Control software for a subsystem
Domain Facilities	Generic facilities for a certain domain
Generic Facilities	Domain-independent facilities
OS & I/O Abstraction	Operating system and hardware abstraction

CPD applications communicate directly with one of the components on a lower layer (specifically the system control layer) to perform actions and measurements on the system. The CPD Applications layer is the highest layer that contains components that fit the scope of this project.

The Metrology and Subsystem layers provide the implementation of the high-level actions performed by the system. Where the components in the Subsystem layer are responsible for the behavior of a single subsystem, components in the metrology layer provide coordination of multiple subsystems.

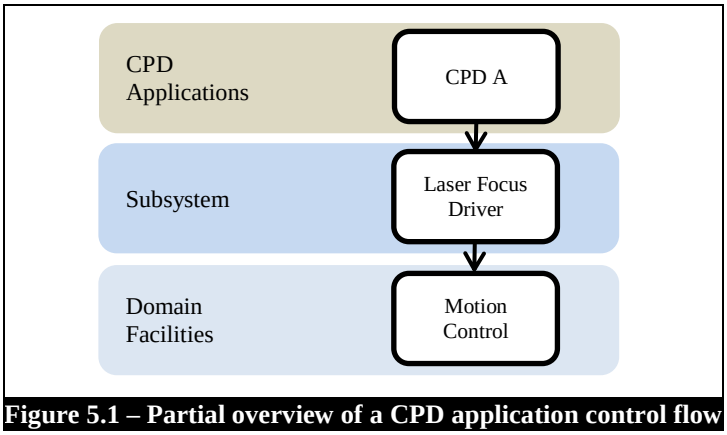

Figure 5.1 – Partial overview of a CPD application control flow

A partial overview of control flow from of a CPD application is shown in Figure 5.1. Most layers and subsystems are omitted for simplicity. The CPD application instructs a subsystem driver to change the focus position of the laser. This component then instructs Motion Control to actuate the hardware components to reach the desired beam focus position. It can be observed that the control flow is only directed to lower layers of the architecture.

5.1.2 Simulation

In simulation, the architecture is extended with simulator components. These components exist on the layer of the software component to which they are related. Table 5.2 lists various simulators with their architectural layers.

The architectural layer on which simulators are connected to production components is called the abstraction layer for the simulation. The abstraction layer separates the system under test from simulation. If the abstraction layer is high, test depth is low and therefore test coverage is potentially low. Much of the application logic is part of simulators. If the abstraction layer is low, test depth is high and therefore test coverage is potentially high.

Table 5.2 – List of software layers of simulators

Simulator	Software Component	Layer
Plasma Simulator		Metrology
Laser Focus Driver Stubs	Laser Focus Driver	Subsystem
Mechanical Simulator	Motion Control	Domain Facilities
Motion Control Stubs	Motion Control	Domain Facilities

Choosing the abstraction layer for simulation creates a separation between production software and simulation. In order to be able to capture as much of the production software as possible in the SUT (maximizing coverage), it is beneficial to use a low layer as an abstraction layer. However, using a low abstraction layer may increase complexity. Choosing an abstraction layer means striking a balance between coverage and complexity.

In the abstraction layer, control flow is directed from a production component to a simulation component. Between simulators on different architectural layers, the control flow is opposite to the control flow of production components on these layers. These simulators may direct control flow to a *higher* layer simulator. The resulting data flow goes in the opposite direction.

Figure 5.2 gives an example of this. It can be seen that in simulation, the control flow is directed from the subsystem layer to the metrology layer, which is higher in the system architecture.

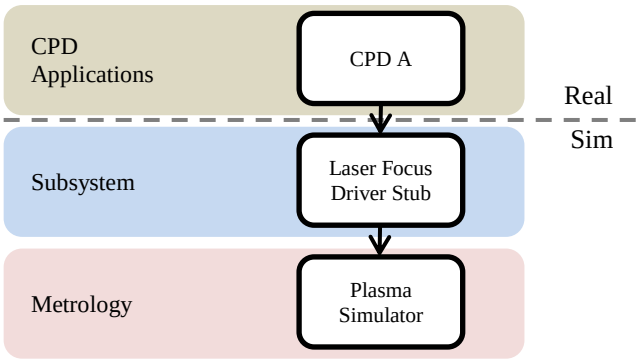

Figure 5.2 – Partial overview of a CPD application control flow using the Plasma Simulator

Figure 5.2 can be illustrated by an example sequence. The CPD application requires the laser focus position to be moved. To this end, it instructs the responsible subsystem to perform this task. The Laser Focus Driver Stub, which replaces the Laser Focus Driver, receives this request. It notifies the Plasma Simulator of the changed focus position, in order to allow it to compute new simulation output.

5.2 Generic Integration of Simulators

The first step in integrating simulation solutions is investigating the requirements of the integrated solution, based on a set of use cases. These requirements will help list the required behavior of the simulators. The reason for assessing these requirements is to investigate if they can all be satisfied using the available simulators. If this is not the case, it may be necessary to develop a new simulator or extend existing simulators in order to satisfy the requirements.

After this, a high-level design is created, which contains the production components as well as the simulators. In this design, care should be taken to choose the desired architectural layer as the abstraction layer. Generally, the abstraction layer that leads to the best test coverage is the *Domain Facilities* layer, which is the lowest domain-specific software layer. The interfaces in this layer are relatively close to the hardware. Choosing this layer includes most application logic in the System under Test, while abstracting from the underlying complexities of, for example, motion networks.

5.2.1 Interfaces

After determining which components should be part of the solution, the interfaces between the components should be determined. The interfaces between production components and simulators are the provided interfaces of the components at the abstraction layer. Reusing these interfaces for the simulators helps ensure that the production software behaves the same whether it is used with a simulator or not. Interfaces between simulators may be missing or incomplete.

An investigation is required to determine what information should be exchanged between simulators. This is based on the interactions between the physical elements in different subsystems that play a role in the evaluated use cases.

While interfaces between production software and simulators are production interfaces with actuators and sensors as important concepts, interfaces between simulators only describe physical data. The reason for this is that simulators base their output on the actual physical model instead of results of measurements of this model.

The directions of the interfaces (providing or requiring) are based on the architectural layers of the simulators. Components in lower architectural layers may not depend on components from higher architectural layers. Because of this, components on lower layers may implement interfaces from higher layers, but not the other way around. This principle holds for both production components and simulators.

As described in Section 5.1.2, this does not apply to control flow, which is mirrored for simulators. Control flow in simulation goes to higher architectural layers and resulting data flow goes to lower architectural layers. As a result, many interfaces have control flow that goes from the component that provides the interface to the component that requires it. Because there is no interface in this direction, the control flow cannot make use of direct calls to the components. Instead, indirect calls are required.

This can be achieved using the Observer design pattern [1], shown in Figure 5.3. In the observer pattern, a certain object (*ConcreteObservable*, also called *ConcreteSubject*) can have a number of associated objects (*ConcreteObserver*) that get notified of changes in this object through a call to their *update* method. The notified *ConcreteObserver* can then request the updated state of the *ConcreteObservable*.

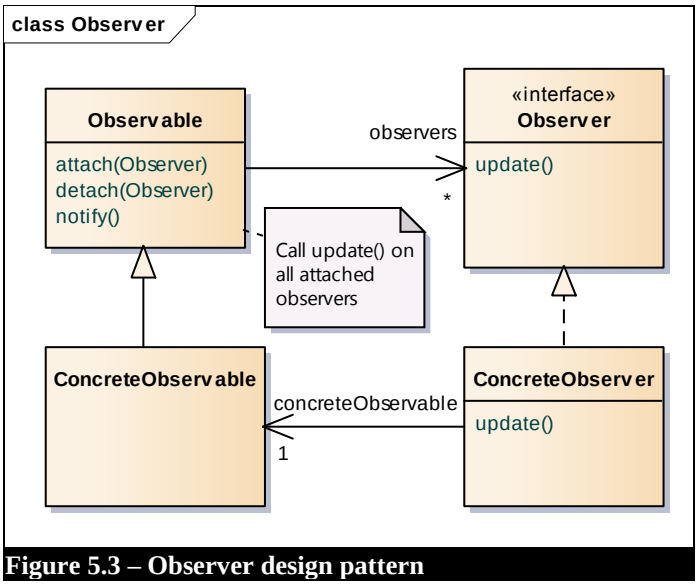

Figure 5.3 – Observer design pattern

5.2.2 Synchronization

When implementing an interface between simulators, it is important to consider how the simulators are synchronized. Synchronization problems may cause missed or delayed communication, which can lead to unreliable and hard-to-reproduce simulation results.

When deploying on the real-time Testbench platform, synchronization can be achieved using the task scheduler, because time behavior is predictable in real-time environments. However, when deploying on Devbench, which does not provide any timing guarantees, additional measures may be required.

Synchronization issues are likely to occur when an active simulator (which continuously updates based on clock ticks) has to react to incoming data. If this data changes multiple times between two clock ticks, not all data is observed by the simulator. Whether this causes issues depends on how this data is used by the simulator.

A possible solution to solving synchronization issues is the use of message queues in the communication channel. This ensures that all messages are available in the receiving simulator. Note that this may not be sufficient in all cases, because the data produced by the receiving simulator may be based on old inputs.

5.2.3 Data Flow

In addition to focusing on specific interfaces, it is necessary to consider data flow in the system as a whole. An important risk in on this level is deadlock or livelock due to cyclic data flow. This may happen, for example, when modeling a feedback loop between reactive simulators

Such a cyclic data flow can be solved in several ways. One of the options is making sure that the feedback loop always terminates, by defining error bounds. Another option is to throttle the communication, allowing higher-priority messages to be handled first. This can be achieved by using prioritizing event queues in the communication. A final option is to make sure that one of the simulators in the cycle is active.

5.3 Case A

As specified in the user requirements, the solution should enable the Laser Focus Driver software to be added to the system under test. A simulator is then needed to provide the underlying model. The Mechanical Simulator is the simulator that is used for testing the laser focus software on Testbench. In combination with the Plasma Simulator, it contains all required functionality for executing the CPD A use case. An architecture is presented that combines these simulators.

Figure 5.4 compares this architecture with the existing architecture, which makes use of the Laser Focus Driver Stub. The component structure of the simulation is changed in such a way that this stub is replaced by the real Laser Focus Driver software.

