ASML simulation development environment

Citation for published version (APA):

Document status and date:
Published: 28/09/2016

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 18. Sep. 2019
ASML Simulation Development Environment

Irina Nikeshkina
September 2016
The design described in this report has been carried out in accordance with the TU/e Code of Scientific Conduct.
Contact
Eindhoven University of Technology
Address
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
SAI report no.
Eindverslagen Stan Ackermans Instituut; 2016/047

Abstract
The final project was devoted to designing a development environment that solves ASML simulator creation problems by enforcing unified simulator architecture, reducing the engineering effort via code generation, and can be used by both software and functional engineers for simulator design.

Keywords
Simulator, Model-Driven Engineering (MDE), Domain-Specific Language (DSL), Domain-Specific Modeling (DSM), Meta Object Facility (MOF), and Code generation

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

Disclaimer
Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the Eindhoven University of Technology or ASML. The views and opinions of authors expressed herein do not necessarily state or reflect those of the Eindhoven University of Technology or ASML, and shall not be used for advertising or product endorsement purposes.

Disclaimer Liability
While every effort will be made to ensure that the information contained within this report is accurate and up to date, Eindhoven University of Technology makes no warranty, representation or undertaking whether expressed or implied, nor does it assume any legal liability, whether direct or indirect, or responsibility for the accuracy, completeness, or usefulness of any information.

Trademarks
Product and company names mentioned herein may be trademarks and/or service marks of their respective owners. We use these names without any particular endorsement or with the intent to infringe the copyright of the respective owners.

Copyright
Copyright © 2016. Eindhoven University of Technology. All rights reserved. No part of the material protected by this copyright notice may be reproduced, modified, or redistributed in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the Eindhoven University of Technology and ASML.

Preferred reference
Foreword

At ASML, great effort is put in ensuring the quality of new products. With the ever increasing complexity and cost of the machines we ship to our customers it becomes less and less affordable to test the software controlling the machine on the machine itself.

For reasons of affordability and availability a distributed simulation approach was taken in the past in which software is tested in isolation as much as possible by simulating the required interfaces and corresponding machine hardware as well as the relevant environment. This distributed approach has led to a large variety of simulation approaches, concepts and architectures.

Unifying on architecture, sharing knowledge on the creation of simulators and enabling the use of different simulators in a larger context for integration testing, are the main short term goals that where set when the Simulation Competence was created last year.

One of the identified needs from the ASML software community was efficiency in creating simulators. For this purpose, and also to enforce a unified architecture, we placed an item on the competence roadmap to create a simulator development environment and executed the roadmap item by creating an OOTI assignment for it, which was selected by ASML management for the OOTI program.

Although she missed some relevant experience, Irina has been given this assignment because she was very motivated in her interview and gave the impression she was capable of gaining knowledge quickly, which was a pre-requisite for this assignment that is executed in a complex environment.

We jump started her assignment by flooding her with ASML specific knowledge, and domain knowledge relevant for the project. We were pleasantly surprised by the ease in which she absorbed this knowledge, she did so in a matter of weeks while it typically takes a new ASML software employee months to learn, and even longer to apply, this amount of knowledge.

I must state that the core team members she interacted with still needed to align on some of the topics relevant for her assignment when she already started and I admire her ability to cope with some of the confusion this sometimes caused and with the sometimes changing requirements which is actually very typical for a dynamic ASML project.

Irina has shown independence in project execution and made great progress by asking for help when needed not only to the core team, but also to the people in her network she created within ASML.

On our request she explored the solution space beyond the alternatives we gave at the start of the assignment and by doing so she came up with elegant solutions that are also in line with other tool developments currently being executed at ASML.

Results shown so far look very promising and I’m confident that before the end of the project we have a prototype of a simulator development environment which will enable us to create basic simulators in a fraction of the time it currently takes us.

Pieter Koper

August 24th, 2016
Preface

The current report documents the final project of Irina Nikeshkina and it is required for completing the Software Technology (ST) program (also known as Ontwerpersopleiding Technische Informatica (OOTI), Dutch) and issuing the Professional Doctorate in Engineering diploma (PDEng). The two-year post-Master technological PDEng program is provided by the Eindhoven University of Technology under the banner of the 4TU.School for Technological Design, Stan Ackermans Institute – a joint initiative of the four universities of technology in the Netherlands.

The final project was executed within nine month at ASML Netherlands B.V. (referred to as ASML). It was devoted to designing a development environment that solves ASML simulator creation problems described in the report. Therefore, the report covers two groups of interests. First, it describes the ASML simulation aspects: the problems, domain, simulator architecture, and problem solutions. These aspects represent the key words for the first interest group. Second, the report demonstrates how a development framework can be built and what technologies may be used. For the second interest group the following key words were detected: Model-Driven Design (MDD), Domain-Specific Language (DSL), Domain-Specific Modeling (DSM), Meta Object Facility (MOF), and Code generation.

This report is a public version. A separate, private version is available at ASML.

August 2016
Acknowledgements

The success and outcome of the project required a lot of guidance and assistance from many people and I was extremely fortunate to get this along the completion of the assignment.

I wish to thank my supervisors for their consistent support, constructive criticism, and direction during my project. I express deepest thanks to my ASML supervisor Pieter Koper. He is a great teacher who helped me to get into the ASML domain very fast. Pieter invested his full effort in guiding me throughout the project and helping to coordinate different project parties. I would also thank my TU/e supervisor Rudolf H. Mak who helped me to coordinate my project especially in writing this report. Rudolf’s questions and suggestions brought new valuable ideas in the simulation domain and stimulated me to improve both the quality of the project and my writing.

A special gratitude I give to members of the Simulation Competence who played the crucial role in my project. Their constructive criticism and commitment to the project improved the system design and my designer skills. I wish to thank Ernest Mithun Xavier Lobo for his contribution to the project. From the most beginning, he encouraged me by demonstrating his interest in the project and showing where the designed system will be applied. Ernest’s experience in supervising OOTI projects and involvement in the current one made him an important part of the project steering group. I would also thank Ben Vandevalde for imparting his knowledge and experience in this project. On the one hand, Ben was one of the main project stakeholders and he clearly expressed his needs in the simulation frame. On the other hand, Ben was my technical aid and he always was ready to help me.

Furthermore, I would also like to acknowledge with much appreciation the members of the Projection Optic Box (POB) group who helped me in a simulator designing. I wish to thank Tim Scheffers and Patrick Sprenger for sharing their experience and giving feedback necessary for a simulator design.

I would like to express my warmest gratitude to all who helped me in the completion of this project. I wish to thank Theo Baan who gave his expertise necessary for the system design. Thanks to Marco Alonso whose suggestion of the system design approaches and technical help were invaluable to me. Thanks to Berrie van den Eijnden for spending his time and sharing knowledge that allowed me to analyze simulation domain from another side – the system point of view. Thanks to Wilbert Alberts whose opinion and suggestions helped me to take right decisions on time.

Last but not least, I would like to thank Ad Aerts, the director of the Software Technology (ST) program, and Desiree van Oorschot, management assistant, for their support and care throughout the program. Additionally, I would like to thank the coaches and trainers of the ST program who helped to improve my soft and technical skills.

Irina Nikeshkina

August 2016
Executive Summary

ASML is the world leading provider of photolithography machines for the semiconductor industry. Lithography is a complex process that requires a reliable control system. The biggest challenge of designing such a system is the realization of an effective and efficient software and hardware testing process. Currently, testing is done on various software levels and on different testing platforms.

When the testing involves electronic or mechanical equipment, the process becomes more expensive and time consuming due to low availability of this hardware compared to testing on pure software platforms. However, integration and system testing can hardly verify software quality without this hardware involved. Therefore, hardware simulation approaches are widely used in ASML for testing purposes.

A number of simulators have already been designed for different subsystems of the machine, in different ASML departments, by different developers, implemented using different programming languages, and as a consequence with different architectures. Such simulator variability exists due to a lack of clear guidelines that enforce a unified architecture and guard the process of a simulator design. Moreover, the knowledge of the simulated hardware is generally outside of the software domain. At the same time, each simulator design involves repetitive work implemented manually. Together, these make the modelling, design and integration of simulators a complex process. This problem brings the following high-level goal of the current project:

Design a framework for creating simulators that enforces unified simulator architecture, reduces the engineering effort by code generation, and can be used by both software and functional engineers for simulator design.

During the project design phase, a simulation component was divided into several parts: interceptor generation, binding generation, simulator modeling that includes equations modelled in Matlab, and code generation.

Various in-house and external development tools are available and used to model and automate a part of the software development process. Such tools are reused in the designed simulator development environment:

- The external Acceleo tool is used for the interceptor generation.
- A code generation part of the in-house DriFE tool is used for binding generation.
- The XText technology is used for creating DSL for the simulator configuration.
- The external EMF, Sirius, Acceleo tools are used for modeling simulators and code generation.
- The in-house MATES tool is used for generating code out of Matlab models deployed in the simulator model.