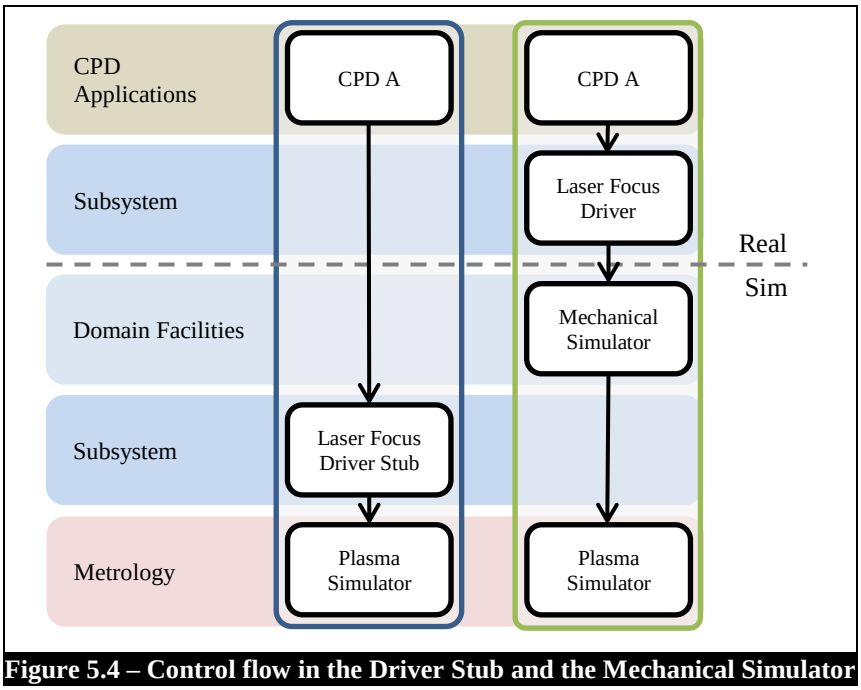

Figure 5.4 – Control flow in the Driver Stub and the Mechanical Simulator

5.3.1 Challenges

Integrating the described simulators leads to two main challenges. The first challenge is that the Mechanical Simulator is an active simulator, whereas the Plasma Simulator is reactive. This can lead to issues related to synchronization and performance. The second challenge is that the Mechanical Simulator does not provide an external software interface for retrieving simulation data. In order to integrate the simulators, these challenges have to be overcome.

Limitation

In a naïve approach to connecting the Mechanical Simulator to the Plasma Simulator, all output data from the Mechanical Simulator leads to an update within the Plasma Simulator. Because the mechanical Simulator is an active simulator, this output data is generated based on a clock frequency. Thus, in this solution, the Plasma Simulator would update based on this frequency, and become an active simulator. This is undesirable, because:

– The update frequency is very high. Plasma simulation requires time to complete, so this would have a strong impact on simulation performance.
– The Devbench platform does not support real-time behavior. Thus, timing is inherently unreliable. Time triggers may be delayed and the required information may not be propagated to the Plasma Simulator in time (before the CPD application measures it). This results in unreliable simulation results that are difficult to reproduce.

Simulation Interface

In order to execute CPD A successfully, information needs to be exchanged between the laser focus and the plasma simulation. In order to do this, an interface is required between these two simulators. None of the existing software interfaces can be used for this task because they are meant for use in production, and simulation may interfere with this.

Another option is the creation of a new interface. Several significant challenges are involved in this approach:

– The Mechanical Simulator is implemented as part of the Motion Control component, which is a production component. Adding an interface to this component adds to the complexity of the software and possibly leads to problems in production.
– The architectural guidelines do not allow the addition of an interface that is used exclusively for software testing to a component that is used in production because this leads to unnecessary risks. Furthermore, it introduces an additional maintenance and testing burden. Because it is used in production, this burden is heavier than for a component that is only used in testing.
– Due to very limited knowledge of the Motion Control component, a significant investment of time and effort is required to not only add an interface to this component, but also connect it to the internals of the component.

5.3.2 Solution Alternatives

Because of these challenges, a more refined approach is needed. In order to avoid synchronization issues, this approach has the following restrictions:

– The Plasma Simulator should remain reactive. This means that for any set of external inputs (triggers from production software), an upper bound can be placed on the number of times the Plasma Simulator is updated.
– All plasma state updates that result from a trigger from production software must be processed before the software measures the change.

This second restriction depends on the software that performs the measurement. In the evaluated use cases, no measurements are performed during motion. We choose to define the behavior during motion as undefined. Thus, as long as a motion is in progress from the perspective of the software, any measurement cannot be considered valid. During this time, the state of the plasma can be updated.

The advantage of doing this is that the system can be treated as fully reactive, because the system state is always stable at the time of measurement. The main disadvantage to doing this is that any measurements that *are* done during motion (for example, due to a bug) may lead to the wrong results, without being detected. Furthermore, in order to support future scenarios that require support for measurements during motion, the simulator design needs to be revised.

Various possible solutions were evaluated. Because the creation of a new simulation interface for the Mechanical Simulator was not considered feasible, one of the production interfaces was used. A filter component was evaluated, which evaluates the data to find out when to retrieve data from the system. Furthermore, a proxy element was evaluated, which evaluates internal system communication to find out when to retrieve data.

Filter

The first alternative is to introduce a lightweight "filter" element in the communication between the Motion Control component and the CPD application. This filter constantly reads the machine state from the Motion Control component. It then inspects the data to see if it is necessary to send an update to the Plasma Simulator.

The main challenge here is to define what the conditions for the update should be. Because of the low level of abstraction, no set of conditions was found that would lead to a bounded number of updates.

Furthermore, because the solution is not run in a real-time environment, it cannot be guaranteed that all generated information is read in time (before measurements are performed). This may lead to missed updates, which can ultimately lead to undefined behavior and lower reproducibility of results.

This alternative exclusively uses existing software interfaces and therefore requires no changes to production code. However, the implementation depends strongly on the domain, so it cannot be applied generically. Finally, it is unclear whether this alternative is feasible, and how robust the solution would be to future changes in the software.

Proxy

The second alternative is to implement a "proxy" element that listens to internal communication in order to time the updates. Whenever a command is finished, the proxy intercepts the completion message, reads out the required data, and forwards the completion message to its original destination.

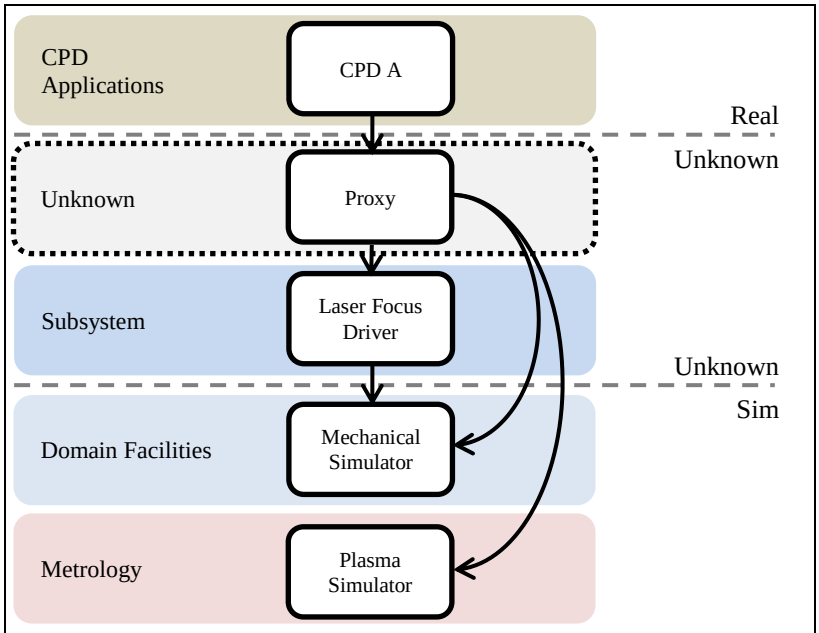

Figure 5.5 – Control flow in a deployment with a proxy component

This way, exactly one update is sent per trigger, and all updates are sent before motions are completed as observed by the software, because the completion message is delayed until after the update is sent.

This alternative leads to an architectural issue. Because the proxy, which is a simulation component, relays control flow directly to the driver, this driver must also be part of simulation, while the requirements state that this should not be the case. Figure 5.5 illustrates this by connecting the components based on control flow.

In relation to this, there is a dependency on the correctness of the intercepted messages. Because these messages come from the system under test, their correctness cannot be assumed. When these messages are not sent correctly, this problem may

not be observable in the simulation or the simulation may behave differently. This can lead to false positives in tests, or failures that are hard to trace.

Stubs

The third alternative is to abstract from Motion Control altogether. Instead of using the Mechanical Simulator, the Motion Control Stubs can be used. By attaching a reactive model to these stubs, it is possible to satisfy the technical requirements for CPD A relating to motion control.

Creating this model requires knowledge of the workings of the motion control, as well as a basic understanding of the laser focus hardware. However, this investment results in a simulation that does not require changes in production code and is easy to extend to support other use cases, such as the CPD B use case.

By adding a model to these stubs, they can no longer be regarded as stubs. In the remainder of this document, this solution is referred to as Motion Simulator.

Comparison

Table 5.3 gives an overview of the presented solutions and compares them based on several criteria. These criteria were selected based on the specific qualities of the various solutions.

Table 5.3 – Comparison of approaches for case A

Criterion	Interface	Filter	Proxy	Stubs
Approach is generic	✓		✓	✓
Data is guaranteed stable	✓		✓	✓
Allows full test coverage of subsystems	✓	✓		✓
Subsystem driver not in simulation	✓	✓	?[2]	✓
Only minimal system knowledge required			✓	
No modifications to production code		✓	✓	✓
No communication from subsystem level	✓	✓		✓
Extendable model				✓

The filter solution is disregarded based on its specific nature. The additional interface was not considered feasible due to its impact on production code. A detailed design and prototype implementation was made of the two remaining alternatives. Section 6.2 describes the design of these alternatives in detail.

5.4 Case B

In order to execute the CPD B use case successfully in simulation, it is necessary to simulate optics within the system. Because no simulator exists for this, a simulator needs to be created. This is done by extending an existing simulator to support this use case. Because of its extensibility, the Motion Simulator, which is the simulator based on the Motion Control Stubs, is used. By extending the model of this simulator, the optical properties that are required by CPD B can be simulated.

5.4.1 Camera Simulator

In addition to being passed to the CPD application, the data resulting from the optical model is used as input for the Camera Simulator. The Camera Simulator provides a means of visualizing the simulated data in the form of realistic camera images. These images can be viewed to evaluate the behavior of the CPD application.

[2] Because the proxy is higher in terms of communication flow and this proxy could be considered a simulation component, the driver can be considered to be running in simulation.

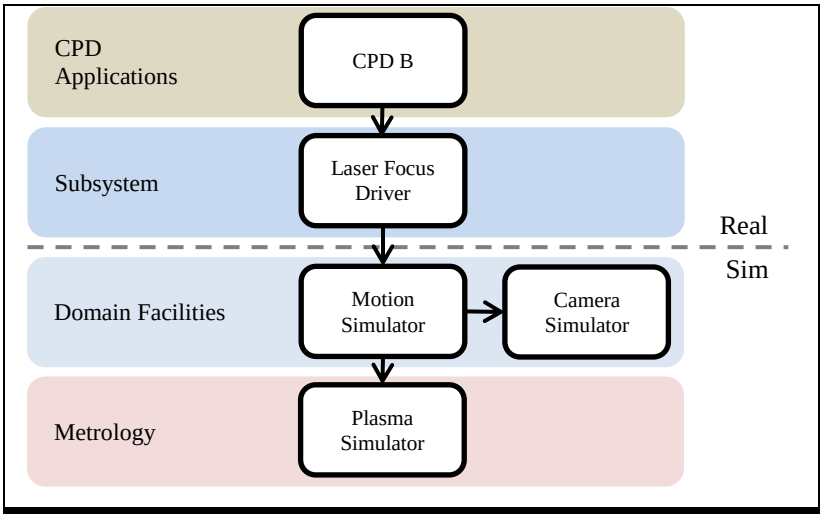

Figure 5.6 – Control flow after the integration of the Camera Simulator

Figure 5.6 shows the control flow in a deployment that includes the Camera Simulator. The Camera Simulator is connected to the local file system. This way, camera pictures get stored in permanent storage, which is useful for validation and demonstration purposes.