As the result of the project, the Asml Simulator development Environment (ASTER) is designed to minimize the engineering effort in order to create a simulator by providing modeling and code generation facilities. In addition, the code generated with ASTER can be packed and easily deployed in the ASML source code (software archive).
Table of Contents

Foreword ................................................................. i
Preface ........................................................................ iii
Acknowledgements ......................................................... v
Executive Summary ........................................................ vii
Table of Contents ............................................................ ix
List of Figures ........................................................................ xi
List of Tables ........................................................................... xiii
1. Introduction ....................................................................... 1
   1.1 Context ......................................................................... 1
       1.1.1 Lithography .......................................................... 1
       1.1.2 Testing Challenges .................................................. 2
       1.1.3 Simulation Challenges ............................................. 4
   1.2 Outline .......................................................................... 5
2. Stakeholder Analysis ........................................................... 7
   2.1 Stakeholders ............................................................... 7
   2.2 Stakeholder Risk Analysis ............................................. 10
3. Problem Analysis ............................................................... 11
   3.1 Context ......................................................................... 11
   3.2 Project Goals ............................................................. 11
   3.3 Design Criteria .......................................................... 12
   3.4 Scope ........................................................................... 13
4. Domain Analysis ............................................................... 15
   4.1 Introduction .................................................................... 15
   4.2 Software Architecture .................................................. 15
   4.3 Inter-Process Communication Mechanism ....................... 16
       4.3.1 Interface and Binding Concept ................................. 18
   4.4 Simulator Classification ............................................... 18
       4.4.1 Interface Selection Switch ....................................... 20
       4.4.2 Server Rebinding .................................................. 20
       4.4.3 Client-side Interceptor ........................................... 21
5. System Requirements ......................................................... 25
   5.1 ASTER Use Cases ....................................................... 25
   5.2 ASTER Requirements .................................................. 26
6. System Architecture ........................................................ 29
   6.1 Introduction .................................................................... 29
   6.2 ASTER Architecture .................................................... 29
7. Design and Implementation .................................................. 31
   7.1 Common Module ........................................................ 31
   7.2 Interceptor Generation Module .................................... 32
   7.3 Binding Generation Module .......................................... 32
   7.4 Equations Module ........................................................ 33
   7.5 Simulator Modeling Module .......................................... 34
8. Conclusions .................................................................... 37
   8.1 Results .......................................................................... 37
Glossary .............................................................................. 39
Bibliography .......................................................................... 41
   References ........................................................................ 41
   Additional Reading .......................................................... 41
About the Authors ............................................................... 43
A Requirements ................................................................. 45
   A.1 ASTER Requirements .................................................. 45
List of Figures

Figure 1.1 – Chip making process ................................................................. 1
Figure 1.2 – Testing platforms comparison ................................................. 4
Figure 1.3 – Testing and Early Integration using simulation ...................... 5
Figure 2.1 – ASML matrix structure .............................................................. 7
Figure 2.2 – Stakeholder prioritization map ................................................ 9
Figure 4.1 – Software layering architecture ................................................. 15
Figure 4.2 – Broker architecture pattern .................................................... 17
Figure 4.3 – SUT to production component communication ..................... 19
Figure 5.1 – ASTER use cases ..................................................................... 25
Figure 5.2 – The CAFCR framework ............................................................ 26
Figure 7.1 – Simulation project structure ..................................................... 32
Figure 7.2 – Simulator decomposition .......................................................... 35
List of Tables

Table 1.1 – Testing platforms ................................................................. 3
Table 2.1 – Stakeholder analysis ............................................................. 8
Table B.1 - ASTER Requirements ........................................................... 45
1. Introduction

Nowadays, it is difficult to imagine human life without electronic equipment such as smart thermostats, smartphones, and others. They all have the same core element – a computer chip also known as an integrated circuit (IC). Chip manufacturing consists of several steps and the most difficult step – exposure – is performed by ASML machines. With few exceptions, all chips in the world are made by companies that work with ASML machines. The biggest challenge in designing such machines is efficient quality and throughput control. Because of the high price and low availability of ASML machines for testing, hardware simulation is widely used.

1.1 Context

In order to understand the purpose and challenges of the current project, the company context should be analyzed. First, the chip manufacture and lithography processes are described. Then, existing software and hardware testing processes and their challenges are given. Finally, a description of simulators and their weaknesses concludes the discussion of the current project.

1.1.1. Lithography

A computer chip, also known as integrated circuit, is a piece of silicon into which a three-dimensional pattern is etched. The chip making process starts with a silicon crystal molded into a 30cm diameter cylinder. Then a chain of general actions, illustrated in Figure 1.1, is applied to the crystal. [1]

![Figure 1.1 – Chip making process](image)

1. **Slicing.** The silicon crystal is sliced into thin slices. These slices are called wafers and are the base of the chip making process.
2. **Polishing.** The wafer is polished to a mirror-like surface.
3. **Material deposition or modification.** Certain materials are transferred to the disc, modifying its properties.
4. **Photoresist coating.** The wafer is coated with a layer of photoresist. This coating protects the unexposed parts from being etched away in later steps.

5. **Exposure.** A strong light source projects the image from a reticle, which contains a blueprint for the layer, onto the wafer. Light causes the photoresist to harden or soften.

6. **Developing and baking.** The wafer is developed and baked; soft resist is removed. The developing step stops all chemical processes.

7. **Etching and ion implantation.** The revealed material is etched away with a chemical bath. The remaining photoresist protects the material that should not be etched.

8. **Removing the photoresist.** The photoresist is removed and a pattern, etched into the wafer, remains. After this step, it is possible to continue with step 3, in order to form a three-dimensional structure. This is typically done 30 to 40 times.

9. **Completed wafer.** The result of this process is a disc with a wafer-like pattern. This is why these discs are called wafers. Measurements are done to ensure the quality of the wafer.

10. **Separation.** The different blocks – integrated circuits – are separated from each other.

11. **Packaging.** Each integrated circuit is packaged and serves as the core of a chip, for example a processor. [2]

In the chip making process, wafer exposure (Step 5) is the most critical step and it is performed by photolithography systems. ASML is the leader in producing lithography machines (called TWINSCAN) – accounting for 85% of the whole market. Three key drivers allow ASML to lead:

1. **Productivity** or throughput (wafers per hour) makes the cost of chip production cheaper.
2. **Imaging** (nm accuracy) makes the size of chips smaller, increases computational power on the same area, and reduces the power consumption per logic circuit of the chip.
3. **Overlay** (high yield) controls a wafer location in order to place two layers on a wafer that are connected with no more than a 1-2 nanometer fault. This process is called wafer positioning and it ensures that the finished IC performs according to specification. [3]

To achieve the listed key drivers, complex machinery and control systems are required.

### 1.1.2. Testing Challenges

Lithography is a complex process that requires a reliable control system such as TWINSCAN. TWINSCAN is a machine controlled by distributed software executed on a network of computing nodes – hosts. The biggest challenge of designing such the machine in ASML is the realization of an effective and efficient software and hardware testing process. Currently, testing is done on various software levels (for example, software components, functional modules, and system) and on different testing platforms distinguished by functional completeness. Four testing platforms exist in ASML, whose features are described in Table 1.1. [4]


Table 1.1 – Testing platforms

<table>
<thead>
<tr>
<th>Testing Platform</th>
<th>Description</th>
<th>Simulation Approach</th>
</tr>
</thead>
</table>
| Dev Bench        | • Pure software shared among all developers by means of virtualization  
                  • No electronics  
                  • No mechanical equipment  
                  • The entire hardware platform is simulated  
                  • Only the TWINSCAN control host is present (no other computing nodes). | Software-in-the-Loop (SIL) |
| Test Bench       | • The real hardware and software infrastructure are present for the system under test  
                  • Software deployed on computing platforms depending on test requirements  
                  • Electronics, such as amplifiers and sensor boards, may be present  
                  • Mechanical equipment is replaced by simulation.  
                  • Deployed on real computing nodes. | Hardware-in-the-Loop (HIL) |
| Test Rig         | • Software deployed on the same electronics as present in a real machine  
                  • Electronics can be incomplete  
                  • Only limited specific hardware available (approximately, one or two subsystems).  
                  • Deployed on real computing nodes. | Hardware-in-the-Loop (HIL) |
| Proto            | • Full machine configuration  
                  • Part of the equipment may be preproduction prototypes.  
                  • Deployed on real computing nodes. | No Simulation |

The usage of testing platforms depends on several important aspects (see Figure 1.2):

- **Costs.** Because of the high price, there are only a few Proto machines in ASML available for testing. For each test software should be configured for specific features of a machine where it is deployed; this is also very costly. Additionally, software testing always brings a risk to damage the machines. To sum up, usage of Proto machines is one hundred times more expensive than, for example, usage of Test Bench. Obviously, the price grows with the amount of electronics and mechanical equipment involved in a platform. Thus, the cheapest testing platform is Dev Bench.

- **Functional Completeness.** To optimize costs, incomplete configurations are used during software integration. Incomplete configurations impose restrictions on test coverage. For example, testing on Dev Bench is limited, because specific properties of the hardware platform, such as time behavior, cannot be simulated accurately. Only on Proto machines can fully complete functionality be tested. This shows the dependency of test coverage and costs – the more complete the test setup, the more expensive it is.
— **Availability.** By means of virtualization, Dev Bench is constantly available to multiple developers at the same time. Developers can even access the platform from their home. In contrast, the number of Test Bench or Test Rig environments and Proto machines is very limited. Many developers need to test their software on the same machine, but only limited test time is available. This leads to postponing tests and missed opportunities for detecting errors that could be fixed in an early stage.

During software development, all testing platforms described above are used to verify system quality. The initial testing starts with Dev Bench when different subsystems are tested without any hardware. Then then the TWINSCAN software is tested on Test Bench and Test Rig. The final integration tests are executed on Proto with all the hardware, electronics, and optics involved in place.

Testing in Dev Bench can give the most benefit during development. Because of its high availability, it allows software to be developed in short iterations and tested more often than when using other platforms. In short, the more software that is covered by tests in Dev Bench, the fewer tests that fail on Proto, and the cheaper the testing process is. Therefore, the main goal is to test as much software as possible in the Dev Bench platform.

### 1.1.3. Simulation Challenges

Integration and system testing can hardly verify software quality without hardware involved. For example, interaction of software and hardware can be excluded by adding a condition or simplifying a software function behavior. However, if the code of some software modules is changed only for testing purposes, no guarantee exists that these modules behaves the same way in an assembled machine.

On the other hand, a number of errors could still be found on Dev Bench, which minimizes expenses when testing on the next platform. In order to test software on Dev Bench, simulation of software and hardware is used. The integrated software should be tested against a simulator instead of actual hardware as shown in Figure 1.3. Such simulation is called Software-in-the-Loop or SIL approach.

A simulator represents a model of physical elements, such as actuators or sensors, their physical and logical interaction (behavior), and the implementation of the software interface towards these elements. A simulator should communicate in a seamless manner acting as the real hardware elements:
- Get the function calls from the control software.
- Simulate control commands to actuators by a faithful update of the system state.
– Reply to the control software with the hardware sensor values and positions based on the system state.

**Figure 1.3 – Testing and Early Integration using simulation**

Such the SIL approach has the following benefits:

- The simulation is transparent to the production software by implementing the same interfaces as the substituted software component forming the abstraction of the Hardware. Thus, from the control software perspective, there is no difference with communication to the simulated or real components.

- The independent simulator does not pollute the real production code.

- 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 conditions, specific parts of the software that are hard to test can be evaluated.

- Due to inaccuracies and noise in the hardware sensors, a single test running several times with the same input data may produce different output results. Tracing of such results makes it difficult to detect real problems. Using a simulator, it is possible to produce the same behavior within two identical test runs.

Because Dev Bench is easily available for developers, by using simulators, software can be tested more often than on other testing platforms. This means that the testing process is cheaper, because software errors are found on Dev Bench, while Test Bench and Proto are only needed for functional qualification. Therefore, the simulation approach is widely used during TWINSCAN software development.

### 1.2 Outline

This document describes the process of designing ASTER. The process was started from collecting knowledge necessary for ASTER design. First, the stakeholder analysis is described in Chapter 2 where important stakes of the project and their owners are described. Next, a problem and solution opportunities of the project are described in Chapter 3. They are further detailed in Chapter 4 where the simulation domain and related concepts are analyzed.

The ASTER design starts with requirement analysis described in Chapter 5. Then, the system architecture and design with implementation are described in Chapter 6 and Chapter 7, respectively. The project conclusion is described in Chapter 8.
2. Stakeholder Analysis

The success of any project depends on the people involved. On the one hand, stakeholder analysis is very important at the beginning of the project: identifying stakeholder groups helps to understand the project context and problem domain. On the other hand, stakeholder analysis provides information on possible project impacts. For example, a good analysis can even predict whether the project will have a future. The purpose of this chapter is to provide such a stakeholder analysis.

2.1 Stakeholders

Stakeholders are individuals affected by activities, decisions, or outcomes of the project. In the project content, four main stakeholder groups can be identified:

1. Simulation Competence group. In order to understand what are competences, ASML organizational structure should be analyzed. ASML has a matrix organizational structure: it has different departments on one axis and projects the other axis (see Figure 2.1). The special groups called competences are organized across the departments and projects. The purpose of such competences is to define experts in a single domain from different teams within ASML. These experts organize a group where they can improve working processes and share knowledge. For example, the Simulation Competence involve experts of simulations from almost all existing departments of ASML. Not all members of the Simulation Competence have stakes in the project, but only experts who design simulators below the driver layer for TWINSCAN software (more about layers in Section 4.2).

![Figure 2.1 – ASML matrix structure](image)

2. Projection Optic Box (POB) group. The interest of this department is mainly in the outcome of the project. While the Simulation Competence group is interested in ASTER for designing simulators, POB is interested in a simulator for the Mirror Heating driver that was designed using ASTER during the project. This simulator must prove the advantages of the ASTER concepts; in case of success, it will be used by the POB group.

3. TU/e. As part of their educational programs TU/e offers a PDEng program in ST targeting to deliver top-level software designers to industry. Part of the educational trajectory offered in the PDEng program ST involves an extensive internship in the form of a design project (the current project, particularly) in a company. The stakeholders of the University are mainly interested in the process of the project and design quality of both software and report in order to establish whether the criteria imposed by their program are met by the trainee.
4. Trainee. The main interest of the trainee is getting knowledge and experience as a technological designer within a professional context. As the result of the project, the trainee expects successful graduation that depends on satisfaction of all stakeholder groups described above.

Table 2.1 describes the members of each stakeholder group in detail.

<table>
<thead>
<tr>
<th>Table 2.1 – Stakeholder analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>Simulation Competence</td>
</tr>
</tbody>
</table>
| Pieter Koper | ASML project supervisor, Leader of the Simulation Competence group. | - Satisfaction of simulation experts needs.  
- A unified approach of designing simulators with enforced architecture  
- The high quality of ASTER design with well-defined documentation |
| Ernest Mit | TWINSCAN simulation expert. Recently moved to EUV Source simulation | - Building a well-defined process of a simulator design.  
- Minimization of engineering effort when designing simulators  
- Designing simulators by both software and functional people |
| Xavier Lobo | TWINSCAN simulation expert | - Approaches for modeling simulators and their behavior  
- A well-defined process of simulator design. The process can include usage of different in-house or external tools |
| Ben Vandeveldt | System simulation expert. Has expert role. | - Integration different simulators into one system. |
| Berrie van den Eijnden | Metrology software architect. Simulation expert. | - Implements simulation techniques in the metrology department that can be reused in ASTER. |
| POB | | |
| Patrick Sprenger | Architect NXE Projection Software. | - A simulator for the POB group |
| Tim Scheffers | Team Lead NXE Projection Software. | - A simulator for the POB group |
| TU/e | | |
| Ad Aerts | 3TU. Stan Ackermans, Software Technology Program Director | - Quality of project and process  
- Prestige of a ST PDEng program  
- Relationship with industrial companies |
| Rudolf Mak | TU/e project supervisor | - Scientific aspect of report  
- Quality of report  
- Respect of the project time line |
| Trainee | | |
| Irina Nikeshkina | PDEng trainee | - Gaining knowledge and experience in project, stakeholders management  
- Getting experience as a technological designer  
- Getting knowledge in the simulation domain  
- Successful graduation. |

Apart from the main four stakeholder groups described above, other stakeholders exist:
– Simulator designers. All users of ASTER can be divided into the Simulation Competence and simulator designer groups. The responsibility of the Simulation Competence is to bring ASTER to other designers. Thus, it is important to take into account the simulator designer group for the project, but it is not the main stakeholder group.

– Tools owners. In ASML a set of tools that can be fully or partially reused in ASTER has been already designed. The owners of these tools play the role of consultants, but they have a little interest in the project.

– Managers. The interest of this group is in business value of the project. This group assigned a budget for the project and can decide whether the project will be continued. This group has delegated the activities related to their interest to the Competence Leader; therefore it is not in the main stakeholder group.

Figure 2.2 illustrates the stakeholder prioritization map where all stakeholders are divided into four priority groups depending on their power and interest in the project.

The prioritization map demonstrates how much a stakeholder wants to be aware of the project status and progress. The difference in priorities of groups also affects the style of communication with stakeholders. Different levels of communication bring different levels of details. For example, the Simulation Competence group expects a personal approach and technical details, while the management group is satisfied with a brief explanation.

For each prioritization group the communication scenarios were chosen: high interest/high power – face-to-face meetings, regular update on project status by means of e-mail or report, intermediate result demonstrations; high interest/low power – consultations by face-to-face meetings, regular update by e-mail; high power/low interest – occasional update on project progress or issues via individual meetings; low power/low interest – face-to-face consultations, occasional update on project progress via e-mail.
2.2 Stakeholder Risk Analysis

Clashing interests of different stakeholder groups are the biggest risk for the project. All stakeholders groups have to be satisfied, but sometimes this cannot be reached due to conflicting stakeholder needs. For the current project, three possible clashes exist:

- **Simulation competence versus POB.** The competences group is interested in designing ASTER for creating simulators, while POB is interested in creating the simulator. Both sides have requirements for the current project. However, if the requirements of both parties do not fit time line of the project, the interests of one side should be limited. In case of this situation, the action should be made according to the prioritization map described in the previous section: the Simulation Competence group has more prioritization than POB; therefore, the requirements of this group should be performed first.

- **Clashes within a group.** The simulation Competence group includes experts in simulation from different departments. Obviously, these stakeholders have different experiences and choose different approaches when designing simulators. Therefore, they have different interests and priorities for ASTER. This can cause problems in defining requirements that can lead to unreachable goals. Alignment of all stakeholder interests is the responsibility of the trainee. When conflicting requirements cannot be solved during negotiations with the stakeholders, the trainee should reach an agreement with the company supervisor first. At the same time, the trainee should manage all stakeholder expectations adjusted to the agreements with the supervisor. This allows keeping the average level of satisfaction.

- **TU/e versus Simulation Competence.** While the company is mostly interested in a working prototype and treats the final report as supportive documentation, the University pays the most attention to the report: how well the design process is described. Both these groups have the same priority in the stakeholders map. In order to eliminate any interest clashes of these groups, the trainee should plan the project process well and make it transparent to both parties. In case the conflicts still appear, the timing aspect of the project should be taken into account: at the beginning of the project the company needs are the biggest priority; but the closer the project is to the end, the more priority should be given to the University, because the report is the indispensable part for the project evaluation and graduation.
3. Problem Analysis

The analysis of the company context and stakeholder interests allows formulating the problem statement, which the current project is intended to solve. First, this chapter describes the problem context; next, design opportunities on how the problem can be solved, the important design criteria, and the project scope are described.