Because the camera exists on the same architectural level as Motion Control, the Motion Simulator is allowed to depend on the Camera Simulator and control flow is allowed to go from the Motion Simulator to the Camera Simulator. Because of this, it is not necessary to reverse the dependency as in the integration with the Plasma Simulator.

Both the Motion Simulator and the Camera Simulator are reactive, so no synchronization issues need to be taken into account. No cyclic data flow is occurring, so this is ignored at this stage. This leads to a simple design in based on direct communication using the existing interface of the camera simulation. ■

6 System Design

After applying the generic guidelines on the specific use cases, forming a basis for the solutions, we evaluate how these solutions can be designed.

6.1 Introduction

Based on the high-level architecture that is described in Chapter 5, the various components and interfaces that play a role in the system are designed. The designs are separated in the two use cases.

6.2 Case A

Out of the four evaluated architectural options for integrating simulation of the laser focus with plasma simulation, two were selected for a detailed design. The first option that was chosen uses the Mechanical Simulator for simulation and updates the information in the Plasma Simulator by intercepting high-level messages. The second option that was chosen does not use the Mechanical Simulator but uses the Motion Control Stubs instead. Proper simulation is achieved by attaching a model to these stubs. These two alternatives lead to different designs.

6.2.1 Proxy Approach

In order to update Plasma Simulator at the right moment (right after a motion completes, before measurement starts), a proxy class is used. This is an application of the proxy design pattern [1].

The central concept in this pattern, shown in Figure 6.1, is the proxy class. This proxy class, which implements the same interface as the subject, is responsible for delegating calls to the subject. By extending the proxy, it is possible to add additional behavior to some of the calls. For example, it can modify the parameters before delegating the calls, it can have additional behavior before or after delegation, and it can even choose not to delegate certain calls.

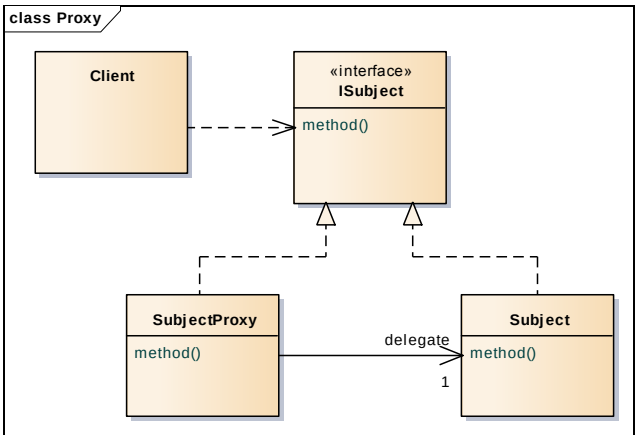

Figure 6.1 – Proxy design pattern

In the case of the Laser Focus Driver, the real driver (*Subject*) and the proxy (*SubjectProxy*) implement the same interface (*ISubject*). Communications with the client (CPD application) are directed to the proxy object. This is done transparently; it has no impact on the client code. The proxy delegates all calls to the real driver unmodi-

fied. Some of the calls have the side effect of triggering the simulator to perform a state update.

Communication

The behavior of the proxy is shown in Figure 6.2. After completion of certain high-level operations (such as initialization and movement of the laser focus point), the proxy is triggered to generate an update. This update is generated before the CPD application is notified that the focus position was changed. This is done to prevent changes to the state of the motion while the update is in progress, thereby eliminating possible synchronization issues.

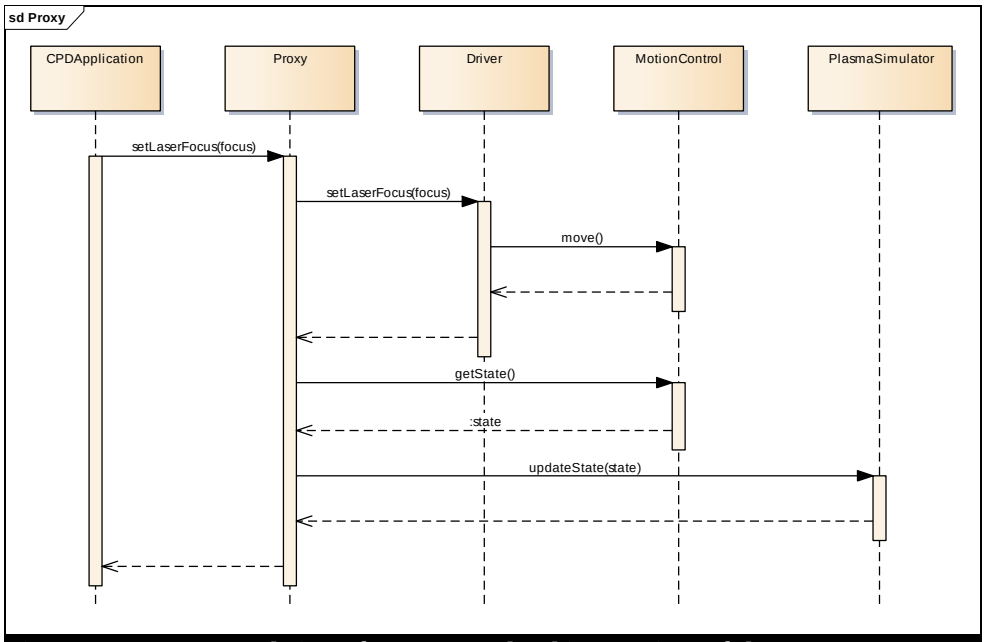

Figure 6.2 – Conceptual view of component-level interactions of the Proxy component

6.2.2 Stubs Approach

Instead of intercepting high-level behavior, calls to the Motion Control component may be used. This is done by means of the Motion Control Stubs. Figure 6.3 describes interactions that lead to a state update in the Plasma Simulator.

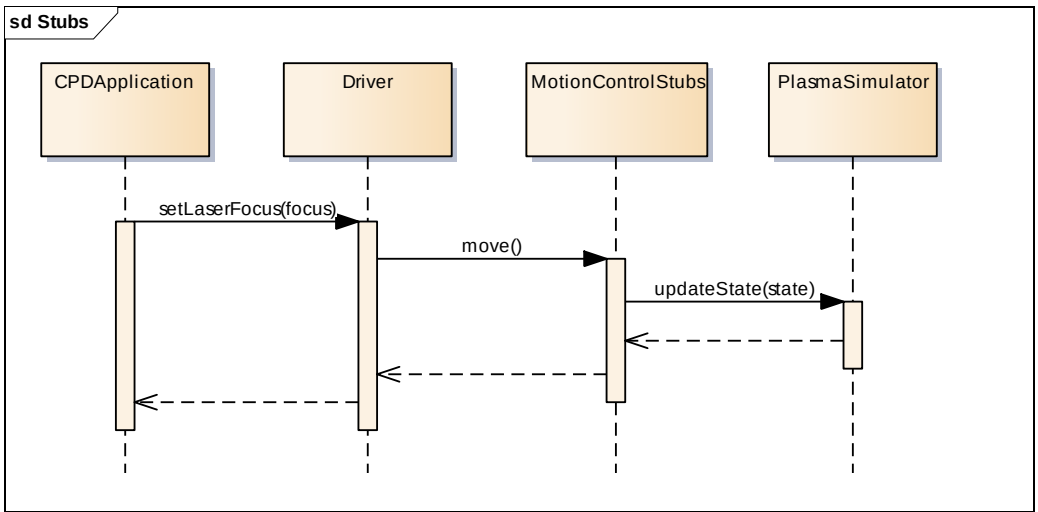

Figure 6.3 – Conceptual view of component-level interactions of the Motion Control Stubs

Model

For simulating motion, a simple model is used. In this model, all motions are instantaneous. The timing that is involved in motion is not modeled. For example, if an

actuator is instructed to move to a certain position within three seconds, the actuator is immediately changed to this position (ignoring the desired timing). This model is sufficient for the use cases in the scope of this project.

Observer

From Figure 6.2 it can be seen that a direct call is going from the stubs to the Plasma Simulator. Therefore, the Plasma Simulator must provide an interface to the stubs, which are on a lower architectural layer. This is undesirable for reasons described in Section 5.2.1. This dependency is reversed by introducing the Observer design pattern.

After applying this pattern, the Plasma Simulator becomes a listener to state changes in the Motion Control Stubs. After each motion, an update is triggered in the Plasma Simulator, allowing it to update based on the new state.

Adapter

In order for the state of the Plasma Simulator to update correctly, it must react to the updates from the Motion Control Stubs. However, the (observer) interface provided by these stubs does not match the interface that is used to update state in the Plasma Simulator. In order to connect the two interfaces, the adapter pattern [1] is applied. This pattern, shown in Figure 6.4, adapts the interface of an object (*Adaptee*) to another interface (*ITarget*) using an additional class (*Adapter*).

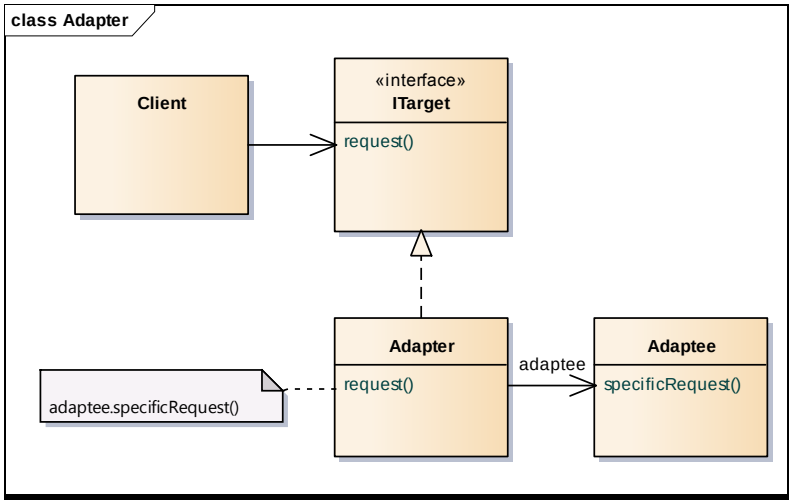

Figure 6.4 – Object adapter design pattern

In the specific case of the stubs, the *ITarget* interface is the observer interface as described in the previous section, the *Adaptee* is the class that is used for updating the Plasma Simulator, and the adapter is a new class that translates an update from the Motion Control Stubs into an update in the Plasma Simulator.