3.1 Context

A number of simulators have already been designed for different subsystems of the TWINSCAN machine. Some of the simulators were analyzed during the project. All these simulators were developed within different departments, by different developers, and implemented using different programming languages and coding styles.

Such variability of the simulators exists due to a lack of clear guidelines that enforce a unified architecture and guard the process of a simulator design. Some differences in architectures are caused by very specific domain requirements of each simulator, but in many cases they are caused by their developers’ own style. The simulator architecture variety causes problems in standardization of simulation design that leads to maintenance difficulties.

Additionally, simulators have common parts, for example a binding code that allows components to communicate with each other. Such common parts could be generated automatically, but currently they are written manually for every simulator. Since these parts require only configurational effort without engineering tasks involved, generating them automatically also would avoid a lot of errors that could appear while manually coding the common parts.

Another challenge in designing a simulator is the knowledge of simulator behavior that needs to be imported from outside the simulation domain. For example, a relationship between physical components modelled in a simulator can be known by a functional specialist of one department, whereas the simulator is designed by a software developer in another department.

3.2 Project Goals

Together, the challenges described above make the modelling, design, and integration of simulators a very complex process. A number of simulators were designed in the past, more simulators are under development now, and more simulators will be designed in the future. Therefore, the simulation design process must be standardized and automated. This problem brings the following high-level goal of the current project:

*Design a framework for creating simulators that enforces a unified simulator architecture and reduces the engineering effort by code generation.*

Guidelines and rules can be collected from the Simulation Competence and their experience of designing simulators mentioned above. For proving the benefits of the ASTER framework, a simulator for Projection Optic Box group should be designed using ASTER.

Various off-the-shelf and in-house development tools are already available and used to model and automate a part of the software development process. For instance, DriFE is an in-house model driven engineering tool for designing drivers that gener-
ate code deployable at TWINSCAN. Another example of an in-house tool is ALIAS, used for modeling interface protocols and generating code out of models.

Such tools can be efficiently used for the simulator design. However, they are stand-alone tools, deployed in different production contexts, and therefore are not available in a standard ASML toolset to create simulators. [5] Thus, another design opportunity states:

*Design a framework that includes facilities of existing in-house tools suitable for modeling simulators and generating code.*

The intention of some in-house tools is to facilitate the separation of functional and software work. For example, MATES is a Matlab tool that takes Matlab files containing difficult algorithms modeled by functional developers and creates a package that can be deployed in the TWINSCAN software archive by software developers. Such an example demonstrates how two professional fields can interact with each other. This approach can also be used for simulation design: a functional engineer can model physical elements and their interaction, while a software developer can model binding necessary for software communication (see Section 4.3.1.). However, the simulation elements also can be designed by the software developer alone. This brings the last design opportunity:

*Design a framework that can be used for creating simulators by both software and functional engineers.*

### 3.3 Design Criteria

The ASTER framework is a new approach in simulation. At the same time, similar artefacts exist in ASML supporting other system part design. For example, DriFE is an in-house tool made specifically for creating drivers. Such artefacts help to understand the criteria important for the design:

- **Genericity.** As the high-level goal of the project is to unify simulator architecture, the genericity of this architecture is the most important criteria. That means that the simulator architecture should fit all simulators below driver level (see architecture levels in Section 4.2.), and therefore, these simulators can be designed with ASTER.

- **Inventiveness.** As described above, the framework, common for creating any driver simulator, does not exist in the Simulation Competence, so it has to be invented. However, inventiveness does not mean the deployment of a novel technology, but it means an innovative combination of existing in-house and external technologies.

- **Functionality.** The stakeholders are clear on the minimal functionality they expect from ASTER: based on the interfaces of the substituted component, ASTER shall generate binding code (see Section 4.3.1.); ASTER should provide a possibility to model a simulator behavior and generate code out of the models; ASTER should support equations written in Matlab and generate code deployable on TWINSCAN.

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

- **Elegance.** Though the ASTER design should be well structured and easily extendible, it reuses other tools that can affect the overall design elegance. As stakeholders are more interested in ASTER as proof of concept for designing simulators, the elegance influenced by usage of other tools is not of high importance.
- **Complexity.** The simulation context is limited by the project scope (see Section 3.4). The generated code complexity should fit the complexity of the simulator for POB group with a simple simulation model.

**3.4 Scope**

The scope of the project is limited by the TWINSCAN software on the Dev Bench. The goal of the project is designing the ASTER framework for creating simulators that substitute for hardware, and therefore can be deployed on Dev Bench. The scope of simulators is limited by the level below driver subsystems.
4. Domain Analysis

The project context and scope described in Chapter 3 clearly define the borders of the current project domain. This chapter concentrates on the technical details of the domain necessary for the ASTER design.

4.1 Introduction

This chapter describes the important concepts of ASML software necessary for understanding the simulation design reasons, requirements, and approaches described in this report.

4.2 Software Architecture

The domain analysis of the ASTER framework starts with an investigation of the software architecture of the TWINSCAN system. The knowledge of overall architecture of this system is very important, because different simulators address different subsystems depending on testing purposes.

The overall functionality of the TWINSCAN software is decomposed into various system functions. Each system function contains one or more functional clusters – logical functions of the system, or facilities for supporting other functions. Functional clusters are structured according to a layered architecture illustrated in Figure 4.1.

![Figure 4.1 – Software layering architecture](image)

**Facilities** (blue layers) are functionalities that can be reused in multiple functional clusters:
• **OS & I/O Abstraction** relates to the generic computing hardware that all software runs on.
  • **Generic Facilities** can be re-used at all ASML machines.
  • **Domain Facilities** can only be re-used within the lithography domain.
  • **CPD Facilities** can only be re-used within the calibrate-performance-and-diagnostics applications within the lithography domain.

**Production functionalities** (gray layers) are software responsible for the lot\(^1\) production functionality:
  • The Subsystems layer is responsible for executing hardware actions autonomously. A subsystem typically consists of a subsystem application part, a subsystem peripheral part, and a hardware interfacing/control part.
  • Subsystem Peripherals hide implementation details of the electronics hardware. Apart from CPD applications and facilities, all higher layers are unaware of the chosen electronics technology and configuration.
  • Subsystem Applications execute queued physical actions, for example, moving hardware or performing measurement.
  • The synchronization layer schedules physical actions and negotiates for optimal scan timing with the involved functional clusters. This layer contributes only to the timing aspect and hides it from other layers.
  • Logical Actions translate logical actions to physical actions and encapsulate physical actions and their ordering.
  • Metrology compensates for system effects by determining the set points for all relevant subsystems during exposure. Metrology also verifies and diagnoses some aspects of system behavior such as overlay and imaging. It requires knowledge of multiple functional clusters, for example, translating wafer coordinates into wafer stage coordinates, or routing the scans over the wafer.
  • Controllers perform sequencing, synchronization, and exception handling of machine activities such as measuring, exposing, and calibrating.

**Applications and system interfacing** (white layers) include applications and facilities that are used for the setup and machine diagnostics. These layers are placed high in the architecture because some of them control the entire machine and use high level production sequences.
  • The CPD applications layer involves application for calibration, performance, and diagnostics of the system.

Two layers above the CPD applications allow customers and external applications to access the machine:
  • System Interfacing allows external tools and automation systems to access and even control the machine.
  • Applications allow human interaction with the machine.

**External Applications** are other systems that interact with the TWINSCAN machine. [1]

Understanding the software architecture helps to identify the level of simulation in the current project – below Subsystems layer also known as the Driver layer.

### 4.3 Inter-Process Communication Mechanism

---

\(^1\) In semiconductor manufacturing terminology, a **lot** is a set of wafers that is processed in a fab (semiconductor fabrication plant) from start to finish; all wafers in the lot receive the same treatment.
The communication mechanism between ASML software components is a key aspect effecting simulation approaches. Section 4.4 describes what simulator types exist depending on communication between the system under test (SUT) and a simulator that fully or partially simulates environment of a substituted production component. Therefore, the communication mechanism is analyzed in this section which gives insight on the challenges described further.

The TWINSCAN application consists of multiple processes that can run on different nodes. When processes on different nodes need to communicate with each other, they act as a single application using remote service invocation. The TWINSCAN software is based on the broker architecture illustrated in Figure 4.2. Such a communication pattern is called inter-process communication in ASML. [6]

![Broker architecture pattern](image)

In principle, the remote service invocation approach does not require the existence of a broker: the caller (client) may be aware of the location and identity of the callee (server). However, this is not possible when processes of the TWINSCAN application run on different hosts. In the Broker pattern, the role of the broker is to obtain necessary referential decoupling: the client does not aware of the server location. Client calls are directed towards the broker forwarding them to a suitable server determined by the binding process.

The advantage of the Broker pattern is in hiding details of component implementation that performs remote service invocation by encapsulating them in a layer other than the component itself. [7] This layer provides an interface to the client that calls services just as it would invoke a local interface.

In Figure 4.2, ServiceInterface is a necessary abstraction that makes distribution possible: it declares a contract about the service provided by the server without exposing the implementation details on the server side.
When implementing the distribution, the client and server proxies handle sending a method invocation and its parameters across the network to the server and then sending the response back to the client. The client simply invokes the performFuncA method on the client proxy as if it were a local call because the client proxy actually implements ServiceInterface. The code change to the client would be minimal and it does not require any knowledge about the distribution nature of the system.