Extension Options

The Motion Control Stubs currently exhibit only simple behavior. In order to cover more use cases, its model can be extended to form a complete reactive simulator for the Motion Control component. The interface to the Plasma Simulator is designed such that this type of extensions does not require changes in the interface. Additional features such as error injection require this interface to be extended.

6.3 Case B

In order to support case B, simulation of optics is embedded in the Motion Control Stubs. This is done by extending the simple model from Section 6.2.2 with a more advanced model that allows various types of interactions between components, turning the Motion Control Stubs into a complete simulator called Motion Simulator.

This section restricts itself to describing the design of the extended model, without describing the optical model that is implemented as part of this design.

6.3.1 Simulation Model

The design of the model, an overview of which is shown in Figure 6.5, is based directly on the domain model for simulation, as presented in Section 1.1.5. The model consists of model elements (sensors and actuators) and interactions between them. Based on the existing interfaces, actuators are modeled as physical actuators with small control systems. As such, they have a position, a setpoint, and a trajectory end position. Sensors only have a value.

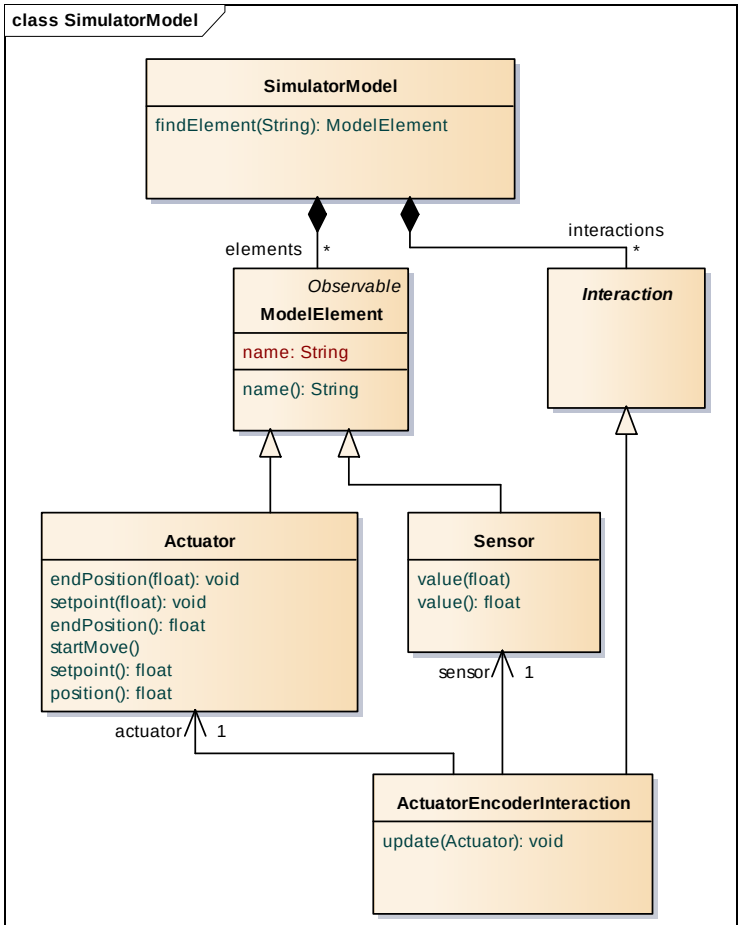

Figure 6.5 – Design of the simulator model

A specialized interaction (*ActuatorEncoderInteraction*) exists to allow the sensors to measure the actuator positions. In order for a model element not to be dependent on its interactions, the observer pattern is used to notify an interaction when the state of one of the model elements on which it depends is changed.

Figure 6.6 shows a sequence of interactions that shows how the observer pattern is deployed in this interaction. When the actuator is requested to move, its setpoint and position are updated. When this happens, all listeners, including the interaction, are updated. The interaction, after being triggered, simply makes the value of the sensor equal to the actuator position.

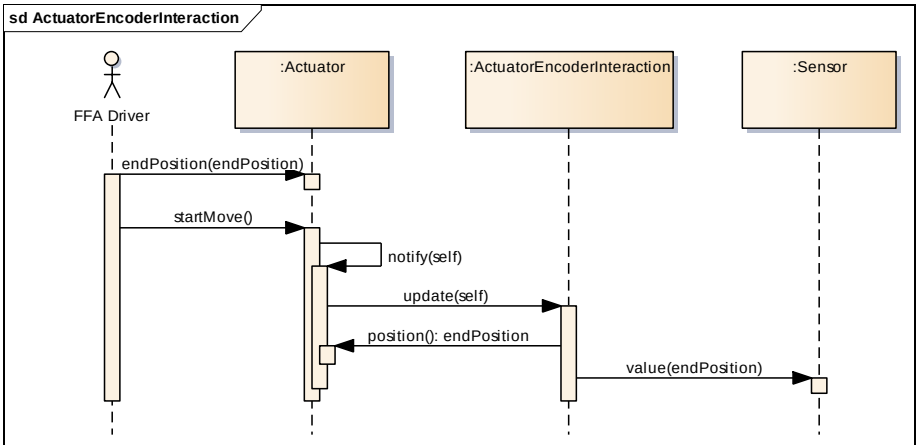

Figure 6.6 – Sequence diagram demonstrating the behavior of an interaction

Although various actuators, sensors, and interactions exist within the system, only a few are modeled, in order to support the current use cases. The model is extendable with more elements to support other use cases.

Behavior

While the simple model that is described in Section 6.2.2 is sufficient for the use cases in the scope of this project, future use cases will require different behavior. Such behavior can include a model of motion that is more accurate with respect to time, or a model that includes broken sensors for testing bad-weather behavior. In order to support this type of extensions, the behavior of the model elements is separated from the state of the model elements. This generic concept allows the behavior of a model element to be changed based on a use case. For example, scenarios with broken sensors can be supported by replacing the behavior of a sensor with the behavior of a broken sensor.

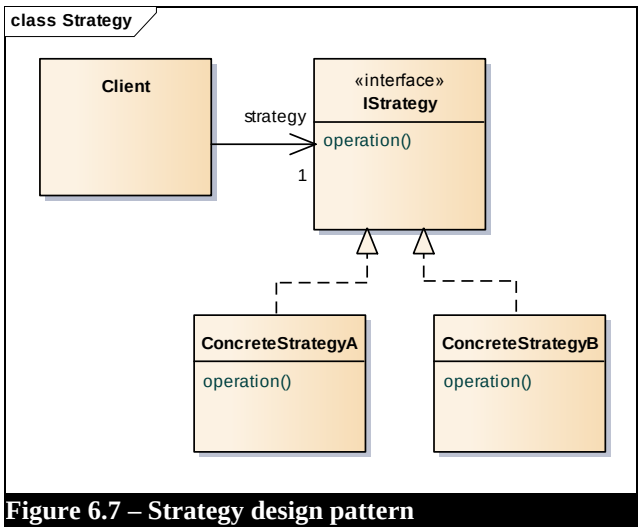

Figure 6.7 – Strategy design pattern

This separation is done by means of the strategy design pattern [1], shown in Figure 6.7. This pattern allows behavior to be defined dynamically. Here, the *Client* is one of the model elements, and each *ConcreteStrategy* contains a description of the behavior of this model element. When requested to perform a certain operation, the model element delegates the request to its associated behavior.

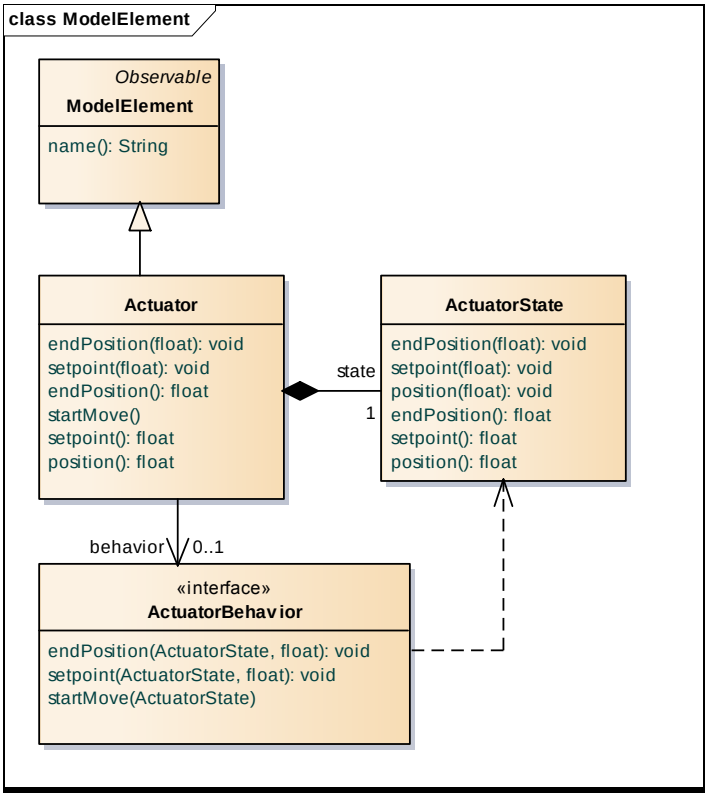

Figure 6.8 – Actuator model element with separated behavior

The application of the pattern results in a structure as shown in Figure 6.8. While this figure shows only the actuator model element, this structure is used as a generic template for the other model elements. In this figure, the *ActuatorBehavior* is the strategy interface (*IStrategy*) and classes implementing this interface describe the concrete behavior. The state is stored in a separate *ActuatorState* object, which is modified only by the concrete behaviors.

The implementation of a model element is separated into commands and queries. Commands are delegated to the behavior, but queries are executed directly on the model. Figure 6.9 shows how this applies to actuators. In this figure, the only behavior for the end position is to change the state, but other types of behaviors may work differently.

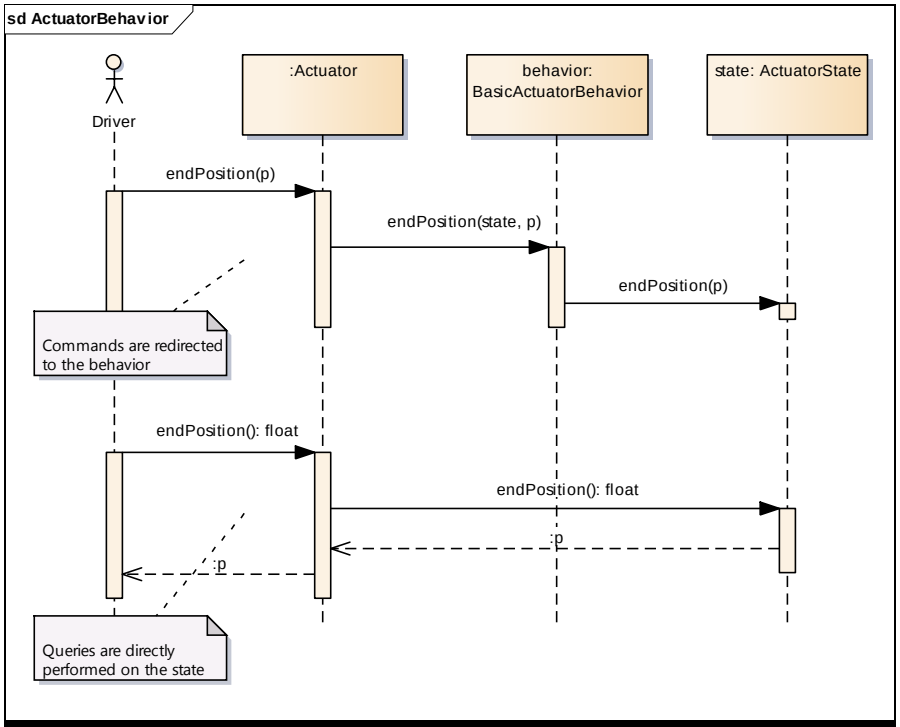

Figure 6.9 – Commands versus queries in Actuator class

An example of a class that implements the *ActuatorBehavior* interface is the *BasicActuatorBehavior* class, which captures the model that is described as part of Section 6.2.2, in which:

- an actuator setpoint is equal to its physical position at all times
- a motion is an instantaneous operation that makes the setpoint and position of an actuator equal to the end position of the motion

This is achieved by providing an implementation of the required *ActuatorBehavior* methods in this class that reflects this model.

Construction

Construction of the behaviors is done by means of the abstract factory pattern [1]. This pattern, which is shown in Figure 6.10, allows a client to instantiate objects without specifying their concrete type. The abstract products in this pattern relate to model elements, such as actuators and sensors.

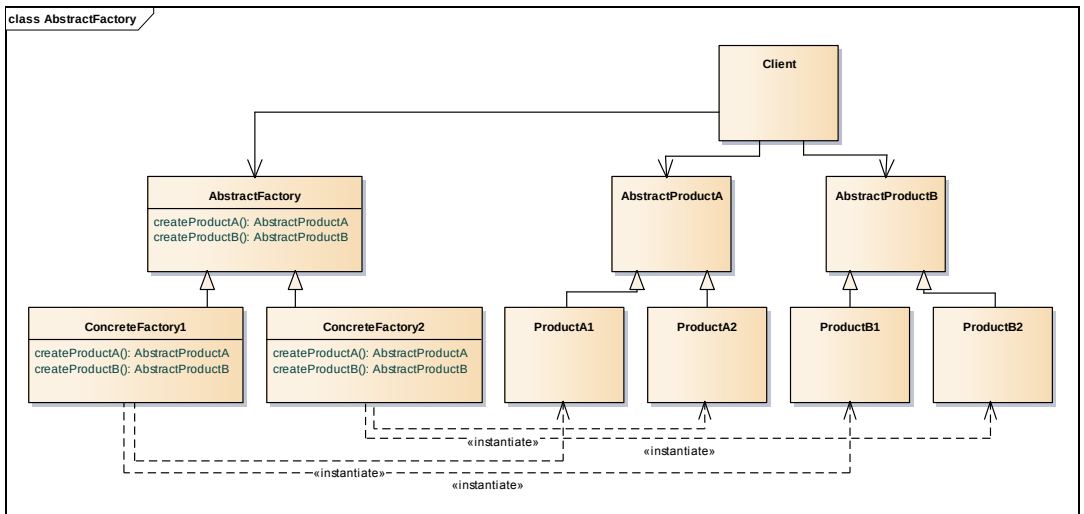

Figure 6.10 – Abstract factory design pattern

For the behaviors, this pattern is applied as in Figure 6.11. By implementing the *BehaviorFactory* interface, a class can determine what type of *ActuatorBehavior* and *SensorBehavior* can be created. Because currently only the *BasicActuatorBehavior* and *BasicSensorBehavior* are defined, the *BasicBehaviorFactory* is the only shown factory.

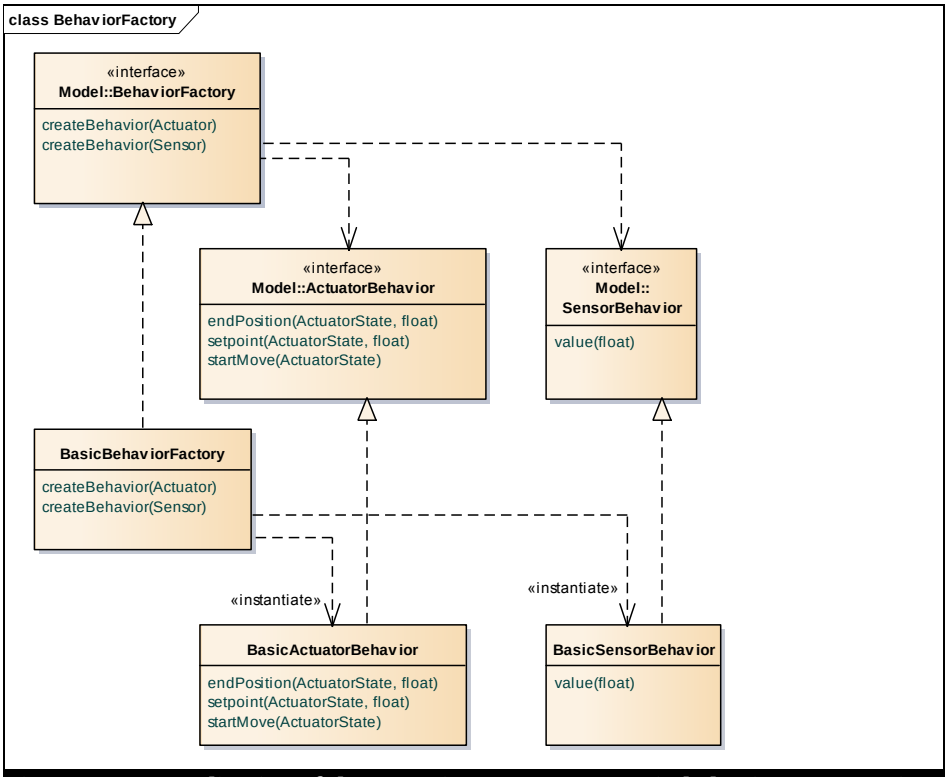

Figure 6.11 – Application of the Generic Factory pattern in behavior construction

Packages

The described simulation model is separated into a number of packages. The overall architecture is separated into a model, view, and controller (MVC architecture), reflected in the names of the packages. In total, five packages are part of the Motion Simulator:

- **Model**
  This package contains the model elements, combined with their states and (abstract) behaviors. In addition to this, it contains the available interactions and the *SimulatorModel* class, which models the entire system by instantiating and connecting the available model elements and interactions.
- **View**
  This package contains the interface that is used for observing the model state. It is used to connect the Motion Simulator to the Plasma Simulator.
- **Controller**
  This package provides the software interface to which the production software connects.
- **Observer**
  This package is a utility package containing the required classes for the observer pattern. This package is reusable between components.
- **BasicBehavior**
  This package provides an implementation of the behavior of the model elements. While this is currently the only defined behavior, it is possible to create other types of behavior by implementing the same set of interfaces from the Model package.

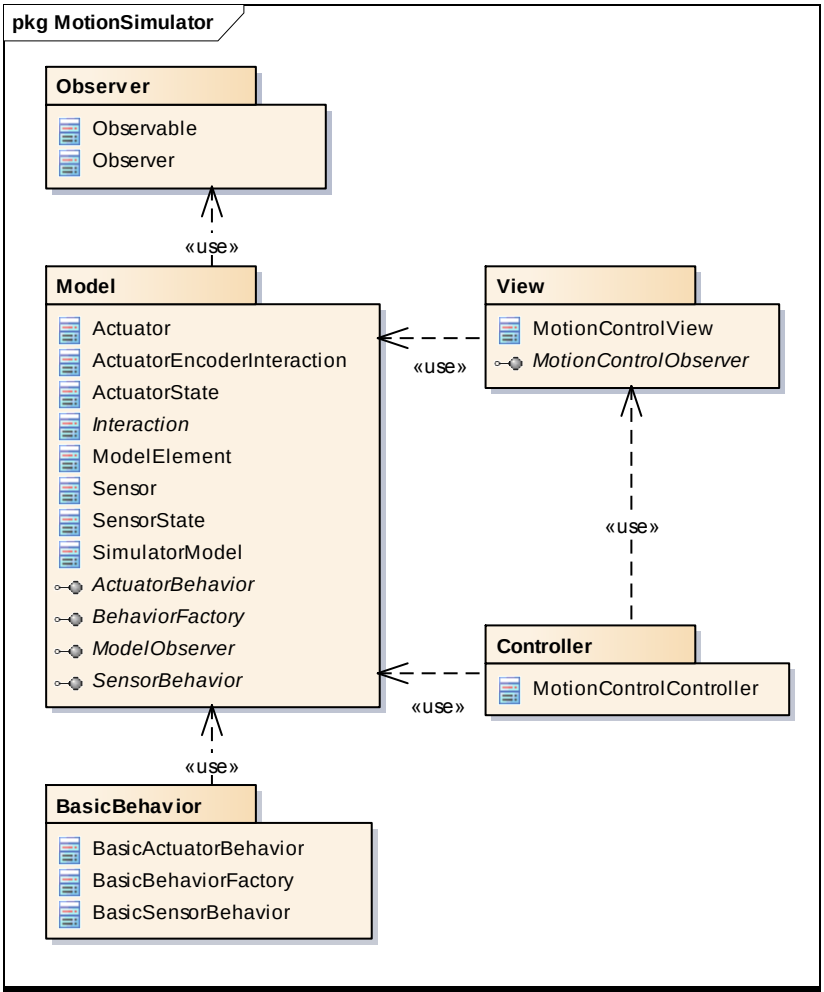

Figure 6.12 – Package diagram of Motion Simulator

Figure 6.12 shows the relations between the packages. The shown dependencies between the *Model*, *View*, and *Controller* are commonly found in MVC architectures. The *Basicbehavior* package provides an implementation of several interfaces in the model and therefore depends on the *Model* package.

The additional model that is required for the CPD B use case is designed within this framework by extending the Model package with additional model elements and interactions. This extended model provides all functionality that is required for the CPD B use case. ■

7 Conclusions

The detailed designs were implemented as prototypes. By applying them in the use cases, we show that they provide the expected benefits. We then finish by giving possible directions for future work that builds on these results.

7.1 Results

A generic approach to the integration of simulators in the EUV Source software was given. This approach was applied to two use cases, in the form of the validation of CPD applications. Doing this resulted in concrete designs and prototype implementations of these designs.

These designs make use of existing simulators where possible. Extensions of these existing simulators were provided to support the test requirements for which simulation was not available, such as simulation of the optical effects that were required for the CPD B use case.

The prototypes were verified by executing the use cases on the prototype solutions. This gave varying results. The CPD A use case was completed successfully, showing that the design leads to working software. The first part of the CPD B use case was also completed successfully, showing that the integration of simulators leads to improved testing capabilities.