If the client proxy communicates with the server proxy directly (local call), the client is aware of the server location. This means that the server cannot be changed or moved to a different location at run time in a transparent way to the client. At the same time, with the broker the client does not know the server location; this allows substituting the server with a simulator component without client code interference. In other words, the SUT is not aware whether it communicates with a real or simulated component. This brings a guarantee that the functionality of the SUT is not changed even if the server is replaced.

### 4.3.1. Interface and Binding Concept

Interfaces play an important role in the inter-process communication mechanism. From one side, in the broker architecture an interface gives the abstraction necessary for distribution. In this case, an interface is separated from the realization of a server that implements this interface.

From another side, the interface abstraction allows bridging different languages. For example, a component of the TWINSCAN application can be implemented in C, C++, Python, and Matlab.

Such interface abstraction is reached by introducing a generic Data Definition Format (DDF, extension .ddf). DDF provides a definition of data structures in a language-independent way. [8] In the ASML environment, such DDFs are also called interfaces, because DDFs are used in the inter-process communication: the data serialization in one language to data deserialization in another language is done through data structures, defined in DDF. For example, Figure 4.2 illustrates ServiceInterface, which in the TWINSCAN software would be presented as ServiceInterface.ddf (although the notation of such the interface does not correspond to the ASML standard).

At compile time a set of files are generated based on the provided DDFs. [9] These files are called binding necessary for communication between components. All binding files are packed into client and server libraries that are used by the client and server described in the broker architecture (see Figure 4.2). For the client-side library, only DDF is required as an input, while the server-side library in addition requires ASML class format (ACF). ACF is a file describing classes to generate in ASML internal format.

At run time the generated client and server libraries, are used in the inter-process communication mechanism: these libraries use the Broker (see Section 4.3 ) for sending and receiving messages over the TCP/IP network.

The binding concept allows creating a component in different languages and the simulation components are not an exception: all analyzed simulators are written in different languages that cause different architectural possibilities. Although the architectural variety of simulators is a disadvantage, in the future, developers still want to be able to design simulators in the desired language.

### 4.4 Simulator Classification
In this section, simulators are classified according to approaches used for communication between the SUT and the simulator. The important metrics that are used to qualify a simulator are:

- Transparent attachment to the SUT. It indicates how many adjustments in the production code of the SUT needs to be done in order to perform switch to a simulator. The best scenario is zero changes. In other words, for the SUT there should be no difference whether it communicates with production or simulation components.

- Production code impact. Apart from the code impact on the SUT, the changes can also appear in a simulated component or configuration files. In addition, if the simulator component becomes part of software delivery, this is also perceived as the production code impact.

- Compatibility of the simulator and system states. When a simulator is launched separately from the system start, it may be not consistent with the system. Assume that the system was started and all drivers were initialized. Then the simulator is launched, but initially it has the non-initialized state. If the SUT sends a request to the simulator, it can return NOK, which does not correspond to what the SUT would get in real life. If the simulator is launched together with the system start, it is consistent with the system. Based on the three described metrics, the advantages and disadvantages of approaches are defined.

In order to analyze simulator type, the communication between the SUT and a production component should be represented. The schematic communication, based on the broker architecture described in Section 4.3, is shown in Figure 4.3. The client component is the SUT, which communicates with the server through proxies as the Inter-Process Communication section describes. If the client and server run in different processes or hosts, the broker is invoked to assist the server allocation; otherwise, the client proxy calls the server proxy directly without any intermediary involved. Both proxies are generated at the server build time based on provided DDFs and ACF.

The section below illustrates simulator examples with respect to Figure 4.3.
4.4.1. Interface Selection Switch

Interface Selection Switch illustrated in Figure 4.4 introduces an approach in which the SUT decides when and whom to talk to. This approach is also known as hacking of the SUT, because the condition when to talk to the production component or simulator is done manually in the SUT code. This means, that the SUT is aware of the simulator existence. Moreover, this programmatic choice between the production component and simulator will be included into the delivering software.

With Interface Selection Switch, the simulator has its own declared simulators bound to server addresses different from the substituted component’s server address. The simulator interfaces can be copies of the production interfaces.

Advantage:
- The approach allows launching the simulator together with the system. In this case, if the simulator state is not changed by the tester externally, the simulator is consistent with the system state.

Disadvantages:
- Adding switch condition into the SUT leads to impact on the production code.
- Attaching the simulator to the SUT is not transparent.

4.4.2. Server Rebinding

With the server rebinding, the simulator implements interfaces provided by the production server. The simulator is started after the production server and is bound to the production server address (see Figure 4.5). When Broker receives a request from the SUT it searches a server by a server address passed with the request; the last found server gets the request. Therefore, if the simulator is bound to the server address after the production component, it receives all subsequent requests from the SUT. In this case, the switch to the simulator is done at the inter-process communication level.
Advantages:
- No SUT code interfering.
- Transparent attach to the SUT.
- No need to define a new server address for the simulator that leads to changes in the production code.

Disadvantages:
- Additional simulator startup is required after system start. This can cause incompatible states of the system and simulator.
- The simulator is forced to implement all the production server interfaces.
- In case of local communication of the SUT and production server, the remote simulator is bound to the server address, but the client calls the server implementation directly without the broker notification. Thus, the remote simulator cannot be called without production code impact.

### 4.4.3. Client-side Interceptor

This approach is based on the interceptor pattern. The interceptor allows changing the message flow of the software. An example is illustrated in Figure 4.6: the request goes from component A to B via the interface; interceptor C receives messages of A, processes them, and then may redirect the message to B. Component C acts like a spy changing the A to B messages in a transparent way for both A and B.

Both client-side and server-side interceptors exist, but for the current project only the client-side interceptor is of interest (see Figure 4.7). Compared to the other simula-
tion types, the biggest advantage of the interceptor is that it can implement part of the interfaces and even part of the functions provided by the production server. This can be done, because the interceptor pattern is implemented with a library preloading linked by a dynamic linker at run time. That means that the executable code contains undefined symbols resolving of which is postponed until a program is run. When the TWINSCAN software runs, the linker tries to resolve symbols by looking for the preload library first. Only then, it searches the regular library dependencies.

This is a very important criterion due to the following fact: interfaces can be defined in different components that might be released separately. For example, if the production component provides an interface that the simulator requires, but both are in different releases, the simulator is not allowed implementing the interface, because the latter can be changed in the production component without simulator update that leads to errors. The preloaded library is a partial client implementation which does not need to be symbol complete and therefore will not break.

The partial interface compliance has one more advantage. Usually, test suites for the SUT seldom require the complete functionality of the substituted component; in this case, part of the functionality is simulated and the other part is stubbed in a simulator. The stub means that functions have static behavior, for example, they always return OK. Stubs are commonly used in simulators. However, the interceptor allows avoiding the stubbed functionality, therefore, making the simulator code neater. At the same time, if the SUT sends a request to a function that is not implemented by the interceptor, the message ends up in the production server with real functionality.

Advantages:
1. Partial interface compliance.
2. Transparent attachment to the SUT and no production code interference.
3.Interceptor can be launched together with the system start.

Disadvantage:
1. As the interceptor should be designed per interface, it cannot fulfill simulation needs, because usually a simulator requires more than one interface. Thus, several interceptors should call the simulator component through new interfaces (a derived copy of the production interface).

![Figure 4.7 – Client-side interceptor](image)
The described simulation approaches are introduced in different simulators. For example, the Interface Selection Switch is used in the Wafer Handler simulator. This simulator introduces its own interfaces and depending on which platform the software is run, the SUT chooses where to send requests. The server rebinding approach is used in the Wafer Positioning simulator that implements the same interfaces as the production component. The interceptor approach was created recently; therefore, no existing simulators implement this approach. However, all modern simulators that are under design include the interceptor pattern.
5. System Requirements

After acquaintance with project stakeholders, the problem, and domain, the system design phase started. First two steps of this phase are the system use cases and the stakeholders’ requirements definition. This chapter describes the two steps for the ASTER design.

5.1 ASTER Use Cases

The processes of defining the system use cases and deriving requirements went concurrently. Specification of initial custom requirements allowed sketching use cases, which in their turn helped to derive new requirements and clarify the existing ones. This process took several iterations until the trainee and stakeholders understood the requirement meanings and their priorities.

ASTER use case model is illustrated in Figure 5.1. The system has one user category – a simulator developer. In most cases, simulators are created by a software developer, but it also can be designed by a functional developer.

Four main use cases exist:

1. Generate Interceptor described in Section 7.2. The scenario of this use case includes the following user-system interaction: the user provides interface specification file (.ddf) as input, from which ASTER derives all operations and suggests the user to choose those operations that should be generated in the interceptor (Section 4.4.3. describes that the interceptor can partially comply to the production component); after the user choose the operations, ASTER generates the interceptor.
2. Generate Binding described in Section 7.3 The scenario of this use case first requires the user to provide DDFs and binding configuration as input, then ASTER generates the binding code.

3. The Model Simulator use case describes how the user should model hardware in a simulator. This use case consists of several parts:
   - Define a simulator structure and behavior (see Section 7.5 )
   - Include Matlab models (see Section 7.4 )
   - Generate code.

4. Prepare the generated code for deployment in the TWINSCAN software archive. The simulator development is performed in the isolated environment in Eclipse IDE. Thus, the generated code by ASTER must be migrated (deployed) in the TWINSCAN software archive. ASTER provides facilities like organizing generated code and packaging it into an archive that simplify the deployment process for the user.

5.2   ASTER Requirements

The top-level architectural description of ASTER was made with the CAFCR framework [10] designed by Gerrit Muller. This framework uses five views helping to understand the inside of a system in relation to its context. These views are shown in Figure 5.2.

![Figure 5.2 – The CAFCR framework](image)

The customer objectives view and the application view provide why from the customer. The functional view describes what of the product, which includes (despite the name) both functional and non-functional requirements. The how of the product is described in the conceptual and realization views, where the conceptual view is changing usually less in time compared to the realization. [10]

Customers of ASTER are stakeholders from the Simulation Competence group (see Chapter 2). In order to understand what customers need, high-level goals of the project were analyzed. Four goals of designing ASTER are described below:

- Minimize engineering effort
- Optimize ASTER for maintenance and usability
- Minimize production SW risks
- Provide efficient simulator development process.

These goals help to derive customer what and understand why the customer has a specific objective. Appendix A.1 ASTER Requirements provides a table containing the high-level goals in the first column, the corresponding customer objectives (what) in the second column, and the customer application in the third column. Each application has an identity starting from “A” (AID), a priority using the MoSCoW framework (Must, Should, Could, Would), and a definition in the fourth, fifth, and sixth columns, respectively.
Iterative revision of the applications, their meaning, and priorities by stakeholders allows understanding whether the trainee builds the right system. The analysis of application satisfaction is the system validation process.

For every customer **how** of the application view corresponding product **what** or software requirements are defined in the seventh column. The requirements have identities starting from “R” (RID) and priority defined according to the MoSCoW framework. The satisfaction of the product requirements verifies that the system was built correctly.

One the one hand, the CAFCR framework helped the trainee to derive the customer and project requirements, find the common languages with the Simulation Competence stakeholder group, and understand what the system should be build. On the other hand, this framework and iterative revisions of the output (see Appendix A.1 ASTER Requirements) helped the Simulation Competence to combine the knowledge of simulation design, align their interests, and range their priorities of the different requirements.

■
After defining the system use cases and requirements, the system architecture was built. The ASTER architecture is represented in this Chapter, while the design and implementation of every system part are described in Chapter 7.

6.1 Introduction

In the current project, two views on the system architecture exist: a conceptual model of a simulator and architecture of ASTER. The simulation architecture was designed that represents decomposition of the simulation component, which is deployed in the TWINSCAN archive. The simulation conceptual module includes ASML proprietary information; therefore the simulation architecture is not demonstrated in this section. The simulation component was divided into five parts, every part is assigned with a color that allows visualizing related parts of the simulator and ASTER decompositions. The simulation decomposition are defined as follows: the interceptor related elements are marked yellow, the binding files are marked pink, elements that must be specified by the user as input are marked blue, environment simulation elements are marked green, and an equation component is marked grey.

For every part of the simulator decomposition a functional unit, called module, was designed in ASTER. Section 6.2 describes the ASTER architecture and demonstrates how the simulation decomposition is related to the ASTER modules.

6.2 ASTER Architecture

ASTER was designed as an extension of Eclipse IDE. An Eclipse application consists of several Eclipse components. A software component in Eclipse is called a plug-in. The Eclipse platform allows the developer to extend Eclipse applications with additional functionalities via plug-ins. So, ASTER is a set of Eclipse plug-in projects that together provide functionality of a simulation development environment.

Set of plug-ins responsible for a single ASTER module is combined into an Eclipse feature. An Eclipse feature is basically a list of plug-ins and other Eclipse features that is a downloadable and installable separate unit. Four Eclipse features were defined per module in addition to the common feature containing the simulation project creation plug-in. The ASTER features are illustrated in Figure 6.1 as packages that list the containing plug-in projects.
Colors of the features correspond to colors of the components in the simulation architecture with one exception: the simulation input (blue simulation components) are included into the Binding Generation module, because the input is configured for binding generation specifically.

The simulation and ASTER architecture represent the overview of the system and illustrates how different modules relate to each other. Next, the design and implementation of every ASTER module are described in Chapter 7.
7. Design and Implementation

ASTER consists of five modules, the design and implementation of which are represented in this chapter.

7.1 Common Module

The common module includes the simulation project creation. This section helps to understand how the project can be created, what structure it has, and what files ASTER generates by default.

ASTER is a set of Eclipse plug-in projects; in other words, ASTER configures Eclipse IDE to provide functionality for designing simulators. As software development in Eclipse is always performed within a project with an assigned nature (for example, Java or Modeling natures), the simulator designing is also performed within a project with a simulator nature. The project creation with the simulator nature is done by the plug-in project called asml.finalproject.simulationWizard. Let’s discuss the asml.finalproject.simulationwizard.wizards package containing three Java classes responsible for the simulation project creation:

- **SimulationProjectNewWizard.java** adds a Simulation Wizards section to the common Eclipse project creation wizards.
- **CustomProjectSupport.java** creates the simulation project as plug-in project with all dependences included.
- **Templates.java** generates the project structure and necessary files.

Once an ASTER feature is installed in the Eclipse IDE, the user is able to create a project with the simulator nature. Figure 7.1 – Simulation project structure illustrates the structure of the simulator project and the generated files by ASTER:

- **src** – a source folder containing a package named as the simulator project. The package includes Java files performing code generation.
- **bld** – a folder containing files generated by ASTER.
- **scripts** – a folder containing files launching code generation. All the files are created by ASTER and the user does not need to change them:
  - `workflow.properties` – a file describing project structure.
  - `generateTargetMM.launch` – a launcher generating targetMM model out of config.bind file (in input folder).
  - `Codegen.launch` – a launcher performing code generation by calling DriFE Codegen in the Targetmodel plug-in project.
  - `simMigration.launch` – a launcher preparing all generated files for packing into an archive before deploying in the TWINSCAN software archive.
  - `exportSources.xml` – a file packaging the generated code into an archive that can be deployed in the TWINSCAN software archive.
- **ddf** – a folder containing manually added DDF files that the simulator implements
- **input** – a folder containing binding configuration file. The empty `config.bind` file is created by ASTER.
- **model** – a folder containing the simulation model (`<project_name>.simmodel`) and graphical representation of the model (`<project_name>.aird`). Two empty files are generated by ASTER. The user should design his simulator structure and behavior using the representation file.
- **src-gen** – a folder keeping the generated code.
- **META-INF and libraries** – folders declaring all dependencies necessary for the simulation project. The User does not need to work with files in these folders.
After the simulation project is created, the user can model his simulator using modules described in Sections 7.2 – 7.5. By user request, every module generates code and keeps it in the src-gen folder. During design phase, the code generated by different modules is not integrated. Therefore, it should be structured before deploying the code in TWINSCAN. The simMigration.launch file performs the code structuring, and exportSources.xml creates the output of ASTER – archives that can be deployed in the TWINSCAN software archive.

7.2 Interceptor Generation Module

The client-side interceptor uses a library preloading approach that adds simulation code between the SUT and the production component at runtime without modifications of the production code.

The preload library implements the same interface as the production component. Preload library has a template implementing interception that includes a lot of repetitive code. Analysis of this template showed that it can be automatically filled based on the interface specification in DDF. Therefore, with DDF as the input, the library preloading implementation can be auto generated by ASTER.

In ASTER, the interceptor module is implemented as the Acceleo plug-in project asml.finalproject.intercept, which is packed into the feature interceptorModule. This plug-in has two model-to-text transformation templates: generateElement.mtl that creates the preload library implementation and generateMakefile.mlt that creates a makefile required for deployment in the TWINSCAN software archive. Both templates import the DDF metamodel.

Before generating the template, the interceptor module fetches all methods from the DDF interface and suggests the user to choose those methods that should be preloaded in the interceptor. The not chosen methods are deleted from the DDF model (not from textual DDF stored on a hard drive). Only after this, the interceptor implementation is generated.

7.3 Binding Generation Module

Analysis of the ASML software started from an exercise on manual binding implementation in C++. This exercise demonstrated that binding creation is a very time consuming process that requires a lot of error fixing and configuration. In addition, a
lot of repetitive code with minor changes is required. In most case, such code is created by copying and pasting an existing pieces of code; this leads to nonobvious errors that are difficult to find. Due to error and configurational issues two days in average are required before a component with the implemented binding can run on Dev Bench. These challenges prove the necessity of the binding creation automation.

The purpose of the exercise was to detect which parts of the binding code can be either generated automatically, implemented by a developer, or configured as an input. Thus, four major questions were stated:

1. **What to generate?** For answering this question, the minimal set of files required in the binding generation module was defined. Some of these files can be fulfilled completely, others – partially.

2. **What to configure?** The following data must be specified as input to the binding generation: Component name, Required DDF, Classes, Processes, Server addresses.

3. **How to generate?** ASML has an in-house tool called DriFE that already provides binding generation for drivers. DriFE stands for subsystem Driver Framework Environment. [11] The mission of DriFE is to increase the effectiveness, efficiency and time of software driver development. The interest of the current project was in the code generation part of DriFE. In order to understand whether the DriFE code generation can be reused in ASTER, three questions were analyzed:

   1. What is the minimal input for the DriFE code generation?
   2. Does this input correspond to the input of ASTER?
   3. Do the generated files cover the simulation needs?

   Answering these questions, the final conclusion was made: the DriFE code generation can be reused for the binding generation module in ASTER.

4. **How to configure?** ASTER should provide the user with a way to specify the simulator inputs that is understandable by DriFE code generation. The decision was to design a domain specific language, called Bind, with which the user can configure his simulator and then ASTER transforms it to a format appropriate for DriFE. This DSL includes concepts of *class, implemented interfaces, and instances of a class deployed in a process* and bound to a *server address*. After the user specifies these concepts in the `config.bind` file (see Figure 7.1), the DSL is transformed for the DriFE code generation. The generated binding code already can be deployed in the TWIN-SCAN software.