Failure of the second part of this CPD application helped identify concrete problems in the functionality of the CPD application itself. These problems, which could otherwise only be found on an actual machine, were easy to identify using the camera simulation. By finding these issues early, on a highly available test platform, it was demonstrated that this method of simulation provides the expected benefits.

7.2 Future Work

Because it was shown that the given approach can be used for the integration of simulators and can lead to better integration test coverage using Software-in-the-Loop simulation, most future work is directed towards applying this approach in other use cases.

In the first place, the current designs can be extended to allow for full qualification of CPD B, as the development of this application is continued. In addition to this, extensions can be made for other CPD applications for which SiL simulation is considered appropriate.

In addition to the subsystems that were in focus of this project, other subsystems can also benefit from this approach. For example, applying the approach to CPD applications for the tin droplets could improve testing in this subsystem.

Another possible extension is mapping this approach to HiL simulation. This is expected to lead to similar benefits as on SiL simulation, but allows better testing of integration with the hardware platform and real-time behavior. This would contribute to the intended testing process, which is shown in Section 4.1. The intention is to qualify the software of the EUV Source using a combination of testing with SiL simulation, HiL simulation, and on a real machine.

A final future direction is related to the design of a simulator model. Part of this design of the simulator model is a template for elements of a simulator model, based on the separation of state and behavior. In order to facilitate the creation of these model elements, code generation can be used. By generating these elements from a

high-level description of a model, only the behavioral aspects of the simulation (interactions and behaviors) have to be provided. Supporting this type of workflow in the creation of simulator models to support new use cases is future work. ■

8 Project Management

In this chapter, we describe the project from a project management perspective. We focus on the used methods, as well as the risks and challenges during the project.

8.1 Introduction

One of the characteristics of this project is that it combines knowledge about the system architecture with detailed knowledge about the evaluated subsystems and their software implementations. In order to manage acquiring this knowledge, the project was divided into a number of milestones, each preceded by its own domain and risk analysis. Based on the results of this analysis, the exact target for the milestone was determined by aligning the views of the main stakeholders.

The user requirements and overall project plan were used as a basis for this target, but the analyses determine the actual steps. This approach is robust against dealing with unknowns by allowing milestone targets to flexible based on the learned information during the moments where this information is most relevant.

Each milestone builds upon the solution design and contains a prototype implementation of the design. The design is part of this document and the code of the prototypes is delivered documented and tested as part of this project.

The project is separated into two phases, each with a focus on a specific use case. The first phase contains two milestones and the second phase contains three milestones.

8.2 Work Breakdown

Figure 8.1 shows the work breakdown structure of the project. This figure shows how the design aspect of the project is split into two phases, both consisting of milestones.

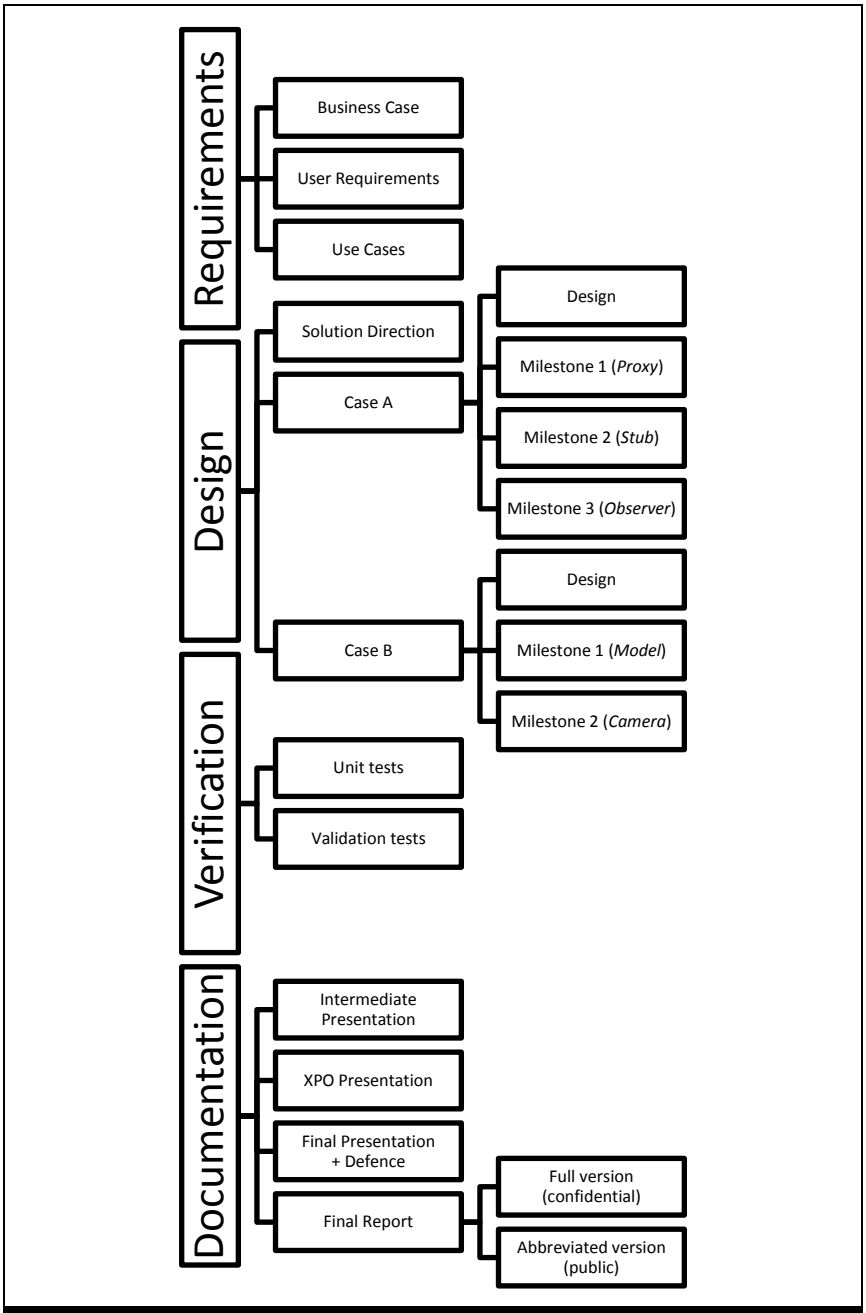

Figure 8.1 – Project work breakdown structure

8.3 Project Planning and Scheduling

The overall project schedule is shown in Figure 8.2. This schedule marks several important dates in the project, including the ends of the two phases and the final presentation.

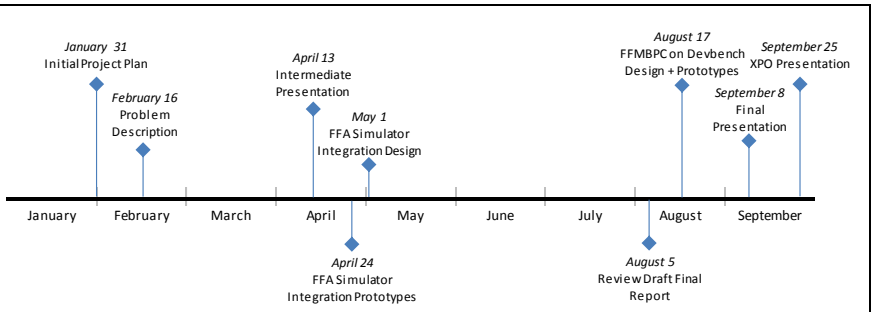

Figure 8.2 – Project timeline containing important dates

Appendix B contains the project Gantt charts. This chart divides the work over the available time. The allocated time slots leave a buffer before deliveries. This allows for delays in deliveries. The time allocation evolves during the project, as more data such as deadlines and presentation dates is available. This results in shifts and changes in the length of the allocated time slots.

It should be noted that the global project plan does not show deadlines. Instead, it shows when each work item will be worked on. At the end of such a period, the item may not be 100% completed. This may happen because, for instance, the item requires verification by other parties. At the end of the period, the required effort before release should be minimal, and the risks for a work item should be addressed.

The overlapping time slots allow for a risk-based time division. In such a case, time is allocated to both items, where the item with the highest risk gets priority.

It can be seen that the final weeks of the project are empty. This time is used for finalizing activities such as support and knowledge transfer. Furthermore, it allows room for extensions to the prototype implementation if needed.

8.3.1 Evolution of Planning

Over the course of the project, several changes were made to the original planning. This original planning is shown in Appendix B.1. Apart from small shifts, caused by new information regarding deadlines for deliverables, there are changes in the milestones. The main difference is that the first phase has one missing milestone, and the second phase has one extra milestone.

This change is caused by the changes from the analysis phases. While it was originally expected that a simulator could be integrated in a single, larger milestone, an investigation led to two distinct approaches. Both were executed as separate milestones, and both could be executed in the original time span set for the first phase.

Similarly, in the second phase, the original expectation was to enable image processing in the first milestone. After analysis, it was found that this was not required in order to satisfy the requirements. Instead, more effort was required in the other elements, due to the missing optical model.

The final change is the distinction between the public and confidential report, which was not present in the original planning.

8.4 Deliverables

The list of deliverables for the project is given in Table 8.1. Because of the possible confidentiality of some of the contained information, public deliverables require a review by the company's Technical Publications Board (TPB) before delivery. The submission date for this review is denoted by the internal deadline. For most public deliveries, an additional week is added on top of the regular two weeks for the TPB review in order to allow processing of the feedback from this review.

Table 8.1 – List of project deliverables

Deliverable	Confidentiality	Deadline (internal)	Deadline (external)	Notes
Initial Project Plan	Confidential	2015-01-30	2015-01-30	
Requirements Document	Confidential	2015-03-05	2015-03-05	*Part of Final Report*
Intermediate Presentation TU/e	Public	2015-03-16	2015-04-13	*Requires TPB Review*
Initial Performance Evaluation	Public	2015-03-31	2015-03-31	*Submitted to OOTI before third PSG Meeting*
Intermediate Performance Evaluation	Public	2015-06-02	2015-06-02	*Submitted to OOTI before fifth PSG Meeting*
Design Document	Confidential	2015-09-21	2015-09-21	*Part of Final Report*
Abbreviated Final Report	Public	2015-08-11	2015-09-01	*Submitted to exam committee after TPB review* *No concept version because of TPB screening*
Draft Final Report		2015-07-14	2015-07-28	*Requires TPB confidentiality check* *Submitted for technical writing review by Judith Strother*
Concept Final Report	Confidential	2015-09-01	2015-09-01	*Submitted to exam committee* *Deadline: 7 working days before Final Presentation*
XPO Presentation	Public	2015-09-03	2015-09-25	*Requires TPB Review* *To be determined not later than the beginning of September*
Final Presentation	Confidential	2015-09-08	2015-09-08	
Final Report	Confidential	2015-09-21	2015-09-21	*After changes proposed at Final Presentation* *Deadline: 3 days after Final Presentation*
Article Project Booklet	Public	2015-09-06	2015-09-20	*Requires TPB review*

8.5 Risk Management

Table 8.2 describes a set of risks for the project, including mitigation strategies. The risks with the highest magnitude (probability multiplied by impact) are at the top of this table.