### 7.4 Equations Module

The simulator below the Driver layer models physical objects that exist in the real environment. The behavior of one object can depend on another object. Such dependence is called interaction and it can exist between two or more physical objects. Sometimes, interaction involves a difficult physical-data transformation that can be expressed as an equation.

Quite often, the knowledge of the interaction between physical objects exists outside of the software domain. In this case, functional developers have to be involved in the process of designing simulators. In most cases, the equations already exist as Matlab models. All equations are of different complexity. The simple equations can be transformed into programming languages as C++ or Python by the software developer.
without the function developer help. In this case the equation is implemented in a method of an interaction class.

However, if the equation is very complex, a software developer cannot cope without the functional developer help. Therefore, ASTER provides the possibility to deploy a Matlab equation directly in the simulation modeling environment. Section 7.5 demonstrates how to deploy a Matlab model in the simulator model, while this section describes what happens behind the scene.

Within ASML Matlab can be used as a programming language. To deploy a Matlab model in TWINSCAN the model must be integrated into a software component and it must be coupled with software languages. This process is automated by an in-house tool called MATES.

MATES states for MATlab Exporter to SDEV (software development) and it exists to

- Eliminate the manual coupling of Matlab models to software applications.
- Decrease additional effort from Matlab experts.
- Improve communication between software applications and Matlab models.

Integration of ASTER with MATES is done through Pymatbridge. Pymatbridge is a set of Python and Matlab functions that allows these two systems to talk to each other. [12] When a simulator developer asks ASTER to generate code, the Equation module extracts the necessary input and invokes a python script that instantiates a Matlab environment; therefore, the Matlab tool must be installed in the system. After the instance is created, the Pymatbridge library uses MATES to generate code that can be deployed in TWINSCAN. This can be done because Pymatbridge allows directly invoking Matlab functions, passed as a parameter, from the Python execution. For example, the following code snippet illustrates how to instantiate Matlab and ask it to assign variable a to 1.

```python
mlab = Matlab()
mlab.start()
result = mlab.run_func('a=1;')
```

The significant feature of Pymatbridge is that the variable a, defined in the Matlab instance, can be accessed from the Python script by calling `mlab.get_variable('a')` function. [12]

When a Matlab equation is added to the simulator model, during code generation (C++) ASTER calls MATES though Pymatbridge. MATES produces a code archive that contains Matlab models and generated interfaces necessary for deployment Matlab models in the TWINSCAN software archive. The deployment simply means translating the generated code into a software component where the simulator is stored. The software developer does not need to deploy the generated Matlab archive manually. This process has been already atomized for archives produced by MATES.

Not all Matlab models can be deployed in TWINSCAN due to type conversion from Matlab to DDF, which is strongly typed. On the other hand, Python and Matlab are both weakly typed. This allows Pymatbridge to use serialization of Matlab objects in json format (simply, string) that is restored in Python. This approach allows the simulator in Python to call a Matlab equation directly without integrating into the TWINSCAN software archive.

### 7.5 Simulator Modeling Module

This section describes the simulator architecture representing decomposition of the hardware simulation that is a part of the simulator concept described in Section 6.1.
The simulator decomposition is shown in Figure 7.2. The decomposition is based on a DCA architecture designed within ASML. This architecture separates data form control and algorithms. The simulator decomposition defines three components that correspond to the DCA architecture structure:

- **Agent** is the control part. It is responsible for starting durative services – interactions. Agent is a state full component that describes what to do and in which order. The order of state changes can be modelled, for example, in a state machine. Such model describes the Agent behavior.

- **Interaction** is the algorithmic part. Interaction describes the dependences of the physical objects. Interaction may need data from physical objects in order to perform a calculation and then it sets objects with the output value. The calculation is called equation and it can be designed as the interaction method or a Matlab model.

- **Physical object** is the data part. The physical object represents the value of a modeled quantity in the simulated environment. This object does not have state, it does not perform calculation, and it only keeps physical quantities.

Agent, Interaction, and PhysicalObject cannot have composition relationship; they can only refer to each other. For example, two interactions may need data from the same physical components; at the same time, a request for a quantity value can go through Adapter avoiding interaction. That is why the ObjectManager singleton is introduced that owns the introduced objects and provides them by request.

Connection of the simulator objects with binding objects is done through the PartImpl and Adapter objects. For the simulator, one Part object is generated per a class specified by the user in the configuration file (see Section 7.3 ). By DriFE convention, for implementing functionality a subclass of Part must be created.
ates the PartImpl object that can pass its calls to multiple adapters. Like Part, the PartImpl instance is the only one per the user class; thus, it owns the ObjectManager singleton.

Adapter objects exist to adjust production interface data types to simulation data types. After data adaptation it can call Agent’s operations. Adapter also translates data before sending response. For example, a request asks for a physical quantity; in this case Adapter asks directly Physical Object for its value, translates its type to appropriate ASML type, and then sends the response.

After designing the simulator architecture, analysis on how the simulator structure can be modelled was done and the Domain Specific Modeling (DSM) approach was chosen. DSM raises the level of abstraction and hides programming languages in the same way as today's programming languages hide assembler. Symbols in a DSM map to things in the domain - the real world. DSM represents three parts: domain metamodel, environment for creating models, and code generation.

The domain metamodel was created based on the designed simulator architecture using Eclipse Modeling Framework. Two graphical editors for simulation structure and behavior modeling were designed based on the domain model using Sirius. Sirius is an Eclipse project that allows creating graphical modeling workbenches by leveraging the Eclipse Modeling technologies. The graphical editors produce simulation model (.simmodel) out of which code can be generated. The code generation was implemented with Acceleo also used in the Interceptor module (see Section 7.2 ).

As the result, using the fully implemented ASTER, a simulator was designed and deployed in the TWINSCAN software successfully.
8. Conclusions

Chapters 5-7 describe the process of the ASTER design in detail. This chapter concludes the main achievements of the current project in Section 8.1 Results.

8.1 Results

During the current project, the generic approach for designing simulators below the Driver layer was defined. It consists of four parts: the interceptor generation, binding generation, simulator modeling that includes equations modeled in Matlab. For each part a corresponding module was designed that together form the Asml Simulator development Environment called ASTER.

ASTER allows simplifying the simulator design, modelling, and integration process by enforcing the unified simulator architecture and reducing the engineering effort via code generation for every module. This satisfies the efficient simulator development process high-level goal of the project.

During a research phase, a number of in-house tools were analyzed whether they can be efficiently used for the simulator design. Two stand-alone in-house tools were applied in ASTER: the DrIFE code generation facility is used to generate the simulator binding and the MATES tool is used to build communication between software applications and Matlab models directly deployed in the simulator structure model.

The simulator modeling module was designed with intention of usage by both functional and software developers in order to satisfy the ASTER usability requirements. The modeling module raises the abstraction of high-level languages by introducing the simulator domain specific modeling engine. The simulator structure domain introduces the following simulator elements: Adapter, Agent, Interaction that can have the Matlab Equation, and Physical Object. Adapter exists to adjust production interface data types to simulation data types. The separation of the simulator domain into Agent, Interaction, and Physical Object corresponds to control, algorithm, and data parts of the DCA architecture defined in ASML. Such the simulator decomposition may lead to integration of ASTER into ASML in-house tools in the future, which brings such benefits as reusing ASML modeling tools.

The ASTER design was verified by creating a simulator for the POB group. The simulator was modelled in Eclipse IDE with ASTER installed and then the generated code was deployed in the TWINSCAN software archive. After deploying the simulator in the TWINSCAN archive, no production software required changes. This satisfies the following high-level goal: minimize production software risks.

The time required for a simulator modeling depends on the developer’s skills; therefore, for proving that ASTER achieved the engineering effort minimization goal, the binding generation was analyzed. Specifying input for the binding module, generating code, and migrating it to the TWINSCAN software archive require three hours, approximately; while creating binding from scratch manually takes around two days.
<table>
<thead>
<tr>
<th>Acceleo</th>
<th>An external tool is used for code generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACF</td>
<td>ASML class format describing the TWINSCAN software classes and implementing interfaces</td>
</tr>
<tr>
<td>ASD</td>
<td>ASML class format that describes classes to generate in ASML internal format</td>
</tr>
<tr>
<td>ASML</td>
<td>The company name. 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</td>
</tr>
<tr>
<td>CAFCR</td>
<td>The framework uses five views (Customer objectives, Application, Functional, Conceptual, Realization) helping to understand the inside of a system in relation to its context</td>
</tr>
<tr>
<td>CPD</td>
<td>Application used for calibration, performance measurement, or diagnostics purposes</td>
</tr>
<tr>
<td>Dev Bench</td>
<td>Testing platform used with software-in-the-loop simulation within the company</td>
</tr>
<tr>
<td>DC</td>
<td>The architecture designed within ASML that separates data form control and algorithms</td>
</tr>
<tr>
<td>DDF</td>
<td>Data Definition Format that provides a definition of data structures in a language-independent way</td>
</tr>
<tr>
<td>DriFE</td>
<td>Subsystem Driver Framework Environment is an in-house tool designed for increasing the effectiveness, efficiency and time of software driver development</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>DSM</td>
<td>Domain-Specific Modeling</td>
</tr>
<tr>
<td>Eclipse IDE</td>
<td>Eclipse integrated development environment is a software application that provides comprehensive facilities to programmers for software development</td>
</tr>
<tr>
<td>EMF</td>
<td>An external tool used for modeling simulators</td>
</tr>
<tr>
<td>HIL</td>
<td>Hardware-in-the-Loop</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated circuit is a set of electronic circuits on a small plate (chip) of semiconductor material (silicon)</td>
</tr>
<tr>
<td>Interceptor</td>
<td>Interceptor is a pattern that allows changing the message flow of the software</td>
</tr>
<tr>
<td>MATES</td>
<td>MATlab Exporter to SDEV (software development) is an in-house tool that improve communication between software applications and Matlab models</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-Driven Design</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Facility</td>
</tr>
<tr>
<td>MoSCoW</td>
<td>The framework for prioritizing requirements by four groups: Must, Should, Could, Would</td>
</tr>
<tr>
<td>OOTI</td>
<td>Ontwerpersopleiding Technische Informatica (see ST)</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>PDEng</td>
<td>Professional Doctorate in Engineering is a Dutch degree awarded to graduates of a Technological Designer (engineering) program</td>
</tr>
<tr>
<td>POB</td>
<td>Projection Optic Box</td>
</tr>
<tr>
<td>Proto</td>
<td>Complete machine used for testing</td>
</tr>
<tr>
<td>Rig Bench</td>
<td>Testing platform used with hardware-in-the-loop simulation within the company</td>
</tr>
<tr>
<td>SIL</td>
<td>Software-in-the-Loop</td>
</tr>
<tr>
<td>Simulation Competence</td>
<td>The special groups organized out of experts in the simulation domain in ASML. The purpose of the group is to improve working processes and share knowledge</td>
</tr>
<tr>
<td>Sirius</td>
<td>An external tool used for modeling simulators</td>
</tr>
<tr>
<td>ST</td>
<td>Software Technology is designers’ program resulting in PDEng degree</td>
</tr>
<tr>
<td>SUT</td>
<td>System under test</td>
</tr>
<tr>
<td>Test Bench</td>
<td>Testing platform used with hardware-in-the-loop simulation within the company</td>
</tr>
<tr>
<td>TU/e</td>
<td>Eindhoven University of Technology</td>
</tr>
<tr>
<td>TWINSCAN</td>
<td>Lithography machine that engraves patterns on silicon wafers, used to create integrated circuits</td>
</tr>
<tr>
<td>XText</td>
<td>The technology is used for creating DSL for the simulator configuration</td>
</tr>
</tbody>
</table>
Bibliography