Table 8.2 – Evaluation of project risks

Threat	Consequences	Strategy	Method
1. A required	Not all requirements	Avoid	Evaluate candidates early.

Table 8.2 – Evaluation of project risks

Threat	Consequences	Strategy	Method
component cannot be integrated because of technical reasons or scope.	are satisfied.	Control	Negotiate whether the integration is essential.
		Accept	Develop a version of the component that can be integrated.
2. A component cannot run on the required system.	The component cannot be integrated.	Avoid	Make an inventory of the components and try to run them on the system.
		Accept	Allocate time for (partial) porting of the components.
3. The main source of information on a component becomes unavailable for an extended period.	Part of the system may have to be reverse engineered, which requires additional time.	Avoid	Find at least one other source for the required knowledge, if possible.
		Control	Get most of the required information as early as possible.
4. Priorities of requirements from stakeholders change. *Example*: Stakeholders may want to deploy on Testbench as opposed to Devbench.	The wrong problem is solved.	Avoid	Have a view of the long-term goals and discuss the priorities beforehand in order to be able to anticipate.
		Control	Discuss implications and feasibility within the project and change task priorities accordingly.
5. New requirements arise.	The solution is not changed with the problem, which may lead to results of little value.	Avoid	Have a view of the important stakeholders and on how the system is going to be used.
		Control	Discuss feasibility, priority and change project plan accordingly.
6. Hidden complexities occur in the design.	More time is needed to implement prototypes.	Avoid	Ask experts on the current architecture about the problems they think will occur.
		Accept	Make sure enough time is allocated to the implementation steps.
7. A supervisor is unavailable for an extended period.	An important party is not supervising process and progress, so the project may move in the wrong direction.	Control	Discuss a replacement supervisor, preferably with knowledge about the project.
8. Office space at the company is lost.	Access to my stakeholders or sources of information is limited.	Control	Discuss contact methods for important stakeholders.

Table 8.2 – Evaluation of project risks

Threat	Consequences	Strategy	Method
9. Data is lost.	Part of the project is undocumented.	Avoid	Save all data in a location that is automatically backed up through the company backup process. The "How to Secure Data" intranet page describes where to store which types of files.
10. Documents or presentations are not approved for publication.	Deliverables cannot be made in time or presentations cannot be given.	Avoid	Plan enough time for modifications after review, before the deadline.
		Control	Leave time to switch presentation dates with a colleague.
		Accept	Make a confidential version and an abridged public version, with most of the details removed.
11. Required software becomes unavailable. *Example*: Build server, Devbench	Code can temporarily not be written or tested.	Avoid	Be aware of planned downtimes and plan activities accordingly.
		Control	In case of an extended period, negotiate about making the tooling (or similar tooling) available for this project.
		Accept	Plan a buffer period for this type of problems.

During the project, some of these risks occurred. Risk 1 occurred in the integration of the Mechanical Simulator. Multiple approaches were evaluated and eventually, another candidate was chosen. Risk 3 occurred with during integration on the Plasma Simulator side. Some reverse engineering was done to understand how data is generated in this component. Because this was caused by a hidden complexity (risk 6), avoidance was not possible. This hidden complexity occurred despite the avoidance step of consulting experts because no expert has knowledge about both the Plasma Simulator and CPD B.

8.6 Conclusions

Early on in the project, a strategy was devised to deal with acquiring knowledge throughout the project. This led to a risk-oriented process that is flexible, in order to accommodate new knowledge, and involves the important stakeholders in decisions about the project direction.

Although this process resulted in a project with a difficult-to-predict course, the focus on risk and benefit for the stakeholders in the decision process ultimately led to a solution that not only satisfies the main requirements, but also gives valuable insights into the current software architecture. ■

9 Project Retrospective

At the end of the project, we look back on the design opportunities, to evaluate their role in the final design. Finally, we reflect on the project as a whole.

9.1 Design opportunities revisited

The main design opportunity in this project was the creation of an architecture that combines the capabilities of multiple simulators by integrating them. This was addressed in both a generic and a specific way, where the latter was shown to be an answer to the problem statement.

In the design, the focus was put on the four design criteria of genericity, realizability, documentation, and impact.

The genericity criterion was addressed by the generic solution. The approach and guidelines in this solution are generic to the EUV Source software, and can therefore be applied to other use cases within this system as well.

The realizability criterion was addressed by the specific solutions. By making designs that focus on specific use cases, it was possible to make prototype implementations to demonstrate feasibility.

The documentation of the approach is part of this document. The solutions are documented at a level that is expected to be understandable for a software engineer or architect working on the EUV Source. This facilitates the application of the documented approach in other parts of the EUV Source.

The impact criterion was addressed by demos of the prototype implementations. Showing that this approach leads to software that can be used easily with existing test scenarios helped demonstrate that the integration of simulators is a feasible and useful method for increasing integration test coverage.

9.2 Reflection

As in many projects in new environments, I spent a lot of time in this project on acquiring the required domain knowledge, getting familiar with the way of working within the company, and finding connections within the company. In order to do this, regular conversations with the various stakeholders were important. For this reason, it was useful to have an overview of the project stakeholders.

In addition to the information relating to the company and the problem domain, information was needed about the system. One of the things that helped me a lot in understanding the system architecture was the creation of the prototype implementations. Because these prototypes make use of various real software components, I developed new insights into the system works by building these prototypes. This hands-on experience with the architecture during the design phases contributed strongly to my understanding of the way the system works as a whole, leading to a better overall design.

During the project, I did not only obtain knowledge that applies specifically to this project. I also developed knowledge and skills in the general topics of simulation and the integration of components within a software architecture. On the organizational side, the main skills I practiced were working with risks and unknowns, as well as communication with stakeholders.

An important realization that I will take away from this project is the importance of *demoability*: the appeal of a product in a demonstration. In the initial stages of the

project, most presentations were accompanied by abstract figures and explanations, which were not easy to follow for some of the stakeholders. In later stages, demos of the prototype implementations helped illustrate the concepts behind the design, while involving the audience more.

In the final stages of the project, we chose to integrate the Camera Simulator into the existing solution. The main reason for this choice was to improve demoability. After creating a prototype implementation including this Camera Simulator, the demos started to look more impressive. This made it easier for the audience to see the added value of the project, leading to enthusiastic reactions from various stakeholders.

This enthusiasm was important because of the intention to use the results of this project in the company. As the project reached its final phases, people gradually became convinced this is a good approach to testing using simulation.

The final prototype for the second phase was evaluated by two of the eventual users of the simulation solution, to get feedback from their experience. They could execute most of the steps in simulation as they could on the actual machine, and the results of the simulation looked realistic. Both users found the prototype very helpful in finding issues in the software without needing the actual machine.

During these evaluations, an actual problem was identified in the functionality of the software. This helped further underline the usefulness of this type of testing during the development process. By finding this issue and having the capabilities of testing the fixed software, a lot of the testing time on the machine can be prevented, which gives significant benefits to the company and validates my work in this project. ■

Abbreviations

Abbreviation	Meaning
ASML	*Not an abbreviation*[3]
CPD	Calibration, Performance Measurement, Diagnostics
EUV	Extreme Ultraviolet
HiL	Hardware-in-the-Loop
IP	Image Processing
MoSCoW	Must do, Should do, Could do, Would *(or Won't)* do
OOTI	Ontwerpersopleiding Technische Informatica *(see ST)*
OS	Operating System
PDEng	Professional Doctorate in Engineering
SiL	Software-in-the-Loop
ST	Software Technology
SUT	System under Test
TU/e	Eindhoven University of Technology

[3] ASML was originally a joint venture between the Dutch companies Advanced Semiconductor Materials International (ASMI) and Philips. The L in its name stands for Lithography.

Glossary

Term	Description	Section
Active Simulator	Simulator that updates its model based on time events	5.2.2
Adapter	Design pattern that maps an interface onto another interface	6.2.2
ASML	Company in Veldhoven that produces machines for photolithography	1
Beam	*See Laser*	
Camera Simulator	Reactive simulator capable of producing camera pictures	5.4.1
CPD Application	Application used for calibration, performance measurement, or diagnostics purposes	4.1
Devbench	Testing platform used with software-in-the-loop simulation within the company	1.1.5
EUV Source	System producing EUV light by pointing a laser at a droplet of molten tin	1.1.2
Hardware-in-the-Loop	Testing method in which mechanics, but not electronics, are replaced by simulation	1.1.5
Laser	Powerful, directional beam of light, input to the EUV Source	1.1.2
Lithography	*See photolithography*	
Mechanical Simulator	Active simulator of motions required for laser focus	5.3
MoSCoW	Method for prioritizing requirements	4
Motion Control	Component for controlling motions for laser focus	5.1
Motion Control Stubs	Stubs for the Motion Control component	
Motion Simulator	Reactive simulator; extension of Motion Control Stubs with a model	5.3.2 6.3.1
Observer	Design pattern in which an object subscribes to updates of the state of another object	5.2.1
Ontwerpersopleiding Technische Informatica	*See Software Technology*	5.3.2
Photolithography	Engraving patterns using light	1.1.1
Plasma Simulator	Reactive simulator that simulates the generation of plasma	5.1.2
Proto	Complete machine used for testing	1.1.5
Proxy	Design pattern in which an object intercepts communication with another object	5.3.2
Reactive Simulator	Simulator that updates its model based on new inputs	5.2.2
Scanner	*See Wafer Scanner*	
Software Technology	Designers' program resulting in PDEng degree	
Software-in-the-Loop	Testing method in which the hardware platform is replaced by simulation	1.1.5
Stub	Simulator with fixed (as opposed to dynamic) behavior	1.1.5
System under Test	System that is being tested for correct operation	
Testbench	Testing platform used with hardware-in-the-loop simulation within the company	1.1.5
Wafer Scanner	Machine that engraves patterns on silicon wafers, used to create integrated circuits	1.1.1

Bibliography

[1] E. Gamma, J. Vlissides, R. Johnson and R. Helm, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education, 1994.

About the Authors

Tom Boshoven received his Bachelor of Science degree from the Department of Mathematics and Computer Science of Eindhoven University of Technology in 2011. After this, he continued studying and received his Master of Science degree in Computer Science and Engineering from the same university in 2013. During his studies, he specialized in formal systems analysis, in particular the model-checking problem. Throughout his studies, he played a role as a software developer for robot soccer team Tech United. During the last year of his studies, he worked part-time as a software developer for an online video platform. After moving to New York City for three months to continue his work for this company as a full-time software engineer, he came back to follow the Software Technology program.

A Use Case Descriptions

This appendix contains detailed descriptions of the use cases presented in Section 4.1.

A.1 Qualify CPD

Two specializations exist of this generic use case. These specializations only influence the scope of the test. They are therefore not described separately.