References


Additional Reading


About the Authors

Irina Nikeshkina received her Engineer diploma in Software for Computers and Automated Systems from the Tomsk State University of Control Systems and Radioelectronic, Russia, in 2012. Her diploma thesis entitled “Automated system for relational database designing”, was designed for data modeling basing on the IDEF1X method. She also received her Bachelor degree in management from Tomsk National Research Polytechnic University, Russia, 2013. The bachelor diploma thesis was “Methods of software implementation management”. After graduation, she worked as a software developer for a Russian company Geological Solution during 1.5 years where she designed software for mineral exploration companies. Then, she worked as an English technical writer for a year. This experience was necessary for improving her English up to the level required by the Software Technology program in TU/e.
# A Requirements

## A.1 ASTER Requirements

<table>
<thead>
<tr>
<th>High-Level Goals</th>
<th>Customer Objective (Customer How)</th>
<th>Application (Product What)</th>
<th>AID</th>
<th>Prior</th>
<th>Description</th>
<th>Functional (Product What – Requirement)</th>
<th>RID</th>
<th>Prior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimize engineering effort</td>
<td>Generate glue code for binding</td>
<td>A01</td>
<td>M</td>
<td>When defining a new ASML class in a component, binding files must be created that includes glue code. This code must be auto generated.</td>
<td>Generate binding (Original requirements R01 and R02 contains proprietary information)</td>
<td>R01</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>Analyze simulator modeling ways</td>
<td>A02</td>
<td>M</td>
<td>The simulator structure and behavior can be modelled using different approaches and tools. Comparative analysis of these tools must be done.</td>
<td>Analyze ways for modeling a simulator structure</td>
<td>R02</td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apply the selected tool for modeling a simulator</td>
<td>A03</td>
<td>M</td>
<td>Based on R03 - R04, the chosen way of modelling the simulation structure and behavior must be included and implemented in ASTER.</td>
<td>Implement possibility to model a simulator structure</td>
<td>R03</td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generate code for modelled simulator</td>
<td>A04</td>
<td>M</td>
<td>If the simulator developer models a simulator using ASTER, by request the model must be transformed into code in compliance with the ASML component infrastructure.</td>
<td>Generate C++ code for the modelled simulator</td>
<td>R04</td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analyze equation approaches</td>
<td>A05</td>
<td>M</td>
<td>Interaction between physical objects can be expressed as equations that can be modelled with different tools.</td>
<td>Model an equation in Matlab</td>
<td>R05</td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generate code for</td>
<td>A07</td>
<td>M</td>
<td>If the developer models an equation using ASTER, by Generate the C++ simulator and the Matlab equation using MATES</td>
<td>Generate the C++ simulator and the Matlab equation using MATES</td>
<td>R07</td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirement</td>
<td>Action</td>
<td>Context</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>--------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>equation</td>
<td>request the model must be transformed into code in compliance with the ASML component infrastructure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generate code for interceptor</td>
<td>A08 S</td>
<td>In the current project, the interceptor is the client-side interceptor. It catches requests from SUT and redirects them to the simulator (see Section 4.4.3.).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Based on input DDFs, generate an interceptor library</td>
<td>R25 S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Provide possibility to choose a subset of functions declared in the DDFs</td>
<td>R26 S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generate code for interceptor</td>
<td>A08 S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Provide orthogonal input organization</td>
<td>A09 M</td>
<td>The input data required for the simulator must be organized in categories that are as independent as possible. Information of different categories is specified in distinct files.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Define a configuration file for binding generation</td>
<td>R27 M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create an environment for modeling simulator structure</td>
<td>R28 M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create an environment for modeling simulator behavior</td>
<td>R29 M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Define directory for input, build, output files</td>
<td>R30 M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Support the minimal ASTER input consisting of a configurational file with appropriate interfaces. This input is enough for code generation</td>
<td>R33 S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connect the ASTER modules without user interference (connect means that requests can go from one to another modules directly):</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- connect generated interceptor and binding</td>
<td>R35 C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- connect structure and behavior of the simulator</td>
<td>R37 S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connect the simulator model with a Matlab equation (A05)</td>
<td>R38 S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimize ASTER for maintenance and usability</td>
<td>A11 M</td>
<td>After implementing the simulator in Eclipse, very important that the generated code integrated with the TWINSCAN software runs without errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After the deploying generated simulator code, update building scope files and build the project with ccmake command without errors.</td>
<td>R40 M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimize ASTER for maintenance and usability</td>
<td>A12 M</td>
<td>The ASTER architecture must allow extending ASTER with a new functionality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Organize the ASTER modules as a set of Eclipse plug-in projects.</td>
<td>R41 M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine all module projects in an Eclipse feature.</td>
<td>R42 C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All generated files must be marked as not changeable</td>
<td>R43 M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Provide a guide on how the developer can extend the generated simulator with a manually written functionality</td>
<td>R44 M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deploy the generated code in the TWINSCAN software archive</td>
<td>R45 M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

46
<table>
<thead>
<tr>
<th>Clear guidelines and layout</th>
<th>Performant simulator</th>
<th>Minimize production SW risks</th>
<th>Efficient simulator development process</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design User Manual</strong></td>
<td><strong>Provide optimal generating time</strong></td>
<td><strong>Make the SUT unaware of server substitution</strong></td>
<td><strong>Define generic Simulator architecture</strong></td>
</tr>
<tr>
<td>A14 M</td>
<td>A15 S</td>
<td>A18 M</td>
<td>A23 M</td>
</tr>
<tr>
<td>Apart the final report, one of the delivery is a user manual that describes in detail the simulator architecture, the ASTER architecture, all research and analysis, and implementation details.</td>
<td>The generation time is time required for generating the simulation code that can be deployed in TWINSCAN.</td>
<td>After substitution the production component with the simulator, the SUT should act the same way as before. For the SUT, simulator is the production component.</td>
<td>Based on simulator analysis and experts input, the generic architecture of simulator must be defined</td>
</tr>
<tr>
<td><strong>Describe:</strong> - ASTER and simulator architecture  - interceptor, binding, equation generation  - structure and behavior modelling tools analysis  - prototype implementation details and usage example</td>
<td>The generation should be no more than linear to the modelled simulator data</td>
<td>Zero line of code changes in the SUT code production code:</td>
<td>Create the simulator structure and behavior modules based on the generic simulator decomposition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The Matlab equation code generation should take not more than generation time required by MATES multiplied 1.5</td>
<td>Design a simulator using the generic simulator architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The migrating time should not be more than linear to the generated code package size.</td>
<td>Run the simulator with the system start (after deploying the simulator in the TWINSCAN software)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R46M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R47S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R48S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R49S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R50S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R51W</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R52M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R53M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R54M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R55M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R56M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R57M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R58C</td>
</tr>
</tbody>
</table>

**Additional Notes:**
- Remove all files generated by DriFE that are not related to simulation.
- Generate the interceptor redirecting SUT requests to the simulator that implements a copy of the production interfaces.
- If the simulator is deployed in a separate component (not in the /tst folder), exclude the component from the production code.
- If the simulator is deployed in the /tst folder, no action required.
- The state of the simulator should be the same as the substituted production component would have. The SUT requests depend on the simulator state. If the state is not as SUT expects, the test results may be wrong.
4TU.School for Technological Design, Stan Akkermans Institute offers two-year postgraduate technological designer programmes. This institute is a joint initiative of the four technological universities of the Netherlands: Delft University of Technology, Eindhoven University of Technology, University of Twente and Wageningen University. For more information please visit: www.4tu.nl/sai