Level	User-Goal
Scope	CPD Application
Brief Description	The CPD application is qualified as efficiently as possible by first qualifying it on a Devbench, then a Testbench, and finally Proto.
Primary Actor	CPD Developer (*DEV*)
Stakeholders and interests	Developer and owner of the application want to release the application as early as possible. Part of this process is qualifying it.
Precondition	CPD Application is implemented and a Test Performance Specification (TPS) is available.
Minimal Guarantees	None
Success Guarantees	CPD Application is fully qualified.
Main Success Scenario	1. *DEV*: Qualify CPD on Devbench 2. *DEV*: Qualify CPD on Testbench 3. *DEV*: Qualify CPD on Proto
Extensions	1. A. Qualification on Devbench is unsuccessful. *DEV*: Evaluate the problem, fix the implementation or test plan, and restart scenario. 1. B. Qualification on Devbench is not possible due to missing essential functionality in a simulator. Skip to step 2. 2. A. Qualification on Testbench is unsuccessful. *DEV*: Evaluate the problem, fix the implementation or test plan, and continue with step 1. 3. A. Qualification on Proto is unsuccessful. *DEV*: Evaluate the problem, fix the implementation or test plan, and continue with step 1.
Comments	If qualification on Testbench or Proto is unsuccessful (2.A., 3.A.), it can be useful to evaluate why the problem was not detected in the earlier steps. This way, failures can help improve simulation. Similarly, if qualification cannot be performed on Devbench or Testbench, it can be useful to add the missing functionality to these environments, so they can be performed in these environments in the future.

A.2 Qualify CPD on Proto

Level	User-Goal
Scope	CPD Application
Brief Description	The test plan is executed on a Proto machine.
Primary Actor	DEV (CPD Developer)
Precondition	CPD Application is implemented and a Test Performance Specification (TPS) is available.
Minimal Guarantees	None
Success Guarantees	CPD Application is fully qualified.

Main Success Scenario	1. *DEV*: Request access to Proto and wait for availability. 2. *DEV*: Install a patch containing the CPD Application on Proto. 3. *DEV*: Execute steps as described in TPS.
Comments	The TPS for the application describes under which conditions qualification is successful.

A.3 Qualify CPD on Testbench or Devbench

These two scenarios are described together, as they are very similar.

Level	User-Goal
Scope	CPD Application
Brief Description	The test plan is executed on a Testbench or Devbench using simulation.
Primary Actor	DEV (CPD Developer)
Precondition	CPD Application is implemented and a Test Performance Specification (TPS) is available.
Minimal Guarantees	None
Success Guarantees	None
Main Success Scenario	1. *DEV*: Request access to the environment and wait for availability. 2. *DEV*: Install a patch containing the CPD Application on the requested environment. 3. *DEV*: Execute steps as described in TPS.
Extensions	1. A. A step in the TPS fails. *DEV*: Log the issue and continue executing the described steps to find more issues. The scenario fails.
Comments	The waiting time in step 1 is short (<1 minute) for Devbench, and may be longer for Testbench.

B Project Planning

This appendix contains Gantt charts showing how the project milestones are distributed over the available weeks.

B.1 Original Planning

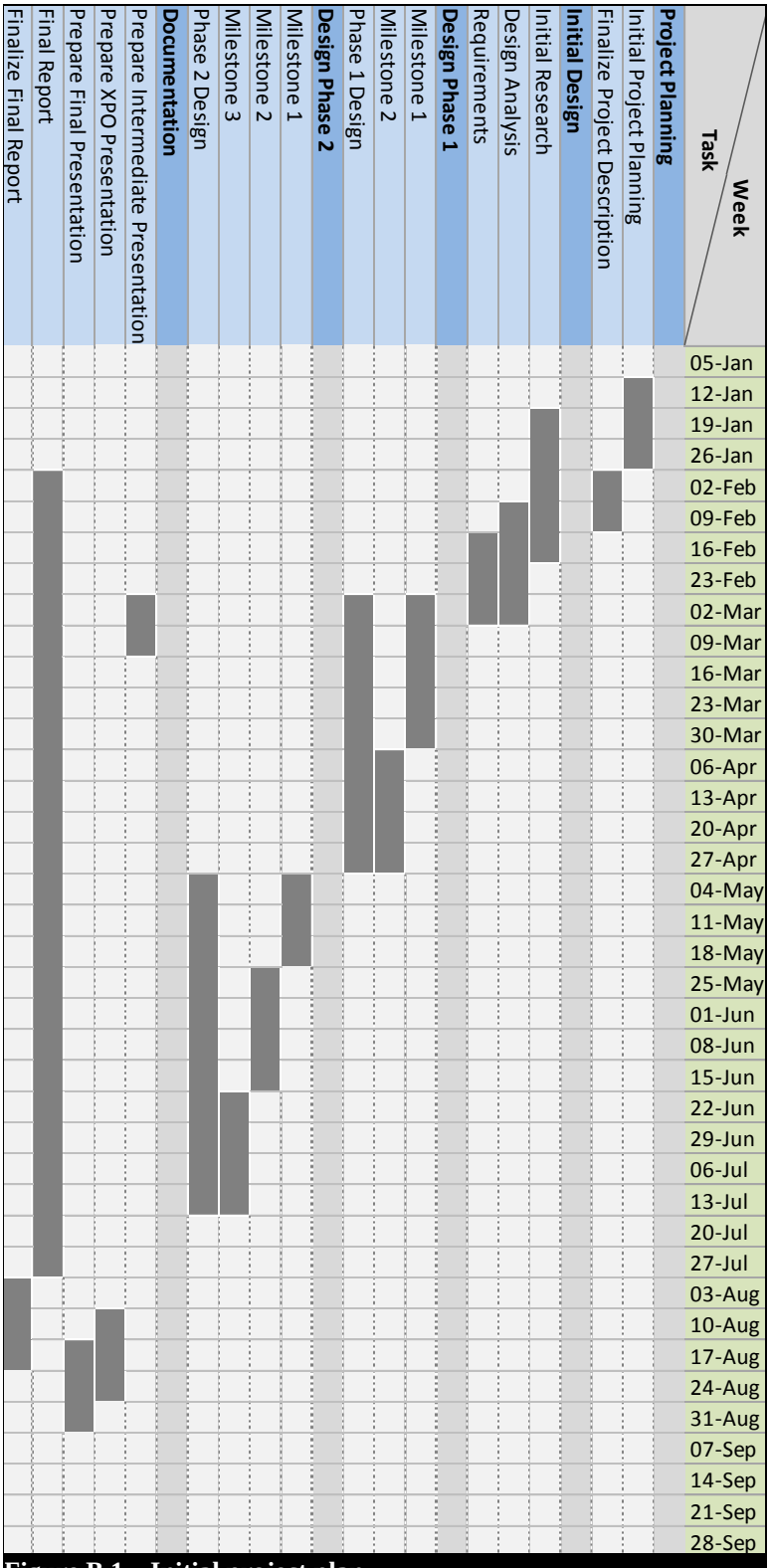

Figure B.1 – Initial project plan

B.2 Final Planning

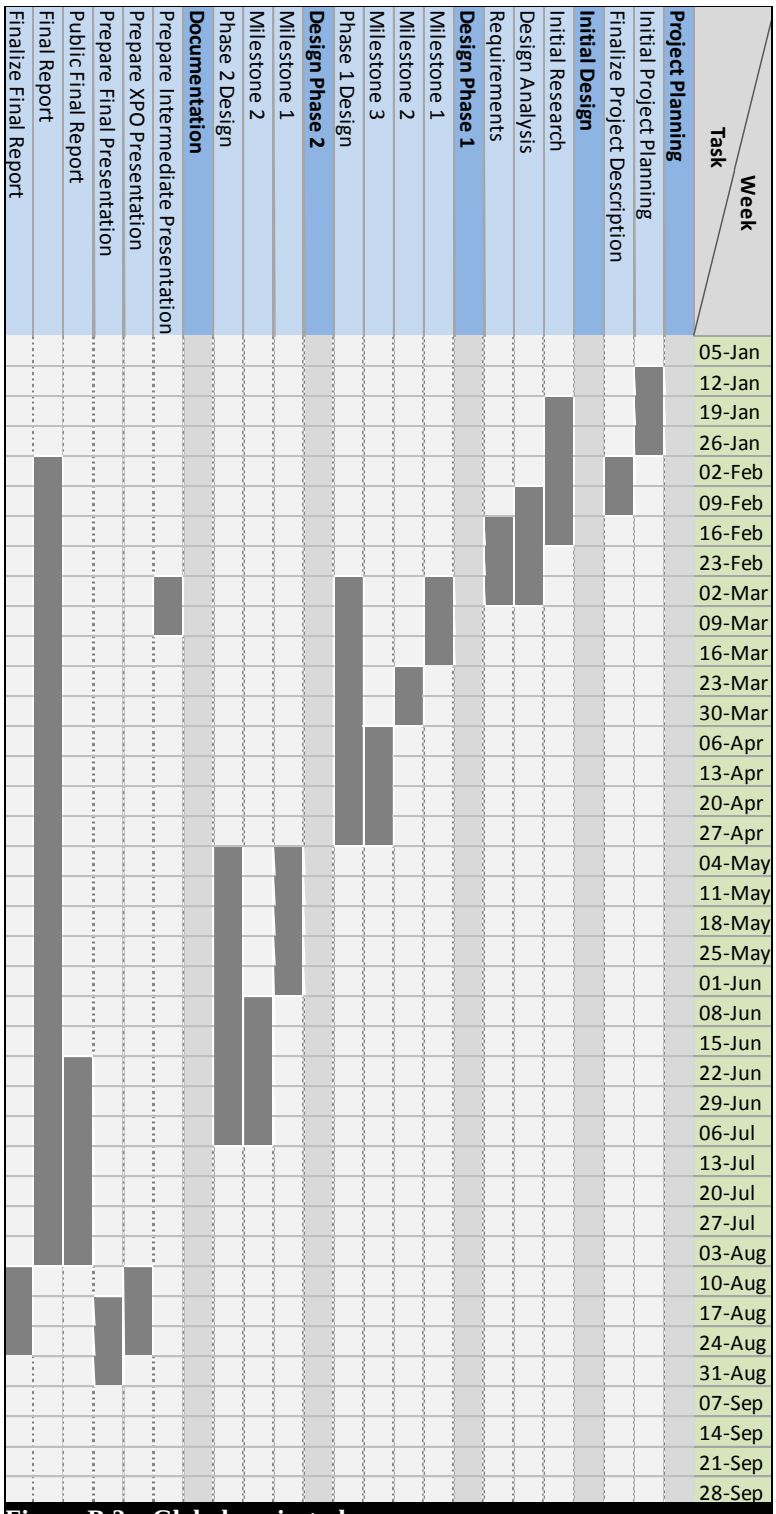

Figure B.2 – Global project plan

3TU.School for Technological Design, Stan Ackermans Institute offers two-year postgraduate technological designer programmes. This institute is a joint initiative of the three technological universities of the Netherlands: Delft University of Technology, Eindhoven University of Technology and University of Twente. For more information please visit: www.3tu.nl/sai.